Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ВИВЧЕННЯ ЗАСОБІВ ТА АРХІТЕКТУРИ ДЛЯ ВЗАЄМОДІЇ МІКРОСЕРВІСІВ У ВЕБЗАСТОСУНКАХ
(тема)

Виконав:
студент 2 курсу, групи ІНФм-22-1
Кіяшко Д.Г.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник проф. Гороховатський В.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Кіяшку Денису Геннадійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Вивчення засобів та архітектури для взаємодії мікросервісів у вебзастосунках

затверджена наказом по університету від 3 листопада 2023 року № 1280Ст

2. Термін подання студентом роботи до екзаменаційної комісії 23 грудня 2023 р.

3. Вихідні дані до роботи архітектурні рішення та патерни мікросервісної архітектури, міжпроцесна комунікація, патерни відмовостійкості, інструменти керування транзакціями у мікросервісах.

4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз застосувань мікросервісної архітектури у вебзастосунках.
2. Засоби та архітектури для взаємодії мікросервісів у вебзастосунках.
3. Результати дослідження архітектур для взаємодії мікросервісів.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) представлення мікросервісної архітектури, масштабування у мікросервісах, концепції узгодженості даних, взаємодія мікросервісів, графік навантаження сервісів, ілюстрація переваг переходу на асинхронну взаємодію у мікросервісів.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	03.11.2023	
2	Аналіз завдання, підбір літератури	03.11.23-05.11.23	
3	Аналіз літератури з досліджуваної проблеми	05.11.23-10.11.23	
4	Аналіз взаємодії мікросервісів	10.11.23-13.11.23	
5	Забезпечення консистентності у мікросервісах	13.12.23-18.12.23	
6	Реалізація патернів відмовостійкості	01.12.23-10.12.23	
7	Оформлення пояснювальної записки	11.12.23-12.12.23	
8	Перевірка на плагіат	15.12.2023	
9	Рецензування	18.12.2023	
10	Підготовка презентації та доповіді	20.12.2023	
11	Занесення роботи в електронний архів	02.01.2024	
12	Попередній захист кваліфікаційної роботи	02.01.2024	

Дата видачі завдання 3 листопада 2023 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Гороховатський В.О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 60 с., 2 табл., 25 рис., 41 джерело.

МІКРОСЕРВІСНЯ АРХІТЕКТУРА, ПАТЕРНИ, МАСШТАБУВАННЯ, HTTP REST, МІЖПРОЦЕСНІ КОМУНІКАЦІЇ, MESSAGE BROKER, EVENT DRIVEN ARCHITECTURE, DOCKER, HTTP, PUBSUB, GOOGLE CLOUD, GOLANG.

Об’єктом дослідження є архітектурні рішення, патерни та технології, які використовуються у взаємодії мікросервісів у вебзастосунках.

Метою даного дослідження є аналіз, оцінка засобів і архітектурних рішень, які використовуються для взаємодії мікросервісів у вебзастосунках, з метою виявлення переваг, недоліків та сфер застосувань.

Проаналізовано теоретичні аспекти мікросервісної архітектури та засобів взаємодії мікросервісів. Розглянуто переваги та недоліки засобів та методів взаємодії. Застосовано на прикладі різноманітні архітектурні рішення для вирішення проблем та недоліків мікросервісної архітектури.

У результаті дослідження реалізовані архітектурні патерни, спрямовані на підвищення стабільності систем, що базуються на мікросервісах.

MICROSERVICES ARHITECTURE, PATTERNS, SCALING, HTTP REST, INTER PROCESS COMMUNICATION, MESSAGE BROKER, EVENT DRIVEN ARCHITECTURE, DOCKER, HTTP, PUBSUB, GOOGLE CLOUD, GOLANG.

The object of research is architectural solutions, patterns, and technologies used in the microservices communication in web applications.

The aim of this research is to analyze and evaluate tools and architectural solutions used for microservices communication in web applications, to identify advantages, disadvantages, and areas of application.

The theoretical aspects of microservices architecture and the means of microservices interaction were analyzed. The advantages and disadvantages of various interaction tools and methods were considered. Various architectural solutions were applied to address the problems and shortcomings of microservices architecture.

As a result of the research, architectural patterns aimed at increasing the stability of systems based on microservices were implemented.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

IPC – Inter-Process Communication (міжпроцесна комунікація)

MITM – Man-in-the-Middle (чоловік посередник)

DoS – Denial of Service (відмова у сервісі)

EDA – Event-Driven Architecture (архітектура керування подіями)

AMQP – Advanced Message Queuing Protocol

MQTT – Message Queue Telemetry Transport

CQRS – Command Query Responsibility Segregation

HTTP – Hypertext Transfer Protocol (протокол передачі гіпертексту)

REST – Representational State Transfer (передача репрезентативного стану)

CLI – Command Line Interface (інтерфейс командного рядка)

GCP – Google Cloud Platform

RPS – Requests Per Second (запитів за секунду)

ВСТУП

Об’єктом дослідження атестаційної роботи магістра є засоби та сучасні архітектури для взаємодії мікросервісів у вебзастосунках. Вивчення цього питання є актуальним на сьогоднішній день, тому що вибір правильних інструментів та архітектур є важливим у створенні успішного вебзастосунка, який буде відповідати усім сучасним вимогам продукта. [1–5].

Дослідження архітектури мікросервісів є ключовим компонентом у функціонуванні сучасних вебзастосунків. Мікросервісна архітектура дозволяє створювати відмовостійки, надійні та масштабовані системи, що є важливим фактором успішності розробки у різноманітних сферах застосування.

Мікросервіси зосереджені на наданні багатьох варіантів вирішення проблем, з якими можна зіткнутися у процесі розроби. Вони являють собою тип сервіс-орієнтованої архітектури, та приховують внутрішню реалізацію від зовнішнього світу, у якій незалежне розгортання є ключовим. Вони не залежать від технологій, що є однією з їхніх переваг, полегшуючи процес розробки та впровадження нового функціоналу [1].

Аналіз інструментів та архітектурних підходів для взаємодії мікросервісів дозволяє виявити найбільш ефективні стратегії та найкращі практики для розробки сучасних вебзастосунків. Важливість таких досліджень також зумовлена постійним розвитком технологій та появою нових методологій, які можуть бути використані для оптимізації процесу розробки та управління системою мікросервісів.

З урахуванням вищезазначеного, кваліфікаційна робота за цією темою є актуальною та важливою для розробників вебзастосунків у різноманітних сферах їх застосування.

1 АНАЛІЗ ЗАСТОСУВАНЬ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ У ВЕБЗАСТОСУНКАХ

1.1 Визначення та переваги мікросервісної архітектури

Розвиток програмного забезпечення пережив цифрових змін, що призвело до появи нового архітектурного підходу – мікросервісної архітектури. Мікросервіси – це спосіб структурувати вебзастосунок, як сукупність невеликих сервісів, кожен с яких виконує свою певну функцію та спілкується за допомоги протоколів передачі даних (рис. 1.1). Це розбиття на невеликі частини дозволяє розробникам концентруватися на конкретних завданнях, використовувати різні мови програмування та підтримувати їх незалежно один від одного [2–5].

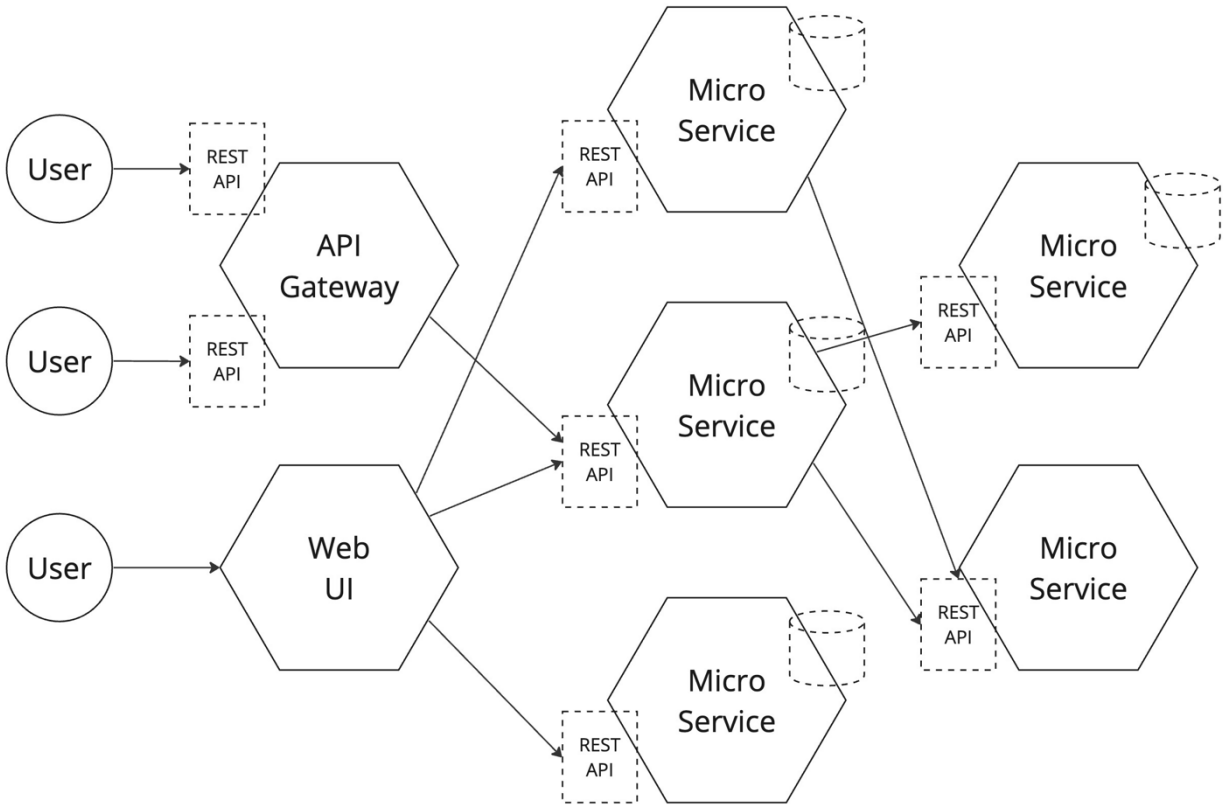

Рисунок 1.1 – Можлива мікросервісна архітектура продукту

Мікросервіси є відповіддю на різноманітні проблеми, які виникають при розробці та підтримці складних програмних систем. Вони забезпечують гнучкість, оскільки окремі сервіси можуть бути розроблені та оновлені незалежно, зменшуючи ризики помилок у інших частинах системи. Мікросервісна архітектура зручна для масштабування застосунків, оскільки окремі компоненти можуть бути розподілені на різні сервери або контейнери і кожен окремий сервіс може масштабуватись незалежно. Це дає можливість ефективно використовувати ресурси та забезпечувати високу доступність.

Однак разом із перевагами мікросервіси успадковують усі проблеми розподілених систем. Накладні ресурси на інфраструктуру, моніторинг, управління багатьма мікросервісами, і важливо враховувати питання безпеки та комунікації між сервісами. Тим не менше, мікросервіси стали популярним інструментом у сучасній розробці програмного забезпечення завдяки їх спроможності полегшувати розробку, підтримку і масштабування складних застосунків.

Мікросервісна архітектура популярна завдяки своїм численним перевагам, що сприяють гнучкості, незалежному масштабуванню, ефективним використанням ресурсів, простотою додавання нових функцій та стійкістю систем.

Ця модулярність сприяє швидшому внесенню змін, оскільки командам розробників не потрібно координувати зміни на рівні цілого застосунка. Вони можуть швидко оновлювати, тестувати та розгортати мікросервіси без значного впливу на решту системи. Така архітектура також підвищує стабільність системи, оскільки потенційні проблеми чи збої в одному мікросервісі менш ймовірно, що вплинуть на інші частини застосунка [3–5].

Завдяки мікросервісній архітектурі організації можуть обирати та впроваджувати окремі технології для кожного конкретного випадку, не обмежуючись одним технологічним стеком на рівні всього застосунка. Мікросервіси можуть бути написані на різних мовах програмування та взаємодіяти між собою без перешкод.

Мікросервіси значно підвищують масштабованість систем, пропонуючи різні стратегії масштабування [6–8]. Горизонтальне масштабування відбувається шляхом додавання нових інстанцій сервісів, що дозволяє розподіляти навантаження більш ефективно та підлаштовуватися під потреби у реальному часі. Вертикальне масштабування, у свою чергу, полягає у збільшенні ресурсів конкретного сервісу, таких як обчислювальна потужність (CPU) або оперативна пам’ять (RAM). Крім того, є ще діагональне масштабування, яке поєднує обидві ці стратегії, дозволяючи одночасно збільшувати ресурси існуючих сервісів та додавати нові інстанції. Це забезпечує ще більшу гнучкість та є ефективним підходом у сучасних розподілених системах.

Вертикальне масштабування, відоме також як масштабування «вгору», полягає у збільшенні потужності існуючих серверів шляхом додавання більш потужних процесорів, більшої кількості оперативної пам’яті, або кращих засобів зберігання даних (рис. 1.2).

Одним з основних аспектів вертикального масштабування є його відносна простота порівняно з горизонтальним масштабуванням.

Однак, вертикальне масштабування має свої обмеження. Щоб додати ресурсів поточним віртуальним машинам, їх потрібно вимкнути та додати нові зі зміненими вимогами до ресурсів. Також існують фізичні обмеження обладнання, межа до якої можна збільшувати потужність одного серверу, і коли ця межа досягнута, подальше підвищення ресурсів стає неможливим. Мова програмування може не підтримувати багатоядерного апаратного забезпечення, або не реалізовувати доступну пам’ять, що призводить до простою ресурсів. Також, модернізація апаратного забезпечення може бути досить вартісною, особливо при необхідності частого оновлення для відповідності зростаючим вимогам.

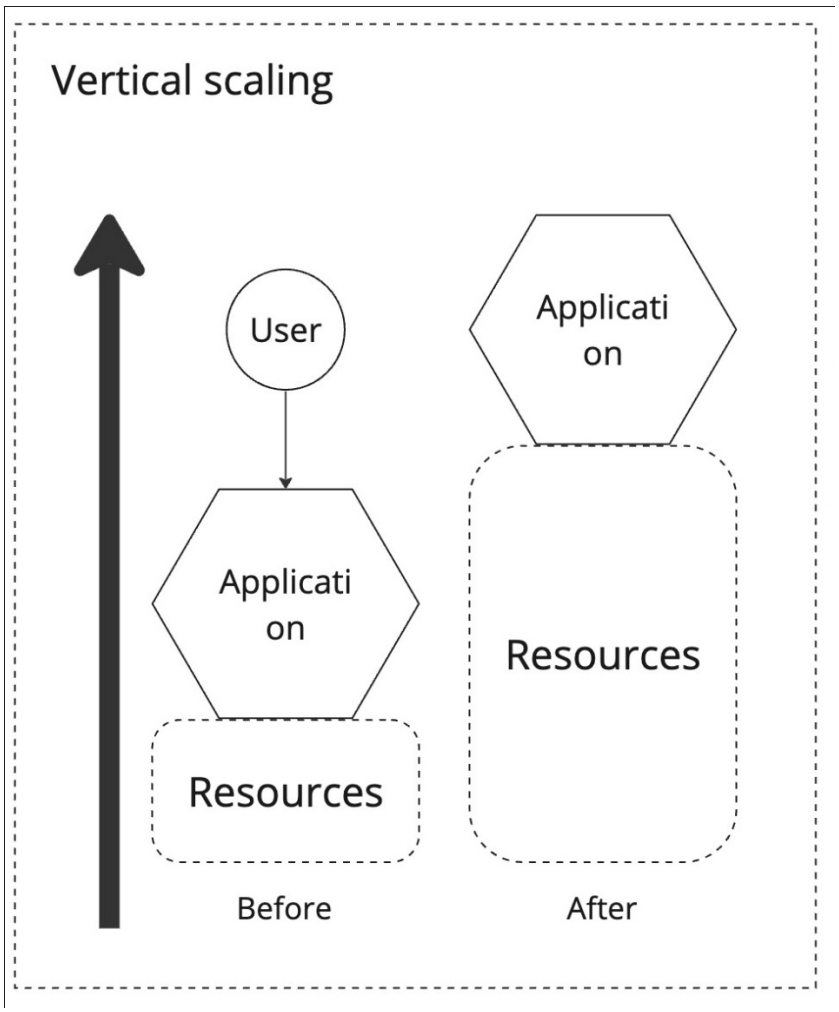

Рисунок 1.2 – Вертикальне масштабування систем

У випадку збою апаратного забезпечення весь сервер або система може стати недоступною, що може мати серйозні наслідки для бізнес-процесів.

Важливим елементом у процесі вертикального масштабування є забезпечення збалансованої конфігурації серверів. Просте збільшення обсягу оперативної пам’яті або процесорної потужності без урахування інших компонентів може не дати бажаного покращення продуктивності. Наприклад, сервер з великою кількістю оперативної пам’яті, але з обмеженою швидкістю диска, може стати «вузьким місцем» у загальній продуктивності системи.

Крім того, вертикальне масштабування вимагає ретельного планування та управління. Перед модернізацією серверів необхідно провести детальний аналіз поточних та прогнозованих вимог до системи, оцінити можливості та вартість оновлень, а також спланувати процес міграції або оновлення з

мінімальним впливом на доступність сервісу. У хмарних середовищах вертикальне масштабування може бути реалізоване швидше та ефективніше, оскільки воно не вимагає фізичної заміни апаратури.

Вертикальне масштабування, є ефективним підходом для вирішення короткострокових потреб у ресурсах та покращення продуктивності без необхідності радикальних змін у системі. Проте, з урахуванням його обмежень, часто це лише один з компонентів комплексної стратегії масштабування.

Горизонтальне масштабування, часто відоме як масштабування «назовні», є однією з основних стратегій управління ресурсами в сучасних системах, стратегія передбачає додавання додаткових віртуальних машин або екземплярів до існуючої системи для збільшення її пропускної потужності та здатності впоратися з більшим навантаженням та знижує ризики, пов’язані з перевантаженням або збоєм одного вузла, і забезпечує більш рівномірне розподілення ресурсів. Крім того, горизонтальне масштабування дає можливість системі гнучко реагувати на зміни в навантаженні, тим самим швидко збільшуючи або зменшуючи кількість виділених ресурсів у реальному часі (рис. 1.3).

Одним із ключових аспектів горизонтального масштабування є розподіл навантаження. Це означає, що запити до системи розподіляються між різними інстанціями або вузлами, забезпечуючи тим самим оптимальне використання ресурсів. Для цього використовуються різні методи балансування навантаження, такі як кругове балансування, балансування на основі мінімального навантаження чи географічне балансування. Ефективне балансування навантаження забезпечує рівномірне розподілення запитів між інстанціями, оптимізуючи використання ресурсів та мінімізуючи затримки в обробці запитів.

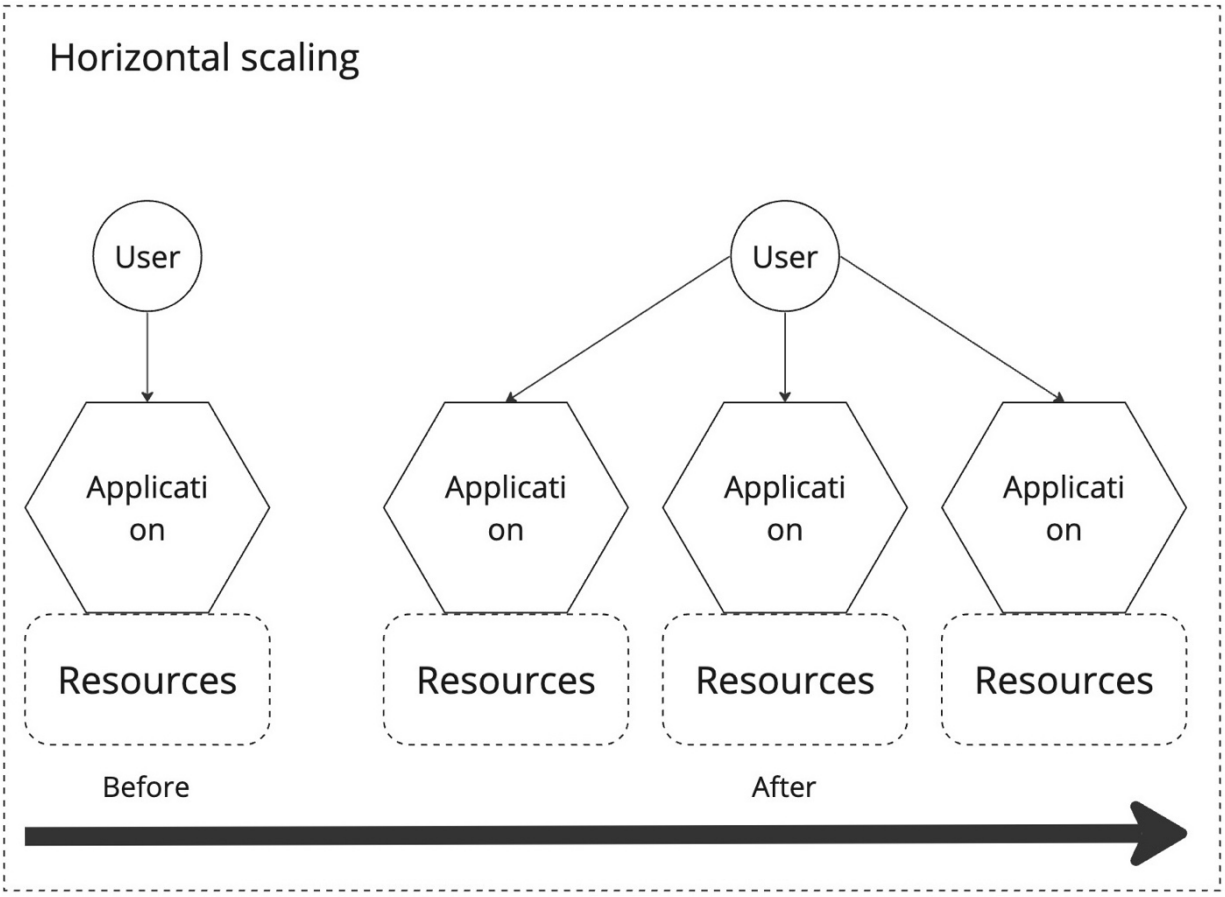

Рисунок 1.3 – Горизонтальне масштабування систем

Система має бути спроектована з урахуванням можливості додавання нових вузлів без необхідності змін у основному коді або архітектурі. Це вимагає слабкого зв’язування між компонентами системи та високого рівня модульності. Компоненти системи повинні бути самодостатніми та здатними працювати незалежно, що дозволяє додавати, видаляти або оновлювати індивідуальні вузли без впливу на решту системи.

Ще одним важливим аспектом горизонтального масштабування є забезпечення консистентності даних та стійкості системи до збоїв. У розподілених системах, де дані можуть зберігатися та оброблятися на різних вузлах, забезпечення консистентності даних стає складним завданням [9].

У сучасних хмарних середовищах та з використанням оркестраційних систем, таких як Kubernetes, можливо реалізувати автоматичне масштабування, яке дозволяє системі самостійно регулювати кількість

робочих вузлів в залежності від поточного навантаження. Це не тільки підвищує ефективність використання ресурсів, але й знижує витрати, оскільки додаткові ресурси задіються лише тоді, коли це дійсно необхідно [10–12].

Діагональне масштабування (Diagonal Scaling) є підходом, що комбінує елементи як горизонтального, так і вертикального масштабування. Це означає, що система може одночасно збільшувати потужність існуючих вузлів (вертикальне масштабування) та додавати додаткові вузли (горизонтальне масштабування) (рис. 1.4).

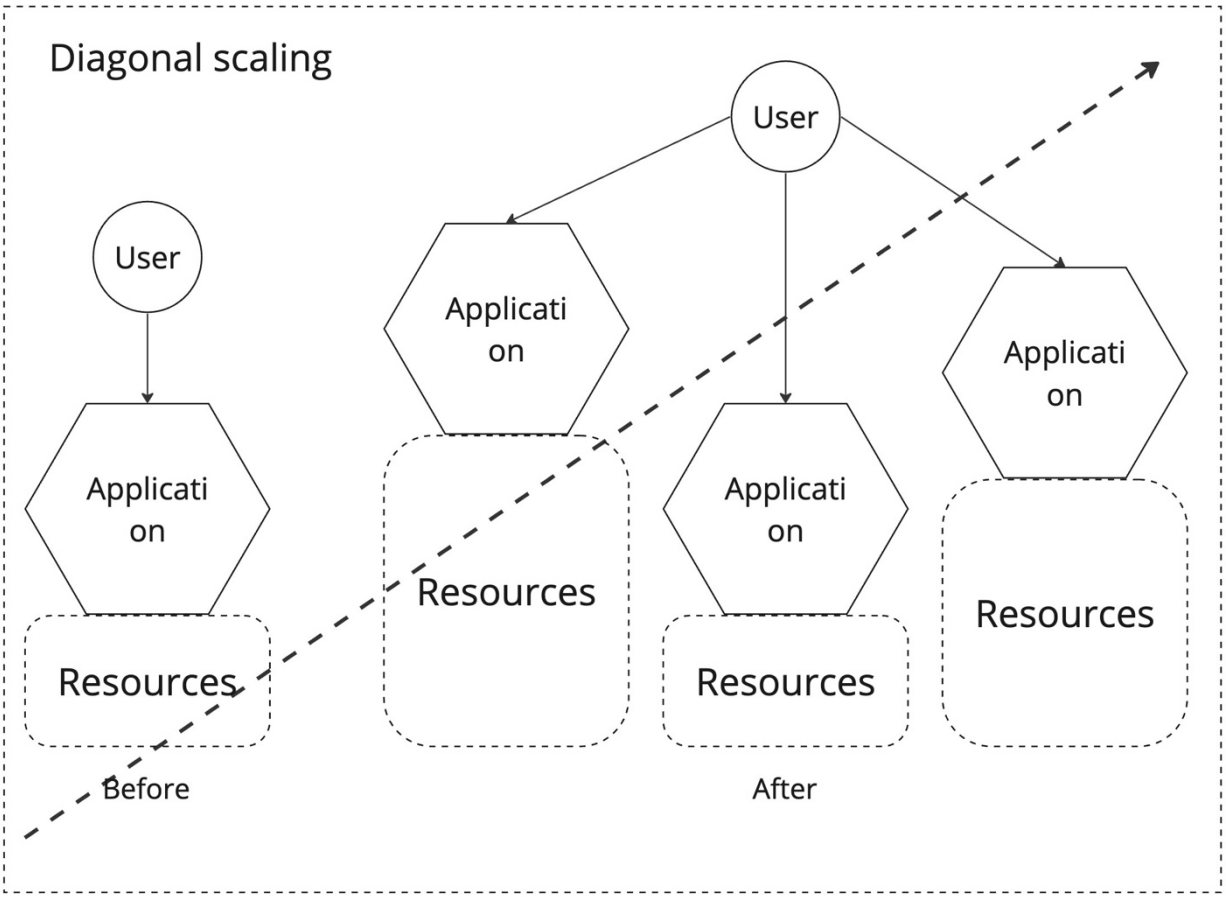

Рисунок 1.4 – Діагональне масштабування систем

Підхід діагонального масштабування дає можливість максимально ефективно використовувати ресурси, адаптуючись до поточних потреб. Наприклад, якщо застосунок потребує більше обчислювальної потужності,

можливо збільшити кількість та/або потужність серверів. Такий підхід забезпечує гнучкість у масштабуванні, оскільки дозволяє адаптуватися до змін у потребах без радикальних перебудов у архітектурі системи.

Діагональне масштабування також допомагає оптимізувати витрати, оскільки дає можливість вибирати найбільш ефективний спосіб масштабування в залежності від конкретної ситуації. Наприклад, у певних сценаріях може бути більш вигідним додати додаткові сервери замість модернізації існуючих, або навпаки.

Однак, слід зазначити, що діагональне масштабування вимагає складного управління та планування. Потрібне ретельне планування стратегії розподілення навантаження між віртуальними машинами, щоб досягти рівномірності, а також забезпечувати відповідну координацію між різними компонентами системи. Крім того, це може збільшити накладні витрати на інфраструктуру, оскільки потрібно одночасно підтримувати та оптимізувати як горизонтальне, так і вертикальне масштабування.

Порівняння характеристик різних стратегій масштабування наведені у таблиці 1.1.

Таблиця 1.1 – Характеристики масштабування

Масштабування	**Горизонтальне**	**Вертикальне**	**Діагональне**
Стратегія	Збільшення або зменшення ресурсів	Додавання або видалення додаткових вузлів	Комбінація горизонтального і вертикального масштабування для ефективнішого розподілу ресурсів.
Ефективність	Неоптимальний	Оптимальний	Оптимальний
Складність реалізації	Низька	Висока	Висока
Відмовостійкість	Низька	Висока	Висока
Вимагає простою	Так	Ні	Ні

1.2 Переваги та недоліки використання мікросервісної архітектури

Мікросервісна архітектура, як і будь-який інший підхід, має свої проблеми та недоліки. Нижче наведено деякі з них, у порівнянні з монолітною архітектурою [13].

Так як мікросервіси успадковали усі проблеми розподілених систем виникають проблеми зі стабільністю мережевого зв'язку між сервісами. Мікросервіси взаємодіють через мережу, що призводить до додаткових затримок і робить систему вразливою до мережевих відмов або нестабільності. Консистентність даних також є проблемою, оскільки кожен мікросервіс може мати свою базу даних, і транзакції, що охоплюють декілька сервісів, стають складними для управління.

Автоматизація та оркестрація необхідні для управління життєвим циклом мікросервісів, але вони також додають свою складність у вигляді додаткових інструментів та процесів, таких як Kubernetes. Service Discovery, чи здатність сервісів динамічно знаходити інші сервіси для взаємодії, також є викликом, який вимагає додаткових рішень.

Мікросервіси повинні бути спроектовані так, щоб вони могли ефективно обробляти помилки та адаптуватися або відновлюватися після збоїв. Це означає, що мікросервіси мають мати можливість автоматично відновлювати свою роботу після відмов без значних перерв у роботі. Синхронізація і блокування також є проблемами, оскільки синхронізація даних між мікросервісами, для підтримки консистентності, може призвести до тимчасових блокувань та зниженню продуктивності системи [2, 13–15].

Отже, розробники повинні ретельно планувати та впроваджувати найкращі практики та методології для управління складністю розподіленої системи мікросервісів.

Однією з ключових проблем є безпека, так як мікросервіси часто взаємодіють через мережу, що робить їх вразливими до різних мережевих

атак, таких як атаки «чоловік посередник» (Man-in-the-Middle, MITM), атаки відмови в обслуговуванні (Denial of Service, DoS) та інші.

Забезпечення безпеки комунікації між мікросервісами є вкрай важливим. Зазвичай для цього використовуються протоколи SSL/TLS для шифрування даних, переданих між сервісами, а також OAuth або JWT для автентифікації та авторизації.

Також необхідно забезпечити безпеку даних, які обробляються та зберігаються мікросервісами. Дані повинні бути захищені від несанкціонованого доступу, модифікації, витоку або втрати. Це може включати в себе різні заходи, такі як шифрування даних, аудит, моніторинг та резервне копіювання.

Контроль доступу також є важливим елементом безпеки, де необхідно визначити, хто (або який сервіс) має право доступу до певних ресурсів або операцій. Кожен мікросервіс повинен мати ізольований доступ до своїх даних та не мати можливість змінювати ресурси, що не відносяться до нього [2, 13].

Масштабування мікросервісів приводить до ряду питань. По-перше, балансування навантаження стає критично важливим, оскільки необхідно ефективно розподіляти запити між численними екземплярами сервісів, що може вимагати впровадження розширених алгоритмів балансування та використовування механізмів виявлення сервісів.

Також масштабування вносить складність у синхронізацію та консистентність даних, оскільки додавання нових екземплярів чи баз даних вимагає розробки механізмів для забезпечення цілісності та узгодженості даних між ними.

Автоматизація та оркестрація масштабування також стають складнішими, оскільки потрібно автоматично створювати, конфігурувати та управляти множинними екземплярами служб в реальному часі, забезпечуючи при цьому їх правильну конфігурацію та взаємодію.

Особливу увагу потребує також забезпечення стабільності та надійності системи під час масштабування, з урахуванням потенційних відмов або збоїв у деяких компонентах системи, що може вимагати впровадження резервування, аварійного відновлення та стратегій завантаження для запобігання відмов.

Отже, масштабування мікросервісів – це непросте завдання, яке вимагає глибокого технічного розуміння та ретельного планування.

Оскільки мікросервіси є розподіленими та незалежними, виникають ускладнення логування та моніторингу. Збір, агрегація та аналіз логів і метрик з усіх сервісів може бути ускладненим. У мікросервісних архітектурах часто застосовується централізований підхід до логування, де логи з різних сервісів збираються та агрегуються в одному місці, що дозволяє легше виконувати аналіз та виявлення проблем. Також важливо створити єдиний стандарт для логування, який допоможе уніфікувати та структурувати логи.

Потреба у реальному часі спостерігати за станом, продуктивністю кожного сервісу також призводить до ускладнення процесів моніторингу. Необхідно впровадження систем моніторингу та сповіщення, які можуть автоматично виявляти та реагувати на потенційні проблеми або відмови та повідомляти про критичні помилки.

Також важливою є трасування транзакцій, яке допомагає відстежувати виконання запитів через різні мікросервіси, що полегшує діагностику та виявлення проблем у взаємодії між сервісами. Всі ці питання вимагають ретельного планування та впровадження ефективних інструментів та практик для логування та моніторингу.

Проблеми розробки та розгортання в мікросервісних архітектурах часто з’являються через незалежність та розподіленість мікросервісів. Розробка вимагає координації та стандартизації, оскільки кожен мікросервіс може бути розроблений різними командами та використовувати різні

технології та мови програмування. Необхідно впровадити єдині стандарти та практики для забезпечення сумісності та співпраці між мікросервісами.

У контексті розгортання, автоматизація, оркестрація та безперервна інтеграція/безперервна доставка (CI/CD) стають ключовими. Автоматизовані процеси розгортання дозволяють швидко та ефективно вносити зміни, але також вимагають високого рівня контролю та стабільності, щоб запобігти помилкам та перебоям у роботі сервісів під час розгортання.

Контейнеризація, зокрема використання Docker та Kubernetes, часто використовується для спрощення та стандартизації процесів розгортання, дозволяючи створювати мікросервіси з їх залежностями у контейнері, які згодом можна легко розгортати та управляти [11, 12].

За все вищесказане, розробка та розгортання мікросервісів вимагають чіткого планування, стандартизації та автоматизації, а також постійної координації між командами розробників.

Керування конфігурацією в мікросервісних архітектурах відіграє ключову роль, але при цьому супроводжується рядом проблем. Однією з головних проблем є необхідність управління конфігураціями множини мікросервісів, які можуть розміщуватися в різних середовищах та мати різні версії. Це стає ще складнішим, коли конфігурації потрібно змінювати динамічно в реальному часі.

Централізоване керування конфігурацією може полегшити цей процес, дозволяючи зберігати та управляти конфігураціями всіх сервісів в одному місці. Проте, це також може призвести до проблем з безпекою та доступом, оскільки конфігураційні дані можуть бути чутливими.

Іншим ускладненням є автоматизація процесів та забезпечення консистентності конфігурацій, так щоб всі сервіси користувалися актуальними та правильними конфігураційними даними [16–18].

Тестування в мікросервісних архітектурах стикається з рядом проблем, викликаних розподіленою природою системи та великою кількістю взаємодіючих компонентів. По-перше, є складність в організації тестування

окремих мікросервісів у відокремленому середовищі, що імітує реальні умови їх взаємодії. По-друге, треба зосереджуватися на інтеграційному тестуванні, щоб перевірити правильність взаємодії мікросервісів між собою, що може виявитися вкрай складним через асинхронність та нестабільність мережевих взаємодій.

Також важливим є навантажувальне тестування, яке допомагає визначити, наскільки добре система може масштабуватися та витримувати великі обсяги запитів. Це допомагає виявити проблеми з продуктивністю та надійністю на ранніх стадіях розробки.

Крім того, в мікросервісних архітектурах особливу увагу слід приділяти автоматизації тестування, оскільки ручне тестування може бути дуже трудомістким та повільним. Автоматизовані тестові сценарії можуть значно підвищити ефективність та швидкість процесу тестування.

Проблема залежностей та співпраці в мікросервісних архітектурах є однією з найбільш важливих. Мікросервіси, як правило, взаємодіють один з одним, щоб надати повний набір функціональності, і ця взаємодія часто супроводжується рядом проблем. Залежності між мікросервісами можуть ускладнювати процеси розробки, тестування та розгортання, оскільки зміни в одному мікросервісі можуть вплинути на інші мікросервіси, з якими він взаємодіє.

Необхідно ретельно керувати залежностями, плануючи та координуючи розробку та розгортання мікросервісів, щоб мінімізувати негативний вплив змін. Крім того, управління версіями стає критично важливим, оскільки різні версії мікросервісів повинні правильно взаємодіяти один з одним.

Важливою є також проблема ізоляції мікросервісів, щоб зменшити вплив залежностей і спростити процеси розробки та управління. Зусилля по оптимізації залежностей та співпраці можуть значно покращити адаптивність, стабільність та швидкість доставки мікросервісних систем.

1.3 Засоби комунікації та архітектурні рішення

Комунікація між мікросервісами є ключовим аспектом мікросервісної архітектури, сервіси постійно взаємодіють між собою для виконання бізнес-процесів. Різні методи комунікації використовуються для забезпечення ефективної та надійної взаємодії. HTTP REST є популярним протоколом для синхронної комунікації між мікросервісами через його легкість та широку підтримку. REST використовує стандартні HTTP методи, такі як GET, POST, PUT та DELETE, для взаємодії між сервісами [20].

GraphQL є іншим сучасним варіантом, який дозволяє клієнтам запитувати конкретні дані, які їм потрібні, і отримувати відповіді в строго визначеному форматі. Це може зменшити кількість даних, які потрібно передавати через мережу, і покращити продуктивність.

Асинхронна комунікація між мікросервісами дозволяє сервісам обмінюватися повідомленнями без прямого та негайного з’єднання. Асинхронний підхід дозволяє системам працювати незалежно, забезпечуючи високу стабільність, масштабованість та ефективність.

Для асинхронної комунікації часто використовуються системи обміну повідомленнями, такі як Apache Kafka чи RabbitMQ. Вони дозволяють мікросервісам публікувати повідомлення в черги, на які інші сервіси можуть підписатися та обробляти в асинхронному режимі. Це може бути корисним для обробки великих обсягів даних, довгострокових процесів та для підвищення стійкості системи [1, 14].

Архітектурні рішення для мікросервісів у вебзастосунках визначають, як система буде реагувати на зміни, масштабуватися та адаптуватися до нових вимог. Однією з ключових концепцій, є «Event-Driven» архітектура. У цій архітектурі мікросервіси комунікують між собою за допомогою подій, що дозволяє системі бути більш гнучкою та здатною адаптуватися до змін у реальному часі. Події представляють зміни у стані системи і можуть бути

опубліковані одним сервісом та спожиті одним або кількома іншими сервісами.

Другою важливою концепцією є CQRS (Command Query Responsibility Segregation), яка дозволяє розділити логіку обробки команд (зміна стану) та запитів (читання стану) на різні сервіси. Це може поліпшити продуктивність, масштабованість та безпеку системи, оскільки кожен сервіс може бути оптимізований та масштабований незалежно.

Також слід розглянути використання патернів проектування, таких як агрегація, сага та API композиція, для керування транзакціями, бізнес-процесами та інтеграцією даних між сервісами. Агрегації дозволяють групувати пов’язані об’єкти та операції, сага може координувати довготривалі транзакції та бізнес-процеси між сервісами, а API композиція допомагає управляти запитами між сервісами для побудови єдиного клієнтського відгуку. API Gateway є центральним вузлом для всіх вхідних запитів до системи, що полегшує управління зв’язками між клієнтами та мікросервісами [9]. Service Discovery допомагає сервісам виявляти один одного в динамічному середовищі, особливо важливо у хмарних рішеннях. Circuit Breaker сприяє запобіганню поширення збоїв шляхом тимчасового припинення операцій, які можуть викликати каскадні помилки.

Додатково, асинхронні запити забезпечують надійний обмін повідомленнями між сервісами без необхідності безпосереднього синхронного зв’язку, використовуючи черги повідомлень або події для асинхронної комунікації [10–13].

Кожен з цих рішень пропонує окремі переваги та може бути застосований залежно від конкретних бізнес-вимог, технічних вимог та цілей організації. Вони повинні сприяти гнучкості, спрощенню проектування, масштабованості, відмовостійкості та продуктивності системи, а також полегшити управління та обслуговування у майбутньому.

1.4 Постановка задачі дослідження

Об’єктом дослідження є вивчення засобів та архітектур для взаємодії мікросервісів у вебзастосунках.

Основною метою дослідження є аналіз, вивчення та оцінка патернів і архітектурних рішень, які використовуються у мікросервісній взаємодії, для виявлення переваг та недоліків.

Для досягнення мети необхідно вирішити такі завдання:

– провести детальний аналіз мікросервісної архітектури та засобів взаємодії;

– вивчити існуючі архітектурні рішення взаємодії мікросервісів у вебзастосунках;

– оцінити переваги та недоліки проаналізованих архітектурних рішень у мікросервісах;

– реалізувати та застосувати на прикладі ефективні патерни стабільності для підтримки узгодженості даних у мікросервісах.

2 ЗАСОБИ ТА АРХІТЕКТУРИ ДЛЯ ВЗАЄМОДІЇ МІКРОСЕРВІСІВ У ВЕБЗАСТОСУНКАХ

2.1 Міжпроцесна комунікація в мікросервісній архітектурі

Міжпроцесні комунікації (Inter-Process Communication, IPC) у мікросервісній архітектурі. IPC є важливою частиною архітектури мікросервісу, яка сприяє ефективності, гнучкості та масштабованості мікросервісів. Модель комунікації може бути синхронною або асинхронною, кожна з яких має свої унікальні характеристики, переваги та недоліки [22].

Синхронна комунікація базується на принципі прямої взаємодії між компонентами системи. Клієнт відправляє запит до сервера та чекає на відповідь. Цей підхід є простим для розуміння та імплементації, адже він створює чітку лінійну послідовність виконання запитів. Однак така стратегія має свої недоліки. Синхронна модель може бути менш масштабованой через блокування ресурсів під час чекання на відповідь від сервера (рис. 2.1). Також ця модель може бути менш відмовостійкою, оскільки збій у одному сервісі може призвести до збоїв у інших, залежних сервісах.

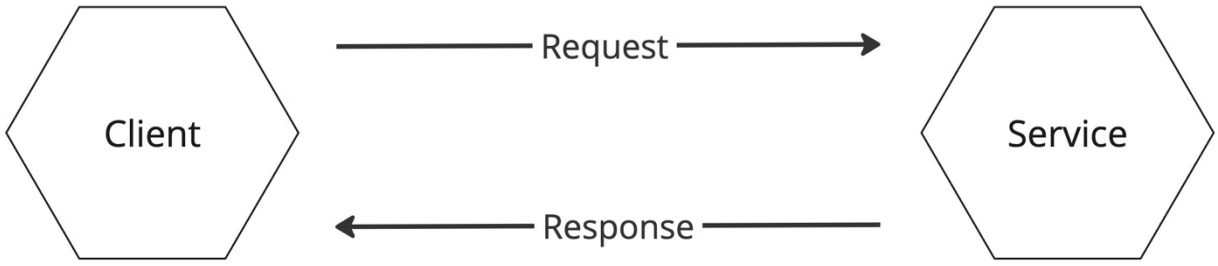

Рисунок 2.1 – Приклад синхронної комунікації

З іншого боку, асинхронна комунікація базується на принципі незалежності операцій. Замість прямої взаємодії, сервіси взаємодіють через

механізми, такі як черги повідомлень або подій. Асинхронна модель дозволяє сервісам продовжувати роботу без очікування на відповідь від інших сервісів, що може поліпшити загальну пропускну здатність системи та її здатність до масштабування (рис. 2.2).

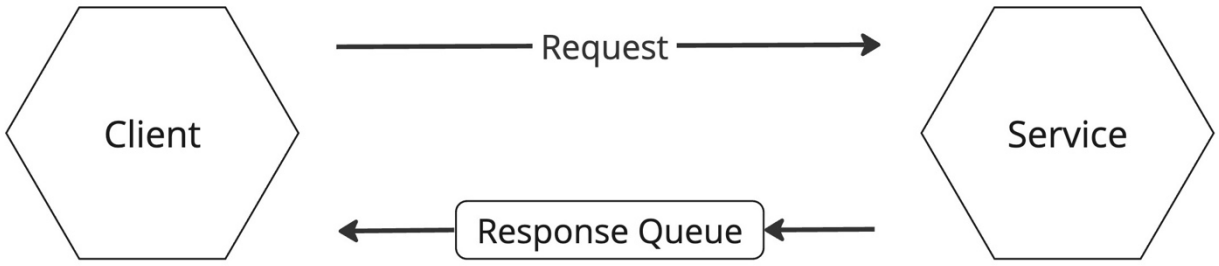

Рисунок 2.2 – Приклад асинхронної комунікації

Наприклад, в інтернет-магазинах асинхронна комунікація може використовуватися для публікації замовлень, або надсилання повідомлень користувачам події. Також мікросервіси можуть надсилати повідомлення про помилки чи стан системи за допомоги асинхронних повідомлень.

Така модель може бути більш складною для реалізації, оскільки вона вимагає управління асинхронними процесами та обробки помилок. Однак, завдяки асинхронності, система може бути більш стійкою до відмов, оскільки сервіси не є тісно зв’язаними між собою, і збій одного сервісу не призведе до прямого збою інших.

Шаблони проектування для асинхронної взаємодії мікросервісів у вебзастосунках створюють основу для ефективної комунікації та обробки даних.

EDA – це шаблон архітектури програмного забезпечення, ключовим аспектом котрого є асинхронність, та кожна частина системи фокусується на генерації або обробці подій. Компоненти, які створюють події, зазвичай не залежать від тих, що їх обробляють, що створює систему зі слабкими взаємозв’язками. Ця особливість забезпечує велику гнучкість, адже кожен

компонент може бути змінений, вилучений або доданий без значних змін в інших частинах системи.

Система, керована подіями, зазвичай складається з джерел подій (або агентів), споживачів подій (або приймачів) і каналів подій, через які події поширюються [19–24].

Для управління подіями в EDA зазвичай використовуються брокери подій або системи управління чергами повідомлень, що функціонують як посередники між виробниками та споживачами подій. Такі системи забезпечують додаткові можливості, як-от гарантування доставки повідомлень, розподілення навантаження, та збалансоване споживання ресурсів. Події передаються через канали, які ізолюють різні частини системи одну від одної, подальше збільшуючи гнучкість та масштабованість.

Використання EDA дозволяє системам бути більш адаптивними до змін, оскільки додавання нових типів обробки подій або реакцій на вже існуючі події може відбуватися без перерви роботи системи. Це також сприяє більш легкому масштабуванню, адже система може бути розширена додаванням нових обробників подій, які можуть працювати паралельно та незалежно від інших частин системи [18].

У архітектурі, орієнтованій на події (EDA), брокер повідомлень (message broker) відіграє критичну роль як ключовий компонент, що забезпечує зв’язок між різними частинами системи. Його основна функція – це посередництво у передачі повідомлень або подій між виробниками (publishers) та споживачами (subscribers), дозволяючи їм взаємодіяти без необхідності знати про внутрішнє розташування одне одного.

Брокера повідомлень управляє потоками даних, розподіляючи повідомлення відповідно до правил та маршрутів, які були задані. Він може приймати повідомлення від будь-якого джерела, перетворювати їх за потреби для споживача та доставляти підписнику. Прикладами таких брокерів повідомлень є Apache Kafka, RabbitMQ, Amazon SQS, Google PubSub, та інші.

Брокери повідомлень забезпечують надійність системи (рис. 2.3). Вони можуть гарантувати, що повідомлення буде доставлено навіть у разі тимчасових збоїв у роботі системи. Для цього використовуються різні стратегії доставки повідомлень, включаючи підтвердження отримання (acknowledgements) та поновлення спроб доставки (retry mechanisms).

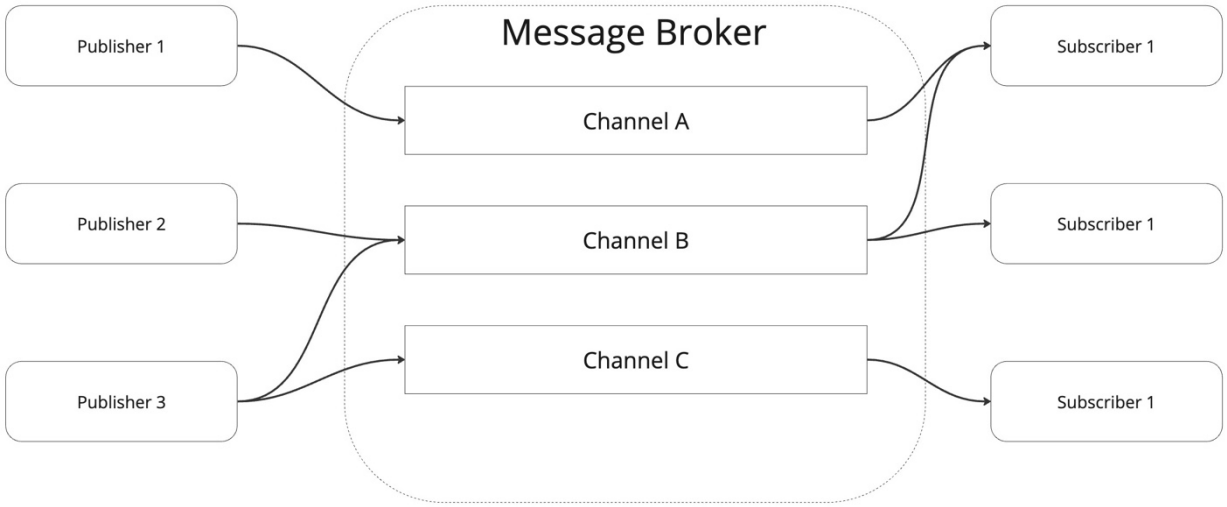

Рисунок 2.3 – Брокер повідомлень

Важливою перевагою використання брокера повідомлень є те, що відправнику не потрібно знати розташування споживача чи його доступність під час публікації повідомлення. Ще одна перевага полягає в тому, що брокер повідомлень буферизує повідомлення, та може зберігати їх, доки споживач не зможе їх обробити.

В EDA, брокера повідомлень відіграє роль основного диспетчера подій, що відбуваються у системі. Події можуть включати все: від простих змін стану до складних бізнес-операцій. Роль брокера полягає у забезпеченні, що ці події будуть перехоплені, оброблені та розповсюджені відповідно до логіки бізнес-процесів.

Крім цього, сучасні message broker пропонують розширені можливості для забезпечення високої доступності та відмовостійкості. Вони можуть пропонувати кластеризацію, автоматичне відновлення після збоїв, та

георозподілене реплікування для забезпечення безперервної роботи в розподілених середовищах [2, 14].

Work Queues, або робочі черги, є шаблоном, що дозволяє розподілити завдання між різними працівниками (workers), які можуть виконувати ці завдання асинхронно (рис. 2.4). Кожне завдання у черзі – це зазвичай невелика, самодостатня одиниця роботи, яку може обробити працівник. Цей підхід дозволяє системі легко масштабуватися, збалансувати навантаження між ними, оскільки можна додати додаткових працівників або видалити їх залежно від поточного навантаження. Кожен екземпляр буде приймати різні повідомлення, обробляти їх і, можливо, зберігати результат у базі даних [30].

У Work Queues важливим аспектом є здатність воркерів до відновлення після збоїв. Це забезпечується шляхом повторної постановки завдання у чергу у разі невдачі або шляхом реалізації механізмів транзакційності та витривалості.

Publish/Subscribe (або pub/sub) є патерном, у якому виробники подій (publishers) не надсилають повідомлення на пряму споживачам (subscribers). Натомість, події надсилаються до централізованої служби (topic або channel), де вони можуть бути підписані одним або кількома споживачами. Споживачі можуть динамічно підписуватися та відписуватися від тем, забезпечуючи велику гнучкість у тому, як події обробляються

Однією з основних переваг Publish/Subscribe є відокремлення виробників і споживачів подій, що дозволяє системам змінюватися безпосередньо, не впливаючи одна на одну. Це також дозволяє реалізувати більш складні шаблони обробки подій, такі як фільтрація та перетворення подій, перш ніж вони досягнуть споживачів (рис. 2.4).

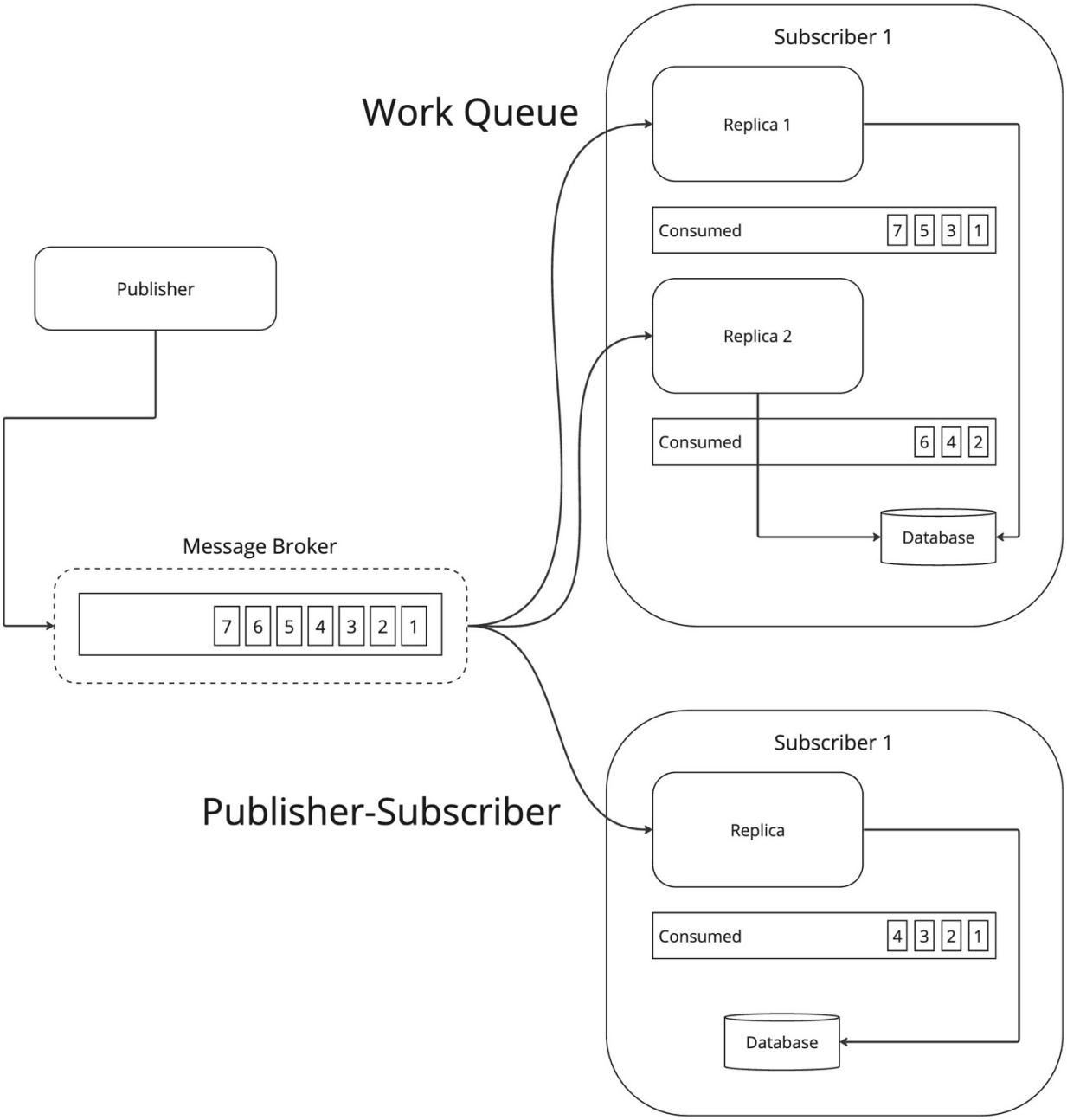

Рисунок 2.4 – Шаблони обміну повідомленнями

2.2 Протоколи обміну повідомленнями

Протоколи посередника повідомлень (message broker protocols) виконують фундаментальну роль у сучасних розподілених системах, забезпечуючи стандартизований спосіб взаємодії між незалежними компонентами. Ці протоколи регулюють форматування, маршрутизацію та

доставку повідомлень, що є життєво важливими для роботи систем, які покладаються на асинхронний обмін даними [2, 14].

AMQP (Advanced Message Queuing Protocol) протокол, спрямований на забезпечення обміну повідомленнями між будь-яким програмним забезпеченням, що підтримує повідомлення, незалежно від варіанту використання, мережевого середовища тощо. Підтримує складні маршрутизаційні сценарії, транзакції та надійну доставку. Поділений на три основні компоненти – обмінники (exchanges), черги (queues) та зв’язки (bindings) – AMQP уможливлює високий рівень абстракції та контролю над потоком повідомлень, незалежно від мови програмування чи платформи.

MQTT, з іншого боку, вирізняється своєю простотою та легковісністю, що робить його ідеальним для IoT застосунків, де ширина каналу та ресурси пристрою можуть бути обмежені. Його модель підписки на теми (topic-based subscription) дозволяє пристроям підписуватися на певні шаблони повідомлень, мінімізуючи непотрібний трафік і оптимізуючи доставку інформації.

STOMP, який стоїть за Simple (or Streaming) Text Oriented Messaging Protocol, базується на тексті, але також дозволяє передавати двійкові повідомлення. Його текстово-орієнтований характер робить його легко використовуваним і інтегрованим в багато різних систем. Розроблено як простий протокол, який легко реалізувати на стороні сервера та клієнта, та який підтримується багатьма мовами програмування. [2, 6, 18].

HTTP та WebSockets також заслуговують на увагу в контексті message broker protocols, навіть якщо вони традиційно асоціюються з вебкомунікаціями. HTTP, який є основою світової павутини, використовується для RESTful взаємодій між вебсервісами. WebSockets це протокол зв'язку, призначений для двосторонніх комунікаційний через з'єднання TCP. Він відрізняється від HTTP, але розроблений для бездоганної роботи з HTTP.

У кожного протоколу є свої сильні і слабкі сторони, і вибір залежить від специфіки задачі. Для великих підприємств, які потребують гарантованої доставки та транзакційної підтримки, AMQP може бути ідеальним рішенням. У контексті IoT і мобільних застосунків, MQTT пропонує необхідну ефективність та легковісність. І хоча HTTP та WebSockets не є вузькоспеціалізованими протоколами для обробки повідомлень, вони залишаються найпоширенішими для вебкомунікацій.

Безумовно, вибір протоколу впливає на архітектуру системи, її масштабованість, надійність та легкість обслуговування. Це також впливає на можливості взаємодії між різними системами, вимагаючи ретельного розгляду сумісності та майбутніх потреб. У світі, де вимоги до систем постійно зростають, і де швидкість, надійність та масштабованість є ключовими, вибір правильного протоколу стає все більш критичним. Від стартапів, що розпочинають свій шлях, до великих корпорацій, розуміння та ефективне використання протоколів є важливим питанням.

2.3 Забезпечення консистентності даних у міжсервісній взаємодії

У мікросервісних системах, де кожен сервіс функціонує незалежно та може мати свою базу даних, забезпечення консистентності даних між різними сервісами стає викликом. Порушення консистентності даних може призвести до ряду проблем, включаючи неправильне відображення інформації користувачам, втрату важливих даних, або неправильні бізнес-рішення, що базуються на неповних або невірних даних [15, 16, 19].

Забезпечення узгодженості даних у мікросервісних архітектурах вимагає використання специфічних стратегій. Ці стратегії допомагають управляти тим, як дані зберігаються, оновлюються та синхронізуються між різними сервісами.

Базова узгодженість, відома як eventual consistency, є ключовою концепцією в мікросервісних архітектурах, особливо у контексті розподілених систем. Цей підхід допускає тимчасові розбіжності у даних між різними сервісами, передбачаючи, що всі копії даних врешті-решт синхронізуються (рис. 2.5). Важливо розуміти, що базова узгодженість не забезпечує миттєвої узгодженості даних після кожної транзакції, натомість вона гарантує, що якщо не відбувається нових оновлень, то данні в кінцевому підсумку стануть консистентними. Основною перевагою цієї стратегії є підвищена доступність та відмовостійкість системи, оскільки вона дозволяє сервісам працювати навіть при часткових збоях або відключеннях мережі. Також, це дозволяє масштабувати системи, оскільки запити можуть оброблятися різними вузлами незалежно один від одного.

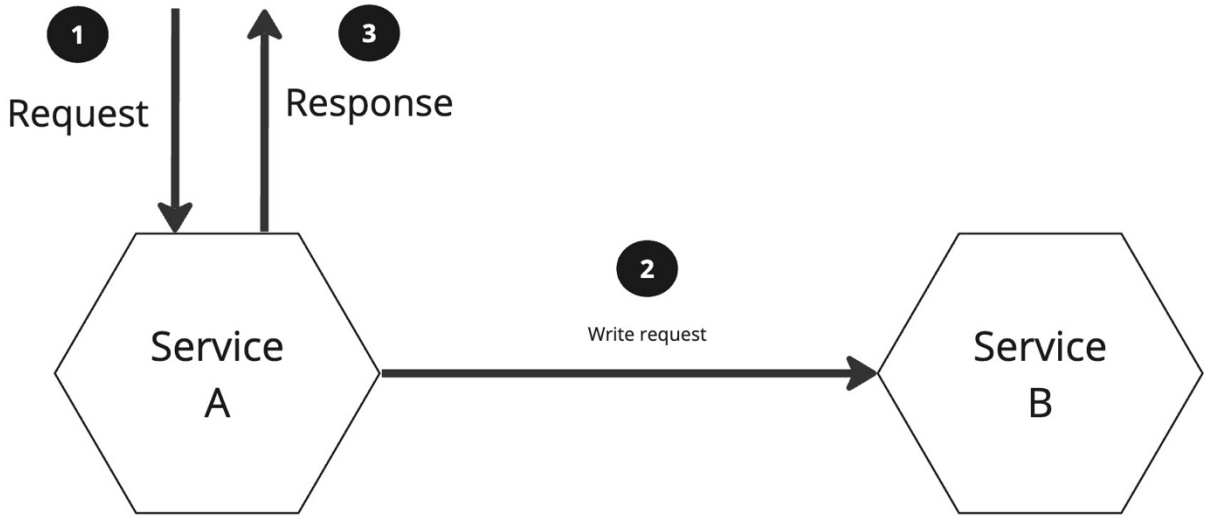

Рисунок 2.5 – Базова стратегія (eventual consistency) узгодженості

Сильна узгодженість, або strong consistency, є стратегією управління даними у мікросервісних архітектурах, яка вимагає, щоб всі копії даних у різних сервісах були одночасно оновлені [1, 2, 14–18]. Відмінність цієї стратегії від базової узгодженості полягає в тому, що вона забезпечує негайне відображення змін у даних по всій системі, забезпечуючи тим самим консистентність даних у будь-який момент часу (рис. 2.6). Основна перевага

сильної узгодженості полягає в тому, що вона гарантує, що всі сервіси працюють з актуальними даними, що є критично важливим для деяких застосунків, особливо тих, де потрібна висока точність та надійність даних, наприклад, у фінансових системах або системах, які відповідають за управління критично важливими процесами.

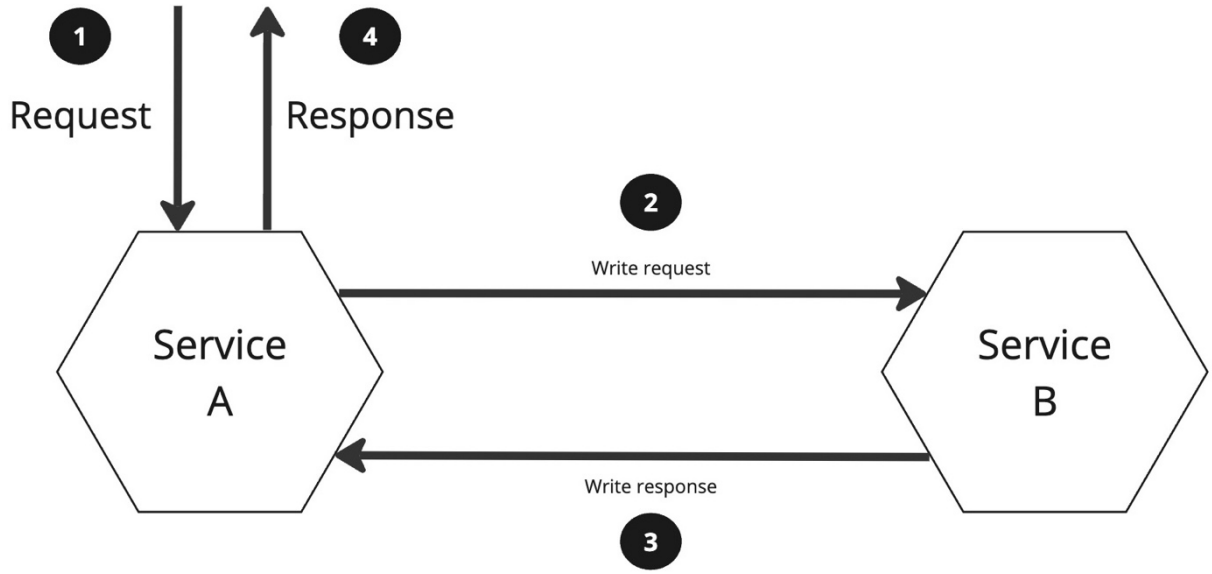

Рисунок 2.6 – Сильна стратегія (strong consistency) узгодженості

Однак, основним недоліком цієї стратегії є те, що вона може значно знижувати продуктивність системи, оскільки вимагає синхронізації між всіма сервісами під час кожного оновлення даних. Така синхронізація може призводити до затримок та зниження загальної пропускної здатності системи. Крім того, сильна узгодженість може знижувати доступність системи в умовах мережевих збоїв або коли один із сервісів стає недоступним [14–18].

Послідовна узгодженість (sequential consistency), передбачає, що всі операції з даними виконуються у визначеному порядку, забезпечуючи, що всі зміни в даних будуть відображені у всіх сервісах в однаковій послідовності. Це означає, що якщо дві операції A та B виконані в одному порядку в одному сервісі, вони будуть відображені в тому ж порядку в усіх інших сервісах.

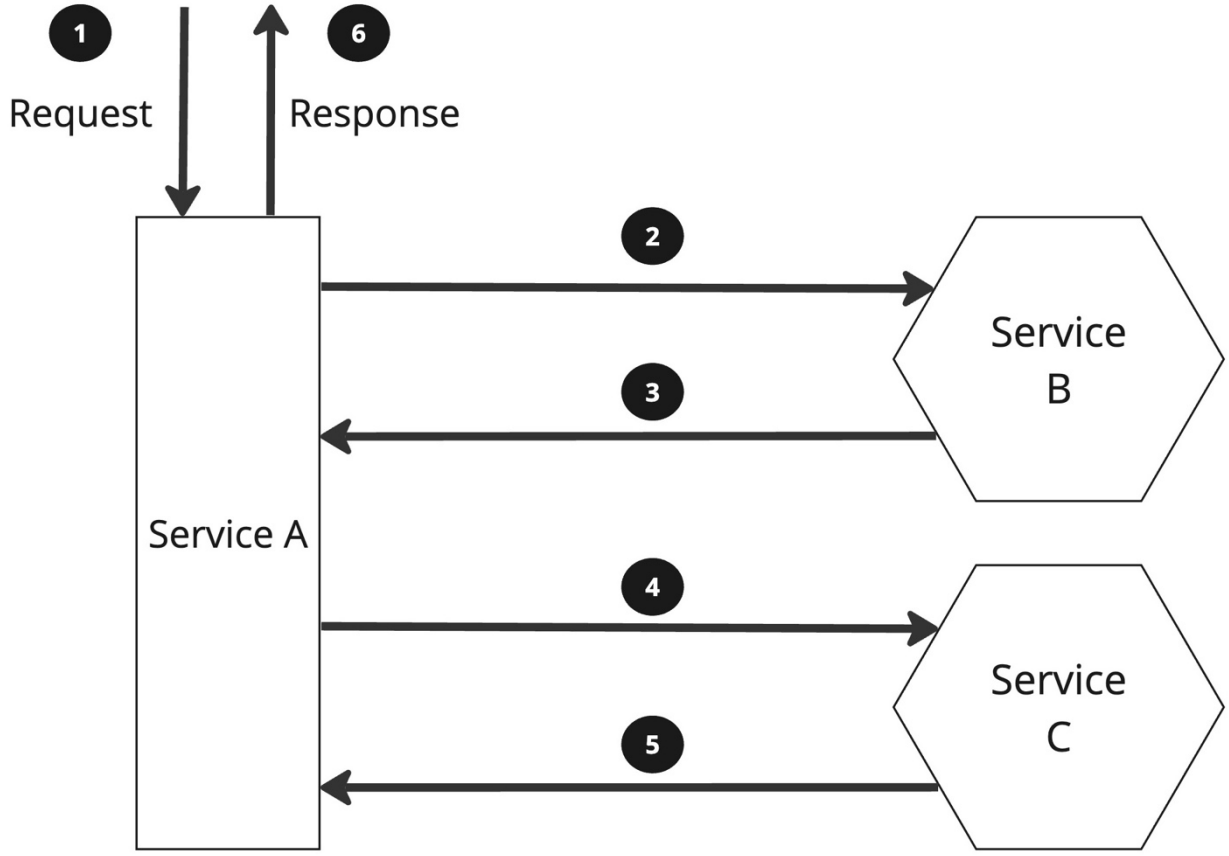

Рисунок 2.6 – Послідовна стратегія (sequential consistency) узгодженості

Перевагою послідовної узгодженості є забезпечення чіткої та зрозумілої моделі взаємодії між сервісами, що сприяє легшому розумінню та передбаченню поведінки системи. Це може бути особливо корисним у складних системах, де важливо розуміти, як різні компоненти взаємодіють один з одним. Однак, вимога до збереження певної послідовності операцій може обмежувати можливості паралелізму та масштабування, оскільки необхідно постійно синхронізувати стан між сервісами для підтримання цієї послідовності.

Компенсаційні транзакції (Compensating Transactions), спрямовані на управління помилками та забезпечення консистентності даних у випадках, коли звичайні транзакції неможливі або недоцільні (рис. 2.6). Цей підхід часто використовується у патернах, таких як Saga, де довготривалі транзакції, що охоплюють кілька сервісів, розбиваються на набір менших транзакцій. Ключовим принципом компенсаційних транзакцій є скасування операцій для

кожної частини основної транзакції. Якщо будь-яка частина основної транзакції зазнає невдачі, відповідні компенсаційні транзакції виконуються для відновлення первісного стану системи, або приведення її до прийнятного стану.

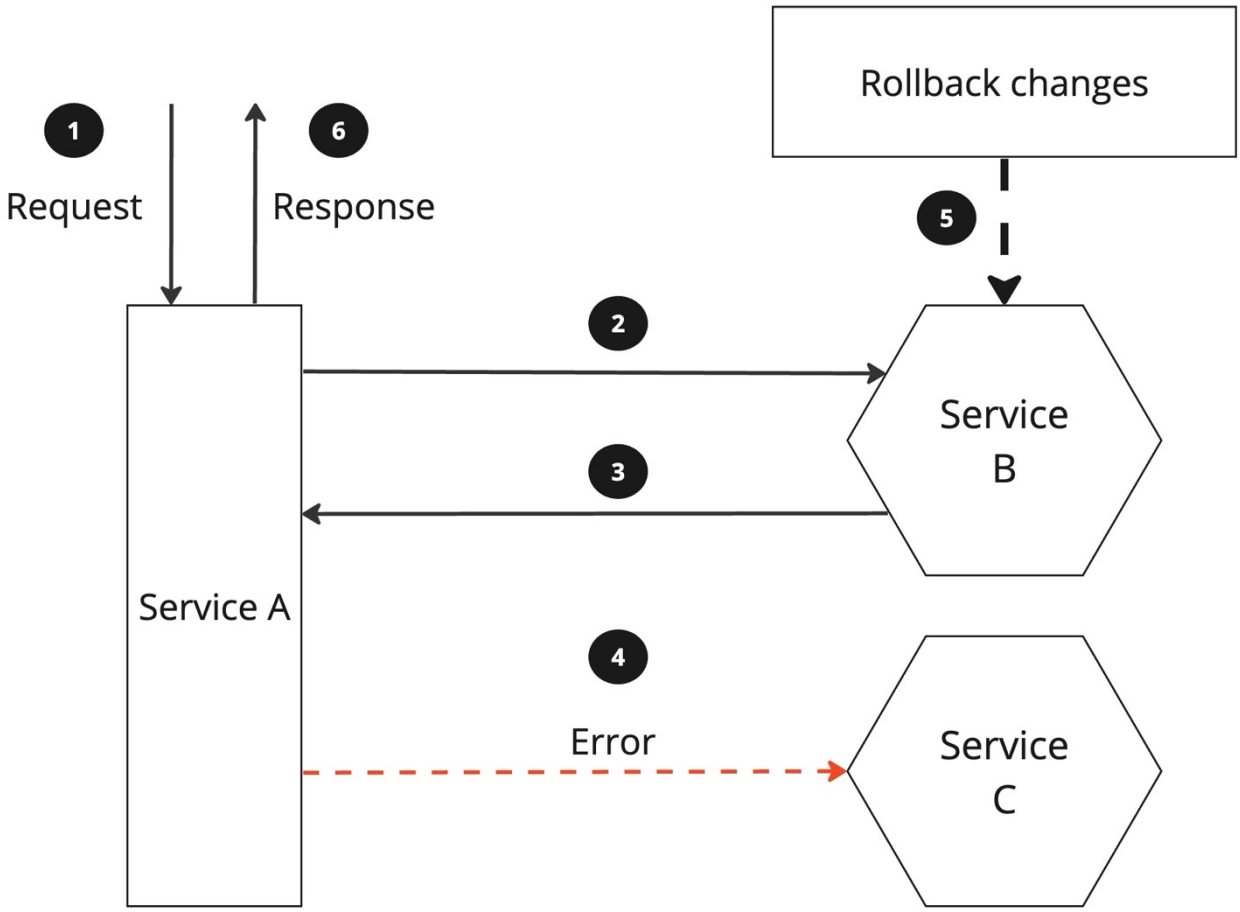

Рисунок 2.6 – Компенсаційна транзакція (Compensating Transactions)

Перевагою цього підходу є гнучкість та можливість забезпечення консистентності даних у розподілених системах без необхідності блокування ресурсів на тривалий час, що є типовим для традиційних транзакцій. Компенсаційні транзакції забезпечують механізм відновлення від помилок, що дозволяє системі продовжувати роботу, навіть коли частини процесу зазнають збоїв.

Порівняння характеристик різних стратегій забезпечення узгодженості наведені у таблиці 2.1.

Таблиця 2.1 – Порівняння стратегій забезпечення узгодженості

Стратегія	**Опис**	**Переваги**	**Недоліки**
Eventual	Допускає тимчасові невідповідності між сервісами	Висока доступність і масштабованість.	Можливість тимчасової втрати консистентності даних.
Strong	Забезпечує негайну узгодженість даних	Висока точність і надійність даних.	Зниження продуктивності та доступності системи
Sequential	Операції виконуються у визначеному порядку в усіх сервісах.	Чітке розуміння взаємодії між сервісами.	Обмежує паралелізм і можливе зниження продуктивності.
Compensating Transactions	Управління помилками, передбачає скасування операцій.	Гнучкість у відновленні, не блокує ресурси.	Складність в плануванні та відстеженні стану системи.

2.4 Використання патерну Сага для управління транзакціями в мікросервісах

Патерн «Saga» є способом управління транзакціями в розподілених системах, де кожна довготривала транзакція розбивається на серію менших, локальних транзакцій. Кожна з цих локальних транзакцій виконується в рамках окремого мікросервісу. Патерн сага забезпечує консистентність даних на високому рівні, навіть у разі помилок чи збоїв у будь-якому з мікросервісів (рис. 2.7).

Ключовою особливістю саги є використання компенсаційних транзакцій. Якщо будь-яка з локальних транзакцій зазнає невдачі, то виконуються відповідні компенсаційні дії, щоб скасувати вже виконані зміни і повернути систему до послідовного стану.

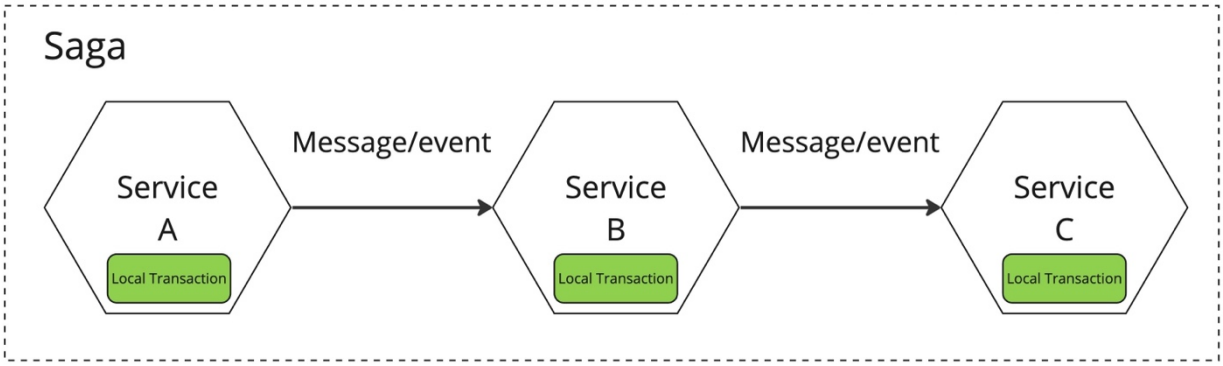

Рисунок 2.7 – Сага патерн

Такий підхід дозволяє більш гнучко управляти транзакціями, адаптуючи їх до вимог мікросервісних архітектур.

Існує два способи узгодження саг, хореографічна, де кожна локальна транзакція публікує події домену, які запускають локальні транзакції в інших службах, та оркестровка, де об’єкт повідомляє учасникам, які локальні транзакції виконувати [25].

Хореографічна сага в мікросервісних архітектурах - це підхід, де кожен сервіс у системі відповідає за власну частину більшої бізнес-операції та за відправлення подій, які сповіщають інші сервіси про зміни або необхідність виконання дій.

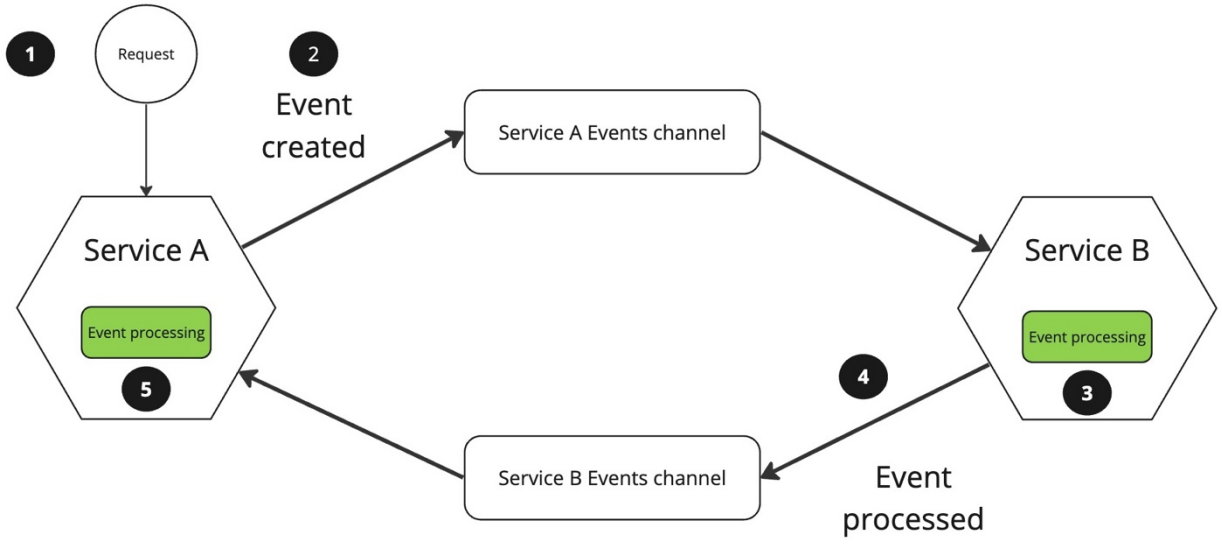

Рисунок 2.8 – Хореографічна сага в мікросервісній архітектурі

Цей підхід дозволяє сервісам функціонувати автономно, реагуючи на події замість того, щоб керуватися централізованим оркестратором. Така модель сприяє зменшенню зв’язності між сервісами, полегшуючи масштабування та модифікацію системи. Однак, це також збільшує складність управління станом та відстеження взаємодії між сервісами, особливо у великих і складних системах. Крім того, відновлення після помилок може бути складним, оскільки потрібно координувати компенсаційні дії між різними сервісами.

Оркестраційна сага використовує централізований сервіс або оркестратор для управління послідовністю та логікою транзакцій між різними сервісами. Оркестратор керує всіма етапами транзакції, від ініціації до виконання кожної операції та обробки помилок.

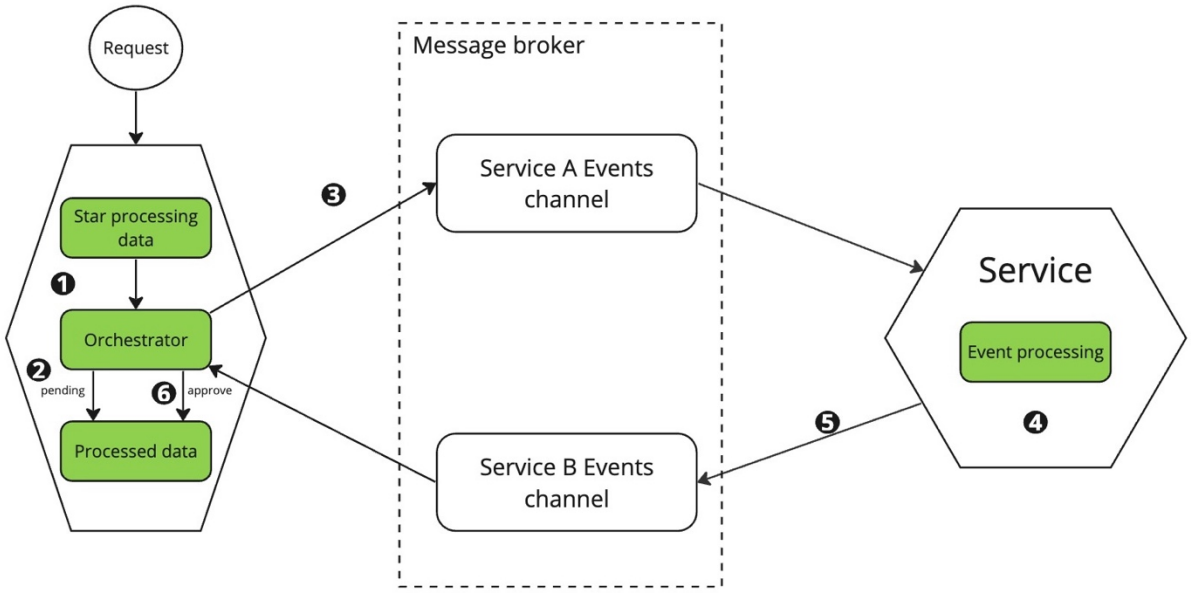

Рисунок 2.9 – Оркестраційна сага в мікросервісній архітектурі

Цей підхід дозволяє мати чіткий контроль і централізоване управління, полегшує відстеження стану транзакцій та координацію компенсаційних дій у разі виникнення помилок. Однак, він також може знизити гнучкість системи та ускладнити масштабування через залежність від одного керуючого компонента.

Оскільки сагу розбивають на окремі транзакції, для відновлення роботи після невдачі, існують стратегії відновлення, такі як зворотне відновлення та пряме відновлення [22].

Backward recovery відноситься до процесу відміни або відновлення вже виконаних кроків операції, які були успішно завершені до виникнення помилки. Цей тип відновлення спрямований на те, щоб система повернулася до попереднього коректного стану після помилки або виникнення виключення.

Forward recovery відноситься до процесу виконання альтернативних кроків або заходів після виникнення помилки з метою досягнення бажаного стану системи. Цей тип відновлення використовується тоді, коли відміна або компенсація вже виконаних кроків неможлива або неефективна.

3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ АРХІТЕКТУР ДЛЯ ВЗАЄМОДІЇ МІКРОСЕРВІСІВ

3.1 Перенесення синхронного запиту-відповіді на асинхронні події

Проаналізовано вебзастосунок для аналізу великих наборів даних, які можуть бути завантажені користувачам.

Спочатку вебзастосунок буде розгорнуто на одному сервері, який обробляє всі запити користувачів. Це означає, що коли користувач надсилає запит з даними для аналізу, сервер приймає ці дані, виконує необхідні аналітичні процедури, і повертає результати, у цей час користувач повинен чекати на завершення аналізу.

Для розгортання цього вебзастосунку використано Google Cloud Run – сервіс, який дозволяє швидко розгортати Docker контейнери у хмарі. Google Cloud Run також вміє автоматично масштабуватися, реагуючи на вхідні запити. Використання Docker контейнерів дозволяє стандартизувати та спростити розгортання застосунків. Контейнери ізолюють середовище виконання застосунку, що робить його портативним і легким у використанні на різних платформах. Також контейнеризація допомагає в ефективному використанні ресурсів та управлінні залежностями.

Для розробки HTTP-сервісу обрано мову програмування Golang.

Для створення ефективного та оптимізованого контейнера за допомогою Docker першочерговим кроком є створення Dockerfile, який є основою для побудови образу контейнера. У цьому файлі визначено серію інструкцій та команд, які необхідні для створення образу даного вебзастосунку (рис. 3.1).

Однією з ключових особливостей цього процесу є використання багатоетапної збірки, яка дозволяє мінімізувати розмір кінцевого образу контейнера без втрати необхідного функціоналу. На першому етапі збірки, використовуючи проміжний контейнер, ми здійснюємо компіляцію

бінарного файлу застосунку. Цей процес включає збірку всіх необхідних залежностей та коду застосунку. В результаті отримано скомпільований бінарний файл, готовий до виконання. Після завершення першого етапу на другому етапі бінарний файл переноситься до нового, чистого образу. Такий підхід дозволяє відокремити виконуваний файл від усіх залежностей, які були необхідні для його компіляції, але не потрібні для виконання застосунку. Це зменшує загальний обсяг кінцевого образу, спрощує його структуру та покращує безпеку, оскільки у фінальному образі відсутні непотрібні інструменти компіляції чи зайві файли. Більше того, багатоетапна збірка в Dockerfile сприяє кращому управлінню кодом, оскільки дозволяє відокремити етапи розробки, забезпечуючи чистіше та безпечніше середовище виконання для застосунку [24]. На рис. 3.1 виконується багатоетапна збірка образа Golang застосунку для обслуговування HTTP трафіку.

```
FROM golang:alpine AS builder

# Set destination for COPY
WORKDIR /app
# Copy dependencies
COPY go.* ./
# Copy the source code
COPY . ./
# Build the application
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o server .

FROM alpine AS final
WORKDIR /dist

COPY --from=builder /app/server .

# Run the application
CMD [ "/dist/server" ]
```

Рисунок 3.1 – Інструкція до створення Docker образа

Виконавши серію команд ми отримаємо розгорнутий вебзастосунок у Google хмарі (рис. 3.2).

```
docker build . -t $(IMAGE_NAME)

docker push $(IMAGE_NAME)

gcloud run deploy $(SERVICE_NAME) \
    --image $(IMAGE_NAME) \
    --platform managed \
    --allow-unauthenticated \
    --region europe-west1 \
        --project $(PROJECT)
```

Рисунок 3.2 – Створення та розгортання Docker образа

Спочатку треба сфокусуватися на визначенні показників продуктивності сервісу без безпосереднього аналізу даних. Це означає, що тестування сервісу буде у режимі, коли він зосереджений виключно на обробці вхідних HTTP запитів, не займаючись складними обчисленнями або аналізом даних. Такий підхід дозволить нам чітко зрозуміти, як сервер веде себе під навантаженням, та визначити його пропускну спроможність.

Для проведення тестування використано CLI інструмент WRK (HTTP benchmarking tool), який тримав по 25 відкритих з'єднань у 10 потоках та тривалістю 5 хвилин (рис. 3.3).

```
Running 5m test
  10 threads and 25 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    37.69ms    6.94ms 411.93ms   97.64%
    Req/Sec    53.31      7.89    60.00     81.71%
  Latency Distribution
     50%   36.85ms
     75%   38.52ms
     90%   40.45ms
     99%   49.86ms
  159672 requests in 5.00m, 40.66MB read
Requests/sec:    532.09
Transfer/sec:    138.74KB
```

Рисунок 3.3 – Результат навантаження сервісу без обробки даних

Як показано на рисунку 3.3, сервіс може обробити приблизно ~500 RPS, та у найгіршому випадку користувач буде чекати ~50 ms на відповідь.

Проведено додатковий навантажувальний тест, але цього разу половина користувачів буде завантажувати дані до обробки (рис. 3.4).

```
Running 5m test
  10 threads and 25 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     2.54s     3.52s   14.04s    79.62%
    Req/Sec    23.76     11.57    60.00     79.43%
  Latency Distribution
     50%   38.62ms
     75%    4.93s
     90%    8.61s
     99%   11.76s
  10833 requests in 5.00m, 2.78MB read
  Non-2xx or 3xx responses: 10663
Requests/sec:     36.10
Transfer/sec:      9.48KB
```

Рисунок 3.4 – Результат навантаження сервісу з запитом на обробку даних

При детальному аналізі та порівнянні результатів навантаження, взятих з рисунків 3.3 та 3.4, спостерігається значні відмінності у продуктивності та відгуку сервісу. Одним з найбільш помітних відмінностей є значне збільшення часу затримки, що спостерігається у 99-му персентилі. При аналізі даних з другого тесту, 99-й персентиль показує затримку майже у 12 секунд, тоді як у першому тесті ця цифра становила всього 50 мілісекунд. Це свідчить про те, що при високому навантаженні, сервер стає значно повільнішим у відповіді на запити.

Другим ключовим показником, який варто розглянути, є кількість запитів на секунду (Requests Per Second, RPS). У тесті з обробкою даних, RPS складає приблизно 36, що вказує на значно нижчу пропускну спроможність порівняно з першим тестом, де RPS становило близько 500.

Третім важливим аспектом є поява помилок 3xx, у даному випадку це помилка 429 (Too Many Requests). Вона вказує на те, що сервер не зміг обробити всі запити користувачів через обмеження кількості одночасних запитів, які можуть бути оброблені, бо CPU сервісу було навантажене до 100 відсотків як показано на рис 3.5. Поява таких помилок є важливим індикатором перевантаження системи, і вони вимагають відповідного реагування, як-от оптимізація сервера, масштабування інфраструктури або впровадження механізмів розподілення системи.

Важливо відзначити, що результати другого тесту мають особливе значення для планування масштабування та вибору інфраструктури.

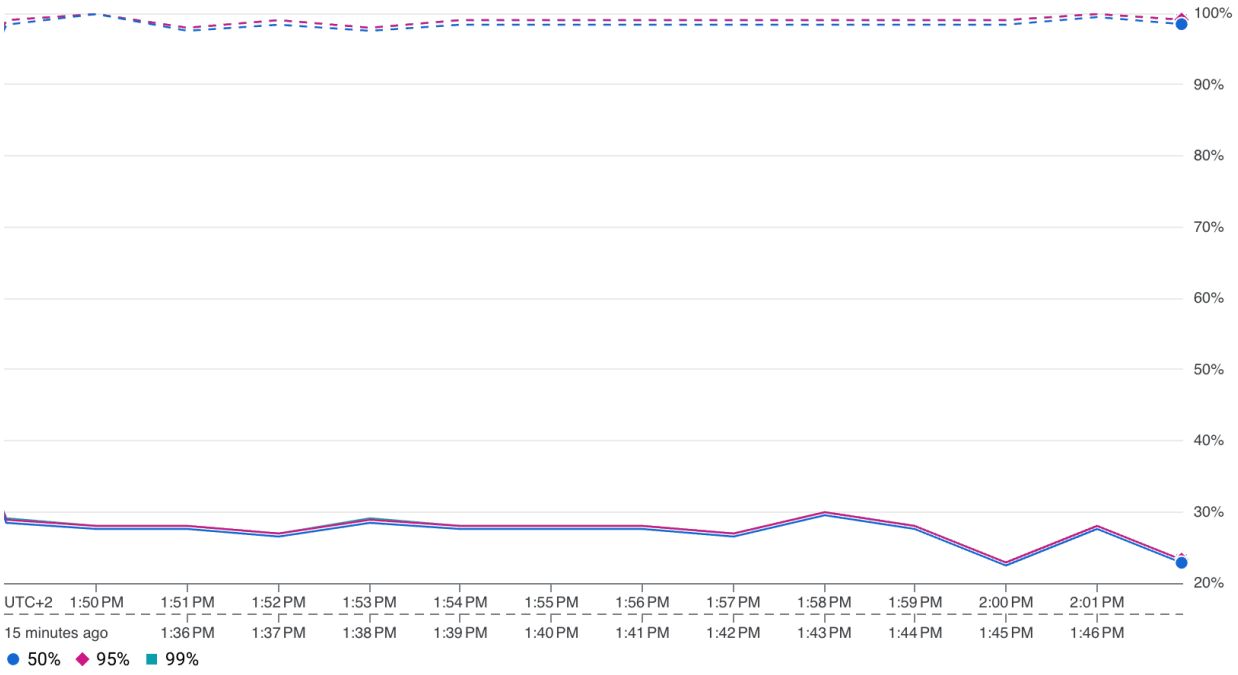

Рисунок 3.5 – Порівняння навантаження на процесор

Оскільки обробка даних у системі виконується синхронно, це призводить до ситуації, коли клієнтські запити залишаються в стані очікування на відповідь, і це створює ланцюг залежностей та затримок, де кожен елемент системи повинен чекати на завершення попередніх операцій.

Перехід від синхронного до асинхронного оброблення подій має покращити продуктивність системи. Ця зміна передбачає винесення частини

коду, відповідального за обробку даних, до окремого сервісу, що забезпечує більшу гнучкість та ефективність у виконанні завдань. Використовуючи Google Cloud PubSub сервіс для комунікації між сервісами, ми можемо значно підвищити продуктивність системи, зменшуючи залежності і затримки, які були характерні для попередньої синхронної моделі.

В асинхронному підході, коли користувач відправляє запит на обробку даних, він не блокується, а відразу переходить до виконання інших завдань. Тим часом, запит обробляється в окремому сервісі, і коли обробка завершується, результат відправляється назад користувачеві через систему обміну повідомленнями.

Pub/Sub дозволяє надійно та ефективно маршрутизувати повідомлення між різними компонентами системи. Він гарантує, що повідомлення доставляються швидко і надійно, навіть у випадку тимчасових збоїв або перебоїв у роботі одного з сервісів.

Перехід на асинхронне оброблення даних також сприяє кращому розподілу ресурсів та підвищення надійності системи. Відокремлений сервіс обробки даних може бути оптимізований під конкретні потреби цього процесу, в той час як основний сервіс може зосереджуватися на інших завданнях. Це зменшує навантаження на основний сервіс і дозволяє більш ефективно використовувати ресурси.

Після перенесення частини коду, відповідальну за обробку даних, в окремий мікросервіс, результати тесту навантаження відображені на рисунку 3.6. Отримано наступні показники: сервіс досяг 350 запитів на секунду (RPS) та максимальна затримка для користувача склала 80 мілісекунд. Ці результати є значним покращенням у порівнянні з попереднім сценарієм синхронної обробки даних, демонструючи ефективність переходу на асинхронні запити.

```
Running 30s test
  10 threads and 25 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    58.13ms    6.76ms 144.46ms   87.22%
    Req/Sec    34.28      7.23    40.00     87.37%
  Latency Distribution
     50%   56.33ms
     75%   59.85ms
     90%   65.39ms
     99%   83.61ms
  10296 requests in 30.04s, 3.07MB read
Requests/sec:    342.72
Transfer/sec:    104.76KB
```

Рисунок 3.6 – Результат навантаження сервісу з асинхронною обробкою даних

Перенесення обробки даних в окремий мікросервіс дозволило розподілити навантаження між різними частинами системи і навіть при 100 відсотковому навантаженні сервісу по обробки даних, помилки відсутні (рис. 3.7).

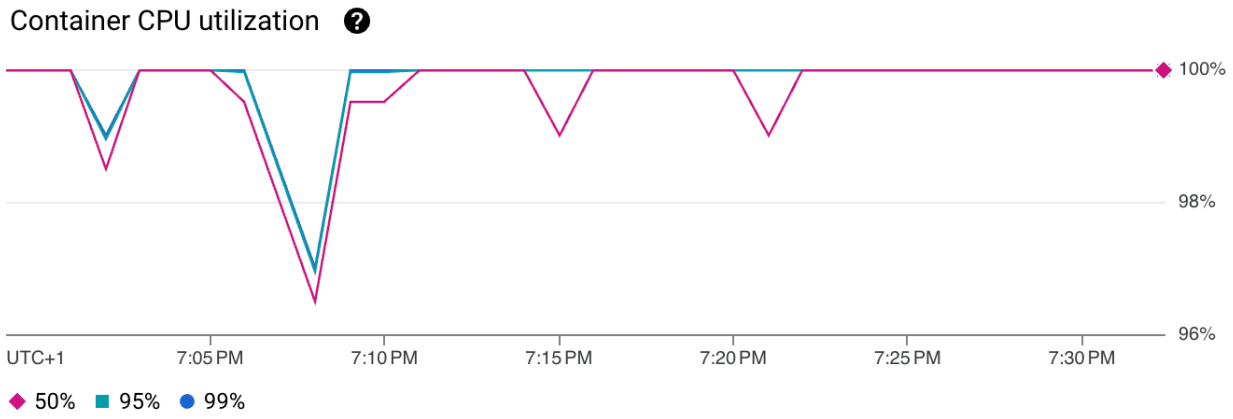

Рисунок 3.7 – Результат використання ЦП з асинхронною обробкою даних

Крім того, сервіси можливо масштабувати окремо, так як для обробки запитів потребується менше CPU часу, ніж для обробки великих даних.

В цілому результати тестування навантаження після впровадження мікросервісної архітектури підтвердили, що такий підхід може покращити

загальну продуктивність та ефективність системи. З одного боку для користувача системи час обробки даних не змінився, тобто час від відвідування ресурсу до отримування результатів приблизно залишився на такому же рівні. Проте, з точки зору системи, було досягнуто значного покращення у відмовостійкості, особливо під час високого навантаження.

3.2 Реалізація оркестранта у розподілених системах

Оркестрант у контексті розподілених систем виконує роль центрального керуючого елемента, який координує взаємодію між множиною мікросервісів, забезпечуючи їх правильну взаємодію та послідовність виконання задач відповідно до визначеної бізнес-логіки [24]. Використовуючи оркестранта, можна створити комплексні робочі процеси, які автоматизують багатоетапні операції, забезпечуючи їх надійність та ефективність.

Важливим аспектом при реалізації оркестранта є управління станом і збереженням контексту робочих процесів. Це може бути досягнуто за допомогою баз даних або інших засобів зберігання стану, що дозволяє відстежувати прогрес виконання завдань та відновлювати процеси у випадку збоїв.

За допомоги мови програмування Go (Golang) був реалізований базовий оркестрант, що демонструє фундаментальні аспекти реалізації патерну оркестрації (рис. 3.8). Цей оркестрант включає дві основні структури, які спрямовані на керування поетапною обробкою задач, демонструючи взаємодію між різними мікросервісами у розподіленій системі. Основний метод Run керує послідовним викликом мікросервісів, забезпечуючи правильне виконання задачі від початку до кінця та має здатність впроваджувати компенсаційні дії у випадку виявлення помилок або збоїв у будь-якому з мікросервісів. Це означає, що якщо під час виконання одного з

етапів виникає проблема, оркестрант може вжити заходів для відновлення системи до стабільного стану або виконати альтернативні дії для мінімізації наслідків помилки (рис. 3.8).

```
type (
        // Step represents a single step in orchestrator
        Step struct {
                Execute    func(id string) error
                Compensate func(id string, err error) error
        }

        // Orchestrator manages the execution of steps
        Orchestrator struct {
                steps []Step
        }
)

// AddStep adds a new step to the orchestrator
func (s *Orchestrator) AddStep(step Step) {
        s.steps = append(s.steps, step)
}

// Run executes all steps in the orchestrator
func (s *Orchestrator) Run(id string) error {
        for _, step := range s.steps {
                if err := step.Execute(id); err != nil {
                        return step.Compensate(id, err)
                }
        }

        return nil
}
```

Рисунок 3.8 – Проста реалізація оркестранта

Щоб ініціалізувати оркестрант, необхідно встановити послідовність етапів виконання, розробити відповідну логіку для кожного кроку та розробити механізми компенсації для ситуацій, коли виникають помилки. На рисунку 3.9 відображена ініціалізація оркестранта, додано два кроки, де в першому кроку робиться запит у сторонній сервіс та при помилці процес завершується поверненням помилки користувачу, але у другому кроку, робиться компенсація першого, а саме очищення збережених даних після першого кроку та зміна статусу виконання у базі даних.

```
// Define steps
var orchestrator Orchestrator

// Send data to preprocess service for cleaning and validation
orchestrator.AddStep(Step{
        Execute: func(id string) error {
                _, err := http.NewRequestWithContext(ctx, "POST", "", &buf)
                if err != nil {
                        return fmt.Errorf("failed to send data: %w", err)
                }
                return nil
        },
        Compensate: func(id string, err error) error {
                return fmt.Errorf("preprocess service compensating: %w", err)
        },
})

// Send data to data analysis service for analysis and prediction
orchestrator.AddStep(Step{
        Execute: func(id string) error {
                return publishMsg("data-analysis-service")
        },
        Compensate: func(id string, err error) error {
                // Remove preprocessed data
                s.preprocess.Purge(id)
                // Refresh records in database
                s.db.SetStatus(id, DATA_STATUS_FAILED)
                return fmt.Errorf("data analysis service compensating: %w", err)
        },
})

return orchestrator.Run(id)
```

Рисунок 3.8 – Ініціалізація оркестранта

3.3 Забезпечення надійності в розподілених системах

У контексті забезпечення надійності важливо враховувати, що навіть надійні сервіси, такі як Google Cloud PubSub, не можуть гарантувати 100 відсотковий час безперебійної роботи. Тому необхідно передбачити ситуації, коли публікація запиту користувача у PubSub може бути неможливою. Для вирішення цього питання можна застосувати ряд патернів і стратегій, які допоможуть підтримувати високий рівень надійності та доступності системи, навіть під час перебоїв у роботі зовнішніх сервісів.

Перший варіант – застосувати патерн Time-Outs [22], який скасує очікування відповіді від стороннього сервісу якщо відповідь не надходить

протягом визначеного часу, що дозволяє системі уникнути надмірного очікування та забезпечити продовження роботи. Для його реалізації у мові програмування Go є вбудований механізм, context, який було конфігуровано на рисунку 3.9, якщо сервіс не відповів за 500 мілісекунд, операція відміняється.

```
// Create a new context with a timeout of 500 milliseconds
ctxt, cancel := context.WithTimeout(ctx, 500*time.Millisecond)

// Defer the cancellation of the context
defer cancel()

// Make a POST request to the preprocess service with timeout
_, err := http.NewRequestWithContext(ctxt, "POST", "", &buf)
if err != nil {
        return fmt.Errorf("failed to send data: %w", err)
}
```

Рисунок 3.9 – Варіант виконання Time-Outs

У випадках, коли є необхідність все ж таки отримати результат від стороннього сервісу, особливо якщо припускаються мережеві помилки, тимчасові проблеми з доступністю сервісу, втрати мережевих пакетів, або через раптові стрибки в навантаженні, повторне виконання таких запитів може бути ефективним рішенням [9, 24], тому що результат буде отримано хоча і с затримкою. Тому реалізовано патерн Retries, який повторює невдалу спробу відправки даних у випадку помилок, таких як 503 (Service Unavailable) або 504 (Gateway Time-out), які зазвичай вказують на тимчасові проблеми, що робить повторні спроби цілком виправданими (рис. 3.10). Важливо розуміти, що не всі помилки підлягають повторному виконанню. Наприклад, помилка з HTTP статусом 404 (Not Found) зазвичай не вимагає повторної спроби, оскільки шанси на успішний результат мінімальні.

```
// retryWithBackoff retries the given function with exponential backoff
func retryWithBackoff(fn func() error, maxRetries int, initial time.Duration) error {
        for i := 0; i < maxRetries; i++ {
                if err := fn(); err == nil {
                        return nil
                }

                // Exponential backoff
                // 1 << i is bit shifting, i.e. 1 << 1 = 2, 1 << 2 = 4
                backoff := initial * time.Duration(1<<i)
                time.Sleep(backoff)
        }

        return fmt.Errorf("operation failed after retries")
}
```

Рисунок 3.10 – Базова реалізація Retries патерну

Також впроваджено затримку перед повторною спробою, щоб запобігти бомбардування сервісу і не погіршити ситуацію, тому що без затримки постійне надсилання нових запитів збільшує навантаження на вже перевантажений сервіс. Реалізована експоненційна затримка (3.1), при якій час очікування між спробами збільшується кожну ітерацію.

$$BackOff = InitialDelay\ x\ 2^{(RetryCount)}. \quad (0.1)$$

Ще одним ефективним підходом є використання внутрішньої черги повідомлень [6, 22]. Якщо публікація в PubSub чи відповідь від стороннього сервісу є неможливою, якщо затримка між повторними викликами очікується значною, система може тимчасово зберігати дані в локальній черзі. Та після відновлення роботи, повідомлення з черги можна буде послідовно відправити. Це забезпечує додатковий рівень відмовостійкості та гарантує, що дані не будуть втрачені під час перебоїв у роботі зовнішніх сервісів.

Реалізовано чергу повідомлень, де тимчасово зберігаються повідомлення у пам’яті сервісу (рис. 3.11). Queue структура може зберігати повідомлення у RAM та відтавати при запиті.

```
// Queue is a simple queue structure
// for storing unprocessed messages
type Queue struct {
        messages []string
        lock     sync.Mutex
}

// AddMessage adds a new message to the queue
func (q *Queue) AddMessage(message string) {
        q.lock.Lock()
        defer q.lock.Unlock()
        q.messages = append(q.messages, message)
}

// RetrieveMessage retrieves a message from the queue
func (q *Queue) RetrieveMessage() string {
        q.lock.Lock()
        defer q.lock.Unlock()
        if len(q.messages) == 0 {
                return ""
        }
        message := q.messages[0]
        q.messages = q.messages[1:]
        return message
}
```

Рисунок 3.11 – Базова реалізація черги

У сценаріях, коли сервіс зазнає надмірного навантаження або перебуває в стані збою, продовження прийому нових запитів може погіршити ситуацію, знижуючи ймовірність швидкого відновлення сервісу до нормального стану, навіть при використанні експоненційній затримці. Це може призвести до ланцюгових реакцій збоїв, які негативно впливають на загальну продуктивність системи.

Circuit Breaker патерн працює як захисний механізм, що тимчасово вимикає виклики сервісу у випадку виявлення проблем, тим самим запобігаючи подальшому надходженню запитів (рис. 3.12) [1, 2, 13, 22, 23]. Цей процес дозволяє сервісу отримати необхідний час для відновлення, не будучи перевантаженим новими запитами. Коли Circuit Breaker активується,

він переходить у стан «відкритого» ланцюга, і всі подальші запити до сервісу відхиляються або обробляються згідно з встановленою політикою.

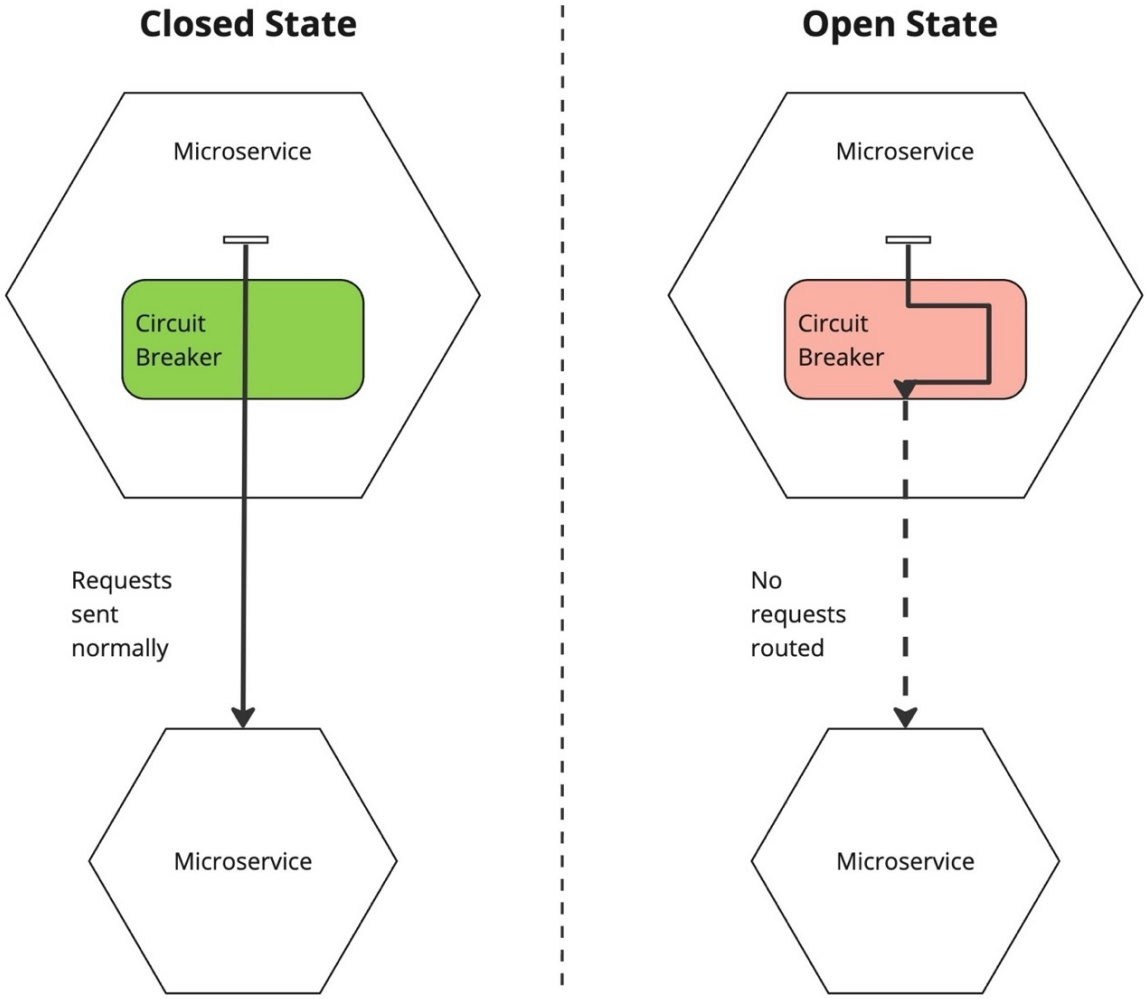

Рисунок 3.12 – Застосування автоматичного вимикача

Однією з ключових особливостей патерну «Circuit Breaker» є його здатність адаптуватися до різних умов. Наприклад, відповідати зі статусом 200, але з повідомленням про те, що всі воркери зайняті, яка може бути надіслана користувачу, щоб повідомити про тимчасову неможливість обробки запитів. Це забезпечує краще розуміння ситуації з боку користувачів і знижує вірогідність негативного сприйняття проблем сервісу.

На рисунку 3.13 представлена реалізація патерну «Circuit Breaker», де спочатку він знаходиться у закритому стані. У цьому стані, якщо виникають три послідовні помилки у відповідях від викликів зовнішнього сервісу,

«Circuit Breaker» активується, переходячи у відкрите положення. У відкритому стані він тимчасово припиняє відправлення нових запитів до сервісу і надсилає користувачам повідомлення про те, що ресурси в даний момент зайняті, та рекомендує спробувати знову пізніше. В такому стані він буде знаходитись 30 секунд, після чого знову закриється. Також скрипт можливо модифікувати та додати ще одне положення, напіввідкритий, у якому після часу очікування, буде спроба виклику зовнішнього сервісу і при першої ж помилки він відкриєтеся або повністю закриєтеся якщо помилок не буде.

```
// Execute manages the execution of the operation and cb states.
func (cb *CircuitBreaker) Execute(operation func() error) error {
        cb.mu.Lock()
        defer cb.mu.Unlock()

        if cb.state == StateOpen {
                return fmt.Errorf("Circuit Breaker is open")
        }

        if err := operation(); err != nil {
                cb.failureCount++
                if cb.failureCount >= cb.failureThreshold {
                        cb.state = StateOpen
                        go cb.reset()
                }
                return err
        } else {
                cb.failureCount = 0
        }

        return nil
}

// reset changes the state of the circuit breaker to closed
func (cb *CircuitBreaker) reset() {
        time.Sleep(cb.resetTimeout)
        cb.mu.Lock()
        defer cb.mu.Unlock()
        cb.state = StateClosed
        cb.failureCount = 0
}
```

Рисунок 3.12 – Керування станом Circuit Breaker

Реалізовані патерни стабільності для підтримки узгодженості даних у мікросервісах є кращими практиками для розробки надійних та відмовостійких систем. Кожен з цих патернів має своє місце у мікросервісній архітектурі та повинен використовуватися під конкретні вимоги бізнес процесів.

ВИСНОВКИ

У рамках кваліфікаційної роботи детально проаналізовано переваги та недоліки застосування мікросервісної архітектури, особливу увагу надано міжпроцесорній комунікації у мікросервісах. Ефективна комунікація між мікросервісам є вкрай необхідною для стабільного функціонування всього продукту. Асинхронні запити підвищують пропускну спроможність системи, додаючи гнучкості та водостійкості.

Також розглянуті різнобічні варіанти масштабування мікросервісів, який дозволяють розумно використовувати потужності та заощаджувати витрати на утримання серверів. Найефективнішим з розглянутих стратегій масштабування являється діагональне масштабування за рахунок можливості масштабуватися у всіх площинах.

Застосовані на практиці патерни відмовостійкості, такі як «Circuit Breaker», «Retries with Backoff», «Saga Orchestrator», «Memory Queue» та «Time-Outs». Впровадження цих патернів дозволяє розробляти надійні та оптимальні системи. Circuit Breaker є ефективним патерном у розподілених системах що може вимикати подальші запити до перенавантаженого сервісу, тим самим відіграючи роль запобіжника. Простіший патерн «Retries» застосовується у випадках, коли ми очікуємо тимчасові мережеві помилки і треба повторити спробу через деякий час. «Saga Orchestrator» керує локальними транзакціями, та має компенсуючі дії у разі помилок, а «Time-Outs» перериває тривалє очікування відповідей від зовнішніх сервісів.

Результати дослідження підтвердили, що впровадження асинхронної комунікації та застосування архітектурних рішень підвищує гнучкість системи, дозволяючи сервісам масштабуватись окремо один від одного, та підвищити загальну продуктивність та відмовостійкість.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Newman, S. (2021). Building Microservices, Second Edition. O’Reilly Media.

2. Richardson, C. (2018). Microservices patterns: With examples in Java. Manning Publications.

3. Ford, N., & Gifford, I. (2021). Fundamentals of Software Architecture: An Engineering Approach. O’Reilly Media.

4. Fowler, S. (2020). Production-ready microservices: Building standardized systems across an engineering organization. O’Reilly Media.

5. Amundsen, M., Dietrich, D., & Mitra, R. (2021). Microservices Up and Running: A Step-By-Step Guide to Building a Microservices Architecture. O'Reilly Media.

6. Urquhart, J. (2021). Flow architectures: The future of streaming and event-driven integration. O’Reilly Media.

7. Atchison, L. (2016). Architecting for scale: High availability for your growing application. O’Reilly Media, Inc.

8. Carnell, J. (2019). Spring Microservices in Action, Second Edition. Manning Publications.

9. Proffitt, J., & Anami, R. (2023). Becoming a Rockstar SRE. Packt.

10. Luksa, M. (2020). Kubernetes in Action, Second Edition. Manning Publications.

11. Alapati, S. (2020). Kubernetes Cookbook: Practical Solutions to Container Orchestration, 3rd Edition. O’Reilly Media.

12. Richards, M. (2020). Software Architecture: The Hard Parts. O'Reilly Media.

13. Newman, S. (2020). Monolith to microservices: Evolutionary patterns to transform your monolith. O’Reilly Media.

14. García, M. M., & Anglin. (2020). Learn Microservices with Spring Boot. Apress.

15. Hohpe, G. (2021). The Software Architect Elevator: Redefining the Architect’s Role in the Digital Enterprise. O’Reilly Media.

16. Davis, C. (2019). Cloud Native Patterns: Designing Change-Tolerant Software. Simon and Schuster.

17. Burns, B. (2018). Designing distributed systems: Patterns and paradigms for scalable, reliable services. O'Reilly Media.

18. Saurabh, S. (2020). Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud. Apress.

19. Wolff, E. (2020). Microservices: A Practical Guide: Principles, Concepts, and Recipes. Springer.

20. Varghese, S. (2015). Web development with Go: Building scalable web apps and RESTful services. Apress.

21. Pigazzini, I. (2021). Building Micro-Frontends: Scaling Developers in the Enterprise. O’Reilly Media.

22. Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media.

23. Urquhart, J. (2021). Flow Architectures: The Future of Streaming and Event-Driven Integration. O’Reilly Media.

24. Build your Go image. URL: https://www.researchgate.net/figure/the-flow-of-improved-BRISK-algorithm_fig1_328946366 (дата звернення 10.10.2023).

25. A pattern language for microservices. URL: https://microservices.io/patterns/index.html (дата звернення 15.10.2023).

26. Гороховатський, В.О., Путятін, Е.П. (2008). Структурне розпізнавання зображень на основі моделей голосування ознак характерних точок. Реєстрація, зберігання і обробка даних, Т.10. №4, С. 75–85.

27. Гороховатский, В.А., Путятин, Е.П., Столяров В.С. (2017). Дослідження результативності структурних методів класифікації зображень, Радіоелектроніка, інформатика, управління, №3 (42), С. 78–85.

28. Гороховатський В.О., Пупченко Д.В., Солодченко К.Г. (2018) Аналіз властивостей, характеристик та результатів застосування новітніх детекторів для визначення особливих точок зображення. Системи управління, навігації та зв'язку. С. 93–98.

29. Gorokhovatskyi V., Gadetska S., Ponomarenko R. (2020) Recognition of Visual Objects Based on Statistical Distributions for Blocks of Structural Description of Image. Proc. of the XV Int. Scientific Conference "Intellectual Systems of Decision Making and Problems of Computational Intelligence" (ISDMCI'2019), Ukraine, May 21–25, 2019, pp. 501-512.

30. Pomazan V., Tvoroshenko I., and Gorokhovatskyi V. (2023) Development of an application for recognizing emotions using convolutional neural networks, International Journal of Academic Information Systems Research, 7(7), 25–36.

31. Гороховатський В.О. (2014) Структурний аналіз та інтелектуальне оброблення даних в комп'ютерному зорі: моногр., СМИТ, 316с.

32. Гороховатський, В. О, Передрій, О. О., Творошенко, І. С., & Марков, Т. Є. (2023). Матриця відстаней для множини компонентів структурного опису як інструмент для створення класифікатора зображень.

33. Gadetska S., Gorokhovatskyi V., Stiahlyk N., Vlasenko N. (2022) Aggregate Parametric Representation of Image Structural Description in Statistical Classification Methods. In CEUR Workshop Proceedings: Computer Modeling and Intelligent Systems (CMIS-2022), 3137, pp. 68-77.

34. Gorokhovatskyi, O., Peredrii, O., Gorokhovatskyi, V., Vlasenko, N. (2023) Explanation of CNN Image Classifiers with Hiding Parts. In: J. Benois-Pineau, R. Bourqui, D. Petkovic, G. Quenot (eds), Explainable Deep Learning Artificial Intelligence, pp. 125-146, Academic Press, 346 p.

35. Gorokhovatskyi, V., Tvoroshenko, I., Kobylin, O., & Vlasenko, N. (2023). Search for visual objects by request in the form of a cluster representation for the structural image description. Advances in Electrical and Electronic Engineering, 21(1), 19-27.

36. Gorokhovatskyi, V., Vlasenko, N. (2021). Редукція опису зображення у складі множини дескрипторів на основі метричного критерію інформативності. Advanced Information Systems, 5(4), pp. 10-16.

37. Гороховатський В., Творошенко І., Сидоренко Д. (2021) Класифікація зображень із використанням кластерного подання, Міжн. наук. симпозіум Інтелектуальні рішення-С. Обчислювальний інтелект. Теорія прийняття рішень: праці міжн. наук. симп. (Вересень 29, 2021). Київ – Ужгород, 44-45.

38. Gorokhovatskyi V., Tvoroshenko I. (2023) Identification of visual objects by the search request. International scientific symposium «INTELLIGENT SOLUTIONS-S». Computational intelligence (results, problems and perspectives). Decision making theory: proceedings of the international symposium, September 28, 2023, Kyiv-Uzhorod, Ukraine, pp. 25-27.

39. Tvoroshenko, I. S., & Tabashnyk, V. A. (2018). Development of a spatial model of geoinformation support for people with disabilities in wheelchairs in Kharkiv. Collection of scientific works of KhNUPS, 1(55), 122-128.

40. Pomazan V., Tvoroshenko I., Gorokhovatskyi V. (2023) Handwritten character recognition models based on convolutional neural networks, International Journal of Academic Engineering Research, 7(9), pp. 64-72.

41. Tvoroshenko I., Gorokhovatskyi V., Kobylin O., and Tvoroshenko A. (2023) Application of deep learning methods for recognizing and classifying culinary dishes in images, International Journal of Academic and Applied Research, 7(9), pp. 57-70.