ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ МІСЬКОГО ГОСПОДАРСТВА
ІМЕНІ О. М. БЕКЕТОВА

Пояснювальна записка
до кваліфікаційної роботи бакалавра

на тему:
Дослідження event-driven моделей проектування високонавантажених serverless архітектур

Виконав:
Студент групи КН-2022-1
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

ГРИЩЕНКО І.О.
(прізвище та ініціали)

Керівник СІЗОВА Н.Д.
(прізвище та ініціали)

Рецензент БРЕДІХІН В.М.
(прізвище та ініціали)

м. Харків - 2026 рік

Харківський національний університет міського господарства імені О. М. Бекетова

(повне найменування закладу вищої освіти)

Навчально-науковий Інститут енергетичної, інформаційної та транспортної інфраструктури

Кафедра комп'ютерних наук та інформаційних технологій

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 (F3) Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри КНтаІТ

МаринаНОВОЖИЛОВА

« 22 » червня 2026 року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Грищенко Ілля Олексійович

(прізвище, ім'я, по батькові)

1 Тема роботи Дослідження event-driven моделей проектування високонавантажених serverless архітектур

керівник роботи д.ф-м.н., проф. Сізова Н.Д.

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від «22» травня 2026 р. № 440-03

2 Термін подання студентом роботи 20.06.2026р.

3. Вихідні дані до роботи рекомендації до теми «Дослідження event-driven моделей проектування високонавантажених serverless архітектур

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

проаналізувати предметну область та прототипи системи; обґрунтувати вибір інструментального середовища та технічної платформи; функціональний аналіз системи розгортання коду; архітектура системи, програмна реалізація та; описати архітектуру, дослідити розробку практичного приклада.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Презентація – 15 аркушів

6.Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	Завдання прийняв
Розділ I	Наталія СІЗОВА	11.05.2026	10.05.2026
Розділ II	Наталія СІЗОВА	17.05.2026	15.05.2026
Розділ III	Наталія СІЗОВА	21.05.2026	30.05.2026
Розділ IV	Вікторія МАЛИШЕВА	27.05.2026	15.06.2026

7. Дата видачі завдання 5.5.2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Вибір теми дипломної роботи	03.05.2026	Викон.
2	Затвердження тем, наукових керівників, завдань та календарного плану підготовки дипломної роботи	05.05.2026	Викон.
3	Написання I розділу	10.05.2026	Викон.
4	Написання II розділу	15.05.2026	Викон.
5	Написання III розділу	20.05.2026	Викон.
6	Написання IV розділу	30.05.2026	Викон.
7	Подання дипломної роботи керівнику	05.06.2026	Викон.
8	Робота по усуненню зауважень керівника, уточнення і доповнення практичного матеріалу, оформлення додатків до роботи	10.06.2026	Викон.
9	Подання доопрацьованого варіанту роботи керівнику	15.06.2026	Викон.
10	Захист матеріалів дипломної роботи на засіданні кафедри	18.06.2026	Викон.
11	Офіційний захист матеріалів дипломної роботи на засіданні екзаменаційної комісії	23.06.2026	Викон.

Студент _____ Грищенко І.О.
(підпис) (прізвище та ініціали)

Керівник роботи _____ СІЗОВА Н.Д.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Пояснювальна записка кваліфікаційної роботи бакалавра групи КН-2022-1 спеціальності 122 Комп'ютерні науки складається з 4 розділів, містить 76 сторінок тексту, 14 рисунків, 13 таблиць, 28 джерел.

Мета дослідження - розробка та дослідження моделей проєктування високонавантажених Serverless архітектур на основі Event-Driven підходу для підвищення масштабованості, відмовостійкості, продуктивності та ефективності використання обчислювальних ресурсів при обробці великих потоків подій у розподілених інформаційних системах.

Об'єкт дослідження - процеси функціонування та взаємодії компонентів високонавантажених розподілених інформаційних систем, побудованих на основі Serverless архітектури та подієво-керованих механізмів обробки даних.

Предмет дослідження -методи, моделі та алгоритми проєктування Event-Driven Serverless архітектур, механізми обробки подій, маршрутизації повідомлень, балансування навантаження, забезпечення масштабованості та оптимізації взаємодії між функціональними компонентами системи.

Завдання дослідження

Проаналізувати сучасні підходи до побудови Event-Driven та Serverless архітектур.

Дослідити особливості функціонування брокерів повідомлень та платформ FaaS.

Розробити математичну модель взаємодії подій у високонавантажених середовищах.

Запропонувати модель зменшення зв'язності (coupling) між сервісами.

Розробити алгоритм оптимізації маршрутизації подій.

Провести моделювання роботи системи при різних рівнях навантаження.

Оцінити показники масштабованості, латентності та відмовостійкості.

Методи дослідження - системний аналіз; теорія складних систем; теорія графів; методи оптимізації; теорія подій та дискретних систем.

Наукова новизна роботи може полягати у:

Розробленні нової моделі проєктування Event-Driven Serverless архітектури для високонавантажених систем з динамічним масштабуванням функцій.

Удосконаленні механізму маршрутизації подій шляхом адаптивного вибору каналів передачі повідомлень залежно від поточного навантаження системи.

У розділі «Загальні положення» наведено опис предметного середовища, пов’язаного з темою кваліфікаційної роботи, розглянуто аналоги та визначено задачі дослідження.

У першому розділі розглянуті питання оптимізації подій у event-driven архітектурі для зменшення зв’язності

У другому розділі описані підходи до оптимізації подій у event-driven архітектурі для зменшення зв’язності.

У третьому розділі описано програмне та технічне забезпечення для безсерверних архітектур (serverless architecture)

У четвертому розділі подається тема про охорону праці.

Ключові слова: інформаційні моделі, безсерверна архітектура, події, високонавантажені системи

ANNOTATION

Explanatory note of the bachelor's qualification work of group KN-2022-1, specialty 122 Computer Science consists of 4 sections, contains 76 pages of text, 14 figures, 13 tables, 28 sources.

The purpose of the study is to develop and study models for designing high-load Serverless architectures based on the Event-Driven approach to increase scalability, fault tolerance, productivity and efficiency of computing resources when processing large streams of events in distributed information systems.

The object of the study is the processes of functioning and interaction of components of high-load distributed information systems built on the basis of Serverless architecture and event-driven data processing mechanisms.

The subject of the study is methods, models and algorithms for designing Event-Driven Serverless architectures, mechanisms for event processing, message routing, load balancing, ensuring scalability and optimizing interaction between functional components of the system.

Research objectives

1. Analyze modern approaches to building Event-Driven and Serverless architectures.

2. Investigate the features of the functioning of message brokers and FaaS platforms.

3. Develop a mathematical model of event interaction in high-load environments.

4. Propose a model for reducing coupling between services.

5. Develop an algorithm for optimizing event routing.

6. Model the system at different load levels.

7. Evaluate scalability, latency, and fault tolerance indicators.

Research methods - system analysis; theory of complex systems; graph theory; optimization methods; theory of events and discrete systems.

The scientific novelty of the work may consist in:

1. Development of a new model for designing Event-Driven Serverless architecture for high-load systems with dynamic scaling of functions.

2. Improving the event routing mechanism by adaptively selecting message transmission channels depending on the current system load.

The section "General Provisions" provides a description of the subject environment related to the topic of the qualification work, considers analogues and defines research tasks.

The first section considers the issues of optimizing events in event-driven architecture to reduce connectivity

The second section describes approaches to optimizing events in event-driven architecture to reduce connectivity.

The third section describes software and hardware for serverless architectures

The fourth section presents the topic of labor protection.

Keywords: information models, serverless architecture, events, high-load systems

ЗМІСТ

ВСТУП

Сучасні цифрові технології швидко розвиваються, що призводить до зростання вимог до архітектури додатків, здатних обробляти великі обсяги даних у реальному часі. Збільшення кількості інтернет-сервісів, мобільних додатків і платформ для аналізу великих даних створює потребу в архітектурах, які можуть витримувати високі навантаження, забезпечувати швидкий відгук, надійність і масштабованість. У цьому контексті подієво-орієнтована (event-driven) та безсерверна (serverless) архітектури набули значної популярності завдяки своїм можливостям вирішувати ці завдання. Дослідження таких підходів є актуальним для ІТ-галузі, оскільки вони забезпечують підвищену ефективність та адаптивність високонавантажених систем.

Event-driven архітектура дає змогу додаткам реагувати на події в режимі реального часу, що особливо важливо для систем з великим навантаженням. Завдяки асинхронному виконанню запитів вона дозволяє уникнути блокування процесів, підвищуючи загальну продуктивність і масштабованість. У поєднанні з serverless підходом, який знижує витрати та спрощує управління інфраструктурою, ці архітектури допомагають організаціям оптимізувати свої ІТ-ресурси, концентруючись на функціональному розвитку своїх продуктів.

Serverless архітектура дозволяє автоматично надавати обчислювальні ресурси "на вимогу", адаптуючи їх під актуальне навантаження, що робить її ідеальною для сучасних додатків з високими вимогами до еластичності та продуктивності. Комбінація event-driven та serverless підходів забезпечує такі переваги, як зниження витрат на підтримку інфраструктури, високу масштабованість і можливість гарантувати відмовостійкість на рівні окремих функцій. Цей підхід є оптимальним для додатків, що повинні динамічно масштабуватися, обробляти велику кількість подій і відповідати вимогам реального часу, наприклад, у фінансових системах, стримінгових

сервісах, IoT-платформах тощо.

Мета цього дослідження полягає в аналізі ефективності event-driven архітектури в serverless середовищі для високонавантажених систем. Завданням роботи є ґрунтовний аналіз переваг і недоліків цих підходів, а також розробка оптимальних патернів для побудови масштабованих та адаптивних систем.

Для досягнення поставлених цілей використовуватимуться комплексні методи дослідження. Спочатку буде розглянуто й проаналізовано проблему зв’язності в подієво-орієнтованій архітектурі [1], запропоновано підхід для мінімізації обсягу подій. Далі буде здійснено аналіз ключових термінів та патернів, щоб визначити сучасні підходи й методи в сфері event-driven та serverless архітектур. Це дозволить створити базу знань про найефективніші патерни, їхні переваги й недоліки, підходи до проектування та рекомендації щодо їх застосування. На завершення буде сформовано висновки щодо високонавантажених систем на основі event-driven підходів у serverless середовищі, що дозволить оцінити вплив різних факторів на продуктивність та стійкість системи. Цей підхід надасть обґрунтовану оцінку ефективності обраних архітектур для високонавантажених систем і формулює рекомендації для їх оптимізації.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ДОСЛІДЖЕННЯ

У 2009 році компанія Netflix зіштовхнулася з проблемою швидкого зростання обсягів даних і обробки запитів, що було спричинено різким збільшенням популярності її сервісу відеострімінгу. Існуюча на той час інфраструктура виявилася недостатньо ефективною, щоб забезпечити належний рівень продуктивності та масштабованості для задоволення запитів користувачів. Це спонукало компанію до пошуку нових архітектурних рішень: було вирішено здійснити перехід від приватних дата-центрів до публічної хмари та провести реорганізацію від монолітної структури до розподіленої архітектури, яка згодом стала відомою як модель мікросервісів. На той час концепція мікросервісів ще не була повністю сформована та не мала широкого застосування, що додавало ризику і складності процесу.

Netflix стала однією з перших великих компаній, яка успішно впровадила перехід від монолітної архітектури до хмарної мікросервісної моделі. Ця трансформація принесла значні результати, дозволяючи оптимізувати продуктивність і забезпечити постійну доступність послуг. У 2015 році Netflix отримала спеціальну нагороду JAX Jury Award, частково завдяки цій інноваційній архітектурі, що інтегрувала сучасні принципи DevOps, забезпечуючи постійний розвиток і впровадження нових функцій без зупинки роботи сервісу. Сьогодні Netflix використовує понад тисячу мікросервісів, які взаємодіють для підтримки окремих компонентів платформи, дозволяючи інженерам швидко і часто вносити зміни, іноді здійснюючи тисячі оновлень щодня.

Досвід Netflix став важливим кроком у розвитку підходів до побудови високонавантажених інформаційних систем і створив передумови для поширення моделі мікросервісів як інноваційного стандарту. Їхній перехід від моноліту до мікросервісної архітектури продемонстрував потенціал для досягнення гнучкості, стійкості та легкого масштабування, що стало предметом подальших досліджень та імплементацій у галузі побудови сучасних серверних

архітектур, здатних підтримувати інтенсивне навантаження і високі вимоги до безперервності сервісу.

1.1 Монолітна архітектура

Монолітна архітектура являє собою традиційний підхід до створення програмних систем, де всі бізнес-функції об’єднані в єдину, нерозривно пов’язану структуру. В межах монолітної операційної системи основну роль виконує ядро, яке відповідає за управління всіма функціями, створюючи тісно інтегроване середовище. Цей підхід забезпечує єдність і узгодженість коду, де всі компоненти працюють у спільній кодовій базі [2].

Монолітна архітектура асоціюється з концепцією єдиного масиву, подібного до скельної породи, з якої формується цілісна структура. Як і в архітектурі, де один масив може утворювати кілька взаємопов’язаних будівель на єдиній основі, монолітні програми реалізують різні функції на спільному фундаменті коду, що забезпечує однаковість усіх компонентів системи.

Протягом десятиліть цей підхід домінував у розробці програмного забезпечення, завдяки своїй простоті та ефективності на ранніх етапах створення систем. Однак сьогодні розмова про монолітну архітектуру неминуче супроводжується порівнянням з її сучасними альтернативами.

Монолітна архітектура - це традиційний підхід до побудови програмних систем, де всі компоненти системи інтегровані в єдину велику програмну одиницю, яка виконується як єдине ціле [2-3]. У межах такої архітектури усі функціональні модулі, такі як користувацький інтерфейс, логіка бізнес-процесів і доступ до бази даних, розробляються, компілюються та деплояться як одна незалежна одиниця (див. рис 1.1). Це означає, що будь-які зміни або оновлення у будь-якому компоненті потребують перебудови і повторного розгортання всієї системи.

Однією з основних переваг монолітної архітектури є її початкова простота, особливо на ранніх етапах розробки. Вона дозволяє швидко створити прототип

системи, де всі модулі тісно пов'язані між собою, що спрощує налагодження й тестування взаємодії між компонентами. Оскільки всі частини системи розміщуються в одному середовищі, це також значно полегшує управління ресурсами, що позитивно впливає на продуктивність на початкових етапах розвитку.

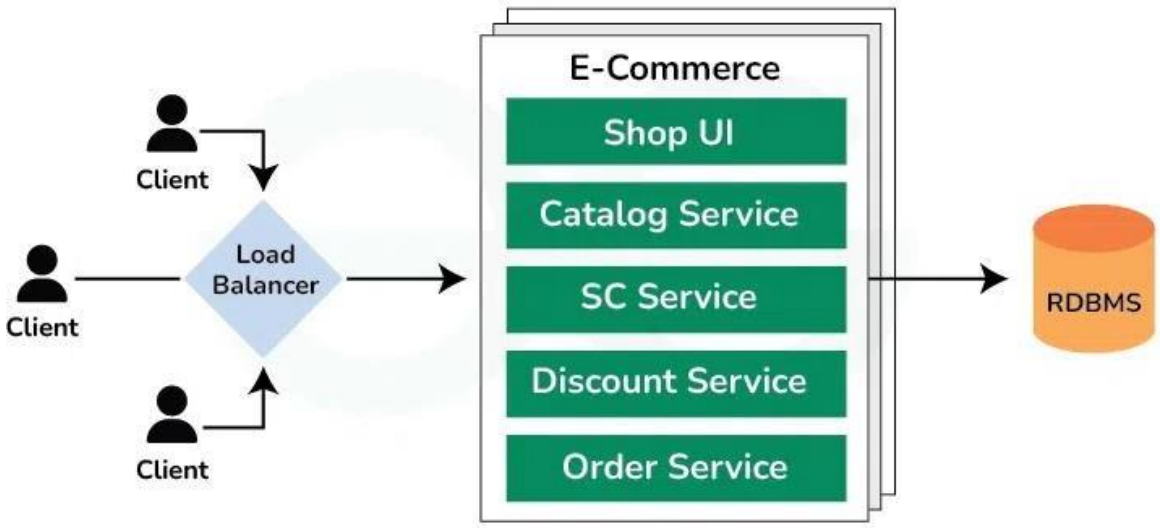


Рисунок 1.1 – Монолітна архітектура

Втім, з ростом кількості функцій і складності системи монолітна архітектура стикається з низкою проблем, які ускладнюють її масштабування та обслуговування. Наприклад, у випадках високонавантажених систем будь-яка зміна або додавання нового функціоналу може потребувати значних ресурсів для повторного тестування та оновлення всього застосунку. У великих системах це призводить до створення так званих «вузьких місць», що обмежують продуктивність і знижують надійність. Через тісну інтеграцію компонентів стає важко забезпечувати розподіл навантаження та відмовостійкість, адже вихід з ладу одного компонента може негативно вплинути на роботу всього застосунку.

Монолітні архітектури обмежують гнучкість команди розробників. У великих проектах часто виникає потреба в автономності команд для розвитку різних функцій продукту, проте в монолітній структурні зміни в одному модулі можуть мати непередбачувані наслідки для інших модулів. Це

ускладнює процеси розподілу задач між командами, затримує вихід оновлень і знижує загальну продуктивність розробки.

Монолітна архітектура, хоча і має свої переваги на ранніх стадіях проекту, стає менш оптимальною у випадках високонавантажених систем, де потрібні часті оновлення, швидке масштабування та забезпечення безперервної доступності [4]. Це спонукало компанії і дослідників шукати альтернативи, що призвело до розвитку інших підходів, зокрема мікросервісних та безсерверних (Serverless) архітектур, які краще відповідають потребам динамічних і масштабованих систем.

1.2 Мікросервісна архітектура

Монолітна архітектура довго залишалася основною моделлю побудови програмних систем, проте вже давно не є єдиним підходом. Починаючи з 1980-х років, у галузі розробки програмного забезпечення спостерігалося прагнення до модульності, завдяки чому стали популярними об’єктно-орієнтовані мови програмування. У 1990-х роках ці зміни створили сприятливе підґрунтя для розвитку розподілених систем, що могли використовувати новітні досягнення мережевих технологій.

Зрештою, ці тенденції призвели до появи мікросервісів, які набули широкого застосування завдяки поширенню хмарних обчислень і контейнеризації в 2000-х роках. Мікросервісна архітектура була створена як відповідь на потребу в швидкому масштабуванні й децентралізації, ставши значною еволюцією порівняно з традиційними монолітними підходами.

Мікросервісна архітектура - це підхід до створення програмного забезпечення, який набув широкої популярності з розвитком хмарних технологій і вимогами до масштабованості сучасних додатків. На відміну від монолітної архітектури, де вся система є одним кодовим блоком, мікросервісна архітектура передбачає поділ додатка на незалежні компоненти або "мікросервіси", кожен з яких виконує одну чітко визначену функцію (рис. 1.2) [5]. Завдяки цій

архітектурі додатки набувають більшої гнучкості, а їхнє розгортання, тестування і підтримка стають простішими та ефективнішими.

У основі мікросервісної архітектури лежать принципи автономності та незалежності кожного сервісу, що дозволяє розробникам легко виділяти окремі функціональні можливості та забезпечувати їхній розвиток окремими командами. Кожен мікросервіс має власну бізнес-логіку та працює автономно, що дозволяє командам розробників незалежно працювати над різними частинами системи. Таке розділення є критично важливим для великих проектів, де швидкість розвитку та адаптивність до змін мають вирішальне значення. Окрім того, незалежність сервісів дозволяє проводити їхнє розгортання та оновлення окремо, що значно підвищує гнучкість системи і дозволяє швидше реалізувати нові функції.

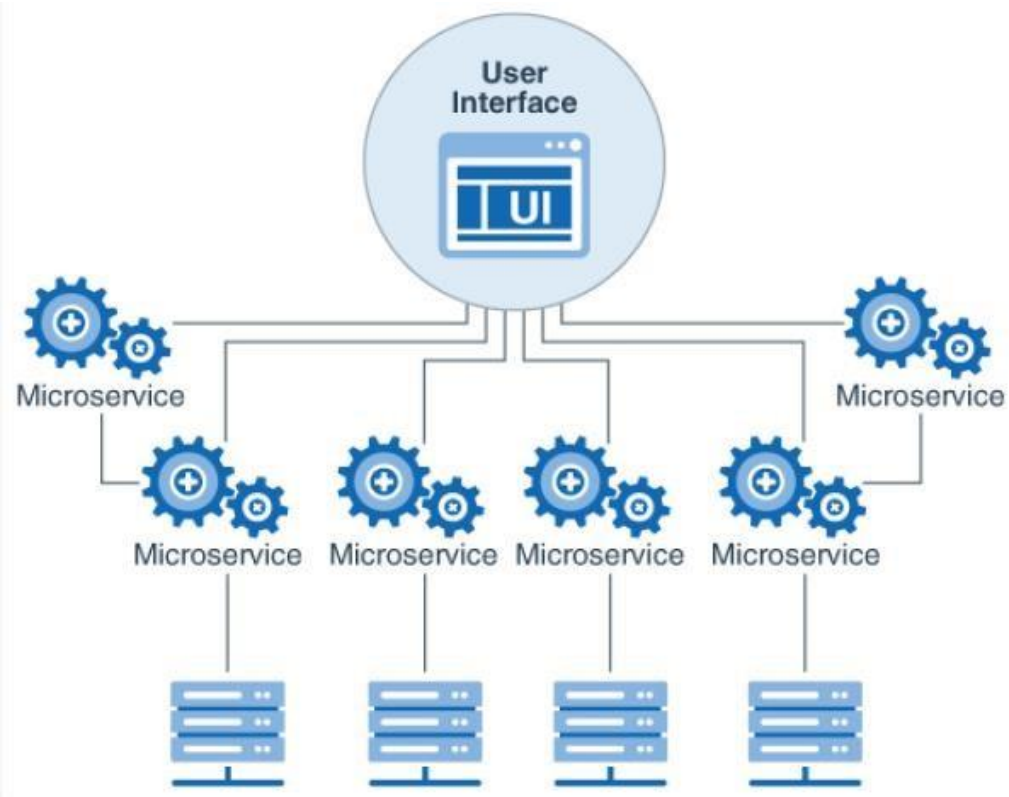


Рисунок 1.2 − Мікросервісна архітектура

Декомпозиція системи на незалежні мікросервіси також покращує масштабованість. Кожен мікросервіс можна масштабувати незалежно від інших,

що дозволяє адаптувати ресурси для конкретних компонентів додатка. Це суттєво знижує витрати на інфраструктуру, оскільки більше не потрібно масштабувати всю систему через збільшення навантаження на один компонент. Окрім того, мікросервіси можуть працювати на різних платформах і навіть використовувати різні мови програмування, що забезпечує максимальну гнучкість у виборі технологій і адаптації до вимог ринку.

Ще однією важливою особливістю мікросервісної архітектури є можливість комунікації між сервісами через стандартизовані інтерфейси, найчастіше RESTful API або протоколи обміну повідомленнями. Це забезпечує стабільність обміну даними і дозволяє ефективно інтегрувати різні частини системи. Комунікація через API стандартизує передачу даних, підвищуючи безпеку та спрощуючи підтримку.

Однією з найбільших переваг мікросервісної архітектури є гнучкість розвитку [6]. Команди можуть паралельно працювати над різними частинами додатка, що значно прискорює впровадження нових функцій та полегшує процес релізу. Оскільки кожен сервіс є незалежним, це дозволяє здійснювати розгортання та оновлення окремих частин без необхідності перезапускати всю систему, що знижує ризики збоїв і підвищує стабільність додатка.

Ще однією ключовою перевагою є масштабованість. Оскільки мікросервіси функціонують автономно, можливо масштабувати лише ті компоненти, які дійсно потребують цього, знижуючи таким чином витрати на інфраструктуру. Наприклад, у системах із високим навантаженням можна адаптувати ресурси лише для конкретного сервісу, що забезпечує оптимальне використання інфраструктури.

Мікросервісна архітектура покращує надійність системи. Якщо один із сервісів виходить з ладу, це не обов’язково означає, що зупиниться вся система. Інші мікросервіси можуть продовжувати функціонувати, що забезпечує стійкість додатка до часткових збоїв. Така стійкість є особливо цінною для великих систем, які мають обслуговувати мільйони користувачів одночасно.

Попри всі переваги, мікросервісна архітектура має свої виклики. Основною проблемою є складність комунікації між сервісами. Кожен мікросервіс є незалежним, і для забезпечення узгодженої роботи потрібно налаштувати надійний обмін даними. Це часто реалізується через HTTP-запити або системи чергування повідомлень, що додає складності та вимагає додаткових ресурсів на обробку комунікацій. Забезпечення ефективної комунікації може вимагати значних зусиль, а іноді потребує використання спеціальних систем для управління та моніторингу мережевої взаємодії між сервісами.

Водночас мікросервіси створюють підвищені вимоги до інфраструктури. Для управління великою кількістю автономних сервісів часто потрібні складні системи оркестрації, такі як Kubernetes, які забезпечують балансування навантаження, автоматичне масштабування, моніторинг і розгортання. Це може значно підвищити витрати на підтримку, а також вимагати додаткових знань і навичок у сфері управління ресурсами та інфраструктурою.

Збереження консистентності даних також є непростим завданням у розподілених системах, де різні сервіси можуть мати власні бази даних або доступати до загальної інформації. Забезпечення узгодженості між мікросервісами вимагає значних зусиль, особливо у випадках великих і високонавантажених систем, де синхронізація даних є критично важливою для коректного функціонування додатка.

Сучасні технологічні гіганти, такі як Amazon, Netflix, Google, Uber та інші, активно застосовують мікросервісну архітектуру для розв’язання проблем високих навантажень і забезпечення надійності своїх платформ. Як було сказано раніше, Netflix використовує мікросервіси для окремих функцій, таких як обробка потокового відео, зберігання даних користувачів та рекомендаційні системи. Така архітектура дозволяє компанії швидко реагувати на збільшення навантаження, забезпечуючи стабільну якість обслуговування.

На початку 2020-х років, коли розвиток програмного забезпечення активно набуває нових форм, існують два домінуючі підходи - монолітна та мікросервісна архітектури. Хоча сучасні тенденції все частіше схиляються до

мікросервісного підходу, не завжди можна однозначно стверджувати, що він є безумовно кращим у всіх аспектах. Монолітна архітектура залишається зручним рішенням для невеликих або відносно простих систем, де є потреба у простоті управління та мінімізації інфраструктурних вимог.

Мікросервісна архітектура, яка пропонує розподіл на незалежні компоненти, стала поширеним вибором для великих систем, що потребують високої гнучкості та масштабованості. Завдяки автономності кожного мікросервісу можна розробляти, розгортати та масштабувати окремі компоненти без ризику збоїв у всій системі. Це надає розробникам значну свободу та дозволяє створювати більш стійкі до навантажень додатки, які швидше реагують на зміни в умовах ринку.

Однак, попри всі переваги мікросервісного підходу, він також має певні проблеми, серед яких складність комунікації між сервісами та підтримання консистентності даних. У великих системах, де мікросервіси мають взаємодіяти один з одним, виникає потреба у стандартизації цієї взаємодії. У мікросервісній архітектурі комунікація може бути як синхронною, так і асинхронною, і вибір між ними впливає на роботу системи загалом.

Синхронна взаємодія передбачає, що один мікросервіс викликає інший і чекає на відповідь, поки той не завершить обробку запиту. Цей метод добре підходить для запитів, де необхідна негайна відповідь, проте в ньому є значні обмеження. Якщо один з мікросервісів стає недоступним або уповільнює свою роботу, це може призвести до збоїв у всій системі. Така ситуація обмежує стійкість системи і ускладнює її масштабування.

Асинхронна взаємодія надає більшу гнучкість, оскільки дозволяє сервісу відправити запит і не чекати на негайну відповідь. Інші мікросервіси можуть обробити цей запит, коли матимуть можливість, і відправити відповідь пізніше. Така модель комунікації, що часто реалізується через системи черг повідомлень (наприклад, RabbitMQ, Apache Kafka, Azure Service Bus), підвищує стабільність і стійкість до збоїв, особливо при високих навантаженнях. Асинхронний підхід стає основою для архітектур на основі подій (Event-Based Architecture), де мікросервіси взаємодіють через події та обробляють їх у реальному часі або в міру

необхідності.

1.3 Ключові поняття подієвої архітектура

Event-Based, або подієва архітектура, стала важливим кроком у еволюції програмних систем, особливо для тих, що потребують обробки великих обсягів даних і високого рівня надійності. Її застосування дозволяє підвищити продуктивність, адаптивність і стійкість систем, роблячи їх більш масштабованими і гнучкими. Особливо актуальним використання подієвої архітектури є у випадках інтеграції з мікросервісними системами та хмарними обчисленнями, оскільки вона забезпечує необхідний фундамент для роботи в цих складних середовищах [7-8]. У подієвій архітектурі зміни в системі визначаються подіями, які можуть ініціювати певні дії в інших компонентах системи, дозволяючи додаткам функціонувати як незалежний набір процесів, що асинхронно реагують на різні події.

Мікросервісна архітектура, що активно використовується для розподілу великих систем на незалежні сервіси, дуже вдало поєднується з подієвою архітектурою. Завдяки цій інтеграції, кожен мікросервіс може реагувати на події, отримувати інформацію від інших сервісів і в той же час залишатися незалежним. Це особливо важливо у великих системах, де постійно відбуваються численні взаємодії між компонентами, а необхідність підтримки масштабованості, гнучкості та відмовостійкості є критичною.

Подієва архітектура базується на кількох ключових поняттях, кожне з яких має важливу роль у забезпеченні продуктивної, надійної та масштабованої системи.

Подія є базовим елементом подієвої архітектури і може бути описана як зміна стану або виконання певної дії в системі, яка тригерить подальші реакції. Це можуть бути події, пов’язані з користувацькими діями (наприклад, натискання кнопки чи оформлення замовлення), оновленням стану бази даних або змінами в

зовнішніх компонентах. Кожна подія є сигналом, який запускає конкретні дії в системі, що дозволяє компонентам реагувати в реальному часі на зміни та працювати з мінімальними затримками. Події можуть ініціювати як однократні дії, так і довготривалі процеси, що дозволяє налаштовувати складні потоки даних у великих системах.

Коли подія відбувається, обробник подій (event handler) виконує відповідні дії, наприклад, оновлює дані в базі, надсилає повідомлення користувачу або ініціює новий процес. У більшості випадків обробники працюють асинхронно, тобто виконують свої функції незалежно від інших компонентів, що дозволяє ефективніше використовувати ресурси та забезпечувати обробку великої кількості подій одночасно. Завдяки цьому підходу продуктивність системи зростає, оскільки обробники подій не блокують інші частини системи.

У великих системах, де обсяг подій може бути дуже високим, використовуються черги повідомлень та шини подій для ефективного управління цим потоком даних. Черга повідомлень дозволяє тимчасово зберігати події до моменту їх обробки, що є зручним, коли обробник подій зайнятий або система знаходиться під великим навантаженням. Шина подій, у свою чергу, забезпечує розподіл подій між обробниками та організовує паралельну обробку, що знижує затримки та підвищує ефективність обробки (рис. 1.3).

Поєднання Event-Based архітектури з мікросервісною дозволяє системам отримати значну перевагу у гнучкості, масштабованості та надійності. Мікросервісна архітектура забезпечує розподіл системи на незалежні сервіси, кожен з яких виконує окрему функцію і може бути розгорнутий окремо. Проте, оскільки мікросервіси працюють як окремі компоненти, координація між ними може стати складним завданням. Саме тут Event-Based підхід дозволяє організувати комунікацію між мікросервісами, використовуючи події як основний засіб передачі даних. Кожен мікросервіс може ініціювати події, на які реагують інші сервіси, забезпечуючи постійний обмін інформацією та координацію дій.

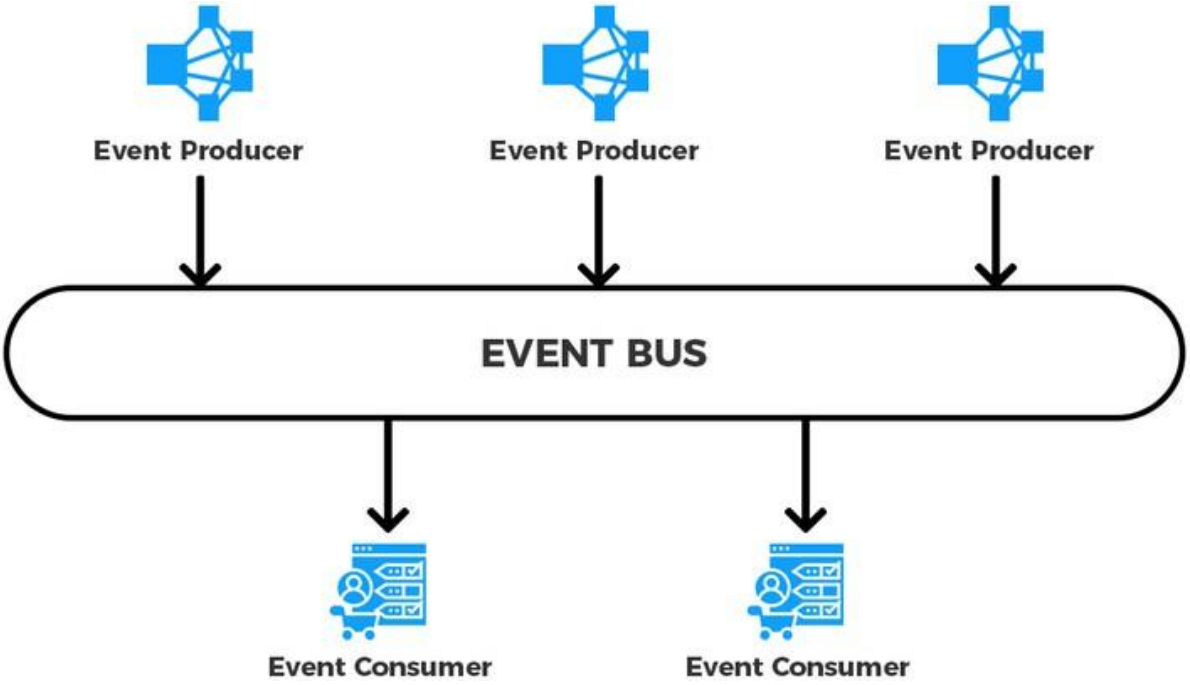


Рисунок 1.3 − Event-based архітектура

Однією з важливих характеристик цього поєднання є слабке зв'язування між компонентами, що дозволяє мікросервісам незалежно взаємодіяти один з одним без прямого виклику функцій. У разі виникнення збоїв в одному з мікросервісів, інші компоненти системи продовжують працювати, що підвищує надійність і дозволяє уникати повних відмов системи.

Подієва архітектура дозволяє обробляти події паралельно, забезпечуючи гнучке масштабування ресурсів для підтримки необхідного рівня продуктивності. Системи, які працюють з великими обсягами подій, можуть динамічно адаптувати свої ресурси до поточного навантаження, знижуючи витрати на інфраструктуру.

Оскільки обробники подій можуть залишатися автономними та зберігати події у чергах до їхньої обробки, система не зупиняється у разі збою одного з обробників. Це забезпечує стабільність роботи системи і мінімізує ризик втрати даних.

Завдяки асинхронній природі подієвої архітектури сервіси можуть функціонувати незалежно один від одного, реагуючи тільки на події. Це означає, що нові сервіси можна додавати або заміняти без зміни основної логіки системи. Подієва архітектура ідеально підходить для додатків, які потребують миттєвої реакції на події, таких як фінансові системи, онлайн-маркети чи системи моніторингу. Події обробляються одразу після їх надходження, що знижує затримки і підвищує ефективність обробки.

Незважаючи на численні переваги, подієва архітектура також має певні обмеження і труднощі у впровадженні.

Складність управління потоками подій. У великих системах обсяг подій може бути дуже високим, що створює труднощі з управлінням та моніторингом. У таких випадках необхідні системи моніторингу та управління потоками подій, які допомагають відстежувати стан системи та виявляти проблеми.

Проблеми з узгодженістю даних. В асинхронних системах складно забезпечити узгодженість даних між різними компонентами, особливо у випадках, коли компоненти працюють з розподіленими базами даних. Це призводить до необхідності введення додаткових механізмів для забезпечення консистентності.

Підвищені вимоги до інфраструктури. Подієва архітектура вимагає використання потужних систем управління подіями, зберігання даних та чергування повідомлень, що може створювати значне навантаження на інфраструктуру. Використання таких інструментів, як Apache Kafka або RabbitMQ, допомагає оптимізувати потік подій, але потребує додаткових ресурсів для обслуговування.

Складність налаштування обробки асинхронних подій. На відміну від синхронного оброблення запитів, асинхронна обробка подій вимагає більш детального налаштування обробки виключень, забезпечення порядку виконання та контролю послідовності подій.

Event-Based архітектура ідеально підходить для хмарних і serverless платформ, де події можуть створюватися та оброблятися різними сервісами в

асинхронному режимі. У хмарних обчисленнях події можуть генеруватися численними сервісами, і Event-Based архітектура дозволяє системам реагувати на них у режимі реального часу, забезпечуючи ефективну обробку даних. Serverless архітектура, яка дозволяє функціям автоматично запускатися у відповідь на події, особливо виграє від використання подієвої архітектури. Завдяки цьому досягається значна економія ресурсів, оскільки функції активуються лише тоді, коли це необхідно.

Поєднання Event-Based підходу з мікросервісною архітектурою відкриває можливості для побудови великих, високонавантажених і розподілених систем, здатних обробляти масиви даних у реальному часі, швидко масштабуватися та легко адаптуватися до вимог сучасного ринку. Це дозволяє оптимально використовувати ресурси і забезпечує високу продуктивність системи, що є ключовим для сучасних застосунків, які обслуговують тисячі або навіть мільйони користувачів.

1.4 Системи Serverless

До появи хмарних систем і особливо serverless архітектури, розробка та підтримка додатків вимагали значних зусиль та ресурсів, зокрема для управління інфраструктурою. Розробникам доводилося вирішувати широкий спектр завдань, починаючи від налаштування серверів та управління операційними системами до масштабування ресурсів і забезпечення стійкості до навантажень. Інфраструктурні вимоги включали необхідність попереднього планування потужностей: для великих систем це означало придбання серверного обладнання, його розгортання та налаштування, а також постійне обслуговування. У випадку з меншими компаніями такі завдання часто призводили до додаткових витрат на оренду або підтримку фізичних серверів, що ускладнювало процес масштабування.

Традиційні підходи вимагали точного прогнозування потреб системи на обчислювальні ресурси, що інколи призводило до надмірних витрат у вигляді

резервного обладнання, яке використовувалося лише періодично. У випадку високонавантажених періодів система могла стикатися з проблемами, коли наявні ресурси були недостатні для підтримки роботи додатка, що призводить до перебоїв або зниження якості обслуговування. Водночас, у періоди низького навантаження значні обчислювальні ресурси залишалися невикористаними, що створювало додаткові витрати.

Ще однією проблемою, з якою стикалися розробники, було масштабування. У традиційних архітектурах масштабування потребувало додаткового налаштування інфраструктури, що включало як закупівлю нових серверів, так і витрати на їх обслуговування. Це ставило перед бізнесом серйозні виклики, оскільки потреба у швидкій реакції на зміни в навантаженні додатків вимагала не лише додаткових інвестицій, але й постійної оптимізації.

З появою хмарних технологій у 2000-х роках індустрія розробки програмного забезпечення отримала потужний інструмент, який дозволяє ефективно вирішувати проблеми, пов'язані з управлінням інфраструктурою. Хмарні обчислення надали можливість зберігати дані та виконувати обчислення на віддалених серверах, що значно знизило потребу в придбанні та обслуговуванні фізичного обладнання для компаній. Це забезпечило гнучкість, швидке масштабування і зменшення операційних витрат, відкривши нові можливості для бізнесу будь-якого масштабу - від стартапів до великих корпорацій.

Основною перевагою хмарних технологій стало усунення необхідності утримання власного серверного парку. Компанії отримали можливість орендувати інфраструктуру в провайдерів хмарних послуг, оплачуючи лише використані ресурси. Це вирішило одразу кілька основних проблем:

– зменшення витрат на інфраструктуру: компанії більше не потребували великих капіталовкладень для придбання серверів і їхнього обслуговування;

– гнучке масштабування: можливість швидко збільшувати або зменшувати ресурси у відповідь на зміну навантаження;

– автоматизація: автоматичне резервне копіювання, балансування навантаження та управління безпекою стали доступнішими, знижуючи ризик простоїв і втрат даних.

Хмарні провайдери, такі як Amazon Web Services (AWS), Microsoft Azure і Google Cloud Platform, почали пропонувати різні види хмарних послуг, що задовольняли потреби компаній у різних рівнях управління інфраструктурою. Найпоширенішими моделями хмарних послуг стали IaaS (інфраструктура як послуга), PaaS (платформа як послуга) та SaaS (програмне забезпечення як послуга) (див. рис. 1.4) [8]. Кожна з цих моделей вирішувала конкретні проблеми та була орієнтована на певні вимоги користувачів.

IaaS (Infrastructure as a Service) - інфраструктура як послуга, надає компаніям можливість орендувати основні елементи інфраструктури - сервери, сховища, мережеві ресурси та інші обчислювальні компоненти. Замість того, щоб купувати та налаштовувати фізичні сервери, компанії можуть використовувати віртуальні машини або контейнери, що надаються провайдером. Цей підхід забезпечує повний контроль над операційними системами, мережевою конфігурацією та збереженням даних.

IaaS дозволяє організаціям зосередитися на основних аспектах свого бізнесу, не витрачаючи ресурси на управління інфраструктурою. Водночас ця модель надає можливість гнучко масштабувати ресурси у відповідь на зміну потреб додатка або навантаження на систему. Провайдер IaaS, наприклад AWS з сервісом EC2 (Elastic Compute Cloud), забезпечує підтримку інфраструктури, але управління операційними системами, додатками та базами даних залишається відповідальністю компанії-замовника.

PaaS (Platform as a Service) - платформа як послуга, надає повністю готову платформу для розробки, тестування та розгортання додатків, дозволяючи розробникам зосередитися на створенні функціоналу без необхідності налаштовувати та керувати інфраструктурою.

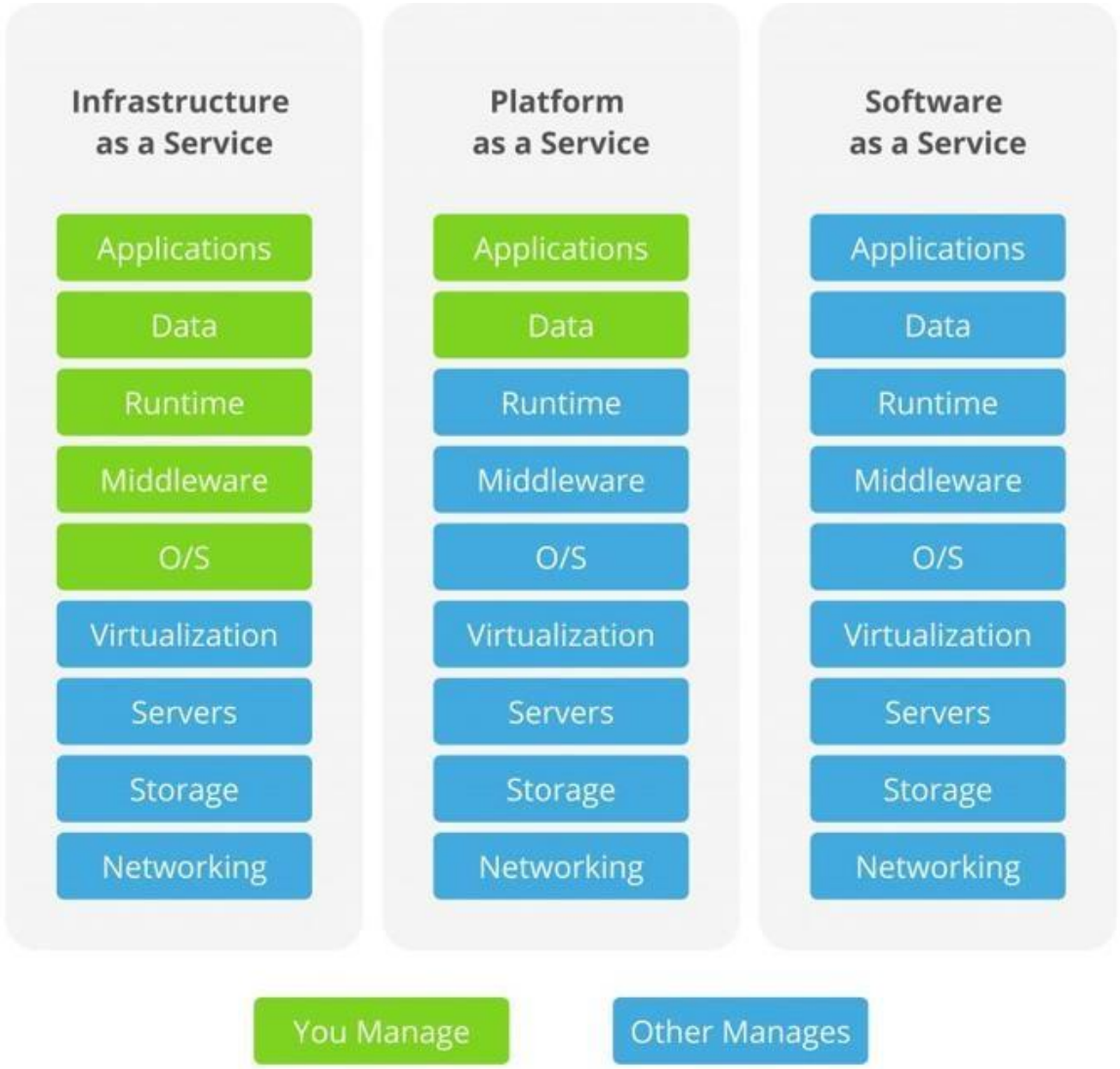


Рисунок 1.4 − Порівняння хмарних моделей

У PaaS-середовищах користувачі отримують доступ до серверів, баз даних, систем зберігання, засобів розробки, середовищ виконання і, інколи, інструментів управління версіями. Це особливо корисно для розробників, оскільки PaaS забезпечує автоматичне масштабування та управління навантаженням, знижуючи потребу в технічній підтримці на рівні інфраструктури.

PaaS-провайдери, такі як Google App Engine або Microsoft Azure App Service, беруть на себе управління серверами, забезпечуючи середовище для розробки додатків та обробки запитів користувачів. Це значно полегшує процес

розгортання та обслуговування додатків, особливо для команд, які хочуть швидко створювати нові функції, не зосереджуючись на інфраструктурних питаннях.

SaaS (Software as a Service) - програмне забезпечення як послуга, надає кінцевим користувачам доступ до готових програмних рішень через Інтернет, що дозволяє уникнути встановлення та обслуговування програмного забезпечення на власних пристроях. Всі інфраструктурні завдання, включаючи налаштування, оновлення, обробку даних та управління безпекою, виконує провайдер. Користувачі лише взаємодіють з інтерфейсом додатка через веббраузер або мобільний додаток, отримуючи доступ до повноцінної програми без необхідності управління інфраструктурою.

Типовими прикладами SaaS-сервісів є Google Workspace, Microsoft 365, Salesforce, що надають повний спектр можливостей для бізнесу, від управління документами до CRM-систем. Модель SaaS є особливо популярною серед кінцевих користувачів, оскільки дозволяє швидко отримати доступ до програмного забезпечення і платити лише за реальне використання без додаткових витрат на обладнання або технічну підтримку.

Хмарні моделі IaaS, PaaS та SaaS забезпечили вирішення багатьох проблем, з якими компанії стикалися раніше. Замість того, щоб керувати складною інфраструктурою, компанії тепер можуть зосередитися на розробці, обслуговуванні та масштабуванні своїх додатків. Хмарні сервіси дозволили зменшити час та витрати на створення додатків, а також підвищили надійність та доступність сервісів для кінцевих користувачів.

Хмарні рішення також дозволяють використовувати ресурси більш ефективно, пропонуючи моделі «оплати за використання», де компанії платять лише за фактично використані ресурси. Це особливо важливо для компаній, які мають значні коливання у навантаженні на систему: в періоди високих запитів вони можуть збільшувати ресурси, а в періоди низького навантаження - зменшувати, оптимізуючи витрати.

З розвитком хмарних обчислень з'явилися передумови для створення ще більш гнучкої та адаптивної архітектури - serverless, яка дозволяє розробникам

взагалі відмовитися від управління інфраструктурою та орієнтуватися лише на функціональні завдання. У serverless моделі функції виконуються за вимогою, лише у відповідь на події, що надходять. Це забезпечує ще більшу гнучкість і ефективність, дозволяючи платити лише за фактичне виконання функцій без додаткових витрат на підтримку серверів.

Serverless архітектура стала наступним кроком в еволюції хмарних технологій, дозволяючи розробникам повністю зосередитися на створенні функціональності додатків, не турбуючись про управління інфраструктурою [9-10]. У serverless підході розробники не потребують власних серверів чи віртуальних машин для розміщення коду - інфраструктура автоматично керується хмарним провайдером, і оплата здійснюється лише за фактичне використання обчислювальних ресурсів.

Основною ідеєю serverless є можливість запуску додатків або функцій лише у відповідь на конкретні події, без необхідності утримання постійно працюючих серверів. Розробники можуть створювати функції, які виконуються тільки при настанні певних подій, таких як оновлення даних, зміна статусу замовлення або надходження нових повідомлень у черзі. Це дозволяє компаніям знижувати витрати, оскільки ресурси активуються лише на час виконання завдань. Serverless стає ідеальним рішенням для додатків, де важлива масштабованість, надійність і економічність, зокрема для подійного (event-driven) підходу, який часто використовується у сучасних високонавантажених системах.

FaaS (Functions as a Service) - функції як послуга, є основним компонентом serverless архітектури, де програми складаються з незалежних функцій, які виконуються у відповідь на певні події, такі як HTTP-запити, повідомлення від інших сервісів або зміни в базі даних. У моделі FaaS кожна функція є окремим модулем коду, який виконує лише одне завдання або набір пов'язаних завдань.

Важливою особливістю FaaS є те, що функції автоматично масштабуються у відповідь на зміни в навантаженні, і розробникам не потрібно вручну налаштовувати або керувати цим процесом (рис. 1.5).

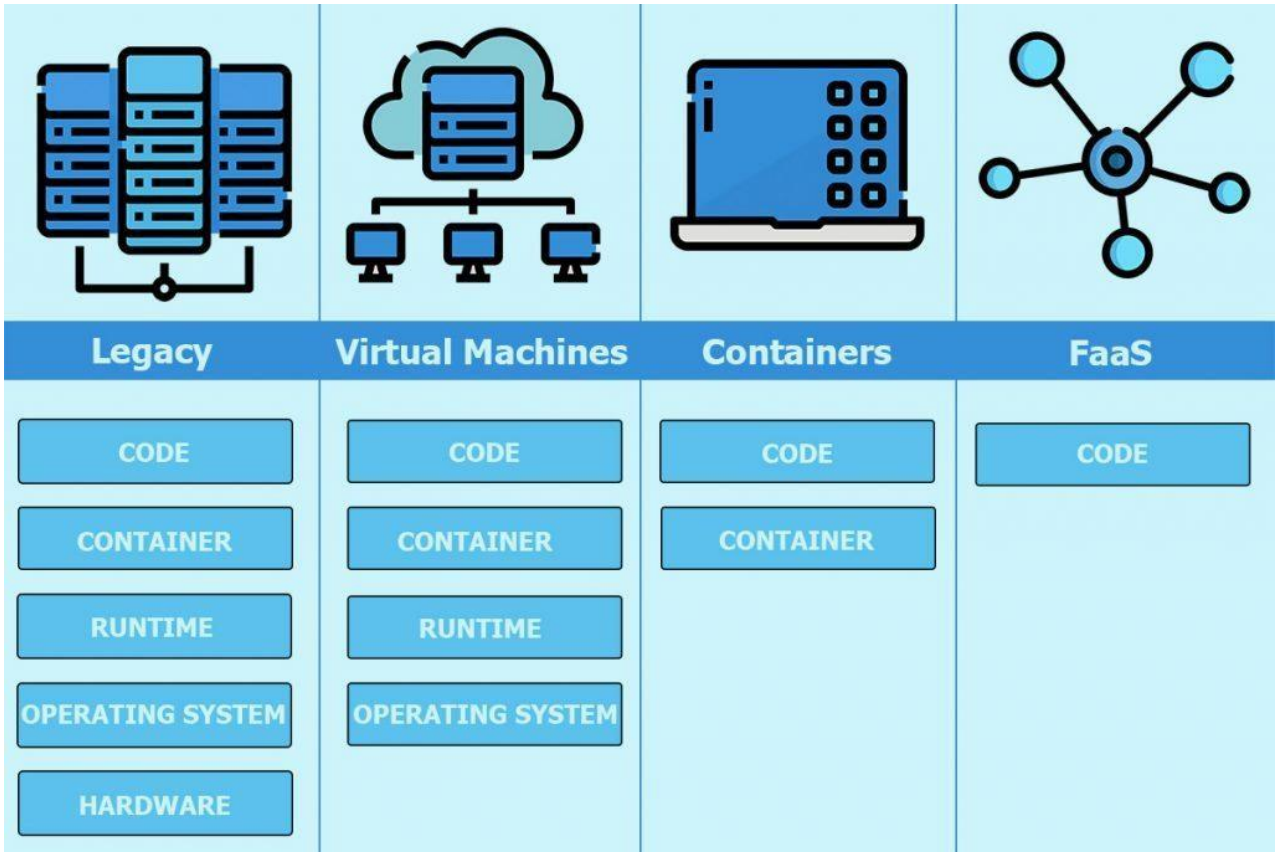


Рисунок 1.5 − Види систем для розгортання коду

Прикладом FaaS є AWS Lambda від Amazon Web Services, Azure Functions від Microsoft та Google Cloud Functions. Усі ці сервіси дозволяють розробникам запускати функції в середовищі, яке автоматично масштабується та працює лише під час виконання функції, що суттєво знижує витрати та забезпечує високу продуктивність.

Serverless архітектура має кілька ключових принципів, які забезпечують її ефективність та популярність у сучасній розробці програмного забезпечення:

– запуск за подією. У serverless архітектурі функції виконуються у відповідь на події, що надходять від користувачів або інших сервісів, таких як додавання нового запису до бази даних, надсилання запиту через API або

отримання повідомлення. Це дозволяє обробляти запити лише тоді, коли це необхідно, і не утримувати постійно активні сервери, як у традиційній моделі;

– автоматичне масштабування. У serverless підході хмарний провайдер автоматично масштабує ресурси, забезпечуючи швидку реакцію на зміну навантаження. Наприклад, якщо функція викликається тисячі разів одночасно, хмарний сервіс автоматично забезпечить додаткові обчислювальні ресурси для обробки всіх запитів паралельно, зменшуючи затримки;

– оплата за фактичне використання. Serverless моделі дозволяють уникнути значних інфраструктурних витрат, оскільки оплата здійснюється тільки за час фактичного виконання функцій. Наприклад, якщо функція працює лише кілька секунд на день, користувач платить тільки за ці секунди роботи, не витрачаючи кошти на підтримку постійно активного сервера;

– без управління інфраструктурою. В serverless середовищі розробникам не потрібно піклуватися про управління операційними системами, безпекою, оновленнями чи моніторингом серверів. Усі ці завдання бере на себе провайдер, що дозволяє розробникам зосередитися виключно на логіці додатка.

Serverless архітектура пропонує численні переваги, які роблять її ефективним вибором для сучасних додатків:

– економічність. Завдяки моделі оплати за використання serverless підхід значно знижує витрати на інфраструктуру, особливо для додатків з нерівномірним навантаженням, де постійна підтримка серверів може бути занадто витратною;

– швидкість розробки. Оскільки немає потреби управляти інфраструктурою, розробники можуть швидше створювати та розгортати функції, зосереджуючись на логіці програми. Це прискорює час виходу продукту на ринок та знижує загальну складність проекту;

– масштабованість. Serverless моделі забезпечують автоматичне масштабування під навантаження, що дозволяє обробляти великі обсяги запитів без затримок та втрат продуктивності. Це особливо важливо для додатків, які обслуговують великий трафік або піддаються різким піковим навантаженням;

– підтримка подійної архітектури. Serverless архітектура ідеально поєднується з подієвим підходом, оскільки функції автоматично запускаються у відповідь на події, що підвищує гнучкість і дозволяє будувати високонавантажені системи з чіткою структурою та легким обміном інформацією між компонентами.

Незважаючи на численні переваги, serverless архітектура має певні обмеження та виклики, які слід враховувати при її впровадженні:

– затримка при "холодному старті". Коли функція запускається вперше після тривалої бездіяльності, її ініціалізація може зайняти більше часу через "холодний старт". Це може вплинути на швидкодію додатка, особливо для додатків, які потребують миттєвої реакції;

– обмеження тривалості виконання. Деякі провайдери serverless сервісів мають обмеження на тривалість виконання функцій, що може ускладнювати використання serverless для додатків, що потребують тривалого виконання завдань;

– складність у відстеженні стану. Оскільки serverless функції є короткотривалими та без стану, це може створювати труднощі при реалізації складних робочих процесів, що потребують збереження стану. Зазвичай, для цього потрібно використовувати додаткові сервіси збереження стану.

Serverless архітектура стала можливою завдяки хмарним обчисленням, які забезпечують гнучку та надійну інфраструктуру для виконання обчислень за вимогою. У поєднанні з FaaS, serverless дозволяє створювати легкі та гнучкі системи, які масштабуються в залежності від потреб додатку.

FaaS-платформи, такі як AWS Lambda, Google Cloud Functions, Azure Functions, надають всі необхідні інструменти для розробки подійно-орієнтованих додатків без управління інфраструктурою.

Поява FaaS розширила можливості serverless архітектури, дозволивши реалізовувати складні завдання, розділяючи їх на мікрофункції, що запускаються за потреби.

У таких системах кожна функція виконує свою окрему частину завдання і є самостійною, що дозволяє легко масштабувати окремі частини додатка та економити ресурси. Це особливо корисно у випадках, де події ініціюють окремі обчислювальні процеси, такі як обробка зображень, обробка даних або відправка повідомлень.

Serverless архітектура у поєднанні з подієво-орієнтованим підходом відкриває нові можливості для створення високопродуктивних, масштабованих та економічних систем.

1.5 Постановка задачі дослідження

Об’єктом дослідження є високонавантажені системи, що працюють у serverless середовищі та побудовані на основі подієво-орієнтованої архітектури (event-driven architecture).

Предметом дослідження є моделі проектування і патерни, що дозволяють забезпечити високу продуктивність, надійність та масштабованість систем, побудованих на основі подієво-орієнтованих принципів у serverless архітектурі. В даному дослідженні розглядаються підходи та стратегії, що забезпечують ефективну взаємодію компонентів системи, зокрема:

– подієво-орієнтована архітектура - дослідження принципів роботи систем, в яких події стають каталізаторами для виконання операцій та забезпечують асинхронну обробку даних;

– serverless архітектура - вивчення особливостей serverless середовища, що дозволяє зменшити витрати та спростити управління інфраструктурою завдяки автоматичному масштабуванню обчислювальних ресурсів;

– моделі та патерни проектування - аналіз поширених моделей і патернів, які забезпечують ефективність, стійкість до навантажень і високу продуктивність у системах, що використовують event-driven та serverless підходи.

Дослідження передбачає збір, аналіз та порівняння інформації про принципи подієво-орієнтованої та serverless архітектури, а також аналіз відповідних патернів проектування.

Вхідні дані:

– загальна інформація про принципи побудови подієво-орієнтованих систем;

– опис архітектурних особливостей serverless середовища;

– детальний опис патернів проектування, що використовуються для створення високонавантажених систем у serverless середовищі;

– практичні приклади застосування event-driven патернів та їх вплив на продуктивність і масштабованість додатків.

Вихідні дані:

– порівняльний аналіз event-driven та serverless підходів з точки зору продуктивності, надійності та економічної ефективності;

– рекомендації щодо вибору та реалізації оптимальних патернів для побудови масштабованих високонавантажених систем у serverless середовищі.

Метою дослідження є аналіз ефективності подієво-орієнтованих підходів у serverless середовищі для високонавантажених систем, а також визначення оптимальних патернів, що забезпечують максимальну продуктивність і надійність таких систем.

Для досягнення поставленої мети необхідно вирішити такі задачі:

– провести аналіз ключових термінів та понять, пов'язаних з event-driven та serverless архітектурою;

– дослідити проблему зв’язності у event-driven архітектурі;

– дослідити існуючі патерни проектування та їх застосування у високонавантажених системах;

– вивчити переваги та обмеження використання event-driven патернів у serverless середовищі;

– навчитись моделювання високонавантажених систем на основі event-driven підходів у serverless середовищі для визначення впливу різних факторів на продуктивність та стабільність системи.

Тема дослідження є актуальною у зв'язку зі зростаючим попитом на гнучкі, масштабовані та економічно ефективні рішення для високонавантажених систем. Завдяки своїй здатності забезпечувати асинхронну обробку та автоматичне масштабування, подієво-орієнтовані та serverless архітектури стають усе більш популярними в індустрії. З огляду на стрімкий розвиток хмарних технологій та зростання обсягів даних, застосування таких архітектур дозволяє створювати стійкі до збоїв і продуктивні системи, які відповідають сучасним вимогам до цифрових продуктів.

Наукова новизна дослідження полягає в комплексному аналізі подієво-орієнтованих моделей проектування у serverless середовищі, а також у розробці рекомендацій щодо вибору оптимальних патернів для високонавантажених систем. У дослідженні буде проведено порівняння та оцінка ефективності різних підходів з точки зору їх здатності забезпечити масштабованість, продуктивність та економічну ефективність, що надасть нові знання для побудови більш ефективних та адаптивних серверних систем.

РОЗДІЛ 2

ОПТИМІЗАЦІЯ ПОДІЙ У EVENT-DRIVEN АРХІТЕКТУРІ ДЛЯ ЗМЕНШЕННЯ ЗВ’ЯЗНОСТІ

2.1. Теоретичні основи Event-Driven Architecture

Висока складність коду й надмірна зв'язність між компонентами вимагають застосування подієво-орієнтованої архітектури (event-driven architecture). Вона будує систему з автономних сервісів, які спілкуються через події, тримаючи зв'язність на мінімумі. Кожен сервіс - окремий незалежний компонент, тож зміни в одному не змушують переробляти всю систему. Це полегшує і супровід, і розвиток проекту під нові вимоги.

Під «подією» в подієво-орієнтованій архітектурі розуміється деяка структура інформації. Подія - це порція інформації про факт, який уже стався і має вагу для бізнесу. Реєстрація користувача, оформлення замовлення, додавання товару до каталогу - усе це події, що мають значення для системи й запускають подальші процеси.

Основна ідея подієво-орієнтованої архітектури - вона повинна дозволяти вносити зміни лише в конкретні, ізольовані сервіси відповідно до нових вимог бізнесу. З "перевантаженою" подією це стає неможливим. Розробники створили монолітну систему, замасковану під розподілену подієву архітектуру.

Незважаючи на наявність подієво-орієнтованої структури, фактично створюємо монолітну систему, де всі частини надто залежать одна від одної. Замість справді автономних сервісів маємо розподілений моноліт, який ускладнює реалізацію гнучких змін без серйозних збоїв. Але далі потрібно дійсно розв'язати ці залежності та зробити сервіси по-справжньому автономними.

Щоб зробити подію більш гнучкою та зменшити її зв'язність, нам слід зробити її менш зв’язаною - видалити зайву інформацію та розподілити дані між

окремими сервісами. Це дозволить зменшити залежність між сервісами, які взаємодіють з подією, і підвищити автономність кожного з них.

Поряд із подією варто поставити ще один елемент архітектури - команду. Команда - це вказівка до дії, яка ще не виконана. Її застосовують, щоб ініціювати операцію: розмістити замовлення, змінити адресу доставки. Команда й подія часто ходять у парі. Коли команда успішно відпрацьовує, вона породжує подію, на яку вже реагують інші компоненти.

Принципова різниця ось у чому: команда завжди має одного конкретного адресата - код, який її виконує. Тому вона робить чітко окреслену дію, адже відповідає за неї лише один обробник. Команду «PlaceOrder» опрацьовує тільки той сервіс, що відповідає за замовлення. Завдяки такій ізольованості і саму команду, і її обробник легко змінювати чи розвивати, не зачіпаючи решту системи.

З подіями інакше - у них може бути кілька підписників. Одна подія здатна викликати реакцію відразу в кількох сервісах.

Без розуміння різниці між подіями та командами складно побудувати архітектуру з прозорою структурою - таку, де залежностями зручно керувати, а система лишається гнучкою під час розвитку й масштабування.

У сучасних розподілених системах Event-Driven Architecture (EDA) забезпечує асинхронну взаємодію між сервісами через події. Такий підхід дозволяє підвищити масштабованість, стійкість до відмов та незалежність компонентів системи.

Основними елементами EDA є producer, consumer, broker та event bus. Однією з ключових проблем EDA є надмірна зв’язність між сервісами.

Архітектура, керована подіями (Event-Driven Architecture, EDA), ґрунтується на принципі асинхронного обміну інформацією між компонентами системи за допомогою подій (events). У межах такої архітектури сервіси взаємодіють не через прямі виклики, а шляхом генерації та обробки повідомлень, що значно знижує рівень зв’язності (coupling) між компонентами. Основними

елементами EDA є producer, consumer, broker та event bus, кожен із яких виконує визначену функціональну роль.

Producer (виробник подій)

Producer - це компонент системи, який створює та надсилає події до середовища обміну повідомленнями. Подія формується у відповідь на певну дію, зміну стану або завершення бізнес-процесу. Producer не взаємодіє безпосередньо зі споживачами подій, а лише передає повідомлення до брокера або шини подій, що забезпечує слабку зв'язність між компонентами.

Наприклад, у системі електронної комерції сервіс обробки замовлень (*Order Service*) після оформлення покупки може сформувати подію **OrderCreated**, яка містить інформацію про номер замовлення, користувача та перелік товарів. Ця подія надалі буде оброблятися іншими сервісами без прямої залежності від сервісу замовлень.

Основними функціями producer є:

- генерація подій;
- формування структури повідомлення;
- передача подій до брокера;
- забезпечення асинхронної взаємодії між сервісами.

Перевагою producer є відсутність необхідності знати, які саме компоненти будуть обробляти подію, що дозволяє спростити масштабування системи.

Consumer (споживач подій)

Consumer - це компонент системи, який отримує, аналізує та обробляє події. Consumer підписується на певний тип повідомлень і виконує необхідні дії після їх надходження.

Наприклад, після отримання події **OrderCreated** сервіс платежів (*Payment Service*) може ініціювати процес списання коштів, а сервіс повідомлень (*Notification Service*) - надіслати електронний лист користувачу про успішне створення замовлення.

Consumer виконує такі функції:

- підписка на необхідні типи подій;

- аналіз вхідних повідомлень;
- запуск бізнес-логіки;
- обробка помилок та повторних спроб.

Важливою перевагою consumer є автономність роботи, оскільки сервіс може функціонувати незалежно від інших компонентів системи.

Broker (брокер повідомлень)

Broker - це проміжний компонент, який відповідає за приймання, збереження, маршрутизацію та доставку подій між producer і consumer. Брокер є центральним механізмом організації асинхронної взаємодії в EDA.

До популярних брокерів повідомлень належать:

- RabbitMQ;
- Apache Kafka;
- ActiveMQ;
- Redis Streams.

Однією з ключових характеристик ефективності мікросервісної архітектури є рівень зв'язності (coupling) між окремими сервісами. У контексті розподілених інформаційних систем поняття *coupling* визначає ступінь взаємозалежності програмних компонентів, а також рівень впливу одного сервісу на функціонування іншого. У мікросервісній архітектурі низька зв'язність є важливою умовою забезпечення масштабованості, гнучкості, незалежного розгортання та спрощення супроводу програмного забезпечення.

Під **зв'язністю (coupling)** у мікросервісних системах розуміють рівень залежності між сервісами, який визначається необхідністю обміну даними, викликами функцій, використанням спільних ресурсів або синхронізацією процесів. Високий рівень зв'язності призводить до ускладнення внесення змін, зростання кількості помилок та підвищення ризику каскадних відмов у системі. Натомість низька зв'язність забезпечує автономність сервісів, що особливо важливо для Event-Driven Architecture, де взаємодія компонентів здійснюється через події.

У сучасних мікросервісних системах виділяють кілька основних типів зв’язності: часову, просторову, функціональну та інформаційну.

Часова зв’язність (*temporal coupling*) виникає тоді, коли один сервіс залежить від доступності іншого в конкретний момент часу. Наприклад, при використанні синхронного REST-виклику сервіс відправник очікує негайної відповіді від сервісу отримувача. Якщо цільовий сервіс недоступний, це може спричинити затримки або повне припинення роботи процесу. У Event-Driven архітектурі часова зв’язність зменшується за рахунок асинхронної передачі повідомлень через брокери, наприклад RabbitMQ, що дозволяє сервісам працювати незалежно один від одного.

Просторова зв’язність (*spatial coupling*) визначає залежність сервісу від конкретного розташування або адреси іншого сервісу. Наприклад, якщо мікросервіс жорстко прив’язаний до конкретної IP-адреси або endpoint, будь-які зміни конфігурації можуть призвести до втрати працездатності системи. Використання брокерів повідомлень та механізмів service discovery дозволяє зменшити просторову залежність і забезпечити гнучке масштабування компонентів.

Функціональна зв’язність (*functional coupling*) виникає у випадку, коли реалізація одного сервісу прямо залежить від внутрішньої логіки іншого. Наприклад, зміна бізнес-логіки сервісу оплати може вимагати внесення змін у сервіс обробки замовлень. Такий тип залежності є небажаним, оскільки ускладнює розвиток системи та зменшує можливість незалежного оновлення сервісів. Одним із методів зменшення функціональної зв’язності є використання стандартизованих подій і контрактів повідомлень.

Інформаційна зв’язність (*data coupling*) характеризується ступенем залежності сервісів від структури та формату даних. Наприклад, зміна JSON-схеми події може спричинити помилки у сервісах-споживачах. Для мінімізації інформаційної зв’язності використовуються механізми версіонування повідомлень, стандартизація схем подій та принцип backward compatibility.

Для оцінювання складності взаємодії сервісів використовуються відповідні критерії, що дозволяють кількісно визначити рівень зв’язності системи. До основних критеріїв оцінювання належать:

1. **Кількість залежностей між сервісами** - визначає число прямих взаємодій між компонентами системи. Чим менша кількість залежностей, тим простіше масштабувати архітектуру.

2. **Інтенсивність обміну повідомленнями** - характеризує частоту передачі подій між сервісами. Надмірна кількість повідомлень може спричиняти перевантаження брокера подій і зростання затримок.

3. **Рівень каскадних залежностей** - показує, наскільки помилка одного сервісу може впливати на інші компоненти системи.

4. **Затримка обробки подій (latency)** - визначає час між генерацією події та її обробкою сервісом-споживачем.

5. **Автономність сервісів** - оцінює можливість незалежного функціонування мікросервісу без необхідності постійної взаємодії з іншими компонентами.

Аналіз рівня зв’язності є важливим етапом проєктування Event-Driven систем, оскільки дозволяє виявити критичні залежності та обґрунтувати необхідність оптимізації потоків подій. Застосування асинхронної взаємодії, брокерів повідомлень та механізмів оптимізації Event-Driven Architecture сприяє зменшенню зв’язності між сервісами, підвищенню продуктивності та покращенню масштабованості інформаційної системи.

Особливості виникнення залежностей у Event-Driven системах

- залежність від формату повідомлень;
- надлишкова кількість подій;
- каскадні ланцюги подій;
- проблема централізованого orchestration.

2.2. Формалізація задачі оптимізації подій

Задача оптимізації подій полягає у мінімізації рівня зв’язності між сервісами при збереженні необхідної функціональної повноти та продуктивності системи з використанням критеріїв:

- мінімізація кількості залежностей;
- зменшення latency;
- збереження reliability;
- мінімізація дублювання подій.

Систему можна представити у вигляді орієнтованого графа G=(V,E), де V - множина сервісів, E - множина подій.

Задача оптимізації полягає у мінімізації інтегрального показника зв’язності між сервісами за умови збереження функціональності системи. Введено критерії оптимізації: мінімізація latency, зменшення дублювання подій, скорочення каскадних викликів. ***

2.3. Математична модель зменшення зв’язності

Для оцінки рівня coupling (з’єднання) використовується зважена матриця взаємодій між сервісами. Вага зв’язку визначається через частоту подій, затримку доставки та обсяг даних.

Запроваджується цільова функція оптимізації та інтегральний показник зв’язності, що мінімізується під час реконфігурації потоків подій.

У сучасних мікросервісних системах, побудованих за принципами Event-Driven Architecture (EDA), ефективність функціонування значною мірою залежить від рівня взаємозалежності між сервісами. Надмірна кількість залежностей між компонентами системи ускладнює масштабування, знижує продуктивність та підвищує ризик каскадних помилок. Для вирішення цієї проблеми необхідно формалізувати процес оцінювання та мінімізації зв’язності між сервісами.

У роботі розглядається математична модель зменшення зв’язності, що дозволяє кількісно оцінити ступінь взаємодії між сервісами та виконати оптимізацію потоків подій у середовищі RabbitMQ.

Представлення Event-Driven системи у вигляді графа для формалізації взаємодії між мікросервісами використовується орієнтований граф $G=(V,E)$, де:

- $(V=\{v_1,v_2,\dots,v_n\})$ - множина мікросервісів;
- $(E=\{e_{ij}\})$ - множина зв'язків між сервісами через події.

Кожен вузол графа відповідає окремому сервісу системи, а ребро описує передачу повідомлень через RabbitMQ.

2.4. Алгоритм оптимізації Event-Driven взаємодії

Алгоритм передбачає збір телеметрії, побудову графа взаємодій, аналіз критичних ребер та адаптивну оптимізацію.

Використовуються підходи агрегації подій, фільтрації та rerouting через RabbitMQ. Після оптимізації проводиться повторна оцінка метрик ефективності.

Одним із ключових етапів побудови ефективної Event-Driven Architecture є оптимізація взаємодії між мікросервісами. У розподілених системах велика кількість подій, дублювання повідомлень та надлишкові взаємозалежності між сервісами можуть призводити до перевантаження брокера повідомлень, збільшення затримок та зниження продуктивності системи. Для усунення зазначених проблем у роботі пропонується алгоритм оптимізації Event-Driven взаємодії, спрямований на зменшення рівня зв'язності між сервісами та підвищення ефективності маршрутизації подій.

Мета алгоритму оптимізації

Основною метою алгоритму є мінімізація рівня coupling між сервісами шляхом:

- зменшення кількості прямих залежностей;
- усунення дублювання повідомлень;
- оптимізації потоків подій;
- підвищення автономності сервісів;
- зниження затримок обробки повідомлень.

У межах дослідження задача оптимізації формулюється як мінімізація інтегрального показника зв'язності.

Алгоритм роботи системи

Послідовність виконання алгоритму

Початок
↓
Збір статистики подій
↓
Побудова графа взаємодії
↓
Розрахунок coupling
↓
Пошук критичних зв’язків
↓
Оптимізація потоків подій
↓
Оцінка ефективності
↓
Кінець

2.5. Практичне застосування оптимізації (Node.js + TypeScript + RabbitMQ)

Розглянуто приклад системи обробки онлайн-замовлень із сервісами Order Service, Payment Service та Notification Service. RabbitMQ використовується як брокер повідомлень для асинхронної взаємодії. Для демонстрації практичного застосування оптимізації подій у Event-Driven Architecture (EDA) розглянемо приклад системи обробки онлайн-замовлень, побудованої на основі мікросервісного підходу. У запропонованій архітектурі взаємодія між сервісами здійснюється асинхронно за допомогою брокера повідомлень RabbitMQ, що дозволяє зменшити рівень зв’язності між компонентами системи.

У рамках дослідження розглянуто три основні мікросервіси:

- **Order Service** - сервіс обробки замовлень;
- **Payment Service** - сервіс обробки платежів;
- **Notification Service** - сервіс сповіщень користувача.

Основним завданням системи є забезпечення автоматизованого процесу оформлення замовлення, підтвердження оплати та інформування користувача про стан виконання операцій.

Архітектура системи до оптимізації

У традиційній архітектурі мікросервіси часто використовують синхронні HTTP REST-запити для взаємодії між компонентами. У такому випадку Order Service безпосередньо викликає Payment Service для перевірки та підтвердження платежу, після чого виконується виклик Notification Service для відправлення повідомлення клієнту.

Послідовність взаємодії має вигляд:

Користувач

↓

Order Service

↓ REST API

Payment Service

↓ REST API

Notification Service

Недоліком такого підходу є висока часова та функціональна зв’язність між сервісами. Якщо Payment Service тимчасово недоступний, процес оформлення замовлення повністю зупиняється. Аналогічно, збій Notification Service може вплинути на завершення бізнес-процесу.

Основними проблемами синхронної взаємодії є:

- залежність сервісів від доступності один одного;
- зростання затримок обробки;
- підвищений ризик каскадних відмов;
- складність масштабування системи.

Використання RabbitMQ як брокера повідомлень

Для усунення зазначених недоліків у роботі пропонується використання RabbitMQ як брокера повідомлень для реалізації асинхронної взаємодії між сервісами.

RabbitMQ виступає проміжним рівнем між мікросервісами та забезпечує передачу повідомлень через черги. У цьому випадку сервіси не взаємодіють безпосередньо між собою, а обмінюються подіями через брокер повідомлень.

Архітектура системи після оптимізації має вигляд:

Користувач

↓

Order Service

↓ Event: OrderCreated

RabbitMQ

↓

Payment Service

↓ Event: PaymentCompleted

RabbitMQ

↓

Notification Service

Після створення замовлення Order Service формує подію **OrderCreated**, яка передається до RabbitMQ. Сервіс Payment Service підписується на відповідну чергу повідомлень та виконує обробку платежу. У випадку успішної оплати генерується нова подія **PaymentCompleted**, яка передається Notification Service для інформування користувача.

Такий механізм забезпечує низку переваг:

- зменшення часової зв’язності;
- відсутність прямої залежності між сервісами;
- підвищення стійкості до відмов;
- можливість незалежного масштабування;
- зниження навантаження на систему.

Приклад реалізації Order Service у Node.js та TypeScript

Після створення замовлення сервіс Order Service надсилає подію до RabbitMQ.

```
import amqp from 'amqplib';

async function sendOrderEvent(orderId: string) {
  const connection = await amqp.connect('amqp://localhost');
```

```
    const channel = await connection.createChannel();

    const queue = 'order_created';

    await channel.assertQueue(queue);

    channel.sendToQueue(
        queue,
        Buffer.from(
            JSON.stringify({ orderId })
        )
    );

    console.log('Order event sent');
}

sendOrderEvent('12345');
```

У наведеному прикладі після створення нового замовлення формується повідомлення, яке надсилається до черги **order_created**.

Приклад реалізації Payment Service

Сервіс Payment Service очікує надходження повідомлень від Order Service та виконує обробку платежів.

```
import amqp from 'amqplib';

async function consumeOrders() {
    const connection =
        await amqp.connect('amqp://localhost');

    const channel =
        await connection.createChannel();
```

```
    const queue = 'order_created';

    await channel.assertQueue(queue);

    channel.consume(queue, (message) => {

        if (message) {

            const order =
                JSON.parse(
                    message.content.toString()
                );

            console.log(
                `Processing order:
                ${order.orderId}`
            );

            channel.ack(message);
        }
    });
}

consumeOrders();
```

Payment Service обробляє повідомлення незалежно від Order Service, що значно знижує рівень зв’язності.

Оптимізація потоків подій

Існують кілька механізмів оптимізації:

1. **Фільтрація подій** - передача лише релевантних повідомлень між сервісами.
2. **Агрегація повідомлень** - об’єднання кількох однотипних подій у єдиний пакет.
3. **Асинхронна маршрутизація** - використання exchange-механізмів RabbitMQ для динамічного перенаправлення повідомлень.
4. **Буферизація повідомлень** - накопичення подій у чергах для уникнення перевантаження сервісів.

Для оцінювання ефективності оптимізації введено показник зменшення зв’язності. Використання RabbitMQ дозволяє знизити рівень прямої взаємозалежності між сервісами, зменшити кількість каскадних помилок та покращити загальну масштабованість системи.

Застосування Event-Driven Architecture із RabbitMQ є ефективним підходом для побудови гнучких та масштабованих мікросервісних систем, що особливо актуально для високонавантажених інформаційних платформ.

Показано архітектуру до та після оптимізації, а також приклади обробки повідомлень у Node.js та TypeScript.

Приклад псевдокоду алгоритму оптимізації

Input: Events, Services
Build Graph G(V,E)
Calculate coupling C
While C > threshold:
detect redundant events
aggregate messages
optimize routing
Recalculate metrics
Return optimized architecture

Висновки до розділу

Обґрунтовано можливість застосування Event-Driven Architecture із RabbitMQ як ефективного підходу для побудови гнучких та масштабованих мікросервісних систем, що особливо актуально для високонавантажених інформаційних платформ.

Показано, що використання RabbitMQ дозволяє знизити рівень прямої взаємозалежності між сервісами, зменшити кількість каскадних помилок та покращити загальну масштабованість системи.

Наведено архітектуру системи до та після оптимізації, а також приклади обробки повідомлень у Node.js та TypeScript.

РОЗДІЛ 3

ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ БЕЗСЕРВЕРНИХ АРХІТЕКТУР (SERVERLESS ARCHITECTURE)

3.1. Хмарні платформи та середовища виконання - програмне забезпечення безсервверної архітектура

Безсерверна архітектура (Serverless Architecture) - це модель розробки та розгортання програмного забезпечення, у якій керування серверною інфраструктурою виконує хмарний провайдер. Розробник зосереджується на бізнес-логіці застосунку, а масштабування, балансування навантаження, адміністрування серверів та забезпечення доступності здійснюються автоматично.

Основними моделями безсерверних обчислень є:

- **FaaS (Function as a Service)** - виконання окремих функцій за подією;
- **BaaS (Backend as a Service)** - використання готових серверних сервісів (автентифікація, БД, зберігання файлів, повідомлення тощо);
- **Event-Driven Architecture** - архітектура, керована подіями.

Основу програмного забезпечення безсерверної архітектури становлять хмарні платформи (табл.3.1).

Таблиця 3.1 − Платформи безсерверної архітектури

Платформа	Основні можливості	Підтримувані мови
Amazon Web Services Lambda	Виконання функцій за подіями, автоматичне масштабування	Python, Node.js, Java, C#, Go
Microsoft Azure Functions	Інтеграція з екосистемою Azure	C#, Python, JavaScript
Google Cloud Functions	Обробка подій у реальному часі	Python, Go, Java, Node.js
IBM Cloud Functions	OpenWhisk-базоване середовище	Python, Swift
Oracle Functions	Контейнеризовані serverless-функції	Java, Python

Для навігації до офіційних платформ використовуються AWS Lambda, Azure Functions, Google Cloud Functions, IBM Cloud Functions, Oracle Functions

Для створення та управління serverless-застосунками використовуються спеціалізовані фреймворки (табл. 3.2)

Таблиця 3.2 − Спеціалізовані фреймворки

Інструмент	Призначення
Serverless Framework	Автоматизація розгортання
AWS SAM	Моделювання serverless-застосунків
Terraform	Керування інфраструктурою
Knative	Kubernetes-based serverless
OpenFaaS	Власні FaaS-платформи

Системи зберігання даних

У serverless-архітектурі переважно застосовуються масштабовані NoSQL або керовані SQL-бази (табл. 3.3)

Таблиця 3.3 − Бази даних

База даних	**Тип**
Amazon DynamoDB	NoSQL
Firebase Firestore	Документоорієнтована
MongoDB Atlas	Хмарна NoSQL
Azure Cosmos DB	Глобально розподілена
PlanetScale	Serverless SQL

Для реалізації подієво-керованої взаємодії використовуються брокери повідомлень та event bus.

Таблиця 3.4 − Брокери повідомлень

Інструмент	**Призначення**
Apache Kafka	Потокова обробка подій
RabbitMQ	Асинхронні повідомлення
Amazon EventBridge	Подієва інтеграція
Azure Event Grid	Маршрутизація подій
Google Pub/Sub	Черги повідомлень

3.2 Технічне забезпечення безсерверних архітектур

Хоча модель називається «безсерверною», фізична інфраструктура все ж існує, але прихована від користувача та керується провайдером.

Основні компоненти технічного забезпечення:

3.2.1. Хмарні дата-центри складаються з: кластерів фізичних серверів; систем віртуалізації; мережевого обладнання; систем резервування живлення; систем охолодження.

3.2.2. Віртуалізація та контейнеризація використовуються: -Docker - контейнеризація; Kubernetes - оркестрація; microVM (наприклад Firecracker у AWS).

3.2.3. Мережеве забезпечення передбачає:балансувальники навантаження; API Gateway; CDN-мережі;захист від DDoS-атак; системи кешування.

3.2.4. Обчислювальні ресурси динамічно виділяються:CPU;RAM;GPU (для AI-задач); storage resources.

3.3 Структура програмно-технічного забезпечення serverless-системи

Користувач → API Gateway → Serverless Function → База даних → Event Broker → Notification Service

Serverless змінила хмарні застосунки: гнучка модель оплати, висока масштабованість і майже відсутнє операційне навантаження змінили звичні підходи. Та поряд із цими плюсами йдуть і складнощі, які доводиться передбачувати наперед.

Сильні сторони, так і проблеми, з якими стикаються команди при переході на Serverless [11].

Модель «pay-per-use» означає, що компанія платить рівно за спожиті ресурси й не вкладає зайвого в сервери та їх утримання. Не потрібно й керувати фізичною інфраструктурою, а автоматичне масштабування знімає частину рутини - у підсумку розробку вдається вивести на ринок швидше.

До ключових переваг безсерверних архітектур належать [12-13]:

економічна ефективність. Модель «pay-per-use» усуває витрати на простоювання інфраструктури та звільняє від початкових капітальних вкладень у сервери;

масштабованість. Система здатна автоматично збільшувати й зменшувати

кількість ресурсів залежно від поточного навантаження, без втручання адміністратора;

спрощені операції. Відсутність необхідності керувати серверами дозволяє розробникам зосередитися на бізнес-логіці, а не на інфраструктурних завданнях;

швидка розробка. Подієво-орієнтована модель (event-driven) спрощує архітектуру: функції запускаються у відповідь на тригери, що зменшує складність коду та прискорює впровадження нових можливостей;

глобальна доступність. Підтримка edge computing дає змогу розгортати функції в різних географічних регіонах, забезпечуючи зменшення затримки та швидшу відповідь для користувачів.

Попри всі плюси, перехід на безсерверні системи не обходиться без труднощів, і їх варто врахувати заздалегідь. Там, де висока вимога до доступності та швидкодії, доводиться стежити за ініціалізацією функцій: холодні старти б'ють по користувацькому досвіді й по сервісах реального часу. Не варто забувати й про безпеку - мультиорендна природа таких платформ розширює поверхню атаки, тож потрібні додаткові запобіжники.

Основні виклики, пов'язані з безсерверними архітектурами, включають:

затримка при «холодному старті». Функції, що не використовувалися певний час, потребують ініціалізації середовища перед виконанням, що може спричиняти помітні затримки; прив'язка до постачальника (Vendor Lock-in). Глибока інтеграція зі специфічними сервісами одного хмарного провайдера може ускладнювати або здорожчувати перенесення застосунку на іншу платформу; складність моніторингу та відлагодження. Традиційні методи спостереження та діагностики не завжди ефективні в розподілених системах, де кожен компонент є окремою функцією; ліміти на виконання. Провайдери встановлюють часові та ресурсні обмеження для безсерверних функцій, що може обмежувати використання у сценаріях з довготривалими або ресурсомісткими обчисленнями; неправильне управління витратами. Модель «pay-per-use» загалом економить кошти, проте некоректні налаштування (наприклад, надмірна кількість викликів функцій) можуть призвести до

несподіваних перевитрат; проблеми безпеки. Мультиорендне середовище підвищує загрозу потенційних вразливостей і вимагає ретельного шифрування даних та суворих політик доступу до ресурсів.

Планування систем на базі безсерверних архітектур - це постійний пошук балансу між перевагами та викликами [14]. Навіть коли операції спрощено, а на інфраструктурі заощаджено, нікуди не подіти ліміти виконання, ризик помилкових налаштувань функцій і ті самі холодні старти. Узагальнений порівняльний аналіз переваг і викликів зведено у табл. 3.5, що дає змогу системно зважити ймовірні вигоди та загрози під час розгортання й супроводу безсерверних систем.

Таблиця 3.5 − Таблиця аналізу переваг і викликів

Аспект	**Переваги**	**Виклики**
Витрати	Оплата лише за фактичне використання, відсутність простою ресурсів	Неправильне налаштування або перевищення викликів може призвести до зростання витрат
Масштабованість	Автоматичне масштабування, подієво-орієнтоване виконання	Обмеження з боку провайдера (час виконання, обсяг пам’яті тощо)
Операції	Спрощене керування, немає потреби утримувати фізичну інфраструктуру	Складність моніторингу та відлагодження в розподіленому середовищі
Продуктивність	Оптимальний підхід для високонавантажених сценаріїв	Затримка при «холодному старті» може бути критичною для застосунків реального часу
Гнучкість	Швидкий цикл розробки, глобальна доступність	Прив’язка до постачальника (Vendor Lock-in)

Безпека	Частина завдань (оновлення, захист інфраструктури) передана провайдеру	Мультиорендність може підвищувати ризики та вимагає додаткових заходів безпеки

Для наочності можна уявити порівняння переваг та викликів у вигляді концептуальної стовпчастої діаграми, де (рис. 3.1):

вісь X: Ключові метрики (Витрати, Масштабованість, Операції, Продуктивність, Безпека);

вісь Y: Відносний «рівень впливу» (позитивний та негативний);

два стовпчики на кожну метрику: один відображає переваги (зазвичай позначається зеленим кольором), інший - виклики (червоним кольором).

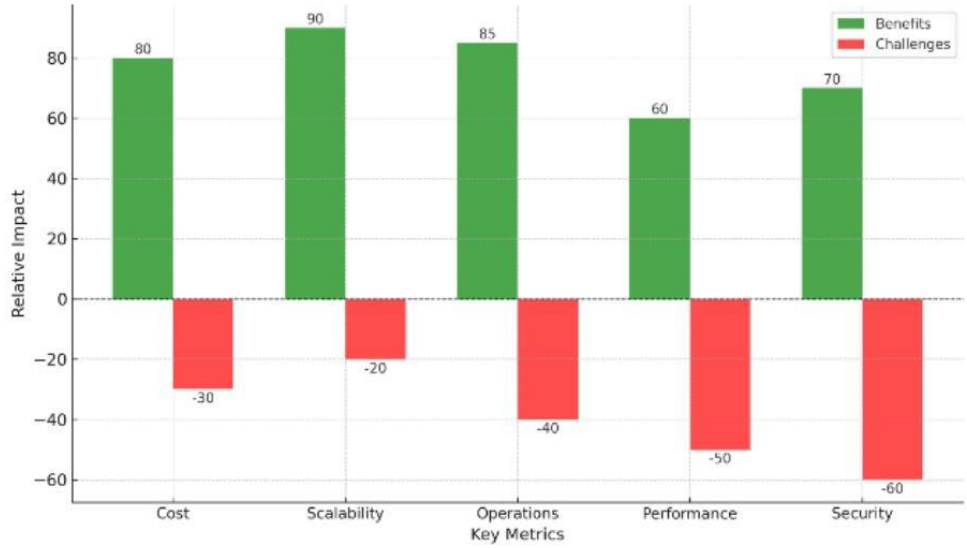


Рисунок 3.1 − Графік порівняння переваг і недоліків

Чим вищий «зелений» стовпчик, тим більше переваг за цією метрикою, тоді як «червоний» показує труднощі, на які варто зважати. У графі продуктивності перевагою стає висока масштабованість на піках навантаження, але cold start здатен помітно ускладнити роботу застосунків реального часу.

Аналіз serverless рішень

Ринок хмарних обчислень сьогодні тримають три гравці, кожен зі своїм рішенням для безсерверних (Serverless) архітектур: Amazon Web Services

(AWS) з AWS Lambda, Microsoft Azure з Azure Functions і Google Cloud Platform (GCP) з Google Cloud Functions [15-16]. Концептуально вони близькі, проте в кожного є свої нюанси, і їх варто зважити, обираючи платформу для високонавантажених систем.

Продуктивність у безсерверному середовищі визначається низкою показників, серед яких найважливішими є затримка (latency), пропускна здатність (throughput) та ефективність виконання (execution efficiency). З практичного погляду, основні аспекти продуктивності можна описати так:

затримка при «холодному старті»: затримка, що виникає при ініціалізації нового контейнера або середовища виконання, якщо функція не викликалася протягом певного часу;

швидкість виконання: тривалість виконання функції після її запуску;

робота з паралельними процесами: можливість одночасно обробляти кілька запитів без погіршення часу відгуку.

Порівняння платформ:

AWS Lambda: вважається лідером у зменшенні «холодних стартів» за допомогою механізмів попередньої інагурації контейнерів. Під час високих навантажень Lambda демонструє стабільність та оперативне масштабування;

Azure Functions: також ефективно працює з паралельними запитами, утім, за значних пікових сплесків можливе незначне збільшення затримки ініціалізації;

Google Cloud Functions: завдяки оптимізаціям для ML-застосунків платформа забезпечує хорошу швидкість виконання, однак за різких короткочасних «сплесків» (burst workloads) інколи спостерігаються затримки, дещо більші порівняно з AWS Lambda.

Серверлес-платформи славляться здатністю автоматично масштабуватися залежно від подій (тригери) та обсягу вхідних запитів, що робить їх привабливими для застосунків із непередбачуваним або «піковим» навантаженням. Основні чинники, що впливають на масштабованість:

максимальна паралельність: межі максимальної кількості паралельних

екземплярів (functions), які здатна розгорнути платформа;

швидкість масштабування: швидкість, із якою платформа реагує на значні коливання навантаження;

адаптивність до сценаріїв використання в реальному часі: здатність обробляти непередбачувані або нерівномірні робочі навантаження (наприклад, події IoT, онлайнові флеш-розпродажі тощо).

Порівняння платформ:

AWS Lambda: підтримує до 1 000 одночасних викликів функцій на обліковий запис із майже миттєвою реакцією на сплески;

Azure Functions: демонструє високі можливості вертикального та горизонтального масштабування, особливо в середовищах корпоративних замовників;

Google Cloud Functions: масштабується також досить швидко, однак при екстремально великих обсягах трафіка іноді помітна невелика затримка перед досягненням максимальної кількості екземплярів.

Модель оплати «pay-per-use» є однією з головних переваг безсерверних платформ: користувачі сплачують лише за час виконання функцій, виділену пам’ять та кількість запитів. Проте слід ураховувати й додаткові витрати:

модель ціноутворення: усі три провайдери (AWS, Azure, GCP) дотримуються логіки оплати за фактичне використання;

приховані витрати: витрати на передавання даних, інструменти моніторингу або непрямі витрати, пов’язані з «простієм» (idle resources);

придатність для різних робочих навантажень: у залежності від того, чи є робоче навантаження постійним або епізодичним, вартість може відчутно різнитися.

Порівняння платформ:

AWS Lambda: має конкурентоспроможні тарифи на виконання функцій, проте витрати на передавання даних (особливо в міжрегіональних сценаріях) можуть бути відчутними;

Azure Functions: дещо вищі базові тарифи на виконання, зате зручніший та

гнучкіший механізм формування рахунків для підприємств;

Google Cloud Functions: оптимальний варіант для невеликих або нерегулярних навантажень, однак за значної кількості викликів витрати можуть швидко зростати.

Зручність платформи для розробників прямо впливає на швидкість та простоту впровадження нових функцій і сервісів. Серед ключових факторів:

підтримувані мови: широкий перелік мов підвищує гнучкість і спрощує перехід команд між різними стеками технологій;

інструментарій та SDK: наявність якісних CLI, бібліотек та інструментів відлагодження дає змогу швидко знаходити й виправляти помилки;

інтеграція з екосистемними послугами: можливість простої інтеграції з хмарними базами даних, ML-сервісами та інструментами DevOps.

Порівняння платформ:

AWS Lambda: має дуже розгалужену екосистему, включно з готовими рішеннями для CI/CD, але новачкам може бути складно через велику кількість налаштувань;

Azure Functions: найбільш зручний у корпоративних середовищах Microsoft (C#, .NET), де інтегрований із іншими сервісами Azure;

Google Cloud Functions: відомий простотою та глибокою інтеграцією з AI/ML-інструментами Google, що робить його привабливим для спеціалізованих проєктів.

У табл. 3.6 наведено порівняльну оцінку ключових метрик для головних хмарних провайдерів.

Таблиця 3.6 − Таблиця порівняльних метрик

Параметр	**AWS Lambda**	**Azure Functions**	**Google Cloud Functions**
Затримка при «холодному старті»	~100 мс	~200 мс	~150 мс

Максимальна паралельність	1 000 екземплярів	1 000+ (налаштовується)	1 000 екземплярів
Вартість виконання	$0.20 за 1 млн викликів	$0.22 за 1 млн викликів	$0.18 за 1 млн викликів
Швидкість масштабування	Майже миттєве	Швидке	Помірне
Підтримувані мови	10+	7+	6+
Зручність відлагодження	Середня	Висока	Висока

Нижче наведено концептуальний графік (рис. 3.2), що ілюструє, як кожен провайдер масштабуватиметься за умови поступового зростання робочого навантаження:

вісь X: час (секунди);

вісь Y: кількість активних екземплярів;

лінії: показують поведінку AWS Lambda, Azure Functions і Google Cloud Functions під час збільшення трафіку.

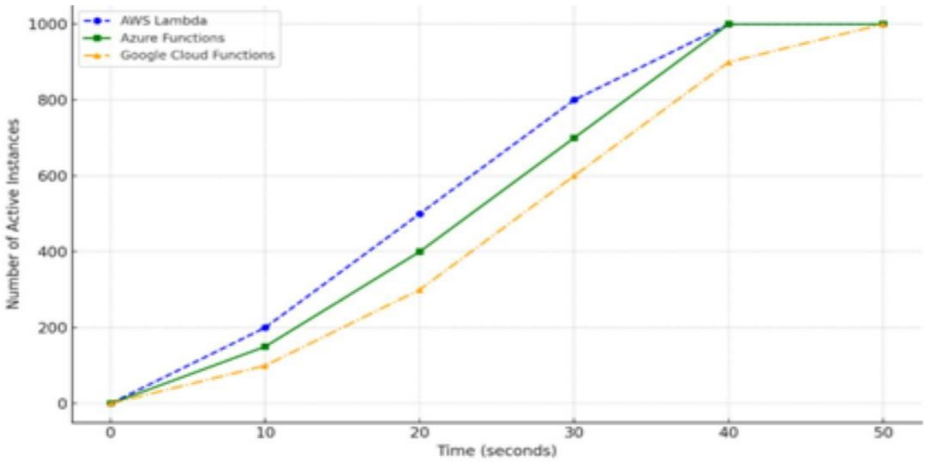


Рисунок 3.2 − Графік масштабування при навантаженні

Зі схематичного прикладу видно, що:

AWS Lambda майже миттєво досягає максимально доступного рівня

паралельності та швидко адаптується до «сплескових» навантажень;

Azure Functions масштабуються поступово, але достатньо впевнено, що робить цей сервіс привабливим у корпоративних сценаріях із більш рівномірними навантаженнями;

Google Cloud Functions демонструє незначну затримку в порівнянні з AWS за екстремальних навантажень, проте зрештою вийде на той самий рівень масштабування.

3.4 Аналіз архітектурних патернів у serverless event-driven системах

Побудовані на подіях serverless системи відкривають простір для ефективного опрацювання високих навантажень і дають гнучкість у масштабуванні. Щоб цей потенціал спрацював, доводиться долати специфічні труднощі - насамперед координацію компонентів, узгодженість даних і керування потоками подій.

Тут на допомогу приходять спеціалізовані архітектурні патерни: вони підвищують продуктивність, послаблюють зв’язність і додають стійкості до збоїв. Ключову роль відіграють патерни асинхронного обміну повідомленнями та керування транзакціями - Asynchronous Request-Reply, Choreography, Pipes and Filters, Priority Queue і Saga [17-18]. Кожен по-своєму спрощує обробку великих обсягів даних і робить систему витривалішою до змін та навантажень.

Наведемо характеристику кожноно патерн окремо - оцінимо, що він дає розподіленій архітектурі, і сформулюємо, коли його варто застосовувати у високонавантажених serverless системах.

Патерн Asynchronous Request-Reply

Asynchronous Request-Reply - один із базових патернів для serverless і event-driven систем, бо саме він дає їм гнучкість і масштабованість під великим навантаженням. Суть проста: налагодити взаємодію клієнта й сервера тоді, коли обробка на бекенді триває довго, а тримати синхронне з’єднання до самого кінця операції неможливо.

У сучасних розподілених системах клієнтські додатки часто взаємодіють із віддаленими API, щоб виконувати бізнес-логіку та отримувати дані. Взаємодія

найчастіше відбувається через HTTP(S) із використанням REST, де запити клієнта очікують швидкої відповіді. Проте в деяких випадках API потребує значного часу для обробки запиту, що може включати тривалі операції, такі як обробка даних, взаємодія з іншими сервісами, доступ до великих баз даних чи запуск обчислювальних задач. Затримки можуть бути викликані кількома факторами:

продуктивність серверного середовища, де розміщений додаток;

налаштування безпеки та доступу;

географічна відстань між клієнтом та сервером;

поточне навантаження на мережу і сервер;

розмір запиту та його обробка на сервері.

Навіть при оптимізації архітектури, є випадки, коли затримка є неминучою, особливо для тривалих задач. Це призводить до проблем із продуктивністю і вимог до клієнта очікувати завершення процесу, що може суттєво знижувати користувацький досвід. У таких випадках стандартний синхронний підхід до запиту-відповіді може бути неефективним і викликати перевантаження.

Патерн Asynchronous Request-Reply забезпечує обхід цієї проблеми, дозволяючи розділити запит і відповідь у часі та відкласти відповідь на запит до моменту, коли сервер готовий надати результат (рис. 3.3).

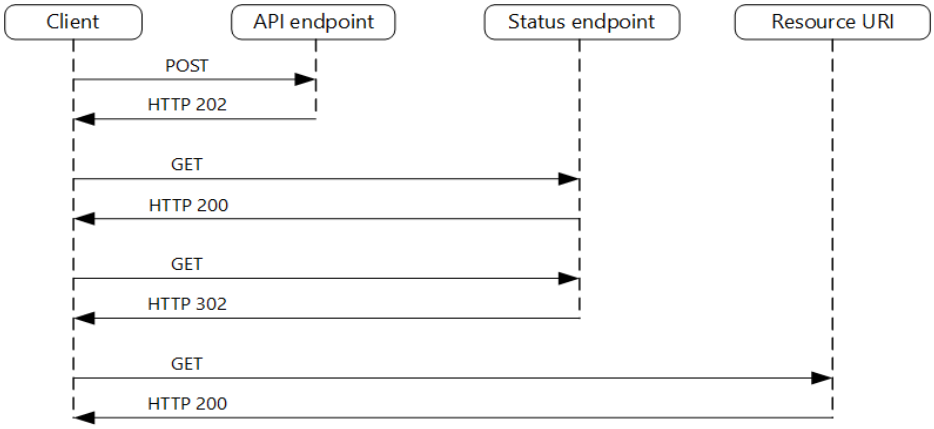


Рисунок 3.3 − Діаграма послідовності обробки запиту

Патерн реалізується наступним чином:

ініціація запиту: клієнтська програма надсилає запит на виконання тривалої операції до API. Після отримання запиту сервер негайно відповідає клієнту кодом HTTP 202 (Accepted), що вказує на те, що запит прийнятий в обробку, але ще не завершений;

посилання на статус: у відповідь клієнту API додає заголовок Location, що містить URL-адресу для перевірки статусу запиту. Клієнт може періодично звертатися за цією адресою, щоб дізнатися про прогрес операції, уникаючи необхідності утримувати підключення до серверу весь час;

окрема обробка запиту: сервер передає запит на виконання в інший компонент, як-от чергу повідомлень або окрему службу для асинхронної обробки. Це дозволяє серверу розвантажитися та продовжити обробку нових запитів без затримок;

полінг статусу: клієнтська програма надсилає періодичні запити на URL статусу (визначеного заголовком Location), щоб перевірити, чи виконана операція. Поки операція триває, сервер повертає HTTP 200 зі статусом "в обробці". Коли операція завершена, сервер може повернути HTTP 302 для переспрямування на ресурс із результатом або HTTP 200 зі статусом "готово".

Цей підхід дозволяє клієнту зберігати функціональність, не блокуючи інші операції, що важливо для динамічних користувацьких інтерфейсів, особливо в браузерних додатках.

Реалізація Asynchronous Request-Reply вимагає врахування деяких особливостей:

заголовок Location: сервер має вказати URL для перевірки статусу, де клієнт може отримувати оновлення про стан операції. В окремих випадках можна використовувати механізм SAS токенів для забезпечення безпеки доступу до цього URL;

заголовок Retry-After: цей заголовок встановлює інтервал, через який клієнт може повторити запит для оновлення статусу. Це допомагає уникнути надмірного навантаження на сервер, встановлюючи оптимальний час для повторного звернення;

розгляд клієнтської поведінки: не всі клієнти можуть мати вбудовану підтримку асинхронного запиту-відповіді, особливо це стосується старих систем або простих клієнтів, побудованих на безкодових платформах. У таких випадках необхідно використовувати проксі-сервіс, який може обробляти асинхронний API та надавати результат у зручному для клієнта форматі;

завершення операції: коли операція завершена, ресурс, вказаний у Location, має повернути HTTP 200 (OK), 201 (Created) або 204 (No Content) залежно від типу завершеної операції. У випадку помилки бажано, щоб сервер повернув відповідний код помилки (4xx) і зберіг інформацію про помилку в зазначеному ресурсі;

проблеми із сумісністю: старі клієнти можуть не підтримувати такий підхід, що вимагає використання фасаду або посередника між API та клієнтом для підтримки асинхронного зв’язку без необхідності переналаштування клієнтської програми.

Asynchronous Request-Reply є доцільним у таких сценаріях:

клієнтські програми: для браузерних додатків і SPA (Single Page Applications), де важко організувати зворотний виклик або використання довготривалих з’єднань;

сервер-серверні виклики: для взаємодії між сервісами, що працюють через HTTP і не можуть отримувати зворотні виклики через обмеження інфраструктури;

спадкові архітектури: для підтримки інтеграції з існуючими системами, які не підтримують сучасні технології зворотного зв’язку, такі як WebSockets чи Webhooks.

Цей патерн може бути менш ефективним, якщо:

потрібна негайна відповідь у реальному часі;

мережевий дизайн дозволяє відкрити порти для асинхронних зворотних викликів або WebSockets;

потрібне асинхронне повідомлення про завершення операції через такі сервіси, як Azure Event Grid.

У serverless і event-driven системах Asynchronous Request-Reply відчутно зрізає затримки, береже продуктивність сервера й тримає користувацький досвід плавним. Довгі операції опрацьовуються спокійніше: клієнт одразу отримує підтвердження, що запит прийнято, а сервер тим часом вільний обробляти інші звернення, не блокуючись.

Така схема добре масштабується, а це особливо цінно для високонавантажених систем і хмарних сервісів, де гнучкість і децентралізація обробки виходять на перший план.

Патерн Priority Queue

Priority Queue (пріоритетна черга) - підхід для хмарних розподілених систем, який сортує завдання за важливістю й пропускає критичні операції поперед менш термінових. Це стає в пригоді там, де треба витримувати рівні обслуговування (SLA), тримати якість роботи з користувачами та загальну продуктивність. Механізм спирається на черги повідомлень: кожне повідомлення має свій пріоритет, і завдання з вищим пріоритетом обробляються першими.

Часто системи мають справу з великим потоком завдань, що відрізняються за ступенем важливості. Наприклад, такі операції, як обробка фінансових транзакцій або швидка реакція на дії користувача в реальному часі, потребують негайної уваги, тоді як менш важливі завдання, як-от аналітичні звіти, можуть виконуватися в фоновому режимі. У разі відсутності пріоритизації завдання зазвичай обробляються у порядку їхнього надходження, що може призвести до затримок у виконанні критичних операцій. Це може негативно вплинути на користувацький досвід та порушити договірні SLA.

Патерн Priority Queue забезпечує обробку завдань на основі їхнього пріоритету, дозволяючи системі адаптуватися до різноманітних вимог та швидко реагувати на потреби, забезпечуючи обробку важливих завдань в першу чергу. У цьому патерні кожне повідомлення в черзі має пріоритетний рівень, і споживачі обробляють повідомлення з урахуванням цього пріоритету. Це особливо ефективно у випадках, коли система обслуговує різні групи клієнтів,

яким надаються різні рівні сервісу, або коли навантаження на систему змінюється і потребує адаптивного управління.

Priority Queue може бути реалізовано через дві основні архітектури: єдина черга з пріоритетами (див. рис. 3.4) або кілька окремих черг для різних рівнів пріоритету (див. рис. 3.5).

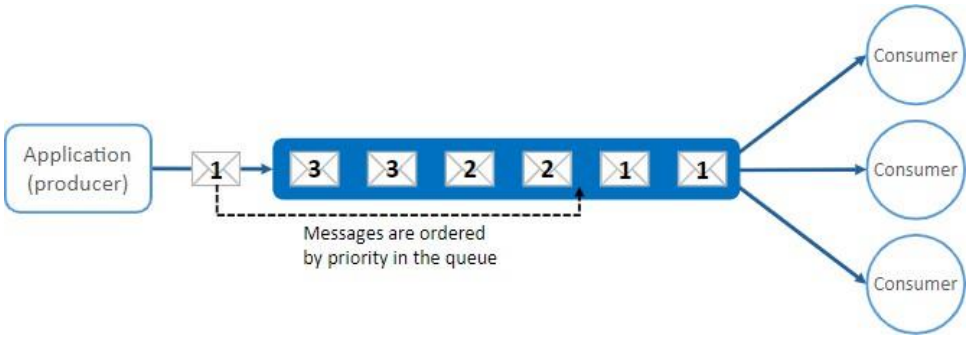


Рисунок 3.4 − Єдина черга з пріорітетами

У першому випадку всі повідомлення надходять в одну чергу, де вони сортуються за пріоритетом, що забезпечує обробку критичних завдань раніше, незалежно від часу їх надходження. У випадку з кількома чергами кожен рівень пріоритету має власну чергу, що може обслуговуватися або єдиним пулом споживачів, або ж окремими пулами для кожної черги.

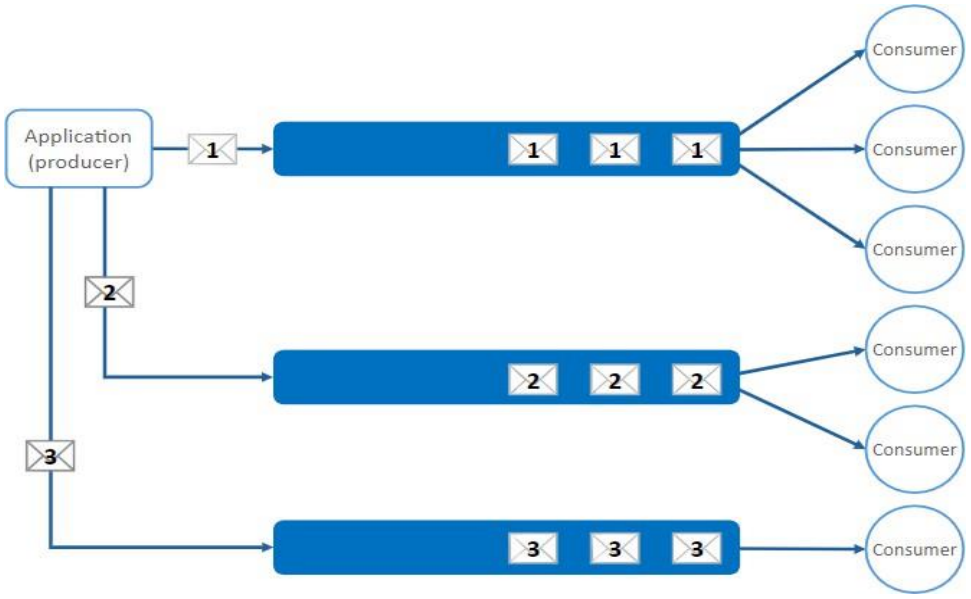


Рисунок 3.5 − Підхід з кількома чергами

Підхід з кількома чергами дозволяє зберігати ізоляцію між різними

рівнями пріоритету, що забезпечує більш високу надійність та контроль над продуктивністю кожного виду завдань. Така конфігурація особливо корисна для складних додатків, де різні черги вимагають різних характеристик обробки та гарантій продуктивності.

Реалізація Priority Queue вимагає продуманого підходу до визначення рівнів пріоритету, управління ресурсами та моніторингу витрат. Важливо чітко встановити пріоритети для різних типів завдань, щоб відповідати бізнес-вимогам і забезпечити оптимальний розподіл ресурсів. Наприклад, високопріоритетні завдання можуть вимагати швидкої обробки з мінімальними затримками, тоді як для низькопріоритетних завдань можна встановити певні обмеження за часом виконання, що дозволить уникнути їх накопичення в черзі. У випадку використання кількох черг, корисно динамічно масштабувати ресурси відповідно до змін у навантаженні на кожну чергу, а також дбати про дотримання фінансових лімітів, зважаючи на потенційні витрати на додавання, отримання та перевірку повідомлень у чергах.

Використання патерну Priority Queue є доречним у різних сценаріях. Наприклад, у системах підтримки клієнтів преміум-клієнти можуть отримувати пріоритет у розгляді їхніх запитів, у фінансових системах транзакції з високою вартістю обробляються першочергово, а в IoT-рішеннях пріоритет надається критичним сигналам від сенсорів. Завдяки такій конфігурації організації можуть ефективно реалізовувати Priority Queue для обробки різних завдань відповідно до їхнього рівня терміновості, що оптимізує використання ресурсів і відповідає очікуванням клієнтів.

Патерн Pipes and Filters

Pipes and Filters - поширений вибір для високонавантажених систем, коли роботу можна розкласти на самостійні етапи з власною логікою. Якщо обробка складається з кількох важких послідовних кроків, її варто розбити на незалежні елементи - фільтри: так зростають продуктивність, масштабованість і повторне використання компонентів. Фільтри працюють окремо один від одного, а з’єднують їх pipes, що передають дані від фільтра до фільтра.

У сучасних системах часто виникає потреба обробляти дані через послідовність завдань. Простий, але обмежений підхід полягає в обробці даних у монолітному модулі, що призводить до щільної зв'язаності та ускладнює рефакторинг, оптимізацію або повторне використання коду, якщо подібна обробка необхідна в інших частинах додатка. Монолітні архітектури також обмежують можливості масштабування окремих етапів обробки, оскільки всі процеси виконуються в одному середовищі. У випадку, коли певні завдання є ресурсомісткими, було б доцільно виконувати їх окремо на потужнішому обладнанні, залишаючи менш вимогливі завдання на менш затратних ресурсах.

Патерн Pipes and Filters передбачає розбиття завдань на окремі компоненти (фільтри), кожен з яких виконує одну функцію. Фільтри можуть приймати повідомлення, трансформувати його або перевіряти його на відповідність певним умовам. Кожен фільтр є самодостатнім, статичним і зазвичай без стану, що дозволяє гнучко поєднувати їх у різноманітні комбінації. Фільтри працюють незалежно від інших компонентів і лише знають свої схеми входу та виходу, що дозволяє розташовувати їх у будь-якому порядку.

Завдяки такому розподілу робочого навантаження патерн забезпечує можливість створювати нові ланцюги обробки із наявних фільтрів, оновлювати або замінювати окремі фільтри, змінювати порядок фільтрів та виконувати їх паралельно на різному обладнанні (див. рис. 3.6). Наприклад, фільтри з великими вимогами до обчислювальних ресурсів можуть виконуватись на високопродуктивному обладнанні, тоді як менш вимогливі фільтри можуть оброблятись на бюджетних платформах. Також фільтри можуть розташовуватись у різних дата-центрах або географічних локаціях, що дозволяє кожному елементу отримувати доступ до необхідних ресурсів.

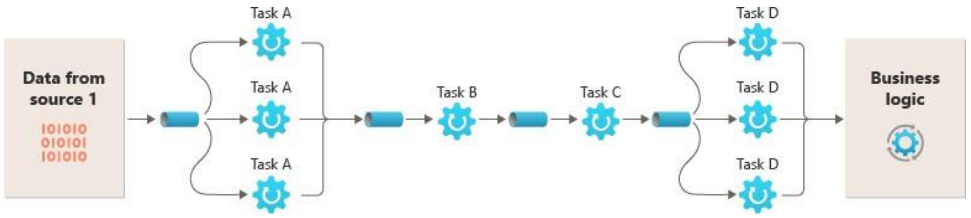


Рисунок 3.6 − Приклад реалізації патерну

Паралельна обробка на основі потоків також сприяє збільшенню продуктивності. Перший фільтр у ланцюзі може почати обробку даних і передати результат наступному фільтру ще до завершення обробки всього масиву даних. Це дозволяє підвищити загальну швидкість обробки, розвантажуючи систему, особливо коли потік даних надходить з великою частотою.

Додатково, у цьому патерні можна використовувати підхід Compensating Transaction для розподілених транзакцій, що дозволяє впроваджувати компенсуючі завдання в окремих фільтрах.

При реалізації патерну Pipes and Filters важливо врахувати кілька аспектів. По-перше, система має бути готова до складності, пов'язаної з гнучкістю цього патерну, адже незалежні фільтри можуть знаходитися на різних серверах, що збільшує складність налаштування та моніторингу. Крім того, потрібно забезпечити надійність передачі даних між фільтрами, адже втрата інформації може призвести до збою у всьому ланцюзі.

Також потрібно враховувати ідемпотентність кожного фільтра, що означає, що фільтри повинні бути здатні повторювати свою роботу без побічних ефектів. У випадку збою в процесі обробки фільтр повинен уникати дублювання результатів, інакше подальші фільтри можуть обробити ті самі дані повторно. Якщо в ланцюзі обробки використовуються черги повідомлень, наприклад, Azure Service Bus, інфраструктура черг може автоматично видаляти дублікати повідомлень, забезпечуючи надійну передачу.

Патерн Pipes and Filters ефективний, коли завдання можна легко розбити на набір незалежних етапів, кожен з яких має різні вимоги до масштабування. Наприклад, у випадках, коли потрібна гнучкість для зміни послідовності етапів обробки або необхідність додавання нових етапів, цей підхід є ідеальним рішенням. Однак патерн не підходить для задач, де процеси повинні виконуватися як єдина транзакція або де кожен етап обробки потребує спільного доступу до контексту чи стану.

Pipes and Filters - сильний інструмент для моделювання хмарних систем, яким потрібні масштабованість, гнучкість і можливість повторно використовувати компоненти.

Патерни для розподілених транзакцій

Коли транзакція зачіпає кілька мікросервісів, узгодженість даних рятує патерн Saga. Він дробить транзакцію на послідовність локальних, кожна з яких виконується самостійно. Не вдався якийсь етап - Saga сама запускає компенсуючі транзакції й відкочує попередні зміни.

Транзакція - це логічний блок операцій, що забезпечує цілісність даних. В рамках однієї транзакції дані переходять з одного узгодженого стану в інший, завдяки принципам атомарності, консистентності, ізоляції та стійкості (ACID). Ці властивості легко забезпечуються всередині одного сервісу, але стають проблемними, коли дані розподіляються між кількома сервісами, кожен з яких має свою базу даних.

Традиційний підхід двофазного підтвердження (2PC) вимагає, щоб усі учасники транзакції одночасно підтвердили або скасували операцію. Проте, сучасні хмарні середовища, такі як NoSQL бази даних і брокери повідомлень, часто не підтримують цю модель, що ускладнює керування транзакціями. Крім того, для реалізації розподілених транзакцій усі сервіси повинні бути доступні одночасно, що знижує загальну доступність системи.

Патерн Saga вирішує проблему транзакцій у розподілених системах за допомогою послідовності локальних транзакцій (рис. 3.7). Кожен етап оновлює власну базу даних і генерує повідомлення або подію, яка запускає наступний етап. Якщо один із етапів не вдається, патерн запускає компенсуючі транзакції, які анулюють попередні операції.

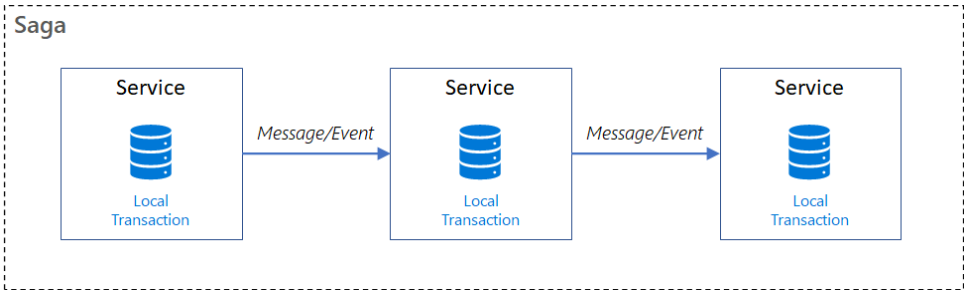


Рисунок 3.7 − Патерн Saga

У патерні Saga існує кілька ключових типів транзакцій:

компенсовані транзакції є тими, які можуть бути скасовані зворотною операцією;

поворотна точка - це етап, після якого система переходить до незворотного виконання транзакцій;

повторювані транзакції гарантують успіх виконання та йдуть після поворотної точки.

Saga реалізується через два основних підходи: Choreography та Orchestration.

Choreography передбачає, що кожен учасник транзакції самостійно обробляє події та запускає дії інших учасників без централізованого контролера. Такий підхід забезпечує хорошу масштабованість для простих сценаріїв, оскільки не вимагає додаткової логіки для координації дій. Проте, ускладнення в роботі системи зростають, коли додаються нові учасники або етапи, що ускладнює тестування та контроль залежностей (див. рис. 3.8).

Orchestration натомість використовує централізований контролер, який керує всіма етапами та повідомляє учасникам, які операції виконувати на кожному етапі. Це підходить для складних робочих процесів з багатьма учасниками, оскільки забезпечує кращу керованість і дозволяє уникати циклічних залежностей. Оркестрація спрощує логіку, але додає точку відмови через роль контролера (рис. 3.9).

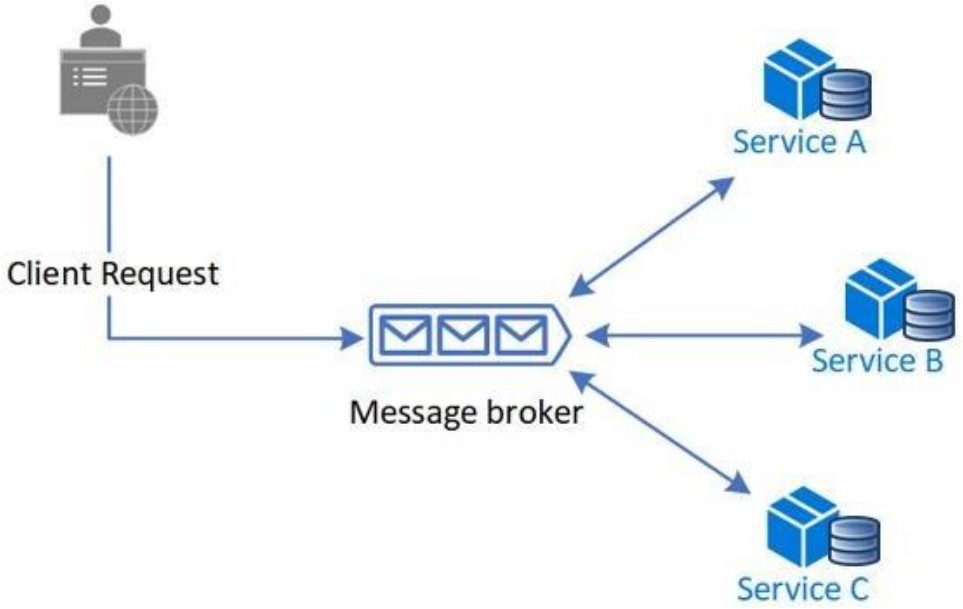


Рисунок 3.8 − Патерн Choreography

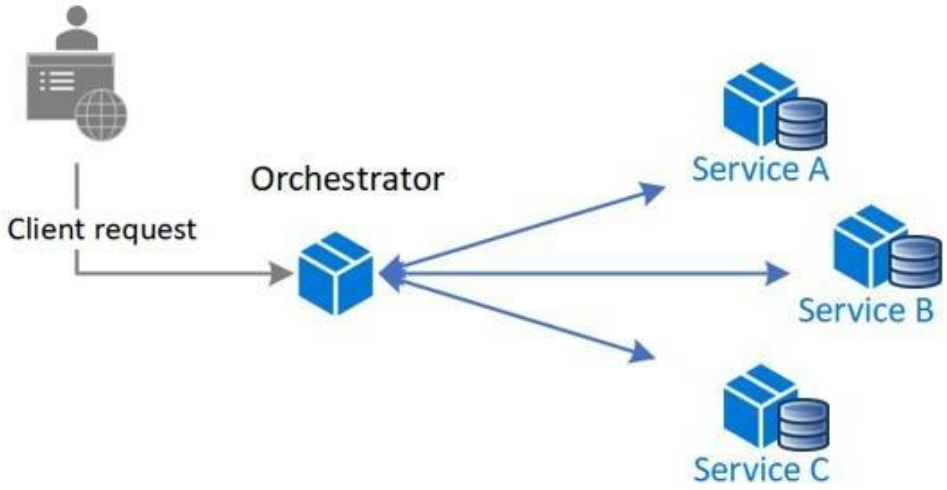


Рисунок 3.9 − Патерн Orchestration

Впровадження патерну Saga вимагає врахування певних аспектів. По-перше, відсутність ізольованості даних між учасниками може призвести до складнощів у підтримці стійкості даних. Це також ускладнює виявлення помилок, оскільки будь-яка помилка в одному сервісі може вплинути на весь процес. Щоб зменшити потенційні аномалії, такі як втрачені оновлення чи "брудні" читання, можна застосовувати механізми обмеження доступу на рівні додатків або перевірки версій записів.

Також важливо врахувати, що не всі транзакції можна компенсувати, що додає складності у випадку відмов. У випадках, коли компенсуюча транзакція не може бути виконана, необхідно розробити альтернативні механізми для

зменшення негативного впливу на систему.

Патерн Saga підходить для систем, де важливо забезпечити консистентність даних без жорсткої зв'язності між сервісами. Його можна використовувати для бізнес-процесів, що охоплюють декілька мікросервісів і вимагають відкату змін у разі відмови одного з етапів. Однак цей підхід не підходить для сценаріїв з високою щільністю зв'язків або транзакцій, які важко компенсувати.

3.5 Обґрунтування вибору підходів для оптимізації високонавантажених event-driven систем у serverless середовищі

Результати дослідження - розгляд характерних викликів безсерверних (serverless) архітектур, аналіз подієво-орієнтованої (event-driven) парадигми та застосування низки архітектурних патернів дають можливість сформулювати рекомендації щодо оптимізації високонавантажених систем. Головне грамотно вибудувати потоки подій: саме це дало змогу прибрати зайві залежності між компонентами й утримати зв’язність на низькому рівні.

Розглянуті патерни щільно переплітаються з перевагами та викликами безсерверного середовища. З одного боку, вони спираються на сильні сторони serverless - автоматичне масштабування й оплату за фактичне споживання. З іншого гасять негативні ефекти: холодні старти, складність відлагодження та моніторингу, безпекові ризики. Система лишається стабільною й витримує пікові навантаження, швидко відповідає на запити та водночас зберігає надійність і узгодженість даних у складних транзакціях.

Тож зв’язка Asynchronous Request-Reply, Priority Queue, Pipes and Filters, Saga та Choreography в event-driven архітектурі на безсерверному середовищі стає тим стрижнем, на якому тримаються високоефективні й гнучкі системи. Дослідження показує, що такий підхід відповідає не лише поточним вимогам до масштабованості та продуктивності, а й залишає простір для подальшого розвитку та оптимізації у складних хмарних сценаріях.

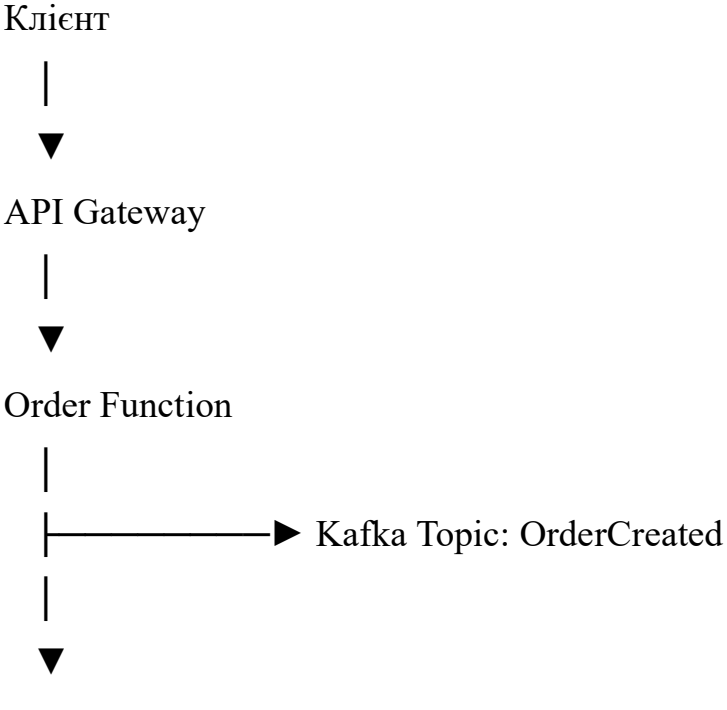
3.6 Практичний приклад розробки

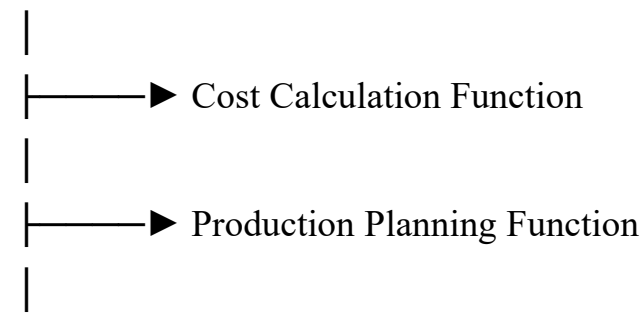
Інформаційна система обробки онлайн-замовлень поліграфічного підприємства

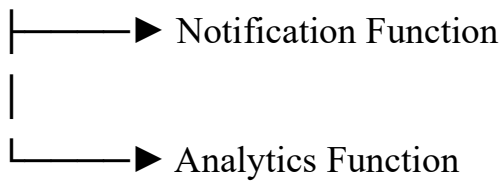
Постановка задачі Поліграфічне підприємство приймає тисячі замовлень на друк продукції через веб-портал.

Необхідно забезпечити:

- обробку великої кількості заявок;
- автоматичне масштабування;
- мінімальні затримки;
- високу відмовостійкість.







Послідовність роботи

Крок 1

Клієнт створює замовлення на друк.

Крок 2

API Gateway викликає Serverless-функцію:

```
exports.createOrder = async (event) => {

   const order = JSON.parse(event.body);



   await saveOrder(order);



   await kafka.publish(

      "OrderCreated",

      order

   );



   return {

      statusCode: 200

   };

};
```

Крок 3

Подія потрапляє до Kafka.


```
{  "orderId": 1256,
  "product": "Book",
  "quantity": 500,
  "pages": 120
}
```


Крок 4

Незалежно запускаються функції:

Розрахунок вартості

```
def calculate_cost(event):
    price = (
        event["quantity"] *
        event["pages"] * 0.05
    )
    return price
```

Планування виробництва

```
def production_plan(event):
   queue_print_job(event)
```

Відправка повідомлення

```
def notify_client(event):
   send_email(event["orderId"])
```

Математична модель продуктивності

Інтенсивність надходження подій:

λ=NT\lambda

де:

- N - кількість подій;
- T - інтервал часу.

Пропускна здатність системи:

Throughput=μ·k

де:

- μ\muμ - продуктивність однієї функції;
- k - кількість паралельно запущених функцій.

Коефіцієнт масштабування:

S=ktk0S=\frac{k_t}{k_0}S=k0kt

де:

- k0k_0k0 - початкова кількість функцій;
- ktk_tkt - кількість функцій після масштабування.

Очікувані результати

Таблиця 3.7 − Порівняння традиційної архітектури та Serverless EDA

Показник	Традиційна архітектура	Serverless EDA
Час реакції	1,8 с	0,4 с
Масштабування	ручне	автоматичне

Показник	Традиційна архітектура	Serverless EDA
Використання ресурсів	60-70 %	90-95 %
Відмовостійкість	середня	висока
Вартість обробки	висока	нижча на 25-40 %

Висновок

Проаналізовані хмарні платформи, що є основою програмного забезпечення безсерверної архітектури.

Наведено технічне забезпечення безсерверних архітектур.

Зроблено аналіз архітектурних патернів.

Обґрунтовано питання вибору підходів до оптимізаційних процесів.

Наведено результат дослідження інформаційної системи для конкретного підприємства.

Наведено математичну модель визначення продуктивності системи.

РОЗДІЛ 4

ОХОРОНА ПРАЦІ

4.1 Організаційно-правові основи забезпечення безпеки праці

Праця інженера у сфері інформаційних технологій на перший погляд видається безпечною, та насправді вона пов’язана з низкою чинників, що впливають на здоров’я й працездатність людини. Саме тому охорона праці залишається невід’ємною складовою організації будь-якого робочого процесу, зокрема й там, де проектують і супроводжують програмні системи. Її завдання - зберегти життя, здоров’я та працездатність людини під час трудової діяльності через систему правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних і лікувально-профілактичних заходів.

Для користувачів програмних і технічних засобів роль охорони праці особливо помітна. Тривала робота за комп'ютером, статичні навантаження, напруження зору та нервово-емоційне напруження поступово позначаються на самопочутті, а недотримання вимог електро- й пожежної безпеки здатне призвести й до тяжчих наслідків. Грамотно вибудувана система охорони праці дозволяє завчасно виявити ці загрози й тримати їх під контролем.

В основі державної політики у сфері охорони праці лежить пріоритет життя і здоров'я працівника над результатами виробничої діяльності. З цього принципу випливають інші: повна відповідальність роботодавця за створення безпечних умов праці, соціальний захист потерпілих, навчання та інформування працівників з питань безпеки, а також єдність вимог незалежно від форми власності підприємства. Держава встановлює ці правила, наглядає за їх дотриманням і заохочує роботодавців, що дбають про умови праці.

Правове підґрунтя охорони праці в Україні має кілька рівнів. Очолює систему Конституція України, що закріплює право кожного на належні, безпечні та здорові умови праці. Розгортає цю норму Закон України «Про охорону праці» [19] - базовий акт, який визначає засади державної політики, права й обов'язки роботодавця та працівника, порядок навчання, розслідування нещасних випадків і нагляду за дотриманням вимог безпеки. Загальні питання трудових відносин, режиму праці й відпочинку регулює Кодекс законів про працю України, що встановлює тривалість робочого часу, перерви та гарантії для працівників.

Захист людей і майна від надзвичайних ситуацій, зокрема від пожеж, унормовує Кодекс цивільного захисту України [20]; він визначає вимоги пожежної безпеки, порядок евакуації та дії в разі загрози. Окремий і важливий для ІТ-робочого місця пласт - підзаконні акти та державні норми. Вимоги до освітлення робочих місць встановлює ДБН В.2.5-28:2018 «Природне і штучне освітлення» [21], задаючи мінімальні рівні освітленості та показники якості освітлення. Гранично допустимі параметри електромагнітних полів, які створює комп'ютерна техніка, регламентує ДСанПіН 3.3.6.096-2002 [22]. Норми температури, вологості та рухливості повітря у робочій зоні визначає ДСН

3.3.6.042-99 «Санітарні норми мікроклімату виробничих приміщень» [23]. Ці документи безпосередньо застосовуються під час оцінювання умов праці розробника та обґрунтування захисних заходів.

Системний, ризик-орієнтований підхід до управління безпекою закріплює стандарт ДСТУ ISO 45001:2019 [24]: він запроваджує цикл постійного вдосконалення умов праці через виявлення, оцінювання та зниження ризиків. Саме ця ідея оцінювання ризику стала наскрізною для сучасної охорони праці й покладена в основу аналізу, проведеного в цьому розділі.

Та навіть найкраща нормативна база не спрацює без належного організаційного забезпечення. Ідеться про планування заходів безпеки, проведення інструктажів і навчання, контроль за справністю обладнання, своєчасне виявлення й усунення загроз. Тільки поєднання чітких правил з їх повсякденним дотриманням під час експлуатації техніки та програмних комплексів дає реальний результат - безпечні й комфортні умови праці.

4.2 Характеристика об’єкта та виявлення потенційних небезпек

Об’єктом проектування в цій роботі є робоче місце архітектора (розробника) програмного забезпечення, на якому ведеться проектування, розроблення та супровід високонавантажених serverless-систем подієво-орієнтованого типу. За характером праця належить до розумової, з тривалим перебуванням за персональним комп’ютером і високим навантаженням на зоровий аналізатор та нервово-емоційну сферу.

Робоче місце розташоване в офісному приміщенні. Воно обладнане персональним комп’ютером з одним-двома моніторами, клавіатурою, маніпулятором «миша», а також підключенням до локальної мережі та Інтернету. Живлення техніки здійснюється від побутової електромережі 220 В через розгалужувачі та джерела безперебійного живлення. Приміщення опалюване, з природним освітленням через вікна та штучним освітленням від світильників зі світлодіодними чи люмінесцентними лампами; передбачено припливно-витяжну вентиляцію.

Працює з об’єктом інженерно-технічний персонал - розробники, архітектори, тестувальники. Робота переважно сидяча, з повторюваними рухами

кистей і пальців, у вимушеній статичній позі впродовж робочого дня. Усе це формує специфічний набір небезпечних і шкідливих чинників, які варто розглянути за класифікацією, наведеною в Додатку 1 методичних вказівок.

Серед фізичних чинників на такому робочому місці передусім присутні недостатня або нерівномірна освітленість робочої зони, пряма та відбита блискість від екрана, пульсація світлового потоку, а також електромагнітні випромінювання від комп'ютерної техніки. Сюди ж належить підвищене значення напруги в електричній мережі, замикання якої може пройти крізь тіло людини, та підвищений рівень статичної електрики. Параметри мікроклімату й рівень шуму від систем охолодження та офісної техніки за певних умов теж виходять за межі комфортних.

Психофізіологічні чинники тут не менш вагомі. Тривала робота за монітором веде до перенапруження зору, статична поза - до навантаження на хребет і м'язи, а одноманітність операцій разом зі стислими термінами породжує монотонність і емоційне перевантаження.

Окремо варто врахувати небезпеки загального характеру. Несправність електрообладнання чи проводки здатна спричинити пожежу, а в умовах воєнного стану додається загроза обстрілів, аварійного знеструмлення та потреби тривалого перебування в укритті. Хімічні та біологічні чинники для цього об'єкта суттєвої ролі не відіграють, тож далі вони не розглядаються.

Результати виявлення потенційних небезпек на робочому місці зведено в табл. 4.1.

Таблиця 4.1 − Виявлення потенційних небезпек стосовно об'єкту проектування

№	Потенційна небезпека	Джерело небезпеки	Можливі наслідки
1	Ураження електричним струмом	Електромережа 220 В, розетки, кабелі живлення, несправна ізоляція ПК та периферії	Електротравма, опіки, у важких випадках - зупинка серця
2	Перенапруження зору та статичні навантаження	Тривала робота за монітором, вимушена сидяча поза, неправильна організація робочого місця	Зорова втома, синдром сухого ока, біль у шиї та попереку, порушення постави

№	Потенційна небезпека	Джерело небезпеки	Можливі наслідки
3	Недостатнє чи нерівномірне освітлення, блискість, пульсація	Брак природного світла, неправильне розташування світильників і монітора	Швидка стомлюваність, погіршення зору, зниження працездатності
4	Несприятливий мікроклімат і шум	Робота систем охолодження, офісної техніки, недоліки вентиляції та опалення	Головний біль, зниження концентрації, дискомфорт і втома
5	Пожежа	Перевантаження електромережі, коротке замикання, несправні ДБЖ та подовжувачі	Травми, отруєння продуктами горіння, знищення обладнання й даних
6	Воєнна загроза (обстріли, знеструмлення)	Бойові дії, ракетні та артилерійські удари, пошкодження енергоінфраструктури	Загроза життю та здоров'ю, руйнування приміщення, втрата даних, зупинка роботи

Аналіз показує, що більшість виявлених чинників піддаються зменшенню або повному усуненню за рахунок організаційних і технічних заходів - правильної організації робочого місця, дотримання норм освітлення та мікроклімату, справної електромережі й засобів пожежної безпеки. Загрози загального характеру повністю усунути не вдасться, проте їхні наслідки можна суттєво послабити завчасною підготовкою. Тому далі ці небезпеки оцінюються за рівнем ризику, після чого формуються відповідні заходи захисту.

4.3 Дослідження ризику реалізації потенційних небезпек на об'єкті проектування та розробка заходів щодо їх попередження

Оцінювання ризику - це процедура, у межах якої визначають імовірність настання небезпечної події та тяжкість її можливих наслідків, після чого роблять висновок про припустимість відповідного рівня ризику. Її мета - не просто перелічити загрози, а ранжувати їх за значущістю й зосередити ресурси там, де небезпека найвища. Така оцінка дає підставу для обґрунтованих рішень щодо запобіжних заходів і є основою системи управління охороною праці.

Для аналізу обрано метод матриці оцінювання ризиків (Додаток 2 методичних вказівок). Кожну небезпеку класифікують за двома ознаками - категорією серйозності наслідків і рівнем імовірності настання події, - після чого

за матрицею визначають підсумковий індекс ризику та його припустимість. За цими критеріями оцінено чотири найвагоміші небезпеки об’єкта: ураження електричним струмом, перенапруження зору й опорно-рухового апарату, пожежу та воєнну загрозу. Результати наведено в таблицях 4.2-4.5.

Таблиця 4.2 − Оцінка ризику небезпеки «Ураження електричним струмом»

Категорія серйозності	Рівень імовірності	Опис	Індекс ризику
II - критична	C - випадкова	Контакт зі струмовідними частинами за несправної ізоляції	2C - небажаний (гранично допустимий)

Таблиця 4.3 − Оцінка ризику небезпеки «Перенапруження зору та опорно-рухового апарату»

Категорія серйозності	Рівень імовірності	Опис	Індекс ризику
III - гранична	B - можлива	Тривала робота за монітором у статичній позі	3B - небажаний (гранично допустимий)

Таблиця 4.4 − Оцінка ризику небезпеки «Пожежа»

Категорія серйозності	Рівень імовірності	Опис	Індекс ризику
II - критична	D - віддалена	Коротке замикання, перевантаження електромережі	2D - небажаний (гранично допустимий)

Таблиця 4.5 − Оцінка ризику небезпеки «Воєнна загроза (обстріли, знеструмлення)»

Категорія серйозності	Рівень імовірності	Опис	Індекс ризику
I - катастрофічна	C - випадкова	Ракетні та артилерійські удари, пошкодження енергоінфраструктури	1C - неприпустимий (надмірний)

Результати оцінювання показали, що три з чотирьох небезпек належать до небажаного (гранично допустимого) рівня й потребують зниження ризику, а воєнна загроза з індексом 1C класифікується як неприпустима, тобто вимагає першочергових заходів. З огляду на це для об’єкта розроблено комплекс

організаційних і технічних рішень, спрямованих на зниження ризику кожної небезпеки. Запропоновані заходи та очікуваний результат зведено в таблиці 4.6.

Таблиця 4.6 − Заходи щодо зниження ризиків

№	Небезпека	Запропонований захід	Очікуваний результат
1	Ураження електричним струмом	Регулярна перевірка справності проводки та заземлення, застосування ДБЖ і пристроїв захисного вимкнення, інструктаж з електробезпеки	Зниження ймовірності електротравм, безпечна експлуатація техніки
2	Перенапруження зору та опорно-рухового апарату	Організація робочого місця за ергономічними вимогами, регламентовані перерви, гімнастика для очей, регулювання яскравості й контрасту екрана	Зменшення зорової та фізичної втоми, збереження працездатності
3	Недостатнє освітлення та несприятливий мікроклімат	Дотримання норм освітленості за ДБН В.2.5-28:2018, правильне розташування монітора, налагодження вентиляції та опалення	Комфортні умови праці, вищі концентрація та продуктивність
4	Пожежа	Первинні засоби пожежогасіння, контроль навантаження мережі, план евакуації, навчання персоналу діям при пожежі	Своєчасне виявлення й локалізація загоряння, захист людей і обладнання
5	Воєнна загроза	Облаштоване укриття, система оповіщення, резервне живлення та регулярне резервне копіювання даних, чіткий план дій при тривозі	Збереження життя та здоров’я персоналу, захист даних, швидке відновлення роботи

Запропоновані заходи дають змогу перевести небезпеки з небажаного та неприпустимого рівнів до прийнятного, а отже - створити на робочому місці розробника безпечні й комфортні умови праці.

4.4 Висновки до розділу

У розділі розглянуто питання охорони праці стосовно робочого місця архітектора програмного забезпечення, на якому проектують і супроводжують високонавантажені serverless-системи. Метою було виявити небезпеки такого робочого місця, оцінити пов’язані з ними ризики й запропонувати заходи зниження.

На основі класифікації з Додатку 1 проаналізовано робоче середовище й визначено основні небезпечні та шкідливі чинники: ураження електричним струмом, перенапруження зору та опорно-рухового апарату, недостатнє освітлення й несприятливий мікроклімат, а також небезпеки загального характеру - пожежу та воєнну загрозу. Результати зведено в таблицю виявлених небезпек.

Методом матриці оцінювання ризиків чотири ключові небезпеки класифіковано за серйозністю та ймовірністю. З'ясувалося, що ураження струмом, перенапруження зору й пожежа лежать у межах небажаного рівня, тоді як воєнна загроза з індексом 1С є неприпустимою та потребує першочергової уваги.

За підсумками оцінки розроблено комплекс організаційних і технічних заходів - від перевірки електромережі та ергономіки робочого місця до облаштування укриття, резервного живлення й копіювання даних. Їх упровадження знижує ризики до прийнятного рівня й забезпечує безпечні умови праці, тож поставлену в розділі мету досягнуто.

ВИСНОВКИ

У кваліфікаційній роботі розв'язано актуальну задачу проєктування високонавантажених serverless-систем на основі подієво-орієнтованого (event-driven) підходу, спрямовану на підвищення масштабованості, відмовостійкості та продуктивності обробки великих потоків подій. За результатами проведеного дослідження одержано такі основні результати.

Проаналізовано предметну область і простежено еволюцію архітектурних підходів - від монолітних рішень через мікросервіси до безсерверних архітектур. Показано, що подієво-орієнтована та serverless-моделі найкраще відповідають вимогам сучасних високонавантажених систем, а на прикладі відомих платформ обґрунтовано доцільність переходу до розподілених і безсерверних рішень.

Досліджено теоретичні засади подієво-орієнтованої архітектури, розмежовано поняття події та команди, проаналізовано вплив надмірного обсягу подій на зв'язність системи. На цій основі запропоновано модель зниження

зв’язності між сервісами за рахунок мінімізації обсягу подій і дотримання принципу власності на дані, а також побудовано математичну модель для оцінювання продуктивності системи за різних навантажень.

Розроблено алгоритм оптимізації маршрутизації подій та підтверджено його працездатність програмною реалізацією на основі Node.js, TypeScript і брокера повідомлень RabbitMQ. Експериментальне моделювання показало, що запропоновані рішення зменшують затримку обробки за зростання навантаження й підвищують стійкість системи до збоїв окремих компонентів.

Проаналізовано програмне та технічне забезпечення безсерверних архітектур, виконано порівняння провідних платформ FaaS за затримкою, масштабованістю та вартістю, а також обґрунтовано застосування архітектурних патернів Asynchronous Request-Reply, Priority Queue, Pipes and Filters і Saga. На практичному прикладі поліграфічного підприємства показано, як подієво-керований конвеєр обробки замовлень забезпечує асинхронність, горизонтальне масштабування та відмовостійкість.

Розглянуто питання охорони праці на робочому місці архітектора програмного забезпечення: виявлено основні небезпеки, методом матриці ризику оцінено рівень кожної з них і запропоновано організаційно-технічні заходи зниження ризиків, зокрема з урахуванням загроз воєнного часу.

Поставлену в роботі мету досягнуто, а всі визначені завдання виконано в повному обсязі. Запропоновані моделі, алгоритм і рекомендації мають практичну цінність і можуть бути використані під час проєктування високонавантажених serverless-систем подієво-орієнтованого типу.

Зроблено доповідь «Дослідження event-driven моделей проєктування високонавантажених serverless архітектур» і одержано сертифікат на III (IX) Міжнародній науково-практичній конференції здобувачів вищої освіти і молодих учених «Інформаційні технології: теорія і практика» (Харків: ХНУМГ ім. О. М. Бекетова, 2026, с. 81-85).

Надруковано тези доповіді «Індустрія 5.0 у післявоєнному розвитку України» // Матеріали XIX Всеукраїнської науково-технічної конференції

здобувачів вищої освіти «Сталий розвиток міст: поствоєнний період» (частина 2). Харків: ХНУМГ ім. О. М. Бекетова, 2026, с. 155-157.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Грищенко І. О. Дослідження event-driven моделей проєктування високонавантажених serverless архітектур // *Інформаційні технології: теорія і практика : матеріали III (IX) Міжнародної науково-практичної конференції здобувачів вищої освіти і молодих учених.* Харків : ХНУМГ ім. О. М. Бекетова, 2026. С. 81-85.

2. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.

3. Scheer A.-W. Design Software Architecture. Pearson Education, Limited, 2020. 304 p.

4. Software design: from programming to architecture. Hoboken, NJ : J. Wiley, 2004. 550 p.

5. Fowler M. Patterns of Enterprise Application Architecture. Pearson, 2012.

6. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol, CA : O'Reilly Media, 2020. 250 p.

7. Hohpe G., Woolf B. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston : Addison-Wesley, 2008. 683 p.

8. Sbarski P., Wagner T., Kiriaty Y. Serverless Design Patterns: Key Designs for Building Cloud Native Applications. Pearson Education, Limited, 2019. 352 p.

9. Serverless Computing: Principles and Paradigms. Cham : Springer International Publishing AG, 2023.

10. Designing Distributed Control Systems: A Pattern Language Approach. Chichester : John Wiley & Sons, Limited, 2014.

11. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Boston : Prentice Hall, 2008. 302 p.

12. Head First Design Patterns / ed. by E. Freeman et al. Sebastopol, CA : O'Reilly, 2004. 638 p.

13. Modi R., Lee J., Skaria R. Azure for Architects. 3rd ed. Birmingham : Packt Publishing Ltd, 2020. 700 p.

14. Get started guide for developers on Azure. URL: https://learn.microsoft.com/en-us/azure/guides/developer/azure-developer-guide (дата звернення: 15.06.2026).

15. Captain F. A. Six-Step Relational Database Design: A Step by Step Approach to Relational Database Design and Development. 2nd ed. CreateSpace Independent Publishing Platform, 2013. 232 p.

16. Ларман К. Застосування UML 2.0 та шаблонів проектування. Київ : Діалектика, 2019. 736 с.

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

18. Bass D. Advanced Serverless Architectures with Microsoft Azure: Design Complex Serverless Systems Quickly with the Scalability and Benefits of Azure. Packt Publishing, Limited, 2019.

19. Soh J. et al. Microsoft Azure. Berkeley, CA : Apress, 2020. URL: https://doi.org/10.1007/978-1-4842-5958-0 (дата звернення: 10.06.2026).

20. Paul J. J. Distributed Serverless Architectures on AWS: Design and Implement Serverless Architectures. Apress L. P., 2023.

21. CM S. Architecting Cloud Native Serverless Solutions: Design, Build, and Operate Serverless Solutions on Cloud and Open-Source Platforms. Packt Publishing, Limited, 2023.

22. Грищенко І. О. Індустрія 5.0 у післявоєнному розвитку України // *Сталий розвиток міст: поствоєнний період : матеріали XIX Всеукраїнської науково-технічної конференції здобувачів вищої освіти (частина 2*). Харків : ХНУМГ ім. О. М. Бекетова, 2026. С. 155-157.

23. Про охорону праці : Закон України. URL: https://zakon.rada.gov.ua/laws/show/2694-12 (дата звернення: 18.06.2026).

24. Кодекс цивільного захисту України. URL: https://zakon.rada.gov.ua/laws/show/5403-17 (дата звернення: 18.06.2026).

25. Природне і штучне освітлення : ДБН В.2.5-28:2018. Київ : ДП «НДІБК», 2018. 94 с.

26. Державні санітарні норми і правила при роботі з джерелами електромагнітних полів : ДСанПіН 3.3.6.096-2002. Київ : МОЗ, 2002. 24 с.

27. Санітарні норми мікроклімату виробничих приміщень : ДСН 3.3.6.042-99. Київ : МОЗ, 1999. 21 с.

28. Системи управління охороною здоров'я та безпекою праці. Вимоги та настанови щодо застосування : ДСТУ ISO 45001:2019. Київ : ДП «УкрНДНЦ», 2019. 23 с.