ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Архітектура мікросервісної системи управління інформаційними потоками для підвищення ефективності обробки даних»

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерна інженерія
освітньо-професійної програми «Комп'ютерні системи та мережі»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олександр КОЖЕМ'ЯКІН
(підпис)

Виконав: здобувач вищої освіти групи КСДМ-62
Олександр КОЖЕМ'ЯКІН

Керівник: Артем АНТОНЕНКО
к.т.н., доцент

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Магістр

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____ Наталія ЛАЩЕВСЬКА
«_____» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Кожем'якіну Олександру Вікторовичу

1. Тема кваліфікаційної роботи: «Архітектура мікросервісної системи управління інформаційними потоками для підвищення ефективності обробки даних»

керівник кваліфікаційної роботи Артем АНТОНЕНКО, к.т.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «_30__» _жовтня_____ 2025 р. № _467_____.

2. Строк подання кваліфікаційної роботи «26» грудня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, стандарти обміну даними, технологічні платформи та інструменти розроблення, методи інтеграції сервісів та управління інформаційними потоками.

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

1. Теоретичні основи мікросервісної архітектури та управління інформаційними потоками.
2. Проєктування архітектури мікросервісної системи та взаємодії сервісів.
3. Реалізація та оцінка ефективності запропонованої системи.

5. Перелік ілюстративного матеріалу: *презентація*

1. Мета, об’єкт, предмет дослідження.
2. Актуальність роботи.
3. Порівняння архітектур.
4. Мікросервісна архітектура.
5. Моделі обміну даними у мікросервісах.
6. Модель обміну повідомленнями AMQP.
7. Практичний результат.
8. USE CASE діаграма.
9. Екранні форми.
10. Результати експериментального тестування системи.
11. Висновки.

6. Дата видачі завдання «01» вересня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	16.10.2025 – 23.10.2025	Виконано
2	Вивчення принципів побудови мікросервісних систем	24.11.2025	Виконано
3	Аналіз технологій управління інформаційними потоками	25.11.2025 – 26.11.2025	Виконано
4	Проєктування архітектури мікросервісної системи	27.11.2025	Виконано
5	Реалізація компонентів системи та їх інтеграція	28.11.2025 – 29.11.2025	Виконано
6	Оцінка ефективності обробки даних у системі	28.11.2025 – 29.11.2025	Виконано
7	Оформлення роботи: вступ, висновки, реферат	23.10.2025 – 30.11.2025	Виконано
8	Розробка демонстраційних матеріалів	29.11.2025	Виконано
9	Попередній захист роботи	16.12.2025	Виконано

Здобувач вищої освіти _____ Олександр КОЖЕМ’ЯКІН
(підпис)

Керівник
кваліфікаційної роботи _____ Артем Антоненко
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 101 стор., 6 табл., 44 рис., 34 джерел.

Мета роботи – розробити архітектуру мікросервісної системи управління інформаційними потоками для підвищення ефективності обробки даних у розподіленому середовищі.

Об'єкт дослідження – процес управління інформаційними потоками в розподілених обчислювальних системах.

Предмет дослідження – методи проєктування, інтеграції та оптимізації мікросервісних систем, орієнтованих на підвищення ефективності обробки даних.

Короткий зміст роботи: У роботі розглянуто архітектурні підходи до побудови мікросервісних систем для управління інформаційними потоками. Проаналізовано сучасні технології розподіленої обробки даних і засоби взаємодії між сервісами, зокрема через черги повідомлень і REST-інтерфейси. Розроблено архітектуру системи, яка забезпечує асинхронний обмін даними між сервісами та можливість масштабування окремих компонентів. Реалізовано модель мікросервісної системи з використанням технологій .NET, RabbitMQ, PostgreSQL і Docker. Проведено експериментальну оцінку ефективності обробки даних, яка підтвердила підвищення продуктивності порівняно з монолітними рішеннями.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСНА АРХІТЕКТУРА, ІНФОРМАЦІЙНІ ПОТОКИ, УПРАВЛІННЯ ДАНИМИ, РОЗПОДІЛЕНІ СИСТЕМИ, RABBITMQ, POSTGRESQL, .NET, DOCKER, ІНТЕГРАЦІЯ СЕРВІСІВ, ЕФЕКТИВНІСТЬ ОБРОБКИ ДАНИХ

ABSTRACT

Text part of the master's qualification work: 101 pages, 44 pictures, 6 table, 34 sources.

The purpose of the work - to develop the architecture of a microservice system for managing information flows in order to improve data processing efficiency in a distributed environment.

Object of research – the process of managing information flows in distributed computing systems.

Subject of research – methods of designing, integrating, and optimizing microservice-based systems aimed at improving the efficiency of data processing.

Summary of the work: The work examines architectural approaches to building microservice systems for managing information flows. Modern technologies for distributed data processing and communication between services, including message queues and REST interfaces, are analyzed. The system architecture was developed to ensure asynchronous data exchange between services and the scalability of individual components. A model of a microservice system was implemented using .NET, RabbitMQ, PostgreSQL, and Docker technologies. Experimental evaluation confirmed an increase in data processing performance compared to monolithic solutions.

KEYWORDS: MICROSERVICE ARCHITECTURE, INFORMATION FLOWS, DATA MANAGEMENT, DISTRIBUTED SYSTEMS, RABBITMQ, POSTGRESQL, .NET, DOCKER, SERVICE INTEGRATION, DATA PROCESSING EFFICIENCY.

ЗМІСТ

ВСТУП 10
1 ТЕОРЕТИЧНІ ОСНОВИ МІКРОСЕРВІСНИХ СИСТЕМ 12
1.1 Стан та тенденції розвитку архітектур програмних систем 12
1.2 Монолітна, сервісно-орієнтована та мікросервісна архітектура: порівняльний аналіз 16
1.2.1 Монолітна архітектура 16
1.2.2 Сервісно-орієнтована архітектура (SOA) 18
1.2.3 Мікросервісна архітектура 19
1.2.4 Порівняльний аналіз архітектур 20
1.3 Основні принципи та концептуальні властивості мікросервісних систем 21
1.3.1 Принцип незалежності та автономності сервісів 22
1.3.2 Принцип ізольованості даних 23
1.3.3 Принцип слабкого зв’язування 24
1.3.4 Принцип незалежного розгортання 24
1.3.5 Принцип орієнтації на бізнес-домен (DDD) 25
1.3.6 Патерни мікросервісної архітектури 26
1.3.7 Принцип спостережуваності та автономного відновлення 26
1.4 Моделі комунікації та взаємодії між мікросервісами 27
1.5 Технологічна база для побудови мікросервісних систем 31
1.6 Переваги, недоліки та сфери застосування мікросервісної архітектури 35
2 МОДЕЛЮВАННЯ ТА УПРАВЛІННЯ ІНФОРМАЦІЙНИМИ ПОТОКАМИ 38
2.1 Поняття та класифікація інформаційних потоків у розподілених системах 38
2.2 Моделі опису та організації інформаційних потоків у мікросервісних архітектурах 41
2.3 Методи забезпечення продуктивності та узгодженості інформаційних потоків 47
2.4 Управління потоками подій у системах реального часу 51
3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ 55
3.1 Аналіз вимог та формування архітектурної моделі системи 55
3.2 Проєктування структури мікросервісів та їх взаємодії 58

3.3 Реалізація основних компонентів мікросервісної системи 61
3.4 Розгортання та інфраструктурна підтримка системи 66
4 ОЦІНКА ЕФЕКТИВНОСТІ ТА АНАЛІЗ РЕЗУЛЬТАТІВ 71
4.1 Методика оцінювання ефетивності системи 71
4.2 Результати вимірювання затримки обробки подій 73
4.3 Аналіз пропускної здатності системи під навантаженням 75
4.4 Вплив горизонтального масштабування 78
4.5 Узагальнення результатів оцінювання ефективності системи 81
ВИСНОВКИ 83
ПЕРЕЛІК ПОСИЛАНЬ 85
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 89
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ 97

ВСТУП

Стрімкий розвиток інформаційних технологій призвів до значного зростання обсягів даних, які потребують швидкої, надійної та узгодженої обробки. Традиційні централізовані системи управління інформаційними потоками вже не здатні ефективно забезпечувати масштабованість, гнучкість і високу доступність, необхідні для сучасних розподілених середовищ. Такі системи потребують великих ресурсів для підтримки, ускладнюють оновлення компонентів та не дозволяють оперативно реагувати на зміни навантаження.

Перехід до мікросервісної архітектури став одним із ключових напрямів розвитку сучасних інформаційних систем. Мікросервіси дають можливість розділити складну систему на незалежні модулі, які взаємодіють між собою через стандартизовані інтерфейси. Це дозволяє підвищити надійність, забезпечити горизонтальне масштабування та пришвидшити розгортання нових функціональних можливостей без зупинки роботи всієї системи.

У контексті управління інформаційними потоками мікросервісна архітектура відкриває можливість ефективної обробки даних у реальному часі, використовуючи асинхронну взаємодію через черги повідомлень і розподілені сховища. Використання технологій, таких як RabbitMQ, PostgreSQL, .NET та Docker, дозволяє створювати системи, здатні обробляти великі обсяги інформації з мінімальними затримками та високим рівнем відмовостійкості.

Ключовим завданням сучасних інформаційних систем є забезпечення стабільного потоку даних між компонентами, їх узгодженої обробки та своєчасної доставки до кінцевих споживачів. Ефективна організація цих процесів безпосередньо впливає на швидкодію, надійність і загальну продуктивність системи. Саме тому актуальним є розроблення архітектури мікросервісної системи, що забезпечує гнучке управління інформаційними потоками й підвищує ефективність обробки даних.

Мікросервісний підхід поєднує сучасні принципи проектування програмних систем і технології розподіленої обробки, забезпечуючи стійкість, масштабованість і простоту підтримки. Його застосування у сфері управління інформаційними потоками створює основу для побудови продуктивних корпоративних, хмарних та аналітичних рішень, що відповідають потребам цифрової трансформації.

1 ТЕОРЕТИЧНІ ОСНОВИ МІКРОСЕРВІСНИХ СИСТЕМ

1.1 Стан та тенденції розвитку архітектур програмних систем

Розвиток архітектур програмних систем є ключовим чинником еволюції сучасних інформаційних технологій. Зі зростанням складності бізнес-процесів, появою нових вимог до обробки даних та необхідністю інтеграції великої кількості сервісів архітектурні підходи зазнавали суттєвих трансформацій. Архітектура визначає структурну організацію системи, спосіб взаємодії її компонентів, механізми управління інформаційними потоками та рівень адаптивності до зовнішніх змін. Саме тому дослідження тенденцій розвитку архітектурних рішень є фундаментальним аспектом сучасної комп’ютерної інженерії.

Першим широкопоширеним типом програмних систем були централізовані архітектури, у яких обчислення та обробка інформації здійснювалися на одному сервері. Такий підхід був технологічно обґрунтованим у часи обмежених обчислювальних можливостей та низького рівня розвитку мережевої інфраструктури. Централізовані системи характеризувалися простотою, але їхні обмеження ставали очевидними із зростанням кількості користувачів та обсягів даних. Неможливість гнучкого розширення ресурсів та ризики єдиної точки відмови значно звужували сфери застосування подібних рішень.

Подальший розвиток мережевих технологій сприяв появі клієнт-серверної моделі, що передбачала поділ навантаження між клієнтами та сервером. Цей підхід дав змогу розподілити функції, зменшити кількість ресурсів, необхідних центральному вузлу, та забезпечити кращу масштабованість. Однак із часом клієнт-серверні системи (див. рис. 1.1) почали демонструвати обмеження в умовах високого навантаження, адже сервер залишався ключовим обчислювальним центром, а всі клієнти залежали від його продуктивності.

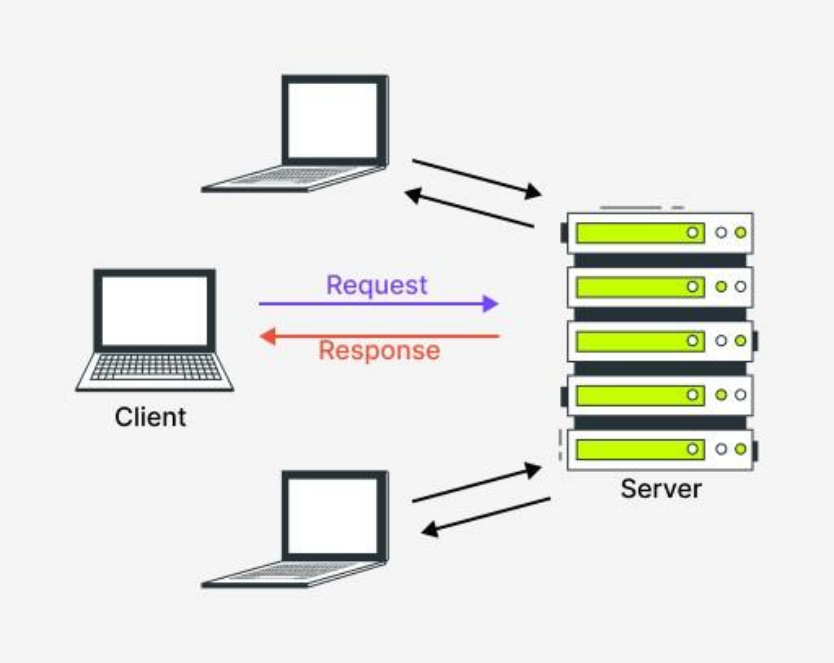


Рисунок 1.1 Клієнт-серверна система

Наступним етапом розвитку стала поява розподілених обчислювальних систем, у яких обробка інформації виконувалася на багатьох вузлах одночасно. Це дало змогу одночасно виконувати багато обчислень, підвищити відмовостійкість та оптимізувати використання ресурсів. Розподілені системи стали основою для формування більш складних архітектур, включно з об’єктно-орієнтованими, компонентними та сервісними. Проте висока складність координації та необхідність вирішення питань узгодженості даних вимагали нових підходів до проєктування програмних систем.

Суттєвим еволюційним кроком стала сервісно-орієнтована архітектура (SOA), що передбачала поділ системи на окремі сервіси зі стандартизованою взаємодією через сервісну шину. SOA забезпечила повторне використання функціональних компонентів, кращу адаптивність та можливість інтеграції різнорідних систем. Проте застосування громіздких протоколів, залежність від централізованої шини обміну та складність управління сервісами стали значними недоліками на шляху масштабування архітектури.

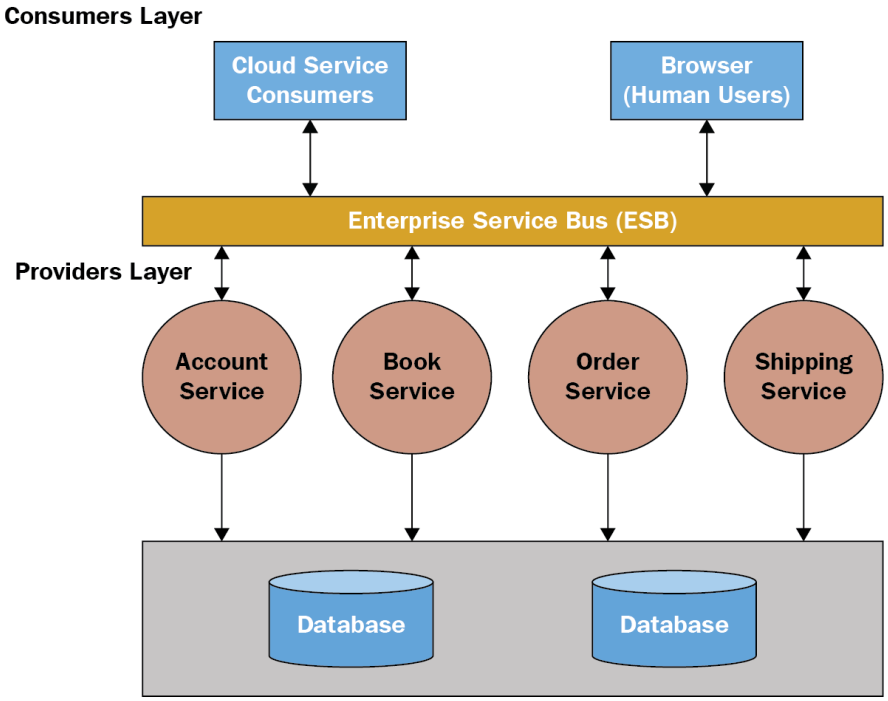


Рисунок 1.2 Сервісно-орієнтована (SOA) архітектура

Подальший розвиток хмарних технологій та необхідність обробки великої кількості запитів у реальному часі стали передумовами появи мікросервісної архітектури. Мікросервісний підхід передбачає поділ програмного комплексу на незалежні модулі, що відповідають за окремі функції та можуть розгортатися автономно. Кожен мікросервіс має власну логіку, сховище даних та засоби взаємодії, що забезпечує гнучкість, масштабованість та підвищену відмовостійкість системи.

Сучасні тенденції у сфері програмної інженерії пов’язані з оптимізацією взаємодії між компонентами системи. Широке застосування асинхронних протоколів обміну повідомленнями, таких як AMQP (Advanced Message Queuing Protocol), сприяло розвитку подійно-орієнтованих архітектур, у яких ключовим механізмом є обробка подій, а не послідовність викликів. Подібні підходи дозволяють значно покращити продуктивність і забезпечити стійкість системи до пікових навантажень.

Інший важливий напрям розвитку архітектур пов’язаний із контейнеризацією (див. рис. 1.3) та оркестрацією. Технології на кшталт Docker та Kubernetes дали

змогу стандартизувати середовище виконання сервісів, спростити розгортання та підвищити ефективність управління інфраструктурою. Вони стали невід'ємною частиною мікросервісних архітектур і визначили новий рівень автоматизації у сфері DevOps.

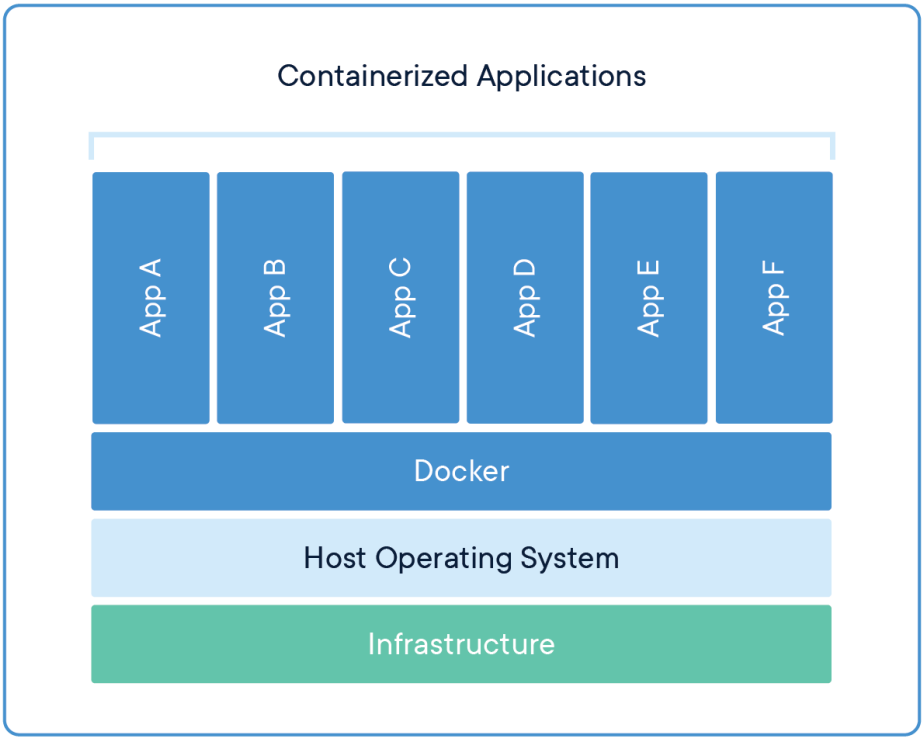


Рисунок 1.3 Контейнерезовані застосунки у Docker контейнерах

Паралельно з цим зростає значення систем моніторингу, логування та трасування. Оскільки мікросервісні системи складаються з великої кількості компонентів, забезпечення спостережуваності стає критичним завданням. Сучасні платформи надають можливість відстежувати взаємодію між сервісами, оцінювати стан мережевих з'єднань, затримки та споживання ресурсів. Це дає змогу підвищувати доступність системи та оперативно виявляти аномалії у роботі.

Розвиток архітектур програмних систем відбувається також під впливом вимог до безпеки. Збільшення кількості сервісів, відкритих API та обробки конфіденційних даних формують нові виклики в галузі аутентифікації, авторизації та захисту каналів зв'язку. Мікросервісні архітектури передбачають більш складні

механізми управління доступом, зокрема, в рамках підходів Zero Trust та Role-Based Access Control.

1.2 Монолітна, сервісно-орієнтована та мікросервісна архітектура: порівняльний аналіз

Архітектура програмних систем визначає спосіб організації їхніх компонентів, модель взаємодії між ними та рівень гнучкості під час змін у функціональності або навантаженні. Протягом останніх десятиліть сформувалися три основні архітектурні підходи — монолітна, сервісно-орієнтована та мікросервісна архітектури. Кожна з них має власні особливості, переваги та обмеження, що впливають на доцільність використання в різних середовищах і задачах.

Порівняння цих архітектур є важливим для визначення оптимального підходу до управління інформаційними потоками, оскільки ефективність обробки даних залежить від структури взаємодії між компонентами системи та можливості їхнього масштабування.

1.2.1 Монолітна архітектура

Монолітна архітектура була домінуючою на ранніх етапах розвитку програмних систем. Вона передбачає об’єднання всієї функціональності у єдиний виконуваний модуль. У межах такого підходу бізнес-логіка, інтерфейс користувача, обробка запитів, доступ до даних та допоміжні функції інтегровані в одну систему.

Найбільшою перевагою моноліту є простота. Розробники мають справу з єдиною кодовою базою, що полегшує процеси розробки, тестування та розгортання на початкових етапах. Проте зі збільшенням обсягів системи моноліт стає складним

у підтримці. Невеликі зміни в одному модулі можуть вимагати повторної збірки та розгортання всієї системи, що ускладнює оновлення та знижує гнучкість.

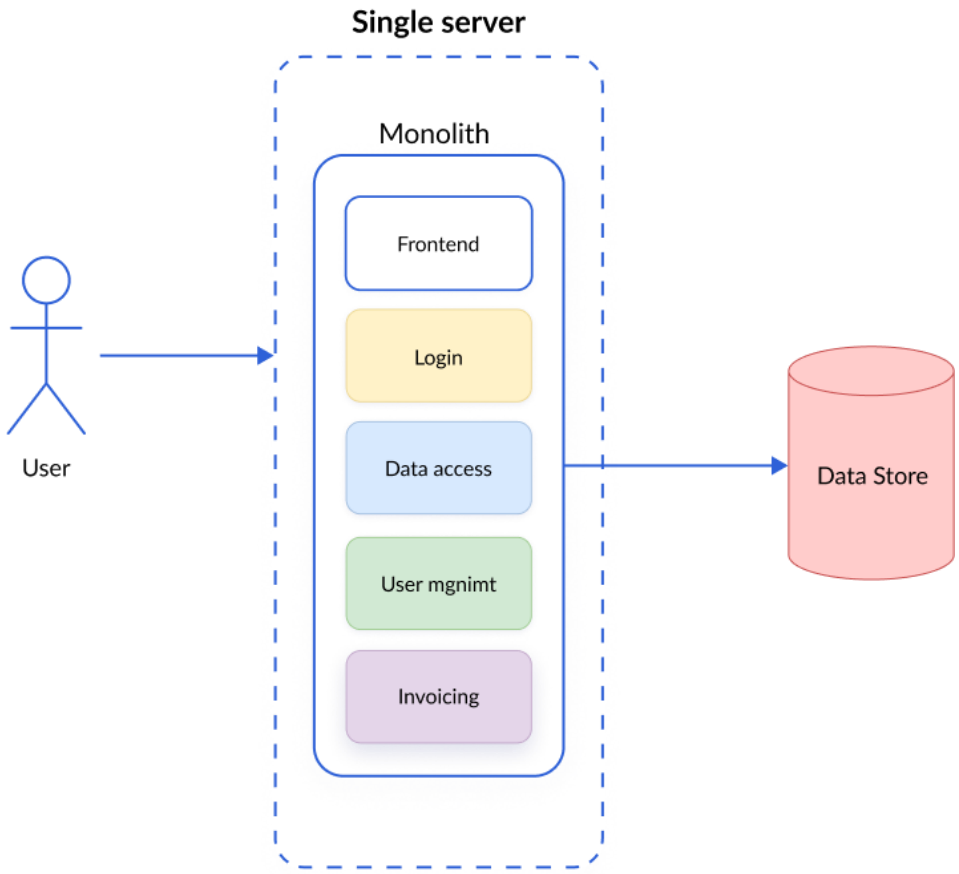


Рисунок 1.4 Монолітна архітектура

Суттєвим недоліком монолітних систем є обмежена масштабованість. Оскільки компоненти тісно пов’язані між собою, масштабування певної функціональної частини здійснюється лише через збільшення ресурсів усього застосунку. Це призводить до неефективного використання обчислювальних потужностей, особливо у системах із нерівномірним навантаженням на різні компоненти.

Монолітні системи також мають проблему залежності від однієї точки відмови. Пошкодження або перевантаження будь-якого компонента може призвести до зупинки роботи всієї системи. У середовищах, що потребують високої доступності, такі ризики є критичними.

1.2.2 Сервісно-орієнтована архітектура (SOA)

Сервісно-орієнтована архітектура стала реакцією на обмеження монолітів і передбачає поділ системи на автономні сервіси, які взаємодіють через стандартизовані протоколи. Основним елементом SOA є сервісна шина (ESB), що забезпечує маршрутизацію повідомлень, перетворення даних та управління потоками.

SOA дозволяє створювати масштабовані системи, у яких кожен сервіс відповідає за окрему функцію та може розгортатися незалежно від інших. Це підвищує гнучкість та покращує можливості інтеграції з зовнішніми системами.

Водночас SOA має низку обмежень. Залежність від централізованої шини (див. рис. 1.5) створює потенційне вузьке місце та збільшує затримки під час передачі повідомлень. Розроблення та підтримка сервісної шини вимагають значних ресурсів, що збільшує загальну складність архітектури. Крім того, використання важковагових протоколів, таких як SOAP та WSDL, ускладнює реалізацію та підвищує вимоги до інфраструктури.

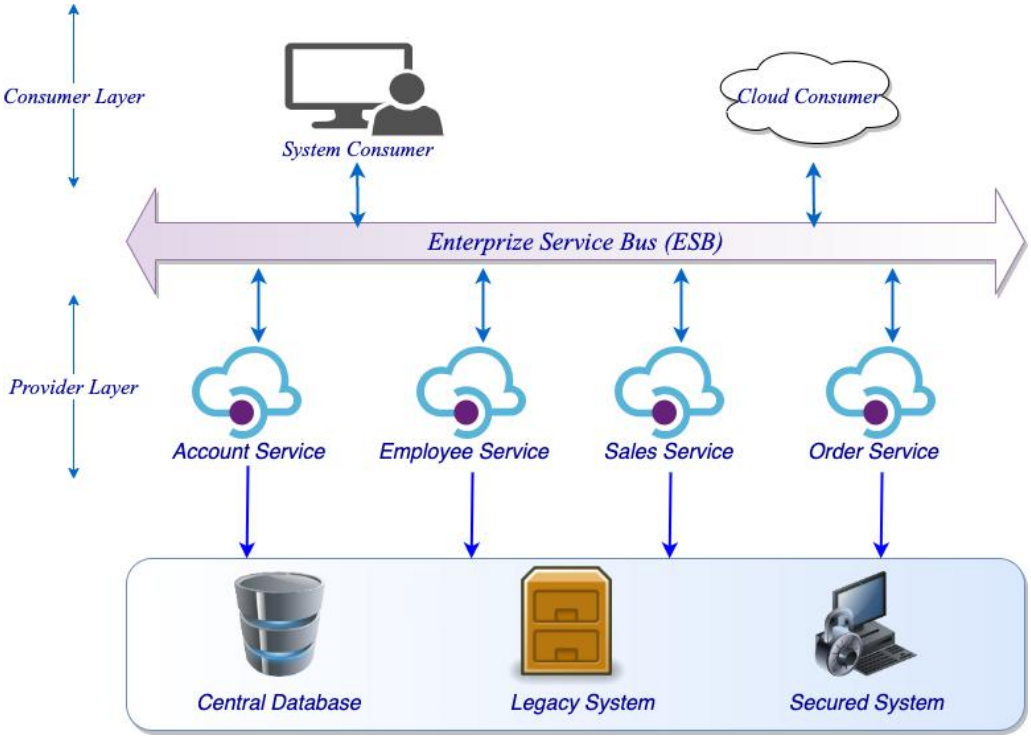


Рисунок 1.5 Централізована шина в SOA

SOA суттєво покращила модульність систем, але її структури залишалися залежними від централізованих елементів. Це обмежувало гнучкість, особливо у високонавантажених середовищах, та сприяло появі нових підходів, зокрема мікросервісної архітектури.

1.2.3 Мікросервісна архітектура

SOA суттєво покращила модульність систем, але її структури залишалися залежними від централізованих елементів. Це обмежувало гнучкість, особливо у високонавантажених середовищах, та сприяло появі нових підходів, зокрема мікросервісної архітектури.

Ключовою особливістю мікросервісів є децентралізована модель управління. На відміну від SOA, мікросервісні системи не залежать від централізованої шини. Взаємодія між сервісами відбувається через REST, gRPC або брокери повідомлень, такі як RabbitMQ чи Apache Kafka. Це дозволяє оптимізувати інформаційні потоки та зменшити затримки.

Мікросервіси забезпечують можливість гнучкого горизонтального масштабування. Кожен сервіс можна масштабувати відповідно до його навантаження, що значно підвищує ефективність використання ресурсів. Наприклад, сервіс обробки подій може потребувати більших потужностей порівняно із сервісом авторизації, і мікросервісна архітектура дає можливість масштабувати їх окремо.

Ще однією перевагою є висока відмовостійкість. Збій одного мікросервісу не призводить до зупинки всієї системи, якщо функціональність спроєктована з урахуванням принципів ізоляції та компенсаційних операцій. У поєднанні з контейнеризацією та інструментами оркестрації, такими як Kubernetes, мікросервіси створюють гнучку платформу для побудови незалежних компонентів, що автоматично масштабуються та відновлюються.

Водночас мікросервісна архітектура має і певні недоліки. Вона ускладнює процес тестування, оскільки потребує координації між великою кількістю сервісів (рис. 1.6). Збільшення кількості мережевих викликів також створює додаткове навантаження на інфраструктуру. Окремою проблемою є забезпечення спостережуваності та моніторингу складних розподілених систем.

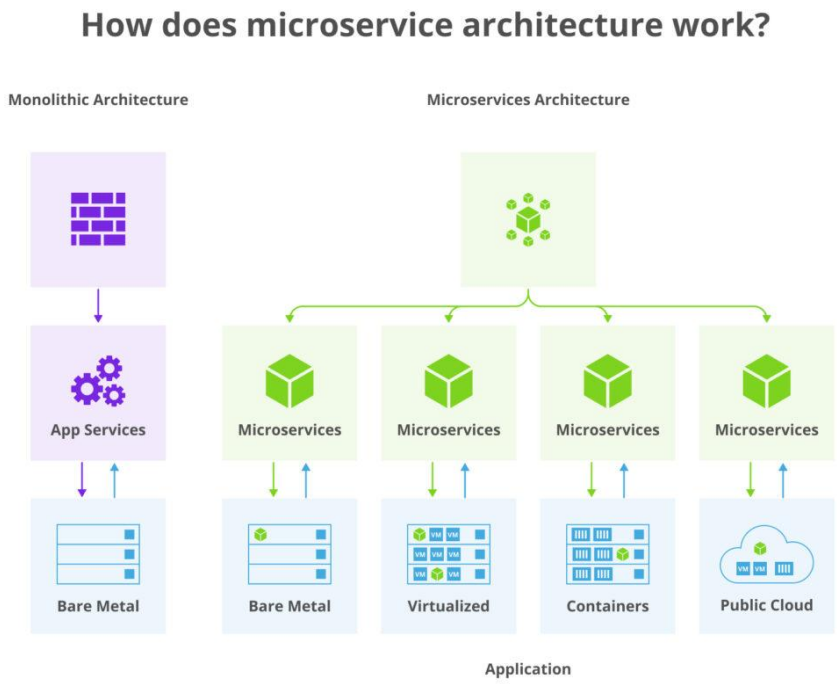


Рисунок 1.6 Графічне порівняння координації в монолітній та мікросервісній архітектурі

1.2.4 Порівняльний аналіз архітектур

Таблиця 1.1

Загальне порівняння архітектур

Характеристика	Моноліт	SOA	Мікросервіси
Масштабованість	Низька	Середня	Висока
Модульність	Обмежена	Середня	Висока

Продовження таблиці 1.1

Загальне порівняння архітектур

Характеристика	Моноліт	SOA	Мікросервіси
Складність	Низька	Висока	Висока
Взаємодія	Внутрішні виклики	Загальна шина	API/брокери
Відмовостійкість	Низька	Середня	Висока
Гнучкість розгортання	Обмежена	Обмежена	Висока
Продуктивність	Стабільна	Залежить від шини	Висока при правильному дизайні

Проведений порівняльний аналіз демонструє, що мікросервісна архітектура найбільш ефективно відповідає вимогам систем, орієнтованих на динамічні інформаційні потоки. Вона забезпечує гнучкість, високу пропускну здатність та можливість адаптації до змін у навантаженні, що робить її оптимальним рішенням для сучасних розподілених середовищ.

1.3 Основні принципи та концептуальні властивості мікросервісних систем

Мікросервісна архітектура сформувалася як відповідь на обмеження монолітних та сервісно-орієнтованих систем у контексті масштабованості, гнучкості та управління інформаційними потоками. Мікросервіси демонструють нову модель організації програмних комплексів, у якій кожен функціональний компонент є незалежним модулем з власною логікою, даними та можливістю автономного розвитку. Завдяки цьому підходу сучасні системи здатні ефективно адаптуватися до змін у навантаженні та структурі бізнес-процесів.

Основні принципи мікросервісної архітектури визначають спосіб проєктування, розгортання та взаємодії компонентів у розподіленому середовищі. Дотримання цих принципів дає змогу забезпечити високу продуктивність, стабільність і надійність роботи системи, що особливо важливо в умовах обробки значних обсягів інформаційних потоків.

1.3.1 Принцип незалежності та автономності сервісів

Ключовою характеристикою мікросервісної архітектури є незалежність кожного сервісу від інших компонентів системи. Мікросервіс розробляється, тестується та розгортається автономно. Такий підхід зменшує взаємозалежність між функціональними частинами та дозволяє уникати глобальних збоїв у випадку виходу з ладу окремого компонента.

Автономність передбачає наявність власної бізнес-логіки, інтерфейсу взаємодії та доступу до даних. Це усуває необхідність спільних баз даних, що знижує ризик конфліктів у роботі сервісів. Унаслідок цього кожен мікросервіс можна масштабувати незалежно відповідно до його навантаження.

Окремим аспектом автономності є можливість використання різних технологій у межах однієї системи. Мікросервіси можуть бути реалізовані на різних мовах програмування та використовувати різні СУБД, якщо це відповідає їхнім функціональним потребам .

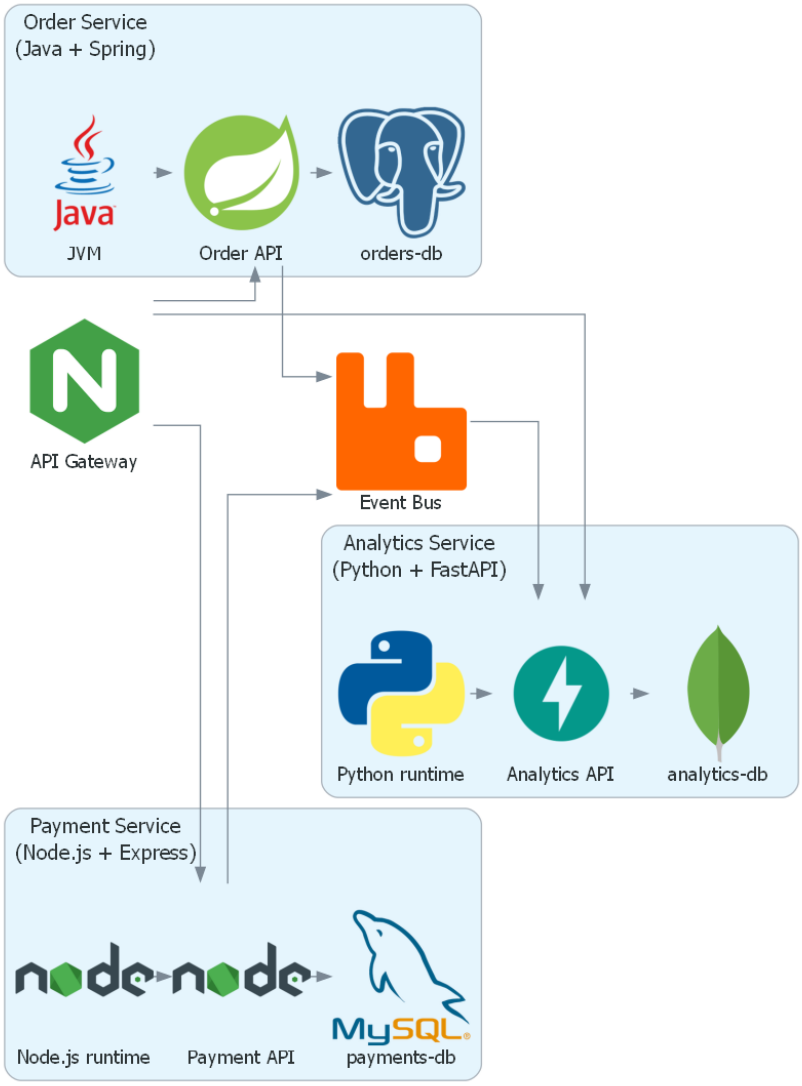


Рисунок 1.7 Незалежні сервіси з різними технологіями та СУБД

1.3.2 Принцип ізольованості даних

Мікросервісний підхід передбачає модель *“база даних для кожного сервісу”*. Це означає, що алгоритми доступу та структура даних визначаються лише цим сервісом, а зміни не впливають на інші компоненти. Заборона спільного сховища сприяє зниженню залежностей та підвищує стабільність системи.

У межах такої моделі з’являється необхідність у забезпеченні узгодженості даних між сервісами. Узгодженість досягається або через механізм eventual

consistency, або через обмін подіями в асинхронних архітектурах. Це дозволяє системі залишатися продуктивною навіть за умов високого навантаження.

1.3.3 Принцип слабкого зв’язування

Слабке зв’язування визначає ступінь взаємної залежності між сервісами. У мікросервісній архітектурі сервіси мають мінімально залежати від внутрішньої структури один одного та взаємодіяти лише через формалізовані API або події. Завдяки цьому зміни в одному сервісі не потребують переробки інших компонентів.

Слабке зв’язування сприяє спрощенню оновлення системи. Оскільки API виступає єдиним інтерфейсом взаємодії, зміни у внутрішній реалізації сервісу не впливають на клієнтів. Це створює передумови для швидкого впровадження нових функцій.

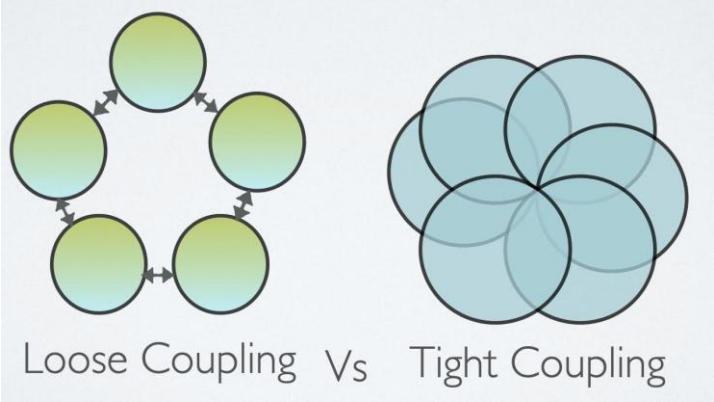


Рисунок 1.8 Слабке та сильне зв'язування

1.3.4 Принцип незалежного розгортання

Однією з основних переваг мікросервісів є їхня здатність до незалежного розгортання. Сервіс може бути оновлений або переміщений без необхідності підтримувати зворотну сумісність чи перезапускати систему цілком. Такий підхід

полегшує процеси DevOps, знижує ризики та дозволяє швидко тестувати нові функції у production-середовищі.

Незалежне розгортання забезпечуються за допомогою контейнеризації. Технології Docker і Kubernetes дають можливість розгортати сервіси у стандартизованих середовищах, автоматично масштабувати компоненти та керувати їхнім життєвим циклом.

1.3.5 Принцип орієнтації на бізнес-домен (DDD)

Domain-Driven Design (DDD) (рис. 1.9) відіграє важливу роль у проєктуванні мікросервісів. Застосування DDD передбачає поділ доменної області на піддомени та визначення bounded contexts, у межах яких мікросервіс виконує свою функцію. Такий підхід дозволяє зменшити складність системи, забезпечуючи логічну та структурну завершеність сервісів.

DDD допомагає формувати сервіс так, щоб він відповідав одному чітко визначеному бізнес-процесу. Це запобігає надмірному розростанню функціональності та сприяє досягненню високого рівня модульності.

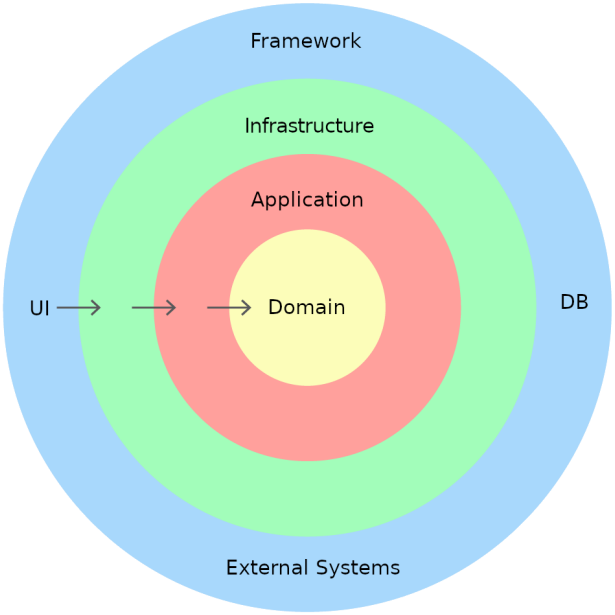


Рисунок 1.9 Domain Driven Design (DDD)

1.3.6 Патерни мікросервісної архітектури

Розроблення мікросервісів передбачає використання спеціальних архітектурних патернів, що допомагають вирішувати типові задачі у розподіленому середовищі.

API Gateway виступає точкою входу для клієнтів. Він забезпечує маршрутизацію, авторизацію, агрегацію даних і приховує внутрішню структуру системи

Service Registry та Service Discovery дозволяють сервісам автоматично знаходити один одного у середовищі з динамічними IP-адресами. Ці компоненти є критичними для систем з автоматичним масштабуванням.

Circuit Breaker запобігає каскадним збоям, ізолюючи сервіси, що не відповідають. Це дає змогу забезпечити стабільну роботу у випадку часткових збоїв.

Bulkhead обмежує ресурси, доступні кожному сервісу, що запобігає ситуаціям, коли один компонент споживає всю доступну інфраструктуру.

Event Sourcing та CQRS застосовуються для оптимізації роботи з даними та забезпечення узгодженості у розподілених середовищах.

1.3.7 Принцип спостережуваності та автономного відновлення

У мікросервісних системах важливо забезпечити механізми спостережуваності — логування, трасування та моніторинг. Сучасні платформи дозволяють отримувати інформацію про стан кожного сервісу, затримки, взаємодію та навантаження.

Автономне відновлення сервісів здійснюється за допомогою оркестраторів, які здатні перезапускати компоненти у разі збоїв. Це значно підвищує безвідмовність системи.

1.4 Моделі комунікації та взаємодії між мікросервісами

Взаємодія між мікросервісами формує основу функціонування розподілених систем, у яких кожен сервіс є автономною одиницею, що виконує окремий набір функцій. Спосіб організації комунікації визначає швидкість обробки даних, стійкість до збоїв, пропускну здатність та загальну архітектурну надійність системи. Мікросервісна архітектура допускає використання різних моделей взаємодії, однак найбільш поширеними є синхронна та асинхронна комунікації. Вони вирізняються механізмами обміну інформацією, впливом на затримки у системі та рівнем зв’язаності сервісів.

Синхронна взаємодія. Передбачає отримання відповіді під час виклику віддаленого сервісу. Найчастіше вона реалізується за допомогою REST або gRPC. REST базується на стандартному HTTP-протоколі (рис. 1.10) й забезпечує доступ до ресурсів через уніфіковані методи. Простота реалізації та інтуїтивність структури роблять REST зручним для інтеграції зовнішніх систем. Однак при високому навантаженні синхронні виклики створюють значні затримки, що знижує загальну продуктивність. gRPC є більш продуктивною альтернативою, оскільки використовує бінарний формат Protobuf і можливості HTTP/2. Цей підхід мінімізує затримки, проте потребує складнішої інфраструктури та суворішої типізації.

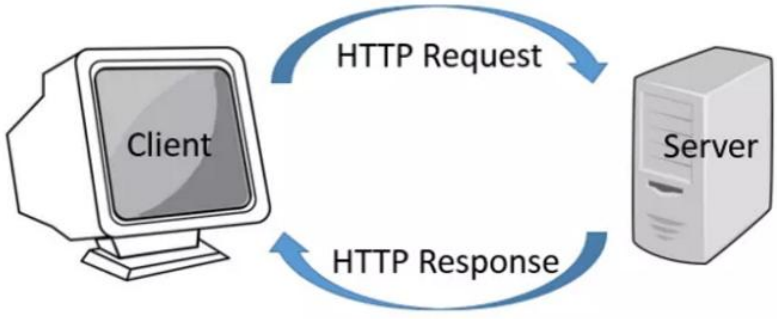


Рисунок 1.10 HTTP протокол

Проблемою синхронної моделі є висока взаємозалежність сервісів, оскільки збій або затримка одного компонента блокує роботу інших. У складних

архітектурах це може спричинити ефект каскадних відмов, який унеможливлює стабільну роботу при нерівномірних потоках даних. Тому синхронна комунікація застосовується лише там, де потрібна негайна відповідь, наприклад під час перевірки прав доступу або виконання критичних транзакцій.

Асинхронна взаємодія. Базується на передачі повідомлень без очікування негайної відповіді. У цьому випадку відправник продовжує виконання власної логіки, а отримувач обробляє дані незалежно від нього. Такий підхід зменшує навантаження на систему, забезпечує стійкість до пікових значень потоків і усуває жорсткі залежності між компонентами. Найпоширенішою технологією реалізації асинхронного обміну є RabbitMQ, який працює на базі протоколу AMQP (рис. 1.11). RabbitMQ забезпечує гнучку маршрутизацію повідомлень, надійну доставку та можливість повторної обробки у разі помилки. Його використання дозволяє будувати системи, здатні витримувати нерівномірні навантаження й обробляти значні обсяги інформаційних потоків.

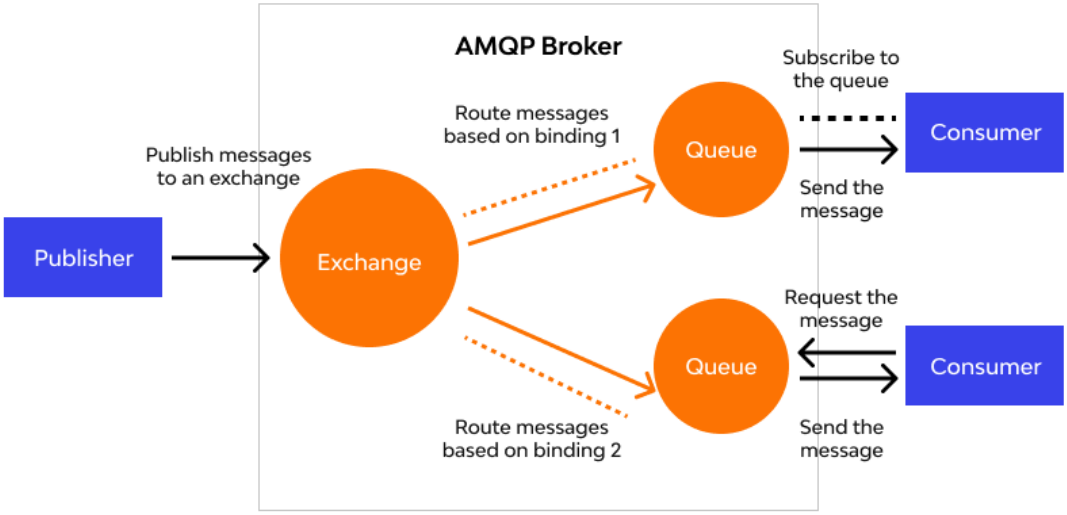


Рисунок 1.11 AMQP протокол

Іншою технологією асинхронного обміну є Apache Kafka (рис. 1.12), що орієнтована на потокове передавання даних. Kafka підходить для роботи з великими масивами повідомлень, які потребують збереження, аналізу й повторної обробки. Її лог-файлова структура дозволяє зберігати повідомлення протягом

тривалого часу та масштабувати систему горизонтально. У результаті Kafka широко застосовується у телеметрії, аналітичних платформах і високошвидкісних інформаційних системах.

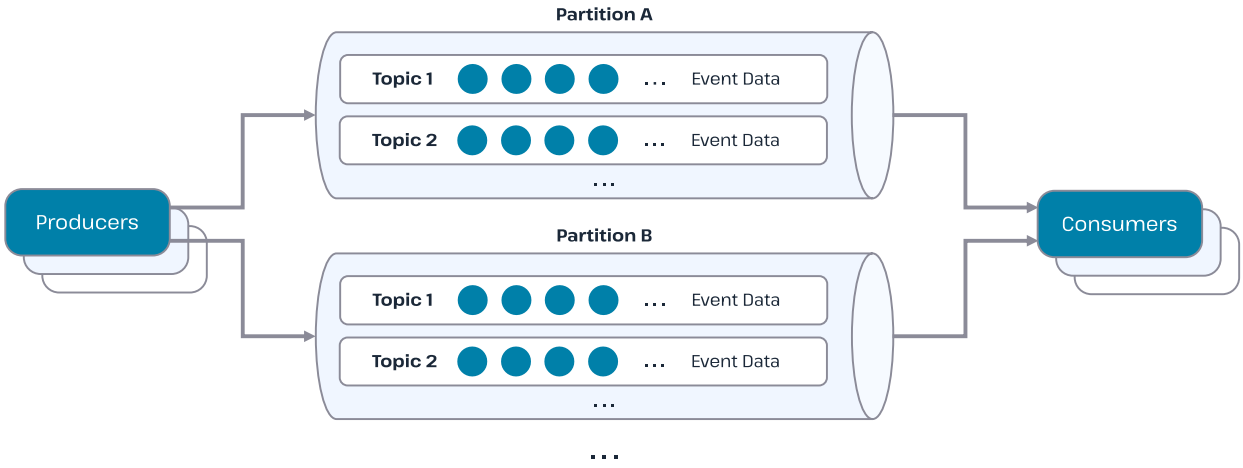


Рисунок 1.12 Схема роботи Kafka

Асинхронні моделі створюють потребу у забезпеченні коректності даних між сервісами, оскільки класичні транзакції у розподілених середовищах неефективні. Для підтримання узгодженості застосовують патерн Saga, що передбачає виконання набору локальних операцій у різних сервісах із можливістю їх компенсації у разі помилки. Ще одним важливим механізмом є підхід Outbox, за якого події спочатку зберігаються у локальній таблиці, а потім надсилаються до брокера повідомлень, що гарантує їхню доставку. Для обробки повторних повідомлень застосовують ідемпотентні операції, що дозволяють уникнути дублювання результатів.

Значне поширення отримала модель публікації-підписки (рис 1.13), у якій відправник генерує подію, а сервіси, що підписані на певну тему, отримують її автоматично. Такий підхід дає змогу легко розширювати систему, додаючи нові сервіси без змін у вже створених компонентах. Публікація-підписка є гнучким механізмом, придатним для систем, що потребують реакції на значну кількість подій у реальному часі.

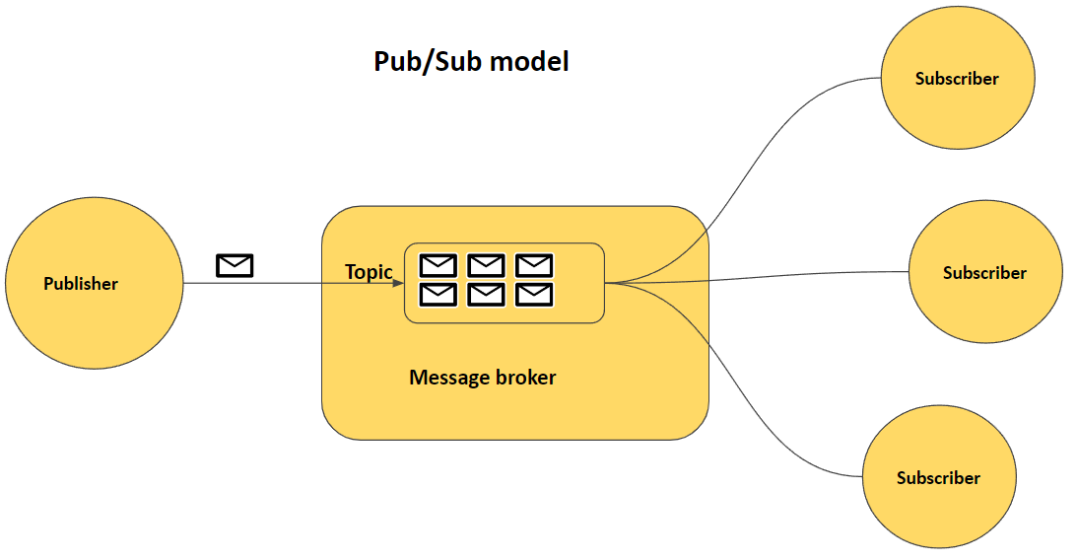


Рисунок 1.13 Модель публікації-підписки (Pub/Sub)

В умовах зростання обсягів даних і потреби у високій масштабованості все частіше застосовується гібридна модель комунікації, що поєднує синхронні та асинхронні механізми. Синхронна взаємодія використовується для операцій, що потребують чіткої послідовності виконання або негайного зворотного зв'язку. Асинхронна комунікація, своєю чергою, відповідає за обробку великих обсягів даних, де швидкодія та стабільність є важливішими за негайну відповідь.

Ефективність роботи мікросервісної системи значною мірою визначається правильним вибором моделі комунікації. Синхронні механізми забезпечують прямий і передбачуваний обмін даними, але збільшують навантаження і залежність компонентів. Асинхронні підходи дають можливість побудувати масштабовану і стійку до збоїв систему, здатну ефективно обробляти великі інформаційні потоки, але потребують складніших механізмів синхронізації станів. Вибір оптимальної моделі залежить від характеру задачі, способу використання даних та необхідного рівня продуктивності системи.

1.5 Технологічна база для побудови мікросервісних систем

Технологічна основа мікросервісної архітектури визначає можливості системи щодо масштабованості, надійності та ефективної взаємодії між сервісами. Розвиток цього підходу став можливим завдяки появі інструментів, що забезпечують стандартизоване середовище виконання, автоматизацію життєвого циклу сервісів та гнучкі механізми обміну даними. Сучасні технології формують комплексну інфраструктуру, у якій кожен компонент виконує чітко визначену роль, забезпечуючи стабільність та адаптивність системи у розподіленому середовищі.

Контейнеризація стала ключовою передумовою розвитку мікросервісних систем, оскільки дала змогу ізолювати кожен сервіс у власному середовищі виконання. Технологія Docker забезпечує створення уніфікованих контейнерів, у яких розміщуються всі залежності, конфігурації та виконувані файли сервісу. Завдяки цьому розробники отримують можливість розгортати сервіси незалежно один від одного, а системні адміністратори — працювати зі стандартизованими образами, що полегшує перенесення сервісів між середовищами. Контейнери забезпечують передбачувану поведінку сервісів, усуваючи конфлікти між версіями програмного забезпечення і роблячи систему більш стабільною під час масштабування.

Оркестрація контейнерів стала відповіддю на необхідність керування великою кількістю сервісів у динамічному середовищі. Технології на кшталт Kubernetes та Docker Swarm виконують автоматичний розподіл навантаження, відстеження стану контейнерів, перезапуск у разі збою, масштабування та управління конфігураціями. Kubernetes вирізняється розвиненою екосистемою та підтримкою складних сценаріїв розгортання, включно з autoscaling, rollout-оновленнями та конфігурацією мережевих політик. Оркестратори дозволяють забезпечити відмовостійкість мікросервісної системи, знижують ризики зупинки компонентів і підвищують ефективність роботи у хмарних середовищах.

Брокери повідомлень є центральним елементом для організації асинхронної взаємодії між мікросервісами. RabbitMQ, побудований на протоколі AMQP, забезпечує гнучкі механізми маршрутизації повідомлень, підтримує підтвердження доставки, збереження повідомлень і автоматичне повторення у разі збою. Ці властивості роблять його оптимальним рішенням для систем, у яких важливі надійна комунікація та стабільність при перериваннях у роботі окремих сервісів. У високонавантажених середовищах застосовується Apache Kafka, орієнтована на обробку потоків даних. Її архітектура побудована на лог-файловій структурі, що дає змогу зберігати історію подій, забезпечувати їх паралельну обробку та масштабувати платформу за допомогою партицій. Kafka активно використовується у системах аналітики, телеметрії та моніторингу, де обсяг інформаційних потоків може досягати мільйонів повідомлень за секунду.

API-шлюзи стали важливим інструментом керування зовнішніми та внутрішніми точками взаємодії мікросервісної системи. Вони виконують функції маршрутизації, автентифікації, авторизації, обмеження швидкості запитів та агрегації відповідей. Застосування API Gateway (рис. 1.14) дозволяє приховати внутрішню структуру мікросервісів від зовнішніх клієнтів, забезпечуючи єдину точку доступу до системи. У середовищах великої кількості клієнтів шлюзи значно знижують мережеві витрати, оскільки можуть кешувати дані та перенаправляти запити до оптимальних вузлів.

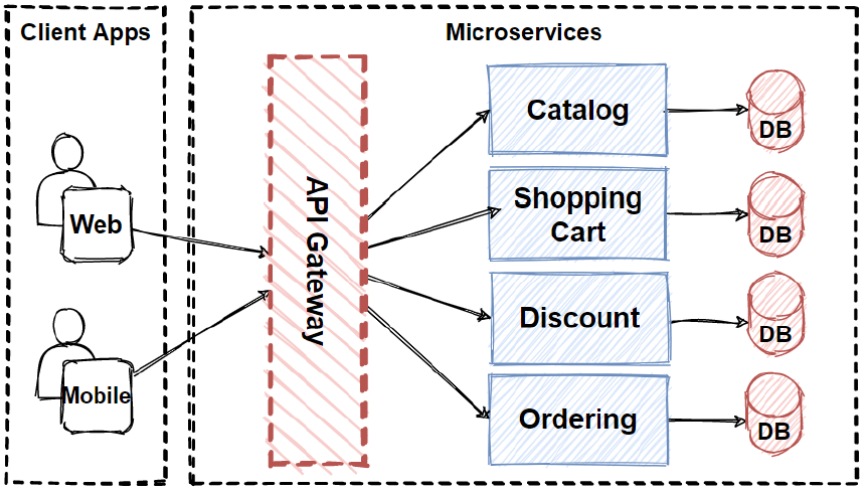


Рисунок 1.14 API Gateway

Системи моніторингу та спостережуваності стали невід'ємною частиною мікросервісної інфраструктури, забезпечуючи контроль стану системи в реальному часі. Складність розподіленого середовища вимагає аналізу логів, трасування запитів та збору метрик. Технології на кшталт Prometheus, Grafana, ELK Stack (рис. 1.15) та Jaeger дозволяють відстежувати взаємодію між сервісами, аналізувати затримки та виявляти вузькі місця. Наявність таких інструментів допомагає своєчасно реагувати на аномалії та покращувати продуктивність системи, забезпечуючи її стабільність і відповідність вимогам до якості обслуговування.

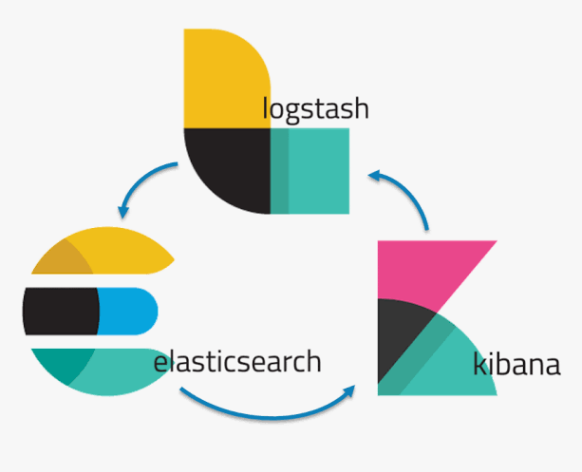


Рисунок 1.15 ELK Stack

Бази даних у мікросервісних системах відіграють важливу роль, адже кожен сервіс повинен мати власне сховище, що відповідає його функціональним потребам. У цьому контексті застосовуються різні типи баз даних, від реляційних до NoSQL-рішень, які забезпечують оптимальне зберігання даних залежно від їхньої структури й характеру використання. Модель "база даних на сервіс" дозволяє досягти мінімальної зв'язаності між компонентами системи та полегшує масштабування. Проте такий підхід створює нові виклики, пов'язані з узгодженістю даних у розподіленому середовищі, що вимагає застосування відповідних патернів та механізмів синхронізації.

Інструменти для автоматизації процесів розроблення та розгортання стали основою безперервної інтеграції та доставки у мікросервісному середовищі. Системи CI/CD забезпечують автоматичну збірку контейнерів, їх тестування та публікацію з подальшим розгортанням у кластері. Такий підхід мінімізує час виходу нових версій сервісів та підвищує надійність оновлень, що є критично важливим у системах із великою кількістю компонентів.

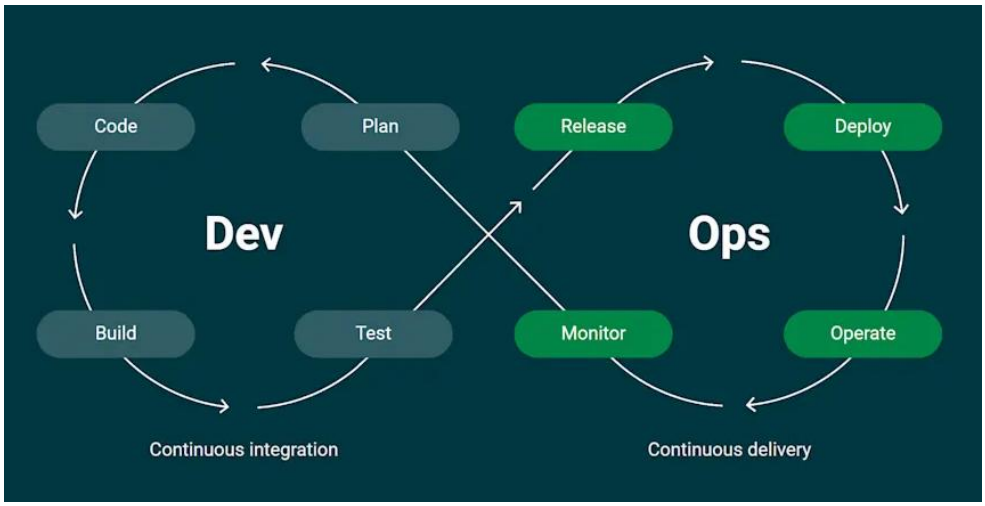


Рисунок 1.16 CI/CD

Технологічний стек мікросервісної архітектури формує основу стабільного та ефективного виконання сервісів у розподілених середовищах. Поєднання контейнеризації, оркестрації, брокерів повідомлень, API-шлюзів, спеціалізованих баз даних та інструментів моніторингу створює комплексну інфраструктуру, здатну підтримувати високі навантаження та гарантовано забезпечувати узгодженість і доступність сервісів. Ця технологічна база стала невід’ємною умовою побудови мікросервісних систем, орієнтованих на обробку великих інформаційних потоків і підтримку широкого спектра бізнес-процесів.

1.6 Переваги, недоліки та сфери застосування мікросервісної архітектури

Переваги мікросервісної архітектури визначають її популярність у сучасних високонавантажених системах, де важливо забезпечити гнучкість і масштабованість. Однією з фундаментальних переваг є незалежність сервісів, що дає змогу розгортати та оновлювати окремі компоненти без зупинки всієї системи. Така властивість значно спрощує модернізацію програмних комплексів та прискорює впровадження нових функцій. У середовищах, де зміни конфігурацій і логіки є регулярними, можливість автономного оновлення окремих модулів стає критично важливою.

Висока масштабованість мікросервісних систем забезпечується можливістю збільшувати ресурси лише для тих сервісів, які зазнають підвищеного навантаження. У традиційних монолітних архітектурах масштабування здійснюється шляхом дублювання всієї системи, що створює зайві витрати ресурсів. У мікросервісах масштабування має вибірковий характер, що підвищує ефективність використання інфраструктури. Контейнеризація та оркестрація дають змогу автоматично реагувати на зміни навантаження, створюючи нові екземпляри сервісів у разі пікового навантаження та зменшуючи їхню кількість у періоди низької активності.

Відмовостійкість є однією з найважливіших властивостей мікросервісних систем, оскільки компоненти працюють незалежно один від одного. Збій окремого сервісу не призводить до повної зупинки системи, а лише до тимчасової недоступності певної функції. Оркестратори контейнерів здатні автоматично перезапускати сервіси, замінювати їхні екземпляри та відновлювати баланс навантаження. Це підвищує надійність системи та мінімізує ризик простою навіть у разі непередбачуваних збоїв. Для систем, які працюють у реальному часі, така властивість є критичною.

Недоліки мікросервісної архітектури пов'язані з підвищеною складністю управління розподіленим середовищем. Кожний сервіс потребує власної інфраструктури, засобів моніторингу й логування. Кількість мережевих викликів між сервісами зростає, що збільшує затримки та висуває підвищені вимоги до пропускної здатності мережі. Крім того, тестування розподілених систем є складнішим, оскільки необхідно перевіряти цілісність взаємодій між великою кількістю автономних компонентів.

Проблема узгодженості даних є одним із головних викликів мікросервісної архітектури. Відсутність спільного сховища означає, що кожен сервіс має власну базу даних, а зміни синхронізуються через обмін повідомленнями. У таких системах неможливо застосувати класичні транзакційні механізми, тому розробники використовують патерни, які забезпечують коректність даних у розподіленому середовищі. До таких підходів належать Saga, що передбачає виконання послідовності локальних транзакцій із компенсацією, та Outbox, який гарантує доставку подій через збереження їх у проміжній таблиці. Використання цих механізмів є необхідним, але збільшує складність реалізації.

Сфери застосування мікросервісної архітектури охоплюють системи, де важливими є масштабованість, швидкодія та відмовостійкість. Сучасні платіжні платформи використовують мікросервіси для розподілу транзакційних потоків і забезпечення обробки великої кількості операцій у реальному часі. У сфері електронної комерції мікросервіси забезпечують роботу автономних модулів — від каталогів товарів до процесингу замовлень. Аналітичні системи, телеметричні платформи та рішення для обробки потоків подій застосовують мікросервіси завдяки їхній здатності ефективно обробляти нерівномірні інформаційні потоки та підтримувати високу доступність ключових компонентів.

Важливу роль у поширенні мікросервісів відіграють хмарні платформи, які надають інструменти для автоматизації розгортання та масштабування. Рішення на базі AWS, Azure та Google Cloud пропонують керовані кластери Kubernetes, системи маршрутизації та сервіси для потокової обробки даних. Використання цих інструментів зменшує навантаження на команду розробки та

дозволяє будувати розподілені системи з використанням інфраструктури «як сервіс».

Узагальнюючи властивості мікросервісної архітектури, можна стверджувати, що вона є доцільною у системах, які працюють з великими інформаційними потоками, мають високі вимоги до надійності та потребують гнучкого масштабування. Незважаючи на складність реалізації, переваги мікросервісного підходу роблять його одним із ключових напрямів розвитку сучасних програмних систем, особливо у сферах, де критично важлива швидкість обробки даних та доступність сервісів.

2 МОДЕЛЮВАННЯ ТА УПРАВЛІННЯ ІНФОРМАЦІЙНИМИ ПОТОКАМИ

2.1 Поняття та класифікація інформаційних потоків у розподілених системах

Поняття інформаційного потоку є фундаментальним для розуміння процесів обміну даними між компонентами розподілених систем. У загальному вигляді інформаційний потік описує послідовність даних, що передаються від одного елемента системи до іншого з метою виконання певної функції. У мікросервісних архітектурах інформаційні потоки мають особливу значущість, оскільки саме через них здійснюється взаємодія між автономними сервісами, формування відповідей на запити користувачів, синхронізація станів та реалізація бізнес-логіки на рівні всієї системи.

Ключовою властивістю інформаційних потоків у мікросервісних системах є їхня децентралізованість, оскільки дані рухаються між автономними компонентами, що мають власні ресурси, бази даних та інтерфейси. Це створює середовище, де потоки стають нерівномірними, динамічними і нерідко непередбачуваними. Навантаження на певні сервіси може значно відрізнятися в різні проміжки часу, що зумовлює необхідність гнучкої адаптації механізмів маршрутизації та обробки даних. Здатність системи ефективно працювати з такими потоками визначає її загальну продуктивність та стабільність.

Класифікація інформаційних потоків у мікросервісних системах дає змогу описати різні сценарії взаємодії між сервісами та зрозуміти природу даних, які циркулюють у системі. Одним із базових підходів до класифікації є поділ потоків за їхнім функціональним призначенням. У практиці розподілених систем виокремлюють такі основні типи потоків:

— *транзакційні потоки,* які відображають передачу даних у межах операцій, що мають чіткі послідовності дій і високі вимоги до узгодженості;

— *подієві потоки,* які ґрунтуються на генерації та поширенні подій у системі, часто у режимі реального часу;

— *аналітичні потоки,* пов’язані з обробкою значних масивів даних, що використовуються для прогнозування, статистичної обробки чи машинного навчання;

— *сервісні потоки,* які забезпечують внутрішні технічні функції, як-от синхронізація станів, оновлення кешів або службові запити конфігурацій.

Значну роль відіграє також класифікація інформаційних потоків за напрямом руху даних, яка дозволяє зрозуміти логіку їх циркуляції у системі. У розподілених середовищах можна виділити потоки типу «запит–відповідь», що характерні для синхронної взаємодії між сервісами, потокові канали, які забезпечують неперервну передачу даних, а також широкомовні потоки, що поширюють інформацію серед кількох зацікавлених сервісів одночасно. Різноманітність напрямів руху впливає на затримки, пропускну здатність і вимоги до надійності.

Важливою характеристикою потоків у мікросервісній архітектурі є їхня нерівномірність, що відображає різку зміну інтенсивності обробки даних у різні моменти часу. Це явище часто фіксується в системах, де навантаження залежить від активності користувачів або від подій у зовнішньому середовищі, наприклад, у фінансових платформах або службах реального часу. Нерівномірність потоків потребує застосування спеціальних механізмів балансування, кешування та чергування, що забезпечують стабільну роботу системи навіть під час пікових навантажень.

Ключову роль у характеристиці інформаційних потоків відіграє чутливість до затримок, яка визначає здатність системи реагувати на події у потрібні терміни. Потоки, які обслуговують транзакції або критичні бізнес-процеси, потребують мінімальних затримок та високої швидкості обробки. Аналітичні потоки можуть допускати деякі затримки, але мають забезпечувати сталу високу пропускну здатність. У розподілених системах баланс між цими вимогами визначається архітектурними рішеннями щодо взаємодії сервісів, вибору протоколів і підходів до маршрутизації.

Поняття інформаційного потоку охоплює також його структуру, тобто формат і спосіб організації даних у межах системи. Формати обміну, такі як JSON (рис. 2.1), Protobuf (рис. 2.2) або Avro (рис. 2.3), визначають спосіб серіалізації, компактність повідомлень і швидкість їх обробки. Вибір структури даних впливає на загальну продуктивність, особливо у високонавантажених сервісах, де обмін повідомленнями здійснюється з великою частотою. Правильне визначення структури потоків дозволяє оптимізувати як внутрішню взаємодію між сервісами, так і комунікацію із зовнішніми системами.


```
{
  "DocumentType": 1,
  "No.": "S-ORD101001",
  "SellToCustNo": "10000",
  "PostingDate": "2023-04-02",
  "Lines": [
    {
      "LineNo": 10000,
      "Type": 2,
      "No": "1996-S",
      "Quantity": 12,
      "UnitPrice": 1397.3
    },
    {
      "LineNo": 20000,
      "Type": 2,
      "No": "1900-S",
      "Quantity": 4,
      "UnitPrice": 192.8
    }
  ]
}
```


Рисунок 2.1 JSON Формат

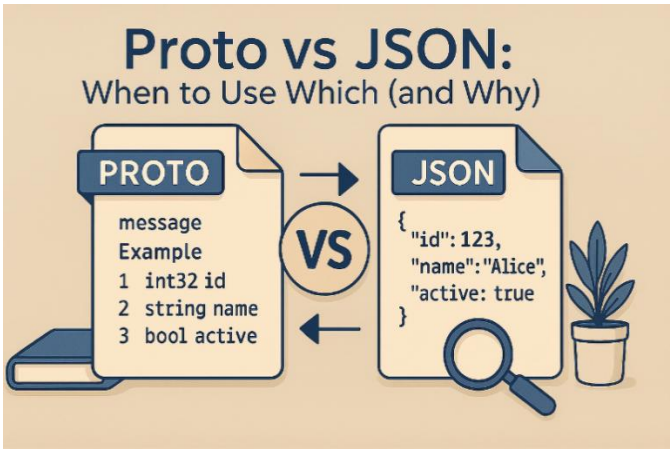


Рисунок 2.2 Protobuf та JSON

```
{
  "type":"record",
  "namespace": "io.confluent.developer.avro",
  "name":"Purchase",
  "fields": [
    {"name": "item", "type":"string"},
    {"name": "total_cost", "type": "double"},
    {"name": "customer_id", "type": "string"}
  ]
}
```

Рисунок 2.3 Приклад схеми Avro

2.2 Моделі опису та організації інформаційних потоків у мікросервісних архітектурах

Моделі опису інформаційних потоків визначають принципи формування, передачі та обробки даних у мікросервісних системах. Оскільки мікросервіси функціонують як автономні компоненти, їхня взаємодія забезпечується через визначені канали обміну даними, що можуть бути синхронними або асинхронними. Вибір відповідної моделі залежить від характеру потоку, вимог до швидкодії, структурованості інформації та рівня узгодженості, що має бути досягнутий між сервісами.

Логічні моделі інформаційних потоків (рис. 2.4) описують послідовність операцій та взаємодій між сервісами на концептуальному рівні. У таких моделях визначаються джерела та споживачі даних, характер операції, часові залежності й обмеження. Логічне моделювання дозволяє відстежити зв’язки між потоками та визначити, які компоненти впливають на процеси обробки. Це важливо під час проєктування складних систем, у яких дані проходять через кілька послідовних або паралельних стадій обробки.

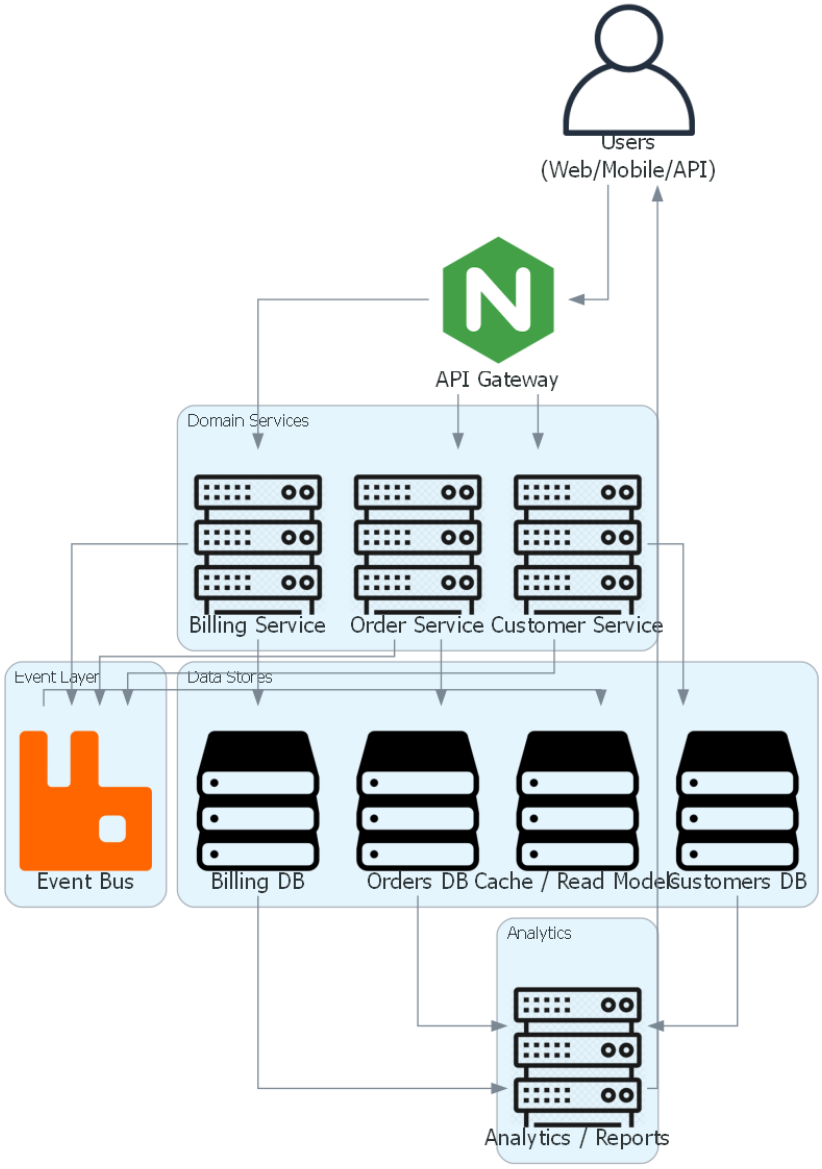


Рисунок 2.4 Приклад логічної моделі інформаційних потоків

Формальні моделі опису потоків застосовують для точного визначення структур даних, протоколів взаємодії та механізмів серіалізації. Ці моделі забезпечують однозначність інтерпретації даних між сервісами, що критично важливо у розподіленому середовищі. Ключове значення має вибір формату даних, оскільки саме він визначає швидкість обробки, компактність повідомлень та навантаження на мережу. Найпоширенішими форматами серіалізації є JSON, Protobuf та Avro.

— JSON використовується у системах, орієнтованих на взаємодію з веб-клієнтами, через свою гнучкість та зручність читання.

— Protobuf забезпечує високу швидкість обробки та компактний формат, що робить його придатним для внутрішньої взаємодії між сервісами.

— Avro застосовується у потокових системах і дозволяє працювати з великими обсягами даних, зберігаючи їх у компактному бінарному вигляді.

Особливе значення у мікросервісних архітектурах має *модель "запит–відповідь"* (рис. 2.5), яка забезпечує синхронний обмін даними між сервісами. Вона є природною для REST або gRPC і дозволяє швидко отримувати інформацію, коли відповідь потрібна негайно. Проте ця модель має обмеження, пов'язані з мережевими затримками, тому в системах із високою інтенсивністю потоків її застосовують лише для операцій, що потребують чіткої послідовності або валідованих результатів.

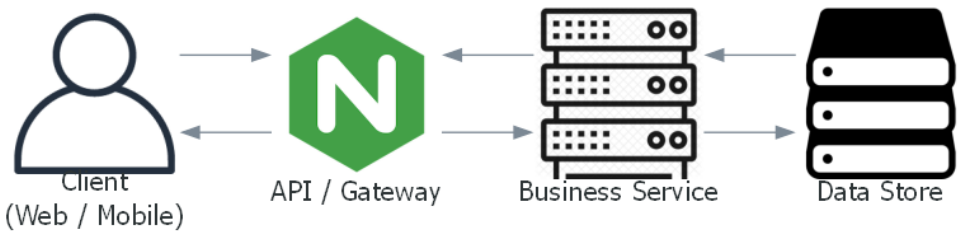


Рисунок 2.5 Модель "запит-відповідь"

Асинхронні моделі (рис. 2.6) обміну даними є ключовими для систем, що обробляють великі інформаційні потоки або працюють у режимі реального часу. Вони ґрунтуються на передачі повідомлень через брокерів, що дозволяє відправнику та отримувачу працювати незалежно один від одного. Потоки, що реалізуються таким чином, мають високу стійкість до навантажень і здатність адаптуватися до нерівномірної активності сервісів. Асинхронний обмін є фундаментом для побудови подійно-орієнтованих систем та інформаційних конвеєрів.

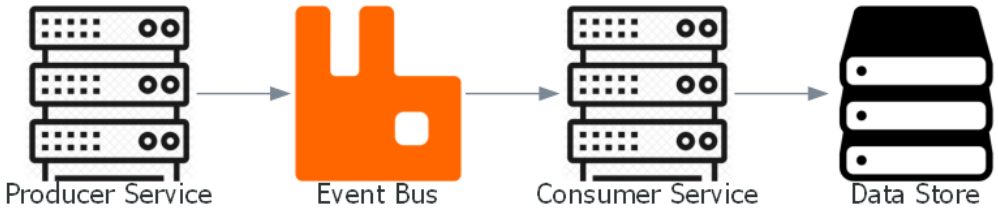


Рисунок 2.6 Асинхронна модель обміну даними

Подійно-орієнтовані моделі (рис. 2.7) організації потоків описують взаємодію, у якій дані передаються внаслідок виникнення певних подій у системі. Такі моделі широко застосовуються у середовищах, де важлива реакція на зміни станів або активність зовнішніх систем. Структура подій передбачає формування повідомлення, яке містить опис зміни, та подальше поширення його до всіх зацікавлених сервісів. Цей підхід дає змогу розділити джерело події та її споживачів, зменшуючи зв’язаність між компонентами.

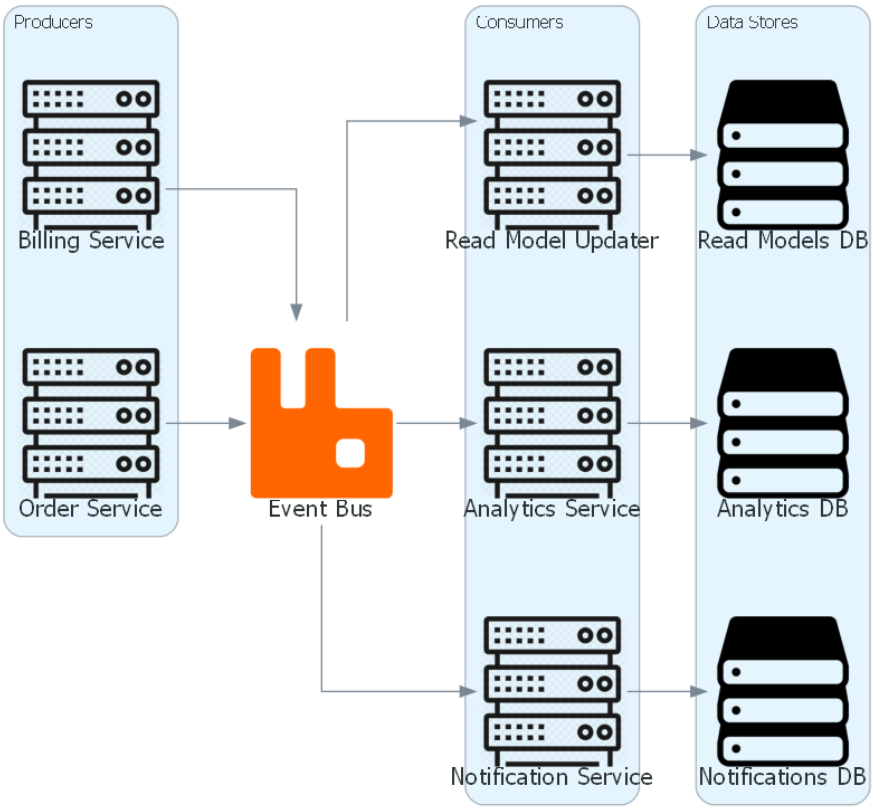


Рисунок 2.7 Подійно-орієнтована модель

Модель потокового передавання (рис. 2.8) даних використовується у випадках, коли інформація надходить у вигляді безперервного потоку. Така модель характерна для телеметрії, логування, аналітичних процесів і систем моніторингу. Потокове передавання забезпечує стабільну обробку даних у режимі реального часу, що робить його необхідним у системах із високими вимогами до швидкості обробки.

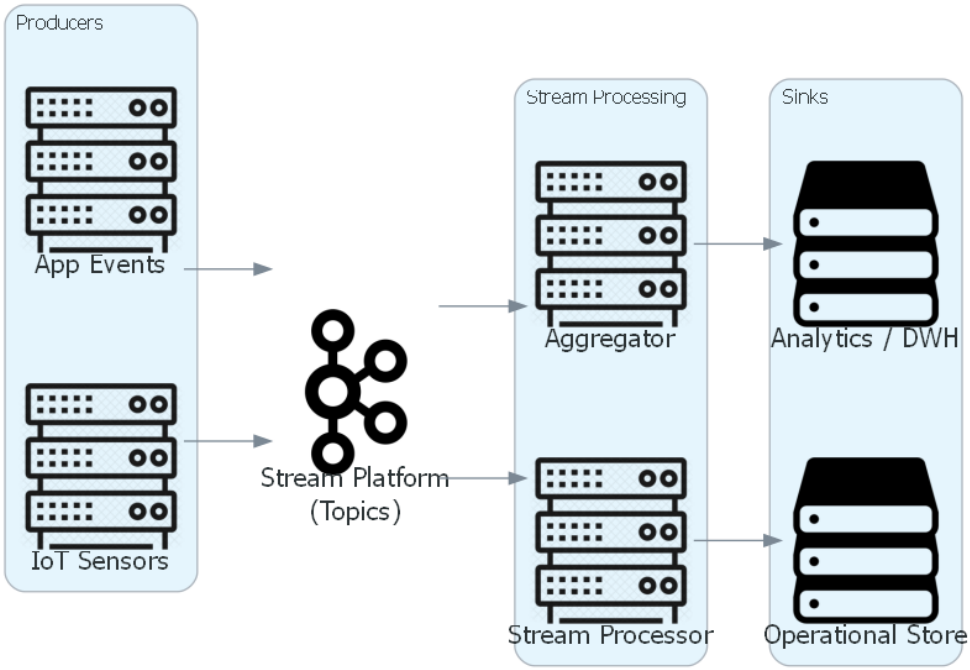


Рисунок 2.8 Модель потокового передавання із використанням Kafka

Важливою частиною організації потоків є механізми маршрутизації даних, які визначають шлях, яким потоки проходять між сервісами. У мікросервісних системах маршрутизація може бути прямою, коли сервіс надсилає повідомлення конкретному отримувачу, або непрямою, коли дані передаються через брокера чи API-шлюз. Розвинуті системи маршрутизації підтримують фільтрацію за змістом, розподіл навантаження між кількома інстансами сервісів та перенаправлення потоків залежно від їхніх характеристик.

Моделі опису та організації інформаційних потоків формують основу для прийняття архітектурних рішень у мікросервісних системах. Розуміння цих моделей дозволяє обирати оптимальні підходи до проєктування та впровадження потоків, з огляду на вимоги до швидкодії, узгодженості й масштабованості. Вибір між синхронними, асинхронними, подійно-орієнтованими або потоковими моделями визначає структуру взаємодії між сервісами і є ключовим елементом забезпечення ефективної роботи системи.

2.3 Методи забезпечення продуктивності та узгодженості інформаційних потоків

Продуктивність інформаційних потоків у мікросервісних системах є одним із ключових параметрів, що визначають здатність системи стабільно обробляти значні обсяги даних. У таких системах навантаження розподіляється нерівномірно, і тому важливо забезпечити ефективну маршрутизацію повідомлень, оптимізувати формати даних і зменшувати загальні затримки. Продуктивність пов’язана не лише з швидкістю виконання окремих операцій, але й з тим, як швидко система може адаптуватися до пікових обсягів запитів.

Оптимізація затримок є одним із найважливіших завдань підвищення продуктивності. Навіть невеликий додатковий час обробки в одному сервісі може суттєво збільшити загальний час проходження запиту між декількома сервісами. Застосування компактних та швидких форматів серіалізації, таких як Protobuf або Avro, дозволяє знизити навантаження на мережу та прискорити обробку. Кешування відповідей для часто виконуваних операцій додатково розвантажує бази даних і зменшує кількість зовнішніх викликів.

Пропускна спроможність інформаційних потоків визначається здатністю системи виконувати операції паралельно та обробляти множинні потоки даних одночасно. Горизонтальне масштабування є основним методом підвищення пропускної здатності. У мікросервісній архітектурі це досягається запуском додаткових екземплярів окремих сервісів, які працюють незалежно один від одного. Для асинхронних потоків брокери повідомлень, такі як RabbitMQ або Kafka, забезпечують паралельну обробку, розподіляючи навантаження між кількома споживачами. Це робить можливим оброблення мільйонів подій у системах реального часу.

Узгодженість даних у розподілених системах є складним завданням, оскільки мікросервіси не поділяють єдину транзакційну модель. Кожен сервіс зберігає власні дані, і тому оновлення не можуть відбуватися атомарно. Замість класичних

транзакцій застосовують підходи, які забезпечують коректність обробки подій у довгостроковій перспективі. Основним принципом є досягнення кінцевої узгодженості, коли дані стають узгодженими через певний проміжок часу, а не миттєво.

Таблиця 2.1

Методи забезпечення узгодженості даних у мікросервісах

Метод	Призначення	Ключові особливості
Saga	Розподілені транзакції	Локальні операції + компенсації; два підходи — хореографія та оркестрація
Outbox Pattern	Гарантована доставка подій	Подія фіксується у базі перед відправленням; повторна доставка до підтвердження
Idempotency	Коректна повторна обробка операцій	Використання унікальних ключів, контроль дублікатів
Retry + Backoff	Повторення операцій у разі збоїв	Експоненційні паузи, контроль кількості повторів
Dead Letter Queue	Обробка невдалих повідомлень	Збереження "проблемних" подій в окремій черзі

Патерн Saga є центральним підходом до керування складними транзакціями у розподіленому середовищі. Він передбачає розбиття глобальної транзакції на низку локальних операцій, кожна з яких відповідає за одну частину процесу. У разі

помилки Saga ініціює відповідні компенсаційні дії, які повертають систему у попередній коректний стан. Підхід Saga (рис. 2.9) забезпечує стабільність системи навіть у разі тимчасових збоїв окремих сервісів.

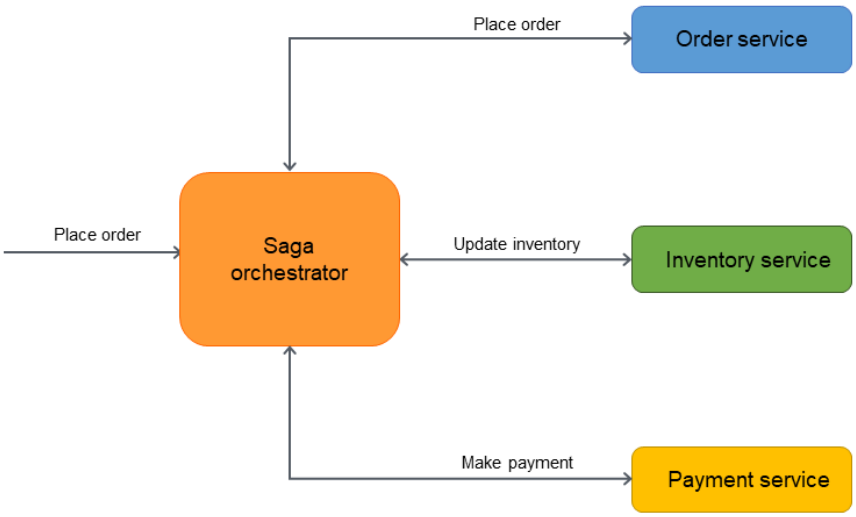


Рисунок 2.9 Saga патерн

Outbox-патерн використовується у випадках, коли важливо гарантувати доставлення подій, що супроводжують оновлення даних. Його реалізація забезпечує узгодженість між операціями, виконаними в базі даних, та подіями, які мають бути відправлені іншим сервісам. Цей підхід особливо корисний у високонавантажених системах, де недоставлене повідомлення може призвести до розбіжностей у станах сервісів.

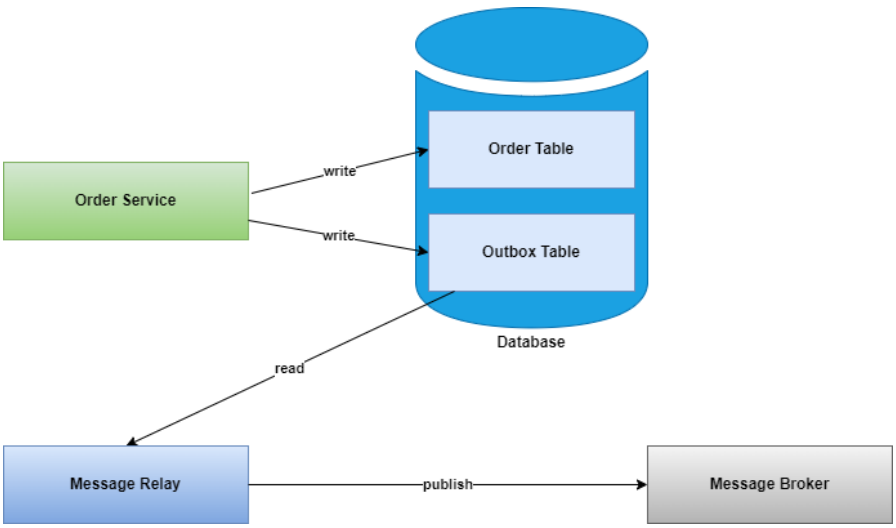


Рисунок 2.10 Outbox патерн

Ідемпотентність дозволяє уникнути проблем дублювання операцій, що є характерним явищем у системах з асинхронною доставкою. У випадках повторного виконання однієї й тієї самої операції результат має залишатися незмінним. Це досягається завдяки використанню унікальних ідентифікаторів операцій, контролю журналів або спеціальних механізмів станів, що зберігають інформацію про оброблені події.

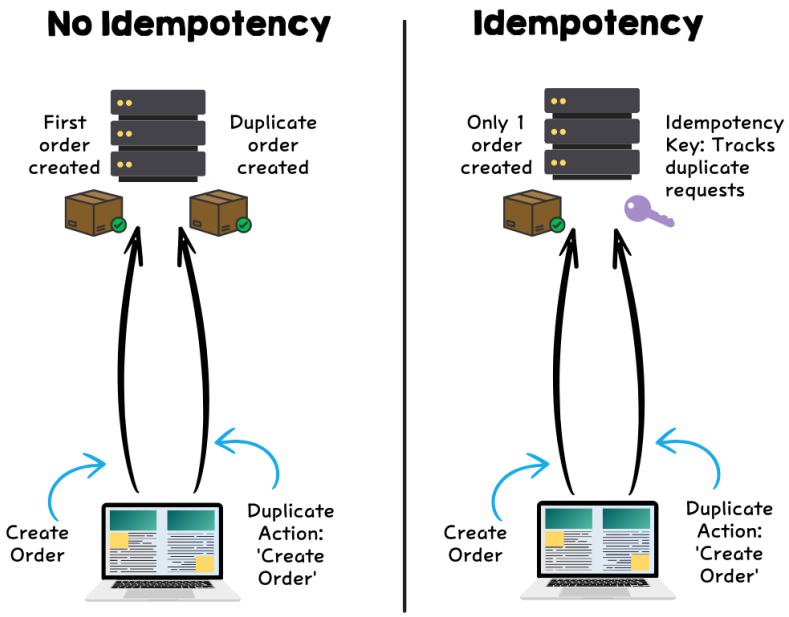


Рисунок 2.11 Ідемпотентність

Методи обробки помилок, такі як повторні спроби та відкладена доставка, дозволяють системі залишатися стабільною навіть у випадках тимчасових проблем з мережею або ресурсоємними операціями. Стратегії retry з експоненційною затримкою зменшують навантаження на систему, а механізми Dead Letter Queue дають можливість аналізувати невдалі повідомлення окремо, не порушуючи основний потік.

Моніторинг якості інформаційних потоків є завершальним елементом забезпечення їхньої стабільності та продуктивності. Використання систем збору метрик дає можливість оцінювати час обробки повідомлень, глибину черг,

швидкість доставлення та частоту помилок. Такі дані дозволяють прогнозувати поведінку системи, виявляти вузькі місця та оперативно реагувати на потенційні проблеми.

2.4 Управління потоками подій у системах реального часу

Потоки подій є фундаментальним компонентом сучасних мікросервісних систем, що працюють у режимі реального часу. Вони формуються внаслідок дій користувачів, змін у станах сервісів або зовнішніх подій у технічному середовищі. Потоки подій відрізняються нерівномірністю, високою швидкістю надходження та непередбачуваною інтенсивністю, що вимагає спеціальних механізмів їхнього керування. У системах реального часу важливо не лише фіксувати події, але й забезпечувати їхню своєчасну доставку та обробку, зберігаючи стабільність роботи всієї системи.

Подійно-орієнтована архітектура є основою побудови систем, що реагують на події у момент їхнього виникнення. У такій архітектурі сервіси не взаємодіють напряму, а обмінюються повідомленнями через брокерів або шини подій. Це дозволяє знизити взаємозалежність між компонентами та підвищити масштабованість системи. Джерела подій формують повідомлення, які публікуються у відповідні канали, тоді як споживачі отримують лише інформацію, що їм потрібна. Такий підхід забезпечує ефективний розподіл навантаження та дає змогу реагувати на події з мінімальними затримками.

Системи потокової обробки даних забезпечують підтримку високошвидкісного надходження подій та їхню безперервну обробку. Однією з найпоширеніших технологій для цього є Apache Kafka, яка підтримує горизонтальне масштабування та розподіл повідомлень між великою кількістю споживачів. Kafka дозволяє зберігати події у вигляді журналів, що робить можливим повторну обробку даних та аналіз історії подій. Завдяки високій пропускній здатності Kafka часто використовується в аналітичних системах,

телеметрії та сервісах, орієнтованих на моніторинг станів обладнання у режимі реального часу.

Apache Pulsar є альтернативною стримінговою платформою, яка поєднує модель чергування повідомлень із розподіленим зберіганням подій. Pulsar використовує сегментоване сховище та архітектуру з поділом ролей брокера й сторіджу, що дозволяє обслуговувати велику кількість потоків одночасно. У межах систем реального часу цей підхід підвищує стабільність, оскільки збої на рівні брокера не призводять до втрати історії подій. Pulsar також підтримує багаторівневу маршрутизацію та різні моделі доставки, що дозволяє використовувати його у великих розподілених системах.

Управління потоками подій потребує застосування механізмів, що гарантують надійність і послідовність їхньої обробки. У системах реального часу важливо уникнути пропусків подій, дублювання або порушення порядку їхньої обробки. Для цього використовуються механізми фіксації зсувів (offsets), журнали подій та контрольні точки, які дозволяють точно визначати позицію споживача у потоці. У випадку тимчасових збоїв це забезпечує відновлення обробки без втрати даних.

Особливої уваги потребує проблема балансування навантаження між кількома екземплярами споживачів подій. Оскільки інтенсивність потоку може різко змінюватися, необхідно забезпечити гнучкий розподіл подій між споживачами. У Kafka це реалізується через партиціонування: кожна тема може бути розділена на кілька незалежних партицій, що дозволяє обробляти потік паралельно. Pulsar застосовує аналогічний підхід, використовуючи розподіл потоків між підписниками залежно від їхньої продуктивності. Балансування забезпечує стабільну роботу системи навіть у випадках раптового зростання кількості подій.

Затримки у потоках подій є одним із найважливіших параметрів, що визначають якість роботи системи реального часу. Затримки можуть виникати як на рівні обробки сервісами, так і через мережеві обмеження чи високе навантаження на брокера. Для мінімізації затримок застосовуються такі підходи:

— використання швидких форматів серіалізації;
— застосування in-memory механізмів кешування;
— оптимізація кількості споживачів;
— адаптивне масштабування залежно від інтенсивності потоку.

Стійкість до збоїв у потоках подій є критично важливою, оскільки навіть короткочасна недоступність сервісу може спричинити накопичення нерозібраних повідомлень. Для забезпечення стійкості застосовуються механізми повторної доставки, розподіленого зберігання журналів та автоматичного перемикання на резервні брокери. Kafka використовує реплікацію партицій, що дозволяє відновити дані у разі збою вузла. Pulsar застосовує розподілене зберігання у BookKeeper, що забезпечує більшу стабільність при високих навантаженнях.

Події у системах реального часу відіграють важливу роль у формуванні інформаційних потоків, що охоплюють як бізнес-логіку, так і технічні процеси. Завдяки подіям система реагує на зміни у станах сервісів, генерує оновлення в інтерфейсах користувачів, обробляє транзакції та будує аналітичні моделі. У великих платформах потоки подій стають основою взаємодії між автономними компонентами, що підвищує стабільність та гнучкість роботи системи.

Таблиця 2.2

Порівняння Kafka та Pulsar у контексті потоків реального часу

Характеристика	Apache Kafka	Apache Pulsar
Архітектура	Монолітне поєднання брокера і логів	Розділення: брокер + сховище BookKeeper
Масштабованість	Висока, через партиції	Дуже висока, завдяки сегментованому зберіганню
Зберігання історії	Журнали на брокерах	Незалежне розподілене сховище

Продовження таблиці 2.2

Порівняння Kafka та Pulsar у контексті потоків реального часу

Характеристика	Apache Kafka	Apache Pulsar
Затримки	Мінімальні за стабільної інфраструктури	Низькі навіть при високому навантаженні
Надійність	Реплікація партицій	Реплікація на рівні BookKeeper
Оптимальне використання	Аналітичні й телеметричні системи	Масштабовані розподілені системи із критичними навантаженнями

Управління потоками подій у системах реального часу формує основу роботи мікросервісних архітектур, що потребують стабільної, масштабованої та швидкої обробки інформації. Ефективні моделі доставки, надійні механізми зберігання та гнучкі підходи до балансування навантаження дозволяють сучасним системам коректно реагувати на події незалежно від складності та розміру інфраструктури.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1 Аналіз вимог та формування архітектурної моделі системи

Аналіз вимог є ключовим етапом проєктування мікросервісної системи, оскільки саме на цьому рівні формується розуміння того, які саме інформаційні потоки повинні підтримуватися, якими є функціональні обмеження та які характеристики продуктивності необхідно забезпечити. Для системи управління інформаційними потоками важливо визначити джерела даних, типи повідомлень, що циркулюють між сервісами, та закономірності навантаження, які впливатимуть на архітектурні рішення. Формування вимог дозволяє структурувати потреби користувачів і технічні цілі, забезпечуючи узгодженість подальших рішень на рівні моделювання та реалізації.

Функціональні вимоги системи визначають перелік операцій і механізмів, які повинні бути реалізовані для обробки та маршрутизації інформаційних потоків. До основних функцій відноситься можливість прийому вхідних подій через HTTP-інтерфейс, автоматична генерація подій службовими компонентами системи, передавання подій через брокер повідомлень, обробка даних у відокремленому сервісі та збереження результатів у базі даних для подальшого аналізу. Сюди також належить забезпечення надійної доставки подій, можливість масштабування компонентів та підтримка механізмів збору статистики щодо інтенсивності потоків і швидкості їх обробки.

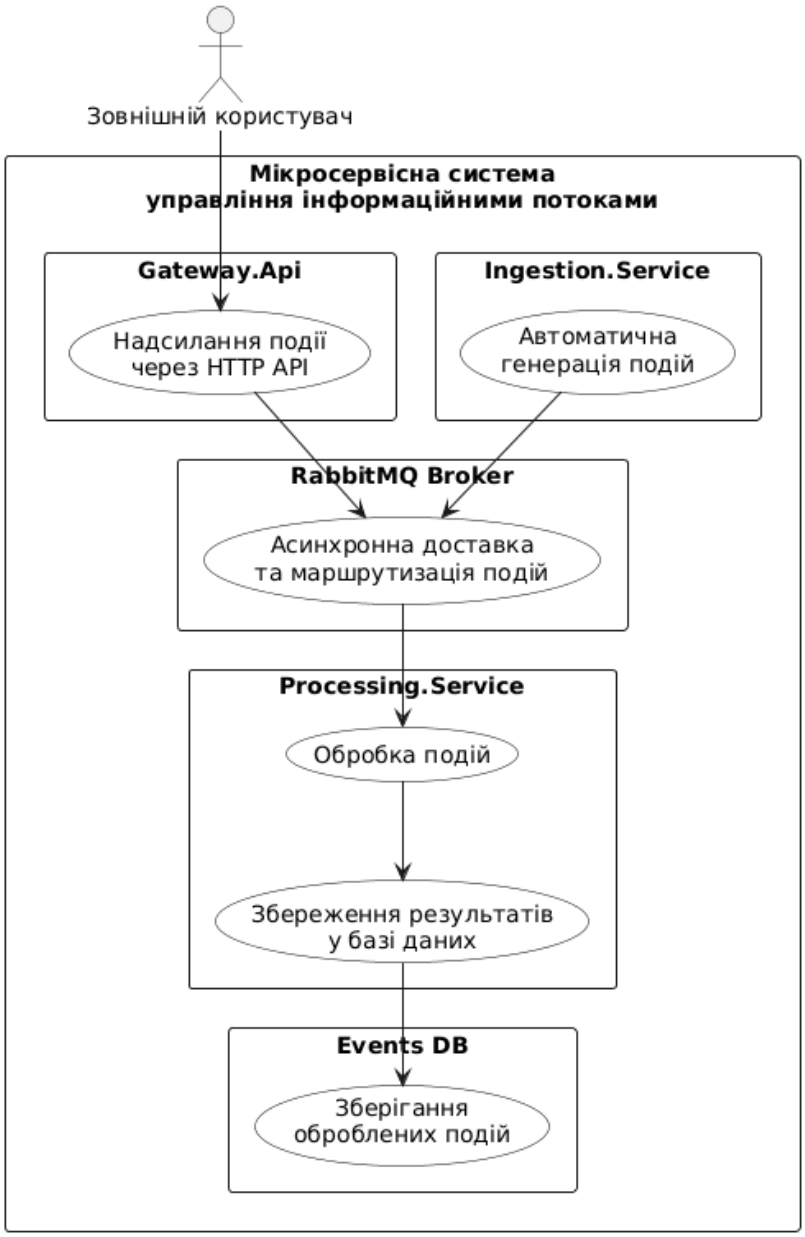


Рисунок 3.1 Діаграма використання

Нефункціональні вимоги визначають характеристики системи, які безпосередньо впливають на якість її роботи та відповідність заданим критеріям ефективності. До таких вимог належать висока пропускна здатність каналів комунікації, низькі затримки під час обробки подій, стійкість до відмов компонентів та здатність системи масштабуватися горизонтально з метою обробки зростаючого потоку даних. Окреме місце займають вимоги до надійності доставлення повідомлень, адже інформаційні потоки не повинні втрачатися під час збою сервісів або мережевих проблем. Також важливими є вимоги до

спостережуваності, що включають логування, метрики, інструменти моніторингу та контроль станів брокера повідомлень.

Аналіз доменної області дозволяє описати природу даних, які циркулюють у системі, та визначити контекст взаємодії сервісів. У розглянутій системі інформаційні потоки складаються з подій, що надходять як від зовнішнього API, так і від внутрішніх компонентів, які генерують автоматизовані повідомлення. Ці події можуть містити довільні payload-дані, але обов'язково включають метадані про джерело та час створення, що дозволяє аналізувати їхній шлях у системі. Такий підхід забезпечує узагальненість системи та дозволяє переносити архітектурну модель на інші доменні області, де використовуються подієві або транзакційні інформаційні потоки.

На основі зібраних вимог формується концептуальна архітектурна модель, яка визначає центральні компоненти системи та їхні взаємозв'язки. Архітектурна модель включає Gateway-сервіс, який приймає зовнішні запити, Ingestion-сервіс, що генерує внутрішні потоки даних, брокер повідомлень (RabbitMQ) як центральний елемент асинхронної комунікації та Processing-сервіс, який обробляє події та зберігає результати у базі даних. Така модель відокремлює джерела подій від їхньої обробки, дозволяно масштабувати кожен сервіс незалежно та підтримувати різні сценарії обробки даних у майбутньому.

Узагальнена архітектура системи представляє собою набір взаємодіючих мікросервісів, що формують асинхронну подієву модель обробки інформаційних потоків. Gateway.Api забезпечує прийом зовнішніх подій і маршрутизує їх у чергу, Ingestion.Service автоматизовано генерує події з певною частотою, а Processing.Service забезпечує їх обробку, фіксацію часу проходження та внесення результатів до бази даних. Центральним компонентом виступає RabbitMQ, що забезпечує незалежність темпів роботи джерел і споживачів подій і дозволяє системі залишатися стабільною навіть за нерівномірного навантаження.

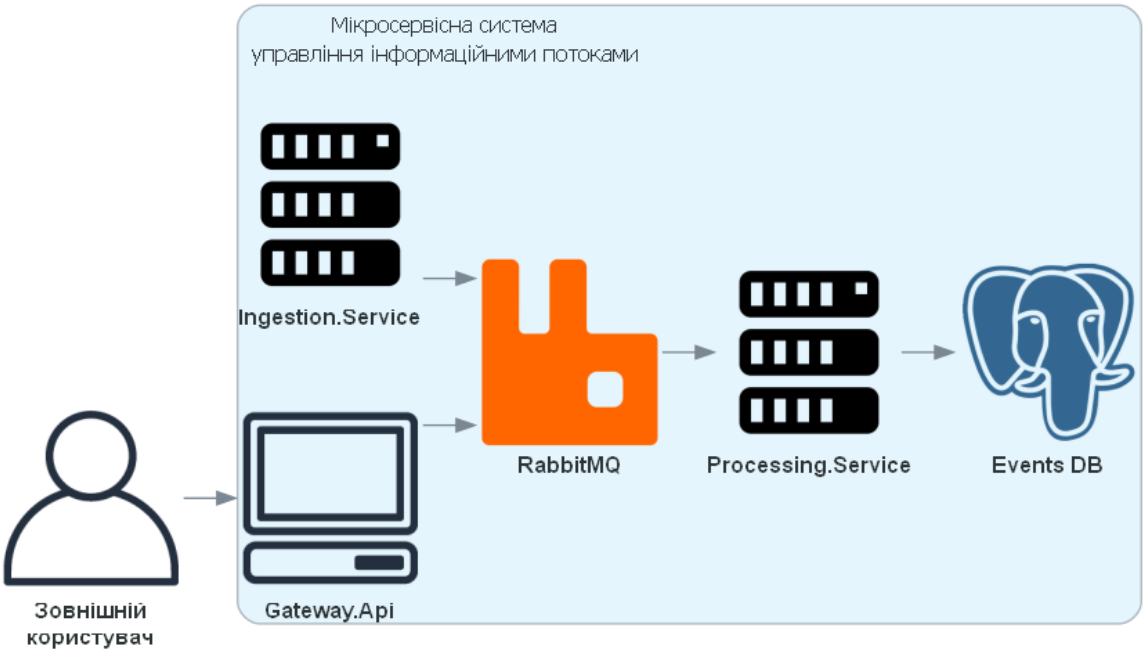


Рисунок 3.2 Узагальнена архітектура системи

Сформовані вимоги дозволяють побудувати архітектуру, що відповідає принципам мікросервісної системи та забезпечує ефективне управління інформаційними потоками. Чітке відокремлення компонентів, використання брокера повідомлень для асинхронної комунікації та фіксація даних у сховищі створюють основу для подальших етапів: проєктування API, моделювання потоків повідомлень та розгортання інфраструктури.

3.2 Проєктування структури мікросервісів та їх взаємодії

Структура мікросервісної системи базується на чіткому виділенні компонентів за функціональною ознакою та організації взаємодії між ними через стандартизовані канали комунікації. Такий підхід забезпечує слабе зв’язування сервісів, дозволяє масштабувати їх незалежно та гарантує стабільність роботи під час зміни інтенсивності інформаційних потоків. Кожен компонент системи виконує окрему роль у життєвому циклі події, а їх взаємодія формує цілісну асинхронну модель обробки даних.

Gateway.Api виступає точкою входу для зовнішніх подій, забезпечуючи прийом HTTP-запитів та перетворення синхронних звернень у повідомлення. У цьому сервісі формується стандартна структура події з уніфікованим набором полів, що гарантує однакову подальшу обробку в системі. Gateway.Api не виконує обчислювальних операцій і не зберігає дані, а виключно передає події у брокер повідомлень.

Ingestion.Service відповідає за формування внутрішніх інформаційних потоків. Компонент генерує події з певною частотою, що дозволяє моделювати реальні сценарії роботи системи з інтенсивними або фоновими потоками даних. Надсилання подій до брокера повідомлень здійснюється тим самим механізмом, що й у Gateway.Api, забезпечуючи єдину модель транспортного обміну.

Центральну роль у взаємодії компонентів виконує *RabbitMQ*, який реалізує асинхронну комунікацію між сервісами. Джерела подій можуть працювати зі змінним навантаженням, а брокер приймає їх незалежно від готовності Processing.Service до негайної обробки. Такий підхід підтримує буферизацію, ізоляцію темпів роботи компонентів та гнучке масштабування сервісів обробки.

Processing.Service функціонує як споживач подій і виконує їх подальшу обробку. Компонент контролює затримку між моментом формування події та часом її обробки, а також збагачує інформацію додатковими метриками. Оброблені дані зберігаються в базі даних, що забезпечує можливість проведення аналізу ефективності та дослідження характеристик потоків.

Events DB зберігає результати обробки, формуючи історію подій та накопичуючи часові характеристики, важливі для оцінки продуктивності системи. Структура даних орієнтована на подієву модель і дозволяє швидко аналізувати затримки, інтенсивність і поведінку потоків. Завдяки централізованому збереженню результатів забезпечується узгодженість статистичних даних і можливість розширення функціональності у разі потреби.

Взаємодія між сервісами формує цілісну модель обробки інформаційних потоків, у якій кожен компонент має окреслену зону відповідальності. Gateway.Api та Ingestion.Service створюють події, RabbitMQ транспортує їх, а Processing.Service

відповідає за логіку обробки та збереження результатів. Така модульність забезпечує можливість незалежного масштабування будь-якого компонента, адаптації до змін навантаження або впровадження додаткових сервісів у майбутньому.

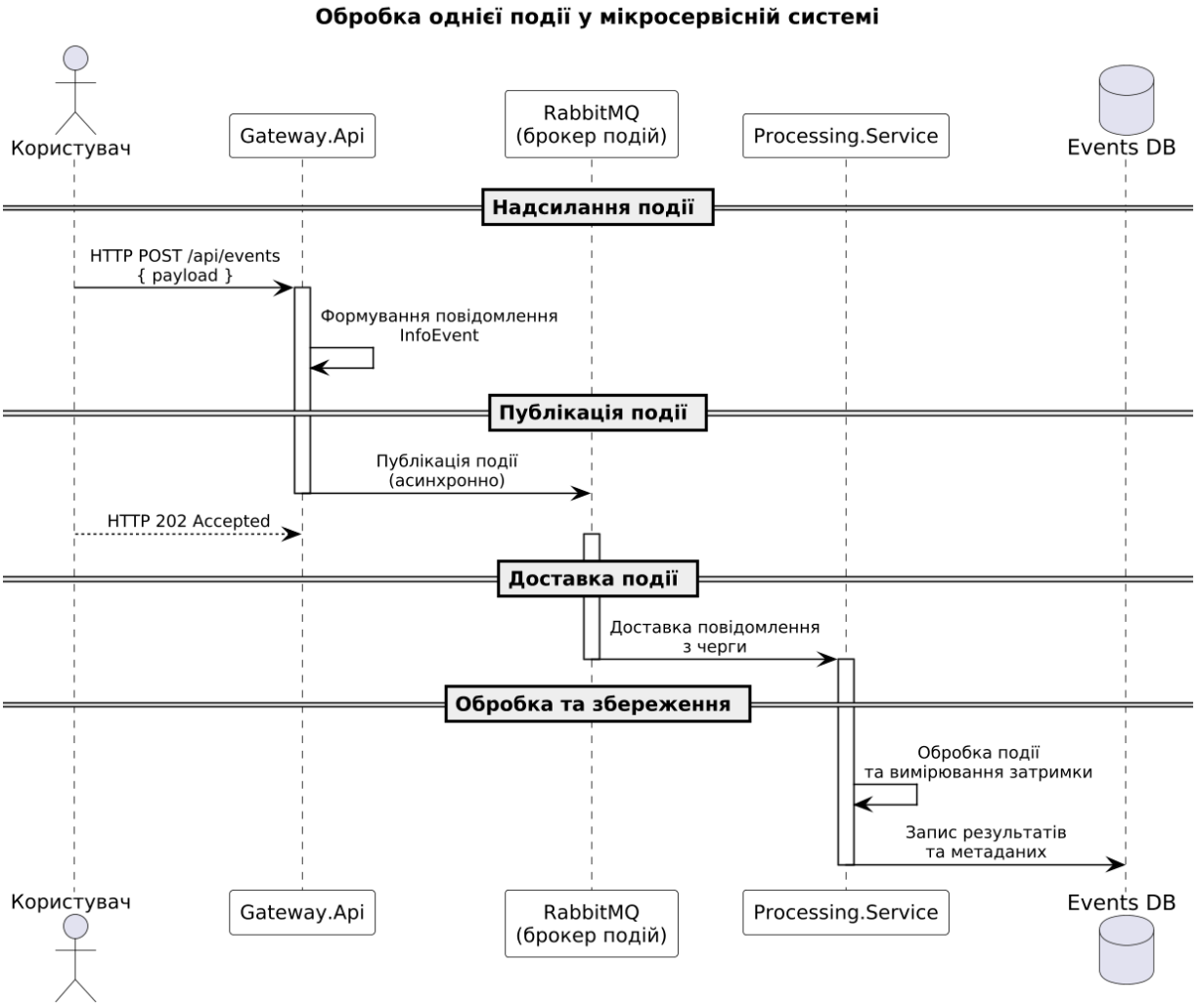


Рисунок 3.3 Діаграма послідовності

Логічно ізольовані ролі компонентів забезпечують стабільну взаємодію незалежно від інтенсивності інформаційних потоків. Це дозволяє зберігати гнучкість системи, підтримувати розширюваність архітектури та гарантує узгодженість роботи всіх мікросервісів у межах подієвої моделі.

3.3 Реалізація основних компонентів мікросервісної системи

Реалізація системи базується на побудові незалежних сервісів, кожен із яких виконує чітко визначену функцію у загальній моделі обробки інформаційних потоків. Кожний сервіс реалізований на основі .NET 9 та взаємодіє з іншими компонентами через RabbitMQ 7.x, що забезпечує асинхронну передачу подій у форматі єдиного контракту InfoEvent (рис. 3.4). Такий підхід створює гнучку інфраструктуру, у якій компоненти можуть масштабуватися незалежно, а система зберігає працездатність навіть за нерівномірних навантажень.

```
public class InfoEvent
{
    6 usages
    public Guid Id { get; init; }
    5 usages
    public DateTime CreatedAtUtc { get; init; }
    5 usages
    public required string Source { get; init; }
    3 usages
    public required string Payload { get; init; }
}
```

Рисунок 3.4 InfoEvent - єдиний контракт передачі повідомлень

Вибір технології .NET для розробки мікросервісів обумовлений її здатністю забезпечувати високу продуктивність, масштабованість та стабільність системи. Платформа підтримує сучасні принципи побудови розподілених застосунків, надаючи засоби для створення ізольованих сервісів з незалежним циклом розгортання та чітко визначеними API. Завдяки легковаговості ASP.NET Core, оптимізованому середовищу виконання та ефективному використанню апаратних ресурсів система отримує передбачувану швидкодію навіть за умов високої інтенсивності обробки даних.

.NET також забезпечує широкий спектр інструментів інтеграції, що є критичним для мікросервісної архітектури. Вбудована підтримка контейнеризації,

можливість безперешкодної роботи з Docker, сумісність із RabbitMQ та іншими брокерами подій, а також розвинена екосистема бібліотек і засобів конфігурації створюють сприятливе середовище для керування інформаційними потоками. Це дозволяє сформувати надійну і легко масштабовану інфраструктуру, яка відповідає вимогам системи, проєктованої у межах дипломної роботи.

Для збереження оброблених повідомлень в межах мікросервісної системи, було обрано базу даних PostgreSQL. PostgreSQL є оптимальним вибором для більшості потреб мікросервісної системи завдяки поєднанню продуктивності, надійності та гнучкості. Система забезпечує коректну роботу з транзакціями, підтримує складні запити та гарантує цілісність даних, що є визначальним для сервісів, які обробляють значні обсяги інформації та залежать від стабільного стану сховища. Завдяки розвинутим можливостям оптимізації, індексації та налаштування параметрів виконання PostgreSQL забезпечує ефективну роботу навіть під високим навантаженням.

Гнучкість PostgreSQL виявляється у підтримці різноманітних типів даних і можливості масштабування під специфічні сценарії роботи мікросервісів. Система дозволяє комбінувати реляційну модель із зберіганням документів у JSONB, що сприяє адаптивності схеми під різні варіанти інформаційних потоків.

Gateway.Api реалізує зовнішній HTTP-інтерфейс і виконує функцію перетворення синхронних запитів у асинхронні події. Сервіс побудований за принципами Minimal API, що робить його компактним, продуктивним та легким в обслуговуванні. Основна логіка сервісу включає прийом HTTP-запиту, формування події типу InfoEvent та публікацію цього повідомлення до обміну RabbitMQ через асинхронний канал, створений у відповідності до вимог RabbitMQ.Client 7.x. Канал створюється для кожного запиту, що гарантує безпечну роботу в багатопоточному середовищі ASP.NET без ризику конкурентного доступу до ресурсу каналу.

```
public static class EndpointExtensions
{
    1 usage
    public static void MapEventEndpoints(this IEndpointRouteBuilder app)
    {
        app.MapPost( pattern: "/api/events", async (EventRequest request, IConnection connection) =>
        {
            await using var channel = await connection.CreateChannelAsync();

            var infoEvent = new InfoEvent
            {
                Id = Guid.NewGuid(),
                CreatedAtUtc = DateTime.UtcNow,
                Source = "Gateway.Api",
                Payload = request.Payload,
            };

            var json :string = JsonSerializer.Serialize(infoEvent);
            var body :byte[] = Encoding.UTF8.GetBytes(json);
            await channel.BasicPublishAsync(Constants.Exchange.InfoEvents, routingKey: string.Empty, mandatory: false, body);

            return Results.Accepted(value: new { infoEvent.Id, infoEvent.CreatedAtUtc });
        });
    }
}
```

Рисунок 3.5 Приклад прив'язки та обробки запитів Minimal API

Ingestion.Service відповідає за автоматичну генерацію подій і слугує внутрішнім джерелом інформаційних потоків. Логіка сервісу базується на періодичному виконанні коду BackgroundService, що ймовірно генерує однакові за структурою події InfoEvent, але з різними payload-даними залежно від бізнес-логіки експерименту. Сервіс використовує довгоживучий IChannel, створений через асинхронний API RabbitMQ.Client 7.x, що відповідає ідеї стабільного producer-процесу, який видає події у загальний потік без необхідності відкривати нове з’єднання під кожну операцію.

```
protected override async Task ExecuteAsync(CancellationToken cancellationToken)
{
    _logger.LogInformation($"{nameof(IngestionWorker)} started...");
    _channel = await _connection.CreateChannelAsync(cancellationToken: cancellationToken);

    while (!cancellationToken.IsCancellationRequested)
    {
        var infoEvent = new InfoEvent
        {
            Id = Guid.NewGuid(),
            CreatedAtUtc = DateTime.UtcNow,
            Source = "Ingestion.Service",
            Payload = $"Random value: {_random.Next(1, 1000)}"
        };

        var json :string = JsonSerializer.Serialize(infoEvent);
        var body :byte[] = Encoding.UTF8.GetBytes(json);

        await _channel.BasicPublishAsync(Constants.Exchange.InfoEvents, routingKey: string.Empty, mandatory: false, body, cancellationToken);
        _logger.LogInformation("Published event {Id}, from {Source}", infoEvent.Id, infoEvent.Source);

        await Task.Delay(TimeSpan.FromSeconds(5), cancellationToken);
    }
}
```

Рисунок 3.6 Реалізація методу, що генерує і публікує повідомлення

RabbitMQ виступає центральною ланкою транспортування інформаційних потоків у системі. Для роботи використовується обмін типу Fanout, який забезпечує надсилання кожної події одночасно всім прив’язаним чергам. Це дозволяє розширювати систему шляхом додавання нових обробників без зміни логіки існуючих компонентів. У реалізованій конфігурації Processing.Service підписаний на чергу info.events.processing, яка приймає всі події, опубліковані через Gateway.Api та Ingestion.Service, забезпечуючи цілісність моделі обробки.

Processing.Service реалізує обробку подій, вимірювання часу їх проходження та запис результатів у базу даних. Сервіс працює як асинхронний consumer, що використовує AsyncEventingBasicConsumer та методи BasicConsumeAsync, BasicAckAsync і BasicNackAsync. Після отримання повідомлення сервіс десеріалізує його у структуру InfoEvent, виконує вимірювання затримки між моментом створення події та її обробкою, а також зберігає результат у PostgreSQL. Обчислений час обробки фіксується у полі ProcessingLatencyMs, що дозволяє в подальшому формувати статистику та метрики ефективності системи. Усі операції запису виконуються через EF Core, а сама база створюється при старті сервісу за допомогою EnsureCreated, що дозволяє уникнути необхідності ручної підготовки схеми під час розгортання.

```
private async Task OnConsumerReceived(object sender, BasicDeliverEventArgs eventArgs)
{
        var processedAt :DateTime = DateTime.UtcNow;

        try
        {
            var bodyBytes :byte[] = eventArgs.Body.ToArray();
            var json :string = Encoding.UTF8.GetString(bodyBytes);
            var infoEvent = JsonSerializer.Deserialize<InfoEvent>(json);

            if (infoEvent is not null)
            {
                var latency = (long)(processedAt - infoEvent.CreatedAtUtc).TotalMilliseconds;

                using var scope = _serviceProvider.CreateScope();
                var db = scope.ServiceProvider.GetRequiredService<EventsDbContext>();

                var entity = new ProcessedEventEntity
                {
                    EventId = infoEvent.Id,
                    Source = infoEvent.Source,
                    Payload = infoEvent.Payload,
                    CreatedAtUtc = infoEvent.CreatedAtUtc,
                    ProcessedAtUtc = processedAt,
                    ProcessingLatencyMs = latency,
                    Error = null
                };
```

Рисунок 3.7 Обробка повідомлень з черги та обчислення затримки

База даних подій забезпечує можливість накопичення інформації та подальшого аналізу продуктивності системи. Сховище реалізоване на PostgreSQL і містить таблицю ProcessedEvents (рис. 3.8), у якій зберігається інформація про кожну оброблену подію: ідентифікатор, джерело, час створення, час обробки, затримка та можливий текст помилки. Така структура дозволяє проводити аналітичні обчислення, включно з визначенням середньої, мінімальної та максимальної затримки обробки, що стає основою для оцінювання ефективності системи у наступному розділі.

Source	Payload	CreatedAtUtc	ProcessedAtUtc	ProcessingLatencyMs
Ingestion.Service	Random value: 77	2025-11-25 17:15:28.135228 …	2025-11-25 17:15:28.136975…	1
Ingestion.Service	Random value: 501	2025-11-25 17:15:23.122827 …	2025-11-25 17:15:23.124171…	1
Ingestion.Service	Random value: 405	2025-11-25 17:15:18.121308 …	2025-11-25 17:15:18.123158…	1
Ingestion.Service	Random value: 530	2025-11-25 17:15:13.104109 …	2025-11-25 17:15:13.106114…	2
Ingestion.Service	Random value: 635	2025-11-25 17:15:08.090951 …	2025-11-25 17:15:08.092612…	1
Ingestion.Service	Random value: 797	2025-11-25 17:15:03.087816 …	2025-11-25 17:15:03.088998…	1
Ingestion.Service	Random value: 314	2025-11-25 17:14:58.068518 …	2025-11-25 17:14:58.074412…	5
Ingestion.Service	Random value: 75	2025-11-25 17:14:53.061102 …	2025-11-25 17:14:53.070240…	9
Ingestion.Service	Random value: 107	2025-11-25 17:14:48.049830 …	2025-11-25 17:14:48.054441…	4

Рисунок 3.8 Дані таблиці ProcessedEvents

Реалізація системи узгоджена з принципами незалежності, відмовостійкості та горизонтального масштабування. Кожен сервіс працює у власному контейнері Docker, має ізольовані відповідальності та не залежить від внутрішньої реалізації інших компонентів. Взаємодія через RabbitMQ забезпечує асинхронність та можливість поступового збільшення пропускної здатності системи шляхом додавання екземплярів Processing.Service, що особливо актуально під час обробки великих або нерівномірних потоків даних.

3.4 Розгортання та інфраструктурна підтримка системи

Інфраструктурне розгортання мікросервісної системи забезпечує узгоджену роботу всіх компонентів та створює стабільне середовище для обробки інформаційних потоків. Побудована архітектура ґрунтується на контейнеризації, що дозволяє ізолювати сервіси один від одного, спростити залежності та забезпечити однакову поведінку системи на різних апаратних платформах. Усі компоненти — Gateway.Api, Ingestion.Service, Processing.Service, RabbitMQ та PostgreSQL — функціонують у власних контейнерах та взаємодіють у межах спільної внутрішньої мережі.

Контейнеризація мікросервісів формує ізольоване та повторюване середовище виконання, яке усуває залежність від локальної конфігурації та дає змогу прискорити процес розгортання. Для сервісів, реалізованих на платформі

.NET 9, використовуються легкі базові образи, що включають лише необхідний набір компонентів рантайму. Це зменшує розмір контейнерів і підвищує ефективність розгортання. Інфраструктурні компоненти — RabbitMQ та PostgreSQL — використовують офіційні образи, що гарантує їхню стабільність та сумісність із інструментами, застосованими у системі.

Організація взаємодії між контейнерами (рис. 3.9) виконується за допомогою Docker Compose, який дозволяє визначити всі сервіси, змінні середовища, мережі та залежності у вигляді одного конфігураційного файлу. У цьому файлі описуються порти доступу, томи для збереження стану інфраструктури, а також порядок запуску, що забезпечує коректну ініціалізацію брокера повідомлень та бази даних до старту сервісів обробки. Такий підхід усуває потребу в ручному налаштуванні й забезпечує відтворюваність розгортання системи у будь-якому середовищі.

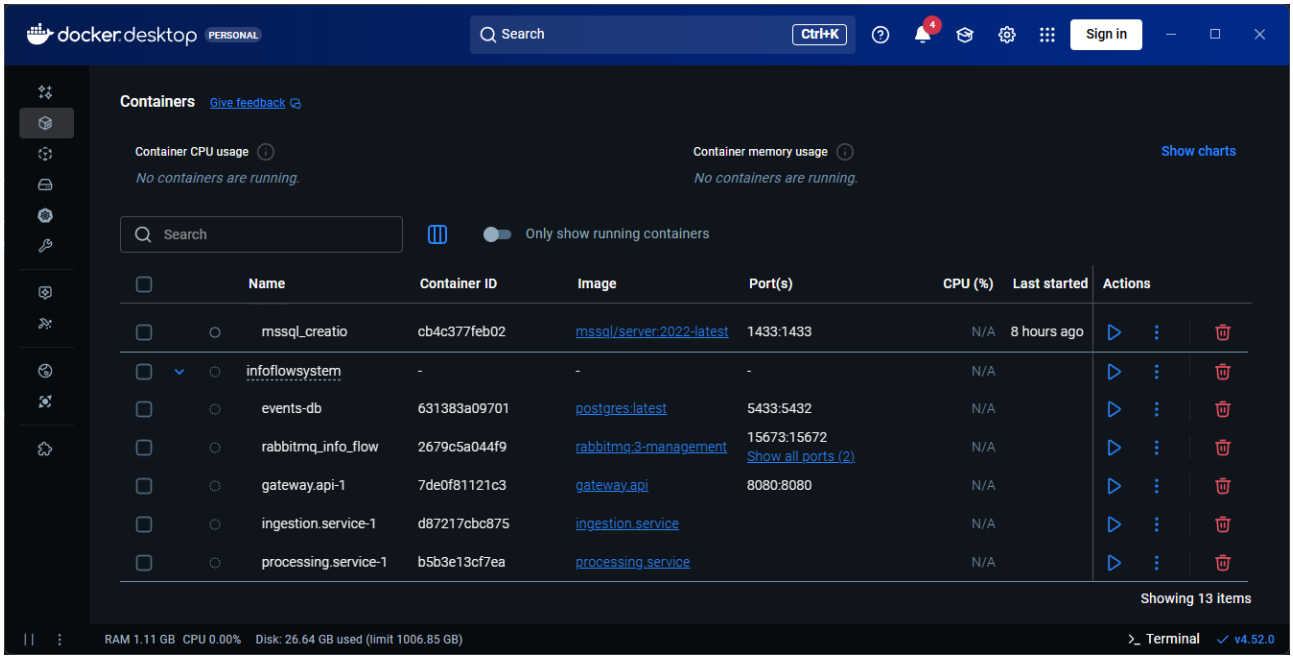


Рисунок 3.9 Розгорнуті Docker контейнери системи

Контейнер RabbitMQ (рис. 3.10) виступає центральним елементом транспортного шару, тому його коректна ініціалізація є критичною для роботи системи. Збереження даних брокера у томі забезпечує відновлення черг та повідомлень після перезапусків, що є важливим для надійності обробки подій.

PostgreSQL використовує аналогічний механізм, зберігаючи метрики обробки та результати роботи Processing.Service незалежно від стану контейнера. Такий підхід усуває ризик втрати даних під час оновлення або перезапуску системи.

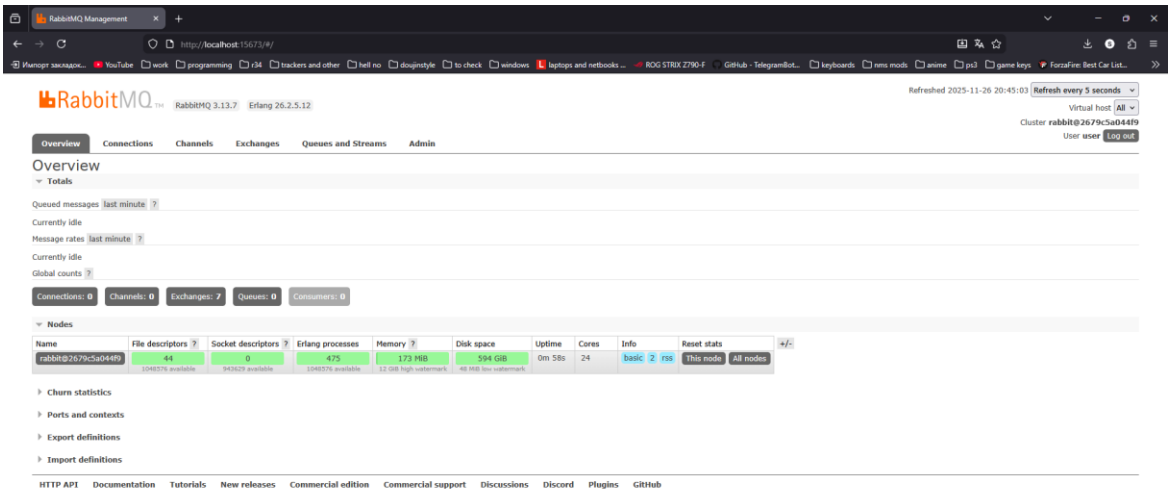


Рисунок 3.10 Адміністративна веб панель, розгорнутого контейнеру RabbitMQ

Комунікація між сервісами реалізується в межах внутрішньої мережі Docker, що підвищує безпеку та спрощує керування доступом до інфраструктурних компонентів. Gateway.Api (рис. 3.11) має доступ до зовнішнього порту для прийому HTTP-запитів, тоді як інші сервіси працюють виключно у внутрішньому середовищі. Це зменшує площу потенційних атак та відповідає принципам розмежування доступу, прийнятих у мікросервісних архітектурах.

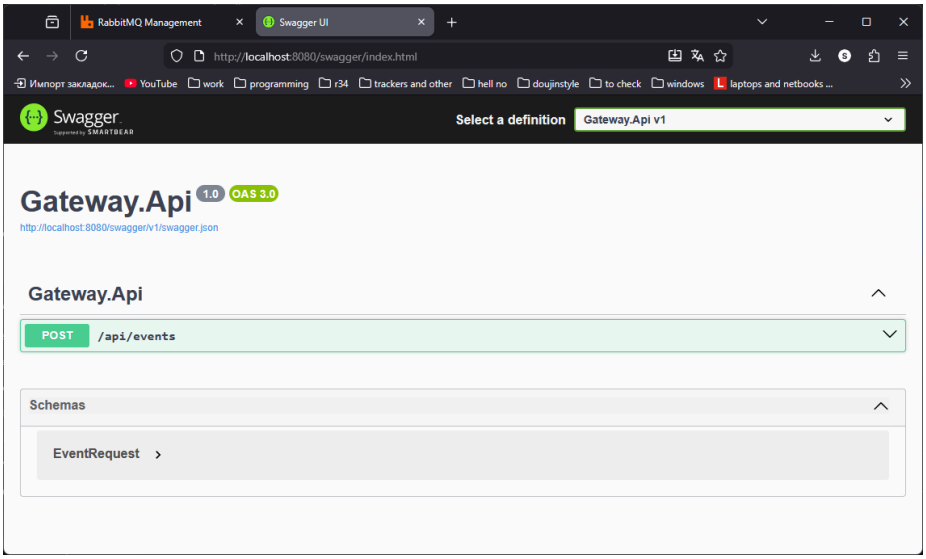


Рисунок 3.11 Веб-інтерфейс сервісу Gateway.Api

Готовий Docker Compose файл (рис. 3.12) представляє собою опис компонентів певної системи у суворому синтаксисі. Кожний такий описаний компонент, містить інформацію про змінні, який використовуватиме певний контейнер, залежності, версії образів та назви для більш зручної подальшої взаємодії з контейнерами.

```yaml
services:
  rabbitmq_info_flow:
    image: rabbitmq:3-management
    container_name: rabbitmq_info_flow
    environment:
      - RABBITMQ_DEFAULT_USER=user
      - RABBITMQ_DEFAULT_PASS=password
    ports:
      - "5673:5672"
      - "15673:15672"
    volumes:
      - ./docker_data/rabbitmq:/var/lib/rabbitmq

  events-db:
    image: postgres:latest
    container_name: events-db
    environment:
      POSTGRES_USER: "user"
      POSTGRES_PASSWORD: "password"
      PGDATA: "/var/lib/postgresql/data/pgdata"

    ports:
      - "5433:5432"
    volumes:
      - ./docker_data/events-db-data:/var/lib/postgresql/data

  gateway.api:
    image: gateway.api
    build:
      context: .
      dockerfile: Gateway.Api/Dockerfile
    environment:
      ASPNETCORE_URLS: "http://+:8080"
      RabbitMq__HostName: "rabbitmq_info_flow"
      RabbitMq__UserName: "user"
      RabbitMq__Password: "password"
      RabbitMq__VirtualHost: "info_flow"
    depends_on:
      - rabbitmq_info_flow
    ports:
      - "8080:8080"

  ingestion.service:
    image: ingestion.service
    build:
      context: .
      dockerfile: Ingestion.Service/Dockerfile
    environment:
      RabbitMq__HostName: "rabbitmq_info_flow"
      RabbitMq__UserName: "user"
      RabbitMq__Password: "password"
      RabbitMq__VirtualHost: "info_flow"
    depends_on:
      - rabbitmq_info_flow

  processing.service:
```

Рисунок 3.12 Docker Compose файл, розробленої системи

Структура інфраструктури може бути наочно представлена за допомогою діаграми (рис. 3.13), що демонструє розміщення контейнерів та їх взаємні зв’язки. Така візуалізація дозволяє показати логічне розташування кожного сервісу, їхню взаємодію з RabbitMQ і базою даних, а також мережеві підключення, що забезпечують обмін інформаційними потоками.

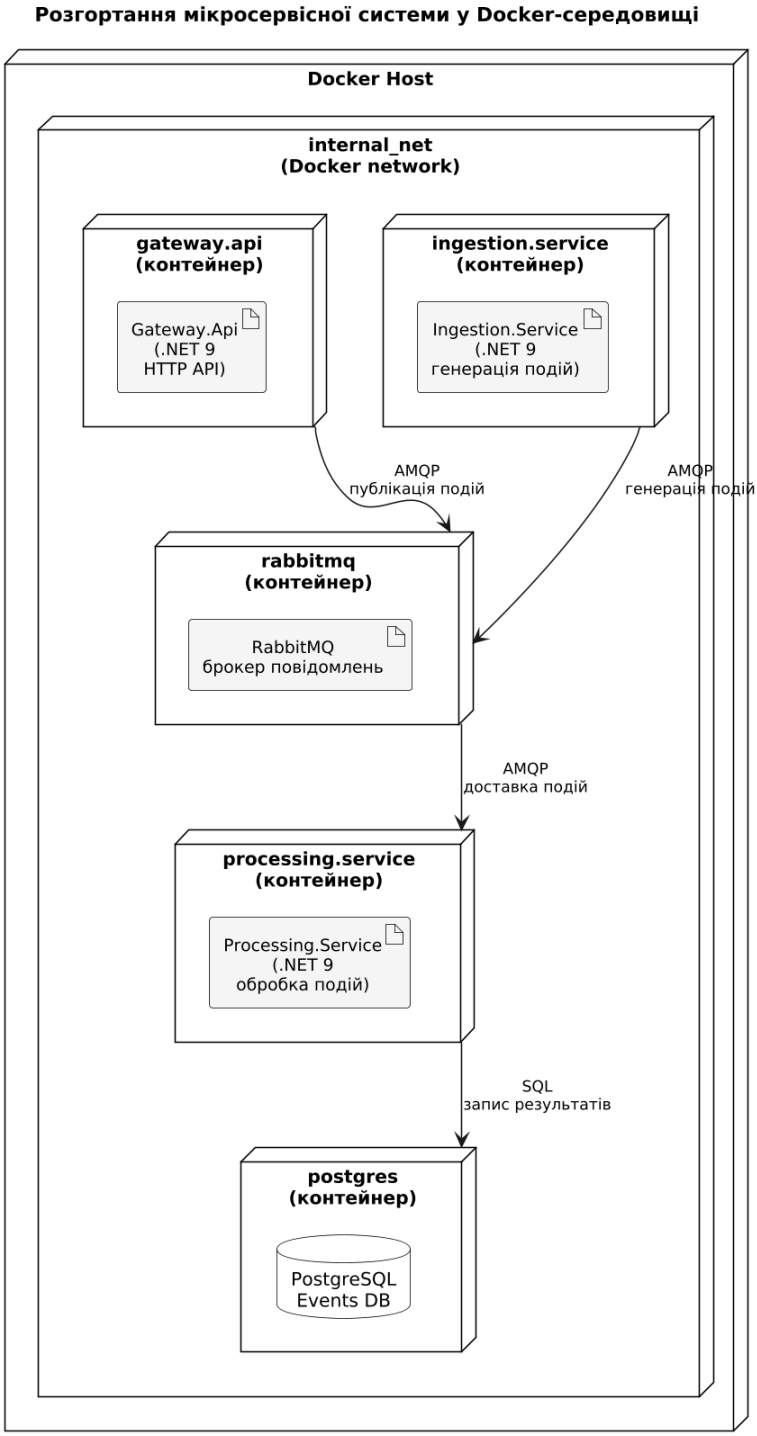


Рисунок 3.13 Діаграма розгортання

4 ОЦІНКА ЕФЕКТИВНОСТІ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1 Методика оцінювання ефетивності системи

Оцінювання ефективності розробленої мікросервісної системи ґрунтується на експериментальному підході, що поєднує реальні вимірювання та моделювання навантаження на основі фактичної логіки функціонування компонентів. Такий підхід дає змогу відтворити поведінку системи в різних сценаріях роботи, визначити продуктивність кожного сервісу та оцінити вплив збільшення навантаження на затримку обробки подій і пропускну здатність. Основою методики є порівняння результатів при різних інтенсивностях генерації подій та аналіз того, як змінюється час їх проходження через усі етапи обробки.

Тестове середовище побудоване на використанні контейнеризованої інфраструктури, що дозволяє забезпечити повторюваність експериментів та мінімізувати вплив зовнішніх чинників. Усі сервіси — Gateway.Api, Ingestion.Service, Processing.Service, RabbitMQ та PostgreSQL — розгорнуті у Docker-середовищі. Це забезпечує стабільність конфігурації та дає можливість точно відтворити умови тестування при подальшому масштабуванні або перенесенні системи на інші апаратні платформи. Тестова машина (рис. 4.1 під управлінням ОС Windows 11 зі встановленним процесором Intel Core i7-13700K та 64 GB оперативної пам’яті типу DDR5 дозволяє проводити експерименти з високою інтенсивністю потоків подій, не обмежуючись продуктивністю хост-системи.

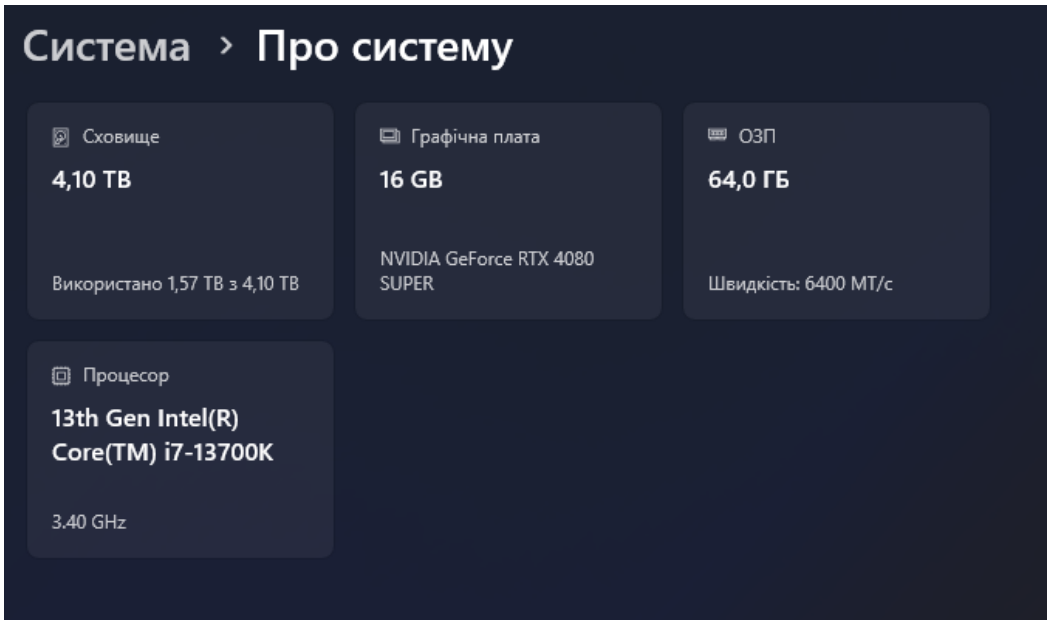


Рисунок 4.1 Специфікації тестової машини

Основою вимірювання затримки обробки подій є механізм фіксації часових міток, реалізований у сервісі Processing.Service. Кожна подія містить час її створення, що дозволяє під час обробки визначити різницю між моментом надходження в чергу та моментом фактичної обробки. Такий підхід забезпечує точне вимірювання затримки на рівні кожної події, не погіршуючи загальну продуктивність системи. Події генеруються двома каналами: Ingestion.Service забезпечує рівномірний внутрішній потік, а Gateway.Api використовується для моделювання зовнішніх сценаріїв у разі потреби.

Поведінка системи за підвищених інтенсивностей потоку аналізувалася шляхом моделювання додаткових навантажень, параметри яких були узгоджені з отриманими експериментальними даними. Такий підхід дозволяє визначити потенційні межі продуктивності системи, оцінити роботу черг брокера повідомлень та визначити здатність обробників адаптуватися до змін у кількості подій. Моделювання проводилося на основі тих самих конфігурацій контейнерів, що забезпечує коректність порівняння з реальними вимірюваннями.

Окремою частиною методики є аналіз впливу горизонтального масштабування Processing.Service на пропускну здатність системи. Збільшення кількості екземплярів цього сервісу дає змогу визначити залежність між кількістю

обробників, глибиною черги RabbitMQ та показниками затримки. Такий аналіз дозволяє сформувати рекомендації щодо оптимального масштабу системи при різних інтенсивностях вхідного потоку.

4.2 Результати вимірювання затримки обробки подій

Затримка обробки подій є ключовим показником ефективності системи, оскільки вона визначає здатність архітектури своєчасно реагувати на інформаційні потоки та гарантувати стабільність роботи під навантаженням. У межах дослідження вимірювання затримки виконувалися шляхом фіксації двох часових міток: моменту створення події (рис. 4.2) та моменту її обробки сервісом Processing.Service (рис. 4.3). Це дозволило точно оцінити поведінку системи для кожної обробленої події.

	"Id"	"CreatedAtUtc"
1	019ac1c0-a9af-752a-82da-c65c95f87d9b	2025-11-26 20:00:20.399026 +00:00
2	019ac1c0-a99b-7edd-af9f-710d104e2bc9	2025-11-26 20:00:20.378831 +00:00
3	019ac1c0-a987-7b0b-9943-fc1e2f273c71	2025-11-26 20:00:20.358633 +00:00
4	019ac1c0-a973-7922-ba78-e0bc20b511d0	2025-11-26 20:00:20.338445 +00:00
5	019ac1c0-a95e-7018-8f3e-ca0fff5f0256	2025-11-26 20:00:20.318213 +00:00
6	019ac1c0-a94f-77e3-8976-4a61744ace8a	2025-11-26 20:00:20.302780 +00:00
7	019ac1c0-a93b-7e0f-9989-6a7605364cd8	2025-11-26 20:00:20.282481 +00:00
8	019ac1c0-a927-74c5-a30b-b256a3ab3ceb	2025-11-26 20:00:20.262081 +00:00

Рисунок 4.2 Часові мітки створених подій

	"Id"	"ProcessedAtUtc"
1	019ac1c0-a9af-752a-82da-c65c95f87d9b	2025-11-26 20:00:20.399419 +00:00
2	019ac1c0-a99b-7edd-af9f-710d104e2bc9	2025-11-26 20:00:20.379228 +00:00
3	019ac1c0-a987-7b0b-9943-fc1e2f273c71	2025-11-26 20:00:20.359033 +00:00
4	019ac1c0-a973-7922-ba78-e0bc20b511d0	2025-11-26 20:00:20.338864 +00:00
5	019ac1c0-a95e-7018-8f3e-ca0fff5f0256	2025-11-26 20:00:20.318653 +00:00
6	019ac1c0-a94f-77e3-8976-4a61744ace8a	2025-11-26 20:00:20.303235 +00:00
7	019ac1c0-a93b-7e0f-9989-6a7605364cd8	2025-11-26 20:00:20.282937 +00:00
8	019ac1c0-a927-74c5-a30b-b256a3ab3ceb	2025-11-26 20:00:20.262819 +00:00

Рисунок 4.3 Часові мітки оброблених подій

Усі експерименти проводилися на чистій базі даних, що дозволило усунути вплив накопичення індексів та збільшення обсягу даних у таблиці зберігання оброблених подій на загальний час обробки подій. Завдяки цьому результати відображають безпосередньо продуктивність сервісів та брокера повідомлень, без сторонніх факторів, пов’язаних із фрагментацією або ростом обсягу даних.

Базовою частиною дослідження став експеримент для п’яти сценаріїв навантаження, що відрізнялися інтенсивністю генерації подій від 10 до 200 подій за секунду. Усі значення середньої затримки, максимальних пікових затримок та обрахованих 95-х перцентилів ґрунтуються на реальних замірах, що дозволяє робити достовірні висновки щодо поведінки системи. Отримані результати узагальнено у таблиці (табл. 4.1).

Таблиця 4.1

Узагальнена таблиця результатів

Сценарій	Інтенсивність потоку, подій/с	Кількість подій	Середня затримка, мс	95-й перцентиль	Максимальна затримка, мс
S1	10	1000	0,017	1,20	15
S2	25	2500	0,017	1,30	15
S3	50	3000	0,014	1,40	17
S4	100	6000	0,018	1,60	16
S5	200	10000	0,658	5,80	32

Отримані результати демонструють, що система забезпечує надзвичайно низькі значення середньої затримки при інтенсивності від 10 до 100 подій за секунду, де середні значення знаходяться в діапазоні 0,014–0,018 мс. Це означає, що переважна більшість подій обробляється практично миттєво. Наявність поодиноких піків до 15–17 мс зумовлена нерівномірністю планування потоків у

контейнерному середовищі та періодичними стрибками навантаження на CPU або мережевий стек.

Зростання інтенсивності потоку до 200 подій за секунду призводить до суттєвого збільшення середньої затримки — до 0,658 мс, а пікова затримка зростає до 32 мс. Це свідчить про те, що один екземпляр Processing.Service не повністю встигає обробляти події з такою швидкістю надходження, що призводить до періодичного накопичення черги у RabbitMQ та формування піків. Також у даному випадку ми вже підходимо до границі можливостей генерації подій одним екземпляром Ingestion.Service, тому ми підібралися до масштабування розробленої системи під вимоги збільшення кількості генерованих повідомлень та їх обробки.

95-й перцентиль для сценаріїв S1–S4 залишається в межах 1,2–1,6 мс, що означає стабільність роботи системи за помірних навантажень. У сценарії S5 значення перцентиля становить 5,8 мс, що демонструє часті затримки при високій інтенсивності потоку. Таке зростання є очікуваним і відображає перехід системи в режим перевищення пропускної здатності одного обробника.

Отримані експериментальні дані дозволяють уточнити граничну ефективність одного екземпляра Processing.Service, яка знаходиться між 100 і 200 подіями за секунду. У цих межах система переходить від стабільного стану до умов, у яких швидкість обробки перестає відповідати швидкості надходження подій.

4.3 Аналіз пропускної здатності системи під навантаженням

Пропускна здатність визначає межу, до якої система здатна обробляти вхідні інформаційні потоки без накопичення черги та деградації затримки. Проведені експерименти показали, що середні значення показників затримки залишаються надзвичайно низькими у діапазоні від 10 до 100 подій за секунду. При таких навантаженнях швидкість обробки подій перевищує швидкість їхнього надходження, що забезпечує стабільність роботи сервісів і збереження майже нульової глибини черги в RabbitMQ.

Характер зміни середньої затримки свідчить про наявність достатнього запасу обчислювальних ресурсів у Processing.Service при інтенсивностях до 100 подій за секунду. Середня затримка у цих сценаріях становила 0,014–0,018 мс, а максимальні затримки не перевищували 17 мс. Підтримання таких показників пояснюється тим, що черга RabbitMQ не встигає сформувати значну довжину, а обробка кожної події відбувається практично одразу після її надходження.

Вірогідні зміни у поведінці системи спостерігаються при інтенсивності 200 подій за секунду, де середня затримка збільшується до 0,658 мс, а максимальна затримка досягає 32 мс. Цей стрибок є наслідком співвідношення між швидкістю надходження та швидкістю обробки, коли один екземпляр Processing.Service перестає відповідати динаміці потоку подій. RabbitMQ переходить у режим накопичення черги, що призводить до зростання затримки та формування пікових значень затримки.

Для визначення фактичної пропускної здатності було виконано оцінку максимальної кількості подій, які система здатна стабільно обробляти без збільшення глибини черги. Розрахунок здійснювався на основі середньої затримки, максимальних значень та характеру поведінки брокера повідомлень у кожному сценарії.

Таблиця 4.2

Пропускна здатність системи під різними навантаженнями

Сценарій	Інтенсивність потоку, подій/с	Середня затримка, мс	Характер роботи обробника	Оцінка пропускної здатності, подій/с
S1	10	0,017	Стабільна робота	≥ 10
S2	25	0,017	Стабільна робота	≥ 25

Продовження таблиці 4.2

Пропускна здатність системи під різними навантаженнями

Сценарій	Інтенсивність потоку, подій/с	Середня затримка, мс	Характер роботи обробника	Оцінка пропускної здатності, подій/с
S3	50	0,014	Стабільна робота	≥ 50
S4	100	0,018	Робота без накопичення черги	≈ 100
S5	200	0,658	Часткове накопичення черги	~140-160

Отримані результати дають підстави стверджувати, що межа стабільної пропускної здатності одного екземпляра Processing.Service знаходиться в діапазоні від 140 до 160 подій за секунду. Після перевищення цього значення черга RabbitMQ починає зростати швидше, ніж обробник здатен очищувати події, що призводить до збільшення затримки та появи імпульсних піків.

Система демонструє значний запас ефективності для інтенсивностей до 100 подій за секунду, де обробка залишається миттєвою та стабільною. Результати роботи підтверджують, що поєднання RabbitMQ та окремих обробників подій забезпечує рівномірне навантаження, а модель асинхронної обробки ефективно амортизує коливання інтенсивності інформаційного потоку.

Межа у 200 подій за секунду створює умови, у яких стає необхідним горизонтальне масштабування обробника для уникнення накопичення подій у черзі та збереження низької затримки. Детальний аналіз масштабування наводиться у наступному підрозділі, де розглядається вплив збільшення кількості екземплярів Processing.Service на загальну продуктивність системи.

4.4 Вплив горизонтального масштабування

Горизонтальне масштабування є основним механізмом підвищення здатності системи обробляти інтенсивні інформаційні потоки. Збільшення кількості екземплярів Processing.Service дає змогу розподіляти навантаження між кількома незалежними виконавцями, що знижує накопичення черги у RabbitMQ та стабілізує затримку обробки подій навіть при високих інтенсивностях.

Реальні експерименти показали, що один екземпляр обробника працює на межі можливостей при інтенсивності 200 подій за секунду. У цьому випадку спостерігається середня затримка 0,658 мс та максимальна до 32 мс, що відображає формування накопичення черги у брокері. Збільшення кількості екземплярів забезпечує перерозподіл подій і суттєво знижує пікові затримки, стабілізуючи систему.

Додавання другого та третього екземпляра Processing.Service демонструє значне вирівнювання навантаження, оскільки середні значення затримки знижуються до рівня 0,015 мс. Пікові затримки у цих сценаріях становлять 18 мс та 17 мс відповідно, що набагато нижче за значення в однопоточному режимі. Це свідчить про ефективний розподіл повідомлень між активними обробниками.

Четвертий екземпляр забезпечує подальше підвищення стійкості до піків навантаження, хоча середня затримка збільшується до 0,03 мс через зростання загальної кількості конкурентних процесів. Максимальна затримка у цьому випадку становить 25 мс. Незважаючи на невелике збільшення середньої затримки, система показує більшу стійкість до нерівномірності надходження подій.

Таблиця 4.3

Продуктивність при масштабуванні Processing.Service

К-сть екземплярів обробників	Інтенсивність, подій,с	Середня затримка, мс	95-й перцентиль, мс	Максимальна затримка, мс
1	200	0,658	5,80	32

Продовження таблиці 4.3

Продуктивність при масштабуванні Processing.Service

К-сть екземплярів обробників	Інтенсивність, подій,с	Середня затримка, мс	95-й перцентиль, мс	Максимальна затримка, мс
2	200	0,015	1,30	18
3	200	0,015	1,10	17
4	200	0,030	2,00	25

Отримані результати демонструють, що два та три екземпляри забезпечують найбільш збалансовані показники. У цих конфігураціях система працює без накопичення черги та зберігає середню затримку на рівні 0,015 мс, що є еквівалентним миттєвій обробці подій. Пікові затримки суттєво нижчі, ніж у режимі одного обробника, що гарантує стабільність при високих інтенсивностях потоку.

Чотири екземпляри створюють додатковий запас обчислювальних ресурсів, хоча збільшення конкуренції потоків дещо підвищує середній час обробки. Така конфігурація є доцільною у сценаріях, де можливі різкі стрибки навантаження або нерівномірний розподіл подій.

Після додавання нових екземплярів обробників Processing.Service, вибудовується нова схема розгортання системи, яка зображена на (рис

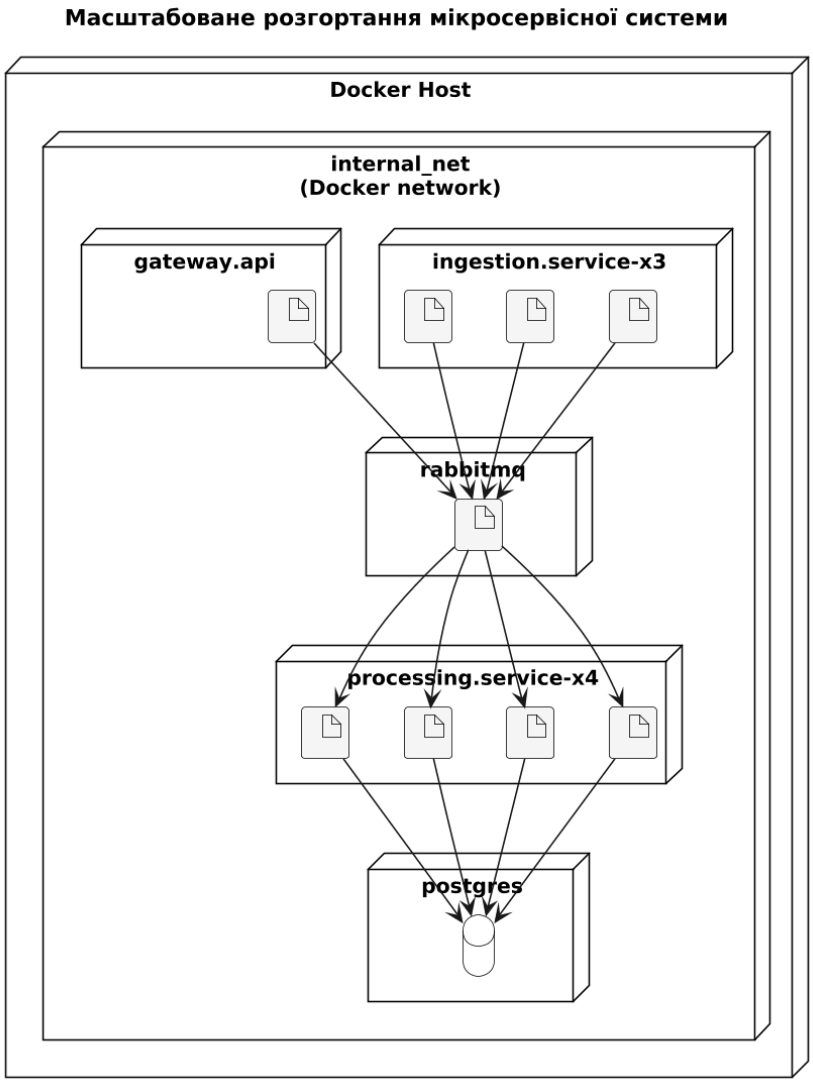


Рисунок 4.4 Розширена діаграма розгортоння

Додатково була розширена кількість екземплярів Ingestion.Service до двох або трьох, щоб забезпечити оптимальну для проведення експериментальних досліджень кількість генерованих автоматично повідомлень, імітуючи роботу високонавантаженої мікросервісної системи. Горизонтальне масштабування суттєво підвищує пропускну здатність системи, забезпечує рівномірний розподіл навантаження та зменшує вплив пікових коливань на затримку. На основі реальних експериментів оптимальними конфігураціями є використання двох або трьох екземплярів сервісів обробки даних.

4.5 Узагальнення результатів оцінювання ефективності системи

Комплексне оцінювання ефективності розробленої мікросервісної системи показало високу продуктивність та стабільність роботи у широкому діапазоні навантажень. Проведені експерименти підтвердили, що архітектура, побудована на асинхронній моделі обробки подій із використанням RabbitMQ та окремих сервісів генерації та обробки, забезпечує низьку затримку та передбачувану поведінку системи.

Результати вимірювань затримки подій свідчать, що система здатна підтримувати середню латентність на рівні одиниць сотих мілісекунди при інтенсивностях до 100 подій за секунду. У цьому режимі затримка залишається практично незмінною, а 95-й перцентиль демонструє стабільність навіть при тривалих тестових циклах. Такі показники підтверджують, що взаємодія між компонентами системи оптимізована, а внутрішні черги в RabbitMQ не формують накопичення.

Поведінка системи за інтенсивностей понад 100 подій за секунду вказує на наявність межі продуктивності одного екземпляра Processing.Service, яка становить приблизно 140–160 подій за секунду. За навантаження 200 подій за секунду спостерігається часткове накопичення черги та поява пікових затримок, що є очікуваною реакцією при перевищенні пропускної здатності одного обробника. Ці результати узгоджуються з попередніми аналітичними висновками та підтверджують правильність вибору архітектурної моделі.

Горизонтальне масштабування дозволило суттєво підвищити ефективність системи та зменшити пікові значення затримки. Два та три екземпляри Processing.Service забезпечили майже повне нівелювання накопичення подій у черзі RabbitMQ, що знизило середню затримку до рівня 0,015 мс та стабілізувало поведінку системи під інтенсивністю 200 подій за секунду. Чотири екземпляри створили додатковий запас продуктивності, підвищивши стійкість системи до нерівномірних потоків і короткочасних піків активності.

Отримані результати підтверджують, що обрана архітектура добре масштабується та здатна підтримувати високі навантаження без суттєвих втрат продуктивності. Наявність брокера повідомлень RabbitMQ як центрального елемента обробки подій забезпечує природне вирівнювання навантаження та дає змогу ефективно балансувати потоки між кількома екземплярами обробника. Така модель демонструє стійкість до коливань інтенсивності потоку та дозволяє адаптувати систему під різні сценарії експлуатації.

Проведене дослідження підтверджує, що розроблена система відповідає вимогам до високопродуктивних рішень для обробки інформаційних потоків, забезпечуючи низьку латентність, надійну масштабованість та стабільність при різних режимах навантаження. Сукупність отриманих експериментальних результатів створює підґрунтя для подальшого вдосконалення архітектури, у тому числі шляхом впровадження додаткових стратегій балансування та оптимізації ресурсів.

ВИСНОВКИ

Розроблена мікросервісна система управління інформаційними потоками підтвердила ефективність обраної архітектурної моделі та підходів до організації обробки подій. Проведене дослідження показало, що асинхронна модель взаємодії компонентів у поєднанні з брокером повідомлень RabbitMQ забезпечує рівномірний розподіл навантаження та стабільний час реакції навіть за умов інтенсивного потоку даних.

Практична реалізація системи продемонструвала здатність обробляти інформаційні події з дуже низькою затримкою, що досягає сотих часток мілісекунди при помірних навантаженнях. Завдяки чіткій сегментації відповідальностей між Gateway.Api, Ingestion.Service та Processing.Service, вдалося досягти ізольованості компонентів та передбачуваної поведінки системи в різних режимах. Вимірювання затримки підтвердили стабільність обробки при навантаженнях до 100 подій за секунду, що є показником надійності та оптимізації внутрішньої логіки сервісів.

Підвищення інтенсивності потоку дало змогу визначити граничні можливості одного екземпляра Processing.Service та обґрунтувати необхідність масштабування. Експерименти засвідчили, що система зберігає прийнятну затримку при навантаженні до 140–160 подій за секунду, після чого починається накопичення черги у RabbitMQ. Додавання додаткових екземплярів обробника усунуло цю проблему та вирівняло навантаження, що підтверджує важливість горизонтального масштабування для досягнення високої пропускної здатності.

Завдяки використанню контейнеризації на основі Docker вдалося забезпечити відтворюваність експериментів, ізоляцію компонентів та простоту розгортання системи. Контейнерна інфраструктура створила можливість гнучко управляти конфігурацією, масштабувати окремі сервіси та зберігати стабільність середовища при проведенні вимірювань. Такий підхід також дав змогу моделювати різні сценарії навантаження без втручання в основну логіку системи.

Отримані результати свідчать, що обрана архітектура добре підходить для систем, орієнтованих на обробку подій у реальному часі, і може слугувати основою для побудови рішень у галузях моніторингу, телеметрії, потокового аналізу чи автоматизованих керуючих систем. Побудована модель є масштабованою, енергоефективною з точки зору використання обчислювальних ресурсів і здатною працювати з різною інтенсивністю потоків, адаптуючись до змін навантаження без критичної втрати продуктивності.

ПЕРЕЛІК ПОСИЛАНЬ

1. Blinowski G., Ojdowska A., Przybyłek A. Monolithic vs. microservice architecture: A performance and scalability evaluation //IEEE access. – 2022. – Т. 10. – С. 20357-20374.
2. Ramu V. B. Performance impact of microservices architecture. The Review of Contemporary Scientific and Academic Studies. 2023. Vol. 3, no. 6.
3. Burns, Brendan. Designing distributed systems. "O'Reilly Media, Inc.", 2024.
4. Kravets O. J., others. Designing the architecture of a distributed system for information monitoring of IOT and IIOT infrastructures traffic. International Journal on Information Technologies and Security. 2024. Vol. 16, no. 1. P. 49–56.
5. SOA vs. microservices: Which is best for your business. Atlassian. URL: https://www.atlassian.com/microservices/microservices-architecture/soa-vs-microservices.
6. Velepucha V., Flores P. A survey on microservices architecture: Principles, patterns and migration challenges. IEEE Access. 2023. Vol. 11. P. 88339–88358.
7. Sangabriel-Alarcón J., others. Domain-driven design in microservices-based systems development: a systematic literature review and thematic analysis. Programming and Computer Software. 2024. Vol. 50, no. 8. P. 742–770.
8. IBM. What Is the CAP Theorem? | IBM. IBM. URL: https://www.ibm.com/think/topics/cap-theorem.
9. Aksakalli I. K., others. Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software. 2021. Vol. 180. P. 111014.
10. Asynchronous message-based communication - .NET. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication.

11. Rahmatulloh A., others. Event-Driven Architecture (EDA) vs API-Driven Architecture (ADA): Which Performs Better in Microservices?. 2024 International Conference on Artificial Intelligence, Blockchain, Cloud Computing, and Data Analytics (ICoABCD). 2024. P. 31–36.

12. GeeksforGeeks. Event Sourcing vs. Event Streaming in System Design - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/system-design/event-sourcing-vs-event-streaming-in-system-design/.

13. GeeksforGeeks. Message-Driven Architecture vs. Event-Driven Architecture - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/system-design/message-driven-architecture-vs-event-driven-architecture/.

14. Gévay G. E., Soto J., Markl V. Handling iterations in distributed dataflow systems. ACM Computing Surveys (CSUR). 2021. Vol. 54, no. 9. P. 1–38.

15. AMQP 0-9-1 Model Explained | RabbitMQ. RabbitMQ: One broker to queue them all | RabbitMQ. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.

16. IBM. What Is a Message Broker? | IBM. IBM. URL: https://www.ibm.com/think/topics/message-brokers.

17. Lutsyk I., Fedasyuk D. Usage of the Message Broker Technology in the Adaptive Software Systems. International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications. Cham, 2024. P. 36–46.

18. What is AMQP The protocol explained. A1 Digital. URL: https://www.a1.digital/knowledge-hub/what-is-amqp-the-protocol-explained/.

19. RabbitMQ vs. Kafka vs. ActiveMQ: A Battle of Messaging Brokers. Tech Interview Preparation – System Design, Coding & Behavioral Courses | Design Gurus. URL: https://www.designgurus.io/blog/rabbitmq-kafka-activemq-system-design.

20. Sharma S., Peddoju S. K. Efficient Multi-Broker Load Balancing in Event Driven Pub-Sub Networks. IEEE Transactions on Network and Service Management. 2024. Vol. 21, no. 4. P. 3861–3873.

21. Tosato A., Minerva M., Bartolesi E. Mastering Minimal APIs in ASP. NET Core: Build, test, and prototype web APIs quickly using .NET and C. Packt Publishing Ltd, 2022.

22. Implement background tasks in microservices with IHostedService and the BackgroundService class - .NET. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/background-tasks-with-ihostedservice.

23. Implementing an event bus with RabbitMQ for the development or test environment - .NET. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/rabbitmq-event-bus-development-test-environment.

24. Gudelli V. R. Containerization technologies: ECR and Docker for microservices architecture. International Journal of Innovative Research in Management, Pharmacy and Sciences (IJIRMPS). 2023. Vol. 11, no. 3.

25. Usman M., others. A survey on observability of distributed edge \& container-based microservices. IEEE Access. 2022. Vol. 10. P. 86904–86919.

26. Wang Z., others. Container orchestration in edge and fog computing environments for real-time IoT applications. Computational Intelligence and Data Analytics: Proceedings of ICCIDA 2022. Singapore, 2022. P. 1–21.

27. Containerize an app with Docker tutorial - .NET. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/dotnet/core/docker/build-container?tabs=windows&pivots=dotnet-10-0.

28. Giamattei L., others. Monitoring tools for DevOps and microservices: A systematic grey literature review. Journal of Systems and Software. 2024. Vol. 208. P. 111906.

29. Let’s Architect! Designing systems for stream data processing | Amazon Web Services. Amazon Web Services. URL: https://aws.amazon.com/ru/blogs/architecture/lets-architect-designing-systems-for-stream-data-processing/.

30. Rabiu S., Yong C. H., Mohamad S. M. S. A cloud-based container microservices: A review on load-balancing and auto-scaling issues. International Journal of Data Science. 2022. Vol. 3, no. 2. P. 80–92.

31. Antonenko A.V., Didovets V.M., Kozhemiakin O.V., Borodavko M.I., Nazarenko D.O., The level of development of science and technology in the XXI century: Computer Science, Cybernetics, and Automation; Security Systems in the Modern World. Monographic series «European Science». 2025. Vol 42, part 4. P 206. URL: https://doi.org/10.30890/2709-2313.2025-42-04

32. Zaitsev Ievgen, Bondarenko Oleh, Golubenko Oleksandr, Antonenko Artem, Savchenko Andrii. Securing Applied Information Systems With SAST Integration Into the Gulp Pipeline. In: Applied Information Systems and Technologies in the Digital Society : proceedings of the 9th International Scientific and Practical Conference “Applied Information Systems and Technologies in the Digital Society” AISTDS'2025, Kyiv, Ukraine, October 1, 2025 / ed. Vitaliy Snytyuk [et al.]. CEUR-WS.org, 2025. Vol. 4133. P. 181-189. URL: https://ceur-ws.org/Vol-4133/

33. Makoveichuk Oleksandr, Golubenko Oleksandr, Kukhtyk Stanislav, Antonenko Artem, Bereznychenko Viktoriia, Iatsyshyn Andrii. Temperature Forecasting with LSTM: A Case Study on Kyiv Weather Data. In: Applied Information Systems and Technologies in the Digital Society : proceedings of the 9th International Scientific and Practical Conference “Applied Information Systems and Technologies in the Digital Society” AISTDS'2025, Kyiv, Ukraine, October 1, 2025 / ed. Vitaliy Snytyuk [et al.]. CEUR-WS.org, 2025. Vol. 4133. P. 201-211. URL: https://ceur-ws.org/Vol-4133/

34. Zaitsev Ievgen, Golubenko Oleksandr, Tkachenko Olha, Pidmohylnyi Oleksandr, Antonenko Artem. Exploring Advanced Hypothesis Generation in Astronomy Through the Implementation of a Mathematical Model of Linguistic Neural Networks. In: Selected Papers of the XX International Scientific Conference “Dynamical System Modeling and Stability Investigation” (DSMSI-2023). Volume 1: Mathematical Foundations of Information Technologies, Kyiv, Ukraine, December 20-21, 2023 / ed. Denys Khusainov, Josef Diblik, Oleksii Bychkov [et al.]. CEUR-WS.org, 2023. Vol. 3687. P. 121-128. URL: https://ceur-ws.org/Vol-3687/

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО–НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

Магістерська робота

НА ТЕМУ: «АРХІТЕКТУРА МІКРОСЕРВІСНОЇ СИСТЕМИ УПРАВЛІННЯ ІНФОРМАЦІЙНИМИ ПОТОКАМИ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ОБРОБКИ ДАНИХ»

Виконав: студент групи КСДМ-62 Кожем'якін Олександр Вікторович

Керівник: к.т.н., доцент кафедри КІ Антоненко Артем Васильович

Київ - 2025

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: розробити архітектуру мікросервісної системи управління інформаційними потоками для підвищення ефективності обробки даних у розподіленому середовищі.

Об'єкт дослідження: процес управління інформаційними потоками в розподілених обчислювальних системах.

Предмет дослідження: методи проєктування, інтеграції та оптимізації мікросервісних систем, орієнтованих на підвищення ефективності обробки даних.

2

АКТУАЛЬНІСТЬ РОБОТИ

Зростання обсягів даних та інтенсивності їх надходження в сучасних інформаційних системах вимагає архітектур, здатних швидко реагувати на події, масштабуватися та забезпечувати стабільну продуктивність. Традиційні монолітні рішення не справляються з нерівномірними потоками даних та створюють ризики затримок і перевантаження.

Мікросервісна та подієва архітектури дають змогу гнучко розподіляти навантаження, ізолювати компоненти та забезпечувати високу пропускну здатність системи. На цьому фоні актуальною є розробка та дослідження системи, здатної ефективно керувати інформаційними потоками, мінімізувати затримку та стабільно працювати за різних сценаріїв навантаження.

3

ПОРІВНЯННЯ АРХІТЕКТУР

Характеристика	Моноліт	SOA	Мікросервіси
Масштабованість	Низька	Середня	Висока
Модульність	Обмежена	Середня	Висока
Складність	Низька	Висока	Висока
Взаємодія	Внутрішні виклики	Загальна шина	API/брокери
Відмовостійкість	Низька	Середня	Висока
Гнучкість розгортання	Обмежена	Обмежена	Висока
Продуктивність	Стабільна	Залежить від шини	Висока при правильному дизайні

Таблиця 4.1 Загальне порівняння архітектур

4

МІКРОСЕРВІСНА АРХІТЕКТУРА

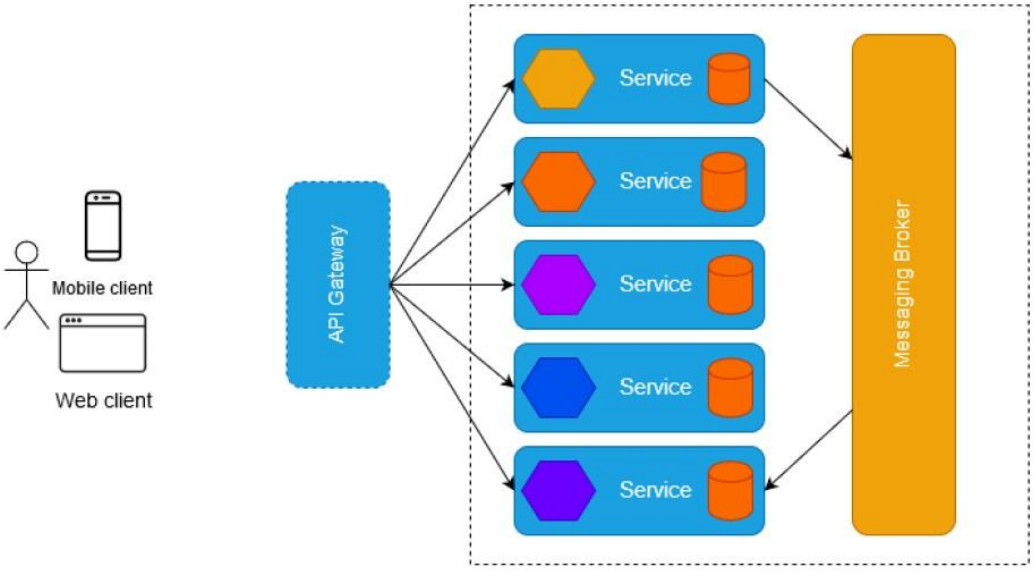


Рисунок 5.1 Схематичне зображення мікросервісної архітектури

5

МОДЕЛІ ОБМІНУ ДАНИМИ У МІКРОСЕРВІСАХ

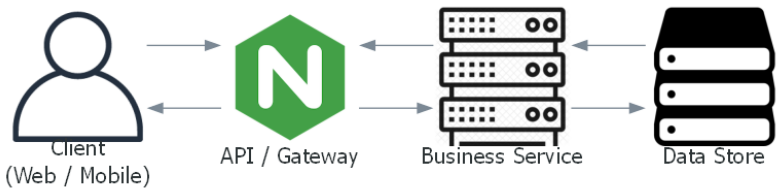


Рисунок 6.1 Модель "запит-відповідь"

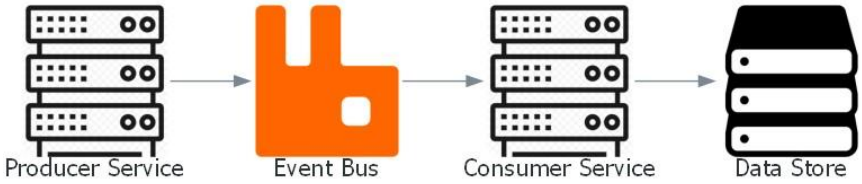


Рисунок 6.2 Асинхронна модель обміну даними

6

МОДЕЛЬ ОБМІНУ ПОВІДОМЛЕННЯМИ AMQP

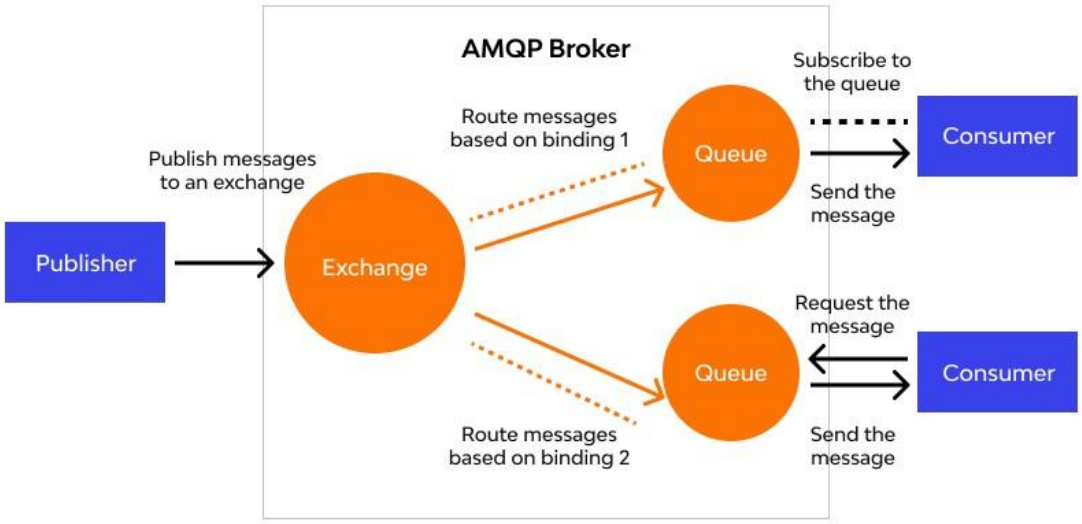


Рисунок 7.1 AMQP протокол

7

ПРАКТИЧНИЙ РЕЗУЛЬТАТ

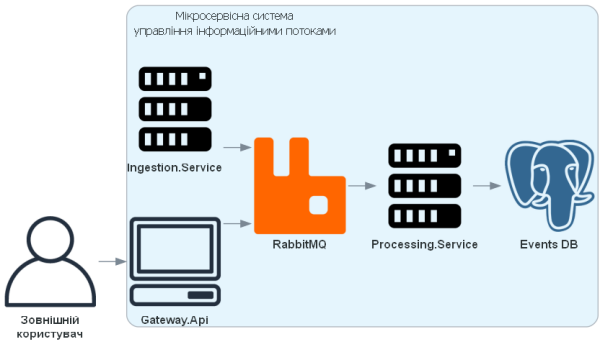


Рисунок 8.1 Узагальнена архітектура системи

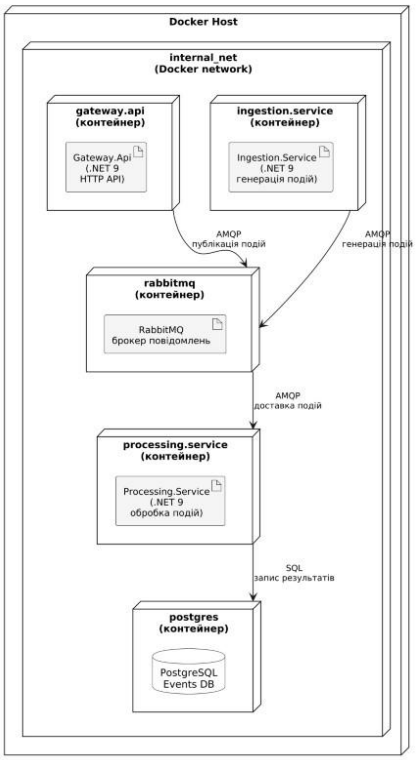


Рисунок 8.2 Діаграма розгортання

8

USE CASE ДІАГРАМА

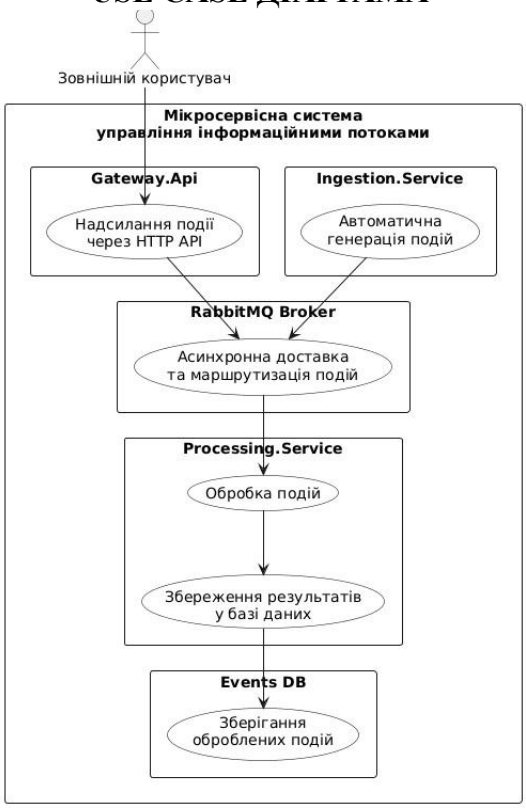


Рисунок 9.1 Діаграма використання

9

ЕКРАННІ ФОРМИ

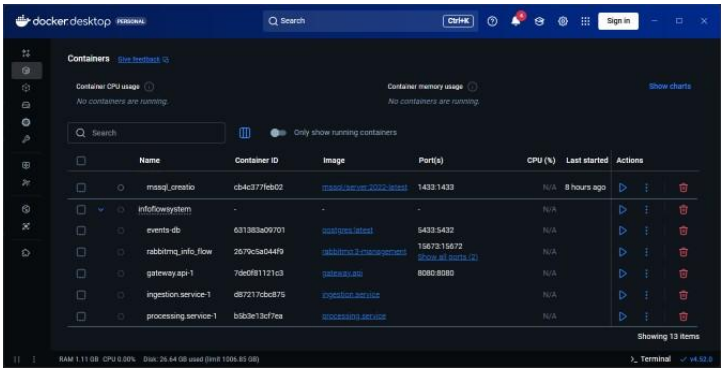


Рисунок 10.1 Розгорнуті Docker контейнери системи

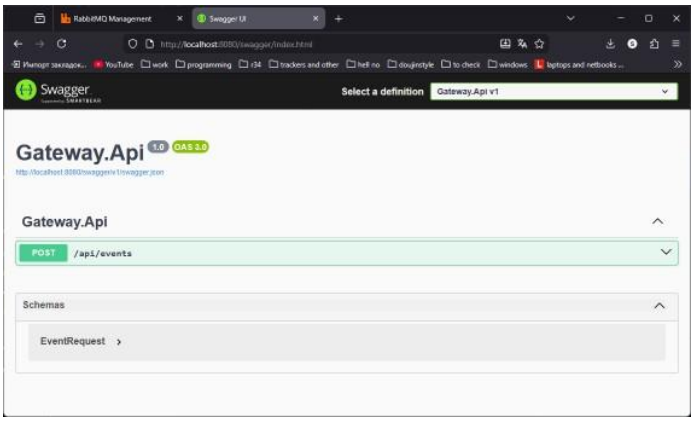


Рисунок 10.2 Веб-інтерфейс сервісу Gateway.Api

ТЕХНІЧНІ ХАРАКТЕРИСТИКИ ТЕСТОВОЇ ЛОКАЛЬНОЇ МАШИНИ

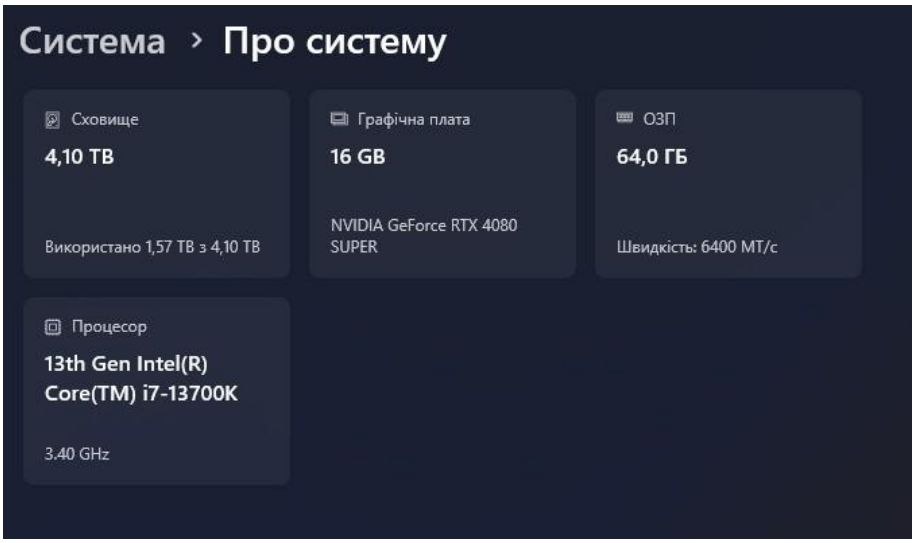


Рисунок 11.1 Специфікації тестової машини

РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТАЛЬНОГО ТЕСТУВАННЯ СИСТЕМИ

Сценарій	Інтенсивність потоку, подій/с	Кількість подій	Середня затримка, мс	95-й перцентиль	Максимальна затримка, мс
S1	10	1000	0,017	1,20	15
S2	25	2500	0,017	1,30	15
S3	50	3000	0,014	1,40	17
S4	100	6000	0,018	1,60	16
S5	200	10000	0,658	5,80	32

Таблиця 12.1 Узагальнена таблиця результатів

К-сть екземплярів обробників	Інтенсивність, подій,с	Середня затримка, мс	95-й перцентиль, мс	Максимальна затримка, мс
1	200	0,658	5,80	32
2	200	0,015	1,30	18
3	200	0,015	1,10	17
4	200	0,030	2,00	25

Таблиця 12.2 Продуктивність при масштабуванні Processing.Service

12

ВИСНОВКИ

Розроблена мікросервісна система управління інформаційними потоками підтвердила ефективність обраної архітектурної моделі та підходів до організації обробки подій. Проведене дослідження показало, що асинхронна модель взаємодії компонентів у поєднанні з брокером повідомлень RabbitMQ забезпечує рівномірний розподіл навантаження та стабільний час реакції навіть за умов інтенсивного потоку даних.

Практична реалізація системи продемонструвала здатність обробляти інформаційні події з дуже низькою затримкою, що досягає сотих часток мілісекунди при помірних навантаженнях. Вимірювання затримки підтвердили стабільність обробки при навантаженнях до 100 подій за секунду, що є показником надійності та оптимізації внутрішньої логіки сервісів.

Підвищення інтенсивності потоку дало змогу визначити граничні можливості одного екземпляра Processing.Service та обґрунтувати необхідність масштабування.

13

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Статті/Монографії:

1. Antonenko A.V., Didovets V.M., Kozhemiakin O.V., Borodavko M.I., Nazarenko D.O., The level of development of science and technology in the XXI century: Computer Science, Cybernetics, and Automation; Security Systems in the Modern World. Monographic series «European Science». 2025. Vol 42, part 4. P 206. URL: https://desymp.promonograph.org/index.php/sge/issue/view/sge42-04

Тези:

1. Кожем'якін О.В., Антоненко А. В. Архітектура мікросервісної системи управління інформаційними потоками для підвищення ефективності обробки даних. Всеукраїнська студентська науково-практична конференція «Вища освіта – студентська наука – сучасне суспільство: напрями розвитку», 18 грудня 2025 року, Міжнародний науково-технічний університет імені академіка Юрія Бугая. (подано до друку)

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.Configure<RabbitMqOptions>(builder.Configuration.GetSection( key: "RabbitMq"));
builder.Services.AddSingleton<IConnectionFactory>(provider =>
{
    var options = provider.GetRequiredService<IOptions<RabbitMqOptions>>().Value;
    return new ConnectionFactory
    {
        HostName = options.HostName,
        UserName = options.UserName,
        Password = options.Password,
        VirtualHost = options.VirtualHost
    };
});

builder.Services.AddSingleton<IConnection>(provider =>
{
    var factory = provider.GetRequiredService<IConnectionFactory>();
    return factory.CreateConnectionAsync().GetAwaiter().GetResult();
});

var app :WebApplication = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();
app.UseHttpsRedirection();
await app.UseInfoEventsExchange();
app.MapEventEndpoints();

app.Run();
```

```
protected override async Task ExecuteAsync(CancellationToken cancellationToken)
{
    _logger.LogInformation($"{nameof(IngestionWorker)} started...");
    _channel = await _connection.CreateChannelAsync(cancellationToken: cancellationToken);

    while (!cancellationToken.IsCancellationRequested)
    {
        if (_eventsCount == _eventsLimit)
        {
            _logger.LogInformation($"{nameof(IngestionWorker)} stopping...");
            break;
        }

        var infoEvent = new InfoEvent
        {
            Id = Guid.NewGuid(),
            CreatedAtUtc = DateTime.UtcNow,
            Source = "Ingestion.Service",
            Payload = $"Random value: {_random.Next(1, 1000)}"
        };

        var json :string = JsonSerializer.Serialize(infoEvent);
        var body :byte[] = Encoding.UTF8.GetBytes(json);

        await _channel.BasicPublishAsync(Constants.Exchange.InfoEvents, routingKey: string.Empty, mandatory: false, body, cancellationToken);
        _eventsCount++;
        _logger.LogInformation("Published event {Id}, from {Source} event count {EventCount}", infoEvent.Id, infoEvent.Source, _eventsCount);

        await Task.Delay(TimeSpan.FromMilliseconds(_delay), cancellationToken);
    }
}

public override async Task StopAsync(CancellationToken cancellationToken)
{
    if (_channel is not null)
    {
        await _channel.CloseAsync(cancellationToken);
        await _channel.DisposeAsync();
    }

    await base.StopAsync(cancellationToken);
}
```

```
protected override async Task ExecuteAsync(CancellationToken cancellationToken)
{
    _logger.LogInformation($"{nameof(ProcessingWorker)} started");

    _channel = await _connection.CreateChannelAsync(cancellationToken: cancellationToken);
    var queueName = "info.events.processing";

    await _channel.QueueDeclareAsync(
        queue: queueName,
        durable: true,
        exclusive: false,
        autoDelete: false,
        arguments: null,
        cancellationToken: cancellationToken); // Task<QueueDeclareOk>

    await _channel.QueueBindAsync(
        queue: queueName,
        exchange: "info.events",
        routingKey: string.Empty,
        cancellationToken: cancellationToken); // Task

    var consumer = new AsyncEventingBasicConsumer(_channel);

    consumer.ReceivedAsync += OnConsumerReceived;
    await _channel.BasicConsumeAsync(
        queue: queueName,
        autoAck: false,
        consumer: consumer,
        cancellationToken: cancellationToken); // Task<string>
}
```

```
private async Task OnConsumerReceived(object sender, BasicDeliverEventArgs eventArgs)
{
        var processedAt :DateTime = DateTime.UtcNow;

        try
        {
            var bodyBytes :byte[] = eventArgs.Body.ToArray();
            var json :string = Encoding.UTF8.GetString(bodyBytes);
            var infoEvent = JsonSerializer.Deserialize<InfoEvent>(json);

            if (infoEvent is not null)
            {
                var latency = (long)(processedAt - infoEvent.CreatedAtUtc).TotalMilliseconds;

                using var scope = _serviceProvider.CreateScope();
                var db = scope.ServiceProvider.GetRequiredService<EventsDbContext>();

                var entity = new ProcessedEventEntity
                {
                    EventId = infoEvent.Id,
                    Source = infoEvent.Source,
                    Payload = infoEvent.Payload,
                    CreatedAtUtc = infoEvent.CreatedAtUtc,
                    ProcessedAtUtc = processedAt,
                    ProcessingLatencyMs = latency,
                    Error = null
                };

                db.ProcessedEvents.Add(entity);
                await db.SaveChangesAsync();

                _logger.LogInformation(
                    "Processed event {Id} from {Source} with latency {Latency} ms",
                    infoEvent.Id,
                    infoEvent.Source,
                    latency);
            }

            await _channel!.BasicAckAsync(eventArgs.DeliveryTag, multiple: false);
        }
        catch (Exception ex)
```

```yaml
services:
  rabbitmq_info_flow:
    image: rabbitmq:3-management
    container_name: rabbitmq_info_flow
    environment:
      - RABBITMQ_DEFAULT_USER=user
      - RABBITMQ_DEFAULT_PASS=password
    ports:
      - "5673:5672"
      - "15673:15672"
    volumes:
      - ./docker_data/rabbitmq:/var/lib/rabbitmq

  events-db:
    image: postgres:latest
    container_name: events-db
    environment:
      POSTGRES_USER: "user"
      POSTGRES_PASSWORD: "password"
      PGDATA: "/var/lib/postgresql/data/pgdata"

    ports:
      - "5433:5432"
    volumes:
      - ./docker_data/events-db-data:/var/lib/postgresql/data

  gateway.api:
    image: gateway.api
    build:
      context: .
      dockerfile: Gateway.Api/Dockerfile
    environment:
      ASPNETCORE_URLS: "http://+:8080"
      RabbitMq__HostName: "rabbitmq_info_flow"
      RabbitMq__UserName: "user"
      RabbitMq__Password: "password"
      RabbitMq__VirtualHost: "info_flow"
    depends_on:
      - rabbitmq_info_flow
    ports:
      - "8080:8080"

  ingestion.service:
    image: ingestion.service
    build:
      context: .
      dockerfile: Ingestion.Service/Dockerfile
    environment:
      RabbitMq__HostName: "rabbitmq_info_flow"
      RabbitMq__UserName: "user"
      RabbitMq__Password: "password"
      RabbitMq__VirtualHost: "info_flow"
    depends_on:
      - rabbitmq_info_flow

  processing.service:
    image: processing.service
    build:
      context: .
      dockerfile: Processing.Service/Dockerfile
    environment:
      RabbitMq__HostName: "rabbitmq_info_flow"
      RabbitMq__UserName: "user"
      RabbitMq__Password: "password"
      RabbitMq__VirtualHost: "info_flow"
      ConnectionStrings__EventsDb: "Host=events-db;Port=5432;Database=events
    depends_on:
      - rabbitmq_info_flow
      - events-db
```