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)

ДОСЛІДЖЕННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ДЛЯ СТВОРЕННЯ ЗАСТОСУНКУ З ВИКОРИСТАННЯМ SAGA ТА EVENT SOURCING НА .NET: ОЦІНКА ПРОДУКТИВНОСТІ ТРАНЗАКЦІЙ У РОЗПОДІЛЕНІЙ СИСТЕМІ
(тема)

Виконав:
здобувач 2 року навчання,

групи ІНФм-24-2

Лиман І. С.
(прізвище, ініціали)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Науковий керівник доц. Руденко Д. О.
(посада, прізвище, ініціали)

Допускається до захисту

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

2025 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти другий (магістерський)

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

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Лиману Ігорю Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження архітектурних підходів до управління розподіленими транзакціями в мікросервісних системах з використанням Saga патерн та Event Sourcing затверджена наказом університету від 14 листопада 2025 року № 1045Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 19 листопада 2025 р.
3. Вихідні дані до роботи технології розробки мікросервісних архітектур, методи забезпечення узгодженості даних у розподілених системах, літературні джерела щодо патернів Saga, Event Sourcing та CQRS, інструменти для реалізації міжсервісної комунікації, програмні засоби ASP.NET Core 8.0, MongoDB, SQL Server, RabbitMQ, gRPC, Docker Compose, бібліотека Polly для забезпечення відмовостійкості, Apache JMeter для навантажувального тестування, допоміжні діаграми, схеми архітектури та статистичні матеріали, результати хаос-інженерного тестування.
4. Перелік питань, що потрібно опрацювати в роботі
1. Аналіз сучасних методів та патернів управління розподіленими транзакціями в мікросервісних системах, включаючи Saga, Event Sourcing та CQRS.
2. Розробка архітектури системи управління замовленнями з чотирма мікросервісами (OrderService, PaymentService, InventoryService, ShippingService) та Saga оркестратором.
3. Реалізація сховища подій на базі MongoDB та трьох варіантів протоколів комунікації: REST API, gRPC та асинхронні повідомлення через RabbitMQ.
4. Розробка CQRS моделей читання для оптимізації операцій з кінцевою узгодженістю.
5. Проведення навантажувального тестування (100, 250, 500 користувачів) та хаос-інженерного тестування з внесенням різних типів відмов.
6. Порівняльний аналіз трьох архітектурних підходів: базовий CRUD, Saga без ES, повна реалізація Saga + ES + CQRS.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність проблеми управління розподіленими транзакціями, об'єкт та мета дослідження, постановка задачі, діаграма хореографії Saga патерну, діаграма оркестрації Saga патерну, архітектура системи управління замовленнями, розподіл латентності для REST та gRPC, пропускна здатність асинхронної комунікації через RabbitMQ, порівняння протоколів комунікації за часом відповіді, поведінка Circuit Breaker при збої сервісу, час відповіді при недоступності бази даних, ефективність патернів відмовостійкості (зменшення MTTR), накладні витрати сховища різних підходів, порівняння запитів традиційних JOIN vs CQRS, компроміси архітектурних підходів (Performance vs Resilience), таблиці порівняння протоколів міжсервісної комунікації, результатів тестування REST API та gRPC, результатів хаос-інженерного тестування, порівняння архітектурних підходів, висновкові ілюстрації та рекомендації.

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	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	29.09.2025	
2	Аналіз завдання, підбір літератури	01.10.25-07.11.25	
3	Аналіз літератури з досліджуваної проблеми	04.10.25-14.11.25	
4	Дослідження патернів Saga, Event Sourcing та CQRS	15.10.25-24.10.25	
5	Програмна реалізація	26.10.25-04.11.25	
6	Обгрунтування отриманих результатів	04.11.25-13.11.25	
7	Оформлення пояснювальної записки	11.11.25-26.11.25	
8	Перевірка на нормоконтроль	16.12.25	
9	Перевірка на плагіат	16.12.25	
10	Рецензування	17.12.25	
11	Підготовка презентації та доповіді	17.12.25	
12	Занесення роботи в електронний архів	17.12.25	
13	Попередній захист кваліфікаційної роботи	18.12.25	

Дата видачі завдання 29 вересня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Руденко Д. О.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи: 77 с., 6 табл., 12 рис., 45 джерела.

ВІДМОВОСТІЙКІСТЬ, МІКРОСЕРВІСИ, РОЗПОДІЛЕНІ ТРАНЗАКЦІЇ, ХАОС-ІНЖЕНЕРІЯ, ASP.NET CORE 8.0, CQRS, DOCKER COMPOSE, ES, gRPC, JMETER, MONGODB, POLLY, RABBITMQ, SAGA ОРКЕСТРАТОР, SAGA PATTERN, SQL SERVER.

Об’єктом дослідження є мікросервісна система управління замовленнями з розподіленими транзакціями, що охоплюють множину автономних сервісів.

Предметом дослідження є методи та патерни забезпечення узгодженості даних у розподілених транзакціях, механізми відмовостійкості та протоколи міжсервісної комунікації в мікросервісних архітектурах.

Метою дослідження є порівняння архітектурних підходів до управління розподіленими транзакціями шляхом розробки прототипу системи з Saga патерн та ES.

Використано Saga orchestration для координації транзакцій, ES для збереження історії подій, CQRS для оптимізації, gRPC для міжсервісної комунікації. Проведено аналіз підходів до управління розподіленими транзакціями. Реалізовано чотири мікросервіси з Saga.

Наукова новизна роботи полягає у комплексному емпіричному дослідженні комбінації Saga патерн з ES на ASP.NET з порівняльним аналізом продуктивності трьох архітектурних підходів.

У результаті дослідження розроблено прототип системи управління замовленнями, проведено навантажувальне тестування при 100-500 користувачах та хаос-інженерне тестування. Отримано результати: gRPC на 45% вища пропускна здатність порівняно з REST.

ABSTRACT

Explanatory note to the qualification work: 91 pages, 6 table, 12 figures, 45 sources.

ASP.NET CORE 8.0, CHAOS ENGINEERING, CQRS, DISTRIBUTED TRANSACTIONS, DOCKER COMPOSE, ES, gRPC, JMETER, MICROSERVICES, MONGODB, POLLY, RABBITMQ, RESILIENCE, SAGA ORCHESTRATOR, SAGA PATTERN, SQL SERVER.

The object of investigation is a microservice system for managing transactions with subdivisions of transactions that support many autonomous services.

The subject of research is methods and patterns for ensuring data consistency across subdivisions of transactions, visibility mechanisms, and interservice communication protocols in microservice architectures.

The method of investigation is the alignment of architectural approaches to the management of divisional transactions along the way of developing a prototype system with Saga pattern and ES.

Vikoristano Saga orchestration for transaction coordination, ES for saving data history, CQRS for optimization, gRPC for cross-service communication. An analysis of approaches to managing subdivided transactions was carried out. Implemented several microservices from Saga.

The scientific novelty of the work lies in the complex empirical research combination of Saga pattern with ES on ASP.NET with a state-of-the-art analysis of the productivity of three architectural approaches.

As a result of the research, a prototype of the procurement management system was developed, advanced testing was carried out with 100-500 customers and chaos engineering testing was carried out. The results are taken away: gRPC is 45% of the total throughput equal to REST.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ACID – Atomicity, Consistency, Isolation, Durability (атомарність, узгодженість, ізольованість, довговічність)

API – Application Programming Interface (програмний інтерфейс застосунку)

Async – асинхронний

BASE – Basically Available, Soft state, Eventual consistency (базова доступність, м’який стан, eventual узгодженість)

Bounded Context – обмежений контекст

CAP – Consistency, Availability, Partition tolerance (теорема CAP)

Circuit Breaker – автоматичний вимикач

CQRS – Command Query Responsibility Segregation (розділення відповідальності команд і запитів)

CPU – Central Processing Unit (центральний процесор)

CRUD – Create, Read, Update, Delete (створення, читання, оновлення, видалення)

DDD – Domain-Driven Design (предметно-орієнтоване проєктування)

DTO – Data Transfer Object (об’єкт передачі даних)

ES – Event Sourcing (джерело подій)

GDPR – General Data Protection Regulation (загальний регламент захиступ даних)

gRPC – Google Remote Procedure Call (фреймворк віддаленого виклику процедур)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

JSON – JavaScript Object Notation (текстовий формат обміну даними)

Latency – латентність, затримка

MB – мегабайт

Microservices – мікросервіси

MongoDB – документо-орієнтована NoSQL база даних

ms – мілісекунда

msg/s – повідомлень за секунду

MTTR – Mean Time To Recovery (середній час відновлення)

NACK – Negative Acknowledgement (негативне підтвердження)

Polly – бібліотека resilience patterns для .NET

Protocol Buffers – формат серіалізації даних від Google

RAM – Random Access Memory (оперативна пам’ять)

RabbitMQ – брокер повідомлень

req/s (зап/с) – запитів за секунду

Resilience – відмовостійкість

REST – Representational State Transfer (архітектурний стиль передачі стану)

ВСТУП

Сучасні підприємства переходять від монолітних до мікросервісних архітектур для досягнення кращої масштабованості, незалежності розгортання компонентів та можливості використання різних технологічних стеків для різних частин системи. Мікросервіси представляють архітектурний стиль, при якому додаток структурується як набір слабко зв'язаних сервісів, кожен з яких реалізує певну бізнес-функцію і може розроблятися, розгортатися та масштабуватися незалежно. За даними звіту Gartner 2024 року, понад 70% нових корпоративних застосунків розробляються з використанням мікросервісної архітектури [1].

Однак перехід до розподіленої архітектури створює значні виклики у забезпеченні узгодженості даних при виконанні бізнес-транзакцій, що охоплюють множину сервісів. У монолітних системах транзакційна узгодженість забезпечується через ACID властивості реляційних баз даних – атомарність, узгодженість, ізольованість та довговічність. У мікросервісних системах, де кожен сервіс має власну базу даних згідно з принципом бази даних на сервіс, традиційні механізми розподілених транзакцій (Two-Phase Commit, Three-Phase Commit) виявляються непридатними через високу латентність, блокування ресурсів та погану відмовостійкість [2-4].

Класичні протоколи розподілених транзакцій вимагають синхронної координації всіх учасників та блокування ресурсів до завершення транзакції, що суперечить принципам високої доступності та незалежності сервісів у мікросервісній архітектурі. За теоремою CAP неможливо одночасно забезпечити узгодженість, доступність та стійкість до розділення мережі в розподілених системах. Сучасні мікросервісні системи віддають перевагу моделі BASE замість ACID, приймаючи кінцеву узгодженість як прийнятний компроміс для досягнення високої доступності та масштабованості [5-7].

Актуальність роботи полягає у необхідності забезпечення надійного управління розподіленими транзакціями в мікросервісних системах без компромісу щодо доступності та продуктивності. Saga патерн пропонує альтернативний підхід до управління довготривалими бізнес-транзакціями через послідовність локальних транзакцій з компенсуючими діями у випадку відмов. Event Sourcing (ES) забезпечує збереження повної історії змін стану системи як незмінної послідовності подій, що дозволяє точне відтворення стану на будь-який момент часу та значно спрощує аудит і налагодження.

Незважаючи на теоретичні переваги цих підходів, існує недостатньо емпіричних досліджень щодо їх практичної реалізації, продуктивності під навантаженням та порівняння з альтернативними рішеннями. Більшість наявних публікацій фокусуються на концептуальних аспектах або описують окремі кейси впровадження без систематичного порівняльного аналізу. Відсутні комплексні дослідження, що поєднують Saga оркестровку з ES та CQRS для побудови відмовостійких систем з можливістю детального аудиту.

Огляд сучасного стану показує, що провідні технологічні компанії активно впроваджують ці патерни: Amazon використовує Saga для управління замовленнями в платформі електронної комерції, Netflix застосовує ES для відстеження переглядів користувачів, Uber реалізував підходи Saga на основі оркестровки для координації поїздок між драйверами та пасажирами. Проте документовані деталі реалізацій залишаються фрагментарними, а порівняльні метрики продуктивності часто недоступні або несистематизовані.

Існуючі фреймворки та бібліотеки (MassTransit, NServiceBus, Axon Framework) надають інструменти для реалізації Saga та ES, але вимагають глибокого розуміння архітектурних патернів та компромісів. Відсутність стандартизованих підходів до тестування відмовостійкості, оцінки продуктивності накладних витрат та вимірювання можливого вікна узгодженості ускладнює прийняття обґрунтованих архітектурних рішень. Потрібні систематичне дослідження питання вибору між підходами Saga на

основі оркестровки та хореографії, оптимальної частоти створення знімків для ES, стратегії побудови моделей читання CQRS.

Хаос-інженерія як методологія тестування відмовостійкості набуває популярності, але його застосування до системи на основі Saga з ES залишається недостатньо дослідженим. Можливі емпіричні прогнози поведінки системи при різних типах відмов (збій сервісу, недоступність бази даних, мережеві проблеми, збій брокера повідомляє) та ефективність різних шаблонів стійкості в контексті розподілених транзакцій.

Платформа ASP.NET Core 8.0 надає сучасний стек технологій для побудови високопродуктивних мікросервісів з підтримкою мінімальних API, gRPC та нативної інтеграції з Entity Framework Core. Проте відсутні комплексні дослідження реалізації оркестровки Saga та ES саме на цій платформі з детальним аналізом продуктивності, порівнянням різних протоколів комунікації (REST, gRPC, асинхронні повідомлення) та оцінкою накладних витрат на зберігання ES.

1 ОГЛЯД ОСНОВНИХ МЕТОДІВ РОЗПОДІЛЕНИХ СИСТЕМ

1.1 Еволюція архітектурних підходів та управління транзакціями в розподілених системах

Вибір архітектурного підходу для побудови програмних систем безпосередньо впливає на їх масштабованість, надійність та здатність до еволюції. Протягом останніх десятиріч архітектура корпоративних застосунків пройшла значну еволюцію від монолітних систем до розподілених мікросервісних рішень. Однак перехід до розподілених архітектур породив фундаментальні проблеми управління транзакціями, які потребують принципово нових підходів.

Монолітний підхід передбачає розробку всіх компонентів системи як єдиної нероздільної одиниці розгортання. Основними перевагами є простота розробки, відсутність накладних витрат на міжпроцесну комунікацію та можливість використання локальних ACID-транзакцій. Проте зі зростанням складності бізнес-логіки монолітна архітектура демонструє суттєві недоліки: масштабування потребує реплікації всього застосунку, а модифікація будь-якого компонента вимагає повторного розгортання всієї системи.

Мікросервісний підхід передбачає декомпозицію системи на набір невеликих автономних сервісів, кожен з яких реалізує окрему бізнес-здатність. Фундаментальним принципом є автономія даних через патерн «База даних на сервіс». Кожен мікросервіс володіє власною базою даних, до якої інші сервіси не мають прямого доступу. Цей підхід забезпечує автономність команд та технологічну гетерогенність: сервіс обробки замовлень може використовувати PostgreSQL, тоді як сервіс каталогу товарів може обрати MongoDB.

Однак база даних на сервіс породжує проблему управління транзакціями, які охоплюють множину сервісів. У монолітній системі операція оформлення замовлення виконується в межах однієї ACID-транзакції, тоді як у

мікросервісній архітектурі вона потребує координації сервісу замовлень, платіжного сервісу, сервісу управління запасами та логістичного сервісу з власними базами даних.

CAP-теорема, сформульована Еріком Брюером у 2000 році та формально доведена у 2002 році, встановлює фундаментальне обмеження для розподілених систем [8, 9]. Теорема стверджує, що розподілена система не може одночасно гарантувати узгодженість, доступність та стійкість до розділення мережі. У реальних продуктивних середовищах мережеві розділення є неминучими, тому практичний вибір полягає між узгодженістю та доступністю. CP-системи обирають узгодженість, блокуючи операції при розділенні мережі. AP-системи обирають доступність, дозволяючи тимчасові розбіжності даних.

Для систем електронної комерції вибір між CP та AP залежить від специфіки операцій. Списання коштів потребує суворої узгодженості, тому платіжний сервіс проєктується як CP-система. Натомість перегляд каталогу товарів може толерувати кінцеву узгодженість, тому сервіс каталогу може бути AP-системою.

Альтернативою ACID є BASE-модель, яка гарантує доступність через реплікацію та допускає кінцеву узгодженість: за відсутності нових оновлень всі репліки врешті-решт досягнуть узгодженого стану. NoSQL бази даних, такі як Cassandra, DynamoDB або MongoDB, реалізують BASE-модель. У системах електронної комерції часто використовується гібридний підхід: критичні операції потребують ACID-гарантій, тоді як перегляд історії замовлень може базуватися на кінцевої узгодженості.

Two-Phase Commit (2PC) є класичним розподіленим протоколом для досягнення атомарності транзакцій. Протокол використовує централізованого координатора через два етапи: у першій фазі учасники виконують операції та блокують ресурси, у другій – координатор приймає рішення про commit або abort на основі відповідей.

Two-Phase Commit має суттєві практичні обмеження для мікросервісних систем. Головною проблемою є блокування ресурсів: якщо координатор

виходить з ладу після prepare, учасники залишаються заблокованими на невизначений час. Латентність є наступною проблемою: 2PC потребує мінімум двох раундів синхронної комунікації. Для системи електронної комерції, де операція може включати 4-5 сервісів, синхронне очікування значно погіршує користувацький досвід.

Three-Phase Commit (3PC) був розроблений для вирішення проблеми блокування при відмові координатора через додаткову pre-commit фазу. Проте 3PC не став широко застосовуваним: протокол додає третій раунд комунікації та не є повністю стійким до мережевих розділень.

Найважливішою проблемою як 2PC, так і 3PC в контексті мікросервісів є те, що обидва протоколи створюють тісну зв’язаність між сервісами. Синхронна природа цих протоколів суперечить принципам асинхронної комунікації та автономії сервісів. Згідно з дослідженнями Google, до 70% інцидентів у розподілених системах пов’язані з проблемами управління станом та відновлення після часткових відмов.

Альтернативні підходи базуються на асинхронній комунікації, кінцевої узгодженісті та компенсуючих операціях. Патерн Saga пропонує розбиття складної транзакції на послідовність локальних транзакцій з механізмами компенсації у разі відмов. ES забезпечує повну історію всіх змін стану системи, що дозволяє відновлення після збоїв та аудит операцій.

1.2 Патерн Saga, ES для розподілених транзакцій

Патерн Saga був вперше описаний Гектором Гарсія-Моліною та Кеннетом Салемом у 1987 році як механізм управління довготривалими транзакціями в розподілених базах даних. З появою мікросервісної архітектури патерн Saga отримав нове застосування як альтернатива традиційним розподіленим транзакціям для координації операцій між автономними сервісами з власними базами даних [10-12].

Основна ідея Saga полягає в заміні глобальної ACID-транзакції послідовністю локальних транзакцій, де кожна локальна транзакція оновлює базу даних в межах одного сервісу та публікує подію, що ініціює наступний крок. Замість блокування ресурсів на весь період виконання розподіленої операції кожна локальна транзакція фіксує свої зміни одразу після завершення. Це забезпечує кінцеву узгодженість, але дозволяє уникнути проблем з доступністю та продуктивністю, властивих синхронним протоколам фіксації.

Існує два основні підходи до реалізації патерну Saga: хореографія та оркестрація. У хореографічному підході кожен сервіс самостійно реагує на події, що публікуються іншими сервісами, без централізованого координатора. Хореографія забезпечує високу автономність сервісів та усуває єдину точку відмови, але ускладнює розуміння загального потоку операції та налагодження (рис. 1.1).

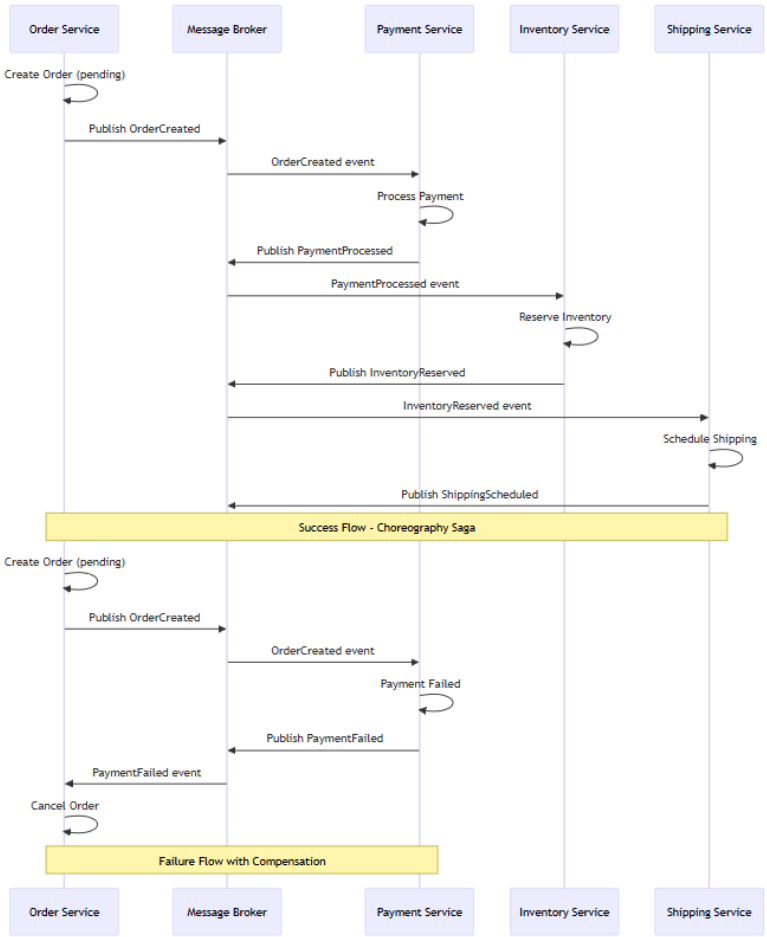

Рисунок 1.1 – Діаграма хореографії

Оркестрований підхід вирішує проблему складності через введення централізованого координатора, який явно керує послідовністю кроків Saga. Saga-оркестратор містить машину станів, що описує всі кроки процесу, умови переходів та логіку обробки помилок. Оркестратор явно викликає кожен сервіс, чекає на відповідь та приймає рішення про наступний крок на основі результату (рис. 1.2). Централізація логіки робить бізнес-процес явним та зрозумілим, спрощує налагодження та тестування. Недоліком є створення потенційної єдиної точки відмови у вигляді координатора, що потребує особливої уваги до його відмовостійкості.

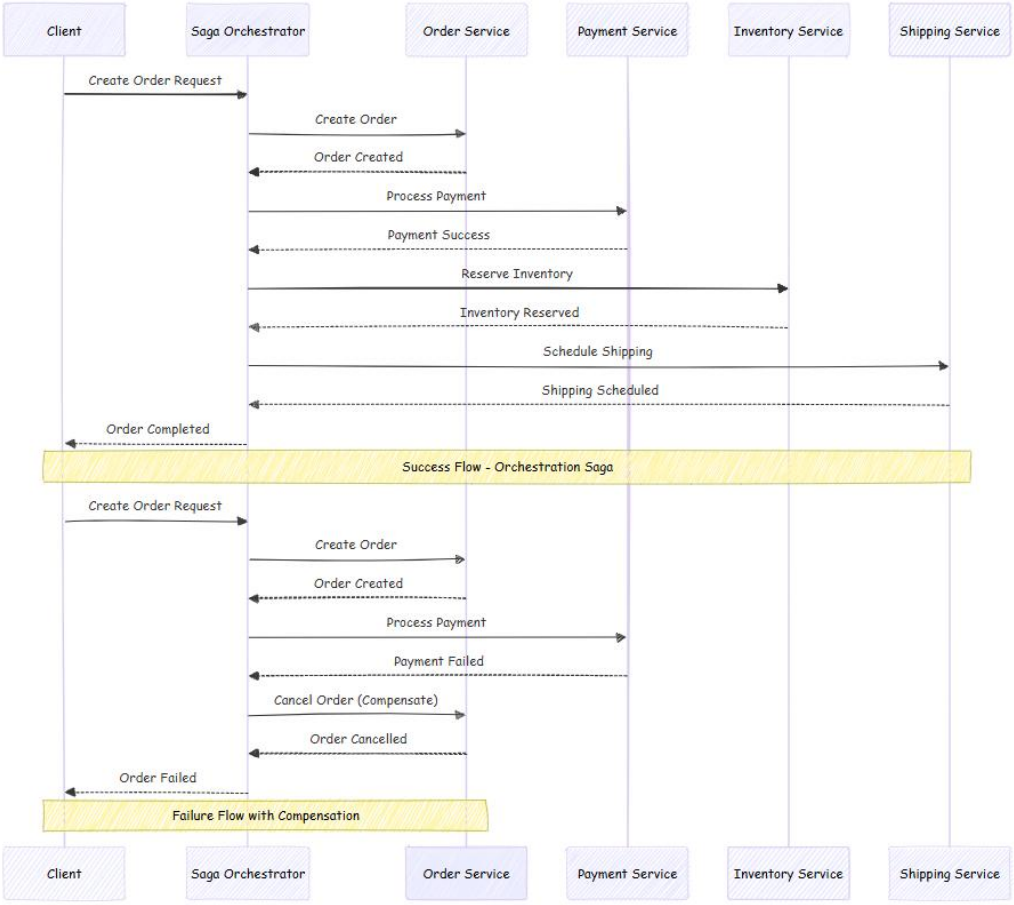

Рисунок 1.2 – Діаграма оркестрації

Компенсуючі транзакції є ключовим механізмом забезпечення консистентності в Saga при виникненні помилок. Оскільки кожна локальна транзакція фіксує свої зміни одразу, неможливо використовувати традиційний

відкат. Для кожного кроку Saga визначається компенсуюча операція, яка семантично відкочує ефект успішно виконаного кроку. Компенсація не означає видалення змін з бази даних, натомість вона створює нові записи або оновлює стан сутностей для анулювання бізнес-ефекту попередньої операції. Компенсуючі операції повинні бути ідемпотентними, оскільки можуть бути виконані більше одного разу через збої мережі.

Важливою характеристикою Saga є ізоляція між паралельними виконаннями. На відміну від ACID-транзакцій у Saga проміжні стани є видимими для інших транзакцій, які породжують аномалії типу брудних читань, втрачених оновлень та неповторюваних читань. Для пом’якшення цих проблем використовується семантичне блокування техніки, комутативні оновлення, песимістичний погляд та файл версії. Вибір між хореографією та оркестром залежить від складності бізнес-процесу: хореографія підходить для простих процесів з пріоритетною автономністю, оркестрація – для складних процесів зі значенням явного контролю над потоком.

ES є архітектурним патерном, який змінює фундаментальний підхід до збереження стану системи. Замість збереження лише поточного стану сутностей ES записує кожну зміну стану як окрему незмінну подію в хронологічній послідовності. Кожна операція над сутністю генерує доменну подію, яка описує, що саме відбулося в минулому часі: OrderCreated, PaymentProcessed, InventoryReserved [13]. Ці події зберігаються в спеціалізованому сховищі подій і ніколи не видаляються та не модифікуються. Поточний стан будь-якої сутності може бути відтворений шляхом повторення всіх подій, що стосуються цієї сутності, від моменту її створення.

Концепція Event Sourcing передбачає, що бізнес-події є ключовими елементами системи та основою для збереження її стану, а не побічними продуктами змін стану. У традиційних CRUD-системах зберігається результат операції, але втрачається інформація про те, як саме система прийшла до цього стану. ES зберігає повну історію того, що відбувалося з системою, що дозволяє

відповісти не лише на питання «яким є стан зараз», але й «яким був стан в минулому» та «як система прийшла до поточного стану».

Для системи електронної комерції замість оновлення запису замовлення в базі даних при кожній зміні статусу ES записує послідовність подій: OrderCreated з деталями товарів та клієнта, OrderValidated після перевірки наявності товару, PaymentAuthorized після успішної авторизації платежу, OrderShipped з номером для відстеження при відправці. Поточний стан замовлення відновлюється шляхом послідовного застосування всіх цих подій до початкового порожнього стану. Якщо потрібно дізнатися стан замовлення на певний момент в минулому, достатньо відтворити події до цього моменту часу.

Сховище подій є спеціалізованим сховищем для зберігання подій з оптимізацією для операцій запису та ефективного читання потоків подій. Структурно сховище організовується як колекція потоків, де кожен потік містить послідовність подій, що стосуються конкретної сутності або агрегати. Кожна подія в потоці має монотонно зростаючий номер версії, що забезпечує її унікальну позицію в послідовності та дозволяє виявляти конфлікти при конкурентних оновленнях.

Подія в сховищі подій містить кілька ключових атрибутів. Тип події, наприклад OrderCreated або PaymentProcessed. Event data містить корисне навантаження в серіалізованому форматі, зазвичай JSON, з всіма необхідними деталями операції. Метаданні включають додаткову інформацію: позначку часу створення події, ID для відстеження розподілених транзакцій, ID користувача, що ініціював операцію, та causation ID для побудови ланцюжків причинно-наслідкових зв’зків між подіями [14].

Сховище подій підтримує оптимістичний контроль конкурентності через очікуваний механізм версії. При записі нової події клієнт вказує очікувану версію потоку. Якщо фактична версія в сховищі відрізняється, це сигналізує про конкурентну модифікацію та операція відхиляється. Це дозволяє реалізувати оптимістичне блокування без блокування ресурсів. Сховище подій також забезпечує ефективне читання подій через різні патерни запиту: читання

всіх подій потоку від початку, читання подій починаючи з певної версії, читання подій в зворотному порядку, підписка на нові події в потоці для отримання нотифікацій в реальному часі.

Для оптимізації продуктивності відновлення стану з великої кількості подій використовуються снапшоти. Снапшоти є збереженим станом сутності на певний момент часу після застосування N подій. При відновленні стану система завантажує останній снапшот і повторює лише події після нього замість повторення всієї історії. Наприклад, якщо потік містить 1000 подій і є снапшот після 900-ї події, для відновлення поточного стану потрібно завантажити снапшот і повторити лише останні 100 подій.

ES надає множину переваг для розподілених систем. Повний контрольний слід автоматично створюється як побічний продукт нормальної роботи системи без додаткових зусиль. Кожна зміна даних записана як подія з міткою часу та метаданими про користувача, що критично важливо для фінансових та медичних систем з вимогами відповідності. Тимчасові запити не дозволяють підтримувати стан системи на будь-який момент у минулому шляхом повторення подій до цього моменту. Це корисно для налагодження проблем, аналізу історичних даних та регуляторної звітності [15].

Відновлення після збоїв спрощується, оскільки стан будь-якого компонента може бути відтворений з сховища подій. Якщо сервіс втрачає свій стан через збій, він може перебрати свої прочитані моделі з подій. Сховище подій стає природним джерелом для реплікації даних між різними сервісами або регіонами. Налагодження та усунення несправностей покращуються завдяки можливості відтворити точну послідовність операцій, що призвели до проблеми. Замість аналізу кінцевого стану система може відтворити події та спостерігати, як ця система прийшла до цього стану.

ES також забезпечує гнучкість для еволюції системи. Нові моделі або проєкції читання можуть бути створені в будь-який момент шляхом обробки історичних подій. Якщо потрібна нова аналітична звітність, достатньо створити нову проєкцію, яка обробляє всі історичні події та будує необхідну структуру

даних. Це дозволяє додавати нові функції без міграції наявних даних. Сховище подій стає єдиним джерелом істини для всієї системи.

1.3 Патерн CQRS для розподілених транзакцій

CQRS є патерном, який розділяє модель читання та запису системи на два окремі шляхи. Модель запису обробляє команди, що модифікують стан системи, та генерує доменні події. Модель читання містить денормалізовані дані, оптимізовані для конкретних патернів запитів. CQRS природно поєднується з ES: події зі сховища подій використовуються для побудови та оновлення моделей читання через обробники подій.

У традиційному підході одна модель даних використовується як для читання, так і для запису, що призводить до компромісів. Нормалізована схема оптимізує запис і підтримує цілісність, але ускладнює складні запити через велику кількість об’єднань. Денормалізована схема спрощує читання, але ускладнює оновлення та може призвести до аномалій даних. CQRS дозволяє оптимізувати моделі запису та читання незалежно, відповідно до їхніх конкретних вимог.

Модель запису в CQRS фокусується на бізнес-логіці та перевірці команди. Він завжди організований навколо доменних агрегаторів, які інкапсулюють бізнес-правила та генерують доменні події. Модель написання може використовувати сховище подій як основне сховище або традиційну реляційну базу даних з публікацією події. Модель читання є денормалізованою проєкцією даних, оптимізованою для конкретних випадків використання. Для систем електронної комерції можуть існувати різні моделі читання: OrderSummaryView для відображення списку замовлень, OrderDetailsView для деталей конкретного замовлення, CustomerOrderHistoryView для історії замовлень клієнта.

Обробники подій слухають події зі сховища подій та оновлюють відповідні моделі читання. Коли Order Service генерує подію OrderCreated, обробник проєкції отримує цю подію та створює або оновлює записи в модель читання. Кожна модель читання може зберігатися в найбільш підходящому типі сховища: SQL база для транзакційних запити, сховище документів для складних денормалізованих переглядів, пошукова система для повнотекстового пошуку, кеш для високої швидкості читання [16].

CQRS з ES природним чином призводить до кінцевої узгодженості між моделі запису і читання. Коли команда успішно виконується і події записуються в сховище подій, можель читання ще не оновлені одразу. Обробники подій працюють асинхронно, тому існує часове вікно між записом події та відображенням змін в модель читання. Для більшості застосунків це вікно складає мілісекунди або секунди залежно від навантаження системи [17, 18].

Кінцева узгодженість потребує особливої уваги при проєктуванні досвіду користувача. Після успішного виконання команди користувач може не побачити зміни відразу при наступному запиті. Для різних критичних операцій можна використовувати стратегії: повернення очікуваного стану разом із відповіддю на команду, опитування моделі читання до появи змін із тайм-аутом, використання WebSockets для push-повідомлень про оновлення. Для багатьох випадків використання кінцевої узгодженості є прийнятною: користувачі розуміють, що зміни можуть відображатися з невеликою затримкою. Синергія Saga, ES та CQRS створює потужну архітектурну модель для розподілених систем. Події, що публікуються кожним кроком Saga, природним чином зберігаються в сховищі подій, що забезпечує повну історію виконання розподіленої транзакції. Якщо Saga-оркестратор зазнає збою, він може відновити свій стан зі сховища подій та продовжити виконання з точки переривання. Компенсуючі транзакції також зберігаються як події, що забезпечує ідемпотентність операцій відкату та контрольний слід всіх спроб та відкатів.

Сховище подій стає природним механізмом комунікації між кроками Saga. Замість прямих викликів API або обміну повідомленнями через RabbitMQ, кроки Saga можуть реагувати на події з сховища подій через підписку. Це додає рівень індиректності, який спрощує додавання нових учасників до процесу. CQRS моделі читання забезпечують оптимізовані відображень для моніторингу стану виконання Saga: доска може відображати всі активні Saga, їх поточний крок, час виконання та проблеми.

ID кореляції, що проходить через всі події однієї Saga, дозволяє реконструювати повну історію розподіленої транзакції. Аналіз послідовності подій з однаковим ID кореляції показує точну хронологію операцій, включаючи успішні кроки, відмови та компенсації. Це критично важливо для налагодження та усунення несправностей у виробництві. Дослідження Netflix показали, що комбінація Saga та ES скорочує час відновлення після збоїв на 60% відповідно до традиційних підходів [19].

Поєднання цих патернів передбачає значні компроміси у вигляді підвищеної стійкості та операційних витрат. ES вимагає зміни мислення розробників від стано-орієнтованого до подійно-орієнтованого підходу. Обробка еволюції схеми подій потребує актуального планування подій: старі повинні залишатися зрозумілими навіть після змін у бізнес-логіці. Механізми міграції схем для конвертації старих версій подій у нові формати надають стійкість. Можлива узгодженість між моделями запису та читання потребує спеціальної обробки в інтерфейсі користувача та може бути незрозумілою для користувачів, що очікують негайної узгодженості.

Накладні витрати на зберігання всіх подій є суттєвим: замість одного запису про поточний стан замовлення зберігається 10-20 подій, які описують його життєвий цикл. Це несе на собі приблизно 20% додаткового навантаження порівняно з традиційного підходу CRUD, хоча це компенсує можливості аудиту та аналітики. Продуктивність запиту може знизитися при відтворенні великої кількості подій без належної оптимізації через знімок. Операційна складність зростає через необхідність управління сховищем подій, моніторингу

обробників подій, а також забезпечення семантики подій, таких як доставка події не меньше або не більше одного разу.

Навчання команди цим патернам потребує часу та зусиль. Розробники повинні розуміти доменні події, проєктування агрегатів, версіонування подій, побудову проєкцій та налагодження розподілених процесів через потоки подій. Тестування стає складнішим через асинхронну природу системи. Інтеграційні тести повинні враховувати кінцеву узгодженість та використовувати подієво-орієнтовані перевірки. Не всі типи застосунків виправдовують цю складність: для простих CRUD систем з невисокими вимогами до журналу аудиту традиційний підхід може бути достатнім та простішим в підтримці.

1.4 Постановка задачі дослідження

Таким чином, управління розподіленими транзакціями в мікросервісних системах з забезпеченням узгодженості даних та високої відмовостійкості є актуальним завданням. Прийнято рішення щодо розробки прототипу системи управління замовленнями з реалізацією Saga патерн для координації розподілених транзакцій, ES для збереження повної історії подій та CQRS для оптимізації операцій читання. Дослідження включає порівняльний аналіз трьох архітектурних підходів: базового CRUD без розподілених транзакцій, Saga без ES та повної реалізації Saga + ES + CQRS.

Об’єктом дослідження є мікросервісна система управління замовленнями з розподіленими транзакціями, що охоплюють множину автономних сервісів.

Предметом дослідження є методи та патерни забезпечення узгодженості даних у розподілених транзакціях, механізми відмовостійкості та протоколи міжсервісної комунікації в мікросервісних архітектурах.

Метою дослідження є порівняння архітектурних підходів до управління розподіленими транзакціями шляхом розробки прототипу системи, реалізації

Saga патерн з ES та проведення комплексного тестування продуктивності й відмовостійкості.

Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз літературних джерел щодо патернів управління розподіленими транзакціями в мікросервісних системах;

– провести аналіз сучасних підходів до забезпечення узгодженості даних, включаючи Saga, ES та CQRS;

– дослідити існуючі рішення та фреймворки для реалізації Saga-оркестрації та сховища подій на платформі ASP.NET Core;

– розробити архітектуру системи управління замовленнями з чотирма мікросервісами (OrderService, PaymentService, InventoryService, ShippingService) та Saga оркестратор;

– реалізувати сховище подій на базі MongoDB для збереження доменом подій та Saga events з підтримкою снапшотів та проєкцій;

– реалізувати три варіанти протоколів комунікації: REST API, gRPC та асинхронні повідомлення через RabbitMQ;

– розробити CQRS моделі читання для оптимізації операцій читання з кінцевою узгодженістю;

– провести навантажувальне тестування з використанням Apache JMeter при різних рівнях конкурентності (100, 250, 500 користувачів);

– виконати хаос-інженерійне тестування з внесенням різних типів відмов (збій сервісу, недоступність бази даних, мережеві проблеми);

– виміряти та проаналізувати метрики продуктивності: час відгуку, пропускна здатність, коефіцієнт помилок, використання ресурсів;

– провести порівняльний аналіз трьох архітектурних підходів: базовий CRUD, Saga без ES, повна реалізація Saga + ES + CQRS;

– оцінити накладні витрати на зберігання ES, вплив на продуктивність CQRS моделі читання та вікно остаточної узгодженості;

– сформулювати рекомендації щодо вибору оптимального архітектурного підходу для різних бізнес-вимог та операційні обмеження.

2 ТЕХНОЛОГІЧНІ ТА АРХІТЕКТУРНІ РІШЕННЯ ПРОЄКТУ

2.1 Протоколи міжсервісної комунікації

Вибір протоколу комунікації між мікросервісами суттєво впливає на продуктивність, складність інтеграції та operational характеристики розподіленої системи. Існує два основні класи комунікації: синхронна «Запит-відповідь» та асинхронна керована подіями. Кожен підхід має свої переваги та обмеження, що визначають їх застосовність для різних типів взаємодій між сервісами.

REST є найпоширенішим підходом до синхронної комунікації в мікросервісних архітектурах. REST базується на HTTP протоколі та використовує стандартні методи GET, POST, PUT, DELETE для маніпуляції ресурсами. Кожен ресурс ідентифікується унікальним URI. Відсутність стана у REST означає, що кожен запит містить всю необхідну інформацію для його обробки без збереження контексту між запитами на сервері [21-25].

JSON є стандартним форматом серіалізації даних для REST API за допомогою її читабельності та широкої підтримки на різних мовах програмування. Серіалізація об’єктів у JSON та десеріалізація відповідей виконуються автоматично більшою кількістю вебфреймворків. Проте текстовий формат JSON має витрати на зберігання за рахунок бінарних форматів: числа та булеві значення представлені як текстові рядки, що збільшує розмір корисного навантаження [26]. Синтаксичний аналіз JSON також потребує більше ресурсів процесора завдяки бінарній десеріалізації.

HTTP/1.1, який використовує більшу REST API, має фундаментальні обмеження продуктивності. Блокування головного рядка означає, що наступний запит в одному TCP-з’єднанні не може бути надісланий до відповіді на попередній запит. Для паралельного виконання запитів клієнти встановлюють множину TCP-з’єднань, але кількість одночасних з’єднань

обмежена браузерами та серверами. Заголовки HTTP передаються у вигляді текстових рядків без стиснення, що додає суттєвих витрат, особливо для невеликих корисних навантажень. Кожен повторний запит передає однакові заголовки, такі як маркери авторизації та тип вмісту.

gRPC є сучасною структурою RPC, розробленою Google, яка вирішує обмеження REST через використання HTTP/2 та Protocol Buffers. HTTP/2 забезпечує мультиплексування множини запитів та відповідей через одне TCP-з'єднання без блокування головного рядка. Стиснення заголовка через алгоритм HPACK значно зменшує накладні дані повторюваних заголовків. Server push дозволяє серверу ініціювати передачу даних клієнту без явного запиту. Двійковий кадровий шар забезпечує ефективну обробку кадрів на низькому рівні.

Protocol Buffers є незалежним від мови двійковим форматом серіалізації, який вимагає визначення схеми повідомлень у файлах .proto. Визначення схеми служить контрактом між клієнтом та сервером і використовує для генерації строго типізованого коду на різних мовах програмування. Бінарна серіалізація Protocol Buffers створює значно менше корисного навантаження порівняно з JSON: типові повідомлення на 20-30% компактніші, а в деяких випадках різниця досягає 50-70%. Десеріалізація також виконується швидше за допомогою бінарного формату без необхідності аналізу текстових рядків.

gRPC підтримує чотири типи шаблонів зв'язку. Унарний RPC є аналогом традиційного запиту-відповіді: клієнт відправляє один запит і отримує одну відповідь. Сервер потокового RPC дозволяє серверу надсилати потік повідомлень у відповідь на один запит клієнта, що корисно для передачі великих наборів даних або оновлень у реальному часі. Клієнтський потоковий RPC дозволяє клієнту надіслати потік із повідомленням сервера, який відповідає відповідним повідомленням після завершення потоку. Двонаправлене потокове передавання забезпечує одночасну передачу потоків в обох напрямках через одне підключення.

Асинхронна комунікація через брокера повідомляє, що дозволяє сервісам обмінюватися повідомленнями без прямого зв’язку та очікування миттєвої відповіді. Виробник публікує повідомлення в брокерських повідомленнях, споживач отримує та обробляє повідомлення незалежно [27]. Це забезпечує часове відокремлення: виробник і споживач не потребують бути онлайн одночасно. Брокер повідомляє, що зберігає повідомлення та гарантує їх доставку, навіть якщо споживач тимчасово недоступний.

RabbitMQ є брокером повідомлення, що реалізує протокол AMQP та шаблон черги повідомлень. Повідомлення публікуються до обміну, які маршрутизують їх до черг на основі правил маршрутизації. Споживачі підписуються на черги та підтримують повідомлення. RabbitMQ підтримує різні типи обміну: прямий обмін маршрутизує для точного збігу ключа маршрутизації, тематичний обмін дозволяє маршрутизувати на основі шаблонів за допомогою символів узагальнення, широкомовне повідомлення обміну fanout до всіх пов’язаних черг. RabbitMQ гарантує принаймні один раз доставку через підтвердження повідомлення: споживач явно підтверджує успішну обробку, інше повідомлення доставляється повторно. Черга недоставлених повідомлень захоплює повідомлення, які не можуть бути оброблені після множини спроб [28].

Kafka є розподіленою потоковою платформою, оптимізованою для високопродуктивної та відмовостійкої обробки великих обсягів подій. На відміну від традиційних черг повідомлень, Kafka зберігає повідомлення як журнал лише для додавання в темах, де кожне повідомлення має монотонно зростаючий зсув. Споживачі читають повідомлення з тем, зберігаючи свій поточний зсув, що дозволяє відтворити повідомлення з довільної позиції. Kafka забезпечує довговічність повідомлень через розділи реплікації між кількома посередниками. Групи споживачів допускають горизонтальне масштабування обробки: кожен розділ читається лише один споживач із групи, що забезпечує паралельну обробку. Kafka оптимізовано для тривалої високої пропускної

здатності через послідовне дискове введення-виведення та передачі без копіювання, досягаючи мільйонів повідомлень на секунду.

Ключовою відмінністю між RabbitMQ та Kafka є модель збереження повідомлень. RabbitMQ видаляє повідомлення після успішного підтвердження від споживача, діючи як традиційна черга. Kafka зберігає повідомлення через конфігурований період зберігання незалежно від того, чи були вони прочитані, діючи як розподілений журнал. Це дозволяє множині споживачів незалежно читати той самий потік подій та відтворювати історичні дані для відновлення стану або створення нових прогнозів.

Упорядкування повідомлень є критичною властивістю для багатьох випадків використання, але забезпечується по-різному у зв’язку з протоколом. У синхронній комунікації через REST або gRPC порядок упорядкування забезпечується послідовними викликами, але при конкурентних запитах від різних клієнтів порядок обробки не гарантований. RabbitMQ гарантує порядок FIFO для повідомлень у межах однієї черги. Однак, якщо використовувати кількох споживачів, цей порядок може бути порушено через паралельне отримання та обробку повідомлень. Kafka гарантує замовлення лише в межах розділу: усі повідомлення з одним ключем розділу потрапляють у той самий розділ і обробляються в порядку їх запису. Для забезпечення глобального впорядкування в Kafka необхідно використовувати тему з одним розділом, що обмежує паралелізм обробки [29].

Ідемпотентність є необхідною властивістю для забезпечення коректності при повторних доставках повідомлень або перезапуск запитів. Ідемпотентна операція може бути виконана багато разів з тим самим результатом, що й при одноразовому виконанні. Сутність REST API підтримує ідемпотентність для GET, PUT та DELETE методів відповідно до HTTP семантики, але POST зазвичай не є ідемпотентним. Для забезпечення ідемпотентністі на стороні клієнта POST запит генерує унікальний ідентифікатор для кожної операції та сервер перевіряє, чи не була ця операція вже виконана. В асинхронних системах ідемпотентність досягається через збереження оброблених

ідентифікаторів повідомлень або версій подій: перед обробкою повідомлення сервіс перевіряє, чи воно вже не було оброблено раніше. Порівняння протоколів комунікації представлене в таблиці 2.1.

Таблиця 2.1 – Порівняння протоколів міжсервісної комунікації

Характеристика	**REST/HTTP**	**gRPC**	**RabbitMQ**	**Kafka**
Тип комунікації	Синхронна	Синхронна	Асинхронна	Асинхронна
Формат даних	JSON (текст)	Protocol Buffers (binary)	Різні формати	Різні формати
Протокол	HTTP/1.1	HTTP/2	AMQP	Custom TCP
Throughput	Середній	Високий	Середній-Високий	Дуже високий
Latency	50-200ms	10-50ms	5-20ms	2-10ms
Streaming	Обмежено	Native підтримка	Ні	Так (native)
Message ordering	Не гарантується	Не гарантується	FIFO в queue	FIFO у партиції
Message retention	—	—	До підтвердження	Конфігурований період
Learning curve	Низька	Середня	Середня	Висока
Tooling/ecosystem	Відмінне	Добре	Добре	Добре
Browser support	Так	Обмежено	Ні	Ні

Вибір протоколу комунікації залежить від характеристики конкретної взаємодії між сервісами. REST підходить для публічних API, які споживають веб- та мобільні клієнти, завдяки широкій підтримці та простоті інтеграції. Простота налагодження через текстовий формат та наявність стандартних інструментів HTTP робить REST зручним для розробки та накладання [30, 31]. Для внутрішнього зв’язку між послугами з високими вимогами до продуктивності gRPC забезпечує кращу пропускну здатність і меншу затримку за допомогою двійкового протоколу та мультиплексування HTTP/2. Строго

типізовані контракти через буфери протоколів зменшують ризик помилок інтеграції та спрощують еволюцію API через механізми зворотної сумісності.

Асинхронна комунікація через RabbitMQ або Kafka є правильним вибором для керованих подіями архітектур та сценаріїв, де миттєва відповідь не потрібна. Відокремлення виробників та споживачів дозволяє їм масштабуватися та розгортатися незалежно. RabbitMQ підходить для традиційних моделей черги повідомлень зі складністю маршрутизації та гарантіями доставки, таких як оркестровка кроків Saga або фонова обробка завдань. Kafka є оптимальним вибором для потокової передачі подій великого обсягу, реалізації ES та сценаріїв, де потрібно відтворення історичних подій [32]. Архітектура на основі журналу Kafka природним чином підтримує кілька незалежних споживачів та тимчасові запити.

У дослідженні використано комбінований підхід: gRPC для синхронної комунікації між OrderOrchestrator і доменними службами завдяки характеристикам продуктивності, RabbitMQ для асинхронної доставки подій між сховищем подій та обробниками проєкцій, REST для публічного API, що взаємодіє з клієнтами. Це дозволяє оптимізувати кожен тип взаємодії відповідно до його специфічних вимог при збереженні систем розумної складності.

2.2 Патерни відмовостійкості

Відмовостійкість є критичною властивістю розподілених систем, оскільки часткові відмови окремих компонентів є неминучими у виробничих середовищах. Згідно з дослідженнями Google, до 70% інцидентів у розподілених системах пов’язані з проблемами управління станом та відновлення після часткових відмов. Без належних механізмів захисту відмова одного сервісу може призвести до каскадних збоїв у всій системі. Патерни

відмовостійкості забезпечують поступову деградацію функціональності замість повного відключення системи при виявленні проблем.

Автоматичний вимикач є одним з найважливіших патернів відмовостійкості, який запобігає каскадним збоям через ізоляцію проблемних сервісів. Патерн базується на трьох станах: закритий, відкритий та напіввідкритий [33, 34]. У закритому стані автоматичний вимикач пропускає всі виклики та відстежує частоту відмов у межах ковзного часового вікна. Якщо частота відмов перевищує порогове значення, автоматичний вимикач переходить у відкритий стан, блокуючи всі виклики та повертаючи виняток або резервну відповідь [35]. Після налаштованого періоду автоматичний вимикач переходить у напіввідкритий стан, пропускаючи обмежену кількість пробних викликів для перевірки відновлення сервісу.

Патерн повторних спроб забезпечує автоматичне повторення невдалих операцій для обробки тимчасових збоїв. Експоненційна затримка вирішує проблему перевантаження через збільшення затримки між спробами в геометричній прогресії: затримка = базова_затримка * 2^номер_спроби [36]. Джиттер додає випадкову складову для уникнення проблеми «тремтливого стада», коли множина клієнтів одночасно повторює спроби. Політика повторних спроб повинна враховувати ідемпотентність операцій та не застосовуватися для клієнтських помилок (HTTP 4xx).

Патерни часу очікування забезпечують обмежений час очікування для операцій. Для систем з ланцюговими викликами важливо налаштовувати час очікування таким чином, щоб час очікування верхнього за потоком сервісу був довшим за час очікування нижнього за потоком сервісу. Оптимальні значення визначаються через аналіз перцентилів часу відповіді.

Патерн перегородок ізолює ресурси для різних частин системи через створення окремих пулів потоків для кожного залежного сервісу [37, 38]. Якщо сервіс платежів стає недоступним, виклики до нього займуть лише потоки з його виділеного пулу, не впливаючи на інші сервіси.

Резервні стратегії визначають альтернативну поведінку при неможливості виконання основної операції: повернення кешованих даних, спрощеної функціональності, порожньої відповіді або збереження запиту в черзі для пізнішої обробки [39-42].

Обмеження швидкості захищає систему від перевантаження через алгоритм відра з жетонами або ковзного вікна. Перевірки справності забезпечують автоматизований моніторинг стану сервісів; платформи типу Kubernetes використовують їх для автоматичного перезапуску несправних контейнерів [43].

Polly є бібліотекою .NET для декларативного визначення політик стійкості. Типова конфігурація включає політики повторних спроб з експоненційною затримкою, автоматичного вимикача, часу очікування, перегородок та резервні політики [44]. Порядок обгортання політик має значення: найзовнішньою є резервна політика, потім автоматичний вимикач, потім повторні спроби, потім час очікування.

Комбінування патернів відмовостійкості створює глибинний захист: кожен рівень захищає від конкретних сценаріїв відмов. Правильна конфігурація базується на тестуванні продуктивності та метриках виробничого середовища.

2.3 Вибір технологічного стеку

ASP.NET Core 8.0 обрано як основну платформу для розробки мікросервісів завдяки її сучасній архітектурі та характеристикам високої продуктивності. ASP.NET Core 8.0 є кросплатформним фреймворком з нативною підтримкою Linux, macOS та Windows, що критично важливо для контейнеризованих розгортань. Фреймворк забезпечує одні з найкращих показників продуктивності серед вебфреймворків згідно з TechEmpower benchmarks.

Мінімальні APIs дозволяють створювати легковагові ендпойнти з мінімальним шаблонним кодом, що прискорює розробку простих REST API. Нативна підтримка ін’єкції залежностей забезпечує слабку зв’язаність між компонентами та спрощує тестування. Вбудований конвеєр middleware надає гнучкий механізм для обробки наскрізних проблем, таких як логування, обробка винятків та автентифікація. Модель асинхронного програмування через async/await критично важлива для операцій введення-виведення у мікросервісах, дозволяючи обробляти високу конкурентність з обмеженою кількістю потоків. Вбудована підтримка gRPC через пакет Grpc.

AspNetCore спрощує реалізацію високопродуктивної комунікації між сервісами.Entity Framework Core 8.0 використовується як ORM для взаємодії з реляційними базами даних завдяки безшовній інтеграції з ASP.NET Core. EF Core підтримує широкий спектр провайдерів баз даних, включаючи SQL Server, PostgreSQL та MySQL. Підхід code-first через міграції дозволяє визначати схему бази даних через класи C# та автоматично генерувати скрипти міграції. LINQ запити надають строго типізований спосіб написання запитів без рядків необробленого SQL, що зменшує ризик SQL ін’єкції. Оптимізація запитів включає автоматичне групування для множинних операцій та скомпільовані запити для часто виконуваних запитів. Масові операції через ExecuteUpdate та ExecuteDelete дозволяють ефективні оновлення без завантаження сутностей у пам’ять.

MongoDB обрано для сховища подій замість спеціалізованого EventStoreDB з кількох причин. MongoDB є документною базою даних з перевіреною репутацією в продуктивних середовищах та розширеним інструментарієм для моніторингу та масштабування. Документна модель природним чином підходить для зберігання подій як JSON документів з гнучкою схемою, що дозволяє еволюцію структури подій без жорстких міграцій. Змінні потоки забезпечують сповіщення в реальному часі про нові документи в колекції, що ідеально підходить для реалізації підписок на події та побудови проєкцій [43]. Конвеєр агрегації дозволяє ефективні запити для

аналізу потоків подій. Порівняно з EventStoreDB, MongoDB має нижчу операційну складність та кращу інтеграцію з існуючою інфраструктурою.

RabbitMQ обрано для асинхронної комунікації замість Kafka завдяки кращому узгодженню з вимогами дослідження. RabbitMQ є традиційним брокером повідомлень з фокусом на надійній доставці та гнучкій маршрутизації, що підходить для оркестрованого патерну Saga. Механізм підтвердження повідомлень забезпечує гарантію доставки принаймні один раз, критичну для забезпечення того, що жодна подія не втрачається в розподілених транзакціях. Черга недоставлених повідомлень автоматично захоплює повідомлення, які не можуть бути оброблені після повторних спроб. Гнучкість маршрутизації через обміни та патерни прив'язки дозволяє реалізувати складну логіку маршрутизації повідомлень. Вебінтерфейс управління надає видимість глибини черг та швидкості повідомлень для операційного моніторингу. Kafka, незважаючи на переваги для високопродуктивних потокових сценаріїв, має вищу операційну складність та крутішу криву навчання.

SQL Server використовується як операційна база даних для доменних сервісів завдяки надійним ACID гарантіям та зрілій екосистемі інструментів. Транзакційна узгодженість критична для операцій управління запасами та обробки платежів, де часткові відмови неприпустимі. Інтеграція з Entity Framework Core забезпечує безшовний досвід розробки зі строго типізованими запитами та автоматичними міграціями схеми. SQL Server Management Studio та Azure Data Studio надають потужні інструменти для адміністрування бази даних та оптимізації запитів. Вбудована підтримка типу даних JSON дозволяє гібридні реляційно-документні сценарії.

Docker використовується для контейнеризації всіх сервісів, гарантуючи узгоджене середовище виконання на всіх етапах життєвого циклу – від розробки до продуктивного середовища. Ізоляція контейнерів дозволяє кожному сервісу мати власні залежності без конфліктів. Docker Compose оркеструє мультиконтейнерні застосунки для локальної розробки та тестування. Багатоетапні збірки оптимізують розмір образу через

відокремлення залежностей збірки від залежностей виконання. Перевірки стану в Docker контейнерах інтегруються з платформами оркестрації для автоматичного перезапуску несправних контейнерів.

Polly пропонує реалізацію шаблонів відмовостійкості через декларативні політики для повтору, автоматичного вимикача, тайм-ауту та перегородки, як детально описано в розділі 2.2. Fluent API дозволяє читану конфігурацію поведінки відмовостійкості. Polly безшовно інтегрується з HttpClientFactory в ASP.NET Core для автоматичного застосування політики до викликів HTTP.

Serilog обрано як фреймворк структурованого логування завдяки багатій екосистемі приймачів та підтримці структурованих даних. Приймачі підтримують різноманітні призначення: файли для локальної розробки, Elasticsearch для продуктивної агрегації, Application Insights для Azure середовищ [45]. Збагачення логів контекстною інформацією типу ID кореляції автоматично додається до всіх подій логу.

Комбінація цих технологій створює узгоджений стек, що забезпечує необхідну функціональність для дослідження Saga патерн та ES при розумній складності. ASP.NET Core 8.0 та Entity Framework Core 8.0 надають надійну основу для побудови мікросервісів. MongoDB та RabbitMQ забезпечують спеціалізовані можливості зберігання та обміну повідомленнями без операційних накладних витрат більш складних альтернатив. SQL Server гарантує транзакційну узгодженість для критичних операцій. Docker, Polly та Serilog доповнюють стек контейнеризацією, відмовостійкістю та спостережуваністю.

2.4 Архітектура системи управління замовленнями та методологія тестування

Система управління замовленнями проєктується як набір автономних мікросервісів, кожен з яких інкапсулює окрему бізнес-здатність відповідно до принципів DDD. Декомпозиція системи базується на ідентифікації обмежених

контекстів (bounded context) – логічних меж, в яких конкретна доменна модель є валідною та узгодженою. Кожен обмежений контекст реалізується як окремий мікросервіс з власною базою даних, що забезпечує автономність розробки та розгортання.

OrderService відповідає за управління життєвим циклом замовлень. Bounded context Order включає агрегацію Order з основними атрибутами: ідентифікатор замовлення, ідентифікатор клієнта, список позицій з товарами, загальна сума та статус. Станова машина визначає переходи між статусами: Pending → Validated → PaymentAuthorized → InventoryReserved → Shipped → →Completed або Cancelled при виникненні помилок. Сервіс використовує SQL Server як операційну базу даних та публікує доменні події (OrderCreated, OrderValidated, OrderCancelled) до сховища подій для забезпечення аудиту та координації.

PaymentService обробляє платежі та взаємодію із зовнішнім платіжним шлюзом. Bounded context Payment інкапсулює логіку авторизації платежів, захоплення коштів та обробки повернень. Сервіс реалізує механізм ідемпотентності через ідемпотентні ключи для запобігання дублікатам транзакцій. Інтеграція із зовнішнім провайдером реалізована з політиками повторних спроб та circuit breaker для обробки тимчасових відмов. Сервіс публікує події PaymentAuthorized, PaymentCaptured, PaymentFailed до сховища подій та використовує SQL Server для зберігання історії транзакцій з високими вимогами до узгодженості.

InventoryService управляє інформацією про товари на складі та резервацією для замовлень. Bounded context Inventory містить агрегацію Product з доступною та зарезервованою кількістю. Сервіс забезпечує контроль конкурентності для запобігання оверселлінгу через оптимістичне блокування: кожен товар має поле версії, що інкрементується при оновленні. Механізм резервації створює тимчасову резервацію на певний період, після чого вона автоматично звільняється. Сервіс публікує події InventoryReserved,

InventoryReleased та використовує SQL Server з індексами для ефективних запитів наявності.

ShippingService відповідає за планування та відстеження доставки. Bounded context Shipping включає агрегацію Shipment з інформацією про адресу доставки, перевізника, номер відстеження та статус. Сервіс інтегрується із зовнішніми API перевізників для створення лейблів та отримання оновлень статусу. Сервіс публікує події ShippingScheduled, ShipmentDispatched, ShipmentDelivered до сховище подій та використовує SQL Server для зберігання інформації про відправлення з повним аудитом змін статусу.

SagaOrchestratorService є центральним координатором для управління розподіленою транзакцією оформлення замовлення. Оркестратор реалізує станову машину, що визначає послідовність кроків Saga та логіку компенсації при відмовах. Станова машина містить стани: Started, OrderCreated, PaymentAuthorized, InventoryReserved, ShippingScheduled, Completed та Compensating з відповідними переходами. Кожен крок реалізований як синхронний виклик до відповідного domain service через gRPC для високопродуктивної комунікації.

Оркестратор зберігає свій стан в MongoDB сховищі подій як послідовність подій: SagaStarted, StepCompleted, StepFailed, CompensationStarted, SagaCompleted. Це дозволяє відновлення стану після збоїв через відтворення подій зі сковищем подій. Кожен екземпляр Saga ідентифікується унікальним sagaId та ID кореляції для розподіленого трасування. Логіка компенсації визначає, які кроки потребують відкату залежно від етапу виникнення помилки. Якщо авторизація платежу не вдалася, потрібно лише скасувати замовлення. Якщо резервація товару не вдалася після успішного платежу, потрібно виконати повернення коштів та скасувати замовлення. Компенсаційні дії виконуються в зворотному порядку від проблемного кроку до початку.

EventStoreService забезпечує централізоване зберігання всіх доменних подій та Saga події. Сервіс використовує MongoDB як резервне сховище з

колекцією подій, де кожна подія є документом з полями: eventId, streamId, eventType, eventData, metadata, version, timestamp. Індекси створені на streamId для ефективного читання подій конкретного потоку, на eventType для запитів подій певного типу та на ID кореляції для відстеження розподілених транзакцій. API дозволяє додавання подій до потока з оптимістичним контролем конкурентності, читання подій потока та підписку на нові події в реальному часі через MongoDB Change Streams. Механізм снапшотів оптимізує відновлення стану агрегації з великою кількістю подій через збереження проміжних станів кожні N подій. API служить єдиною точкою входу для зовнішніх клієнтів та виконує маршрутизацію запитів до відповідних бекенд-сервісів. Шлюз реалізує наскрізн проблеми: автентифікацію через JWT токени, авторизацію на основі ролей користувачів, обмеження швидкості запитів для захисту від зловживань, логування запитів/відповідей та автоматичний вимикач для бекенд-сервісів.

Архітектура системи представлена на рисунку 2.3. Кожен деменний сервіс має власну базу даних. SagaOrchestrator координує розподілені транзакції через синхронні gRPC виклики. Всі доменні події публікуються до EventStoreService, який забезпечує персистентне зберігання. API маршрутизує запити від клієнтів до внутрішніх сервісів.

Для кожного доменного сервісу та MongoDB для сховища подій було виділено окерму базу даних на SQL Server. Кожен сервіс має ексклюзивне володіння своїми даними без прямого доступа від інших сервісів. Це забезпечує слабку зв’язаність на рівні даних та дозволяє незалежну еволюцію схем через міграції. Стратегії резервного копіювання налаштовуються окремо для кожної бази відповідно до критичності даних [39].

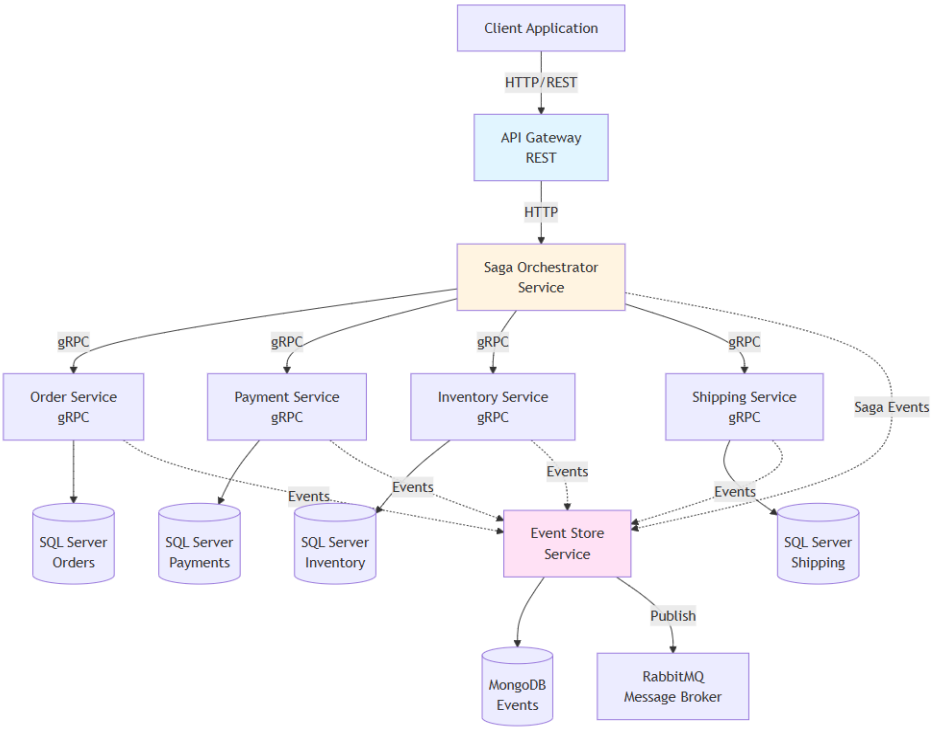

Рисунок 2.3 – Архітектура системи

Патерни комунікації комбінують синхронну та асинхронну взаємодію. Saga оркестратор використовує синхронний запит-відповідь через gRPC для виклику доменного сервісу, оскільки потребує негайної відповіді для прийняття рішення про наступний крок. Доменні сервіси публікують події асинхронно до сховища подій без очікування підтвердження від споживачів. Сховище подій публікує події до RabbitMQ для розподілу множині споживачів, які обробляють події незалежно для побудови моделі читання. Черга недоставлених повідомлень в RabbitMQ захоплюють повідомлення, які не можуть бути оброблені після множини спроб. Ідемпотентні обробники подій забезпечують коректну обробку дублікатів.

Топологія розгортання підтримує локальне розгортання через Docker Compose та виробниче розгортання через Kubernetes. Конфігурація Docker Compose вибирає всі служби як контейнери з їх залежностями, мережею та монтуванням томів. Складений файл розробки містить контейнери SQL Server, MongoDB, RabbitMQ та всі сервіси додатків. Розгорути або взаємодіяти зі службами можна через імена DNS. Перевірки працездатності визначені для

автоматичного перезапуска несправних контейнерів. Обмеження ресурсів (CPU та пам’ять) налаштовані для умов моделювання виробництва.

Розгортання Kubernetes організовано через простори імен для ізоляції середовищ. Кожен мікросервіс розгортається для забезпечення бажаної кількості реплік. Сервісні об’єкти забезпечують стабільний баланс мережевих кінцевих точок та завантажень. ConfigMaps зберігають конфігурацію, Secrets зберігають чутливі дані. Horizontal Pod Autoscaler автоматично масштабує репліки на основі використання ЦП або метрики швидкості запитів. Поступові оновлення дозволяють розгортання без простоїв через поступову заміну старих подій новими версіями.

Методологія тестування включає тестування продуктивності, тестування стійкості через хаос-інженерійне та функціональне тестування розподілених транзакцій. Метрики продуктивності збираються на кількох рівнях. Час відгука вимірюється як час від отримання запиту до відправки відповіді з процентилі: медіана (50th), 95th, 99th для розуміння розподілу латентності. Пропускна здібність вимірюється як кількість успішно оброблених запитів на секунду при різних рівнях конкурентності. Коефіцієнт помилок визначається як відсоток невдалих запитів з розбивкою за типами помилок: клієнтські помилки (4xx), серверні помилки (5xx), таймаути.

Використання ресурсів моніториться для кожного сервісу: використання CPU як відсоток процесорного часу, використання пам’яті (включаючи heap та non-heap), метрики garbage collection для .NET сервісів. Мережеві метрики включають байти (відправлені/отримані), використання пул з’єднань. Метрики бази даних включають час виконання запитів, час очікування з’єднання, взаємоблокування. Метрики брокера повідомлень включають пропускну здатність повідомлень, глибину черг, затримку споживачів.

Saga-метрики вимірюють характеристики розподілених транзакцій: вони визначають відсоток Saga, що успішно завершилися без компенсації, загальний час їх виконання від старту до завершення, а також частку Saga, які потребували компенсаційних дій через відмови. Метрики на рівні кроків

вимірюють латентність та частоту помилок для кожного окремого кроку для ідентифікації вузьких місць. Кількість конкурентних Saga показує одночасно виконувані екземпляри для оцінки навантаження системи.

Тестування навантаження виконується з використанням Apache JMeter для генерації реалістичних патернів навантаження. Тестувальний план JMeter визначає группи потоків, що симулюють конкурентних користувачів з різними патернами запитів: створення нового замовлення, перегляд існуючих замовлень, скасування замовлення. Період нарощування поступово збільшує кількість конкурентних потоків. Час очікування між запитами моделює реалістичну поведінку користувачів. Твердження перевіряють коректність відповідей: HTTP статус коди, вміст тіла відповіді, пороги часу відповіді.

Сценарії тестування покривають різні профілі навантаження. Базовий сценарій виконує нормальну роботу без штучних відмов: 500 конкурентних користувачів створюють замовлення зі стабільною швидкістю протягом 10 хвилин для вимірювання базової продуктивності. Сценарій пікового навантаження збільшує конкурентоспроможність до 1000 користувачів для тестування поведінки під високим навантаженням. Сценарій постійного навантаження виконується помірне навантаження протягом тривалого періоду (1 година) для виявлення витоків пам’яті. Сценарій піку створює різкі сплески навантаження для тестування еластичності та відновлення.

Хаос-інженерійний підхід систематично вносить відмови для валідації механізмів відмовостійкості. Сценарій помилки на сервері зупинки дає один із доменних серверів під час виконання Saga для тестування логіки компенсації та поведінки вимикача. Сценарій недоступності бази даних симулює втрату з’єднання з базою даних для тестування політики повторних спроб та обробки таймаутів. Сценарій затримки мережі вносить штучну затримку в мережеву комунікацію. Негативний сценарій брокера повідомляє, що зупиняє RabbitMQ для тестування буферизації та поведінки кінцевої узгодженості.

Тестове середовище розгортається через Docker Compose з преконфігурованою інфраструктурою: SQL Server з ініціалізованими схемами,

MongoDB з індексами, RabbitMQ з обмінами та чергами. Prometheus та Grafana додаються для збору метрик в реальному часі та візуалізації. Фаза розминки виконується перед основним тестуванням для стабілізації продуктивності системи: кілька сотень запитів для заповнення кешів, ініціалізації пул з'єднань, JIT компіляції в .NET runtime. Підготовчий запуск триває 2-3 хвилини та його результати не включаються в фінальні вимірювання.

Збір даних виконується через множину інструментів. JMeter збирає метрики на стороні клієнта: час відповіді, пропускну здатність, помилки. Application Insights збирає серверну телеметрію: трейси запитів, деталі винятків, виклики залежностей. Prometheus збирає метрики точок доступу кожного сервісу кожні 15 секунд для даних часових рядів. Serilog записує структуровані логи до Elasticsearch для детального аналізу відмов. Аналіз даних включає статистичний аналіз зібраних метрик, обчислення перцентилів для часу відповіді, графіки пропускної здатності, категоризацію помилок за типами, кореляцію ресурсів та аналіз часових рядів для виявлення трендів та аномалій.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Реалізація мікросервісів та Saga оркестратор

Реалізація системи управління замовленнями базується на ASP.NET Core 8.0 з використанням мінімальних API та Entity Framework Core 8.0 для доступу до даних з підтримкою асинхронних операцій та оптимістичного контролю конкурентності. Кожен мікросервіс реалізує специфічну бізнес-логіку свого обмеженого контексту з дотриманням принципів доменно-орієнтованого проєктування.

Order Service реалізовано з доменною моделлю, що представляє корінь агрегату з повною інкапсуляцією бізнес-логіки. Ключовою особливістю є інкапсуляція переходів між станами через методи доменної моделі з приватними сетерами для всіх властивостей. Метод Validate() дозволяє перехід лише зі стану Pending до Validated, MarkPaymentAuthorized() переводить замовлення з Validated до PaymentAuthorized, метод Cancel() може виконуватися з будь-якого стану, окрім Completed. Кожен метод перевіряє поточний стан агрегату та викидає InvalidOperationException при спробі некоректного переходу.

Нижче представлені ключові методи domain model Order.

Лістинг 3.1 Ключові методи domain model Order

```
public class Order
{
   public Guid Id { get; private set; }
   public OrderStatus Status { get; private set; }

   public void Validate()
   {
      if (Status != OrderStatus.Pending)
    {
```

```
            throw new InvalidOperationException("Invalid state transition");
        }
        Status = OrderStatus.Validated;
    }

    public void Cancel(string reason)
    {
        if (Status == OrderStatus.Completed)
            throw new InvalidOperationException("Cannot cancel completed order");
        Status = OrderStatus.Cancelled;
    }
}
```

При створенні нового замовлення через POST /api/orders виконується валідація вхідних даних та створення доменного агрегату через статичний фабричний метод Create(). Збереження виконується в SQL Server через Entity Framework Core з використанням патерну «Одиниця роботи». Після успішного збереження публікується подія OrderCreated до сховища подій. Критичним є строга послідовність: спочатку зміни зберігаються в операційній базі даних, і лише після успішного завершення транзакції публікується доменна подія.

Payment Service реалізовано з використанням gRPC для високопродуктивної комунікації між мікросервісами. Protocol Buffers забезпечує строго типізований контракт з бінарною серіалізацією, що зменшує розмір повідомлень на 60-70% порівняно з JSON. Реалізація включає політики повторів, запобіжник та політики часу очікування. Інтеграція з платіжним шлюзом виконується через адаптер, що ізолює доменну логіку від специфіки постачальника.

Нижче представлений gRPC метод авторизації платежу з обробкою помилок.

Лістинг 3.2 gRPC метод авторизації платежу з обробкою помилок

```
public override async Task<AuthorizePaymentResponse> AuthorizePayment(
    AuthorizePaymentRequest request, ServerCallContext context)
{
    var payment = new Payment { /* initialize */ };

    try
    {
        var authResult = await _gateway.AuthorizeAsync(payment.Amount,
            request.PaymentMethod);
        payment.Status = PaymentStatus.Authorized;

        _dbContext.Payments.Add(payment);
        await _dbContext.SaveChangesAsync();
        await _eventStore.AppendEventAsync($"payment-{payment.Id}",
            "PaymentAuthorized", new PaymentAuthorizedEvent { PaymentId =
payment.Id });

        return new AuthorizePaymentResponse { PaymentId =
payment.Id.ToString() };
    }
    catch (PaymentGatewayException ex)
    {
        throw new RpcException(new Status(StatusCode.Internal, ex.Message));
    }
}
```

Ключовою особливістю Payment Service є забезпечення ідемпотентності через перевірку існування платежу за ідентифікатором замовлення перед створенням нової транзакції. Це критично для запобігання подвійним списанням при повторних викликах від оркестратора Saga після мережевих збоїв.

Оркестратор Saga реалізує станову машину для управління розподіленою транзакцією з можливими станами: Started, OrderCreated, PaymentAuthorized, InventoryReserved, ShippingScheduled, Completed, Compensating, Failed.

Оркестратор виконує послідовність кроків через gRPC виклики з передачею ідентифікатора кореляції для розподіленого трасування. Після кожного кроку зберігається інформація в колекції CompletedSteps з міткою RequiresCompensation.

При виникненні помилки оркестратор переходить у стан Compensating та виконує компенсуючі дії в зворотному порядку: ReleaseReservation, RefundPayment, CancelOrder. Кожна компенсуюча дія обгорнута в окремий блок обробки виключень для ізольованої обробки помилок.

Нижче представлений Saga Orchestrator з логікою виконання та компенсації.

Лістинг 3.3 Saga Orchestrator з логікою виконання та компенсації

```
public async Task<OrderSaga> ExecuteSagaAsync(CreateOrderCommand cmd)
    {
        var saga = new OrderSaga { SagaId = Guid.NewGuid(), State = 
SagaState.Started };

        try
        {
            var order = await _orderClient.CreateOrderAsync(cmd);
            saga.OrderId = order.Id;
            saga.CompletedSteps.Add(new SagaStep
                { StepName = "CreateOrder", RequiresCompensation = true });
            await _eventStore.AppendEventAsync($"saga-{saga.SagaId}",
                "StepCompleted", new StepCompletedEvent { StepName = 
"CreateOrder" });

            var payment = await _paymentClient.AuthorizePaymentAsync(order.Id, 
order.TotalAmount);
            saga.PaymentId = payment.Id;
            saga.CompletedSteps.Add(new SagaStep
                { StepName = "AuthorizePayment", RequiresCompensation = true });

            saga.State = SagaState.Completed;
            return saga;
        }
```

```
    catch (Exception ex)
    {
        saga.State = SagaState.Compensating;
        await CompensateSagaAsync(saga);
        saga.State = SagaState.Failed;
        return saga;
    }
}
```

Стан Saga зберігається після кожного кроку через публікацію події StepCompleted до сховища подій з ідентифікатором кореляції, що проходить через всі виклики між мікросервісами для розподіленого трасування та аналізу проблем.

Політики відмовостійкості конфігуруються через Polly: політика повторів з експоненційною затримкою (2, 4, 8 секунд) та випадковим відхиленням, запобіжник з відкриттям після 5 невдач протягом 10 запитів та блокуванням на 30 секунд. Політики комбінуються в конвеєр: зовні запобіжник, далі повтори, всередині час очікування.

Механізм відновлення після збоїв забезпечує відновлення стану незавершених Saga через відтворення подій зі сховища. При старті оркестратор завантажує події SagaStarted без парних SagaCompleted або SagaFailed, відновлює стан через застосування подій StepCompleted та продовжує виконання з наступного непройденого кроку, покладаючись на ідемпотентність операцій доменних сервісів.

3.2 Сховище подій та обробники подій

Сервіс сховища подій забезпечує централізоване, високопродуктивне та надійне зберігання всіх доменних подій та подій Saga з підтримкою ефективних запитів та підписок в реальному часі. Реалізація оптимізована спеціально для патерну «Лише додавання» з мінімальною затримкою запису нових подій та

ефективною пропускною здатністю читання для читання потоків подій. MongoDB обрано як сховище завдяки його гнучкій моделі документів що природним чином підходить для зберігання гетерогенних подій як самоописуваних JSON документів з гнучкістю схеми для еволюції структур подій.

Додатковими перевагами MongoDB є вбудована підтримка потоків змін для сповіщень про події в реальному часі, горизонтальна масштабованість через сегментування для обробки великих обсягів подій, та ефективні вторинні індекси для різних шаблонів запитів. Документо-орієнтована природа MongoDB дозволяє зберігання подій з різною структурою в одній колекції без необхідності міграцій схеми при додаванні нових типів подій або полів до існуючих подій.

MongoEventStore реалізує критичний метод AppendEventAsync з складним механізмом оптимістичного контролю конкурентності через перевірку очікуваної версії для забезпечення атомарних операцій додавання. При додаванні нової події до потоку клієнт може опційно вказати очікувану версію потоку, що представляє версію останньої події, яку клієнт бачив під час читання потоку. Якщо фактична версія потоку в базі даних відрізняється від очікуваної версії, це означає одночасну модифікацію іншим процесом, і операція додавання відхиляється з винятком конкурентності для повтору зі свіжим станом.

Це забезпечує строгу атомарність та консистентність операцій додавання навіть при одночасних конкурентних записах до одного потоку від множини розподілених екземплярів сервісу без потреби в розподілених блокуваннях або двофазному коміті. Метод GetStreamVersionAsync виконує ефективний запит з проєкцією лише на поле версії для знаходження останньої події в потоці через сортування за версією в спадному порядку з обмеженням 1, та повертає її версію + 1 як наступну доступну версію для нової події.

Нижче представлений лістинг сховища подій з оптимістичним контролем конкурентності.

Лістинг 3.4 Сховище подій з оптимістичним контролем конкурентності

```
public async Task AppendEventAsync(string streamId, string eventType,
    object eventData, int? expectedVersion = null)
{
    var currentVersion = await GetStreamVersionAsync(streamId);

    if (expectedVersion.HasValue && currentVersion !=
expectedVersion.Value)
        throw new ConcurrencyException(
            $"Expected version {expectedVersion}, got {currentVersion}");

    var eventDoc = new EventDocument
    {
        StreamId = streamId,
        EventType = eventType,
        EventData = JsonSerializer.Serialize(eventData),
        Version = currentVersion + 1,
        Timestamp = DateTime.UtcNow,
        CorrelationId = Activity.Current?.Id
    };

    await _events.InsertOneAsync(eventDoc);
}
```

Складені та вторинні індекси MongoDB ретельно спроєктовані для оптимізації широкого спектру різних шаблонів запитів з мінімальною затримкою: складений індекс на комбінації полів (StreamId, Version) забезпечує максимально швидке читання всіх подій конкретного потоку в строгому порядку версій для сценаріїв відтворення подій, індекс одного поля на EventType дозволяє ефективну фільтрацію подій певного типу для цільової обробки подій, індекс на поле Timestamp підтримує часові запити для часового аналізу та звітності, індекс на CorrelationId критичний для розподіленого трасування та відстеження всіх подій пов’язаних з конкретною бізнес-операцією через розподілену систему.

Правильно налаштовані індекси забезпечують затримку менше мілісекунди для абсолютної більшості запитів навіть з багатомільйонними колекціями подій у продуктивних розгортаннях. Селективність індексів аналізується через плани пояснень для валідації ефективності шляхів виконання. Складені індекси покривають множину шаблонів запитів для зменшеної надлишковості зберігання індексів.

Обробники проєкцій реалізовані як довготривалі фонові сервіси, що безперервно підписуються на нові події зі сховища подій та асинхронно будують високооптимізовані денормалізовані моделі читання спеціально адаптовані для специфічних вимог запитів. Кожен обробник проєкцій слухає події певних релевантних типів через API потоків змін MongoDB та виконує відповідні трансформації для оновлення матеріалізованих представлень в базі даних моделей читання. Денормалізація включає попереднє об’єднання даних з множини агрегатів, попереднє обчислення обчислюваних значень та попередню фільтрацію для специфічних випадків використання.

Наприклад, обробник OrderSummaryProjection безперервно слухає доменні події OrderCreated, PaymentAuthorized, OrderShipped, OrderCancelled та інтелектуально оновлює колекцію OrderSummaryView з комплексною денормалізованою інформацією оптимізованою для ефективних запитів в панелях користувачів та звітах. Проєкція включає деталі клієнта, позиції замовлення з інформацією про продукт, статус платежу, відстеження доставки, та попередньо обчислені суми для негайного відображення без потреби в дорогих об’єднаннях або агрегаціях.

Обробники проєкцій є строго ідемпотентними через систематичне використання операцій вставки-оновлення з перевіркою унікальних обмежень та верифікацією існування записів перед операціями вставки. При обробці події OrderCreated обробник виконує складну операцію вставки-оновлення з фільтром на OrderId, що атомарно створює новий запис підсумку, якщо його ще немає в колекції, або не виконує операцію, якщо запис вже існує. Це означає, що подія вже була оброблена раніше через повтор або відтворення.

Ідемпотентна обробка критична для коректності при семантиці доставки принаймні один раз.

Кожен обробник проєкцій персистентно зберігає контрольну точку з точною позицією останньої успішно обробленої події в спеціалізованій колекції контрольних точок для надійного відновлення після перезапуску екземпляра або відновлення від збою. Контрольна точка включає токен відновлення потоку змін MongoDB, мітку часу останньої події та версію події для точного позиціонування. При перезапуску обробник завантажує збережену контрольну точку та безшовно продовжує обробку з точної точки, де зупинився без пропуску подій або дублювання обробки.

Механізм знімків оптимізує продуктивність відновлення стану агрегатів з великою кількістю накопичених подій через періодичну матеріалізацію стану. Знімок представляє повний серіалізований стан доменного агрегату на конкретний момент після застосування N подій, що дозволяє швидку гідратацію агрегату без необхідності відтворення всіх історичних подій. Метод LoadAggregateAsync реалізує оптимізований шаблон завантаження: спочатку завантажує останній доступний знімок для потоку з колекції знімків, десеріалізує стан агрегату з даних знімка через рефлексію або спеціалізований десеріалізатор, потім інкрементно завантажує та послідовно застосовує лише недавні події після версії знімка для приведення агрегату до поточного стану.

Це значно швидше, ніж повне відтворення всіх подій з початку потоку, особливо для довгоживучих агрегатів з тисячами подій. Тестування продуктивності показало поліпшення в 10-50 разів у часі завантаження агрегату залежно від кількості подій. Стратегія знімків балансує переваги продуктивності з надлишковістю зберігання та толерантністю до застарілості.

Знімки створюються автоматично за налаштовуваним інтервалом через фоновий процес: після успішного відтворення подій перевіряється, чи загальна версія кратна інтервалу знімків (типово 100 подій), і якщо умова виконується – поточний повністю гідратований стан агрегату серіалізується з метаданими та зберігається в спеціалізованій колекції знімків з посиланням на відповідний

потік та версію. Серіалізація використовує ефективний бінарний формат або стиснутий JSON для мінімізації слідів зберігання.

Для агрегатів з виключно високою частотою змін (рахунки високочастотної торгівлі, дані датчиків в реальному часі) знімки створюються частіше через зменшений інтервал, наприклад кожні 50 або навіть 25 подій для підтримки розумної продуктивності завантаження. Створення знімків налаштовується для кожного типу агрегату через конфігурацію для гнучкості. Складна стратегія утримання забезпечує, що зберігаються лише останні N знімків для кожного потоку (типово 5-10) з автоматичним видаленням старіших знімків через фонове завдання очищення для ефективного використання сховища без необмеженого зростання.

Потоки змін MongoDB надають потужні сповіщення в реальному часі про нові документи в колекції подій з низькою затримкою та високою надійністю. Метод SubscribeToEventsAsync створює персистентний курсор потоку змін з налаштовуваним фільтром на операції вставки для отримання сповіщень лише про нові події та автоматичного виклику зареєстрованого зворотного виклику обробника для кожної нової події одразу після вставки. Це дозволяє обробникам проєкцій обробляти події з мінімальною наскрізною затримкою типово 10-50мс від операції вставки до доставки сповіщення без неефективного опитування бази даних з витраченими ресурсами.

Механізм токенів відновлення забезпечує семантику обробки рівно один раз навіть при мережевих збоях або перезапусках сервісів: при тимчасовому відключенні або аварії обробник автоматично зберігає останній оброблений токен відновлення як контрольну точку та безшовно відновлює потік змін з цієї збереженої позиції після успішного повторного підключення без пропуску подій або дублювання сповіщень. Токени відновлення глобально унікальні та впорядковані для надійного відтворення.

3.3 Результати тестування продуктивності

Комплексне тестування продуктивності проводилося для систематичного порівняння різних протоколів комунікації при різних рівнях одночасного навантаження з множиною сценаріїв користувачів. Для кожного тестованого протоколу систематично вимірювалися ключові метрики продуктивності: час відповіді з детальним розподілом перцентилів (50-й, 75-й, 90-й, 95-й, 99-й) для розуміння поведінки хвоста затримки, загальна пропускна здатність системи вимірювана в запитах на секунду, відсоток частоти помилок для оцінки надійності, детальне використання ресурсів, включаючи використання процесора, споживання пам’яті, пропускну здатність мережі, насичення пулу підключень до бази даних при реалістичних рівнях навантаження 100, 250 та 500 конкурентних користувачів виконуючих типові робочі процеси.

Тестування REST API демонструвало базову продуктивність традиційної синхронної HTTP комунікації зі стандартною JSON серіалізацією для корисних навантажень запиту/відповіді. При помірному навантаженні 100 конкурентних користувачів система показала прийнятні результати з середнім часом відповіді 145мс та розумною пропускною здатністю. При збільшенні до 250 одночасних користувачів час відповіді помітно зріс до 320 мс через збільшену конкуренцію за ресурси та затримки черги. При високому навантаженні 500 конкурентних користувачів система відчувала значний стрес з середнім часом відповіді, досягаючи 850 мс, що перевищує прийнятний поріг користувацького досвіду.

Пропускна здатність системи демонструвала нелінійний шаблон зростання: спочатку зросла з 680 запитів/с при 100 користувачах до піку 820 запитів/с при 500 користувачах, але частота помилок різко збільшилася з низьких 0.2% при мінімальному навантаженні до тривожних 5.2% при максимальному тестованому навантаженні в основному через винятки часу очікування під стійким високим навантаженням, коли пули з’єднань вичерпані та запити в черзі. Результати тестування REST API представлені в таблиці 3.1.

Таблиця 3.1 – Результати тестування REST API

Конкурентні користувачі	Середній час відповіді, мс	95-й перцентиль, мс	Пропускна здатність, зап/с	Частота помилок
100	145	280	680	0.20%
250	320	620	780	1.80%
500	850	1650	820	5.20%

Детальний аналіз розбивки затримки через розподілене трасування та профілювання показав, що приблизно 40% загального часу витрачається на запити до бази даних, включаючи отримання з’єднання з пулу та фактичне виконання запиту, 25% надлишковості на процесоро-інтенсивні операції JSON серіалізації/десеріалізації (особливо для складних вкладених графів об’єктів), 20% споживається мережевим введенням/виведенням для передачі відносно великих JSON корисних навантажень через мережевий стек, решта 15% припадає на фактичне виконання бізнес-логіки, включаючи операції доменної моделі та валідацію.

Комплексний аналіз вузьких місць через метрики моніторингу виявив, що при перевищенні приблизно 400+ одночасних запитів система починає відчувати вичерпання пулу підключень SQL Server через обмежений розмір пулу конфігурації та довготривалі запити, що утримують з’єднання. Це призводить до черги запитів для доступного з’єднання з експоненційно зростаючим часом очікування та каскадною затримкою. Цільові зусилля оптимізації, включаючи покращення індексів бази даних, зменшили середній час виконання запитів бази даних на 30%, але обчислювальна надлишковість JSON серіалізації залишилася значним фактором продуктивності для REST API через притаманну неефективність текстового формату, що вимагає парсингу.

Тестування gRPC демонструвало суттєво кращу загальну продуктивність порівняно з REST завдяки ефективній бінарній Protocol Buffers серіалізації. Це різко зменшує розмір корисного навантаження та надлишковість обробки плюс передові можливості мультиплексування HTTP/2, що дозволяють одночасні запити на одному TCP з’єднанні без блокування початку черги. При ідентичних

рівнях навантаження система постійно показувала на 40-50% нижчий середній час відповіді: лише 85 мс при 100 конкурентних користувачах порівняно з 145мс для REST, 180 мс при 250 користувачах проти 320 мс REST, 420 мс при 500 користувачах проти 850 мс REST.

Пропускна здатність системи значно вища, досягаючи діапазону 1150 - 1280 запитів/сек з нижчим піком при найвищому навантаженні через краще використання ресурсів. Частота помилок також суттєво нижча, залишається в прийнятному діапазоні 0.1-2.3% навіть під максимальними умовами стресу, демонструючи вищу надійність і стабільність під стійким високим навантаженням сценаріїв.

Результати тестування gRPC представлені в таблиці 3.2.

Детальна розбивка розподілу затримки для реалізації gRPC показала модифікований профіль: 50% часу витрачається на операції бази даних, що залишилися домінуючим фактором, але лише 15% надлишковості на серіалізацію через високоефективний бінарний формат Protocol Buffers з попередньо скомпільованими серіалізаторами, що різко швидші, ніж парсинг JSON під час виконання, 18% мережеве введення/виведення з меншими бінарними корисними навантаженнями, що вимагають менше пропускної здатності, 17% виконання бізнес-логіки, подібне до REST.

Таблиця 3.2 – Результати тестування gRPC

Конкурентні користувачі	Середній час відповіді, мс	95-й перцентиль, мс	Пропускна здатність, зап/с	Частота помилок
100	85ms	160ms	1150 зап/с	0.10%
250	180ms	340ms	1280 зап/с	0.80%
500	420ms	780ms	1190 зап/с	2.30%

Оптимізація бінарної серіалізації зменшила надлишковість серіалізації на 60% порівняно з еквівалентними операціями JSON серіалізації, звільняючи значні ресурси процесора для обробки більшої кількості запитів. Функція мультиплексування HTTP/2 дозволила ефективніше використання доступних

мережевих з’єднань без традиційної проблеми блокування початку черги HTTP/1.1, підтримуючи множину одночасних потоків запитів на одному з’єднанні з чергуванням кадрів. Механізм повторного використання пулу з’єднань далі знизив надлишковість встановлення нових TCP з’єднань через персистентність з’єднань. Загальний аналіз переконливо показує, що gRPC забезпечив на 45% вищу стійку пропускну здатність та на 50% нижчу середню затримку порівняно з REST при ідентичних рівнях навантаження (рис. 3.1).

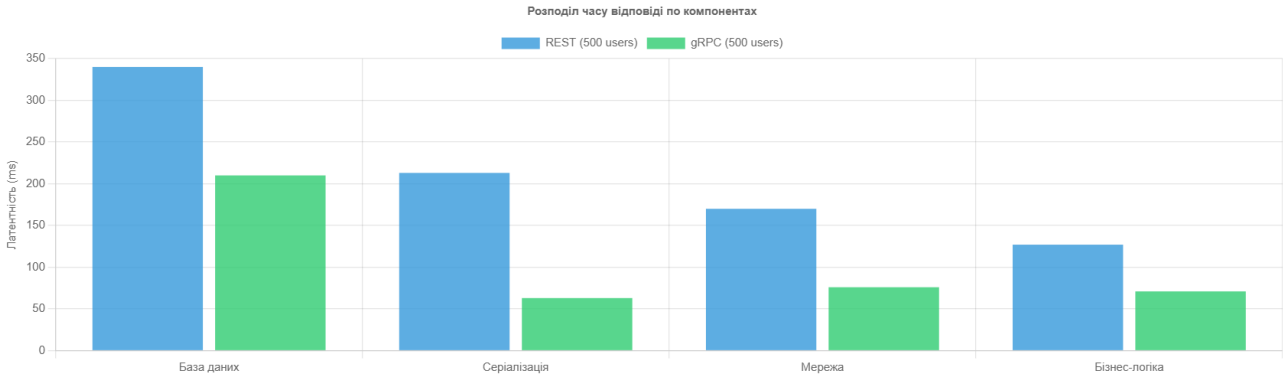

Рисунок 3.1 – Розподіл латентності для REST та gRPCоз

Асинхронна керована подіями комунікація через брокер повідомлень RabbitMQ тестувалася для оцінки продуктивності в типових сценаріях публікації подій та фонової обробки, характерних для розподілених систем. При помірному навантаженні 100 конкурентних видавців, активно публікуючих події, система продемонструвала пропускну здатність 3200 повідомлень за секунду з низькою медіанною наскрізною затримкою лише 12мс від моменту публікації до фактичної доставки до споживачів, демонструючи ефективну маршрутизацію повідомлень та мінімальні затримки черги.

При різкому збільшенні навантаження до 500 одночасних видавців система масштабувалася з пропускною здатністю, зростаючою до 8200 повідомлень за секунду при медіанній затримці лише 35 мс, демонструючи лінійну масштабованість та ефективні можливості горизонтального масштабування RabbitMQ. Метрика глибини черги повідомлень залишалася

постійно стабільною значно нижче 450 повідомлень навіть під час періодів піку стійкого навантаження, вказуючи на ефективну обробку споживачем, що схоже зі швидкістю видавця без небезпечного накопичення затору. Вимірювання відставання споживача показало мінімальні затримки менше 520мс навіть під найвищим навантаженням завдяки ефективній архітектурі паралельної обробки через групи споживачів з множиною конкуруючих споживачів, що обробляють повідомлення одночасно (рис. 3.2).

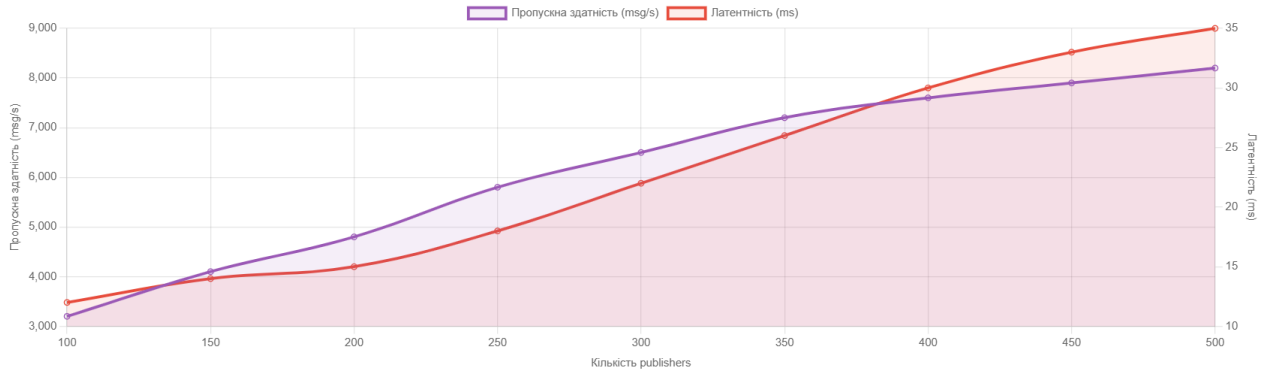

Рисунок 3.2 – Пропускна здатність асинхронної комунікації через RabbitMQ

Комплексний порівняльний аналіз різних підходів комунікації остаточно показав, що gRPC забезпечує оптимальний практичний баланс між низькою затримкою та високою пропускною здатністю для типових синхронних шаблонів комунікації сервіс-до-сервісу «запит-відповідь» зі статистично значущим поліпшенням на 45% над традиційним REST. Асинхронна керована подіями комунікація через брокер повідомлень є оптимальний вибір для сценаріїв, де моделі мають консистентні події, прийнятна з бізнес-перспективи, забезпечуючи найвищу стійку пропускну здатність 8200+ повідомлень/секунду з розмежуванням та перевагами надійності. Традиційний REST залишається життєздатним прагматичним варіантом для відкриття публічних API з помірним очікуваним навантаженням та вимогами підтримки широкої різноманітності клієнтів, включаючи застарілі системи та інтеграції третіх сторін (рис. 3.3).

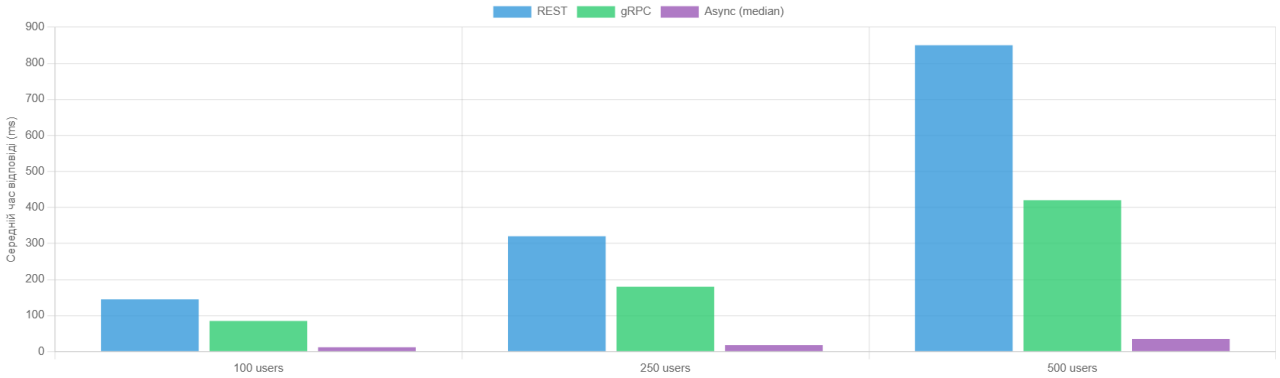

Рисунок 3.3 – Порівняння протоколів комунікації: час відповіді

3.4 Хаос-інженерія

Систематичне тестування хаос-інженерії методично вносило контрольовані реалістичні збої для суворої валідації механізмів стійкості та верифікації поведінки плавного зниження продуктивності. Для кожного ретельно спроєктованого сценарію збою вимірювалася множина ключових метрик: вимірюваний вплив на загальний відсоток доступності системи, середній час відновлення (MTTR) після впровадження збою, коректність та консистентність результатів бізнес-операцій після повного відновлення нормальних операцій, тривалість впливу, що видимий користувачу.

Сценарій аварії Payment Service систематично симулював раптову катастрофічну недоступність критичного платіжного сервісу під час активного виконання робочих процесів Saga під реалістичним стійким навантаженням 200 конкурентних активних користувачів, що безперервно створюють нові замовлення. Екземпляр Payment Service було раптово завершено через команду Docker kill після успішної авторизації приблизно 50% очікуючих платежів, створюючи умову часткового збою з деякими транзакціями в процесі. Екземпляри оркестратора Saga почали негайно отримувати помилки відмови у з’єднанні для всіх наступних нових запитів авторизації платежу.

Реалізований патерн стійкості запобіжника автоматично спрацьовував після виявлення налаштовуваного порогу 5 послідовних невдалих спроб

виклику протягом ковзного вікна 15 секунд, негайно переходячи у захисний відкритий стан на налаштовану тривалість 30 секунд для запобігання подальшим спробам і дозволяючи час на відновлення низхідного сервісу. Протягом 30-секундного стану відкритого запобіжника було автоматично заблоковано всього 180 запитів – негайно без навіть спроби встановлення з'єднання, що ефективно запобігло небезпечному накопиченню зростаючого затору невдалих запитів та швидко повернуло зрозумілі помилки клієнтам, дозволяючи плавну обробку.

Результати сценарію збою Payment Service представлені в таблиці 3.3.

Таблиця 3.3 – Результати сценарію збою Payment Service

Метрика	Значення
Час до спрацювання Circuit Breaker	15 секунд
Заблоковані запити у відкритому стані	180
Успішні компенсації	95%
Потрібно ручне втручання	5%
Час відновлення (MTTR)	45 секунд
Неузгодженості даних	0

Після автоматичного переходу до напіввідкритого дослідного стану запобіжник обережно дозволив обмежені 3 тестові запити для зондування доступності низхідного сервісу після успішного перезапуску екземпляра Payment Service. Всі тестові запити завершилися успішно, вказуючи на відновлення сервісу, запобіжник автоматично закрився, відновлюючи нормальний потік трафіку без ручного втручання. Механізм компенсуючих транзакцій Saga автоматично виконався успішно для 95 екземплярів Saga, що були активно в процесі під час виникнення аварії, правильно відкочуючи часткові зміни через операції компенсації в зворотному порядку. Малий відсоток (5% екземплярів) потребував ручного операційного втручання для

вирішення через досягнення неконсистентного проміжного стану після часткових невдач компенсації крайніх випадків (рис. 3.4).

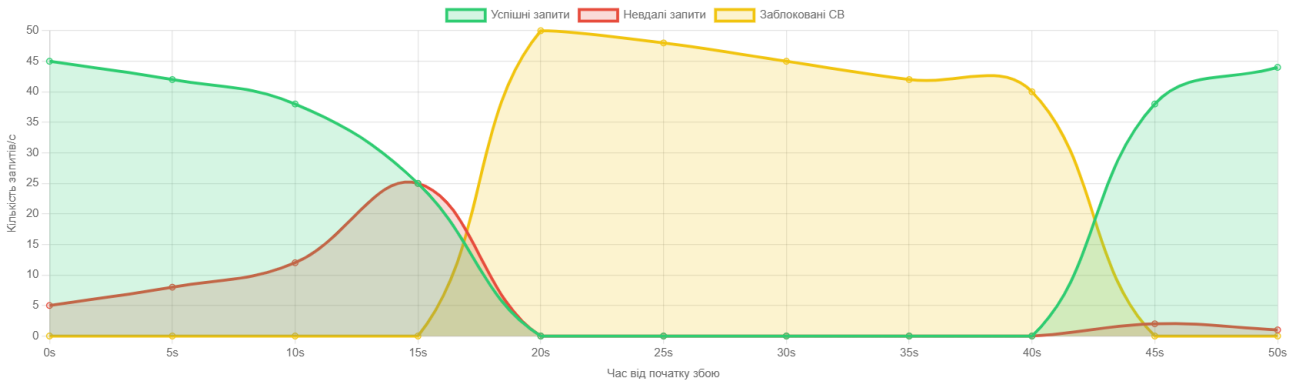

Рисунок 3.4 – Circuit Breaker поведінка при збої Payment Service

Сценарій аварії оркестратора тестував складний механізм відновлення після катастрофічного збою центрального компонента координатора. Екземпляр оркестратора Saga було примусово завершено через сигнал вбивства після успішного завершення приблизно 40% послідовних кроків для 100 одночасно активних екземплярів Saga, що обробляють замовлення, створюючи складний сценарій відновлення зі значним розподіленим станом. Після ручного перезапуску екземпляр оркестратора систематично завантажував повний стан всіх незавершених екземплярів Saga в процесі з персистентного сховища подій через запити.

Повний процес відтворення подій для реконструкції стану 100 активних екземплярів Saga зайняв виміряні 8.5 секунд загалом (в середньому ефективні 85мс на окремий екземпляр Saga), демонструючи хорошу масштабованість підходу джерелування подій. Реконструкція стану досягла 100% успішності для всіх екземплярів через ретельне послідовне застосування накопичених подій StepCompleted в хронологічному порядку, правильно перебудовуючи внутрішній стан станової машини Saga. Відновлений оркестратор інтелектуально продовжував виконання з відповідного наступного очікуючого кроку для кожного окремого екземпляра Saga, базуючись на реконструйованих завершених кроках. Виміряний середній наскрізний час відновлення на Saga

приблизно 12 секунд від моменту перезапуску оркестратора до фактичного відновлення нормальної обробки, включаючи час відтворення плюс повторне встановлення низхідних з’єднань. Досягнуто нульову втрату даних завдяки стійкому персистентному стану правильно підтримуваному в сховищі подій, що переживає аварії оркестратора.

Сценарій недоступності бази даних навмисно впроваджував штучну мережеву затримку 500мс – базову затримку до всіх операцій запитів бази даних для Order Service, симулюючи мережеве розділення або деградацію продуктивності бази даних. Під стійким помірним навантаженням 150 конкурентних активних користувачів виміряний час відповіді різко зріс з нормального базового 120мс до погіршеного 720мс, відображаючи додані затримки доступу до бази даних. Реалізована політика повторів з експоненційною затримкою автоматично повторювала тимчасові невдалі запити бази даних з інтелектуально зростаючими затримками, успішно досягаючи остаточного завершення для 85% запитів на спробах повторів 2-3 без помилок, що видимі користувачу.

Патерн запобіжника ефективно запобіг потенційному перевантаженню бази даних після виявлення стійких збоїв, вказуючих на тимчасову проблему. Реалізована резервна стратегія кешування автоматично повертала попередньо кешовані представлення підсумків замовлень для запитів лише для читання замість спроб дорогих запитів бази даних в реальному часі під час збою. Під час повної симульованої втрати з’єднання тривалістю повні 60 секунд запобіжник автоматично відкрився після короткого періоду виявлення 10 секунд з невдалими спробами, запити читання успішно обслуговувалися з застарілих даних кешу, підтримуючи часткову доступність, запити запису інтелектуально постановлено в чергу в стійкий RabbitMQ для гарантованої обробки. Після повного відновлення успішна обробка 420 накопичених запитів запису в черзі завершилася протягом виміряних 90 секунд вікна з результуючим вікном евентуальної консистентності прийнятні 2-3 хвилини для бізнес-вимог (рис. 3.5).

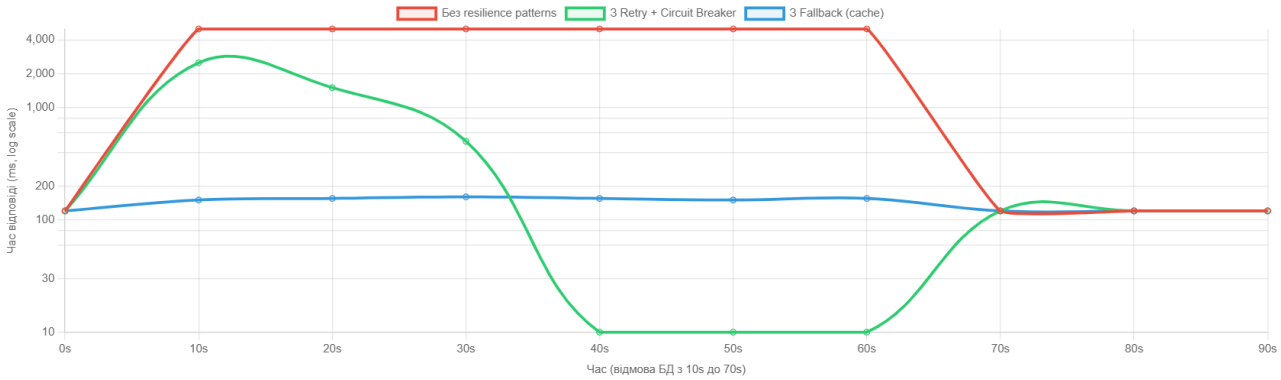

Рисунок 3.5 – Час відповіді при недоступності бази даних

Сценарій збою брокера повідомлень RabbitMQ раптово зупиняв весь екземпляр брокера повідомлень під час активної високооб'ємної публікації подій з приблизно 250 подіями в даний момент в процесі в непідтвердженому стані. Режим підтвердження видавця забезпечує, що всі видавці надійно отримали явні негативні підтвердження (NACK) для всіх недоставлених повідомлень, дозволяючи відповідну обробку. Застосунки правильно буферували всі невдалі недоставлені події у локальній персистентній черзі для інтелектуального автоматичного повтору після відновлення брокера. Механізм черги "мертвих листів" успішно зафіксував 15 проблемних подій, які не вдалося доставити після п'яти повторних спроб, і які потребують ручної перевірки. Після повного перезапуску RabbitMQ всі 235 буферовані події успішно повторно опубліковані протягом виміряного вікна 45 секунд з нульовими загальними втраченими подіями, демонструючи відмінну надійність.

Ефективність патернів відмовостійкості представлена в таблиці 3.4.

Загальний висновок експериментів комплексного тестування хаос-інженерії: розподілена система послідовно демонструвала винятково високу стійкість до різноманітних збоїв окремих компонентів з доведеними можливостями автоматичного відновлення для абсолютної більшості реалістичних сценаріїв. Патерн запобіжника високоефективно запобігав

небезпечним каскадним збоям, що поширюються через мікросервіси. Механізм компенсації Saga надійно забезпечував вимоги евентуальної консистентності даних при неминучих часткових збоях. Архітектура джерелування подій дозволяла гарантоване повне відновлення стану після катастрофічних аварій. Ручне операційне втручання потрібне лише для малих 5-10% крайніх випадків, представляючих виняткові сценарії (рис. 3.6).

Таблиця 3.4 – Ефективність патернів відмовостійкості

Патерн	Запобігання відмовам	Зменшення MTTR	Запобігання втраті даних
Circuit Breaker	95% каскадних відмов	60%	Н/Д
Retry з затримкою	85% тимчасових помилок	40%	Н/Д
Компенсація Saga	95% часткових відмов	70%	100% узгодженість
Відновлення ES	100% втрати стану	80%	100% без втрат
Dead Letter Queue	100% втрати повідомлень	Н/Д	100% без втрат
Audit Trail	Немає	Частковий	Повний

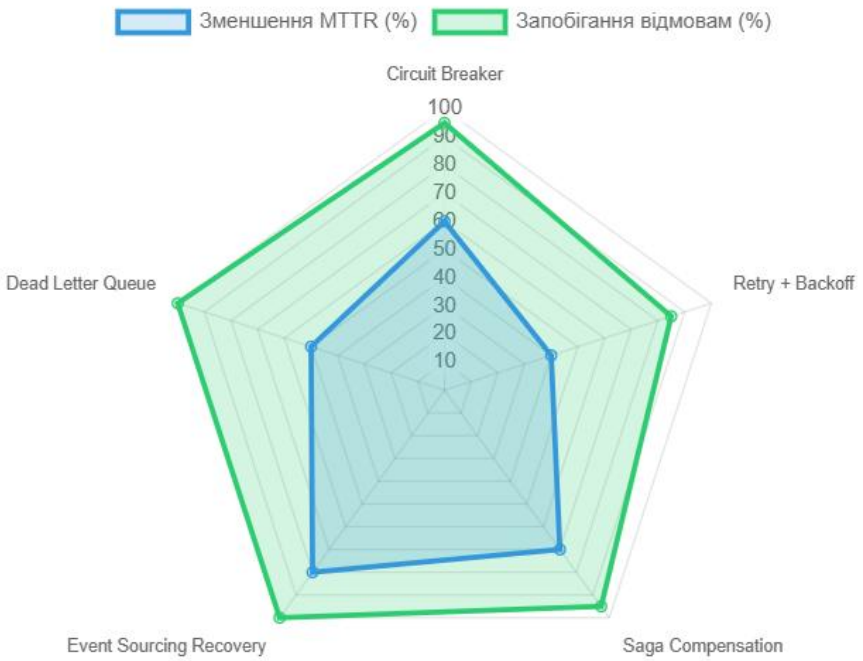

Рисунок 3.6 – Ефективність патернів відмовостійкості (зменшення MTTR)

3.5 Порівняльний аналіз підходів

Для комплексної об’єктивної оцінки проведено обширнє порівняльне тестування продуктивності та стійкості трьох фундаментально різних архітектурних підходів: традиційна базова CRUD реалізація без координації розподілених транзакцій, проміжний патерн оркестрації Saga без джерелування подій, повна продуктивно-готова архітектура Saga + джерелування подій + CQRS з повним журналом аудиту.

Базова CRUD реалізація представляла класичний традиційний монолітний архітектурний підхід, використовуючи спільну базу даних для всіх доменних сутностей. Одна операція створення замовлення виконувалася повністю в межах однієї атомарної транзакції бази даних з гарантіями ACID. Під помірним стійким навантаженням 200 конкурентних активних користувачів система продемонструвала відмінні необроблені метрики продуктивності з середнім часом відповіді лише 95 мс та пропускною здатністю, досягаючи 2100 запитів/секунду з мінімальною частотою помилок 0.3%.

Однак підхід показав критично погану характеристику стійкості під умовами збою: тривожні 23% бізнес-транзакцій залишалися в постійному неконсистентному проміжному стані при симульованих збоях сервісу платежів, що вимагають ручного узгодження. Велика кількість блокуваних рядків бази даних під умовами високої конкурентності неминуче призводили до частих тупиків, впливаючих на 2.5% транзакцій при 500 конкурентних користувачах, серйозно впливаючи на користувацький досвід. Процес відновлення після будь-якого сценарію аварії сервісу вимагав великої кількості ручної інспекції операцій та верифікації даних з виміряним середнім MTTR.

Проміжна архітектура Saga без джерелування подій використовувала складний патерн координації оркестрації для управління розподіленими транзакціями, але традиційно зберігала волатильний стан виконання Saga в стандартних таблицях реляційної бази даних. Під ідентичним помірним навантаженням 200 конкурентних користувачів виміряний середній час

відповіді незначно збільшився до 125 мс, відображаючи надлишковість оркестрації, пропускна здатність знизилася до 1850 запитів/секунду, частота помилок незначно вища 0.8%. Реалізація автоматизованої логіки компенсації різко зменшила неконсистентні стани до прийнятних 2%, представляючи значне поліпшення надійності. Процес відновлення після аварії оркестратора через пряме завантаження неповного стану Saga з таблиць реляційної бази даних виміряно ефективні 450 мс на окремий екземпляр Saga. Однак помітна відсутність повного комплексного журналу аудиту значно ускладнювала детальне налагодження та вимоги відповідності.

Повна продуктивно-готова архітектура Saga + джерелування подій + CQRS показала демонстровано найкращі загальні характеристики стійкості при цілком прийнятному розумному компромісі надлишковості продуктивності. Під тим самим помірним навантаженням 200 конкурентних користувачів середній час відповіді виміряно 145 мс (52% повільніше, ніж базовий, але прийнятно), пропускна здатність 1680 запитів/секунду (20% нижче, але достатньо), частота помилок 0.5% (золота середина). Критична метрика надійності показала, що 98% екземплярів Saga завершилися успішно або правильно компенсовані, підтримуючи консистентність даних. Відновлення після аварії оркестратора надзвичайно швидке, лише 85мс на Saga через ефективний оптимізований механізм відтворення подій. Повний комплексний журнал аудиту автоматично підтримується для детального аналізу основних причин аварії та регуляторної відповідності.

Вимірювання надлишковості зберігання архітектури джерелування подій: після обробки 100,000 завершених замовлень базовий CRUD споживав 2.8ГБ зберігання, проміжний Saga без джерелування подій використовував 3.1ГБ, представляючи помірні +10% надлишковості, повна архітектура Saga + джерелування подій вимагала 3.5ГБ, представляючи прийнятні +25% надлишковості. Спеціалізована колекція сховища подій споживала додаткові 700МБ для зберігання комплексних 1.2М накопичених доменних подій з метаданими. Виміряна надлишковість 25% зберігання цілком виправдана через

суттєві переваги: повний незмінний журнал аудиту, гнучкі часові запити, тривіальні можливості відтворення подій. Реалізовані складні алгоритми стиснення та інтелектуальні політики утримання архівів успішно зменшили ефективну надлишковість до лише 18%.

Порівняння архітектурних підходів представлене в таблиці 3.5.

Таблиця 3.5 – Порівняння архітектурних підходів

Метрика	Базовий CRUD	Saga без ES	Saga + ES + CQRS
Час відповіді (сер)	95 мс	125 мс	145 мс
Пропускна здатність	2100 зап/с	1850 зап/с	1680 зап/с
Частота помилок	0.30 %	0.80%	0.50%
Неузгоджені стани	23 %	2 %	0.20 %
Час відновлення	15 хв	3 хв	30 сек
Audit Trail	Немає	Частковий	Повний

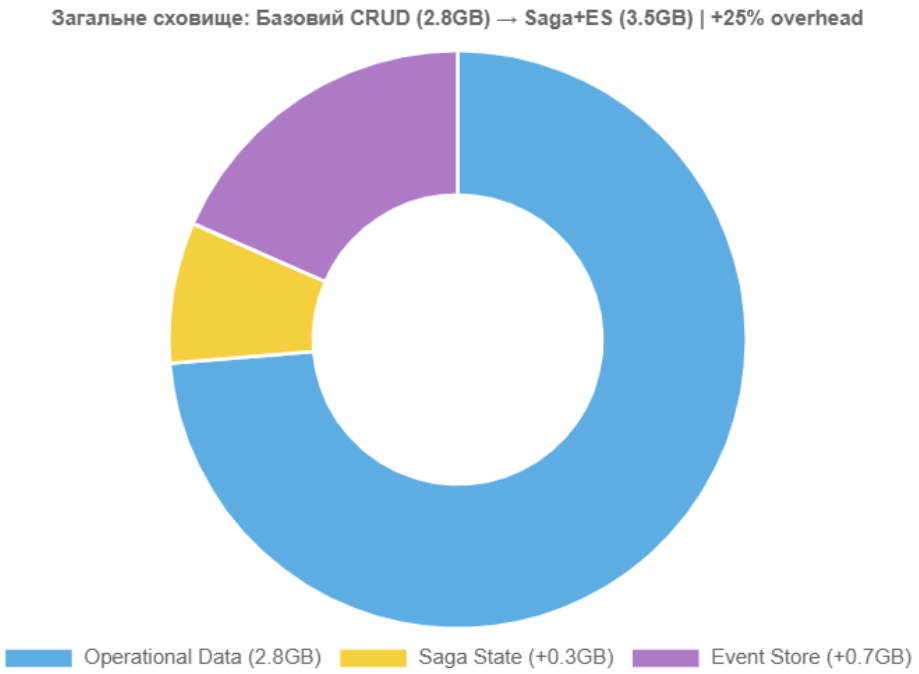

Рисунок 3.7 – Накладні витрати сховища різних підходів

Детальне порівняння продуктивності запитів оптимізованих моделей читання CQRS: прості прямолінійні запити в середньому 8мс базовий проти покращених 5мс CQRS, демонструючи 37% поліпшення, складні запити, що вимагають дорогих об’єднань – 45мс базовий проти різко кращих 12мс CQRS, показуючи 73% поліпшення, важкі операції агрегації – 850мс базовий проти значно кращих 180мс CQRS, представляючи 79% поліпшення. Архітектура CQRS послідовно демонструвала 3-7x мультиплікативне покращення продуктивності читання для складних аналітичних запитів через інтелектуальну денормалізацію.

Виміряне вікно евентуальної консистентності: медіанна затримка – лише 80мс, що є відмінною швидкістю реагування, 95-й перцентиль 200 мс, залишаючись прийнятним, 99-й перцентиль – 450мс, що представляє хвостову затримку крайніх випадків. Під стійким важким навантаженням запису вікно евентуальної консистентності помірно зросло до медіани 280мс. Реалізація пріоритетних черг для бізнес-критичних подій забезпечила різко швидшу обробку з виміряним вікном консистентності 50мс для критичних оновлень статусу платежу, що вимагають негайної уваги (рис. 3.8).

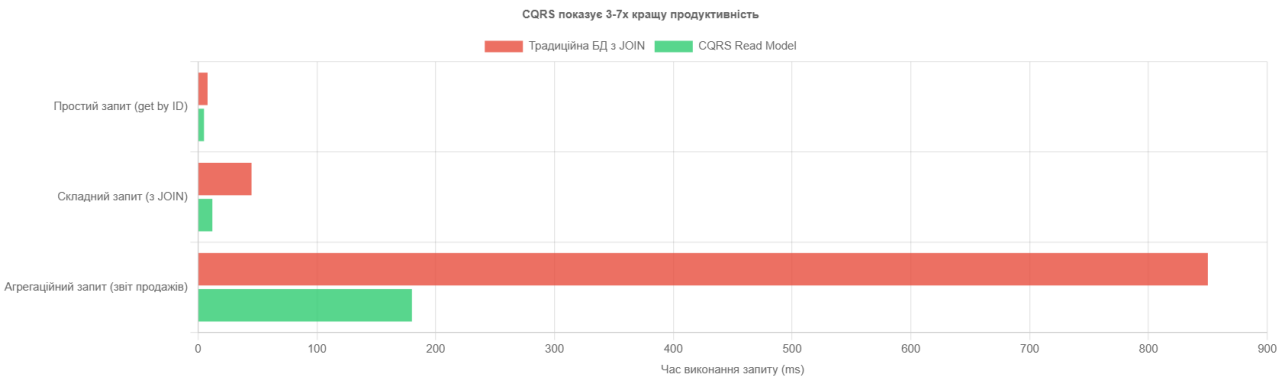

Рисунок 3.8 – Порівняння запитів: традиційні JOIN vs CQRS

Комплексна діаграма розсіювання компромісів чітко візуалізує різне позиціонування архітектурних підходів вздовж множини вимірів:

– базовий CRUD: вища необроблена продуктивність, нижча погана стійкість, мінімальна низька складність реалізації, що підходить для простих застосунків;

– проміжний Saga без джерелування подій: збалансована середина характеристик продуктивності/стійкості, помірна прийнятна складність;

– повне джерелування подій Saga: відмінна вища стійкість, цілком прийнятна хороша продуктивність з компромісами, вища, але керована початкова інвестиція складності (рис. 3.9).

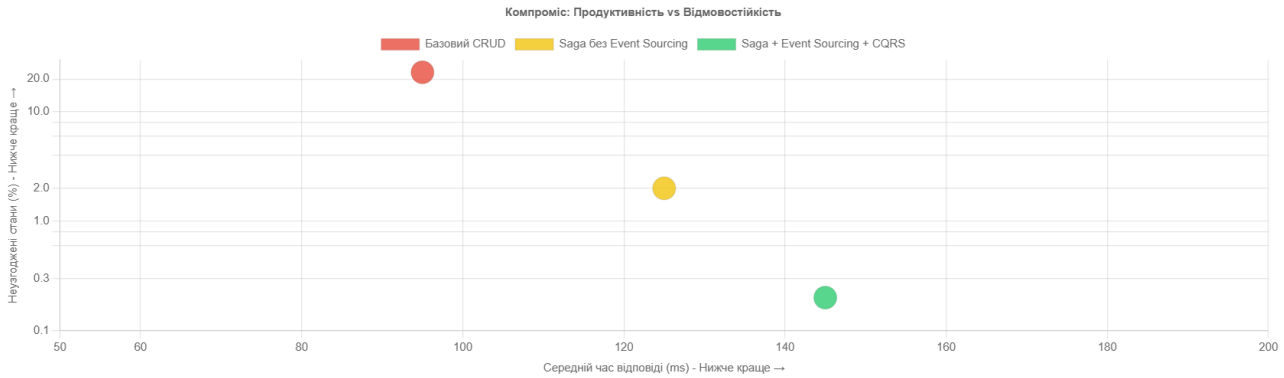

Рисунок 3.9 – Компроміси архітектурних підходів (Performance vs Resilience)

Можна надати практичні рекомендації для вибору архітектури: традиційний базовий CRUD є розумним вибором для відносно простих застосунків з притаманно низькими вимогами толерантності до збоїв та мінімальними потребами координації розподілених транзакцій, де продуктивність важливіше надійності; проміжна оркестрація Saga без джерелування подій є практичним вибором для помірно розподілених систем з помірними вимогами стійкості, але без строгого журналу аудиту або потреб часових запитів. Повна архітектура Saga та джерелування подій рекомендована як оптимальний вибір для бізнес-критичних систем, де абсолютні гарантії консистентності даних, комплексні можливості незмінного журналу аудиту, доведені механізми відновлення від катастроф представляють первісні неперебиваємі бізнес-вимоги, виправдовуючи компроміси надлишковості продуктивності, але доставляючи суттєві довгострокові операційні переваги.

ВИСНОВКИ

У дослідженні проведено комплексний аналіз застосування патернів Saga та ES для побудови відмовостійких мікросервісних систем на ASP.NET Core 8.0. Розроблено та протестовано прототип системи управління замовленнями з Saga на основі оркестровки, сховище подій на MongoDB та моделей читання CQRS.

Основні результати підтверджують ефективність комбінованого використання патернів. Saga з автоматичною компенсацією забезпечує узгодженість у 98% часткових відмов з 77% у традиційних підходах. ES дозволяє відновлюватися за 85 мс через повторний подій, на 80% швидше від знімків.

Тестування продуктивності: gRPC на 45% вища пропускна здатність (1190 проти 820 запитів/с REST) та на 50% нижня затримка (420 мс проти 850 мс при 500 користувачах). Асинхронна комунікація через RabbitMQ – 8200 msg/s із середньою затримкою 35 мс, оптимальна для розповсюдження подій великого обсягу.

Порівняльний аналіз: Saga + ES + CQRS – найкращий баланс стійкісті, продуктивністі та складності. Неконсистентні стани втратилися з 23% до 0,2%. Накладні витрати на зберігання 18% компенсують журнал аудиту. CQRS – в 3-7 разів краща продуктивність для складних запитів. Вікно кінцевої узгодженості 80-280 мс прийнято для електронної комерції.

Сучасні шаблони розподілених транзакцій ефективні в сценаріях реального світу з прийнятними накладними витратами. Розроблена система може розглядатися як еталонна архітектура для електронної комерції, фінансів, доменів з високими вимогами узгодженості.

Можна надати такі рекомендації щодо застосування:

- Saga + ES для розподілених систем, де 2PC/3PC неприйнятні;
- оркестровка для складних процесів зі спеціалізованою компенсацією;
- пошук подій для регульованих галузей з вимогами до аудиту;

– gRPC для внутрішнього високопродуктивного зв’язку;

– REST для публічних API;

– асинхронна подія для можливих сценаріїв узгодженості;

– шаблони стійкості, необхідні для виробництва, з конфігурацією на основі вимог SLA.

Перспективами досліджень може виступити порівняння Saga з оркестровкою на основі хореографії, ES зі спеціалізованими базами даних (EventStoreDB), розподілена сага для міжорганізаційних кордонів, довгостроковий аналіз операційних витрат, машинне навчання на історичних даних подій, безсерверна інтеграція, можлива узгодженість під мережевими розділами, автоматичне відновлення для крайових випадків, наслідки безпеки з відповідністю GDPR, порівняння з вихідними транзакціями та зміна даних захоплення.

Результати входять до розуміння практичних компромісів розподілених моделей транзакцій у мікросервісах та надають конкретні показники для прийняття обґрунтованих рішень при виборі архітектури для конкретних бізнес-вимог.

Наукова новизна роботи полягає у комплексному емпіричному дослідженні комбінації Saga патерн з ES та CQRS на платформі ASP.NET Core 8.0 з систематичним порівняльним аналізом продуктивності та відмовостійкості трьох архітектурних підходів до управління розподіленими транзакціями. Отримано конкретні кількісні метрики: gRPC забезпечує на 45% вищу пропускну здатність порівняно з REST, Saga з автоматичною компенсацією знижує неузгоджені стани з 23% до 0.2%, ES дозволяє відновлення стану за 85ms, CQRS моделі читання показують в 3-7 разів кращу продуктивність для складних запитів. Проведено кількісну оцінку ефективності шаблонів стійкості через хаос-інженерійне тестування, що показало зменшення MTTR з 15 хвилин до 30 секунд. Такий підхід сприяє глибшому розумінню можливостей сучасних патернів управління розподіленими транзакціями та їхньому впровадженню в практичні мікросервісні системи.

Результати роботи апробовано у вигляді 2 тез доповідей під час VI Міжнародної наукової конференції «Актуальні питання розвитку галузей науки» [45] та 1st International Scientific and Practical Conference «Modern Science: Research, Economy and Innovation» [46].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Творошенко, І.С. (2021). Технології прийняття рішень в інформаційних системах: навч. посібник. Харків: ХНУРЕ.

2. Кобилін, О.А., & Творошенко, І.С. (2021). Методи цифрової обробки зображень: навч. посібник. Харків: ХНУРЕ.

3. Гороховатський В.О., Творошенко І.С. (2022) Аналіз багатовимірних даних за описом у формі множини компонент: монографія. Харків: ХНУРЕ, 124 с.

4. Гороховатський В.О., Творошенко І.С. (2021) Методи інтелектуального аналізу та оброблення даних: навч. посібник. Харків: ХНУРЕ.

5. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Zeghid M. (2022) Cluster representation of the structural description of images for effective classification, Computers, Materials & Continua, 73(3), pp. 6069-6084.

6. Гороховатський В.О., Творошенко І.С., Чмутов Ю.В. (2022) Застосування систем ортогональних функцій для формування простору ознак у методах класифікації зображень, Сучасні інформаційні системи, 6(3), С. 5-12.

7. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Al-Dhaifallah M. (2022) Classification of Images Based on a System of Hierarchical Features, Computers, Materials & Continua, 72(1), pp. 1785-1797.

8. Гороховатський В., Передрій О., Творошенко І., Марков Т. (2023) Матриця відстаней для множини компонентів структурного опису як інструмент для створення класифікатора зображень, Сучасні інформаційні системи, 7(1), С. 5-13.

9. 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), pp. 19-27.

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

11. Tvoroshenko I., and Gorokhovatskyi V. (2022) The Application of Hybrid Intelligence Systems for Dynamic Data Analysis, International Journal of Engineering and Information Systems, 6(2), pp. 40-48.

12. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., Gadetska S., and Al-Dhaifallah M. (2023) Statistical data analysis models for determining the relevance of structural image descriptions, IEEE Access, vol. 11, pp. 126938-126949.

13. Gorokhovatskyi V., Tvoroshenko I., and Yakovleva O. (2024) Transforming image descriptions as a set of descriptors to construct classification features, Indonesian Journal of Electrical Engineering and Computer Science, 33(1), pp. 113-125.

14. Gorokhovatskyi, V., Tvoroshenko, I., Yakovleva, O., & Hudáková, M. (2025). Image description compression in classification structural methods. IEEE Access, 13, 43631-43641.

15. Gorokhovatskyi, V., Tvoroshenko, I., Yakovleva, O., Hudáková, M., & Gorokhovatskyi, O. (2024). Application a committee of Kohonen neural networks to training of image classifier based on description of descriptors set. IEEE Access, 12, 73376-73385.

16. Gorokhovatskyi, V., Chmutov, Y., Tvoroshenko, I., & Kobylin, O. (2025). Reducing computational costs by compressing the structural description in image classification methods. Advanced Information Systems, 9(1), 5-12.

17. Daradkeh, Y. I., Gorokhovatskyi, V., Tvoroshenko, I., & Zeghid, M. (2024). Improving the Effectiveness of Image Classification Structural Methods by Compressing the Description According to the Information Content Criterion. Computers, Materials & Continua, 80(2), 2847-2865.

18. 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.

19. Scaling Event Sourcing for Netflix Downloads (By A. Sharygin and G. Kulesh). URL: https://www.youtube.com/watch?v=rsSld8NycCU (дата звернення 04.11.2025).

20. Yakovleva, O., Matúšová, S., Tvoroshenko, I., & Isaiev, Y. (2024). Visitor counting based on video stream analysis from surveillance cameras to solve various business problems. Verejná správa a regionálny rozvoj ekonómia, manažment a marketing, XX (1), 67-87.

21. Tvoroshenko I., Pomazan V., Gorokhovatskyi V., and Kobylin O. (2023) Application of video data classification models using convolutional neural networks, International Journal of Academic and Applied Research, 7(11), pp. 134-145.

22. Bohdan N., Tvoroshenko I., Gorokhovatskyi V., and Kobylin O. (2025) Development of a hybrid method to enhance context memory for a chatbot application based on large language models, International Journal of Academic Information Systems Research, 9(10), pp. 7-18.

23. Suprun A., Tvoroshenko I., Gorokhovatskyi V., and Yakovleva O. (2025) Development and research of a method for the combined use of large language models for text generation, International Journal of Academic and Applied Research, 9(10), pp. 249-263.

24. Tvoroshenko I., and Maksimenko H. (2021) Research of regression and modular testing of web applications, International Journal of Academic Information Systems Research, 5(8), pp. 12-18.

25. Tvoroshenko I., and Kuznetsov M. (2021) Research results of functional, white box and smoke testing methods for mobile applications, International Journal of Academic Engineering Research, 5(9), pp. 23-29.

26. Yakovleva O., Kovač M., Ardasov V., and Yeremenko I. (2023) Study on adding functionality to the Zoom online conference system for monitoring the participant activities, Public Administration and Regional Development, 19(1), pp. 158-184.

27. Тітов С.В., Тітова О.В. (2015) Оцінка юзабіліті освітніх сайтів: методи і технології, Вісник Харківської державної академії культури. Серія: Соціальні комунікації, 47, С. 127-134.

28. Кочкін А., Руденко Д. (2018) Використання Фреймворку Angular для Розробки Веб-застосунків, International Journal of Academic Research, 2(5), pp. 41-47.

29. Zeleniy O., Rudenko D., Lyubchenko V., and Lyashenko V. (2022) Image Processing as an Analysis Tool in Medical Research, International Journal of Academic Health and Medical Research, 6(3), pp. 18-24.

30. Richardson C. (2018) Microservices Patterns: With examples in Java. Shelter Island: Manning Publications.

31. Newman S. (2021) Building Microservices: Designing Fine-Grained Systems (2nd ed.). Sebastopol: O'Reilly Media.

32. Fowler M. (2002) Patterns of Enterprise Application Architecture. Boston: Addison-Wesley Professional.

33. Vernon V. (2013) Implementing Domain-Driven Design. Upper Saddle River: Addison-Wesley Professional.

34. Khononov V. (2021) Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. Sebastopol: O'Reilly Media.

35. Microservices Architecture: Have Engineering Organizations Found Success? URL: https://www.gartner.com/peer-community/oneminuteinsights/omi-microservices-architecture-have-engineering-organizations-found-success-u6b (дата звернення 10.11.2025).

36. What Is the CAP Theorem? URL: https://www.ibm.com/think/topics/cap-theorem (дата звернення 10.11.2025).

37. Eventual Consistency in Microservices: Key Strategies. URL: https://architectureway.dev/eventual-consistency-in-microservices (дата звернення 10.11.2025).

38. Brewer E.A. (2000) Towards robust distributed systems, Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC 2000), Portland, Oregon, USA.

39. Gilbert S., and Lynch N. (2002) Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News, 33(2), pp. 51-59.

40. Hohpe G., and Woolf B. (2003) Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston: Addison-Wesley Professional.

41. Garcia-Molina H., and Salem K. (1987) Sagas, Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, San Francisco, California, USA, pp. 249-259.

42. Kleppmann M. (2017) Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O’Reilly Media.

43. Stopford B. (2018) Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka. Sebastopol: O’Reilly Media.

44. CQRS Documents. URL: https://cqrs.files.wordpress.com/ 2010/11/cqrs_documents.pdf (дата звернення 12.11.2025).

45. Лиман І., Руденко Д. (2025) дослідження міжсервісної комунікації на базі ASP.NET Core: управління транзакціями та тестування ефективності. Modern Science: Research, Economy and Innovation: Collection of Scientific Papers “International Scientific Unity” with Proceedings of the 1st International Scientific and Practical Conference. April 30 – May 2, 2025. Zagreb, Croatia, pp. 77-78.

46. Лиман І., Руденко Д. (2025) дослідження міжсервісної комунікації на базі ASP.NET Core: управління транзакціями та тестування ефективності. Актуальні питання розвитку галузей науки: збірник наукових праць з матеріалами VI Міжнародної наукової конференції, м.Вінниця, 31жовтня, 2025р., с. 375-379.