Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна
Навчально-науковий інститут комп'ютерних наук та штучного інтелекту

Рекомендовано до захисту:
протокол засідання кафедри
№ ___ від ____.____.2025 р.
Завідувач кафедри КІПСіТ____
_____ Олешко О.І.
(підпис) (прізвище та ініціали)

Пояснювальна записка

до кваліфікаійної роботи бакалавра

на тему «Аналіз проектних патернів та інструментальних засобів для узгодженості даних у мікросервісних застосунках»

Захищено на засіданні ЕК № _____
протокол № ____ від ____.____.2019 р.
Оцінка _____/ _____
Голова ЕК
_____
(підпис) (прізвище та ініціали)

Виконав:
студент 4 курсу, групи КС- 41
Спеціальності:
122 – Комп’ютерні науки_____
(шифр і назва спеціальності підготовки)
Жестік Марія Вячеславівна_____
(прізвище, ім’я та по батькові, підпис)
Керівник ст. викладач Паршенцев Б. В.
(ступень, звання, прізвище та ініціали, підпис)
Рецензент зав. кафедри, кандидат, ст. дослідник, доцент Хруслов М. М.
(ступень, звання, прізвище та ініціали, підпис)

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Навчально-науковий інститут комп'ютерних наук та штучного інтелекту
Кафедра інтелектуальних програмних систем і технологій
Рівень вищої освіти (освітньо-кваліфікаційний рівень) бакалавр
Спеціальність 122 – Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри ІПСіТ

_____ Олег ОЛЕШКО
підпис ім'я, прізвище

"____" _____2025 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Жетік Марія Вячеславівна
(прізвище, ім'я, по батькові студента)

1. Тема роботи: «Аналіз проектних патернів та інструментальних засобів для узгодженості даних у мікросервісних застосунках».

керівник роботи ст. викладач КІПСіТ Паршенцев Богдан Володимирович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "16" квітня 2025 року № 4101-5/962

2. Строк подання студентом роботи 05.06.2025 р.

3. Перелік питань, які потрібно розробити:

ознайомитись з принципами мікросервісної архітектури та визначити основні підходи даних у розподілених системах, провести аналіз підходів до забезпечення узгодженості а також патернів, дослідити інструментальні засоби для реалізації патернів узгодженості, розробити архітектуру мікросервісного застосунку з використанням відповідних патернів і сервісів, Реалізувати прототип системи з інтеграцією обраних рішень щодо узгодженості, забезпечити логування, моніторинг та CI/CD, Провести тестування системи при типових сценаріях та відмовах, оцінити ефективність використаних патернів та інструментів.

4. План роботи

№ з/п	Назви етапів роботи
1	Формулювання мети, об'єкта, предмета дослідження та повного переліку завдань для виконання кваліфікаційної роботи (КР).
2	Пошук, відбір і аналіз наукових публікацій та технічної документації за темою узгодженості даних у мікросервісах.
3	Ознайомлення з принципами мікросервісної архітектури та особливостями розподілених систем.
4	Аналіз проблем узгодженості даних у МСА: сильна, слабка та остаточна узгодженість.
5	Дослідження проектних патернів забезпечення узгодженості даних: Saga (оркестрація/хореографія), Event Sourcing, CQRS.
6	Вибір інструментів і технологій для реалізації патернів.
7	Проектування архітектури прикладного мікросервісного застосунку з урахуванням узгодженості даних.
8	Реалізація прототипу застосунку з впровадженням трьох видів узгодженості та обраних патернів.
9	Проведення експериментального тестування на відповідність системи вимогам до узгодженості при типових сценаріях та збоях.
10	Аналіз отриманих результатів, оцінка ефективності обраних підходів та підготовка документації для захисту КР.

5. Дата видачі завдання 10.02.2024 р.

Студент _____ Жестік М.В.
(підпис) (ініціали, прізвище)

Керівник роботи _____ Паршенцев П.В.
(підпис) (ініціали, прізвище)

АННОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи на тему «Аналіз проектних патернів та інструментальних засобів для узгодженості даних у мікросервісних застосунках» складається з 51 сторінки та містить 3 розділи, 3 рисунки та список з 25

Основною метою цієї кваліфікаційної роботи є розробка мікросервісного програмного забезпечення, в якому реалізовано сучасні патерни забезпечення узгодженості даних у розподілених системах. У роботі розглядається актуальна проблема забезпечення цілісності та коректності даних у розподіленому середовищі, де кожен мікросервіс є автономним, з власним сховищем даних та логікою обробки, що унеможливлює використання класичних підходів до транзакцій, як у монолітних додатках. Дослідження фокусується на проблематиці підтримки консистентного стану даних у мікросервісній архітектурі, де кожен сервіс має власне джерело істини, а загальна транзакційність системи не може бути забезпечена традиційними методами.

У роботі проаналізовано сучасні підходи до реалізації транзакцій у мікросервісах та розглянуто ключові патерни, що забезпечують узгодженість: Saga (зокрема варіанти з оркестрацією та хореографією), Event Sourcing та CQRS. Визначено основні виклики, пов’язані з асинхронною обробкою, дублікатами повідомлень, забезпеченням ідемпотентності та контролем цілісності бізнес-логіки в умовах відмов. Застосунок реалізовано із використанням кількох типів узгодженості даних — сильної, слабкої та остаточної — з використанням сучасних інструментів, зокрема брокера повідомлень Apache Kafka, технології контейнеризації Docker та архітектурного стилю REST API.

В рамках практичного розділу створено мікросервісну систему з доменами «Замовлення», «Склад» та «Master Data», у якій реалізовано механізми обміну подіями, керування життєвим циклом транзакцій та

запобігання дублюванню повідомлень. Система дозволяє не лише забезпечити консистентність за допомогою подієвої взаємодії, але й масштабувати окремі компоненти без втратиузгодженості. Проведено тестування поведінки системи при відмовах окремих сервісів та перевірку сценаріїв відновлення стану, що підтвердило стабільність, надійність та відповідність архітектури поставленим вимогам. Результати роботи демонструють ефективність обраних архітектурних підходів у побудові надійних, масштабованих та узгоджених мікросервісних систем, здатних функціонувати в умовах розподіленого середовища.

Ключові слова: МІКРОСЕРВІСИ, УЗГОДЖЕНІСТЬ ДАНИХ, SAGA, CQRS, EVENT SOURCING, KAFKA, ТРАНЗАКЦІЇ, REST API, DOCKER.

ABSTRACT

Explanatory Note for the qualification work on the topic "Development of an Intelligent System for Generating Test Questions Based on Given Content" consists of 513 pages and includes 3 chapters, 3 figures and a list of 25 sources.

The primary goal of this qualification work is the development of microservice-based software that implements modern data consistency patterns in distributed systems. This work addresses the pressing challenge of maintaining data integrity and correctness in distributed environments, where each microservice operates autonomously, with its own data store and processing logic—making traditional transaction management approaches, typical of monolithic applications, inapplicable. The research focuses on the problem of ensuring a consistent system state in a microservice architecture, where each service maintains its own source of truth, and global transactional guarantees cannot be achieved using classical methods.

This work analyzes contemporary approaches to transaction management in microservices and explores key consistency patterns: Saga (including both orchestration and choreography variations), Event Sourcing, and CQRS. It identifies the main challenges associated with asynchronous processing, message duplication, idempotency, and maintaining business logic integrity in failure scenarios. The application implements various types of data consistency — strong, weak, and eventual — using modern tools such as the Apache Kafka message broker, Docker containerization technology, and REST API architectural style.

As part of the practical section, a microservice system was developed with "Order", "Inventory", and "Master Data" domains. It implements event-driven communication, transaction lifecycle management, and deduplication mechanisms. The system ensures consistency through event-based interactions while supporting the independent scaling of its components. Testing was conducted to assess system behavior during individual service failures and state recovery scenarios, demonstrating the architecture’s stability, reliability, and compliance with the

specified requirements. The results confirm the effectiveness of the chosen architectural approaches in building reliable, scalable, and consistent microservice systems capable of operating in distributed environments.Keywords: MICROSERVICES, DATA CONSISTENCY, SAGA, CQRS, EVENT SOURCING, KAFKA, TRANSACTIONS, REST API, DOCKER.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ЗНАЧЕНЬ, ТЕРМІНІВ

ACID (Atomicity, Consistency, Isolation, Durability) – набір властивостей транзакцій у реляційних базах даних, що гарантує коректність операцій.УПМ – Управління продовольства та медикаментів

BASE (Basically Available, Soft state, Eventually consistent) – модель послідовної узгодженості даних, що часто використовується в розподілених системах на відміну від ACID.

Saga Pattern – патерн координації довготривалих транзакцій у розподілених системах через послідовність локальних транзакцій, які взаємодіють через події.

CQRS (Command Query Responsibility Segregation) – архітектурний патерн, який розділяє операції читання та запису даних для покращення масштабованості та продуктивності.

Event Sourcing – архітектурний підхід, за якого стан системи зберігається як послідовність подій, що відображають зміни.

Kafka (Apache Kafka) – розподілена платформа обробки потокових даних, яка забезпечує публікацію, підписку та зберігання повідомлень.

REST (Representational State Transfer) – архітектурний стиль для створення веб-сервісів із використанням HTTP-протоколу.

Docker – платформа для контейнеризації застосунків, яка дозволяє створювати, розгортати та масштабувати сервіси в ізольованому середовищі.

Message Broker – проміжне програмне забезпечення (наприклад, Kafka), що дозволяє сервісам обмінюватися повідомленнями асинхронно.

ВСТУП

Мікросервісна архітектура поступово стала одним з найпоширеніших підходів у розробці масштабованих програмних систем. Вона дозволяє структурувати складні застосунки як набір незалежних сервісів, кожен з яких виконує окрему бізнес-функцію. Такий підхід відкриває нові можливості для масштабування, розподілу відповідальності між командами та швидкого впровадження нових функціональностей. Водночас із перевагами з'являються і нові виклики, серед яких забезпечення узгодженості даних між окремими мікросервісами залишається однією з ключових проблем сучасної розробки. У централізованих монолітних системах проблема консистентності зазвичай вирішується класичними ACID-транзакціями, проте в умовах мікросервісної архітектури, де кожен сервіс має свою базу даних і працює незалежно, такі підходи вже не є ефективними або взагалі неможливими.

Проблема консистентності даних в умовах розподілених обчислень набуває особливого значення в доменах, де точність і надійність критично важливі — наприклад, у фінансових системах, електронній комерції чи медичних інформаційних платформах. У таких випадках навіть незначна неузгодженість даних між сервісами може призвести до серйозних помилок, втрати даних або порушення логіки бізнес-процесів. Згідно з дослідженням RedHat [1], більшість компаній, що переходять на мікросервісну архітектуру, стикаються з труднощами в реалізації транзакцій між сервісами, оскільки класичні методи виявляються непридатними або занадто дорогими в реалізації.

На цьому тлі набувають поширення спеціалізовані архітектурні патерни, які дозволяють зберігати консистентність у розподілених системах без використання централізованих транзакцій. Серед таких підходів особливу увагу привертають Saga, Event Sourcing та Command Query Responsibility Segregation (CQRS). Вони використовуються для побудови транзакцій, що охоплюють кілька сервісів, та дозволяють мінімізувати наслідки збоїв. Так,

патерн Saga забезпечує поступове оновлення стану системи через низку локальних транзакцій з компенсуючими діями у разі невдачі. Event Sourcing фіксує кожну зміну у вигляді подій, що дозволяє відтворити будь-який стан системи у будь-який момент часу. CQRS, у свою чергу, розділяє запити на читання та зміну даних, оптимізуючи кожну з цих операцій під конкретні завдання. Кожен з цих патернів активно впроваджується у промислових рішеннях, зокрема компаніями типу Netflix, Uber, Spotify.

Попри існування численних рішень, їх реалізація вимагає глибокого розуміння архітектурних принципів, розподіленої обробки подій та забезпечення надійності. На практиці питання консистентності часто ігнорується або вирішується частково, що знижує стабільність систем і призводить до важковідтворюваних помилок. Саме тому дослідження ефективних способів забезпечення узгодженості даних у мікросервісній архітектурі залишається актуальним завданням для розробників та архітекторів програмного забезпечення.

Об'єктом дослідження в цій роботі є мікросервісні архітектури — підхід до побудови програмних систем, що базується на розподіленій природі виконання незалежних компонентів. Предметом дослідження є конкретні аспекти забезпечення узгодженості даних у таких архітектурах, зокрема використання патернів Saga, Event Sourcing та CQRS у практичних застосуваннях. Метою роботи є розробка, реалізація та аналіз прикладної мікросервісної системи, в якій демонструються різні підходи до забезпечення узгодженості даних. У ході роботи було реалізовано архітектуру з трьома незалежними сервісами: «Замовлення», «Склад» і «Master Data». Кожен з них має власну базу даних, працює автономно та взаємодіє з іншими сервісами через брокер подій Apache Kafka. Це дозволяє змоделювати типову ситуацію, де необхідно зберігати консистентність між сервісами без прямої синхронізації.

Унікальність розробленого підходу полягає у комплексному підході до тестування кожного патерну в умовах різних сценаріїв взаємодії

сервісвключно з обробкою збоїв, затримок, дублювання подій та верифікацією стану. Усі рішення побудовані з урахуванням сучасних технологій — Spring Boot, Apache Kafka, PostgreSQL, що дозволяє легко масштабувати та інтегрувати систему в існуючі програмні середовища.

Актуальність даного дослідження визначається зростанням попиту на високонадійні та масштабовані програмні системи, які можуть стабільно функціонувати навіть у випадках часткових відмов. Питання консистентності даних у таких системах безпосередньо впливає на якість і стабільність роботи критичних застосунків, а впровадження ефективних патернів дозволяє уникнути втрати даних і забезпечити цілісність інформації на всіх етапах життєвого циклу продукту. Новизна роботи полягає у практичному порівнянні трьох популярних патернів консистентності в умовах однієї системи, що дозволяє зробити висновки щодо ефективності кожного з них залежно від архітектурного контексту та вимог до продуктивності.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПРОБЛЕМАТИКИ

1.1 Консистентність даних у розподілених системах

У сучасних інформаційних системах, особливо в контексті мікросервісної архітектури, консистентність даних залишається однією з ключових проблем, що визначає якість роботи усієї розподіленої системи. На відміну від централізованих систем, де дані зберігаються й обробляються в одному середовищі, у розподілених системах компоненти працюють незалежно один від одного, часто на різних фізичних вузлах або навіть у різних дата-центрах. Це створює додаткові виклики, пов’язані з необхідністю підтримання узгодженого стану даних між цими компонентами.

У загальному випадку консистентність у розподілених системах означає те, що всі частини системи “бачать” однаковий стан даних, навіть якщо між ними відбуваються певні затримки у комунікації або виникають часткові збої. В залежності від вимог до системи, розрізняють декілька основних моделей консистентності. Найбільш строгим є підхід strong consistency (сильна узгодженість), який гарантує, що після будь-якої зміни даних усі наступні звернення до цих даних повернуть саме змінене значення. Такий підхід часто реалізовується через синхронну реплікацію і застосовується, наприклад, у системах, де критично важливо уникати будь-яких розбіжностей у даних, таких як банківські операції або бухгалтерські системи [2].

З іншого боку, існує eventual consistency (остаточна узгодженість), яка допускає тимчасову розбіжність станів даних у різних частинах системи. Головна ідея цього підходу полягає в тому, що при відсутності нових змін, усі частини системи з часом прийдуть до єдиного узгодженого стану. Така модель є типовою для високомасштабованих систем, де важливіше забезпечити доступність і продуктивність, ніж абсолютну точність у реальному часі. Вона застосовується, зокрема, в таких розподілених сховищах, як Amazon Dynamo, Apache Cassandra та Riak [3].

Окрім зазначених підходів, у сучасних дослідженнях також розглядається проміжна форма — causal consistency (причинно-обумовлена консистентність). [4] Цей тип узгодженості гарантує, що операції, які мають причинно-наслідковий зв’язок, будуть виконані у правильному порядку для всіх учасників системи. Наприклад, якщо один користувач створює коментар до повідомлення, то всі інші повинні спочатку побачити саме повідомлення, а вже потім — коментар до нього. Причинно-обумовлена консистентність знаходиться посередині між сильною та остаточною, забезпечуючи більшу гнучкість без повної відмови від логічної послідовності.

Особливо гостро питання консистентності постає у мікросервісних архітектурах, де кожен сервіс зазвичай має власну базу даних, а транзакції, що охоплюють декілька сервісів, ускладнені або взагалі неможливі. У таких умовах важко забезпечити атомарність операцій, тобто гарантувати, що всі зміни будуть або повністю виконані, або повністю скасовані. Це обмежує використання традиційної ACID-моделі (Atomicity, Consistency, Isolation, Durability), яка є основою для класичних реляційних баз даних. Натомість, у розподілених системах частіше застосовується альтернатива — модель BASE (Basically Available, Soft state, Eventual consistency). [5] Вона приймає певну "м’якість" стану системи і допускає, що дані можуть бути тимчасово неузгодженими, але доступність та масштабованість системи залишаються високими.

Ці компроміси між консистентністю, доступністю та стійкістю до розділення мережі чудово описує так звана CAP-теорема, сформульована Еріком Брюером. [6] Згідно з цією теоремою, розподілена система не може одночасно гарантувати усі три властивості: узгодженість (Consistency), доступність (Availability) та толерантність до поділу (Partition tolerance). У випадку виникнення розриву мережевого з’єднання між частинами системи (тобто "partition"), доводиться обирати між підтриманням доступності або узгодженості. У реальних умовах більшість мікросервісних застосунків обирають доступність і толерантність до поділу, жертвуючи строгою

узгодженістю даних, що змушує розробників впроваджувати спеціалізовані підходи до синхронізації стану між сервісами.

Таким чином, питання консистентності даних у розподілених системах є не лише технічним, а й архітектурним викликом, що вимагає прийняття свідомих компромісів. Залежно від вимог до продуктивності, стійкості й логічної цілісності, обираються відповідні моделі узгодженості. У контексті мікросервісної архітектури така гнучкість є необхідною умовою масштабованості, проте одночасно потребує впровадження додаткових механізмів забезпечення узгодженого стану, що детально розглядається у подальших підрозділах цієї роботи.

2.1 Особливості міроксервісної архітектури

Мікросервісна архітектура стала домінуючим підходом у створенні масштабованих, гнучких і легко підтримуваних програмних систем. Її основною ідеєю є поділ застосунку на окремі незалежні сервіси, кожен з яких виконує чітко визначену бізнес-функцію. Ці сервіси можуть розгортатися, масштабуватися й оновлюватися незалежно один від одного, що суттєво підвищує гнучкість розробки та експлуатації програмного забезпечення [7].

Кожен мікросервіс, як правило, має власний інтерфейс API, власну логіку обробки і найважливіше — власну базу даних, ізольовану від інших. Такий підхід дозволяє уникнути тісного зв’язування між компонентами системи, що властиве монолітним архітектурам. Проте саме ця ізоляція породжує цілу низку складнощів, пов’язаних із забезпеченням цілісності даних, особливо тоді, коли бізнес-операції охоплюють кілька сервісів одночасно. На відміну від монолітних систем, де транзакції в межах однієї реляційної бази даних можуть гарантувати властивості ACID, у мікросервісах реалізація глобальних транзакцій є практично недосяжною без суттєвих ускладнень [8].

Відсутність централізованої транзакційної моделі означає, що традиційний підхід до гарантування узгодженості — через двофазні коміти

(2PC) — або взагалі не застосовується, або застосовується у спрощеній формі, яка не витримує вимог до продуктивності та масштабованості. Крім того, 2PC не є стійким до збоїв мережі та може спричинити блокування ресурсів у разі втрати зв’язку з координатором транзакції. [9] У світлі цього мікросервісна архітектура змушує шукати нові способи забезпечення узгодженості, які б відповідали її децентралізованій природі.

Проблематика ще більше загострюється в умовах асинхронної взаємодії між сервісами. В сучасних мікросервісних системах поширеною є модель взаємодії на основі подій, де сервіси не викликають один одного безпосередньо, а натомість публікують повідомлення в загальну чергу або брокер повідомлень (наприклад, Apache Kafka або RabbitMQ). Це дозволяє досягти високої розчепленості між компонентами, але водночас робить процес узгодження стану системи менш контрольованим. Відсутність прямої синхронізації між сервісами означає, що зміни у стані одного сервісу можуть стати відомими іншим з певною затримкою, що потенційно призводить до тимчасових розбіжностей у даних [10].

Крім технічних аспектів, слід враховувати й архітектурні наслідки. Мікросервіси зазвичай реалізуються як окремі процеси або навіть контейнери, що керуються системами оркестрації на кшталт Kubernetes. Вони можуть бути розгорнуті у різних зонах доступності або навіть у різних географічних регіонах. Це збільшує латентність при взаємодії та ускладнює гарантування синхронності обміну повідомленнями. Водночас, зростає ризик часткових збоїв або недоступності окремих сервісів. Таким чином, архітектура сама по собі передбачає необхідність врахування “нестабільного” середовища виконання як норми [11].

Все це призводить до необхідності впровадження нових патернів забезпечення узгодженості, які адаптовані до умов розподілених систем. Йдеться, зокрема, про такі патерни як Saga (розподілені транзакції через послідовність локальних операцій), Event Sourcing (зберігання історії змін у вигляді подій) та CQRS (розділення моделей читання і запису). Ці підходи не

тільки дозволяють зберігати контроль над консистентністю, але й підвищують масштабованість і стійкість до відмов. Проте їх впровадження потребує глибшого розуміння розподілених систем, правильної організації міжсервісної взаємодії та суворого дотримання принципів ідіоматичної реалізації [12].

Загалом, мікросервісна архітектура, незважаючи на свою популярність і переваги, створює серйозні виклики для забезпечення узгодженості даних. Це вимагає від інженерів і архітекторів не лише глибоких технічних знань, але й готовності до використання новітніх підходів, орієнтованих на асинхронність, відмовостійкість і гнучкість у керуванні життєвим циклом даних у системі.

1.3 Патерни забезпечення консистентності

У контексті мікросервісної архітектури, де кожен сервіс є ізольованою одиницею з власною базою даних, проблема узгодженості даних стає особливо актуальною. Через відсутність централізованих транзакцій необхідно використовувати альтернативні підходи до досягнення консистентного стану системи в цілому. Саме тому в індустрії набули поширення спеціалізовані патерни, які дозволяють управляти узгодженістю даних у розподілених умовах. Серед них найпоширенішими є патерни Saga, Event Sourcing і CQRS, кожен з яких має свої сильні сторони, обмеження та сфери застосування.

1.3.1 Saga

Патерн Saga є одним із найвідоміших підходів до реалізації розподілених транзакцій. [13] Його суть полягає в тому, що велика бізнес-операція, яка охоплює кілька сервісів, розбивається на послідовність локальних транзакцій. Кожна з них виконується у межах окремого мікросервісу, а після її завершення ініціюється наступна транзакція через передачу події або виклик API. У разі виникнення помилки виконується компенсуюча дія, яка «відкочує» ефекти попередніх транзакцій, забезпечуючи узгодженість.

Наприклад, в системі електронної комерції оформлення замовлення може включати кілька етапів: резервування товару, зняття коштів з рахунку покупця, створення доставки. Якщо після зняття коштів не вдається створити доставку, то Saga ініціює повернення грошей і скасування бронювання. Перевагою цього підходу є його відмова від централізованого менеджера транзакцій, що робить його природним для мікросервісного середовища. Існують два основні варіанти реалізації Saga — оркестрація (централізований координатор керує послідовністю дій) і хореографія (кожен сервіс сам вирішує, як реагувати на події). Обидва підходи мають застосування залежно від складності бізнес-логіки та необхідного рівня контролю [14].

1.3.2 Event Sourcing

Патерн Event Sourcing змінює традиційний підхід до зберігання стану об'єктів. Замість збереження поточного стану в базі даних, система зберігає послідовність подій, які привели до цього стану. Таким чином, стан у будь-який момент часу може бути відтворено шляхом програвання подій з початку. Кожна подія описує окрему зміну — наприклад, «кошти знято», «товар додано до замовлення», «доставка підтверджена» [15].

Цей підхід дозволяє зберігати повну історію змін, що важливо для аудиту, аналітики та відлагодження. Крім того, Event Sourcing ідеально поєднується з асинхронною передачею даних між сервісами, оскільки кожна подія є автономною одиницею інформації, яку можна ретранслювати, обробляти й зберігати в інших частинах системи. Серед ключових переваг патерну — можливість відновлення системи після збою, створення проекцій для швидкого читання та гнучкість у масштабуванні. Проте реалізація Event Sourcing потребує додаткових зусиль для гарантування ідемпотентності, керування версіями подій і забезпечення стабільного порядку їх обробки [16].

1.3.3 CQRS

Патерн CQRS (Command Query Responsibility Segregation) пропонує розділення системи на дві моделі: одну — для виконання команд (модифікація стану), іншу — для виконання запитів (читання даних). Це дозволяє оптимізувати обидва аспекти незалежно: модель команд може бути складною та строго перевіряти правила бізнес-логіки, а модель запитів — максимально простою і швидкою для ефективного доступу до даних [17].

У контексті мікросервісів CQRS часто використовується у зв'язці з Event Sourcing. Події, що виникають у результаті обробки команд, передаються до проєкторів (read models), які створюють оптимізовані структури для запитів. Наприклад, у системі управління банківськими рахунками після зняття коштів генерується подія «зняття виконано», яка оновлює агреговану інформацію про баланс у read model. Таким чином, модель читання може бути реплікована, масштабована і побудована спеціально під потреби інтерфейсу користувача. Проте, використання CQRS накладає додаткову складність на підтримку консистентності між моделями та ускладнює тестування через необхідність координації подій між компонентами [18].

1.4. Огляд готових технологічних рішень

Розглянемо сучасні технологічні платформи та фреймворки, що підтримують реалізацію таких патернів, як Saga, Event Sourcing і CQRS. Ці інструменти значно спрощують впровадження логіки консистентності у мікросервісних застосунках, автоматизуючи обробку транзакцій, подій та взаємодію між сервісами.

1.4.1 Axon Framework, Temporal, Netflix Conductor, EventStoreDB

Одним із найвідоміших фреймворків, що поєднує реалізацію CQRS та Event Sourcing, є Axon Framework. Він надає розробникам готові засоби для обробки команд, подій і запитів, включаючи підтримку агрегатів, саг та розподілених транзакцій. Axon працює з власним брокером подій (Axon Server), але також може інтегруватися з іншими засобами передачі

повідомлень. Завдяки глибокій інтеграції з Java-екосистемою, Axon є зручним вибором для команд, які будують мікросервіси на базі Spring Boot [19].

Ще потужнішим і масштабованішим рішенням є Temporal — платформа для побудови довготривалих, надійних бізнес-процесів у вигляді "workflow"-ів. Temporal дозволяє розробникам описувати логіку workflow як звичайний код на Java, Go чи TypeScript, при цьому сама система автоматично гарантує ідемпотентність, відновлення після збоїв, повторення кроків, таймаути, дедлайни тощо. Temporal дозволяє реалізовувати як патерн Saga, так і складніші варіанти керування консистентністю, не турбуючись про збої інфраструктури або подвійне виконання. Його використовують такі компанії, як Snap, Box і Netflix [20].

До схожих рішень можна віднести Netflix Conductor — платформу оркестрації мікросервісів, яка дозволяє створювати workflows як послідовність мікросервісних задач. Хоча Conductor не є повноцінною реалізацією патерну Saga, він дозволяє централізовано керувати послідовністю виконання операцій і забезпечує контроль над їхнім станом. Розробники можуть створювати складні схеми бізнес-логіки з можливістю ретраїв, компенсуючих дій і умовних гілок. Основною перевагою є гнучкість і хороша масштабованість, а також інтеграція з іншими компонентами Netflix OSS [21].

Що стосується зберігання подій, то тут особливу роль відіграє EventStoreDB — спеціалізована база даних, побудована для зберігання потоків подій. Вона надає розробникам простий механізм для зберігання подій у вигляді потоків, що особливо важливо для реалізації Event Sourcing. EventStoreDB підтримує іменовані стріми, оптимістичне блокування, а також механізми підписки на події, що дозволяє іншим сервісам реагувати на зміни у стані системи. Завдяки тому, що дані не змінюються, а лише додаються нові події, EventStoreDB гарантує відтворюваність і надійність, що є критично важливим у критичних бізнес-системах [22].

1.4.2 Apache Kafka як подієвий транспорт

Особливе місце серед інструментів забезпечення консистентності посідає Apache Kafka — розподілений лог подій і брокер повідомлень, який виступає не лише транспортом, а й ключовим архітектурним компонентом для побудови event-driven систем. Kafka дозволяє зберігати події в топіках, до яких можуть підписуватись численні споживачі, незалежно обробляючи події в реальному часі або з певною затримкою. Завдяки високій пропускній здатності, горизонтальному масштабуванню та збереженню подій протягом довгого часу Kafka використовується як «системний хребет» у багатьох великих розподілених системах [23].

Kafka підтримує гарантії доставки (at-least-once, exactly-once), що дозволяє будувати на її основі надійні механізми обміну даними між сервісами. Вона також добре поєднується з іншими компонентами: наприклад, Event Sourcing може використовувати Kafka як сховище або проміжний буфер подій, а CQRS може базуватися на потоках Kafka для оновлення проекцій в режимі реального часу. Крім того, Kafka Streams і Kafka Connect дозволяють будувати цілі обчислювальні пайплайни з трансформацією, агрегацією та маршрутизацією подій [24].

У контексті Saga Kafka може використовуватись як транспорт між сервісами у патерні хореографії. Сервіси, що беруть участь у сазі, публікують і слухають події, не маючи централізованого координатора, що зменшує зчеплення, але водночас ускладнює спостереження та відлагодження. У такому випадку необхідно ретельно контролювати послідовність подій і стан саги, щоб уникнути розсинхронізації.

1.5 Порівняльний аналіз підходів

Попри те, що сучасні технології та архітектурні патерни забезпечують широкий вибір підходів до узгодженості даних у мікросервісах, жодне з рішень не є універсальним. Кожен підхід має свої переваги й обмеження, і вибір конкретного рішення залежить від вимог до продуктивності, надійності, складності підтримки та масштабу системи. У цьому підрозділі порівнюються

ключові патерни та технології, описані раніше, з метою виявлення загальних тенденцій у практичному застосуванні.

Патерн Saga (як оркестрація, так і хореографія) забезпечує достатній рівень логічної узгодженості в умовах відсутності глобальних транзакцій, особливо коли важливішим є збереження бізнес-процесу в цілому, ніж миттєва атомарність усіх дій. Saga легко масштабуються, але потребують ретельної реалізації компенсуючих дій та добре продуманої обробки помилок. Платформи на кшталт Temporal значно спрощують реалізацію цього патерну, надаючи готову інфраструктуру для керування довготривалими процесами.

Event Sourcing, з іншого боку, дозволяє фіксувати кожну зміну стану системи у вигляді події, що забезпечує високу трасованість і можливість «повернення в часі». Він підходить для сценаріїв, де важлива повна історія змін (наприклад, у фінтехі або логістиці), однак ускладнює розробку, особливо при потребі у запитах, які об’єднують багато сутностей. Застосування EventStoreDB або Kafka як подієвого сховища допомагає реалізувати цю модель на практиці

CQRS дозволяє ефективно масштабувати читання й запис за рахунок їх розділення. У поєднанні з Event Sourcing це дає потужну архітектуру для високонавантажених систем, але також додає складності через потребу синхронізації між командами і запитами. Axon Framework забезпечує вбудовану підтримку цієї парадигми, що робить її більш доступною для команд із Java-стеком.

Щодо інструментів, Temporal та Axon Framework надають повний набір функціональності для реалізації узгодженості, включаючи управління сагами, збереження подій, обробку команд і запитів. Temporal краще підходить для складних бізнес-процесів з тривалим життєвим циклом, тоді як Axon краще інтегрується з Java-проєктами та CQRS. Netflix Conductor орієнтований на централізовану оркестрацію задач, але не містить готових рішень для Event Sourcing або CQRS.

У свою чергу, Apache Kafka виділяється як подієвий транспорт, який добре масштабується та надає механізми повторного програвання подій. Його використання доцільне в системах із високими вимогами до пропускної здатності та обробки подій у реальному часі. Проте Kafka не реалізує логіку саг чи CQRS безпосередньо, і для цього потребує інтеграції з додатковими компонентам.

Узагальнено, гнучкість найкраща у Kafka й Temporal, надійність — у Temporal та Event Sourcing (особливо з EventStoreDB), простота використання — у Axon (для Java-проєктів), а продуктивність — у Kafka-підходів, які найменше обмежують систему централізованими викликами. У практиці найбільш популярним варіантом залишається комбінація кількох підходів: наприклад, CQRS з Event Sourcing для критичних бізнес-даних і Saga (через Temporal або Kafka) — для узгодженості в сервісах.

1.6 Висновки до розділу 1

У першому розділі було проведено ґрунтовний аналіз проблематики забезпечення консистентності даних у мікросервісних застосунках. Розглянуто основні моделі узгодженості — strong, eventual та causal consistency — їхні характеристики, сильні й слабкі сторони, а також виклики, які виникають у мікросервісній архітектурі в контексті розподіленого зберігання даних та відсутності глобальних транзакцій.

Окремо проаналізовано сучасні технологічні рішення, які реалізують зазначені патерни на практиці — Axon Framework, Temporal, Netflix Conductor, EventStoreDB, Apache Kafka та інші. Наведено порівняння їхніх функціональних можливостей, складності впровадження, рівня масштабованості та зручності використання.

На підставі проведеного огляду можна зробити висновок про актуальність обраної тематики. З розвитком мікросервісних архітектур питання забезпечення узгодженості даних набуває особливої ваги, адже

класичні транзакційні підходи в таких умовах втрачають свою ефективність. Це створює потребу в гнучких і спеціалізованих рішеннях, які дозволяють зберігати баланс між консистентністю, доступністю та продуктивністю. Таким чином, подальше дослідження практичного застосування зазначених патернів і технологій є доцільним і обґрунтованим у межах цієї кваліфікаційної роботи.

2 ПРОЕКТУВАННЯ ТА РОЗРОБКА

2.1 Консистентність даних у розподілених системах

Метою цієї роботи є дослідження підходів до забезпечення консистентності даних у мікросервісній архітектурі шляхом створення демонстраційної системи, яка реалізує декілька моделей узгодженості: strong, eventual та causal. Для досягнення поставленої мети було спроєктовано та реалізовано прототип мікросервісного застосунку з трьома незалежними сервісами — Order, Inventory та Master-Data, — кожен із яких працює з власною базою даних та демонструє одну з моделей консистентності.

Об'єктом дослідження є програмна система, що реалізує розподілену обробку транзакцій у середовищі мікросервісної архітектури. Предметом дослідження виступають архітектурні патерни (Saga, Event Sourcing, CQRS) та інструменти забезпечення консистентності у розподілених системах.

Основні задачі, що вирішує система:

- Забезпечити узгодженість даних між сервісами в умовах відсутності централізованої транзакційної моделі;
- Реалізувати три підходи до консистентності: strong consistency (Order Service), eventual consistency (Inventory Service), causal consistency (Master-Data Service);
- Впровадити архітектурні патерни — Saga для керування транзакціями замовлень, Event Sourcing для історії змін стану замовлення, CQRS у поєднанні з causal consistency для сервісу довідкових даних;
- Організувати взаємодію між сервісами через подієву модель на базі брокера повідомлень;

Задля забезпечення гнучкості розгортання, зручності розробки, модульності та відтворюваності середовища було використано сучасний стек технологій, до якого входять: Spring Boot — як основний фреймворк для

розробки мікросервісів; Apache Kafka — як засіб реалізації подієвої взаємодії між сервісами; PostgreSQL — як сховища даних; Docker — для контейнеризації усіх компонентів системи.

Очікувані результати системи:

- Демонстрація практичного застосування патернів забезпечення консистентності в умовах взаємодії автономних сервісів;
- Наглядне порівняння strong, eventual та causal consistency в єдиній системі;
- Виявлення переваг і недоліків кожного підходу з урахуванням складності реалізації, гнучкості, продуктивності та надійності;
- Створення технічної бази для подальшого масштабування або інтеграції із зовнішніми системами.

На підставі проведеного огляду можна зробити висновок про актуальність обраної тематики. З розвитком мікросервісних архітектур питання забезпечення узгодженості даних набуває особливої ваги, адже класичні транзакційні підходи в таких умовах втрачають свою ефективність. Це створює потребу в гнучких і спеціалізованих рішеннях, які дозволяють зберігати баланс між консистентністю, доступністю та продуктивністю. Таким чином, подальше дослідження практичного застосування зазначених патернів і технологій є доцільним і обґрунтованим у межах цієї кваліфікаційної роботи.

2.2 Архітектура розробленої системи

З огляду на загальну архітектуру, далі буде детально розглянуто кожен із компонентів розробленої системи. Кожен сервіс виконує чітко окреслену роль у бізнес-логіці застосунку, реалізує специфічний шаблон забезпечення консистентності та має власні технічні особливості. Такий поділ дозволяє глибше проаналізувати, яким чином обрана модель узгодженості впливає на проєктування, реалізацію і поведінку мікросервісу в умовах розподіленої системи.

2.2.1 Загальна структура системи

Компоненти системи працюють у середовищі Docker-контейнерів, що спрощує процес розгортання та тестування. Взаємодія з Kafka організована через відповідні Kafka-клієнти у кожному сервісі, що слухають або надсилають події у визначені топіки. Уся архітектура побудована таким чином, щоб дозволити незалежне масштабування будь-якого компонента, у разі зростання навантаження або появи нових бізнес-вимог.

2.2.2 Основні компоненти

Order Service відповідає за створення замовлень і є центральним елементом системи, де реалізовані основні патерни забезпечення консистентності. Цей сервіс працює за моделлю strong consistency, виконуючи критичні транзакції в межах локальної бази даних, що гарантує їх атомарність і надійність. Для координації розподілених транзакцій з іншими сервісами застосовується патерн Saga, який організовує виконання складних бізнес-процесів через послідовність локальних дій із можливістю запуску компенсуючих операцій у разі помилок. Крім того, у Order Service реалізовані підходи Event Sourcing та CQRS, що дозволяє зберігати історію змін через послідовність подій, а також розділяти операції читання і запису для підвищення продуктивності та масштабованості.

Inventory Service відповідає за облік товарних запасів і реалізує модель eventual consistency. Зміни в запасах відбуваються на основі подій, що надходять із Kafka. Стан системи формується через накопичення цих подій, що підвищує гнучкість і забезпечує надійність, хоча й з певною затримкою у відображенні актуальних даних.

Master-Data Service управляє довідковими даними, які активно використовуються іншими сервісами, але змінюються рідко. Для цього сервісу застосовано модель causal consistency, що забезпечує логічний порядок операцій між пов’язаними подіями без необхідності жорсткої глобальної синхронізації.

2.2.3 Обгрунтування використання різних рівнів консистентності

Рішення про застосування трьох різних моделей консистентності було прийнято з урахуванням специфіки функціональності кожного сервісу, реальних сценаріїв використання та архітектурних обмежень.

Strong consistency була обрана для Order Service, оскільки будь-яке порушення узгодженості замовлень може призвести до суттєвих логічних або фінансових помилок. Користувач повинен отримати однозначний результат — або замовлення створено, або ні — без проміжних станів. Саме тому в основі цього сервісу лежать ACID-принципи і синхронне збереження транзакцій.

У Inventory Service була реалізована eventual consistency, оскільки в контексті обліку запасів допустиме тимчасове відставання між фактичним і відображеним станом. Події обробляються асинхронно, що дає змогу покращити продуктивність та уникнути блокувань. Event Sourcing в поєднанні з Kafka забезпечує можливість повторної обробки подій у разі помилок, що особливо цінно у розподілених системах.

Для Master-Data Service підходить causal consistency, оскільки дані цього модуля — наприклад, список товарів або їх категорій — змінюються рідко, але мають логічну послідовність. Використання causal consistency дозволяє зберігати зв’язки між подіями (наприклад, спочатку створення товару, потім — його оновлення), не навантажуючи систему жорсткими блокуваннями або глобальним порядкуванням. Це дозволяє сервісу бути швидким, масштабованим та легким в обслуговуванні.

Комбінація трьох рівнів консистентності в одній системі дала змогу провести повноцінне дослідження їх взаємодії, сильних і слабких сторін, а також практичну оцінку ефективності патернів Saga, CQRS та Event Sourcing у реальних умовах.

2.3 Застосування патернів консистентності

У процесі розробки системи було важливо забезпечити узгодженість даних між сервісами без використання глобальних транзакцій. Для цього були

впроваджені перевірені архітектурні патерни — Saga, Event Sourcing та CQRS, кожен з яких відповідає за певний аспект консистентності в межах мікросервісної взаємодії. У наступних підпунктах детально описано реалізацію кожного з них у межах проєкту.

2.3.1 Реалізація патерну Saga

У розробленому мікросервісі Order Service патерн Saga реалізовано для координації процесу оформлення замовлення з урахуванням перевірки наявності товару в Inventory Service. Спершу Order Service приймає заявку на створення замовлення, виконує валідацію продукту через Master Data Service та надсилає запит на резервування товарів до Inventory Service. Якщо резервування пройшло успішно, Order Service створює замовлення з підтвердженим статусом, фіксує подію у базі даних і надсилає повідомлення Kafka. У разі помилки на будь-якому з попередніх етапів (наприклад, продукт не знайдено або недостатньо залишку), замовлення не створюється, і система переходить у безпечний стан без необхідності ручного втручання.

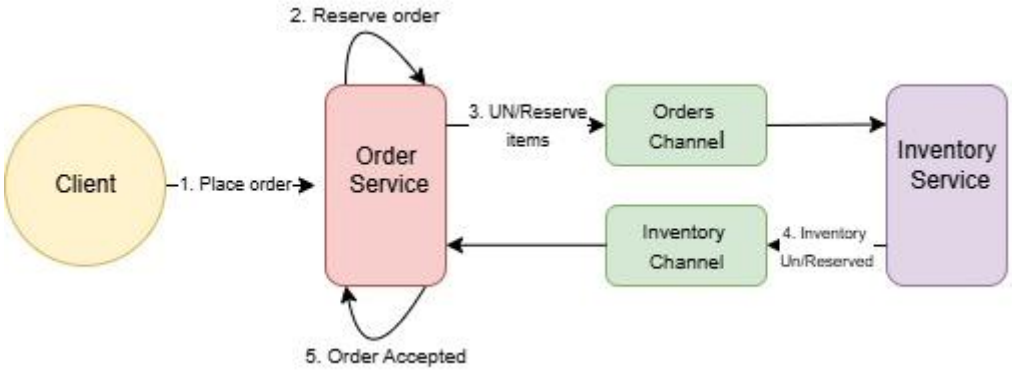

Рис. 2.1 – Діаграма реалізації патерна Saga

Лістинг 2.1 – Приклад коду обробки Saga в Order Service

```
    @KafkaListener(topics = "inventory-updated-topic", groupId = "order-inventory-group")
    public void handleInventoryUpdate(InventoryUpdatedEvent event) {
```

```
        log.info("Received InventoryUpdatedEvent: {}", event);
        Optional<Order>              optionalOrder              =
orderRepository.findById(event.orderId());
        if (optionalOrder.isPresent()) {
            Order order = optionalOrder.get();
            order.setStatus(event.status());   //   може   бути
CONFIRMED або CANCELLED
            orderRepository.save(order);
            log.info("Updated  Order  ID  {}  to  status:  {}",
order.getId(), event.status());
        } else {
            log.warn("Order not found for InventoryUpdatedEvent:
{}", event.orderId());
        }
    }
```

Основною перевагою такої реалізації є можливість виконувати довготривалі транзакції без блокувань, що є суттєвим для систем з високою навантаженістю та великою кількістю паралельних запитів. Компенсуючі дії в окремому вигляді явно не винесені, проте логіка сервісу передбачає, що у разі невдачі на будь-якому з етапів до створення замовлення, ніякі зміни у базу даних не записуються. Також реалізовано метод для скасування замовлення, що може використовуватись як компенсуюча операція вручну або програмно. Таким чином, часткова реалізація патерну дозволяє зберігати узгодженість станів навіть у разі помилок.

Обробка компенсуючих дій є критичною складовою патерну Saga, оскільки дозволяє відкотити частково виконані транзакції. У поточній реалізації Order Service забезпечує відмовостійкість за рахунок черговості дій і транзакційності локальних операцій: якщо перевірка продукту або резервування не проходять, замовлення навіть не створюється. Крім того, реалізовано метод для скасування замовлення, який може бути викликаний у

разі необхідності — як вручну, так і автоматизовано. Це дозволяє запобігти станам, у яких дані різних сервісів несумісні між собою.

2.3.2 Реалізація Event Sourcing

У розробленому мікросервісі Order реалізовано патерн Event Sourcing, який полягає у збереженні всіх змін стану об’єкта у вигляді послідовності подій (events), а не лише його поточного стану. Такий підхід дозволяє не тільки фіксувати історію змін, але й легко відтворювати повну картину життєвого циклу об'єкта, забезпечуючи прозорість, трасування та можливість відновлення стану у будь-який момент часу.

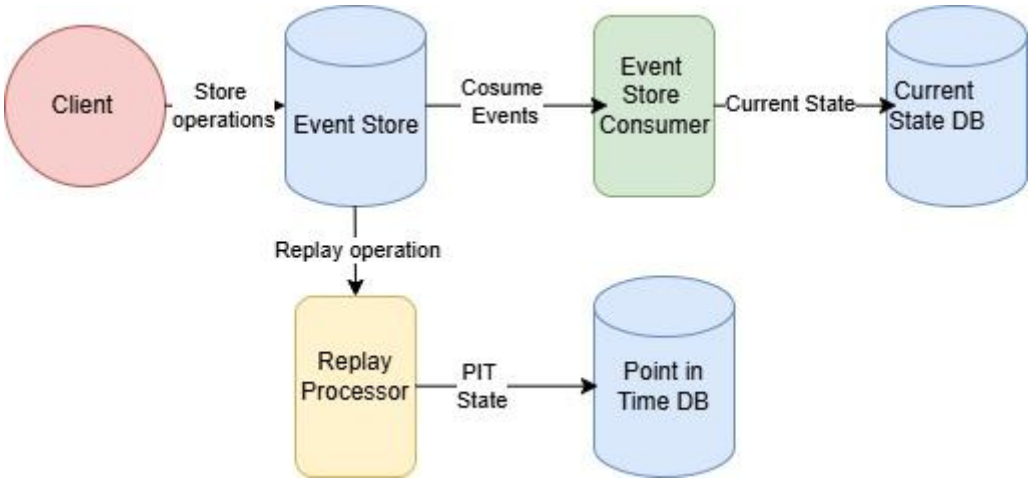

Рис. 2.3 - Діаграма реалізації патерна Event Sourcing

Для реалізації цього підходу в базі даних створено таблицю order_events, у якій фіксуються всі події, що стосуються змін у замовленнях. Кожен запис у таблиці містить унікальний ідентифікатор, посилання на замовлення (order_id), тип події (event_type), дані події у форматі JSON (event_data) та мітку часу (created_at), коли подія була згенерована.

Лістинг 2.2 – SQL для таблиці order_events

```
CREATE TABLE order_events (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
```

```
    order_id BIGINT NOT NULL,
    event_type VARCHAR(50) NOT NULL,
    event_data TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

У Java-додатку для роботи з цією таблицею створено сутність OrderEvent, яка описує модель події, та інтерфейс репозиторію OrderEventRepository, що дозволяє зберігати та отримувати події за ідентифікатором замовлення. Самі події описуються окремими класами, які інкапсулюють зміни, що відбулися (наприклад, створення, підтвердження або скасування замовлення).

Під час створення або зміни замовлення в системі одночасно із основними даними зберігається відповідна подія. Це дозволяє відокремити сам факт зміни стану від логіки, яка викликає цю зміну.

Лістинг 2.3 – Приклад збереження події

```
OrderEvent orderEvent = OrderEvent.builder()
    .orderId(order.getId())
    .eventType("ORDER_CREATED")
    .eventData(objectMapper.writeValueAsString(payload))
    .createdAt(LocalDateTime.now())
    .build();

orderEventRepository.save(orderEvent);
```

Завдяки такій архітектурі можливо реалізувати реконструкцію стану замовлення шляхом послідовного програвання збережених подій. Це означає, що навіть якщо сталася втрата або пошкодження основних даних замовлення, його стан можна відновити, застосувавши всі події у хронологічному порядку. Такий підхід також відкриває можливості для реалізації time travel debugging, аналітики змін та побудови аудиту.

Лістинг 2.3 – Псевдокод реконструкції

```
public Order reconstructOrder(Long orderId) {
    List<OrderEvent> events =
orderEventRepository.findByOrderIdOrderByCreatedAt(orderId);
    Order order = new Order();
    for (OrderEvent event : events) {
        switch (event.getEventType()) {
            case "ORDER_CREATED":
                // оновлення стану замовлення
                break;
            case "ORDER_CONFIRMED":
                order.setStatus("CONFIRMED");
                break;
            case "ORDER_CANCELLED":
                order.setStatus("CANCELLED");
                break;
        }
    }
```

Загалом, використання Event Sourcing у сервісі Order забезпечує гнучкість, надійність та масштабованість, а також полегшує інтеграцію з іншими мікросервісами за допомогою публікації подій у брокер повідомлень (наприклад, Kafka чи RabbitMQ), що є наступним етапом еволюції архітектури.

2.3.3 Реалізація CQRS

У процесі проєктування та реалізації мікросервісу Order Service було впроваджено архітектурний патерн CQRS (Command Query Responsibility Segregation), що передбачає чітке розділення відповідальностей між операціями, які змінюють стан системи (команди), та тими, які лише читають

дані (запити). Основна мета впровадження CQRS — підвищення масштабованості, продуктивності та гнучкості сервісу.

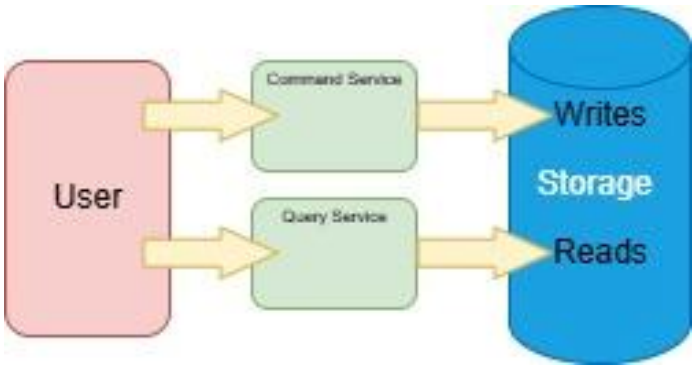

Рис. 2.2 – Діаграма реалізації патерну CQRS

У класичній CRUD-моделі одна й та сама модель та сервіс обробляють як команди, так і запити, що ускладнює розвиток системи при зростанні навантаження або бізнес-вимог. CQRS вирішує цю проблему шляхом розділення логіки на дві незалежні гілки:

- Command side — відповідає за створення, оновлення та видалення даних (write operations).
- Query side — обробляє лише операції читання, не змінюючи стан системи (read operations).

У межах Order Service відповідне розділення було реалізовано на рівні сервісних класів:

- OrderService обробляє команди: створення, підтвердження та скасування замовлення, оновлення статусу тощо. Він працює з транзакціями та, за потреби, генерує події (у рамках Event Sourcing).
- OrderQueryService відповідає за запити: отримання списку всіх замовлень, детальної інформації про конкретне замовлення, фільтрацію за статусами тощо. Цей сервіс не використовує транзакцій, що дозволяє легко впроваджувати додаткову

оптимізацію, наприклад, кешування або створення окремих read-моделей.

У REST-контролерах реалізовано чіткий поділ логіки:

- Метод getAllOrders() та подібні були переміщені з OrderService до OrderQueryService, що відповідає принципам CQRS і дозволяє їх масштабувати окремо від командної частини.
- Операції на кшталт createOrder(), confirmOrder() або cancelOrder() лишилися в OrderService, як частина write-side.

Таке розділення дозволяє:

- Зменшити зв’язаність коду, оскільки зміни в логіці запитів не впливають на логіку команд, і навпаки.
- Оптимізувати читання, наприклад, через агрегацію у спеціалізовані read-моделі або використання проєкцій.
- Спростити масштабування: query side може бути масштабовано горизонтально незалежно від command side.
- Покращити продуктивність, оскільки запити не блокуються через транзакції або операції запису.
- Впровадити аналітику та пошук без порушення основної бізнес-логіки.

У результаті впровадження патерна CQRS архітектура сервісу замовлень стала більш чітко структурованою та модульною. Командна і запитна частини були розділені за відповідальністю, що підвищило керованість і зрозумілість коду. Завдяки цьому зменшилась ймовірність конфліктів між операціями читання та запису, що зробило систему більш стійкою до високих навантажень. Крім того, така архітектура забезпечує високу гнучкість: її легко адаптувати до змін у вимогах, зокрема — змінювати структуру запитів, додавати кешування або аналітику без ризику порушити логіку обробки команд.

2.4 Розгортання та масштабування

Для забезпечення високої гнучкості, стабільності та портативності розроблена система була реалізована з використанням сучасної технології контейнеризації — Docker. Всі ключові компоненти системи, такі як мікросервіси, бази даних, а також брокер повідомлень, були винесені в окремі контейнери. Це дозволяє запускати систему у повністю ізольованому середовищі, що можна легко відтворити як на локальних машинах розробників, так і в хмарних платформах, а також у рамках автоматизованих CI/CD пайплайнів. Завдяки такому підходу суттєво підвищується стабільність роботи, оскільки середовище виконання залишається однаковим незалежно від платформи та часу запуску.

Основна ідея контейнеризації полягає у створенні окремого, чітко визначеного середовища для кожного з основних мікросервісів: Order Service, Inventory Service та Master Data Service. Кожен сервіс виконується у власному контейнері, що повністю ізолює його процеси, залежності та конфігурації. Для збереження даних кожен мікросервіс підключений до власної бази даних PostgreSQL, яка також розгорнута у вигляді окремого контейнера. Така організація забезпечує незалежність збереження даних, унеможливлює конфлікти через спільний доступ і полегшує управління життєвим циклом кожної бази. Крім того, вона дозволяє гнучко керувати масштабуванням та оновленням окремих частин системи без впливу на інші.

Обмін подіями між мікросервісами реалізовано за допомогою брокера повідомлень Apache Kafka, що підтримує асинхронну комунікацію і забезпечує слабку зв’язність між компонентами. Це значно підвищує стійкість системи: навіть якщо один із сервісів тимчасово недоступний, події накопичуються у чергах і обробляються, як тільки сервіс відновлює роботу. Для коректної роботи Kafka задіяно окремий контейнер ZooKeeper, який відповідає за координацію і підтримку стану кластера, забезпечуючи надійність і балансування навантаження між брокерами.

Автоматична конфігурація сервісів є ще одним важливим елементом інфраструктури. Параметри підключення до баз даних, брокера подій та інших зовнішніх сервісів задаються через змінні середовища (environment variables), які передаються контейнерам при запуску. Такий підхід забезпечує гнучкість і зручність управління середовищем: зміна налаштувань не потребує перекомпіляції або перезбирання образів, а лише коригування конфігураційних параметрів. Запуск усіх контейнерів здійснюється одночасно за допомогою docker-compose, що суттєво спрощує розгортання та тестування системи для розробників і DevOps-інженерів.

Окрема увага приділялася можливості масштабування системи. Завдяки чіткому розділенню відповідальностей між мікросервісами, реалізованому згідно з принципами мікросервісної архітектури та патерном CQRS (Command Query Responsibility Segregation), кожен сервіс можна масштабувати незалежно від інших. Наприклад, Order Service, який відповідає за обробку команд створення і зміни замовлень, може масштабуватися горизонтально шляхом запуску додаткових екземплярів сервісу. При цьому Kafka автоматично розподіляє події між активними інстансами, що забезпечує паралельну і ефективну обробку повідомлень, знижуючи затримки та підвищуючи пропускну здатність.

Аналогічним чином, Inventory Service і Master Data Service функціонують автономно і можуть масштабуватися відповідно до поточного навантаження на обробку запитів або подій. Такий підхід дозволяє максимально ефективно використовувати ресурси, оскільки підвищене навантаження на один сервіс не впливає на продуктивність інших. Окрім того, розміщення окремих баз даних для кожного мікросервісу виключає конкуренцію за ресурси на рівні системи управління базами даних, що позитивно позначається на загальній продуктивності системи та її стабільності.

Таким чином, застосування контейнеризації в комбінації з мікросервісною архітектурою і брокером повідомлень Kafka забезпечує

високу гнучкість, надійність і масштабованість розробленої системи. Вона може без проблем адаптуватися до зростаючих вимог бізнесу, працювати у різних середовищах без необхідності суттєвих змін, а також швидко реагувати на збої або пікові навантаження. Цей підхід не тільки покращує розробку і тестування, але й значно спрощує підтримку й подальший розвиток системи.

2.5 Реалізація взаємодії між сервісами

У процесі реалізації мікросервісної архітектури ключовим завданням стало забезпечення надійної, гнучкої та масштабованої взаємодії між окремими сервісами. Оскільки кожен сервіс виконує чітко визначену функцію в межах свого bounded context, їхня взаємодія повинна бути організована таким чином, щоб мінімізувати зв’язність і водночас забезпечувати цілісність бізнес-процесів. У нашому випадку було прийнято рішення реалізувати міжсервісну взаємодію через обмін подіями — тобто за допомогою подієво-орієнтованої архітектури.

Замість прямого виклику одного сервісу з іншого, як це типово буває у монолітних системах або tightly coupled мікросервісах, у системі використовується Kafka як брокер повідомлень. Наприклад, коли користувач створює нове замовлення, OrderService не викликає напряму InventoryService, а просто публікує подію про створення замовлення. Інші сервіси, зокрема сервіс інвентарю, реагують на цю подію у власному контексті, самостійно вирішуючи, як і коли її обробити.

Такий підхід не лише роз’єднує сервіси технічно, але й дозволяє кожному з них еволюціонувати незалежно, зберігаючи стабільність усієї системи. Якщо якийсь із сервісів недоступний або тимчасово перевантажений, це не зупиняє роботу інших — події зберігаються у Kafka й будуть оброблені, щойно сервіс відновить свою роботу.

Окрім того, в системі застосовувались різні моделі консистентності, залежно від критичності операцій. Наприклад, для фінансово важливих процесів реалізовано механізми повторної доставки повідомлень, idempotent

обробку, локальні транзакції у межах сервісу та чітку перевірку послідовності подій. У менш критичних місцях застосовувалась відкладена обробка з подальшим підтвердженням результату.

Такий підхід дав змогу поєднати гнучкість, продуктивність і контроль. Сервіси можуть незалежно масштабуватись, оновлюватись і розвиватись, не порушуючи цілісності системи. Водночас, завдяки структурованому обміну подіями та добре визначеним контрактам між сервісами, досягається прозорість і передбачуваність взаємодії у розподіленому середовищі.

2.6 Проблеми, що виникли під час розробки

Під час розробки розподіленої мікросервісної системи виникла низка суттєвих проблем, пов’язаних із особливостями архітектури та технологій. Кожна з цих проблем вимагала окремого підходу для ефективного вирішення.

Проблема 1: Втрата сильної консистентності при впровадженні патерну Saga в Order Service

Під час впровадження патерну Saga у Order Service виникла складність із підтримкою сильної узгодженості даних. Через те, що Saga зазвичай реалізується як асинхронний процес із компенсуючими транзакціями, відмовлено від традиційної сильної консистентності. Це могло призводити до розбіжностей у станах між сервісами на різних етапах транзакції.

Шляхи вирішення:

Для збереження сильної узгодженості в Order Service патерн Saga було реалізовано неповноцінно, обмеживши деякі операції синхронним виконанням, що дозволило гарантувати послідовність оновлень. Такий компроміс дав змогу підтримувати консистентний стан, хоча і за рахунок деякої втрати асинхронності. Крім того, впровадження ідемпотентної обробки подій допомогло уникнути дублювання операцій і підвищити надійність системи.

Проблема 2: Відновлення стану замовлення з подій

У процесі реконструкції замовлення через події іноді виникали труднощі, пов’язані з коректним застосуванням подій у правильному порядку, особливо при зміні структури даних у подіях або при наявності неповних/пошкоджених даних.

Шляхи вирішення:

Для вирішення цієї проблеми було впроваджено чітку послідовність обробки подій та додано валідацію даних при десеріалізації. Також логуються помилки, щоб швидко ідентифікувати проблемні події. Для майбутніх змін у структурі подій планується використовувати версіонування payload, щоб підтримувати сумісність.

Проблема 3: Логування і трасування процесу обробки подій

Відсутність централізованого логування та кореляції подій ускладнювала пошук причин помилок під час реконструкції замовлень, особливо при масштабуванні сервісу.

Шляхи вирішення:

Було впроваджено централізовану систему логування з використанням унікальних ідентифікаторів (correlation-id), що дозволяють відстежувати послідовність подій для конкретного замовлення. Це значно спростило діагностику та прискорило усунення несправностей.

Проблема 4: Керування різними статусами замовлення

Під час зміни статусів замовлення виникали проблеми з їх синхронністю: іноді статус залишався некоректним через помилки у застосуванні подій.

Шляхи вирішення:

Запроваджено чіткий набір правил зміни статусів (наприклад, "створено", "підтверджено", "скасовано"), які контролюються в коді реконструкції. Також додано логіку обробки непередбачуваних подій і повідомлення про них у логах для подальшого аналізу.

Проблема 5: Обробка винятків при десеріалізації подій

Іноді при конвертації JSON-подій у Java-об’єкти виникали помилки, що призводили до зупинки процесу реконструкції.

Шляхи вирішення:

Додано обробку виключень із логуванням детальної інформації про помилки, а також реалізовано механізм пропуску некоректних подій із подальшим оповіщенням розробників. Це дозволяє не зупиняти весь процес через окремі некоректні записи.

Проблема 6: Масштабованість запитів до бази даних

При великій кількості подій для одного замовлення час реконструкції міг значно збільшуватись через послідовне завантаження та обробку великого обсягу даних.

Шляхи вирішення:

Оптимізовано запити до бази даних, додано індекси на ключові поля, а також планується імплементація кешування стану замовлення для швидшого доступу, щоб зменшити навантаження та прискорити обробку.

3 АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Досягнення поставленої мети

Метою даної роботи було розробити та реалізувати прикладну мікросервісну систему, яка демонструє застосування різних архітектурних патернів для забезпечення узгодженості даних у розподіленому середовищі. Основна задача полягала в побудові системи з незалежними сервісами, які взаємодіють асинхронно, при цьому гарантуючи цілісність і консистентність даних без використання централізованих транзакцій.

У процесі реалізації було впроваджено три ключові патерни — Saga, Event Sourcing та Command Query Responsibility Segregation (CQRS). Кожен із них відповідає за певний аспект управління станом системи та узгодженістю даних, а їх комбінація дозволила сформувати комплексний підхід до роботи з розподіленими транзакціями і забезпечення високої надійності.

Патерн Saga забезпечив можливість послідовного виконання локальних транзакцій із компенсуючими діями у разі помилок, що суттєво підвищило стійкість системи до часткових відмов. Водночас реалізація цього патерну виявила певні складнощі зі збереженням сильної узгодженості в деяких сервісах, що було враховано при подальшій оптимізації.

Event Sourcing дозволив фіксувати всі зміни стану системи у вигляді послідовності подій, що забезпечило прозорість, можливість аудиту та відтворення будь-якого стану у довільний момент часу. Це значно підвищило гнучкість у роботі з даними та спростило відстеження історії змін.

CQRS, у свою чергу, сприяв оптимізації операцій читання і запису, розділяючи ці два потоки на окремі моделі даних. Це покращило продуктивність системи, дозволило уникнути блокувань і зробило систему більш масштабованою.

В результаті комплексної реалізації цих патернів було досягнуто поставленої мети — створено мікросервісну систему, яка відповідає сучасним вимогам до масштабованості, надійності та консистентності. Завдяки

реалізації різних патернів і моделюванню їх роботи в типових сценаріях взаємодії сервісів, включно з обробкою збоїв і аномалій, стала можливою оцінка ефективності кожного підходу у практичних умовах.

Важливо підкреслити, що виконане дослідження не лише підтвердило життєздатність цих патернів, але й виявило їхні сильні і слабкі сторони, що є цінною інформацією для подальшого застосування у виробничих системах. Таким чином, досягнення поставленої мети дозволило сформувати практичні рекомендації для архітекторів та розробників, які працюють із мікросервісними системами.

3.2 Оцінка результатів роботи

Розроблена мікросервісна система успішно реалізує комплексний підхід до забезпечення узгодженості даних між сервісами за допомогою впровадження патернів Saga, Event Sourcing та CQRS. Кожен із цих патернів показав свою ефективність у забезпеченні консистентності та стійкості системи, а також гнучкості в обробці різних бізнес-сценаріїв.

Водночас у процесі розробки було виявлено низку технічних викликів, які вдалося подолати:

- часткова реалізація патерну Saga для збереження сильної узгодженості у сервісі «Замовлення» (забезпечено баланс між узгодженістю та продуктивністю);
- необхідність реалізації ідемпотентної обробки повідомлень для уникнення дублювання операцій у системі обміну подіями через Apache Kafka;
- оптимізація структури подій та їхнього збереження в Event Sourcing для підвищення прозорості та відтворюваності стану системи;

- розділення моделей даних у CQRS для покращення продуктивності запитів із одночасним ускладненням архітектури та підтримки.

Слід зазначити, що, попри ефективність реалізованих рішень, система потребує подальшого вдосконалення для масштабування у великих навантаженнях, а також людського контролю у випадках складних збоїв або неоднозначних ситуацій, що є типовим для розподілених систем.

3.3 Врахування світових тенденцій

Отримані в ході дослідження результати демонструють відповідність актуальним світовим тенденціям у розвитку мікросервісної архітектури та забезпеченні консистентності даних у розподілених системах. Сьогодні провідні технологічні компанії, такі як Netflix, Uber, Amazon, активно впроваджують мікросервісні підходи, що дозволяють створювати масштабовані та стійкі до збоїв програмні системи. Ці організації використовують перевірені патерни, зокрема Saga, Event Sourcing та CQRS, які стали стандартом для забезпечення узгодженості бізнес-логіки в умовах розподілених обчислень.

У цьому контексті розроблена система не лише відповідає світовим стандартам, а й враховує особливості сучасних технологій і архітектурних підходів. Особливістю розробки є комплексне впровадження трьох основних патернів консистентності в межах однієї системи, що дозволяє проводити їх порівняльний аналіз і оцінку ефективності в реальних сценаріях взаємодії мікросервісів.

Важливо підкреслити, що реалізація патерну Saga у системі була адаптована для досягнення сильної узгодженості, що відповідає сучасним вимогам надійності та цілісності даних. Використання Apache Kafka як брокера подій відповідає сучасним практикам асинхронної комунікації, що широко застосовуються у провідних хмарних і розподілених рішеннях.

Використання таких технологій, як Spring Boot і PostgreSQL, також відповідає світовим трендам у побудові гнучких і масштабованих сервісів.

На відміну від багатьох існуючих рішень, розроблена система пропонує глибокий аналіз і тестування поведінки патернів у різних умовах, включно з обробкою збоїв, дублюванням повідомлень і затримками. Такий підхід дозволяє не лише відтворити типові проблеми, а й знайти практичні шляхи їх вирішення, що відповідає потребам сучасного бізнесу в стабільності та надійності мікросервісних систем.

Таким чином, створена система є не лише прикладом впровадження передових світових тенденцій, а й робить вагомий внесок у розвиток національних IT-рішень із врахуванням специфіки локального ринку. Вона відкриває можливості для подальших досліджень і впроваджень у сфері розподілених систем, що важливо у світі, де цифровізація і автоматизація бізнес-процесів набирають обертів.

3.4 Передбачувані обласні застосування

Розроблена система та впроваджені у ній архітектурні патерни мають широкий спектр потенційних застосувань у різних галузях і сферах, де необхідна висока надійність, масштабованість та узгодженість розподілених даних. Мікросервісна архітектура, доповнена такими патернами, як Saga, Event Sourcing та CQRS, сьогодні широко використовується в багатьох провідних світових компаніях для забезпечення безперебійної роботи критичних бізнес-процесів.

Серед найбільш перспективних сфер застосування слід виокремити фінансові технології, електронну комерцію, логістику, телекомунікації та охорону здоров’я. У фінансових системах, де будь-яка помилка у транзакції може призвести до значних збитків, забезпечення консистентності розподілених сервісів є критично важливим. Розроблена система, завдяки своїй гнучкості та надійності, може слугувати основою для створення

високонавантажених платіжних систем, платформ обробки замовлень та систем управління ризиками.

У сфері електронної комерції, де необхідно координувати величезну кількість операцій із замовленнями, складськими запасами та доставкою, впроваджені патерни дозволяють ефективно управляти складними бізнес-логіками без втрати даних і помилок у станах. Це забезпечує високу якість обслуговування клієнтів та зменшує ризики операційних збоїв.

Також передбачаються застосування у системах управління ланцюгами постачання, де різні компоненти взаємодіють у реальному часі, і важливо швидко реагувати на зміни в запасах, поставках або виробництві. Тут узгодженість даних у розподілених сервісах забезпечує точність і своєчасність прийняття рішень.

Крім того, розроблена архітектура може бути інтегрована у хмарні сервіси та платформи з автоматичним масштабуванням, що робить її привабливою для глобальних технологічних компаній, які працюють із розподіленими користувачами та великими обсягами даних.

Отже, розроблені у роботі рішення не лише відповідають сучасним вимогам світового ринку, а й відкривають можливості для подальшого розвитку високонадійних, масштабованих і гнучких програмних систем, здатних працювати у складних розподілених середовищах.

3.5 Господарська, наукова та соціальна значущість

Розроблена мікросервісна система, що забезпечує узгодженість даних у розподілених архітектурах, має вагоме значення з огляду на її практичне застосування у різних сферах господарської діяльності. Надійність і масштабованість таких систем дозволяють підвищувати ефективність бізнес-процесів, знижувати витрати на підтримку інфраструктури та покращувати якість обслуговування клієнтів. Впровадження розроблених рішень сприяє оптимізації робочих процесів у фінансовому секторі, електронній комерції,

логістиці та інших галузях, де критично важливою є точність і своєчасність обробки даних.

Наукова значущість роботи полягає у системному підході до дослідження патернів узгодженості в мікросервісній архітектурі та практичній реалізації і порівнянні різних методів. Це дозволяє краще розуміти сильні та слабкі сторони кожного з підходів і сприяє розвитку теорії розподілених систем. Отримані результати можуть слугувати основою для подальших наукових досліджень, розробки нових методів та інструментів забезпечення консистентності у складних системах.

З соціальної точки зору, підвищення якості і надійності програмних систем має безпосередній вплив на кінцевих користувачів. Системи, які працюють без збоїв і забезпечують коректну обробку даних, сприяють формуванню довіри до цифрових сервісів, що особливо важливо у сфері охорони здоров’я, освіти та державного управління. Крім того, розвиток вітчизняних технологій підвищує конкурентоспроможність країни на світовому ринку IT, створює нові робочі місця та стимулює інновації.

Результати дослідження мають комплексне значення, оскільки поєднують практичну користь для бізнесу, внесок у наукову спільноту та позитивний вплив на суспільство. Вони сприяють підвищенню ефективності підприємств, розширенню теоретичних знань у галузі розподілених обчислень та покращенню якості цифрових сервісів для користувачів. Такий підхід забезпечує баланс між технологічним прогресом, економічним розвитком і соціальною відповідальністю, що робить розроблену систему важливим інструментом для сучасного інформаційного суспільства.

ВИСНОВКИ

Розроблена в межах цього дослідження мікросервісна система успішно реалізує комплексний підхід до забезпечення узгодженості даних із використанням трьох ключових архітектурних патернів — Saga, Event Sourcing та CQRS. Враховуючи актуальність проблеми консистентності у мікросервісних архітектурах, система, створена у рамках роботи, стала майданчиком для експериментального порівняння ефективності кожного з цих підходів у реальних умовах розподіленої взаємодії сервісів.

По-перше, у роботі було показано, що застосування патерну Saga є найбільш природним і зрозумілим способом управління узгодженістю у бізнес-процесах, які можна розбити на серію локальних транзакцій із компенсуючими діями у разі помилок. Це дозволяє досягати eventual consistency, при цьому зберігаючи контроль над послідовністю операцій. Водночас, Saga виявилася не завжди ефективною в сценаріях із високою частотою відмов або потребою сильної консистентності, що відповідає теоретичним обмеженням даного підходу.

Другий патерн — Event Sourcing — надав унікальну можливість відтворювати стан системи з історії подій, що забезпечує прозорість, гнучкість та можливість аудиту. Це особливо корисно для систем із високими вимогами до відновлення після збоїв і збереження історії змін. Водночас застосування Event Sourcing потребує додаткових ресурсів на зберігання та обробку подій, а також ретельного проєктування моделей домену, що може ускладнити реалізацію.

Третій патерн — CQRS — продемонстрував свою ефективність у розділенні операцій читання та запису, що дозволяє оптимізувати роботу із запитами і підвищувати продуктивність системи загалом. Особливо ця стратегія виявилася корисною у випадках, де частота читання значно перевищує кількість записів, а також при реалізації складних запитів до даних.

Поєднання цих трьох підходів у єдиній системі дало змогу на практиці дослідити їх переваги і недоліки, а також визначити контексти, у яких кожен із них є найбільш доцільним. Такий комплексний підхід є унікальним внеском у практику побудови мікросервісних систем, оскільки дозволяє не лише теоретично оцінити патерни, а й перевірити їхню поведінку під реальним навантаженням і в умовах типової для промислових систем роботи.

З технічної точки зору, реалізація через технології Spring Boot, Apache Kafka та PostgreSQL довела свою життєздатність і масштабованість. Використання брокера подій Apache Kafka забезпечило ефективну асинхронну комунікацію між сервісами, мінімізуючи вплив збоїв і підвищуючи стійкість системи. Особливо важливим було впровадження механізмів обробки помилок, ідемпотентності операцій і компенсаційних дій, що суттєво підвищило надійність платформи.

Важливою особливістю дослідження стала орієнтація на реальні умови експлуатації: імітація відмов сервісів, затримок у мережі, дублювання повідомлень і некоректного порядку подій дозволила оцінити, наскільки стійкими є впроваджені патерни і які слабкі місця слід враховувати при промисловому впровадженні.

Якщо продовжувати роботу над цим проєктом, доцільно розглянути такі напрями:

- Розробка більш гнучких та адаптивних алгоритмів компенсації помилок, які дозволять системі самостійно вибирати оптимальну стратегію відновлення в залежності від конкретного контексту та характеру збоїв.
- Впровадження розширених механізмів моніторингу та трасування, що дасть змогу у режимі реального часу відслідковувати стан транзакцій, подій та узгодженості між сервісами, сприяючи швидшому виявленню і локалізації проблем.

- Дослідження можливостей інтеграції сучасних технологій штучного інтелекту та машинного навчання для прогнозування збоїв і автоматичної оптимізації поведінки системи.
- Розширення архітектури для підтримки мультихмарних розгортань та гібридних середовищ, де питання узгодженості стають ще більш комплексними через географічний розподіл інфраструктури.
- Оптимізація зберігання і обробки подій у Event Sourcing шляхом використання нових форматів даних, стиснення та делегування обчислювальних операцій.
- Розробка зручних інтерфейсів та інструментів для користувачів і розробників, що полегшать конфігурування, адміністрування та масштабування системи.

Загалом, проведена робота підтверджує, що сучасні патерни узгодженості даних, зокрема Saga, Event Sourcing і CQRS, можуть бути ефективно застосовані в практичних мікросервісних рішеннях. Водночас їх успішне впровадження вимагає комплексного підходу, врахування специфіки бізнес-логіки, характеристик навантаження та особливостей інфраструктури.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Red Hat . Distributed transaction patterns for microservices compared. URL: https://www.dicomstandard.org/about
2. Vogels, W. (2009). Eventually consistent. Communications of the ACM, 52(1), 40–44.
3. DeCandia, G., Hastorun, D., Jampani, M., et al. (2007). Dynamo: Amazon's Highly Available Key-value Store. Proceedings of the 21st ACM Symposium on Operating Systems Principles, 205–220.
4. Lloyd, W., Freedman, M. J., Kaminsky, M., & Andersen, D. G. (2011). Don't settle for eventual: Stronger consistency for wide-area storage with COPS. Proceedings of the ACM Symposium on Operating Systems Principles, 401–416.
5. Pritchett, D. (2008). BASE: An Acid Alternative. Queue, 6(3), 48–55.
6. Brewer, E. (2012). CAP twelve years later: How the "rules" have changed. IEEE Computer, 45(2), 23–29.
7. Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media
8. Richards, M. (2019). Microservices vs. Service-Oriented Architecture. O'Reilly Media.
9. Fowler, M. (2014). Microservices. URL: https://martinfowler.com/articles/microservices.
10. Kreps, J. (2011). The Log: What every software engineer should know about real-time data's unifying abstraction. URL: https://engineering.linkedin.com/distributed-systems/
11. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. Communications of the ACM, 59(5), 50–57.
12. Vernon, V. (2013). Implementing Domain-Driven Design. Addison Wesley.
13. Garcia-Molina, H., Ullman, J.D., & Widom, J. (2008). Database Systems: The Complete Book. Pearson.

14. Pautasso, C., Zimmermann, O., & Leymann, F. (2017). Microservices in Practice, Part 1: Reality Check and Service Design. IEEE Software, 34(1), 91–98.
15. Young, G. (2010). Event Sourcing. URL: https://martinfowler.com/eaaDev/EventSourcing.html
16. Vernon, V. (2016). Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley.
17. Fowler, M., & Richardson, C. (2011). Command Query Responsibility Segregation. URL: https://martinfowler.com/bliki/CQRS.html
18. Helland, P. (2015). Immutability Changes Everything. Communications of the ACM, 58(2), 60–67.
19. AxonIQ. (2024). Axon Framework Documentation. URL: https://docs.axoniq.io
20. Temporal Technologies. (2024). Temporal Documentation. URL: https://docs.temporal.io
21. Netflix OSS. (2023). Conductor Documentation. URL: https://netflix.github.io/conductor
22. EventStore Ltd. (2024). EventStoreDB Docs. URL: https://developers.eventstore.com
23. Kreps, J., Narkhede, N., & Rao, J. (2011). Kafka: Distributed Messaging System for Log Processing. LinkedIn Engineering.
24. Apache Software Foundation. (2024). Apache Kafka Documentation. URL: https://kafka.apache.org/documentation