МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Дослідження засобів побудови програмних систем з подійно-орієнтованою архітектурою**

Виконав: студент 2 курсу, групи 8.1214-2іпз
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
Ю.А. Яхно
(ініціали та прізвище)
Керівник: к. ф.-м. н., доцент Г. П. Коломоєць
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел» П.О.Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра _____ електроніки, інформаційних систем та програмного забезпечення

Рівень вищої освіти _____другий (магістерський)_____

Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)

Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Тетяна КРИТСЬКА_
“_01_” __квітня_____2025_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Яхну Юрію Анатолійовичу
(прізвище, ім’я, по батькові)

1.Тема роботи __Дослідження засобів побудови програмних систем з подійно-орієнтованою архітектурою_____

керівник роботи ___Коломоєць Геннадій Павлович, к.ф.-м.н., доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від 28.03. 2025 р. № 361-е__

2. Строк подання студентом кваліфікаційної роботи_____30.11.2025_____

3. Вихідні дані магістерської роботи

- комплект нормативних документів ;
- технічне завдання до роботи.

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

- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми розпізнавання мов та розробка методів її вирішення;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) 18 слайдів презентації.

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.04.2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	22.02 - 10.03.2025	виконано
2	Літературний огляд 1 розділ - Системи обробки повідомлень	14.03 - 31.03.2025	виконано
3	Літературний огляд 2 розділ - Проєктування систем з подійно-орієнтованою архітектурою	15.04 - 01.05.2025	виконано
4	Дослідницька частина 1 розділ - Архітектура та логіка підсистеми оновлення каталогу системи продажів Декатлон Україна.	01.06 - 20.06.2025	виконано
5	Дослідження показників використання ресурсів застосунком в активній та пасивній фазах роботи	01.07 - 14.07.2025	виконано
6	Профілювання роботи застосунку при різних типах конфігураційних налаштувань	20.07 - 10.08.2025	виконано
7	Дослідження методів оптимізації використання ресурсів застосунком при роботі з різною інтенсивністю створення подій Apache Kafka	20.08 - 20.09.2025	виконано
8	Дослідницька частина 2 розділ - Оптимізація підсистеми оновлення каталогу	20.10 - 16.11.2025	виконано
9	Оформлення фінальної версії магістерської роботи	16.11 - 25.11.2025	виконано

Студент _____ Яхно Ю.А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Коломоєць Г.П.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник Ію.А.
(підпис) (прізвище та ініціали

АНОТАЦІЯ

Сторінок: 87

Рисунків: 20

Таблиць: 3

Джерел: 51

Яхно Ю.А. Дослідження засобів побудови програмних систем з подійно-орієнтованою архітектурою : кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник Коломоєць Г.П. Запоріжжя : ЗНУ, 2025. 87 с.

Мета дослідження полягає у вивченні засобів побудови мікросервісних систем з подійно-орієнтованою архітектурою. Об’єктом дослідження є програмні системи з подійно-орієнтованою архітектурою. Предметом дослідження є засоби побудови та оптимізації програмних систем з подійно-орієнтованою архітектурою.

Для дослідження особливостей роботи систем з подійно-орієнтованою архітектурою, виявлення викликів та недоліків, аналізу методів оптимізації було розглянуто застосунок оновлення каталогу товарів підсистеми Декатлон Україна. Практичне значення одержаних результатів дослідження полягає у створенні оновленого технічного рішення для актуалізації каталогу товарів підсистеми Декатлон Україна з покращеною швидкодією обробки повідомлень при оновлені даних з використанням меншої кількості необхідних ресурсів. Напрацьовані методи та отримані результати можуть бути використані при побудові та оптимізації роботи інших інформаційних систем.

Ключові слова: *подійно-орієнтована архітектура, система обміну повідомленнями, Apache Kafka, RabbitMQ, генератор теми, споживач теми.*

SUMMARY

Pages: 87

Figures: 20

Tables: 3

Sources: 51

Yakhno Yu.A. Research on tools for building software systems with event-driven architecture : qualification work of the master of specialty 121 «Software engineering» / Science head Kolomoiets H.P. Zaporizhzhia : ZNU, 2025. 87 p.

The aim of the research is to study the tools for building microservice systems with event-driven architecture. The object of the study is software systems based on event-driven architecture. The subject of the study is the tools for constructing and optimizing software systems with event-driven architecture.

To examine the operational characteristics of the systems with event-driven architecture, identify challenges and drawbacks, and analyze optimization methods, the product catalog update application of Decathlon Ukraine subsystem was reviewed. The practical significance of the obtained research results lies in the development of an updated technical solution for synchronizing the product catalog of Decathlon Ukraine subsystem with improved message processing performance while using fewer required resources. The developed methods and obtained results can be applied to the construction and optimization of other information systems.

Key words: *event-driven architecture, message exchange system, Apache Kafka, RabbitMQ, topic generator, topic consumer.*

ЗМІСТ

ВСТУП..... 8
1 СИСТЕМИ ОБРОБКИ ПОВІДОМЛЕНЬ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ 14
1.1 Проєктування систем на основі мікросервісної архітектури 14
1.2 Організація взаємодії мікросервісів, недоліки взаємодії засобами REST 18
1.3 Класифікація систем обробки повідомлень в подійно-орієнтованій архітектурі..... 24
2 ПРОЄКТУВАННЯ СИСТЕМ З ПОДІЙНО-ОРІЄНТОВАНОЮ АРХІТЕКТУРОЮ..... 31
2.1 Поняття подій при взаємодії сервісів та їх проєктування 32
2.2 Особливості проєктування систем із використанням архітектурного шаблону “подійно-орієнтована архітектура”..... 37
2.3 Методи вирішення викликів при використанні подійно-орієнтованої архітектури..... 44
3 АРХІТЕКТУРА ТА ЛОГІКА ПІДСИСТЕМИ ОНОВЛЕННЯ КАТАЛОГУ СИСТЕМИ ПРОДАЖІВ ДЕКАТЛОН УКРАЇНА 48
3.1 Загальний принцип та схема оновлення каталогу підсистеми Декатлон Україна 49
3.2 Принцип та схема роботи підсистеми оновлення каталогу та взаємодія з зовнішніми джерелами інформації..... 52
3.3 Недоліки поточної логіки роботи підсистеми оновлення каталогу 58
4 ОПТИМІЗАЦІЯ ПІДСИСТЕМИ ОНОВЛЕННЯ КАТАЛОГУ..... 62
4.1 Базовий набір технологій та можливості оптимізації 62
4.2 Зміна конфігурації елементів підсистеми 64
4.3 Оновлений принцип роботи підсистеми актуалізації каталогу 67
4.4 Оптимізація роботи застосунку у режимі пасивної роботи 74
4.5 Тестування оновленої логіки підсистеми оновлення каталогу..... 77

4.6 Висновки з розділу 4 79
ВИСНОВКИ 82
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 83

ВСТУП

Актуальність теми

Мікросервісна архітектура довела свою ефективність при розробці програмного забезпечення. На відміну від монолітних систем, створення самостійних та менших за розміром елементів пришвидшує процес розробки, зменшує залежність від технічної реалізації (кожен мікросервісний застосунок може бути написаний іншою мовою програмування, яка дозволятиме вирішувати поставлену задачу більш ефективно), полегшує масштабованість всієї системи в цілому, дозволяє чітко окреслити зону відповідальності кожної частини тощо.

Переваги мікросервісного підходу в розробці програмного забезпечення у свою чергу породжують виклики, які не є актуальними для монолітних систем. Однією з основних задач є створення системи ефективної комунікації між кожним елементом (мікросервісом) у межах загальної системи. Шаблон “подійно-орієнтована архітектура” за своєю суттю має вирішувати це питання. Проте, даний підхід також має свої складнощі та виклики:

- складність проєктування систем, зокрема обробки помилок і підтримки консистентності даних;
- необхідність ретельного тестування для забезпечення коректності роботи в асинхронному середовищі;
- управління транзакціями для забезпечення спроможності повернути бізнес-сутності до попереднього стану у випадку виникнення помилок та нестандартних винятків;
- розгалужена система обробки подій може вимагати більшої кількості ресурсів протягом значного періоду часу при роботі системи (у порівнянні з синхронною комунікацією в межах одного монолітного елементу, коли кількість необхідних та зарезервованих ресурсів є сталою і прогнозованою);
- будь-який додатковий шар в системі комунікації є фактором ризику втрати даних та може призводити до затримок при обробці повідомлень, що може бути критичним фактором для конкретної бізнес-системи.

Сучасні технології для роботи з подійно-орієнтованою архітектурою пропонують доволі великий вибір рішень, які відрізняються типами розгортання (“хмарні” сервіси на базі конкретних постачальників, такі як Amazon чи Google, або самостійні сервіси, такі як Apache Kafka чи RabbitMQ), методами обробки подій, засобами збереження даних та гарантування доставки даних від генератора до споживача, необхідною кількістю ресурсів та залежністю цих ресурсів від кількості повідомлень в обробці тощо.

Різноманітність підходів, технологій та викликів при роботі з системами подійно-орієнтованої архітектури зумовлює необхідність в комплексному аналізі кожного з рішень для розуміння зони використання, потенційних витрат на розробку, підтримку та масштабування, актуальності для конкретних бізнес-задач.

Мета і завдання дослідження

Мета дослідження полягає у вивченні засобів побудови мікросервісних систем з подійно-орієнтованою архітектурою. Для досягнення поставленої мети необхідно було вирішити такі завдання:

- порівняти наявні технології обробки повідомлень та методів комунікації;
- проаналізувати виклики при організації комунікації між елементами мікросервісної системи;
- проаналізувати поточний застосунок актуалізації каталогу товарів підсистеми Декатлон Україна;
- оптимізувати застосунок для застосування змін в режимі реального часу та зменшення використання ресурсів.

Об’єкт дослідження

Об’єктом дослідження є програмні системи з подійно-орієнтованою архітектурою.

Предмет дослідження

Предметом дослідження є засоби побудови та оптимізації програмних систем з подійно-орієнтованою архітектурою.

Методи дослідження

Для вирішення завдання дослідження були використані наступні методи:

1. Аналіз особливостей побудови мікросервісних програмних систем з подійно-орієнтованою архітектурою.
2. Порівняння та аналіз принципів роботи різних типів систем обробки повідомлень, виділення особливостей методів доставки та збереження даних.
3. Аналіз викликів при організації комунікації між елементами мікросервісної системи, методів рішення в залежності від бізнес-задач та наявних технологій.
4. Аналіз поточного програмного рішення оновлення каталогу товарів підсистеми Декатлон Україна.
5. Експериментування з технічними та конфігураційними методами оптимізації застосунку актуалізації каталогу.
6. Вимірювання впливу оновленої технічної і конфігураційної логіки на швидкодію при обробці повідомлень та на необхідні ресурси при роботі застосунку.
7. Аналіз апробованих результатів оптимізації для розробки оновленої логіки підсистеми оновлення каталогу.

Наукова новизна одержаних результатів

Наукова новизна одержаних результатів дослідження полягає у ґрунтовному аналізі та порівнянні сучасних систем обробки повідомлень з виділенням їх переваг, недоліків, технічних викликів та прикладів практичного застосування.

Практичне значення одержаних результатів

Практичне значення одержаних результатів дослідження полягає у створенні оновленого технічного рішення для актуалізації каталогу товарів підсистеми Декатлон Україна з покращеною швидкодією обробки повідомлень при оновлені даних з використанням меншої кількості необхідних ресурсів. Напрацьовані методи та отримані результати можуть бути використані при побудові та оптимізації роботи інших інформаційних систем.

Апробація одержаних результатів

Результати дослідження були представлені на XVIII університетській науково-практичній конференції студентів, здобувачів PhD, докторантів і молодих вчених Запорізького національного університету “Молода наука-2025” [1] та на II Міжнародній науково-практичній конференції Інженерного навчально-наукового інституту ім. Ю. М. Потебні “Інженерні інновації та розбудова національної економіки” [2], опубліковані у науковому журналі “Information Technology: Computer Science, Software Engineering and Cyber Security” [3].

Глосарій

Мікросервісна архітектура (англ. Microservices architecture) – підхід до розробки програмного забезпечення, в якому компоненти загальної системи (мікросервіси) розгортаються та працюють незалежно від інших та виконують тільки конкретно визначений перелік задач у межах своєї зони відповідальності.

Предметно-орієнтоване проєктування (англ. DDD – Domain driven design) – підхід до моделювання програмного забезпечення на основі бізнес-концепцій, спільної мови між запрограмованими сутностями та бізнес-поняттями, обмеженням програмної частини предметною областю.

Подійно-орієнтована архітектура (англ. EDA – Event-driven architecture) – парадигма програмної архітектури, в якій самостійні

компоненти системи взаємодіють шляхом генерування та споживання подій, що забезпечує слабку зв’язаність між компонентами, високу масштабованість і гнучкість.

Розподілене сховище подій (англ. Distributed event storage) – система зберігання подій для споживання різними сервісами і застосунками у кількох вузлах або локаціях, зазвичай із забезпеченням відмовостійкості та масштабованості.

Система обміну повідомленнями (англ. Message exchange system) – механізм комунікації для обміну повідомленнями між різними компонентами або системами з гарантуванням надійної та масштабованої передачі повідомлень між відправниками та отримувачами.

Apache Kafka – розподілена платформа потокової обробки подій з відкритим кодом, призначена для побудови конвеєрної обробки даних у реальному часі та потокових застосунках.

Генератор теми (англ. Topic producer) – система або сервіс, що генерує та надсилає події до виділеної теми платформи Apache Kafka.

Споживач теми (англ. Topic consumer) – система або сервіс, що підписується на певну тему та обробляє події, які надходять від генераторів платформи Apache Kafka.

Затримка обробки теми (англ. Topic processing lag) – затримка або накопичення необроблених подій в межах теми, що виникає, коли споживач не встигає обробляти повідомлення з такою ж швидкістю, з якою вони генеруються.

RabbitMQ – система-посередник для обміну повідомленнями з відкритим кодом, яка полегшує взаємодію між застосунками або сервісами шляхом керування чергами, передачею та маршрутизацією повідомлень.

Черга повідомлень (англ. Message queue) – механізм комунікації, що зберігає повідомлення в стані очікування до їх обробки споживачами для впорядкованої та асинхронної обробки.

Посередник повідомлень (англ. Message broker) – програмний компонент або проміжне програмне забезпечення, що керує передачею повідомлень між системами або сервісами.

Затримка обробки повідомлень (англ. Message processing latency) – час, необхідний для повної обробки повідомлення після його надсилання від генератора до споживача.

Електронна комерція (англ. E-commerce) – система для підтримки купівлі та продажу товарів або послуг через інтернет.

Оновлення каталогу товарів – процес оновлення або модифікації інформації про товари в онлайн-магазині або платформі електронної комерції.

1 СИСТЕМИ ОБРОБКИ ПОВІДОМЛЕНЬ У МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

1.1 Проєктування систем на основі мікросервісної архітектури

Мікросервісна архітектура не є новим явищем в процесах розробки програмного забезпечення. Проте питання ефективної взаємодії між мікросервісами при збільшенні кількості даних та власне компонентів системи стає дедалі більш гострим [4]. Швидкість споживання інформації користувачами зростає, що в свою чергу змушує пришвидшувати потік даних на програмному рівні, підтримуючи рівень консистентності бізнес-сутностей на всіх рівнях взаємодії. Користувачі очікують миттєвих застосувань своїх змін в усіх наявних інтерфейсних елементах. Кожна затримка або невідповідність очікуваному стану може змусити їх замінити поточну платформу на іншу (що в бізнес-контексті означатиме втрату споживача, тобто фактично втрату майбутнього прибутку). Чим більше елементів (мікросервісів) починають брати участь у бізнес-процесах, тим більшою є ймовірність затримок в актуалізації даних та неконсистентності сутностей [5].

Однією з першочергових задач при побудові комплексної мікросервісної системи є її поділ на логічні частини. Кожен мікросервіс має бути дійсно незалежним від інших для того, щоб його можна було оновлювати, змінювати та розгортати окремо. Для ефективного поділу системи на компоненти потрібно зважати на три концепції [5]:

1. Приховування інформації. Кожен мікросервіс має розкривати назовні лише ту інформацію, яка потрібна іншим мікросервісам для взаємодії з ним. Жоден зовнішній сервіс не має знати про специфіку внутрішньої роботи конкретного компонента. Для всіх інших частин системи кожен мікросервіс має бути “чорною скринькою” з чітко окресленими каналами взаємодії.
2. Зв’язаність зони відповідальності. Кожен мікросервіс має тримати в собі достатню логіку та кодову структуру для виконання своєї основної задачі. Чим

більше кожен компонент є самодостатнім в своїй роботі, тим легше його змінювати, оновлювати, розгортати (та видаляти за необхідності).

3. Міжсервісна взаємозалежність. Ця концепція є логічним продовженням попередньої. Чим менше мікросервіс залежить від інших частин системи, тим більш стабільним та ефективним він є. Спілкування та операційна взаємодія між різними сервісами має бути мінімальною.

Поняття поділу загальної системи на компоненти тісно переплітається з базовою задачею предметно-орієнтованого проєктування. Виділення предметної області, обмеженого контексту та базових бізнес-сутностей є першими і найважчими завданнями для ефективного моделювання та подальшого впровадження системних елементів [6]. Обмежений контекст дозволяє розділити всі бізнес-процеси на функціональні елементи, які в результаті знайдуть своє відображення в програмних мікросервісах. Чим тісніший є зв'язок між виділеними функціональними частинами (майбутніми мікросервісами) та бізнес-процесами, які ця частина має виконувати, тим ефективнішим буде спілкування з представниками бізнесу при програмній розробці, додаванні нових функцій та підтримці наявних [7].

Представники бізнесу (у більшості випадків – користувачі системи) мають говорити з представниками команди розробки одними поняттями, бізнес-сутностями, однією мовою. Поняття єдиної загальної мови (англ. – Ubiquitous language, що більш дослівно перекладається як всепроникна, чи всюдисуща мова) має бути одним з вирішальних при прийнятті рішення про виділення певної функціональної частини в окремий мікросервіс. Особливостями та перевагами використання єдиної мови в межах одного компонента системи є [8]:

- мова бізнес-моделі може бути більш чітко втілена в моделі технічної реалізації, що зменшить ризик створення компонента з зоною відповідальності, яка не відповідає бізнес-задачам;
- в межах одного обмеженого контексту має функціонувати лише один набір понять, які мають бути зрозумілими для експертів предметної області;

- обмежений контекст має включати в себе мінімальну самостійну частину бізнес-задач, які охоплюють поняття єдиної мови;
- єдина мова має своє значення лише в межах конкретного обмеженого контексту, тобто бізнес-сутності окремого компонента не можуть і не мають корелювати з бізнес-сутностями і назвами (хоча вони можуть бути схожі) будь-якого іншого компонента загальної системи;
- розробка єдиної мови може бути тривалим процесом з залученням експертів предметної області та представників команди розробки, проте коректне її створення буде мати переваги при розробці компонента та подальшій взаємодії команди.

Наступним кроком після окреслення єдиної загальної мови для ефективного проєктування елементів мікросервісної архітектури є виділення зон відповідальності, каналів комунікації та внутрішньої логіки. Серед багатьох варіантів моделювання мікросервісів можна виділити концепцію Семи Суттєвих Кроків для Дизайну Сервісів (англ. SEED(S) – Seven essential evolutions of design for services) [9]. Вказані кроки пропонують структурований поступовий процес для проєктування та впровадження мікросервісів на основі бізнес-логіки предметної області. Варто зазначити, що вказаний процес може бути доволі довготривалим, проте перевагою при його використанні є аналіз та перевірка результатів попередніх кроків перед наступними, що в результаті дозволить отримати надійну технічну систему. Концепція пропонує використовувати наступні сім етапів [9]:

1. Ідентифікація дійових осіб. При проєктуванні мікросервіса потрібно чітко розуміти його користувачів: хто саме бути використовувати цей компонент? Цей етап може бути частиною побудови єдиної загальної мови та виділення предметної бізнес-області, тому достовірні результати попередніх підготовчих етапів мають допомогти коректно відповісти на питання на цьому кроці.

2. Розуміння задач дійових осіб. Після ідентифікації користувачів майбутнього мікросервіса потрібно зрозуміти, як вони планують працювати з

ним: що вони очікують від взаємодії з компонентом, які задачі він має виконувати, який результат він має повертати після виконання задач?

3. Дослідження шаблонів взаємодії через діаграми послідовності. Зрозумівши задачі, які виконують користувачі, потрібно зрозуміти конкретні дії та очікувані результати. Використання уніфікованої мови моделювання (англ. UML – Unified modeling language) у вигляді діаграм послідовності [10] дозволить структурувати задачі дійових осіб в усіх моментах взаємодії з майбутнім мікросервісом.

4. Аналіз дій та запитів на основні “Теорії робіт” (англ. JTBD – Jobs to be done). Це фінальний етап дослідження верхнього рівня взаємодії, результатом якого має бути вичерпний список всіх дій та результатів, які має виконувати мікросервіс.

5. Формалізація дій та запитів через специфікації каналів взаємодії. У результаті цього етапу має бути розроблено вичерпний список дій, які може виконати користувач з мікросервісом з повним списком даних для відправки (якщо актуально) та отримання у результаті (якщо актуально).

6. Отримання зворотного зв’язку від користувачів щодо специфікації взаємодії. Створений перелік стандартизованих доступних дій та результатів має виконувати очікувану задачу від користувачів, всі відкриті питання мають обговорюватись для досягнення згоди між представниками бізнесу та командою розробки.

7. Розробка та впровадження мікросервісу. Це фінальний технічний етап, у результаті якого має бути створений мікросервіс згідно узгодженою технологічною базою та специфікацією.

У результаті розробки концепції бізнес-моделі для системного компоненту, узгодження предметної бізнес-області, проєктування та технічного впровадження, мікросервіс має забезпечувати наступні характеристики [5, 11]:

- реалізація на основі та для виконання бізнес-потреб;

- відкриті та зрозумілі канали зовнішньої комунікації та прихована для зовнішніх елементів технічна реалізація;
- самостійний вибір набору використовуваних технологій та управління внутрішніми потоками даних;
- інфраструктурна незалежність та індивідуальне розгортання;
- відкритість до масштабування та подальшої підтримки і потенційного розширення чи зміни функціональності.

1.2 Організація взаємодії мікросервісів, недоліки взаємодії засобами REST

Кожен компонент у межах мікросервісної архітектури не є абсолютно незалежним, хоча незалежність і позиціонується як одна з найвагоміших переваг цього підходу до розробки програмного забезпечення. Мікросервіси на етапі створення (та можливо на початкових етапах розгортання та впровадження) можуть розглядатись окремо, у “вакуумному” середовищі по відношенню до загальної системи. Проте у реальному стані повноцінно робочої системи з багатьох мікросервісів кожен з елементів має комунікувати з іншими, отримувати та надсилати дані, реагувати на зміни станів бізнес-сутностей тощо. Коректне проєктування загальної системи та виділення функціональних частин відповідно до положень, висвітлених у розділі 1.1., дозволяє структурувати всю зовнішню комунікацію, зробити її максимально наближеною до реальної комунікації між бізнес-підрозділами, мати чіткі канали та формалізовані специфікації. Проте жодне проєктування не має на меті позбутись комунікації між мікросервісами в цілому – це не є можливим та доцільним для ефективного покриття всієї бізнес-системи.

Комунікація між мікросервісами у більш загальному уявленні може і має бути наближеною до комунікації між підрозділами бізнесу, на чому наголошує закон Конвея [12]. Проте фактична комунікація на програмному рівні найчастіше має свої обмеження і багато в чому залежить від коректного

проєктування та архітектурного моделювання. Методи взаємодії та комунікації можуть бути реалізовані через використання однієї з трьох структур [4]:

1. На основі бізнес-комунікації. Цей підхід є фактичною реалізацією бізнес-процесів, може швидко змінюватись при реструктуризації всередині компанії, та в залежності від її розміру може еволюційно оновлюватись доволі часто.

2. На основі технічної реалізації. Цей підхід є дуже поширеним в монолітних системах, і саме в такому прикладі він окреслює свої обмеження та недоліки. Застарілі методи реалізації можуть штучно обмежувати або розкривати чутливі дані для великої кількості користувачів, бути перевантаженими або навпаки недостатніми.

3. На основі даних. Цей підхід має бути реалізованим як результат впровадження структури бізнес-комунікації у програмній реалізації. Фактично ця структура має відкривати дані для користувачів там, де ці дані є потрібними для виконання їхніх обов’язків, та приховувати дані, які не є необхідними в певній частині системи.

Взаємодія між мікросервісами є прямим втіленням взаємодії між бізнес-підрозділами, задачі яких ці мікросервіси охоплюють. При цьому важливо розділяти взаємодію та внутрішню комунікацію всередині сервіса та зовнішню комунікацію з іншими мікросервісами. Взаємодія та обробка виконання дій виклик – відповідь є нескінченним процесом під час роботи кожного мікросервіса, тому питання внутрішньої комунікації є важливим та має свої особливості. Потрібно виділити декілька факторів, які дуже відмінні у внутрішній комунікації мікросервіса у порівнянні з зовнішньою взаємодією з іншими компонентами [5]:

- Продуктивність. Внутрішні виклики швидкі, вони можуть бути автоматично оптимізовані мовними інструментами технічної реалізації навіть без додаткових дій зі сторони розробника. Програмні платформи, бібліотеки та середовища в межах внутрішніх викликів намагаються використовувати

лише мінімально необхідну кількість процесів для виконання певної операції. Зовнішні виклики потребують набагато більше ресурсів, вони вимагають використовувати мережеві методи спілкування, а тому є завжди більш повільними та менш ефективними.

- Залежність від інтерфейсів. Будь-яка зміна в структурі оголошення інтерфейсу всередині мікросервісу може бути легко виконана та впроваджена без ризику негативного впливу на зовнішні компоненти. Проте будь-яка зміна в формалізованих специфікаціях каналів взаємодії з зовнішніми користувачами є доволі складним процесом. По-перше, потрібно гарантувати роботу попередньої специфікації. По-друге, потрібно мінімізувати негативні наслідки від одночасної підтримки попереднього та нового методів взаємодії. По-третє, потрібна ефективна міжкомандна комунікація для коректності переходу на використання оновлених структур взаємодії.
- Обробка помилок. Доволі часто в межах одного мікросервіса перевірка на валідність даних може бути мінімальною завдяки використанню вбудованих правил типізації, статичним аналізаторам тощо. При цьому, кожен тип помилки може мати відмінне опрацювання в залежності від ситуації виникнення. Обробка помилок при зовнішній комунікації завжди є більш ризикованою та ресурсозатратною. Враховуючи залежність від інтерфейсів комунікації потрібно завжди стежити за тим, як і які дані мікросервіс отримує від зовнішніх компонентів.

Вибір можливостей міжсервісної комунікації є доволі широким та багато в чому залежить від частоти необхідної взаємодії, типів даних для передачі, залежності від часу відповіді. Можна виділити п’ять основних типів комунікації між мікросервісами, які можна об’єднати в схему, зображену на рисунку 1 [5].

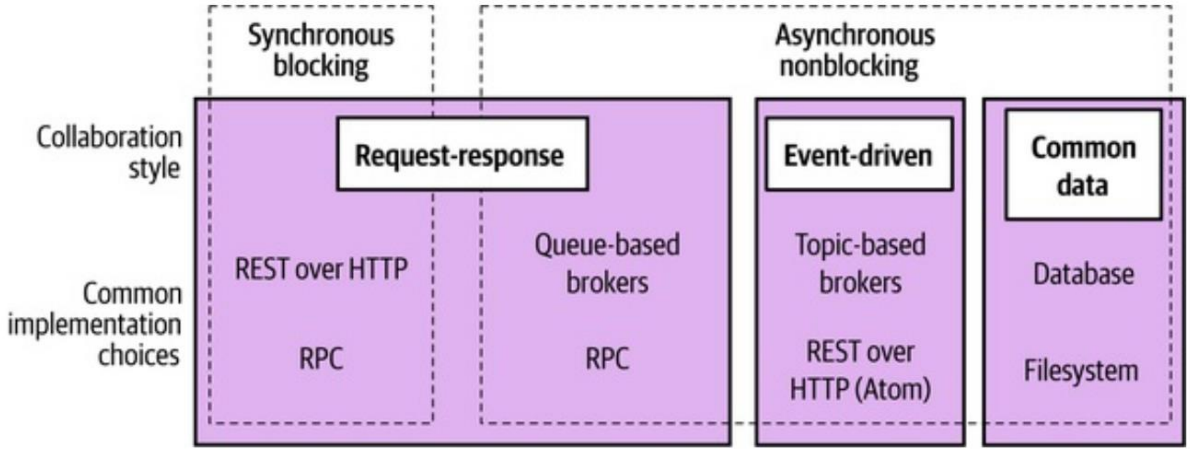


Рисунок 1 *Типи комунікації між мікросервісами* [5]

В загальному випадку всі типи комунікації можна розділити на дві глобальні категорії [5, 13]:

- синхронна комунікація – мікросервіс робить виклик до іншого мікросервіса, очікує та обробляє відповідь; решта процесів блокуються до повної обробки відповіді, тобто є фактично повноцінна залежність від отриманих даних; затримка при управлінні викликом спричиняє затримку в роботі базового мікросервіса;
- асинхронна комунікація – мікросервіс робить виклик до іншого мікросервіса або сховища даних та продовжує свою роботу; відповідь (за її наявності) обробляється при її отриманні, при цьому результат відповіді ніяк не впливає на основну роботу базового мікросервіса.

Варто наголосити, що обидві категорії є актуальними в залежності від ситуації та типів даних для обробки. Наприклад, дія “Аутентифікація користувача” може вимагати синхронної комунікації через те, що результат виконання дії буде фактично впливати на те, яким чином користувач зможе далі взаємодіяти з системою. При цьому, дія “Завантаження статистики використання” може бути більш ефективною при асинхронній комунікації, адже у користувача буде можливість продовжувати взаємодіяти з інтерфейсом та виконувати інші дії в системі, поки статистичні дані підготуються, обробляться та будуть готові для відображення.

В залежності від методів передачі, засобів збереження та обробки даних можна виділити такі типи комунікації [5]:

1. Використання спільного сховища даних. Ідея цього типу полягає в тому, що обидва мікросервіси (генератор даних та споживач даних) насправді не комунікують між собою безпосередньо. Вони можуть навіть не знати про існування одне одного. Вони фактично взаємодіють лише через сховище даних, яке є спільним для обох. При цьому можливе обмеження в правах при взаємодії зі сховищем, коли один мікросервіс може лише записувати дані, інший (або інші) – лише читати. Основною перевагою подібного методу є те, що його можна використовувати при будь-яких інших обмеженнях для взаємодії (наприклад, при неможливості використовувати REST чи RPC). Недолік цього типу викликаний перевагою – споживачі не можуть знати про те, що з'явились нові дані або змінились наявні. Як результат, мусить бути реалізований механізм періодичної перевірки оновлень. Цей тип комунікації може бути лише асинхронним. Найчастіше реалізація підходу відбувається через використання бази даних або файлового сховища.

2. Використання підходу запит-відповідь. Ідея цього типу полягає в тому, що один мікросервіс (споживач даних) надсилає виклик до іншого мікросервісу (генератор даних) та обробляє результат. Це може бути як запит на виконання певної дії, так і запит на отримання якихось даних (або на виконання обидвох дії одночасно). Найчастіше даних підхід є актуальним саме при синхронному використанні, коли результат потрібен саме зараз, та він впливатиме на те, як споживач даних має працювати далі. Проте, існує можливість використання цього підходу для асинхронної комунікації також. Найчастіше реалізація підходу відбувається через використання REST HTTP або RPC викликів.

3. Подійно-орієнтована комунікація. Основна ідея полягає в тому, що взаємодія між мікросервісами при даному підході полягає не в конкретній прямій комунікації чи залежності від результату відповіді. В даному випадку один мікросервіс (генератор даних) створює подію про зміну певного стану

бізнес-об’єкта. Ця подія може буде оброблена іншим мікросервісом або мікросервісами (споживчами), або ні. При цьому, генератор даних про це не знатиме. Його задача – зафіксувати певну зміну, чи буде (і як буде) використовуватись ця інформація для нього значення не має. Даний підхід прибирає будь-яку пряму взаємодію безпосередньо між мікросервісами, що у більшості випадків може мати позитивний результат для швидкодії усіх учасників обробки даних. Цей тип комунікації може бути лише асинхронним. На відміну від асинхронної взаємодії у форматі "запит-відповідь" через брокери повідомлень, яка все ще передбачає очікування відповіді від одержувача, подійно-орієнтований підхід повністю від'єднує відправника і споживачів, що забезпечує вищий рівень незалежності сервісів.

Серед розглянутих типів спільне сховище даних використовується найчастіше через технічну простоту реалізації, проте саме цей підхід розглядається найрідше у контексті взаємодії мікросервісів [5, 9]. В сучасних системах питання вибору методу комунікації зазвичай звужується до використання підходу запит-відповідь або подійно-орієнтованого дизайну. В багатьох випадках перевага надається саме використанню першого варіанту. Він виглядає найпростішим з точки зору розуміння процесу, та є фактично реалізацією взаємодії між мікросервісами як з будь-якими іншими інтерфейсами програмування застосунків – API [9]. Найбільш популярними технологіями для реалізації цього підходу є взаємодія через протокол HTTP з використанням REST чи RPC (gRPC).

Комунікація з використанням HTTP (REST) має суттєві недоліки, які потрібно вирішувати при виборі цього підходу [13, 14]:

- HTTP є синхронним протоколом – навіть у випадку асинхронності обробки результату запиту на стороні клієнта сам виклик блокує процес, поки не буде отримано відповідь;
- HTTP запит обмежений двома учасниками – є єдиний відправник та єдиний отримувач даних; у випадку, якщо дані потрібно надіслати великій

кількості отримувачів, необхідна відповідна кількість атомарних викликів через протокол для кожного отримувача;

- обмеження розміру даних для відправки – в залежності від платформи обмеження на максимальний розмір даних для відправки або отримання через HTTP може бути від 30 до 100 мегабайтів.

Як результат – HTTP може бути обґрунтованим вибором при необхідності синхронної обробки даних або для передачі даних з ефективною мережевою підтримкою (тобто дія запит-відповідь не буде викликати затримок при обробці), проте є неефективним при необхідності надсилати великі масиви даних.

1.3 Класифікація систем обробки повідомлень в подійно-орієнтованій архітектурі

Подійно-орієнтована архітектура вирішує виклики та недоліки, які можуть виникати при комунікації мікросервісів через протокол HTTP (REST). Основними перевагами використання подібного підходу є [15]:

- асинхронність обробки – можливість отримувати та обробляти декілька подій одночасно без необхідності синхронної взаємодії між різними мікросервісами;
- незалежність від розміру даних для обробки, які створюються генератором, завдяки чому швидкість генерації даних не впливає на швидкість обробки даних споживачами;
- збільшення гнучкості та масштабованості кожного окремого мікросервіса та системи в цілому;
- підвищення надійності завдяки можливості повторної обробки подій або відновлення попереднього стану бізнес-сутностей;
- полегшення інтеграції з іншими системами та мікросервісами завдяки асинхронній взаємодії та відсутності прямої комунікації;

- повна незалежність від процесу генерації подій, їх кількості та стану – всі події можуть оброблятись споживачами в будь-який зручний час, у потрібній кількості та зі змінною швидкістю.

Проте, даний підхід також має свої складнощі та виклики [5, 15]:

- проєктування кожного окремого мікросервісу та системи в цілому ускладнюється через необхідність більш гнучкої обробки помилок і підтримки стабільного стану даних при зовнішній комунікації;
- використання асинхронних методів комунікації ускладнює процеси тестування та валідації всіх змін на рівні програмного забезпечення через збільшення кількості перевірок при взаємодії з зовнішніми компонентами;
- необхідність використання транзакцій (та підходу “все або нічого”) на всіх етапах зовнішньої комунікація для підтримки коректного стану внутрішніх бізнес-сутностей;
- розгалужена система обробки подій може вимагати більшої кількості ресурсів протягом невизначеного періоду часу при роботі системи (у порівнянні з періодичною синхронною комунікацією, коли ресурси використовуються лише протягом прямої взаємодії між генератором та споживачем, адже час виконання команд відомий заздалегідь);
- до викликів також можна віднести доволі великий вибір технологій для використання, які можуть мати свої особливості та обмеження в залежності від методів використання.

Наявні системи для роботи з подіями (повідомленнями) можна класифікувати за кількома характеристиками, а саме: за способом розгортання, за функціональними технологіями, за способом організації обробки та збереження даних. Кожна з класифікацій дозволяє обрати потрібний варіант реалізації в залежності від конкретних потреб, наявних ресурсів та задач, які повинні вирішувати мікросервіси при їх комунікації.

За способом розгортання системи обробки повідомлень можна поділити на локальні (використання бібліотек з відкритим кодом для розгортання системи на власних фізичних серверах, наприклад Apache Kafka [17] або

RabbitMQ [16]) та хмарні (можливе використання аналогічних бібліотек з відкритим кодом, які розгортаються на хмарних виділених серверах від постачальників послуг, таких як Google, Azure чи Amazon, або використання спеціалізованих технологій постачальників послуг для роботи з подіями, повідомленнями чи потоками даних [18]).

За функціональними технологіями системи обробки повідомлень можна поділити на:

1. Сервіси черг повідомлень – пропонують найлегшу та найпростішу асинхронну взаємодію між генератором події та споживачем (зазвичай лише одним), де кожна згенерована подія зберігається в черзі до її обробки. Кожне повідомлення надсилається лише до однієї черги, при цьому відсутнє додаткове перенаправлення чи трансформація повідомлень [19]. Всі повідомлення обробляються за схемою “перший прийшов перший вийшов” (англ. FIFO – First in first out), при якій зберігається хронологічна послідовність. Приклади реалізації сервісів [18]: Microsoft Message Queuing (MSMQ), Amazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS), Azure Service Bus, Azure Queue Storage, Google Cloud Tasks. Схематично найпростіший варіант реалізації черг повідомлень зображено на рисунку 2.

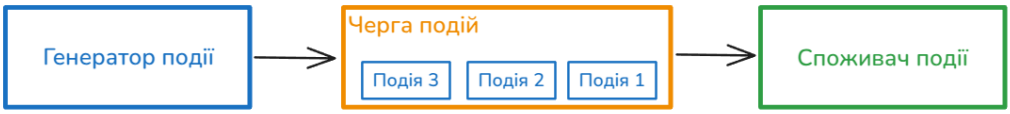


Рисунок 2 *Схема обробки даних сервісів черг повідомлень*

2. Посередники обробки повідомлень – пропонують складнішу логіку проміжної обробки повідомлень між відправником (генератором) та отримувачем (споживачем). Вони можуть мати розгалужену логіку трансформації, фільтрації та перенаправлення повідомлень. Посередники дозволяють створювати події для кількох черг одночасно, для можливості різних споживачів отримувати різні дані [19]. Приклади реалізації сервісів

[16]: ActiveMQ, RabbitMQ, Google Pub/Sub, Amazon Kinesis, Azure Service Bus Messaging. Схематично варіант реалізації системи з використанням посередників обробки повідомлень зображено на рисунку 3.

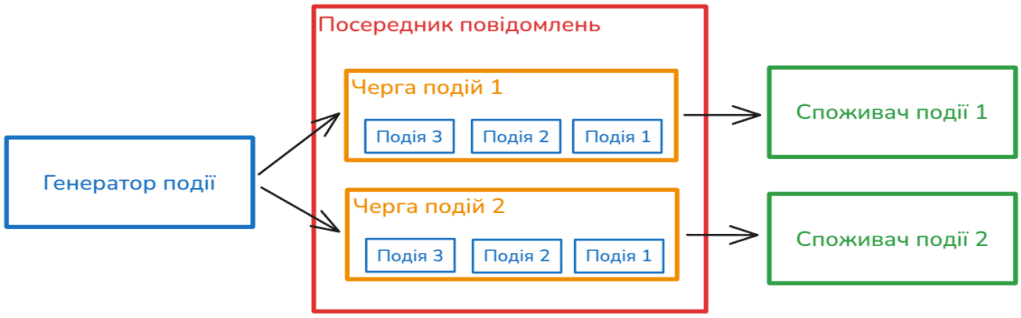


Рисунок 3 *Схема обробки даних через посередників повідомлень*

3. Сервіси потокової обробки даних – дозволяють обробляти найбільшу кількість даних зі складною внутрішньою структурою та можливістю багатьом споживачам приєднуватися до різних потоків та отримувати всі повідомлення в потоці (темі). Цей тип взаємодії підходить для обробки даних у реальному часі, коли постійні зміни стану бізнес-сутностей одного мікросервісу мають безпосередньо впливати на стани бізнес-сутностей іншого. Приклади реалізації сервісів: Apache Kafka, Apache Pulsar, Amazon Kinesis. Схематично варіант реалізації системи з використанням потоків обробки повідомлень зображено на рисунку 4.

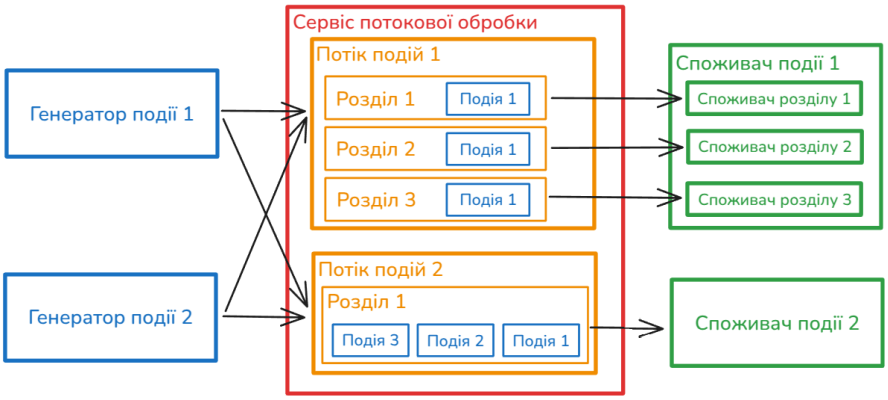


Рисунок 4 *Схема обробки даних через потоки повідомлень*

За способом організації обробки та збереження даних можна виділити основні шаблони (автори монографії [20] загалом виділяють 65 шаблонів, нижче вказані ті, які є актуальними саме для класифікації систем подійно-орієнтованої архітектури):

1. Запит-відповідь (англ. – Request-reply) – відправник надсилає запит і очікує відповіді від отримувача. Цей підхід додає жорстку зв'язаність між компонентами, є актуальним для синхронної комунікації.

2. Публікація-Підписка (англ. – Publish-subscribe) – відправник (генератор) розповсюджує повідомлення серед декількох отримувачів (споживачів) без прямого адресування. Цей підхід актуальний для асинхронної комунікації.

3. Канал Точка-Точка (англ. – Point-to-point channel) – гарантує, що повідомлення отримає лише один споживач, навіть якщо їх декілька. Черги повідомлень зазвичай реалізують цей шаблон для ефективного розподілу навантаження та для уникнення дублювання.

4. Брокер Повідомлень (англ. – Message broker) – виступає посередником між відправниками та отримувачами, забезпечуючи комунікацію без прямої залежності між ними. За необхідності повідомлення перед надсиланням отримувачам можуть підлягати додатковому перетворенню, валідації тощо.

5. Подійно-орієнтоване повідомлення (англ. – Event-driven messaging) – дозволяє компонентам спілкуватися асинхронно у відповідь на певні події. Цей шаблон фактично реалізований в класичній інтерпретації подійно-орієнтованої архітектури.

6. Конкуруючі Споживачі (англ. – Competing consumers) – кілька споживачів обробляють повідомлення з однієї спільної черги. Кожне повідомлення обробляється лише одним споживачем, що забезпечує паралельну обробку без дублювання. Це корисно у сценаріях, де потрібно динамічно балансувати навантаження.

7. Маршрутний Лист (англ. – Routing slip) – замість фіксованої послідовності обробки цей підхід використовує гнучку маршрутизацію на основі умов виконання. Цей шаблон часто використовується в автоматизації

бізнес-процесів, де повідомлення повинні проходити через різні етапи затвердження.

8. Маршрутизатор на Основі Змісту (англ. – Content-based router) – вибір різних отримувачів повідомлення в залежності від змісту повідомлення відповідно до заздалегідь визначених правил. Цей шаблон корисний у ситуаціях, коли різні типи повідомлень потребують різної обробки.

9. Фільтр Повідомлень (англ. – Message filter) – вибірково пересилає повідомлення на основі певних критеріїв. Це допомагає зменшити непотрібну обробку, дозволяючи лише відповідним повідомленням досягати певних компонентів.

10.Канал Невдалих Повідомлень (англ. – Dead letter channel) – повідомлення, які не можуть бути успішно оброблені через помилки надсилаються до спеціальної черги. Завдяки цьому вони не блокують загальний потік обміну даними.

11.Шаблон Повторної Спроби (англ. – Retry pattern) – автоматична додаткова спроба надіслати повідомлення через певний інтервал, якщо воно не було опрацьовано раніше. Цей підхід корисний у розподілених системах, де тимчасові збої мережі або блокування бази даних можуть викликати втрату доступу до даних.

12.Гарантована Доставка (англ. – Guaranteed delivery) – забезпечує надійну доставку повідомлень навіть у випадку збоїв. Зазвичай реалізується через використання стійкого сховища, підтвердження отримання та механізми повторної передачі.

13.Транзакційний Обмін Повідомленнями (англ. – Transactional messaging) – повідомлення обробляються в межах транзакції, що гарантує або їх повне виконання, або повне скасування у разі збою. Це запобігає частковим оновленням даних і забезпечує цілісність інформації в розподілених системах.

14.Агрегатор Повідомлень (англ. – Message aggregator) – об’єднує кілька повідомлень в один вихідний результат, зменшуючи кількість даних, які

потрібно обробляти далі. Цей шаблон корисний для пакетної обробки даних для зменшення кількості додаткових взаємодій між мікросервісами.

Більшість сучасних технологій та систем обробки повідомлень застосовують кілька шаблонів зі списку вище для підвищення стабільності, відмовостійкості та гарантування коректної доставки даних від генератора до споживача. У таблиці 1 наведено агреговані результати проведеного автором дослідження з даними по використанню шаблонів найбільш популярними сервісами обробки повідомлень.

Таблиця 1 – Використання шаблонів сервісами обробки повідомлень

Сервіс / Шаблон обробки	**P2P**	**Pub-Sub**	**Competing Consumers**	**Retry**	**Txn Msg**	**Guaranteed Delivery**
Amazon SNS		✅				
Azure Web PubSub		✅				
Google Cloud PubSub		✅		✅		✅
Apache Kafka		✅	✅	✅		✅
Apache Pulsar		✅	✅	✅		✅
RabbitMQ	✅	✅	✅	✅	✅	✅
MSMQ	✅			✅	✅	✅
Amazon SQS	✅		✅	✅		✅

Azure Service Bus	✅	✅	✅	✅	✅	✅

Продовження таблиці 1

Сервіс / Шаблон обробки	**P2P**	**Pub-Sub**	**Competing Consumers**	**Retry**	**Txn Msg**	**Guaranteed Delivery**
Azure Queue Storage	✅		✅			
Google Cloud Tasks	✅			✅		
Amazon Kinesis		✅	✅			✅
Azure Service Bus Messaging	✅	✅	✅	✅	✅	✅

2 ПРОЄКТУВАННЯ СИСТЕМ З ПОДІЙНО-ОРІЄНТОВАНОЮ АРХІТЕКТУРОЮ

2.1 Поняття подій при взаємодії сервісів та їх проєктування

Першочерговим завданням при проєктуванні системи з подійно-орієнтованою архітектурою є виділення елементарної частини – події. У розділі 1 поняття події та повідомлення використовувались взаємозамінно для позначення одиниці інформації при комунікації між мікросервісами. Проте ці два терміни мають різне значення в контексті їхньої участі у взаємодії між частинами загальної системи. Найбільш інтуїтивно зрозумілі та прості для використання визначення цих понять надані у монографії [5]:

- подія – це факт, констатація результату виконаної дії з базовою інформацією про цей факт;
- повідомлення – це посередник, конкретна інформаційна одиниця, яка надсилається від генератора отримувачу.

Таким чином, в контексті систем з подійно-орієнтованою архітектурою йдеться про передачу даних про події через використання повідомлень, які надсилаються між різними компонентами. Основною задачею комунікації є передача даних про подію, і повідомлення використовується як метод для такої передачі.

Наступним важливим завданням при проєктуванні та розробці системи є виділення зони відповідальності для кожної події. Потрібно зрозуміти, яким чином мають використовуватись події, про які результати дій чи стану бізнес-сутностей вони мають знати, яку інформацію мають містити та яким чином мають передаватись між компонентами. Серед шаблонів для використання подій в залежності від наданої їм задачі та зони відповідальності можна виділити декілька найбільш розповсюджених [15]:

1. Подія-сповіщення (англ. – Event notification) – генератор події надсилає деталі про зміну стану своїх бізнес-сутностей всім зацікавленим споживачам.

При цьому, у загальному випадку генератор не очікує ніякої відповіді та не є залежним від результату обробки цієї інформації споживачем або споживачами. Перевагою подібного підходу є те, що при його використанні можлива повна незалежність та асинхронність обробки. Фактично генератор події та її споживач можуть не знати про існування один одного, кожен з них працює над своїми задачами автономно. Недолік подібного підходу є наслідком вказаної вище переваги – подібне використання подій можливе лише у випадку, коли не може бути потенційних сценаріїв залежності факту обробки події споживачем для генератора. В такому випадку можливі ситуації появи стану гонитви (англ. – Race condition), коли фактично одна бізнес-сутність оновлює свій стан в різних частинах системи в результаті роботи різних компонентів. При використанні цього підходу розмір даних в події зазвичай мінімальний. Основна задача події – лише сповістити про зміну.

2. Подійна передача стану (англ. – Event-carried state transfer) – генератор події надсилає повну інформацію про поточний стан бізнес-сутності всім зацікавленим споживачам. Аналогічно до попереднього шаблону, в більшості випадків генератор події не очікує ніякої відповіді та не залежить від факту обробки даних одним зі споживачів. Перевагою подібного підходу є те, що подія містить в собі всю інформацію про стан об’єкта, тобто відсутня необхідність в додаткових комунікаціях зі сторони споживачів. Недоліком цього підходу є те, що він зумовлює передачу більшої кількості даних між компонентами системи. Крім того, при використанні цього шаблону частина відповідальності за коректне опрацювання даних перекладається на споживача. Чим більше даних передається, тим більшою має бути кількість перевірок на консистентність та відповідність на стороні обробників даних.

3. Джерела подій (англ. – Event sourcing) – кожна зміна стану бізнес-сутності зберігається у вигляді події, яка може бути оброблена споживачами в будь-який момент часу. Основною перевагою цього підходу є те, що при його використанні можливе повне відновлення стану бізнес-сутності в межах різних компонентів системи на момент кожного подібного збереження. Подію

у даному контексті можна розглядати як зліпок стану об’єкта в певний момент часу. Цей шаблон у загальному вигляді є розширенням попереднього підходу подійної передачі стану, проте використання джерел подій є архітектурним рішенням з відповідними наслідками. Будь-яка зміна стану бізнес-об’єкта мусить мати відповідний запис в списку подій без винятків [21]. Це накладає обов’язки на всі елементи загальної системи. Перевагами підходу є легкий аудит, можливість профілювання та відлагодження з мінімальними витратами часу та ресурсів. Недоліками є робота з великими масивами даних на рівні кожного компонента системи, необхідність в чіткому дотриманні правил генерації та обробки подій, ймовірність появи стану гонитви.

4. Поділ відповідальності між наказом і запитом (англ. CQRS – Command query responsibility segregation) – використання подій для обробки наказів про зміну стану бізнес-сутностей. Цей шаблон не є класичним застосуванням подійно-орієнтованої архітектури, проте він є фактично найбільш коректним методом передачі даних при створенні наказів. У контексті мікросервісної системи запис та зчитування даних можуть відбуватись на рівні різних компонентів, що зумовлює необхідність комунікації при виконанні кожної з дій. При цьому концепція події у даному контексті схожа до вказаної у шаблоні подійної передачі стану. Цей шаблон є також архітектурним рішенням, що зумовлює його використання на рівні всіх елементів системи. Перевагою цього підходу є чіткий розподіл зон відповідальності між компонентами, які виконують запис або оновлення даних та компонентами, які зчитують дані і можуть лише використовувати бізнес-сутності без зміни їхнього стану. Недоліками підходу є складність технічної реалізації, потенційне дублювання даних та логіки для розмежування процесів запису та зчитування [22].

В залежності від обраного шаблону та архітектурного рішення при використанні подій в системі потрібно зрозуміти, який набір інформації має містити кожна подія. На цей вибір також впливають:

- наявність єдиної бази даних, доступ до якої є у генератора та у споживача;
- можливість використання одного типу події (з однаковим набором доступних даних) для різної кількості споживачів;
- тип обробки даних про подію споживачами;
- очікувана кількість операцій взаємодії між компонентами (кількість згенерованих та оброблених подій за певний проміжок часу).

Можна виділити три варіанти набору даних для кожної окремої події [5]:

1. Лише ідентифікатор. Цей варіант є актуальним при використанні шаблону події-сповіщення. Він може бути прийнятним у випадку використання генератором та споживачем одного джерела даних, коли немає обмеження на стороні споживача отримати доступ до потрібного бізнес-об'єкту для подальшої роботи з ним. У випадку відсутності доступу до єдиного джерела даних подібний підхід змушуватиме споживачів виконувати додаткові запити до генератора для отримання потрібної інформації про дані з події. На рисунку 5 зображено приклад реалізації цього варіанту.

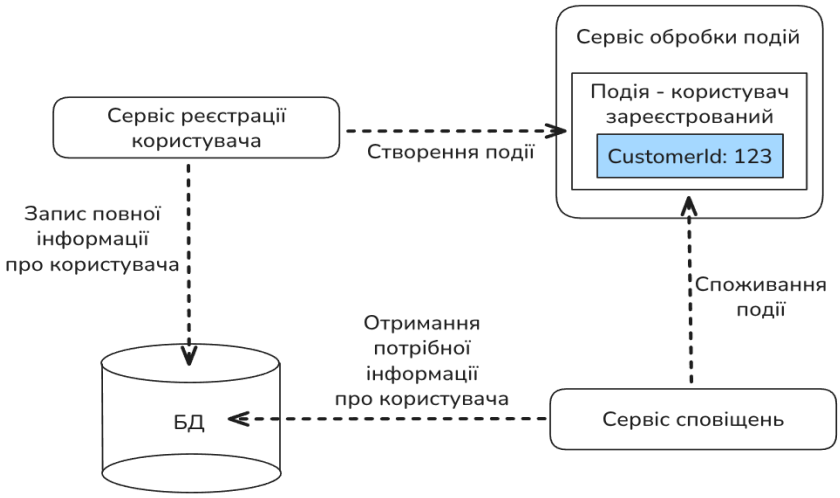


Рисунок 5 *Подія містить вказівник на ідентифікатор бізнес-сутності*

2. Часткова інформація стану бізнес-сутності. Цей варіант є актуальним у випадку, коли споживачу потрібно знати лише про певний набір атрибутів бізнес-сутності. При цьому, є обмеження доступу споживача до джерела даних, яке доступне генератору події. Недоліком цього підходу може бути

наявність більшої кількості споживачів, коли різні компоненти мають працювати з різним набором даних з обмеженням доступу до іншої інформації (наприклад, один компонент має працювати з платіжними деталями, інший – з особистими даними користувача тощо). На рисунку 6 зображено приклад реалізації цього варіанту.

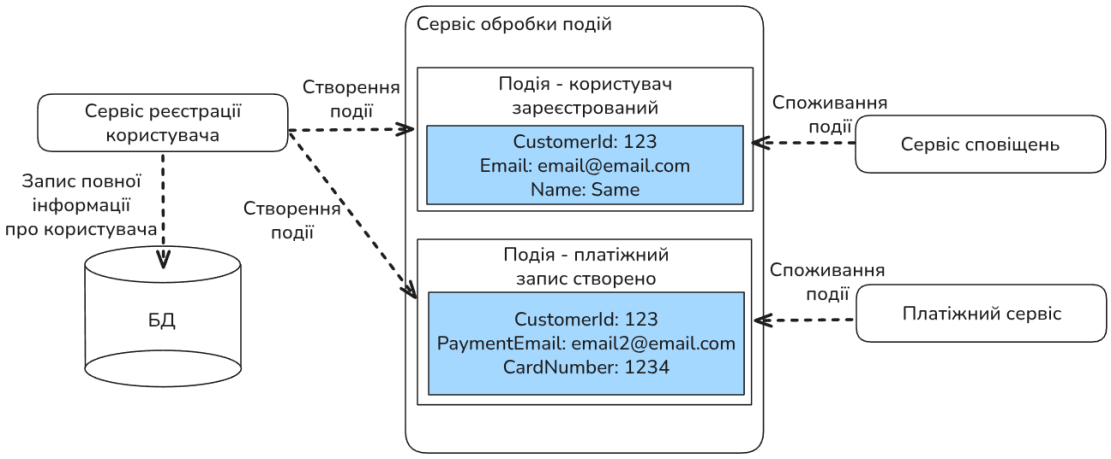


Рисунок 6 *Подія містить мінімальний набір даних бізнес-сутності*

3. Повна інформація стану бізнес-сутності. Цей варіант є актуальним при використанні шаблонів подійної передачі стану та джерел подій. На рисунку 7 зображено приклад реалізації цього варіанту.

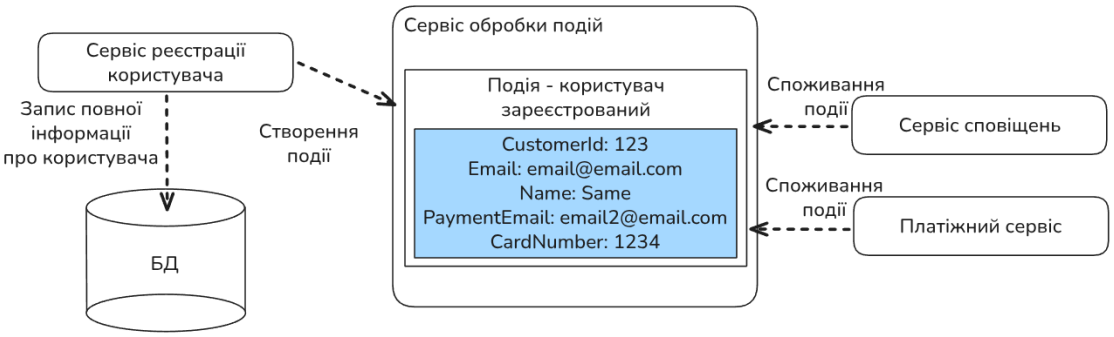


Рисунок 7 *Подія містить повний набір даних бізнес-сутності*

2.2 Особливості проєктування систем із використанням архітектурного шаблону “подійно-орієнтована архітектура”

Проєктування системи з подійно-орієнтованою архітектурою є багатоетапним процесом з необхідністю приймати виважені рішення під час кожної ітерації. Після розробки бізнес-моделі реальних процесів, розподілу структурних частин на компоненти відповідно до предметної області, моделювання шляхів комунікації та вибору шаблону застосування подій наступним кроком є розробка безпосередньої взаємодії між елементами відповідно до впорядкованої структури діяльності.

При виборі технологічної реалізації міжсервісної комунікації потрібно зважати на наступні характеристики [5]:

- зворотна сумісність при зміні технологій, набору мікросервісів в системі або каналів комунікації має досягатись максимально легко;
- зовнішні інтерфейси в міжсервісній комунікації мають бути явними та зрозумілими для всіх генераторів та споживачів подій;
- зовнішні інтерфейси комунікації не мають залежати від вибраних технологій реалізації;
- сервіси для зовнішньої комунікації мають бути простими для розуміння зі сторони споживачів;
- внутрішня реалізація обробки даних не має впливати на зовнішню взаємодію між компонентами системи.

Використання готових зовнішніх технологічних рішень (наприклад, RabbitMQ для черг повідомлень чи Apache Kafka для потокової обробки даних) може накладати обмеження на можливість досягнення повної зворотної сумісності. З іншого боку, ймовірність заміни технологічного рішення для обробника подій в будь-якій комплексній системі є малоймовірним. В контексті загальної сталості системи варто зробити висновок на основі вказаних характеристик про важливий принцип міжсервісної взаємодії: якщо розгортання нової версії одного мікросервіса вимагає оновлення версій інших мікросервісів, які беруть участь у взаємодії, вся система є архітектурно вразливою [23]. Таким чином, завдання ефективного проєктування подійно-орієнтованої мікросервісної системи можна звести до доволі простої у

формулюванні але нетривіальної у вирішенні задачі: потрібно підтримувати незалежність компонентів системи зі збереженням цілісності та консистентності бізнес-сутностей.

Можна виділити 10 найбільш розповсюджених шаблонів забезпечення цілісності даних в мікросервісних системах, порівняння основних характеристик яких наведено в таблиці 2 [24]. Шаблони “Event sourcing” та “CQRS” були детально розглянуті в розділі 2.1. в контексті архітектурних рішень при використання подій в системі. Вони є ефективними рішеннями досягнення загальної цілісності системи завдяки 4 властивостям – атомарності, узгодженості, ізольованості та довговічності (англ. ACID – Atomicity, Consistency, Isolation, Durability) [25], проте мають суттєві недоліки через складність впровадження та підтримки. Альтернативними популярними шаблонами для досягнення стану цілісності даних можна вважати “Розподілені транзакції” (англ. – Distributed transactions) та “Сага” (англ. – Saga).

Таблиця 2 – Шаблони забезпечення цілісності даних в системах з EDA

Метод	**Складність впровадження**	**Масштабованість**	**Можливий рівень цілісності**
Сага (англ. – Saga)	Середня	Висока	Високий
Джерела подій (англ. – Event sourcing)	Середня	Висока	Високий
Поділ відповідальності між наказом і запитом (англ. CQRS – Command query responsibility segregation)	Середня	Висока	Високий

Продовження таблиці 2

Метод	Складність впровадження	Масштабованість	Можливий рівень цілісності
Розподілені транзакції (англ. – Distributed transactions)	Висока	Низька	Високий
Об’єднання Api (англ. – Api composition)	Низька	Середня	Середній
Один сервіс – одна база даних (англ. – Database per service)	Низька	Середня	Середній
Кінцева цілісність (англ. – Eventual consistency)	Низька	Висока	Середній
Захоплення змін даних (англ. – Change data capture)	Середня	Висока	Середній
Компенсуючі транзакції (англ. – Compensating transactions)	Середня	Висока	Високий
Розподілене кешування (англ. – Distributed caching)	Середня	Висока	Середній

Використання розподілених транзакцій дозволяє контролювати процес запису та оновлення даних різними мікросервісами “батьківським” процесом, який можна вважати координатором або менеджером всіх етапів виконання загальної транзакції. При цьому, процес роботи над транзакцією ділиться на два етапи: етап голосування (підготовки) та етап виконання [26]. Під час етапу голосування координатор транзакції звертається до всіх учасників з запитом про можливість виконати їхню частину дії у транзакції. За відмови одного з учасників вся транзакція відміняється. У випадку підтвердження відбувається повноцінна обробка транзакції. Схематично логіку взаємодії між координатором та обробниками даних під час обох етапів зображено на рисунках 8 та 9 [5].

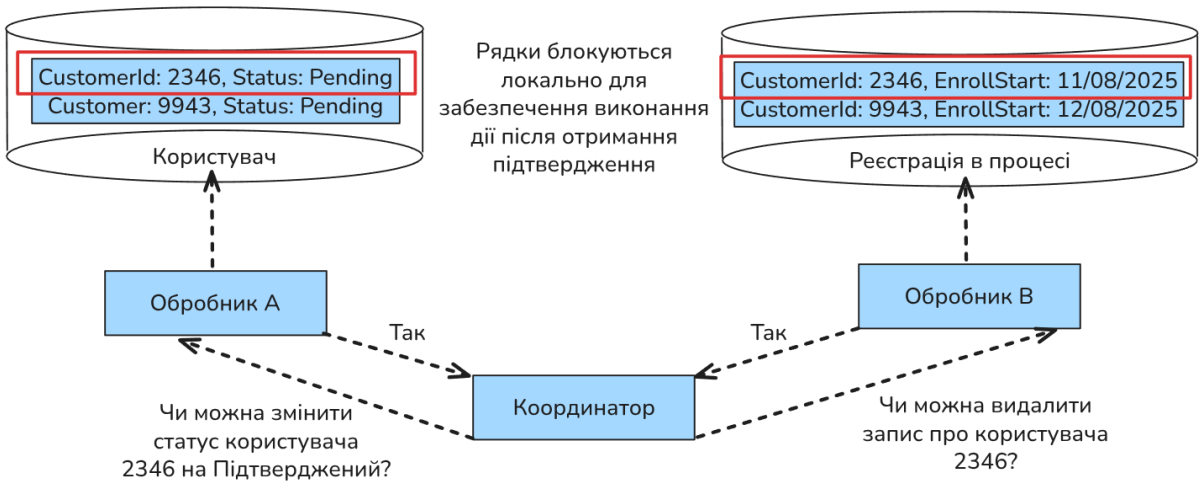


Рисунок 8 *Етап голосування в розподіленій транзакції*

Розподілені транзакції можуть здаватись доволі очевидним та відмовостійким методом досягнення стану цілісності. Проте, наявність декількох етапів комунікації між мінімум трьома учасниками процесу, залежність від одностайних відповідей та ризик до затримок в обробці роблять цей підхід вразливим та нестійким [26].

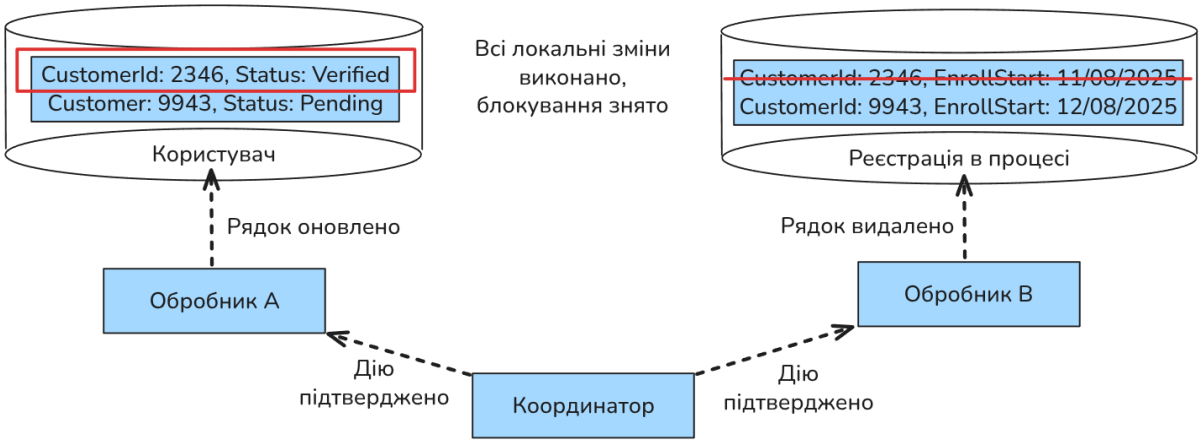


Рисунок 9 *Етап виконання в розподіленій транзакції*

На противагу вказаним недолікам шаблон “Saga” [27] використовує підхід виконання окремих транзакцій в межах кожного сервісу для оновлення стану бізнес-сутностей свого контексту. Кожна окрема дія запускає наступну для іншого сервісу. У випадку помилки на якомусь з етапів попередньо виконані транзакції відміняються [24].

Існує два варіанти застосування шаблону “Saga” [27]:

1. Хореографія. Кожен сервіс обробляє свою частину актуалізації даних та повідомляє інші сервіси про завершення обробки через загальний канал комунікації. Фактично кожен сервіс повідомляє про те, що він завершив свою частину операції, і процес може продовжуватись іншим або іншими сервісами. У випадку, якщо виникає помилка під час одного з наступних етапів обробки, аналогічним чином надсилається повідомлення в загальний канал комунікації, і попередній сервіс може виконати зворотну дію, повернувши стан бізнес-сутності до актуалізації. Цей підхід дозволяє не використовувати єдиний центр координації, проте може бути вартісним та доволі складним для реалізації в існуючих системах. Він також може викликати надлишок створюваних в загальній системі подій [29].

2. Оркестрація. Під час кожної операції є один основний сервіс (оркестратор), який контролює виконання всієї операції, запускає наступні дії іншими сервісами. Фактично кожен сервіс отримує повідомлення про необхідність виконання дії актуалізації стану лише тоді, коли оркестратор приймає рішення про те, що попередні дії були виконані коректно та загальний процес може продовжуватись. Перевагою цього підходу є наявність єдиної точки входу, що значно полегшує роботу з розширенням функціональності, пошуку помилок, підтримки. Недолік підходу пов’язаний з перевагою: оркестратор може стати вразливою частиною системи, у випадку порушення його роботи процес повністю зупиняється [30]. Іншим недоліком є ризик розширення зони відповідальності оркестратора задачами, які мають бути в контексті інших сервісів [5].

Важливо зазначити, що використання варіантів оркестрації та хореографії не є взаємовиключним в межах однієї системи. Оркестрація може мати значні переваги у контексті з меншою кількістю дій, в той час як хореографія має значно більше переваг у розширених та складних контекстах. На рисунку 10 зображена схема комунікації між елементами системи з використанням обох варіантів обробки подій [26].

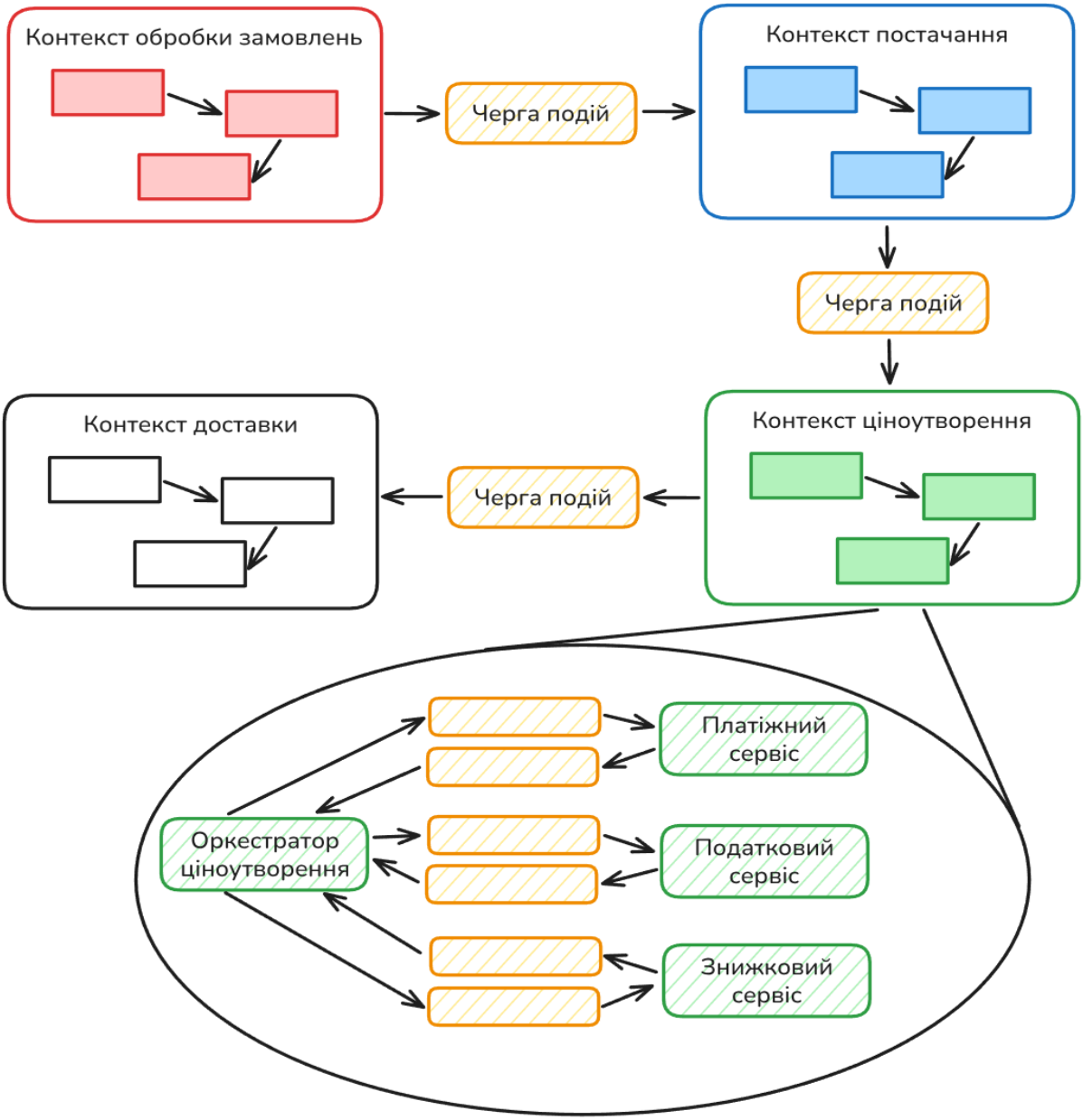


Рисунок 10 *Використання "Saga" оркестрації та хореографії в системі*

Кардинально іншим підходом до вирішення питання підтримки консистентності даних при комунікації між мікросервісами є кінцева цілісність (англ. – Eventual consistency). Розглянуті вище шаблони мають на меті підтримку системи в стані постійної цілісності. Проте, ймовірність незначних затримок при оновленні стану бізнес-сутностей в межах різних мікросервісів може бути допустимою [28]. Використання асинхронних методів комунікації зумовлює факт наявності певних часових проміжків, коли подія про оновлений стан бізнес-сутності вже була створена в межах одного

сервісу, проте ще не була оброблена іншим сервісом. Проте, при використанні технологій з гарантуванням доставки можна говорити про кінцеву цілісність даних, яка досягається з плином часу. Тобто, в кожен конкретний момент системи бізнес-сутності можуть бути в різних станах, проте ці стани будуть тяжіти до досягнення стану повної цілісності і консистентності в процесі обробки раніше створених в системі подій.

Вибір шаблону для підтримки цілісності даних впливатиме на наступні етапи проєктування системи з подійно-орієнтованою архітектурою [26]:

- обробка одночасних, застарілих та хронологічно загублених подій;
- досягнення стійкості та надійності роботи системи під навантаженням та змінною кількістю подій;
- схема організації створення подій різними контекстами;
- досягнення стану відмовостійкості та методи відновлення у випадку краху.

У результаті виконання всіх базових етапів проєктування системи з подійно-орієнтованою архітектурою кожен мікросервіс можна вважати готовим до технічної реалізації та розгортання. Для перевірки готовності до цього фінального етапу можна використати контрольний список [31], де кожен пункт відповідає критично важливим характеристикам для елемента системи та міжсервісної комунікації:

1. Кожен сервіс та канали комунікації є стабільними та надійними. Методи розробки та розгортання погоджуються з технічною командою, підтверджуються задокументованим процесом та кроками для виконання.

2. Кожен сервіс є масштабованим, канали комунікації не залежать від незначної зміни кількості подій для опрацювання. За необхідності сервіс може бути масштабовано горизонтально чи вертикально, що не буде мати негативних наслідків на наявні екземпляри процесу.

3. Кожен сервіс є відмовостійким без наявності єдиної точки збою. Для кожного елементу системи існує план аварійного відновлення з досяжними кроками.

4. Кожен сервіс та канали комунікації покриті системами моніторингу з кількісними метричними показниками для спостереження та аналізу. Логування наявне для доречних взаємодій з використанням відповідних до ситуації рівнів.

5. Кожен сервіс та міжсервісний канал комунікації мають технічну та бізнес-документацію. Ця документація оновлюється регулярно відповідно до технічних змін в системному елементі.

2.3 Методи вирішення викликів при використанні подійно-орієнтованої архітектури

Переваги використання шаблону “подійно-орієнтована архітектура” в системах супроводжуються ризиками, недоліками та задачами для підтримки стабільного та надійного стану бізнес-сутностей в межах кожного з учасників комунікації.

Одним з найбільш актуальних завдань, яке має вирішуватись постійно при використанні подій і є майже завжди нерелевантним у монолітних системах завдяки можливості легко контролювати потоки даних та канали взаємодії між модулями, є наявність паралельних (створених майже одночасно чи протягом незначного часового періоду) подій. Паралелізм у свою чергу викликає ризик наявності хронологічно загублених подій або подій у порушеному порядку. Найчастіше паралелізм вирішується за рахунок використання двох діаметрально протилежних стратегій [32]:

- Оптимістична – кожна подія обробляється за припущення відсутності ризику наявності паралельної зміни, яка може викликати конфлікти при обробці. Цей підхід притримується простої логіки: всі дані актуалізуються одразу при їхній появі. У випадку помилки операції відбуваються додаткові перевірки для відновлення попереднього стану бізнес-сутності. Оптимістична стратегія є пріоритетною у системах з назначною частотою створення подій та у системах з невеликою кількістю елементів. Чим менше ймовірність

одночасного оновлення стану бізнес-сутності в різних контекстах, тим більший зиск від використання цього підходу. У той же час в системах з великим навантаженням та частотою появи нових подій ця стратегія буде зумовлювати надмірне використання ресурсів та операцій відміни через необхідність повернення попереднього стану об’єктів.

- Песимістична – перед обробкою кожної події відбувається блокування даних від ймовірного паралельного оновлення. Фактично нова подія може бути оброблена тільки тоді, коли попередня операція була успішно завершена. Ця стратегія гарантує відсутність паралельної зміни стану бізнес-сутності, тобто підтримує принцип постійної цілісності. В той же час кожна подібна операція блокування вимагає більшої кількості ресурсів для обробки події та може впливати на час обробки. Цей підхід є пріоритетним в системах з великою кількістю подій, коли запити на оновлення стану бізнес-сутностей можуть створюватись часто зі сторони різних мікросервісів.

Обидві стратегії можуть використовуватись в різних контекстах однієї системи. Вибір може бути обумовлений очікуваним станом частоти подій та кількістю учасників у процесі взаємодії. Складність такого вибору полягає у тому, що питання частоти доволі часто є змінним та може залежати від багатьох внутрішніх та зовнішніх факторів, як-от час доби, глобальні знижкові кампанії на свята тощо [26]. Більш універсальним та гнучким методом вирішення питання паралельності обробки подій є використання наскрізних розділів. Основна ідея підходу полягає у розподілі всіх елементів потоку між декількома групами (розділами), де кожен елемент закріплюється за певним розділом, нові події для цього елементу мають порядковий номер та хронологічно можуть створюватись лише в межах цього розділу. Ці характеристики дозволяють повністю нівелювати ризик появи паралельності чи порушених у порядку подій. Одним з найбільш відомих інструментів з реалізацією цього підходу є Apache Kafka [17].

Іншим важливим завданням при роботі з подіями та асинхронною комунікацією в цілому є підтримка надійності. Незалежність та автономність

компонентів є першопричиною наявності ризиків того, що різні елементи системи можуть бути в різному стані в певний момент часу: один сервіс може перезавантажуватись, другий може мати проблеми з мережевим з'єднанням, третій може бути зайнятий опрацюванням великої кількості поточних запитів тощо. Велика кількість факторів може впливати на те, що швидкість та факт опрацювання події може дуже сильно відрізнятись для різних учасників комунікації. В контексті семантики гарантування доставки можуть використовуватись три стратегії [26]:

1. Доставка щонайбільше один раз (англ. – At-most-once delivery) – всі повідомлення при комунікації між сервісами можуть бути доставлені не більше одного разу. Для доставки виконується тільки одна спроба. У випадку наявності якоїсь помилки (відсутнє мережеве з'єднання, тимчасова недоступність споживача тощо) повідомлення не буде доставлене. Ця стратегія найчастіше використовується при взаємодії між модулями монолітної системи через невелику ймовірність затримок чи ризиків наявності перешкод при комунікації. В той же час цей підхід може також використовуватись в мікросервісній архітектурі для контекстів з великою кількістю опціональних повідомлень, коли швидкість опрацювання даних є головним фактором, й ризики втрати певної кількості даних не є суттєвими. Завдяки відсутності додаткових перевірок та спроб цей підхід вимагає набагато менше ресурсів як під час кожної комунікації, так і в стані бездіяльності (немає необхідності використовувати додатковий шар пам'яті для тимчасового збереження даних).

2. Доставка принаймні один раз (англ. – At-least-once delivery) – всі повідомлення будуть обов'язково доставлені хоча б один раз, проте можливе дублювання даних у випадку затримки з підтвердженням факту отримання на стороні споживача. Ця стратегія вимагає використання додаткового шару пам'яті для тимчасового збереження даних для виконання наступних спроб доставки. У цьому випадку гарантується доставка повідомлення, проте необхідна більша кількість ресурсів для збереження стабільності та

відмовостійкості. Цей підхід дозволяє підтримувати систему в стані цілісності та надійності постійно, проте питання продуктивності через наявність додаткових перевірок можуть робити його неактуальним для деяких контекстів.

3. Доставка рівно один раз (англ. – Exactly-once delivery) – ідеальний варіант реалізації стратегії з гарантуванням доставки повідомлення одержувачу тільки один раз. Цей підхід використовує повторні спроби доставки тільки у випадку невдалих попередніх, при цьому немає ризиків появи дублікатів. В технічному плані цей варіант є найскладнішим та може вимагати додаткових шарів при роботі з даними. У більшості випадків ця стратегія вважається майже неможливою для реалізації в асинхронному середовищі, проте комбінація транзакційності, потокової обробки та чітка робота з зонами відповідальності в системі можуть дозволити досягти цієї мети [33].

В контексті підтримки цілісності даних та надійності системи важливо підтримувати баланс між кількістю необхідних ресурсів для ефективної комунікації між компонентами та коректним станом бізнес-сутностей. В деяких випадках додаткові заходи для гарантування та підтримки цілісності можуть коштувати набагато дорожче потенційних ризиків при втраті частини даних. Питання необхідності мати постійну цілісність бізнес-об’єктів може бути доволі дискусійним, тому в загальному випадку наявність кінцевої цілісності може бути прийнятним для більшої кількості контекстів [5].

Одним з комплексних варіантів підтримки подійно-орієнтованої системи в стані цілісності та надійності є використання шаблону багатоетапності при роботі з подіями (англ. SEDA – staged event-driven architecture) [34]. Цей метод є актуальним для систем з ризиком створення великої кількості одночасних подій з різною непередбачуваною частотою генерації. В той же час він буде створювати зайві ускладнення розробки, розуміння та підтримки в простих системах без наднавантажень. Основна ідея багатоетапності полягає в регулюванні вхідних даних для кінцевих споживачів

за рахунок використання додаткових шарів – посередників з поглибленою логікою обробки вхідних потоків даних. Кількість подібних етапів залежить від генераторів вхідних даних з різною очікуваною частотою створення подій [35]. Кожен етап має внутрішній обробник потоку вхідних подій, які можуть додатково зберігатись для поступової відправки та регулювання швидкості генерації зовнішніх подій для кінцевих одержувачів. На рисунку 11 зображено схему роботи етапу для обробки вхідних даних [34].

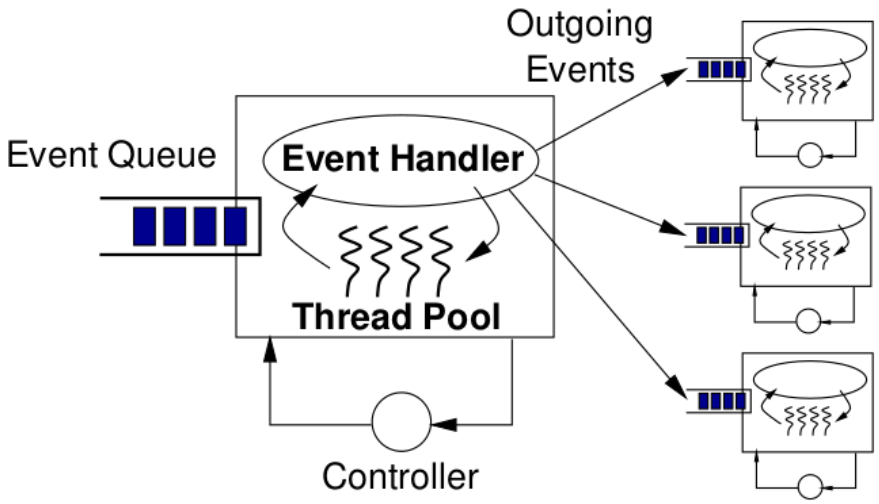


Рисунок 11 *Етап при використанні SEDA* [34]

3 АРХІТЕКТУРА ТА ЛОГІКА ПІДСИСТЕМИ ОНОВЛЕННЯ КАТАЛОГУ СИСТЕМИ ПРОДАЖІВ ДЕКАТЛОН УКРАЇНА

3.1 Загальний принцип та схема оновлення каталогу підсистеми Декатлон Україна

Для дослідження особливостей роботи систем з подійно-орієнтованою архітектурою, виявлення викликів та недоліків, аналізу методів оптимізації було розглянуто застосунок оновлення каталогу товарів підсистеми Декатлон Україна [36]. Поточна система актуалізації стану каталогу в межах інтернаціональної групи Декатлон складається з декількох частин, які у спрощеному вигляді схематично представлені на рисунку 12:

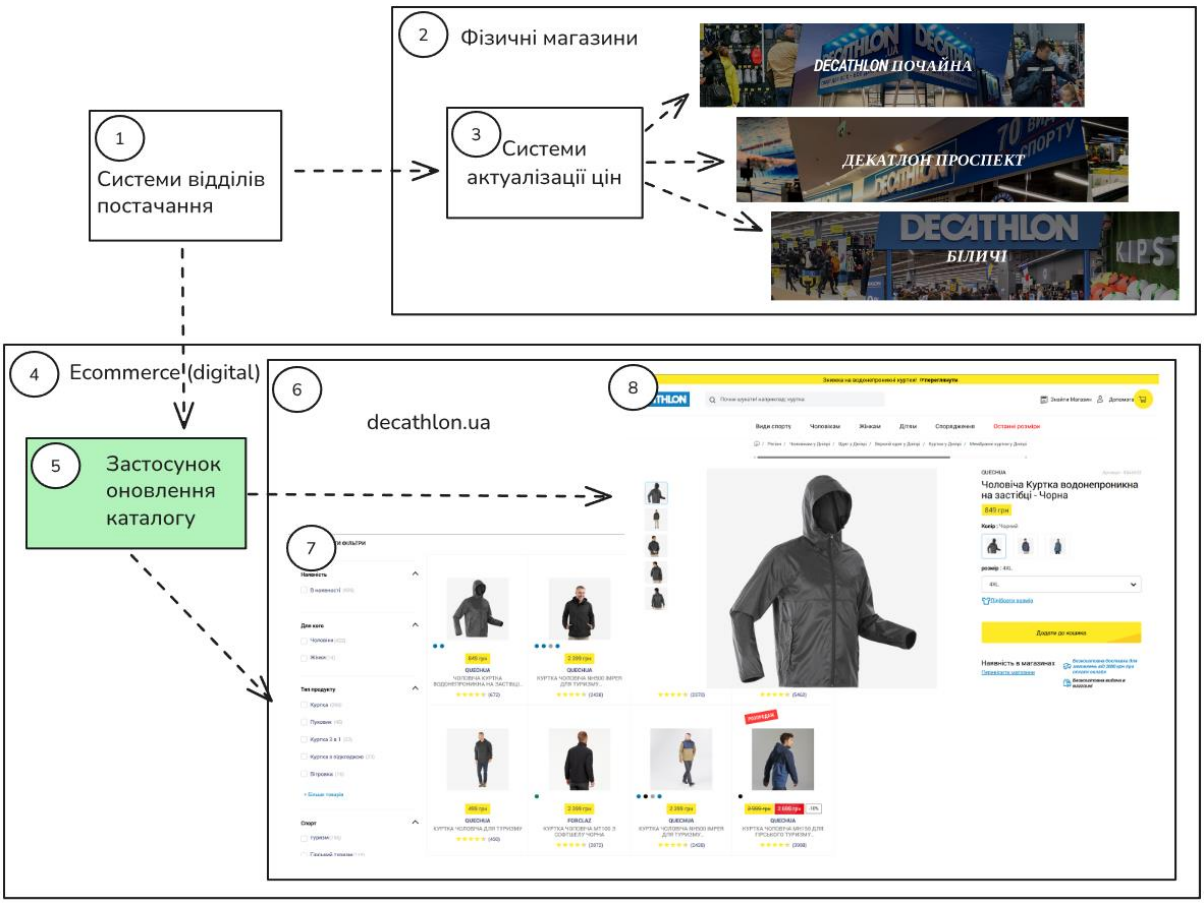


Рисунок 12 *Схема актуалізації стану каталогу Декатлон Україна*

1. Системи відділів постачання – зовнішні джерела формування загального стану каталогу, які регулюють набір товарів, цінові політики, актуалізацію характеристик. Кожна з систем відповідає за окремі атрибути товару, які надсилаються спільній системі агрегування стану для подальшої передачі в системи актуалізації цін у фізичних магазинах (3) та застосунку актуалізації каталогу електронної комерції (5). Ці компоненти мають комплексну внутрішню логіку, складаються з багатьох рівнів взаємодії та знаходяться поза зоною впливу системи Декатлон Україна, тому додатково не досліджувались в рамках поточної роботи.

2. Фізичні магазини – торгові заклади, в яких товари каталогу доступні для безпосереднього огляду, вибору та купівлі покупцями. Наразі Декатлон Україна налічує 3 фізичні магазини, всі вони знаходяться у м. Київ.

3. Системи актуалізації цін фізичних магазинів – внутрішні компоненти обробки оновлень каталогу, які отримують дані від систем відділів постачання (1), формують та друкують цінові мітки для товарів, опрацьовують рекламні пропозиції, регулюють ціни для груп товарів з залишковими розмірами тощо.

4. Система електронної комерції (англ. – Digital ecommerce) – набір застосунків, технічних рішень та безпосередньо інфраструктури веб-сайту decathlon.ua. В контексті взаємодії з каталогом товарів можна виділити дві частини системи – застосунок оновлення каталогу та веб-сайт.

5. Застосунок оновлення каталогу – умовно незалежний компонент у рамках системи електронної комерції, який отримує дані повного стану каталогу товарів (має бути ідентичним до стану каталогу товарів для фізичних магазинів), оновлень характеристик кожного продукту, зміни доступності для кожного з розмірів тощо, які в результаті обробки застосовуються для веб-сайту decathlon.ua (6). Цей застосунок є безпосередньо основним елементом дослідження поточної роботи.

6. Веб-сайт decathlon.ua – компонент взаємодії користувачів з електронною версією каталогу з можливістю пошуку товарів за різними характеристиками та критеріями, вибору відповідного товару за умови його наявності у

фізичному магазині, оплатою та оформленням доставки. В контексті роботи з каталогом його можна додатково поділити на компонент взаємодії зі списком доступних товарів (7) та компонент взаємодії з конкретним товаром (8).

7. Сторінка пошуку і категоризації – виводить набір товарів, які відповідають пошуку користувача через пошукову панель або з приналежністю до категорії, яку відкрив користувач через доступні елементи меню. Кожен елемент видачі результатів відповідає конкретному товару.

8. Сторінка товару – виводить найбільш повну та докладну інформацію по товару, який користувач вибрав через сторінку пошуку або сторінку категоризації (7). Вона має містити актуальні дані по наявності товару, цінах (з коректною інформацію по наявних знижках, якщо це актуально), деталях використання товару, складу компонентів, умов зберігання та експлуатації тощо.

Актуальність каталогу Декатлон Україна полягає в тому, щоб у будь-який момент часу коректний стан товарів від зовнішніх систем відділів постачання (1) був представлений у кожному з фізичних магазинів (2) та системі електронної комерції (4). Основна задача застосунку оновлення каталогу (5) полягає в тому, щоб усі оновлення змін у каталозі застосовувались для всіх елементів взаємодії користувачів з продуктами каталогу через веб-сайт (6). Саме наявність двох незалежних структурних елементів зумовлює найбільший виклик при роботі з каталогом продуктів.

Згідно з внутрішніми дослідженнями та проведеними опитуваннями серед покупців продукції Декатлон Україна в місті Київ, близько 60% користувачів починають взаємодію з каталогом товарів через веб-сайт, адже це пришвидшує пошук потрібних даних та ознайомлення з доступним набором продуктів. Це дозволяє вибрати потрібний товар в межах очікуваних цінових діапазонів, бажаних характеристик та доступності. Крім того, можна швидко порівняти аналогічні товари різних виробників та продавців, щоб в подальшому значно звузити пошук. При цьому, більше половини потенційних покупців їдуть в фізичний магазин для безпосередньої купівлі товарів. Подібні

дані підтверджуються загальними світовими трендами та дослідженнями [37, 38]. Ефект поєднання фізичних магазинів з онлайн-майданчиками отримав назву “шукай онлайн, купуй офлайн” (англ. ROPO – research online, purchase offline), та є домінуючим на ринку фізичних послуг та виробництва товарів масового споживання протягом останніх 10 років [39].

Потенційні покупці очікують побачити ідентичний стан наявності, цін, характеристик та розмірів (якщо актуально для конкретного продукту) обраних товарів в фізичному магазині відповідно до переглянутого раніше через веб-сайт. Подібний стан відповідності товару на онлайн-вітрині веб-сайту з реальним товаром у фізичних магазинах є головним критерієм ефективності роботи застосунку актуалізації каталогу Декатлон Україна.

3.2 Принцип та схема роботи підсистеми оновлення каталогу та взаємодія з зовнішніми джерелами інформації

Задачу актуалізації стану каталогу веб-сайту Декатлон Україна виконує умовно незалежний застосунок, написаний мовою PHP версії 8.4 через каркас Symfony з використанням архітектурної парадигми предметно-орієнтованого проєктування та контейнеризації готових збірок. Умовна незалежність в даному контексті полягає в тому, що застосунок зберігає автономність під час своєї роботи, проте є абсолютно залежним від зовнішніх джерел інформації через генератори тем Apache Kafka. У випадку зупинки процесу заповнення теми новими подіями з оновленням стану набору продуктів застосунок не зможе виконувати свою задачу, адже він є фактично посередником в загальному процесі.

Застосунок складається з окремих частин, в тому числі

- споживач зовнішньої теми Apache Kafka [17];
- брокер, генератор та споживачі повідомлень RabbitMQ [16];

● демони-планувальники періодичних задач (англ. – Cron-job) для створення групи повідомлень RabbitMQ, додаткової обробки помилкових даних, до-опрацювання процесів на некоректних станах тощо [40].

Поточна спрощена схема роботи застосунку актуалізації стану каталогу веб-сайту представлена на рисунку 13:

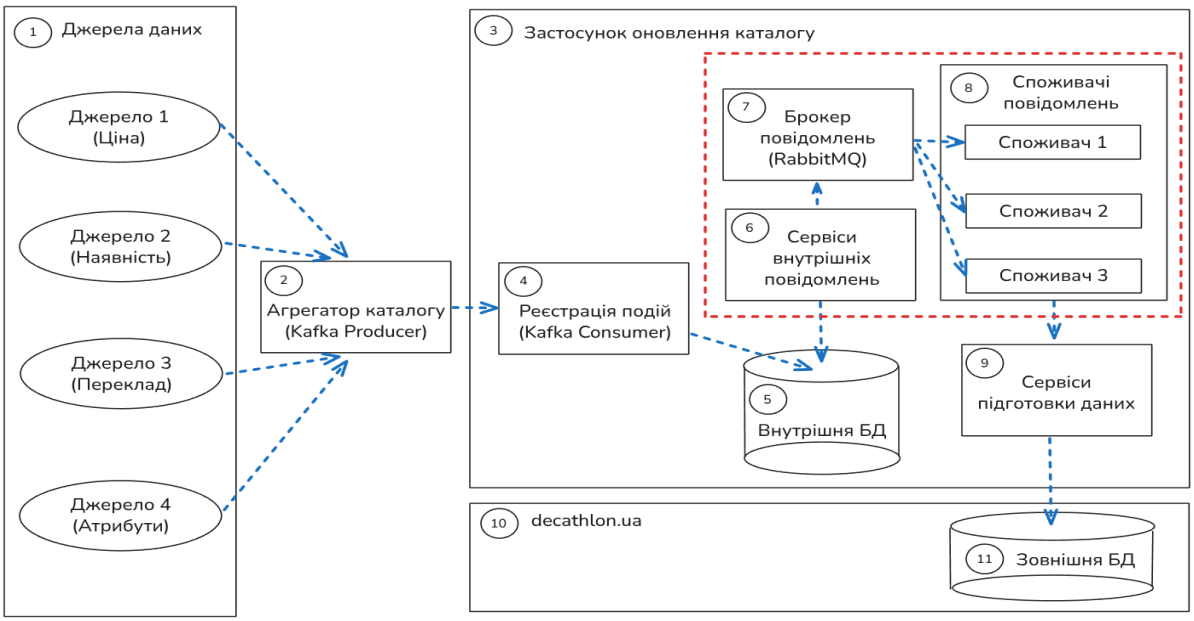


Рисунок 13 *Схема елементів застосунку оновлення каталогу веб-сайту decathlon.ua*

1. Зовнішні джерела даних, серед яких є системи управління цінами, наявністю, перекладами, деталями тощо. Ці джерела контролюються відділами постачання, міжнародними вузлами виробництва товарів, відділами стратегічного планування.

2. Агрегатор даних каталогу, який представляє собою генератор теми Apache Kafka [17], та є єдиною точкою входу для споживачів актуалізації стану каталогу. Кожна тема (для Декатлон Україна використовується одна тема для всього каталогу з прив’язкою до єдиної мови на сайті на даний момент – української) розділена на 4 розділи, використовується підхід компактності подій теми [17].

3. Власне застосунок оновлення каталогу складається з декількох структурних частин (4-9) та комплексною внутрішньою логікою обробки, підготовки та збереження даних.

4. Реєстратор подій оновлення каталогу – є фактично споживачем теми Apache Kafka [17], основною задачею якого є збереження даних актуального стану продукту відповідно до інформації з отриманої події у внутрішню базу даних застосунку (5). Реєстратор подій заповнює отримані дані та надсилає відповідь про успішну обробку генератору подій (2), що дозволяє мати актуальний стан зміщень та затримок для кожного розділу теми.

5. Внутрішня база даних (БД) – в поточній версії застосунку Декатлон Україна є docker контейнером [41] з сервісом БД MongoDB [40] (нові версії додають можливість використовувати зовнішні сервіси з БД PostgreSQL [43]). Основною задачею компонента є збереження стану останньої події по кожному продукту, логування змін та потенційних помилок при обробці даних. Дані з внутрішньої БД також використовуються для моніторингу стану застосунку та процесу оновлення каталогу в цілому.

6. Сервіси створення та відправки внутрішніх повідомлень – фактично є генераторами повідомлень RabbitMQ [16], які надсилаються брокеру (7). Основною задачею компонента є періодична перевірка наявності нових зареєстрованих подій у внутрішній базі даних та генерація повідомлень для внутрішньої асинхронної обробки. Сервіс використовує консольну команду каркасу Symfony [44], яка додана в розклад демона-планувальника Linux [40] для відпрацювання кожні 10 хвилин.

7. Брокер RabbitMQ – реалізує обробку вхідного потоку повідомлень від внутрішніх сервісів (6) та регулює розподіл їх між внутрішніми споживачами (8).

8. Споживачі повідомлень – сервіси асинхронної обробки повідомлень черги RabbitMQ [16] від брокера для паралельної обробки зовнішніх вхідних даних в декілька потоків. На поточний момент використовується 3 споживачі, які працюють одночасно з єдиною чергою повідомлень та забезпечують

паралельне опрацювання даних кількома потоками. В залежності від кількості вхідних даних цей показник може регулюватись, що дозволяє використовувати масштабованість обробки оновлень каталогу в залежності від кількості зовнішніх змін.

9. Сервіси підготовки даних – службові внутрішні сервіси для валідації отриманих зовнішніх даних з подій та трансформації їх у готові стани продуктів каталогу, які у результаті обробки застосовуються для веб-сайту (10). Сервіси є фактично частиною асинхронної обробки даних споживачами повідомлень (8) та виконують службову функцію трансформації та нормалізації даних згідно з очікуваною структурою кінцевим споживачем.

10.Веб-сайт decathlon.ua – кінцевий споживач даних оновлення каталогу. В процесі обробки кожної події застосунком відбувається запис до зовнішньої БД (11).

11.Зовнішня БД – хмарний сервіс з розгорнутою БД MySQL, яка є основним джерелом інформації при взаємодії користувачів на онлайн-майданчику (10).

З архітектурної точки зору програмна частина застосунку складається з 4 шарів, які представлені на рисунку 14:

1. Змістове ядро (англ. – Domain) – набір сутностей, об’єктів-значень та логіки для обробки даних з зовнішніх подій (отриманих з теми Apache Kafka) через використання внутрішніх подій предметної області застосунку (використовується обробник подій каркасу Symfony [45]). Фактично тут відбуваються основні процеси отримання, збереження у внутрішню базу даних, валідації та трансформації даних для можливості їх подальшого використання у коректній структурі для зовнішньої бази даних веб-сайту Декатлон Україна.

2. Шар застосунку (англ. – Application) – проміжний шар між змістовим ядром з бізнес-логікою застосунку (1) та шаром взаємодії (3). Основною задачею є обробка даних події теми Apache Kafka та трансформація їх через використання об’єктів передачі даних (англ.

DTO – Data transfer object) [46] для подальшої обробки згідно з бізнес-правилами.

3. Шар взаємодії (англ. user interface – UI) – набір консольних команд [44] для запуску споживачів теми Apache Kafka та службових додаткових сервісів (моніторинг, перевірка стану системи, очистка загублених даних, повторна обробка даних з помилками тощо).

4. Шар забезпечення (англ. – Infrastructure) – додаткові сервіси для обробки та трансформації даних, які не залежать від бізнес-вимог застосунку та можуть у випадку необхідності бути замінені на інші інструменти.

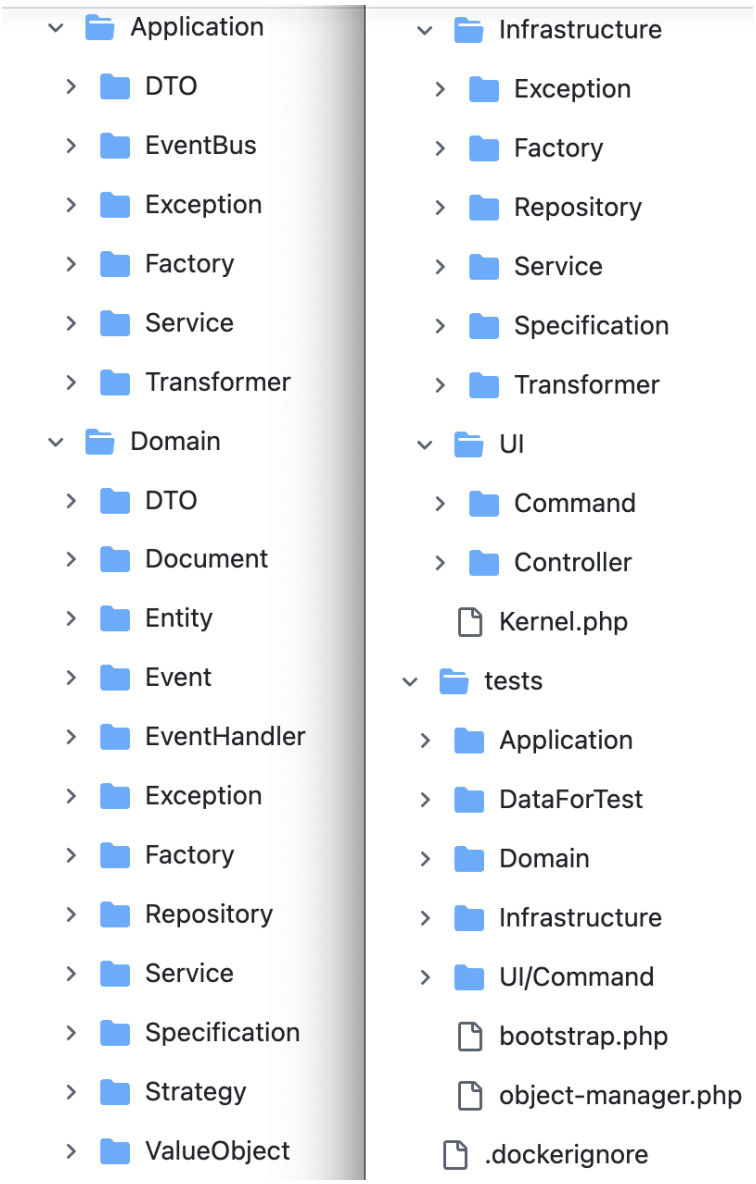


Рисунок 14 *Архітектура застосунку оновлення каталогу Декатлон Україна*

Усі базові класи застосунку мають відповідні модульні тести бізнес-логіки та розгалужень поведінки, завдяки чому досягається покриття тестами застосунку на рівні 88%.

Завдяки використанню docker контейнеризації [41] у робочому стані застосунок складається з кількох контейнерів, зображених на рисунку 15:

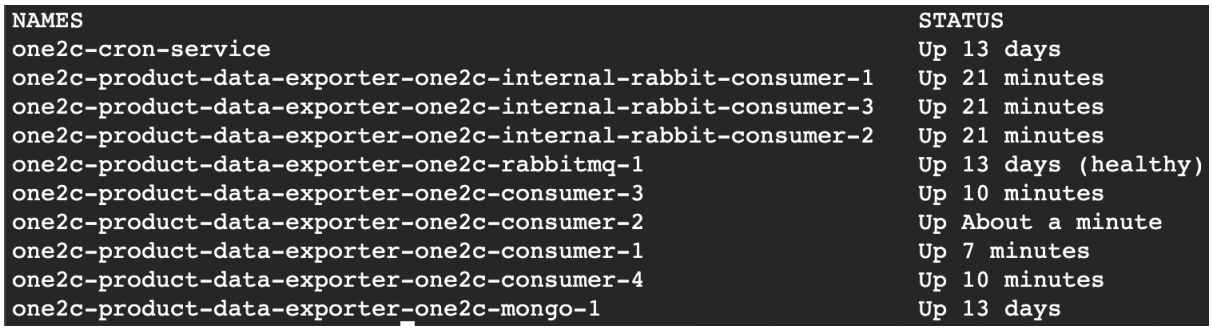


Рисунок 15 *Docker контейнери застосунку оновлення каталогу Декатлон Україна*

1. Демон-планувальник (контейнер cron-service) – використовується для періодичного запуску сервісів [40] перевірки наявності нових подій, моніторингу застосунку, повторної обробки подій при виникненні помилок тощо.

2. Внутрішні споживачі RabbitMQ (контейнер internal-rabbit-consumer) – сервіси споживання даних через чергу від брокера [16]. У стандартному режимі роботи застосунку кількість копій для можливості асинхронно обробляти дані встановлено на рівні 3.

3. Брокер RabbitMQ (контейнер rabbitmq) – сервіс для брокера потоку даних [16].

4. Споживачі зовнішньої теми Apache Kafka (контейнер consumer) – сервіси споживання зовнішніх подій підрозділів загальної теми [17] та їх реєстрації у внутрішній базі даних для подальшої обробки.

5. Сервіс БД MongoDB (контейнер mongo) – екземпляр бази даних для збереження записів на рівні застосунку.

3.3 Недоліки поточної логіки роботи підсистеми оновлення каталогу

У результаті дослідження поточного стану та логіки роботи застосунку оновлення каталогу було виділено декілька важливих технічних та конфігураційних особливостей:

- використання багатоетапної обробки подій [34] зумовлює ризик появи бізнес-сутностей у нестабільному стані, через що виникає необхідність у періодичній очистці даних та повторному запуску циклу опрацювання внутрішніх подій;
- поточний варіант опрацювання подій функціонує коректно при сталій частоті створення подій генератором Apache Kafka, проте має суттєві недоліки неефективного використання ресурсів при змінній кількості створених подій за певний проміжок часу;
- ускладненість внутрішньої структури обробки даних (елементи 6-8 на рисунку 13, на схемі виділені червоною пунктирною лінією); саме ці елементи були першочерговими об’єктами дослідження можливого покращення ефективності роботи застосунку.

Під час дослідження наявної інформації, технічних вимог та документації по роботі застосунку було приділено особливу увагу питанню використання додаткового шару внутрішньої системи обробки повідомлень RabbitMQ. Згідно з отриманими даними, підхід багатоетапної обробки був використаний для нівелювання ризиків появи великої кількості нових подій по одному продукту у дуже короткі проміжки часу. Подібна ситуація гіпотетично може призвести до суттєвого збільшення кількості викликів оновлення даних каталогу зі сторони застосунку для веб-сайту. Було прийнято рішення використати схему, за якої

- усі нові події реєструються в режимі реального часу, при цьому у випадку появи нової події по продукту, який щойно вже отримав іншу подію, остання перезаписує стан попередніх даних;

- кожні 10 хвилин відбувається періодична перевірка всіх зареєстрованих подій за цей проміжок часу, які обробляються шляхом створення внутрішньої черги повідомлень;
- швидкість загальної обробки даних майже не зазнає негативних наслідків за рахунок використання декількох асинхронних споживачів повідомлень черги RabbitMQ;
- дані по кожному продукту каталогу зберігаються у внутрішній базі даних, при цьому для кожної нової події відбувається перевірка змінених даних у порівнянні з попереднім збереженим станом; у результаті кожна нова подія, яка ще не була оброблена, додає до списку змінених даних набір атрибутів, які необхідно обробити під час створення нового внутрішнього повідомлення;
- завдяки обробці лише однієї (завжди останньої) події по кожному продукту під час подібних періодичних перевірок перекривається ризик завеликої кількості оновлень одного продукту протягом короткого проміжку часу.

Незважаючи на те, що задачу перекриття потенційного дублювання даних методом багатоетапної обробки було вирішено, поточна логіка та технічна реалізація застосунку оновлення каталогу має недоліки:

1. Поєднання двох технологій обробки подій в межах однієї системи є неефективним. Будь-яка додаткова технологія в межах застосунку, яка ускладнює його роботу, призводить до більшої кількості необхідних ресурсів, перетворень даних та виконаних дій для отримання однакового результату.

2. Погіршується поріг входження для роботи з підтримки та розробки застосунку. Велика кількість деталей функціональності робить його складним для розуміння не тільки зі сторони бізнес-користувачів, але також для технічних спеціалістів.

3. Велика кількість шарів даних збільшує необхідну кількість ресурсів на виконання однієї дії. У результаті додавання прошарку RabbitMQ дані необхідно спочатку зареєструвати у внутрішню БД, потім прочитати при

виконанні сервісів періодичної обробки, потім знову записати результат виконання.

4. Будь-яке оновлення функціональностей застосунку займає більше часу. Через необхідність оновлювати велику кількість модульних тестів різних частин системи час виконання навіть простих задач збільшується.

5. Обробка кожної події, яка генерується через зміну даних в одному з зовнішніх джерел, займає більшу кількість часу через те, що порушуються принципи асинхронності. У результаті поточного підходу неможливо досягти оновлення даних у реальному часі. Кожна подія лише реєструється в межах першого етапу роботи застосунку, проте фактична обробка відбувається з затримкою через використання групового оновлення сервісами створення та відправки внутрішніх повідомлень. В результаті обробка кожної окремої події з моменту її фактичної генерації займає 10-15 хвилин.

6. Брокер RabbitMQ має працювати постійно, адже поява нових подій не є прогнозованою. У результаті аналізу даних по резервуванню ресурсів застосунку сервіс RabbitMQ займає 3 місце по неефективності, на що вказують дані зі статистики використання ресурсів на рисунку 16.

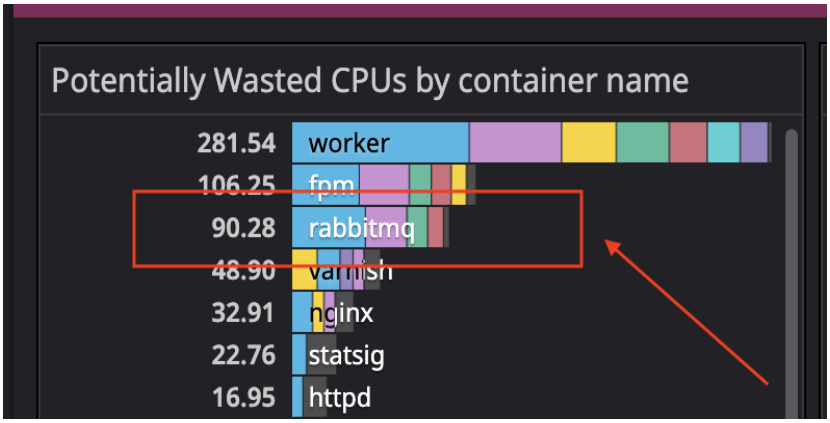


Рисунок 16 *Зарезервовані та невикористані ресурси сервісом RabbitMQ*

7. Багатоетапність обробки подій впливає на ризик виникнення нестабільності в стані внутрішніх бізнес-сутностей. Задля підтримки цілісності кожного об’єкту для фіксації зовнішньої події вводиться поняття

статусу обробки. Цей статус змінюється на кожному з етапів опрацювання (наприклад, “Зовнішня подія зареєстрована”, “Подія відправлена для обробки внутрішнім потоком”, “Подія успішно опрацьована” тощо), він має чітко відповідати поточному етапу в обробці. Через велику кількість етапів часто виникає порушення цілісності, що змушує ініціювати повторну обробки для події з початку процесу.

Іншим суттєвим недоліком поточної логіки обробки даних є складність в прогнозуванні поведінки застосунку при зміні частоти створення зовнішніх подій. Зміни в зовнішніх джерелах інформації майже повністю відсутні у вихідні дні, при цьому в першій половині дня в понеділок (або в перший день після інших неробочих днів) та в останній день перед вихідними і іншими неробочими днями кількість подій про зміну стану товарів в каталозі суттєво збільшується. Незважаючи на те, що ці тенденції можна розрахувати наперед та втілити в технічну специфікацію роботи застосунку, існує великий ризик того, що подібні зміни вплинуть на невідповідність між запланованою та фактичною кількістю необхідним ресурсів для роботи. У результаті подібні конфігураційні зміни можуть лише погіршити стан ресурсо-залежності застосунку.

Отримані результати аналізу роботи підсистеми оновлення каталогу вказують на необхідність пошуку варіантів оптимізації поточної логіки роботи застосунку.

4 ОПТИМІЗАЦІЯ ПІДСИСТЕМИ ОНОВЛЕННЯ КАТАЛОГУ

4.1 Базовий набір технологій та можливості оптимізації

Для проведення ефективного дослідження можливостей покращити стан застосунку оновлення каталогу потрібно враховувати деякі особливості та обмеження, відповідно до яких можливо профілювати та аналізувати подальші результати.

Перш за все, базовими технічними вимогами команди розробки є використання мови програмування PHP і каркасу Symfony, та умова застосування оптимізації до поточного застосунку. Також всі конфігураційні та технічні зміни мають бути придатними до контейнеризації за допомогою інструментарію Docker та бути готовими для розгортання в хмарному середовищі за допомогою інструментів оркестратора Kubernetes.

Використання Агрегатора каталогу (елемент 2 на рисунку 13) є зовнішньою технічною специфікацією, яка не може бути змінена у межах поточної задачі. Крім цього, повністю відсутня можливість впливати або контролювати наповнення даними теми Apache Kafka, інтенсивність генерації подій та конфігураційні налаштування генератора теми. В результаті дослідження наявної документації роботи цієї частини підсистеми було виявлено, що вона технічно оптимізована та використовує наявні механізми ефективної генерації та агрегації даних:

- Використовуються налаштування компактності теми [17], де для кожного ідентифікатора (що є товаром каталогу) зберігається лише остання подія, яка є завжди актуальним станом всього набору атрибутів.
- Тема розділена на 4 розділи. Завдяки цьому відбувається покращення ефективності створення та обробки даних на стороні генератора, при цьому є можливість використовувати до чотирьох одночасних асинхронних споживачів, кожен з яких буде обробляти окремий розділ або кілька розділів, якщо подібних користувачів буде менше [17].

Враховуючи вказане вище, єдиним варіантом впливу на частину роботи елементів Apache Kafka є збільшення кількості споживачів до 4 (на противагу використання лише одного споживача в поточній версії застосунку).

Другим варіантом технічної оптимізації роботи застосунку є аналіз доцільності використання RabbitMQ, як внутрішнього елемента забезпечення асинхронності обробки даних. RabbitMQ у порівнянні з Apache Kafka може гарантувати значно меншу затримку при обробці повідомлень [47]. Проте, додаткова технічна складність та необхідність супроводжувати ще один додатковий елемент системи вказують на те, що одночасне використання обох технологій в межах роботи одного застосунку є недоцільним ускладненням.

Третім методом покращення ефективності роботи застосунку є зміна конфігурації при обробці як зовнішніх подій Apache Kafka, так і внутрішніх повідомлень RabbitMQ. Зміна в налаштуваннях кожного з елементів підсистеми застосунку може вплинути на кількість подій, які застосунок може опрацьовувати протягом певного проміжку часу, та на необхідну кількість ресурсів, яку застосунок потребує при різній інтенсивності генерації зовнішніх подій.

Четвертим методом оптимізації роботи застосунку є зменшення кількості ресурсів, які резервуються без подальшого використання. На рисунку 17 вказано моніторингові дані зарезервованих ядер процесора (CPU) та оперативної пам’яті (RAM), які є надлишковими через відсутність їх використання на рівні управління оркестратора Kubernetes. Фактично це ресурси, які застосунок резервує у хмарному середовищі при розгортанні, проте не використовує під час обробки подій. Додаткові моніторингові дані засвідчили, що при низькій частоті генерації зовнішніх подій та в періоди повної відсутності нових подій від генератора теми Apache Kafka кількість подібних ресурсів збільшується. Додаткове дослідження ефективності використання ресурсів є необхідним для пошуку оптимізації поточної роботи застосунку.

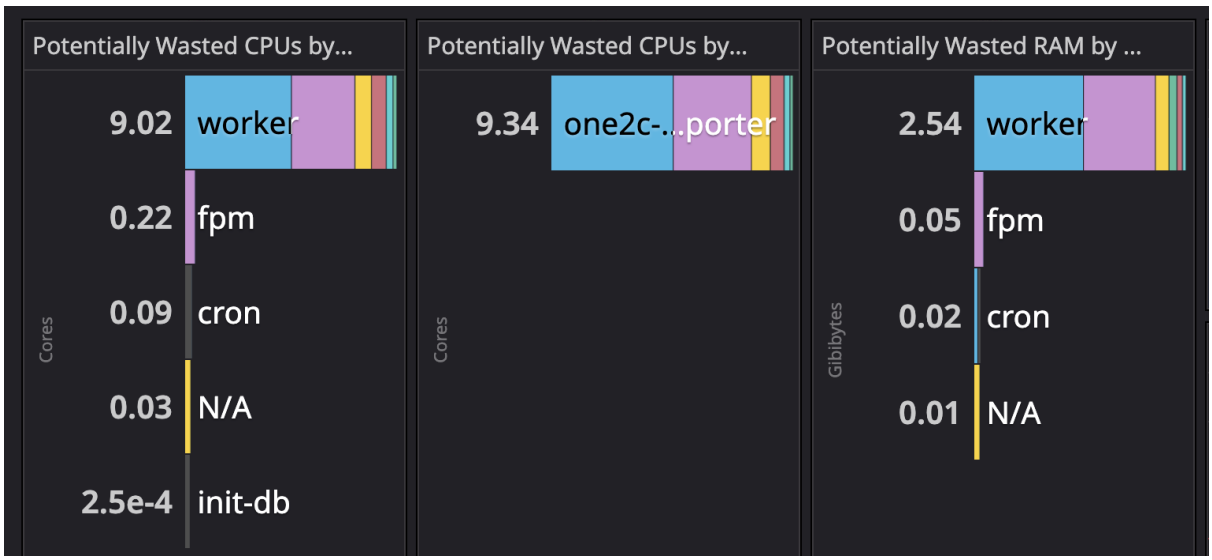


Рисунок 17 *Невикористані зарезервовані ресурси застосунку оновлення каталогу*

4.2 Зміна конфігурації елементів підсистеми

Одним з основних варіантів контролю кількості ресурсів при роботі застосунку та швидкості обробки зовнішніх подій без програмних змін є актуалізація конфігурації його функціональних елементів. У таблиці 3 представлено результати дослідження зміни конфігурації застосунку при його запуску у локальному середовищі (використовувалась робоча машина з загальною оперативною пам’яттю 32GB та 12 ядрами процесора) на кількість оброблених зовнішніх подій за один проміжок у 10 хвилин. Суть дослідження полягала у запуску споживання всієї теми Apache Kafka на чистий стан внутрішньої бази даних при різній кількості споживачів зовнішньої теми, внутрішніх повідомлень RabbitMQ, екземплярів споживачів RabbitMQ. При цьому, відбувався замір кількості зареєстрованих подій та повністю оброблених подій за один проміжковий цикл.

Таблиця 3 – Вплив зміни конфігурації на кількість оброблених подій

Тип конфігурації	Кількість Kafka	Налаштування генератора	Кількість екземплярів	Кількість оброблени

	споживачі в	**внутрішніх повідомлень**	**споживачі в RabbitMQ**	**х подій за 10 хвилин**
Безпечний (повільно)	1 (може зареєструвати до 4000 нових подій)	2000 (cron-job запускається кожні 10 хвилин)	3	2000
Стандартний (середньо)	2 (може зареєструвати до 8000 нових подій)	4000 (cron-job запускається кожні 10 хвилин)	5	4000
Ризикований (максимальна)	4 (може зареєструвати до 16000 нових подій)	6000 (cron-job запускається кожні 5 хвилин)	5	10000
Критичний	4 (може зареєструвати до 16000 нових подій)	8000 (cron-job запускається кожні 5 хвилин)	10	10000

У результаті проведеного дослідження можна зробити висновки по кожному з використаних типів конфігурації:

1. Безпечний – використовує найменшу кількість необхідних одночасно ресурсів. При цьому необхідна кількість ресурсів для частини RabbitMQ не залежить від оброблених даних. Елемент більшу частину часу знаходиться у стані простою. Пікова кількість ресурсів при обробці даних не перевищувала 4 GB оперативної пам’яті, 4 ядра процесора.

2. Стандартний – використовує середню кількість необхідних одночасно ресурсів. За рахунок збільшення вдвічі кількості споживачів теми Apache Kafka збільшено вдвічі кількість зареєстрованих подій за той самий проміжок часу. Як результат, загальна кількість оброблених подій прямо залежить від кількості генерації внутрішніх повідомлень RabbitMQ. При цьому, збільшення кількості споживачів повідомлень починає впливати на загальний результат

набагато менше. Пікова кількість ресурсів при обробці даних не перевищувала 6 GB оперативної пам’яті, 4 ядра процесора.

3. Ризикований – використовує максимально можливу кількість споживачів теми Apache Kafka (4). Генератор внутрішніх повідомлень RabbitMQ може створити 12 тисяч повідомлень протягом циклу в 10 хвилин (запускається раз на 5 хвилин, створює 6 тисяч повідомлень під час кожного запуску). При цьому, загальна кількість оброблених подій не перевищує 10 тисяч. Таким чином, при цьому типу конфігурації збільшення кількості екземплярів споживачів RabbitMQ перестає відігравати важливу роль – фізична кількість опрацьованих даних досягає своєї фактичної межі можливої кількості записів в зовнішню базу даних. При цьому типі запуску не відбувається простоювання ресурсів RabbitMQ, проте існує зворотній ризик того, що створені повідомлення в системі RabbitMQ не будуть оброблятись за один 5-хвилинний цикл. Як результат, буде відбуватись складання повідомлень на стороні брокера RabbitMQ. Це не має бути критичним, адже тема має свої ліміти, тому з часом всі повідомлення будуть оброблені. Проте, подібне використання ресурсів є неоптимальним. Альтернативний варіант – штучно зменшити кількість зареєстрованих подій для того, щоб встигати обробляти все, що було зареєстровано. Збільшення часу затримки обробки подій для зовнішньої системи генератора Apache Kafka є очікуваною поведінкою, тому є набагато більш бажаним станом застосунку. З точки зору використання ресурсів пік при обробці даних не перевищував 8 GB оперативної пам’яті, 6 ядер процесора. Проте, потрібно також брати до уваги місце фізичної пам’яті для зберігання необроблених повідомлень RabbitMQ. Через вказану вище особливість розробки кількість необхідного місця буде збільшуватись пропорційно.

4. Критичний – використовує максимальну кількість необхідних одночасно ресурсів. При цьому, незважаючи на збільшення кількості згенерованих внутрішніх повідомлень для RabbitMQ – 16 тисяч протягом циклу 10 хвилин (запускається раз на 5 хвилин, створює 8 тисяч повідомлень

при кожному запуску), загальна кількість оброблених подій не змінюється у порівнянні з попереднім типом конфігурації. Збільшення вдвічі кількості екземплярів споживачів внутрішніх повідомлень також ніяк не вплинуло на ці дані. Проте, подібний тип конфігурації має низку негативних показників, які вказують на те, що він є абсолютно недієздатним у довгостроковій перспективі. По-перше, кількість споживачів повідомлень RabbitMQ створює загрозу наявності елементів системи у стані простою. Враховуючи отримані результати з інших типів конфігурації, оптимальною кількістю є 3-5, отже майже половина екземплярів при подібному запуску будуть мати зарезервовані ресурси, проте вони ніколи не будуть задіяні. По-друге, занадто велика кількість створених внутрішніх повідомлень впливає на те, що буде збільшено кількість необроблених повідомлень на стороні брокера RabbitMQ, тому ризики, про які було докладно вказано для типу конфігурації “Ризикований”, становлять ще більшу загрозу тут. Наостанок, в даному випадку пікова кількість використовуваних ресурсів сягнула 9 GB оперативної пам’яті, 7 ядер процесора.

У результаті проведеного дослідження зміни конфігурації було підтверджено, що ліміти поточного стану роботи застосунку є найбільш значущими саме у частині RabbitMQ. Отже, найбільш ефективним варіантом покращення ефективності роботи застосунку та зменшення необхідних ресурсів (у тому числі ресурсів у незадіяному зарезервованому стані) є видалення внутрішньої системи обробки повідомлень RabbitMQ.

4.3 Оновлений принцип роботи підсистеми актуалізації каталогу

Оптимізована спрощена схема роботи застосунку актуалізації каталогу веб-сайту представлена на рисунку 18.

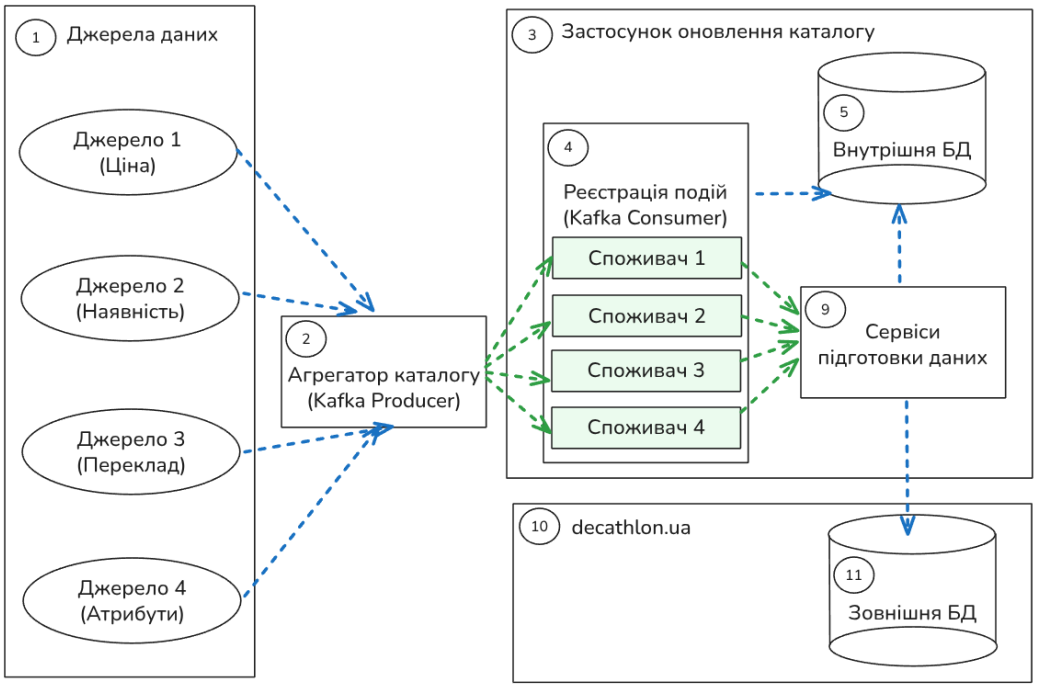


Рисунок 18 *Оновлена схема елементів застосунку актуалізації каталогу без використання шару RabbitMQ*

Спрощена логіка включає в себе повне видалення системних елементів 6-8 шару RabbitMQ у порівнянні зі схемою на рисунку 13. Як результат, при оновленій логіці роботи були змінені елементи та функціональні задачі:

- 4 – Реєстрація подій оновлення каталогу – крім споживання теми Apache Kafka, основною задачею тепер є фактична обробка зовнішніх подій через сервіси трансформації (9) для збереження даних для товару каталогу в зовнішню базу даних (11). Додатково було виконано масштабування кількості споживачів теми Apache Kafka до 4 для можливості асинхронної обробки всіх 4 розділів теми окремими екземплярами. Подібний підхід має зробити споживання подій теми більш ефективним та збільшити швидкість оброблюваних даних за певний проміжок часу.
- 9 – Сервіси підготовки даних будуть викликатись кожним зі споживачів теми при обробці кожної окремої події. Дані нової події порівнюються з попередньо збереженими даними з бази даних застосунку (5). Як результат, кожна нова подія буде ініціювати оновлення даних у реальному часі

відповідно до змінених атрибутів у порівнянні з попереднім станом даних для відповідного товару.

Реалізація програмних змін в поточній логіці застосунку включає декілька основних частин:

1. Видалення змінних середовища для конфігурації черг повідомлень.
2. Видалення залежностей пакетів бібліотеки Enqueue Bundle [48] у менеджері пакетів composer PHP.
3. Видалення конфігураційних файлів для пакетів бібліотеки Enqueue Bundle та актуалізація конфігурації застосунку.
4. Зміна типу конфігурації диспетчера внутрішніх подій Symfony для синхронності обробки.
5. Зміна логіки роботи основної кодової бази всіх шарів застосунку – Domain, Application, Infrastructure. Додатково було реалізовано зміни у модульних тестах для покриття оновлень у кодовій базі.

У результаті реалізації оновленої логіки було змінено 49 файлів застосунку, додано 1.2 тисячі рядків коду, видалено 3.8 тисячі рядків коду, про що вказують дані на рисунку 19.

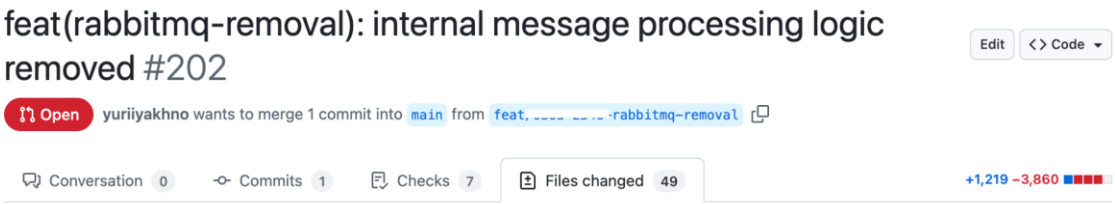


Рисунок 19 *Загальний стан запропонованих змін технічної реалізації*

Базова програмна частина застосунку після реалізованих змін складається з основних функціональних елементів для виконання покрокової логіки споживання та обробки подій Apache Kafka:

1. Базовий сервіс, який ініціює створення екземпляра споживача Apache Kafka, викликає допоміжні сервіси для трансформації отриманого повідомлення та його подальшої реєстрації продемонстровано у лістингу 1.

Лістинг 1 – Базовий сервіс споживання події Apache Kafka

```
final class One2CConsumerService implements One2CConsumerServiceInterface
{
  public function __construct(
     private readonly OneServicesKafkaConsumer $consumer,
     private readonly InitializeDataExporterConfigurationServiceInterface $initializeConfigurationService,
     private readonly EventBusMessageToArrayTransformerInterface $eventBusMessageToArrayTransformer,
     private readonly One2CEventDTOFactoryInterface $one2CEventDTOFactory,
     private readonly RegisterOne2CEventServiceInterface $registerOne2CEventService,
     private readonly KafkaConfigurationDTOFactoryInterface $kafkaConfigurationFactory,
  ) {
  }

  /**
   * {@inheritDoc}
   */
  public function execute(KafkaTopicType $topicType): void
  {
     try {
        $dataExporterConfiguration = $this->initializeConfigurationService->execute();
        $topics =
$dataExporterConfiguration->getLocalesConfiguration()->getKafkaTopicNameListByType($topicType);
        $kafkaConfiguration = $this->kafkaConfigurationFactory->create($topicType);

        $this->consumer->init(
           $topics,
           $kafkaConfiguration->getGroupOffset(),
           $kafkaConfiguration->getGroupId()
        );

        $message = $this->consumer->consume();
        $messageData = $this->eventBusMessageToArrayTransformer->transform($message);
        $driverMessage = $message->getDriverMessage();

        $one2CEventDTO = $this->one2CEventDTOFactory->create($messageData, $driverMessage, $topicType);
        $this->registerOne2CEventService->execute($one2CEventDTO, $dataExporterConfiguration);

        $this->consumer->commit($message);
     } catch (Exception|SchemaRegistryException $exception) {
        throw new KafkaConsumerException($exception->getMessage(), $exception->getCode(), $exception);
```

```
    }
  }
}
```

2. Сервіс реєстрації нової події та організації логіки персистентності для основних сутностей застосунку представлено у лістингу 2.

Лістинг 2 – Сервіс реєстрації нової події

```
final class RegisterOne2CEventService implements RegisterOne2CEventServiceInterface
{
  public function __construct(
    private readonly EntityManagerInterface $entityManager,
    private readonly ProductOne2CEventReportFactoryInterface $productOne2CEventReportFactory,
    private readonly MessageDataEventProcessingServiceInterface $messageDataEventProcessingService,
  ) {
  }

  /**
   * {@inheritDoc}
   */
  public function execute(One2CEventDTO $one2CEventDTO, DataExporterConfiguration
$dataExporterConfiguration): void
  {
    try {
      $productOne2CEventReport = $this->productOne2CEventReportFactory->create($one2CEventDTO);
      $this->entityManager->persist($productOne2CEventReport);

      /** @var ProductOne2CStateReportRepository $stateReportRepository */
      $stateReportRepository = $this->entityManager->getRepository(
        ProductOne2CStateReport::class
      );

      $productOne2CStateReport =
$stateReportRepository->findOrCreateByProductSkuAndTopicNameAndTopicType(
        $one2CEventDTO->getMessageKey(),
        $one2CEventDTO->getTopicName(),
        $one2CEventDTO->getTopicType()
      );

      $productOne2CStateReport->setIsUpdateRequired(true);
```

```
            $productOne2CStateReport->setUpdateState(ProductStateReportUpdateState::PROCESSING->name);
            $this->entityManager->persist($productOne2CStateReport);

            $this->messageDataEventProcessingService->execute(
                $productOne2CEventReport,
                $productOne2CStateReport,
                $dataExporterConfiguration,
            );

            $this->entityManager->persist($productOne2CStateReport);
            $this->entityManager->flush();
        } catch (Throwable $exception) {
            throw new One2CEventWasNotRegistered($exception->getMessage(), $exception);
        }
    }
}
```

3. Сервіс перетворення даних зовнішньої події Apache Kafka у внутрішню подію Symfony продемонстровано у лістингу 3.

Лістинг 3 – Сервіс перетворення даних у внутрішню подію Symfony

```
final class MessageDataEventProcessingService implements MessageDataEventProcessingServiceInterface
{
    public function __construct(
        private readonly DecoderInterface $decoder,
        private readonly ProductDataFactoryInterface $productDataFactory,
        private readonly ProductDataUpdatedEventFactoryInterface $productDataUpdatedEventFactory,
        private readonly EventDispatcherInterface $dispatcher,
        private readonly ProductOne2CPriceDataFactoryInterface $productOne2CPriceDataFactory,
    ) {
    }

    /**
     * {@inheritDoc}
     */
    public function execute(
        ProductOne2CEventReport $productOne2CEventReport,
        ProductOne2CStateReport $productOne2CStateReport,
        DataExporterConfiguration $dataExporterConfiguration,
    ): void {
```

```
        $messageLocaleConfiguration = $dataExporterConfiguration
            ->getLocalesConfiguration()
            ->getLocaleConfigurationByTopicNameAndTopicType(
                $productOne2CEventReport->getTopicName(),
                $productOne2CStateReport->getTopicType(),
            );
        $messageData = $this->decoder->decode($productOne2CEventReport->getRawMessage(),
JsonEncoder::FORMAT);

        $productData = $this->productDataFactory->create($messageData);
        $productOne2CPriceData = $this->productOne2CPriceDataFactory->create($productOne2CStateReport,
$productData);

        $event = $this->productDataUpdatedEventFactory->create(
            $productData,
            $productOne2CStateReport,
            $productOne2CEventReport,
            $messageLocaleConfiguration,
            $dataExporterConfiguration->getCountryZones(),
            $productOne2CPriceData ? $productOne2CPriceData->getCurrentPriceData() : [],
            $productOne2CPriceData ? $productOne2CPriceData->getFullPriceData() : [],
            $productOne2CPriceData?->getCessionSalePrice(),
            $productOne2CPriceData?->getVatRate(),
        );

        $this->dispatcher->dispatch($event);
    }
}
```

4. Програмний клас одного з підписників на внутрішню подію Symfony для виклику сервісів перетворення та персистентності даних у зовнішню базу даних веб-сайту decathlon.ua представлено у лістингу 4.

Лістинг 4 – Підписник на внутрішню подію оновлення товару каталогу

```
final class ETLProductDataUpdatedEventHandler implements EventSubscriberInterface
{
    public function __construct(
        private readonly DatabaseConnectionFactoryInterface $databaseConnectionFactory,
        /** @var iterable<UpdateETLDataByConnectionAndEventServiceInterface> */
        private readonly iterable $etlUpdateServices,
        private readonly EntityManagerInterface $entityManager,
        private readonly PersistFailedOne2CStateReportServiceInterface $failedOne2CEventReportPersistService,
```

```
    ) {
    }

    #[ArrayShape([ProductDataUpdatedEvent::class => 'array[]'])]
    public static function getSubscribedEvents(): array
    {
        return [
            ProductDataUpdatedEvent::class => [
                ['onProductDataUpdatedEvent', 50],
            ],
        ];
    }

    /**
     * @param ProductDataUpdatedEvent $event
     */
    public function onProductDataUpdatedEvent(ProductDataUpdatedEvent $event): void
    {
        $one2CStateReport = $event->getProductOne2CStateReport();

        try {
            $connection = $this->databaseConnectionFactory->create();

            /** @var UpdateETLDataByConnectionAndEventServiceInterface $etlUpdateService */
            foreach ($this->etlUpdateServices as $etlUpdateService) {
                $etlUpdateService->execute($connection, $event);
            }

            $connection->close();
            $one2CStateReport->setEtlLastUpdatedAt(new DateTime());
            $this->entityManager->persist($one2CStateReport);
        } catch (Throwable $exception) {
            if (isset($connection) && $connection->isConnected()) {
                $connection->close();
            }

            $this->failedOne2CEventReportPersistService->execute(
                $one2CStateReport,
                $event->getProductOne2CEventReport()->getId(),
                $exception->getMessage(),
            );

            $event->stopPropagation();
        }
    }
}
```

4.4 Оптимізація роботи застосунку у режимі пасивної роботи

Питання використання ресурсів застосунком при відсутності нових подій є одним з найбільш важливих при пошуку методів оптимізації.

Використання стандартної логіки створення копій контейнерів оркестратора Kubernetes [49] може вирішити питання лише частково, адже подібний підхід не дозволяє підтримувати повноцінну динамічність створення та знищення подібних копій.

Автомасштабувальник KEDA [50] дозволяє налаштувати розгортання та знищення екземплярів застосунку в залежності від стану зовнішніх факторів. У випадку застосунку оновлення каталогу подібних фактором може бути кількість активних зовнішніх подій Apache Kafka, які перебувають в певний момент часу в черзі на опрацювання.

На рисунку 20 представлено конфігурацію KEDA для застосунку актуалізації стану каталогу Декатлон Україна.

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
spec:
  pollingInterval: 15
  cooldownPeriod: 60
  maxReplicaCount: 4
  minReplicaCount: 0
  triggers:
    - type: kafka
      metadata:
        tls: enable
        unsafeSsl: "true"
        bootstrapServers: ...
        topic: ...
        activationLagThreshold: "100"
        lagThreshold: "1000"
        consumerGroup: ...
        limitToPartitionsWithLag: "true"
```

Рисунок 20 *Конфігурація KEDA автомасштабування екземплярів споживачів подій Apache Kafka*

Основні деталі конфігурації:

- minReplicaCount – стан контейнера за замовчуванням; фактично 0 в даному полі означає, що жоден з контейнерів не буде створений при

початковому стані розгортання, тобто повністю відсутнє будь-яке резервування чи фактичне використання ресурсів;

- maxReplicaCount – максимально можлива кількість копій контейнера; в даному випадку значення 4 означає, що за наявності великої кількості активних подій для опрацювання одночасно може працювати 4 екземпляри споживача теми Apache Kafka, що відповідає максимальній кількості доступних розділів в темі;
- triggers – група атрибутів, які зумовлюють зміну налаштування екземплярів контейнера для споживання теми; в даному випадку тип kafka вказує на те, що сервіс KEDA буде використовувати стан споживання теми застосунком та регулювати кількість екземплярів для запуску відповідно;
- activationLagThreshold – мінімальна кількість подій, яка може ініціювати створення першого екземпляра споживача;
- lagThreshold – кількість подій, яка ініціює створення додаткового екземпляра для споживання теми; значення контролює, скільки фактичних споживачів має працювати одночасно (наприклад, у випадку створення 4000 подій через генератор Kafka подібне налаштування ініціює створення 4 активних екземплярів для роботи).

Вказані деталі конфігураційного налаштування автомасштабувальника KEDA дозволяють досягти майже повної відсутності використання ресурсів застосунком актуалізації каталогу у пасивному стані. Жоден екземпляр споживача теми Apache Kafka не є розгорнутим у початковому стані. У випадку відсутності активних подій для опрацювання застосунок не потребує ресурсів. Як тільки зовнішні джерела інформації ініціюють створення нових 100 подій на стороні генератора перший екземпляр споживача буде автоматично створено автомасштабувальником. У випадку наявності невеликої кількості подій для опрацювання один споживач опрацює всі наявні та буде знищений. У випадку, якщо кількість активних подій перевищить 1000, буде розгорнутий додатковий споживач. При цьому, відбудеться автоматичний перерозподіл наявних 4 розділів споживання між активними

споживачами. За подібної логіки максимум 4 споживачі (для кожного з розділів) будуть створені лише за наявності згенерованих 4000 подій для опрацювання. При цьому, вони будуть активними лише до завершення обробки даних відповідного розділу, після чого відповідний екземпляр буде знищено автоматично.

Запропонована логіка використання автомасштабувальника KEDA має потенційний ризик, який необхідно узгоджувати з вимогами бізнесу. Значення показника мінімальної кількості активних подій для розгортання першого екземпляра споживача фактично дозволяє значно заощаджувати ресурси для створення та знищення відповідного операційного елемента Kubernetes, проте воно також вказує на наявність зони ризику у випадку створення недостатньої кількості подій для активації. У даному випадку за наявності 99 активних подій для опрацювання жоден з екземплярів не буде розгорнутий, тобто ця кількість подій буде і надалі у списку на опрацювання до моменту створення події номер 100. Зменшення показника буде відповідно ініціювати створення та знищення нових екземплярів зі значно більшою частотою, що призведе до зростання необхідної кількості ресурсів. Значення у 100 подій було обрано у результаті аналізу частоти створення подій протягом 4 тижнів з урахуванням трендів для вихідних та робочих днів. У випадку активної зміни характеристик товарів кількість згенерованих подій на стороні Apache Kafka перевищує 130 за проміжок у кілька хвилин, тобто значення вказаного ліміту для запуску застосунку буде досягатись швидко. Періоди низької активності актуалізації стану товарів на стороні зовнішніх джерел даних (наприклад, у вихідні та святкові дні) зазвичай характеризуються повною відсутністю генерації нових подій.

4.5 Тестування оновленої логіки підсистеми оновлення каталогу

У результаті виконання технічних та конфігураційних змін роботи застосунку було досягнуто декількох вагомих результатів, які були

підтверджені при локальному тестуванні та після застосування оновлень в реальному середовищі:

1. Видалення всього шару внутрішньої обробки повідомлень з використанням технології RabbitMQ дозволило зменшити кількість зовнішніх залежностей застосунку, загальну кількість програмного коду та складність технічної реалізації в цілому. Кількість необхідних екземплярів елементів застосунку було зменшено на 5 одиниць за рахунок видалення окремого елементу брокера RabbitMQ, 3 асинхронних споживачів черги повідомлень RabbitMQ, сервісу періодичної перевірки та створення повідомлень відповідно до зареєстрованих зовнішніх подій Apache Kafka. У результаті було досягнуто зменшення необхідної зарезервованої кількості ресурсів для застосунку. Додатковою перевагою видалення цієї програмної частини є виправлення концепції обробки зовнішніх даних у реальному часі. За рахунок відмови від використання додаткового шару асинхронної обробки всі зовнішні події від генератора Apache Kafka обробляються та записуються в зовнішню базу даних веб-сайту одразу при їх реєстрації та без використання багатоетапності.

2. Тестування оновленої логіки обробки подій у локальному середовищі при повному споживанні теми Apache Kafka на чисту внутрішню базу даних показало результат опрацювання 16000 зовнішніх подій (де кожен з 4 екземплярів споживачів опрацьовував в середньому 4000 подій) за проміжок 10 хвилин. Таким чином, час обробки однієї події скоротився в середньому в 11 разів, кожне опрацювання в середньому займало 0.15 секунд. Апробування впровадженої логіки в реальному середовищі довело покращення результату опрацювання: при максимальній частоті генерації зовнішніх подій теми кожен зі споживачів опрацьовував в середньому 5800 подій за 10 хвилин. Таким чином, кожна окрема подія була опрацьована в середньому протягом 0.11 секунд. Подібні результати свідчать про досягнення стану актуалізації оновлення каталогу в режимі реального часу без суттєвих затримок.

3. Під час локального тестування не було помічено значного збільшення навантаження на зовнішню базу даних під час повного споживання теми

всього каталогу. В реальному середовищі стан зовнішньої бази даних контролювався в системі моніторингу DataDog [51]. Під час обробки всієї теми каталогу було зафіксовано збільшення кількості запитів до бази даних від застосунку на рівні 17%, при цьому кількість необхідних ядер процесора (CPU) для роботи на рівні бази даних була збільшена на 8%. Подібний стан споживання ресурсів було визнано прийнятним у стані активної зміни каталогу, вказані показники не виходили за межі норми та лімітів для використання ресурсів в хмарному середовищі екземпляром бази даних.

4. Завдяки використанню автомасштабувальника KEDA було зменшено кількість зарезервованих ресурсів, які не використовуються застосунком при роботі, приблизно в 2.5 разів (дані використання ресурсів після впровадження змін вказано на рисунку 21 у порівнянні з даними на рисунку 17).

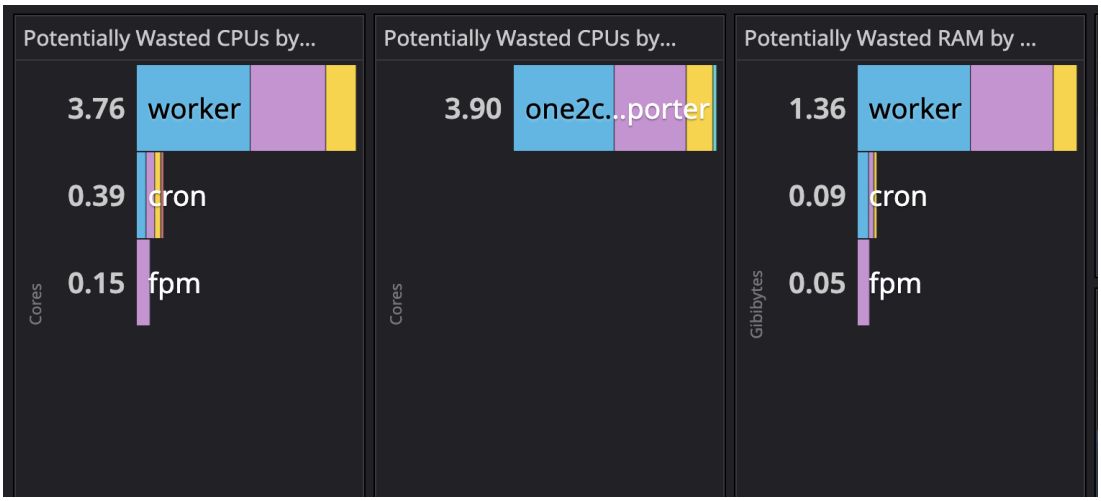


Рисунок 21 *Зарезервовані та невикористані ресурси застосунком з автомасштабуванням KEDA*

4.6 Висновки з розділу 4

Для дослідження особливостей роботи систем з подійно-орієнтованою архітектурою було проаналізовано застосунок оновлення каталогу товарів підсистеми Декатлон Україна [36]. Оновлення каталогу веб-сайту у реальному часі є однією з основних задач технічної команди для забезпечення наявності

актуальної інформації для користувачів відповідно до даних зі сторони зовнішніх джерел формування цін, наявності, перекладів та деталей всіх товарів.

У результаті дослідження, аналізу та технічного профілювання поточного застосунку було зроблено висновок про те, що технічна реалізація мала в собі ряд недоліків, який полягав в першу чергу у використанні в межах одного застосунку декількох технологій роботи з асинхронними багатопотоковими даними одночасно: Apache Kafka для споживання зовнішніх подій та RabbitMQ для асинхронної обробки внутрішніх повідомлень. Також було виявлено надмірність у використанні ресурсів, порушення загальної концепції оновлення даних у реальному часі, загальну неефективність застосунку у стані пасивної роботи.

Для зменшення кількості необхідних ресурсів для досягнення мети оновлення даних каталогу у реальному часі, полегшення технічної складності застосунку, зменшення кількості структурних елементів було розглянуто декілька варіантів оптимізації поточної логіки роботи:

- зміна характеристик споживання розділів теми Apache Kafka,
- зміна конфігурації роботи елементів застосунку для збільшення швидкості обробки подій та ефективного використання ресурсів,
- видалення частини застосунку по роботі з RabbitMQ,
- зменшення використання застосунком ресурсів при низькій частоті генерації зовнішніх подій Apache Kafka або при їх повній відсутності.

У результаті апробування різних варіацій оптимізації та їхніх комбінацій було запропоновано наступні оновлення (технічні та конфігураційні):

1. Збільшення кількості споживачів для зовнішньої теми Apache Kafka з 1 до 4 (відповідно до наявних розділів у темі) задля збільшення кількості одночасно оброблюваних подій.

2. Систему внутрішньої системи обробки повідомлень та технічну реалізацію використання RabbitMQ було повністю видалено з застосунку.

3. Впровадження конфігурації автомасштабувальника KEDA в межах загальної Kubernetes оркестрації контейнерів дозволило оптимізувати необхідну кількість ресурсів, які потрібні застосунку при обробці зовнішніх подій.

Завдяки впровадженим змінам було досягнуто значних покращень роботи застосунку:

1. Відновлення застосування концепції актуалізації каталогу товарів у режимі реального часу.

2. Відмова від використання багатоетапності при роботі з подіями та шару додаткової асинхронної обробки внутрішніх повідомлень RabbitMQ, що дозволило досягти значного зменшення часу роботи застосунку та необхідних ресурсів для опрацювання подій. Додатковою перевагою стало полегшення технічної та архітектурної логіки застосунку.

3. Скорочення часу на обробку однієї події в середньому в 11 разів в порівнянні з попереднім станом.

4. Оптимізація використанні ресурсів дозволила зменшити кількість зарезервованих невикористаних ресурсів приблизно у 2.5 разів.

ВИСНОВКИ

1. Досліджено методи використання засобів подійно-орієнтованої архітектури у мікросервісних системах для гнучкого налаштування комунікації між елементами.

2. Розглянуто можливості регулювання рівня приховування інформації, зв'язності зон відповідальності та міжсервісної взаємозалежності при використанні двох підходів комунікації: синхронної для тісно пов'язаних елементів та асинхронної для абсолютно незалежних елементів.

3. Досліджено наявні технологічні рішення та підходи при роботі з подіями для нівелювання ризику паралельного створення даних або їх дублювання, гарантування стану цілісності сутностей при роботі різних кількостей генераторів та споживачів подій, гарантування надійності в будь-який момент обробки даних, ефективної обробки помилок та використання ресурсів.

4. Досліджено поточний застосунок актуалізації каталогу товарів підсистеми Декатлон Україна, виділено недоліки технічної реалізації, запропоновано методи оптимізації роботи при активній фазі обробки подій з високою частотою створення та пасивній фазі при відсутності нових подій теми Apache Kafka.

5. Апробовано варіації програмних та конфігураційних змін для досягнення стану застосування оновлень каталогу товарів у режимі реального часу з використанням мінімальної кількості ресурсів.

6. Розроблено технічне та конфігураційне рішення без використання багатоетапної обробки подій та шару додаткової асинхронної обробки внутрішніх повідомлень RabbitMQ для полегшення технічної та архітектурної логіки застосунку, зменшення часу роботи та необхідних ресурсів при обробці зовнішніх подій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Яхно Ю., Коломоєць Г.П. Дослідження засобів побудови програмних систем з подійно-орієнтованою архітектурою. Збірник наукових праць студентів, здобувачів PhD, докторантів і молодих вчених “Молода наука-2025” / Запорізький національний університет. Запоріжжя : ЗНУ, 2025. Т.5. С. 427-428.

2. Яхно Ю., Коломоєць Г.П. Використання засобів подійно-орієнтованої архітектури в мікросервісних системах. Матеріали II Міжнародної науково-практичної конференції Інженерного навчально-наукового інституту ім. Ю. М. Потебні “Інженерні інновації та розбудова національної економіки”. Одеса : Видавничий дім “Гельветика”, 2025. С. 506-508.

3. Коломоєць Г.П., Яхно Ю. Використання засобів подійно-орієнтованої архітектури для оптимізації механізму оновлення каталогу товарів. Науковий журнал “Information Technology: Computer Science, Software Engineering and Cyber Security”, випуск 4. Одеса : Видавничий дім “Гельветика”, 2025.

4. Adam Bellemare. Building Event-Driven Microservices. Sebastopol (USA) : O'Reilly Media Inc., 2020. ISBN: 9781492057895.

5. Sam Newman. Building Microservices, 2nd Edition. Sebastopol (USA) : O'Reilly Media Inc., 2021. ISBN: 9781492034025.

6. Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. London : Addison-Wesley Professional, 2003. ISBN: 9780321125217.

7. Martin Fowler. Bounded Context. URL : https://martinfowler.com/bliki/BoundedContext.html (дата звернення: 28.03.2025).

8. Vaughn Vernon. Implementing Domain-Driven Design. London : Addison-Wesley Professional, 2013. ISBN: 9780133039900.

9. Ronnie Mitra, Irakli Nadareishvili. Microservices: Up and Running. Sebastopol (USA) : O'Reilly Media Inc., 2020. ISBN: 9781492075455.

10.Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd Edition. Boston : Addison-Wesley Professional, 2003. ISBN: 9780321193681.

11.James Lewis, Martin Fowler. Microservices. URL : https://martinfowler.com/articles/microservices.html (дата звернення: 29.03.2025).

12.Melvin Conway. Conway's law. URL : https://www.melconway.com/Home/Conways_Law.html (дата звернення: 19.04.2025).

13.Cesar de la Torre, Bill Wagner, Mike Rousos. .NET Microservices: Architecture for Containerized .NET Applications. Redmond : Microsoft Developer Division, 2023.

14.Bhavesh R Kashikar. Interservice Communication for Microservices. URL : https://www.einfochips.com/blog/interservice-communication-for-microservices/ (дата звернення: 29.03.2025).

15.Martin Fowler. What do you mean by “Event-Driven”? URL : https://martinfowler.com/articles/201701-event-driven.html (дата звернення: 29.03.2025).

16.RabbitMQ Documentation. URL: https://www.rabbitmq.com/docs (дата звернення: 29.03.2025).

17.Kafka Documentation. URL : https://kafka.apache.org/documentation/ (дата звернення: 29.03.2025).

18.Compare AWS and Azure services to Google Cloud. URL : https://cloud.google.com/docs/get-started/aws-azure-gcp-service-comparison (дата звернення: 29.03.2025).

19.Comparing Message Queues and Message Brokers: Understanding the Differences. URL : https://medium.com/memphis-dev/comparing-message-queues-and-message-brokers-understanding-the-differences-dc776f0c3002 (дата звернення: 29.03.2025).

20.Gregor Hohpe, Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. London : Addison-Wesley Professional, 2003. ISBN: 9780321200686.

21.Martin Fowler. Event Sourcing. URL : https://martinfowler.com/eaaDev/EventSourcing.html (дата звернення: 22.04.2025).

22.Martin Fowler. CQRS. URL : https://martinfowler.com/bliki/CQRS.html (дата звернення: 22.04.2025).

23.Denat Hoxha. Managing Version Dependencies Between Microservices. URL : https://medium.com/@denhox/managing-version-dependencies-between-microservices-648d1d8dd4ca (дата звернення: 27.04.2025).

24.10 Methods to Ensure Data Consistency in Microservices. URL : https://daily.dev/blog/10-methods-to-ensure-data-consistency-in-microservices (дата звернення: 27.04.2025).

25.Haerder, T., Reuter, A. Principles of transaction-oriented database recovery. URL : https://archive.org/details/sim_acm-computing-surveys_1983-12_15_4/page/286/mode/2up (дата звернення: 27.04.2025).

26.Hugo Filipe, Oliveira Rocha. Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices. New York : Apress, 2021. ISBN: 9781484274682.

27.Hector Garcia-Molina, Kenneth Salem. “Sagas”. URL : https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf (дата звернення: 27.04.2025).

28.Denat Hoxha. Sharing Data Between Microservices. URL : https://medium.com/@denhox/sharing-data-between-microservices-fe7fb9471208 (дата звернення: 27.04.2025).

29.Chris Richardson. Pattern: Saga. URL : https://microservices.io/patterns/data/saga.html (дата звернення: 29.04.2025).

30.Saga distributed transactions pattern. URL : https://learn.microsoft.com/en-us/azure/architecture/patterns/saga (дата звернення: 29.04.2025).

31.Susan J. Fowler. Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization, 1st Edition. Sebastopol (USA) : O'Reilly Media Inc., 2016. ISBN: 9781491965979.

32.Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. ACM Transactions on Database Systems, 1990. DOI:10.1145/77643.77647.

33.Szymon Pobiega. Exactly-once message delivery. URL : https://exactly-once.github.io/posts/exactly-once-delivery/ (дата звернення: 30.04.2025).

34.Matt Welsh, David Culler, Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. SOSP '01 Proceedings of the eighteenth ACM symposium on Operating systems principles, 1st Edition. Association for Computing Machinery, 2001. ISBN: 1581133898.

35.Bhanu Kishore Diddi. What is Staged Event Driven Architecture. URL : https://medium.com/@bhanukishore22/what-is-staged-event-driven-architecture-edd4f39a102c (дата звернення: 29.04.2025).

36.Декатлон Україна. URL: https://www.decathlon.ua.

37.Must Know Omnichannel Statistics For Marketers (2025). URL : https://www.uniformmarket.com/statistics/omnichannel-shopping-statistics (дата звернення: 07.06.2025).

38.52+ Omnichannel Stats You Can’t Afford to Ignore in 2024 (Or Risk Losing Customers!). URL : https://firework.com/blog/omnichannel-statistics (дата звернення: 07.06.2025).

39.Grace Baldwin. Everything You Need to Know About the ROPO Effect. URL : https://www.omniaretail.com/blog/everything-you-need-to-know-about-the-ropo-effect (дата звернення: 07.06.2025).

40.Daemon – Linux manual page. URL : https://man7.org/linux/man-pages/man7/daemon.7.html (дата звернення: 08.06.2025).

41.Docker - Reference documentation. URL : https://docs.docker.com/reference/ (дата звернення: 08.06.2025).

42.MongoDB Documentation. URL : https://www.mongodb.com/docs/ (дата звернення: 08.06.2025).

43.PostgreSQL 16.9 Documentation. URL : https://www.postgresql.org/docs/16/index.html (дата звернення: 08.06.2025).

44.Symfony Console Commands Documentation. URL : https://symfony.com/doc/current/console.html (дата звернення: 08.06.2025).

45.Symfony Events and Event Listeners. URL : https://symfony.com/doc/current/event_dispatcher.html (дата звернення: 08.06.2025).

46.Martin Fowler. Data Transfer Object. URL : https://martinfowler.com/eaaCatalog/dataTransferObject.html (дата звернення: 13.07.2025).

47.Benchmarking Apache Pulsar, Kafka, and RabbitMQ. URL : https://www.confluent.io/blog/kafka-fastest-messaging-system/ (дата звернення: 14.11.2025).

48.Enqueue Dev package documentation. URL : https://php-enqueue.github.io/#documentation (дата звернення: 15.11.2025).

49.Kubernetes ReplicaSet Documentation. URL : https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (дата звернення: 15.11.2025).

50.Kubernetes KEDA Concepts. URL : https://keda.sh/docs/2.18/concepts/ (дата звернення: 15.11.2025).

51.DataDog documentation. URL : https://docs.datadoghq.com/ (дата звернення: 15.11.2025).

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Яхно Юрій Анатолійович, студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальності 121 «Інженерія програмного забезпечення», адреса електронної пошти:ipz23bd-112@stu.zsea.edu.ua, підтверджую, що написана мною кваліфікаційна робота на тему «**Дослідження засобів побудови програмних систем з подійно-орієнтованою архітектурою»**

відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст. 42 Закону України «Про освіту», зі змістом яких ознайомлений.

Заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії.

Також згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-систем, а також на архівування моєї роботи в базі даних цієї системи.

Дата: 30.12.2025 р. Яхно Юрій Анатолійович
(студент)

Дата: 30.12.2025 р. Коломоєць Геннадій Павлович
(науковий керівник)