Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

РОЗРОБКА ТА ДОСЛІДЖЕННЯ СИСТЕМИ НА БАЗІ ПОДІЙНО-ОРІЄНТОВАНОЇ АРХІТЕКТУРИ

DEVELOPMENT AND RESEARCH OF A SYSTEM BASED ON EVENT-DRIVEN ARCHITECTURE

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконав: здобувач вищої освіти
групи КНм-21
Шпанчик Петро Миколайович
_____
(підпис)
Керівник: к.т.н., доцент
Ліщина Валерій Олександрович
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Валерій Олександрович
_____
(підпис)

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерних наук*
Ступінь вищої освіти: *магістр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *122 Комп'ютерні науки*
Освітня програма: *«Комп'ютерні науки»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Валерій ЛІЩИНА
«14» травня 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ДРУГОГО (МАГІСТЕРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Шпанчик Петро Миколайович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка та дослідження системи на базі подійно-орієнтованої архітектури»

Керівник к.т.н., доцент Ліщина Валерій Олександрович

затверджені наказом закладу вищої освіти від «*14*» *травня* *2025* р. № 255/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«05» грудня 2025 р.*

3. Вихідні дані до роботи: _____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):
Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз і вибір засобів проектування, опис функціонального наповнення об'єкта проектування, розробка й обґрунтування системного наповнення, експерементальне дослідження результативності предмету дослідження.

_____

5. Перелік графічного матеріалу:

_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблематики за темою роботи та постановка завдань дослідження	*Ліщина В. О.*		
Теоретичне дослідження та практична реалізація предмету дослідження	*Ліщина В. О.*		
Експеременталне дослідження результативності предмету дослідження	*Ліщина В. О.*		
Показник запозичень тексту	_____ %		
Інструментальна перевірка	*Кошелюк В. А.*		
Нормоконтроль	*Сачук В. О.*		
Гарант ОПП	*Ліщина В. О.*		

7. Дата видачі завдання *«14» травня 2025 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*до 30.06.2025 р*	
2	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*до 01.09.2025 р.*	
3	*Розробити функціональну схему роботи програмного продукту*	*до 01.10.2025 р*	
4	*Описати засоби розробки об'єкта проектування*	*до 15.10.2025 р.*	
5	*Практична реалізація об'єкта проектування*	*до 10.11.2025 р.*	
6	*Провести експеременталне дослідження результативності предмету дослідження*	*до 25.11.2025 р.*	
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*до 05.12.2025 р.*	

Здобувач вищої освіти _____ Петро ШПАНЧИК

Керівник роботи _____ Валерій ЛІЩИНА

АНОТАЦІЯ

Шпанчик П. М. Кваліфікаційна робота магістра присвячена актуальній проблемі сучасної програмної інженерії – пошуку балансу між архітектурною надійністю та гнучкістю в умовах динамічних бізнес-вимог. У роботі проведено комплексне дослідження методів побудови високонавантажених систем на основі подійно-орієнтованої архітектури (Event-Driven Architecture, EDA) в контексті мікросервісного підходу.

Об’єктом дослідження є процеси проектування та розробки розподілених програмних систем E-commerce. Предметом дослідження є методи та засоби забезпечення еволюційної гнучкості архітектури через використання асинхронного обміну повідомленнями.

В роботі детально проаналізовано концепцію «гнучкої архітектури» (Agile Architecture) та поняття архітектурно значущих вимог (ASR). Спроектовано та реалізовано прототип системи електронної комерції з використанням стеку технологій Python (Django), Apache Kafka, PostgreSQL, Redis та Docker.

Практична цінність роботи полягає у демонстрації ефективності EDA для вирішення проблем «аналітичного паралічу» та забезпечення можливості інтеграції нових складних функціональних модулів (таких як повнотекстовий пошук та clickstream-аналітика) на пізніх етапах життєвого циклу без модифікації існуючого коду.

Ключові слова: подійно-орієнтована архітектура, мікросервіси, agile, apache kafka, e-commerce, гнучка архітектура, архітектурно значущі вимоги.

ABSTRACT

Petro Shpanchyk. The master's qualification thesis is devoted to the urgent problem of modern software engineering – finding a balance between architectural reliability and flexibility in the context of dynamic business requirements. The thesis conducts a comprehensive study of methods for building high-load systems based on Event-Driven Architecture (EDA) within the microservice approach.

The object of research is the processes of design and development of distributed E-commerce software systems. The subject of research is methods and tools for ensuring evolutionary architectural flexibility through the use of asynchronous messaging.

The concept of Agile Architecture and the notion of Architecturally Significant Requirements (ASR) are analyzed in detail. A prototype of an e-commerce system was designed and implemented using the Python (Django), Apache Kafka, PostgreSQL, Redis, and Docker technology stack.

The practical value of the work lies in demonstrating the effectiveness of EDA for solving analysis paralysis problems and ensuring the ability to integrate new complex functional modules (such as full-text search and clickstream analytics) at late stages of the lifecycle without modifying existing code.

Keywords: event-driven architecture, microservices, agile, apache kafka, e-commerce, agile architecture, architecturally significant requirements.

ЗМІСТ

ВСТУП

Сучасні інформаційні системи дедалі частіше функціонують у розподілених середовищах, обслуговують змінні потоки запитів та інтегруються з великою кількістю зовнішніх і внутрішніх сервісів. У таких умовах традиційні архітектурні підходи з високою зв’язаністю компонентів ускладнюють гнучке масштабування та швидке впровадження змін. Одним із перспективних напрямів розв’язання цих проблем є застосування подійно-орієнтованої архітектури (event-driven architecture, EDA), у якій взаємодія між сервісами здійснюється через події, що відображають зміну стану об’єктів або факт настання зовнішніх дій.

Подійно-орієнтована архітектура ґрунтується на моделі асинхронної взаємодії, де ключовими компонентами виступають сервіси-виробники подій, маршрутизатори подій та сервіси-споживачі. Виробники публікують події до маршрутизатора, який фільтрує та розповсюджує їх між споживачами; при цьому виробники та споживачі пов’язані між собою переважно через чергу/брокер повідомлень, що забезпечує можливість незалежного масштабування та оновлення компонентів системи [1].

Актуальність теми зумовлена потребою створення архітектурних рішень, здатних підтримувати високу гнучкість, масштабованість і відмовостійкість у системах з інтенсивними потоками бізнес-подій. EDA забезпечує низьку зв’язність сервісів, що позитивно впливає на еволюційність системи та спрощує інтеграцію нових модулів; також вона демонструє значний потенціал масштабування й підвищену стійкість до відмов за рахунок розподіленої обробки подій і використання черг повідомлень. Окремою перевагою є асинхронна «fire and forget» модель, яка зменшує залежність сервісів від синхронних відповідей і може покращувати загальну швидкодію системи. Водночас EDA має специфічні виклики: потребує складнішої інфраструктури міжсервісної комунікації та може породжувати тимчасову розбіжність станів через eventual consistency, що ускладнює контроль транзакційності й обробку

виняткових ситуацій. Отже, розроблення та дослідження системи на базі EDA є доцільним як з практичної, так і з науково-прикладної точки зору. Висвітлення актуальності подається стисло відповідно до методичних рекомендацій.

Мета роботи – розробити та дослідити інформаційну систему (прототип) на базі подійно-орієнтованої архітектури та експериментально оцінити ефективність запропонованих архітектурних рішень щодо забезпечення вимог до швидкодії, масштабованості та надійності.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

− проаналізувати сучасний стан проблеми побудови розподілених систем із високими вимогами до масштабованості та гнучкості;

− дослідити принципи, моделі та компоненти подійно-орієнтованої архітектури;

− сформувати функціональні та нефункціональні вимоги до системи;

− розробити архітектурну модель подійної взаємодії сервісів і визначити контракти подій;

− обґрунтувати вибір технологічних засобів реалізації (зокрема інфраструктури обміну подіями);

− реалізувати прототип системи на основі EDA;

− розробити методику та провести експериментальне дослідження з використанням сценаріїв навантаження;

− проаналізувати результати експериментів і сформулювати практичні рекомендації щодо застосування EDA в системах відповідного класу.

Об’єкт дослідження – процеси проєктування та функціонування розподілених інформаційних систем.

Предмет дослідження – методи, моделі та програмно-технологічні засоби реалізації подійно-орієнтованої архітектури в інформаційних системах. Формулювання об’єкта й предмета подано з урахуванням методичних визначень.

Методи дослідження включають аналіз літературних і нормативних джерел з архітектури розподілених систем; системний аналіз вимог;

архітектурне моделювання подійних потоків; прототипування; експериментальне тестування та порівняльний аналіз результатів.

Наукова новизна полягає в удосконаленні підходу до проєктування подійної взаємодії сервісів з урахуванням архітектурно значущих вимог і в розробленні/уточненні методики експериментальної перевірки характеристик системи, побудованої на EDA. Підхід до формулювання новизни узгоджується з вимогами методичних рекомендацій щодо відображення відмінностей отриманих результатів від відомих раніше.

Практична цінність роботи полягає у створенні прототипу подійно-орієнтованої системи, який може бути використаний як основа для подальшого розвитку прикладних рішень у відповідній предметній області, а також у формуванні рекомендацій щодо вибору архітектурних і технологічних засобів для забезпечення вимог швидкодії та масштабованості. Зазначені аспекти практичної цінності випливають із переваг EDA щодо незалежного масштабування виробників і споживачів та можливостей інфраструктури типу Kafka.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ТЕОРЕТИЧНІ ЗАСАДИ ПОДІЙНО-ОРІЄНТОВАНОЇ АРХІТЕКТУРИ

1.1 Огляд і аналіз предметної області та результатів існуючих досліджень

Подійно-орієнтована архітектура (Event-Driven Architecture, EDA) сформувалася як відповідь на зростаючу потребу у створенні розподілених систем, здатних реагувати на зміни стану в режимі, близькому до реального часу, та підтримувати високі навантаження. У таких системах міжсервісна взаємодія відбувається через повідомлення/події, які несуть інформацію про зміну стану об’єкта або про зовнішній тригер. Класична модель EDA включає три ключові ролі: сервіси-виробники подій, маршрутизатор/шину подій та сервіси-споживачі. Виробники публікують події, маршрутизатор їх фільтрує та доставляє споживачам, а зв’язність між компонентами мінімізується завдяки використанню черги/брокера повідомлень. Така організація дозволяє незалежно масштабувати, оновлювати та розвивати сервіси [2-3].

У сучасній інженерії ПЗ EDA часто розглядають у поєднанні з мікросервісним стилем та підходами event streaming. Дослідження останніх років підкреслюють потенціал EDA щодо підвищення модульності, масштабованості та паралельності виконання, але одночасно звертають увагу на ризики зростання складності проєктування та супроводу. Зокрема, у роботі Lazzari та Farias (2023) виконано емпіричне порівняння EDA і REST-стилю з погляду модульності під час еволюції застосунку: результати свідчать, що EDA може покращувати separation of concerns, однак за рядом показників (зв’язність, згуртованість, складність, розмір) поступається альтернативам у конкретних сценаріях розвитку системи. Це важливий сигнал для практиків: вибір EDA має ґрунтуватися на вимогах домену та очікуваних сценаріях еволюції, а не лише на трендах [4].

Окремий напрям досліджень стосується технологічних платформ, що реалізують або підтримують EDA на практиці. Найпоширенішим індустріальним прикладом є Apache Kafka, яка концептуально спирається на модель розподіленого журналювання та забезпечує високопродуктивну доставку подій між багатьма продюсерами й консьюмерами. Перелік академічних публікацій про Kafka та event streaming, зібраний Apache Software Foundation, демонструє сталість наукового інтересу до цієї платформи й наявність фундаментальних робіт щодо її архітектури та продуктивності. Зокрема, базова праця Kreps, Narkhede, Rao (2021) описує Kafka як систему для високих обсягів лог/подій з низькою затримкою та показує її конкурентні переваги в продуктивності порівняно з традиційними брокерами повідомлень [5].

Важливим концептуальним наслідком переходу до EDA і мікросервісів є зміна підходів до узгодженості даних та транзакційності. За наявності патерну «database per service» глобальні ACID-транзакції стають практично неефективними, тому у фокусі опиняються підходи eventual consistency та шаблони керування розподіленими бізнес-операціями. Практично й теоретично найбільш уживаним рішенням є Saga-патерн, який визначається як послідовність локальних транзакцій, що ініціюють наступні кроки через події/повідомлення, а у випадку збою запускають компенсуючі транзакції. Окремо виділяють два механізми координації: choreography (через доменні події) та orchestration (через координатора) [6].

Сучасні дослідження поглиблюють розуміння обмежень класичних саг. Наприклад, Daraghmi та ін. (2022) аналізують проблему відсутності ізоляції в Saga та пропонують удосконалення, які підвищують коректність і продуктивність у мікросервісному e-commerce-сценарії. У свою чергу, Bashtovyi та Fechan (2024) експериментально демонструють, що розподілені транзакції в мікросервісах збільшують латентність порівняно з монолітом через мережеві взаємодії, але Saga-підхід із асинхронним обміном через Kafka може бути практично виправданим за умов, коли бізнес-вимоги допускають eventual

consistency. Такі результати узгоджуються із загальною логікою EDA: архітектура виграє у гнучкості та масштабованості ціною складніших механізмів контролю стану й помилок [6].

Окрім транзакційності, у наукових роботах і практичних матеріалах систематично акцентують на організаційних та інженерних викликах EDA: необхідності стандартизації технологічного стеку, уніфікації протоколів взаємодії, уникнення циклічних залежностей і чіткого виділення шарів абстракції. Також важливо враховувати, що EDA та мікросервісний підхід не є оптимальними для малих або простих систем, де витрати на інфраструктуру та підтримку можуть перекрити архітектурні переваги.

Отже, сучасний стан предметної області свідчить, що подійно-орієнтована архітектура є перспективним технологічним напрямом для розроблення розподілених систем з високими вимогами до масштабованості та реактивності. Водночас емпіричні результати останніх років показують неоднозначність впливу EDA на модульність і складність еволюції систем, а питання керування узгодженістю даних, спостережуваності та інженерної дисципліни залишаються критичними для успішного впровадження. Це формує підґрунтя для подальшого дослідження та обґрунтовує актуальність теми магістерської роботи, зосередженої на розробці та експериментальній оцінці системи на базі EDA [7].

1.2 Огляд і аналіз методів та засобів розробки системи на базі EDA

Проєктування системи на базі подійно-орієнтованої архітектури доцільно починати з уточнення архітектурно значущих вимог (ASR) та систематизації нефункціональних характеристик, адже саме вони визначають ключові компроміси між продуктивністю, масштабованістю, надійністю, керованістю й складністю супроводу. У наукових і методичних працях з архітектури програмних систем наголошується на необхідності відокремленого аналізу

таких вимог і побудови архітектурних рішень, що напряму на них спираються (зокрема підходи ASR та NFR).

Для формалізованого прийняття архітектурних рішень у межах даної теми доцільно застосувати Attribute-Driven Design (ADD) як метод проєктування архітектури саме на основі ASR. Це дозволяє зв’язати вимоги до масштабованості, відмовостійкості та швидкодії з конкретними архітектурними механізмами EDA і мікросервісів. Одночасно корисним інструментальним підходом оцінювання обраної архітектури може бути Architecture Tradeoff Analysis Method (ATAM), який орієнтований на аналіз компромісів між якісними атрибутами та ризиками архітектурних рішень [8].

У контексті сучасних розподілених систем EDA найчастіше реалізується у зв’язці з мікросервісним стилем. Дослідження та практичні матеріали з мікросервісів підкреслюють, що ефективність такої архітектури забезпечується сукупністю патернів інфраструктурної взаємодії (API Gateway, service discovery, композиція API), які керують зовнішнім доступом до сервісів та їх узгодженою роботою. Водночас важливо враховувати, що EDA не є універсальним рішенням для малих систем: при обмеженому функціоналі та невеликих командах надмірна інфраструктурна складність може нівелювати її переваги.

З технічного погляду ключовим засобом реалізації подійної взаємодії виступає брокер/шина подій. У ролі такого компонента у промислових EDA-рішеннях широко використовується Apache Kafka та екосистема конекторів, що підтримують інтеграцію з пошуковими та аналітичними сховищами. Це забезпечує масштабування обробки подій і побудову потокових конвеєрів даних для різних доменів. На рівні архітектурних властивостей EDA дозволяє досягати низької зв’язності сервісів, покращеної масштабованості та відмовостійкості, а також вигравати у швидкодії завдяки асинхронній моделі обробки «fire and forget» [9].

Разом із тим при впровадженні EDA необхідно враховувати типові проблеми: ускладнення міжсервісної комунікації через потребу в системі

черг/брокерів і тимчасову розбіжність станів унаслідок eventual consistency, що підвищує вимоги до дисципліни проєктування подійних контрактів, обробки помилок та моніторингу.

Таким чином, для вирішення поставленої задачі у цій роботі доцільно поєднати методи ASR-орієнтованого проєктування (ADD) **та** оцінювання компромісів архітектури (ATAM) з практичними патернами мікросервісної екосистеми і застосуванням брокера подій (наприклад, Kafka). Це створює методичну й технологічну основу для обґрунтованого проєктування та подальшого експериментального дослідження системи на базі подійно-орієнтованої архітектури [10].

1.3 Постановка завдання на кваліфікаційну роботу магістра

У межах теми «Розробка та дослідження системи на базі подійно-орієнтованої архітектури» визначено такі цілі:

− виконати аналіз сучасного стану розвитку подійно-орієнтованих і мікросервісних систем та узагальнити результати опублікованих досліджень щодо переваг і обмежень EDA;

− визначити архітектурно значущі вимоги до системи з урахуванням потреб масштабованості, швидкодії, надійності та гнучкості;

− сформувати функціональні й нефункціональні вимоги до майбутньої системи та уточнити сценарії її використання;

− спроєктувати модель подій, їхні контракти та потоки взаємодії між сервісами відповідно до принципів EDA, де виробники й споживачі подій пов’язані через чергу/брокер повідомлень;

− обґрунтувати вибір технологічних засобів реалізації подійної взаємодії та інфраструктури обробки подій;

− реалізувати прототип системи на основі подійно-орієнтованої архітектури;

− розробити методику експериментального дослідження та провести серію випробувань на предмет відповідності системи визначеним вимогам;

− здійснити аналіз результатів експериментів і сформулювати практичні рекомендації щодо доцільності застосування EDA у системах відповідного класу.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ СИСТЕМИ НА БАЗІ EDA

2.1 Обґрунтування вибору шляхів, технологій та алгоритмів

Вибір предметної області для перевірки підходу EDA. Для коректного дослідження переваг і обмежень EDA доцільно використати предметну область, у якій:

− наявні кілька доменних модулів із різними темпами розвитку;

− можливі різкі зміни навантаження; часто надходять нові архітектурно значущі вимоги (ASR).

E-commerce-системи є вдалим тестовим варіантом для сервіс-орієнтованих і подійних підходів через динаміку ринку, змінність вимог та потребу у швидкому розвитку функціоналу. Водночас EDA неефективна як «важка» архітектура для малих застосунків із обмеженим функціоналом та невеликими командами, де її складність може перекрити переваги [11].

Будемо розглядати демонстраційну тестову e-commerce-систему з базовими доменами «замовлення», «платежі», «склад», «доставка», що дозволить відтворити типові функціональні сценарії та перевірити поведінку архітектури при зміні ASR.

Гіпотеза дослідження. Враховуючи сутність подійного підходу та практичні спостереження, сформулюємо робочу гіпотезу: застосування EDA з використанням брокера подій забезпечить підвищення масштабованості та прийнятну швидкодію системи в умовах змінного навантаження і частих архітектурних змін.

У «EDA» акцентовано, що вимогу швидкодії в мікросервісах часто ускладнюють мережеві затримки між сервісами, але EDA частково нівелює цей недолік завдяки асинхронній природі обробки запитів. Масштабованість у такому випадку досягається незалежним масштабуванням продюсерів і споживачів подій, а також можливістю збільшення кількості партицій і інстансів обробників у Kafka [12].

Вибір інструмента подійної взаємодії. Ключовим засобом реалізації EDA є інструмент збереження та транспортування подій між сервісами. Найбільш відомі сучасні рішення RabbitMQ, Kafka та AWS SQS. RabbitMQ і Kafka де-факто стали стандартами для задач подійної взаємодії, хоча мають різні архітектурні підходи.

З огляду на потребу:

- зберігати події як потік даних;
- забезпечити високу пропускну здатність;
- масштабувати обробку через партиціювання.

Обґрунтованим є вибір Apache Kafka як основного брокера/шини подій. Інформація про події системи зберігається у Kafka, а взаємодія сервісів демонструється через часові діаграми; при цьому проведена оцінка показала відповідність архітектури початковим функціональним і нефункціональним вимогам.

Окремо важливим аспектом EDA на Kafka є контроль порядку повідомлень: гарантований порядок забезпечується в межах партиції, що потребує правильного вибору ключів розподілу подій або використання обмеженої кількості партицій для критичних потоків.

Узагальнення підходів до формування вимог. Вимоги до ПЗ доцільно структуризувати на функціональні та нефункціональні (NFR). Функціональні вимоги описують задачі, які система повинна виконувати, тоді як NFR визначають, «якою» має бути система, зокрема щодо надійності, захищеності, масштабованості та підтримуваності. Саме така класифікація є основою для формування ASR і подальшого архітектурного проєктування [13].

Наведемо узгоджений набір вимог для демонстраційної e-commerce EDA-системи. Їх можна оформити як дві таблиці та надалі деталізувати у вигляді SRS/специфікації (табл. 2.1).

Таблиця 2.1 – Функціональні вимоги тестової системи

Код	Зміст вимоги
FR-01	система повинна забезпечувати реєстрацію та автентифікацію користувачів з розмежуванням ролей (клієнт, адміністратор, оператор)
FR-02	система повинна підтримувати створення та зміну статусів замовлення із фіксацією доменних подій «OrderCreated», «OrderConfirmed», «OrderCancelled»
FR-03	сервіс платежів повинен обробляти події замовлень і генерувати події «PaymentAuthorized», «PaymentFailed», «PaymentCompleted»
FR-04	складський сервіс повинен реагувати на події оплати й резервувати товар, формуючи події «StockReserved», «StockRejected»
FR-05	сервіс доставки повинен ініціювати відправлення після підтвердження резерву та генерувати події «ShipmentCreated», «ShipmentDelivered»
FR-06	система повинна підтримувати механізм компенсаційних дій для розподілених бізнес-процесів (зокрема за підходом Saga)
FR-07	система повинна забезпечувати журналювання подій та можливість їх перегляду для адміністративного контролю
FR-08	система повинна надавати інтеграційні API для зовнішніх сервісів (опціонально) без порушення подійних контрактів

Логіка таких сценаріїв відповідає практиці EDA для e-commerce та може бути відображена часовими діаграмами взаємодії сервісів (табл. 2.2).

Таблиця 2.2 – Нефункціональні вимоги (NFR) та ASR-орієнтири

Код	Зміст вимоги
NFR-01	система повинна забезпечувати горизонтальне масштабування продюсерів і споживачів подій незалежно один від одного
NFR-02	система повинна зберігати працездатність при відмові окремих сервісів без зупинки всього бізнес-процесу
NFR-03	система повинна підтримувати асинхронну модель «fire and forget» для зменшення критичних затримок у ланцюжках обробки
NFR-04	система повинна забезпечувати контроль порядку обробки подій у критичних потоках шляхом коректного партиціювання
NFR-05	система повинна забезпечувати спостережуваність: централізоване логування, трасування подій, метрики черги та сервісів
NFR-06	система повинна бути розширюваною для додавання нових доменів і функціональних модулів без зміни існуючих контрактів
NFR-07	система повинна гарантувати базові вимоги безпеки (аутентифікація сервісів, контроль доступу до подій і конфігурацій)

Пункти NFR-01–NFR-04 безпосередньо випливають із властивостей EDA та практики масштабування й упорядкування подій у Kafka.

2.2 Вибір методології проектування системи

У процесі проектування сервіс-орієнтованої архітектури тестової системи важливо обрати методологію, яка дозволяє поєднати гнучкість розробки з контрольованим прийняттям архітектурних рішень. Оскільки майбутня система будується на принципах мікросервісної та подійно-орієнтованої архітектур, її ключовими драйверами є не лише функціональні сценарії, а й нефункціональні вимоги, що мають суттєвий вплив на структуру, комунікацію компонентів та еволюційність рішення. Саме тому найбільш доцільним підходом для формалізації архітектури тестової системи обрано дизайн на основі атрибутів (Attribute-Driven Design, ADD), оскільки він має ітеративну природу та безпосередньо орієнтований на врахування якісних атрибутів і архітектурно значущих вимог (ASR). Така аргументація узгоджується з базовим описом ADD у тексті роботи [14].

Метод ADD є ефективним для проектування розподілених систем з декількох причин. Ітеративність і масштабованість процесу проектування**.** Мікросервісна система природно складається з автономних бізнес-компонентів. ADD дозволяє розпочати з архітектури системи як цілого, а далі уточнювати дизайн кожного сервісу окремими ітераціями. Це знижує ризик надмірного «Big Design Up Front» і залишає простір для еволюції архітектури.

Фокус на якості та ризиках. Для e-commerce системи критичними є безпека, доступність, продуктивність, масштабованість і зручність використання. ADD створений як підхід, де архітектурні рішення приймаються не лише з погляду функціоналу, але й з огляду на те, які NFR/ASR є найважливішими для бізнесу та користувачів.

Природна сумісність з Agile-практиками. ADD не суперечить ідеї поступового виникнення архітектури, а забезпечує керований компроміс між «emergent architecture» та попереднім плануванням.

Згідно з логікою ADD, стартові дані для проектування мають включати функціональні вимоги (основні бізнес-сценарії), набір бажаних якісних

атрибутів, архітектурні обмеження (вибір технологій, модель розгортання, допустимі механізми інтеграції тощо).

Результатом застосування ADD має стати набір архітектурних ескізів, які фіксують ключову структуру та правила взаємодії сервісів, а також визначають напрямки подальшого уточнення дизайну під час наступних ітерацій розробки

ADD застосовується як покроковий алгоритм проектування архітектури тестової e-commerce системи.

Крок 1. Вибір частини системи, що проєктується. На першій ітерації обирається система в цілому як верхньорівневий об’єкт проектування. Метою є виділення основних сервісів, визначення типу взаємодії між ними та узгодження глобальних архітектурних рішень (наприклад, потреби API Gateway, сервісу аутентифікації, механізмів синхронної та асинхронної взаємодії). У наступних ітераціях фокус зміщується на окремі доменні підсистеми: користувачі, каталог товарів, кошик, замовлення, відгуки, сповіщення тощо.

Крок 2. Визначення ASR для обраної частини. Для верхнього рівня системи пріоритетними архітектурно значущими вимогами є: безпека та контроль доступу; доступність сервісу; продуктивність типових користувацьких операцій; масштабованість при пікових навантаженнях; підтримуваність і технологічна узгодженість.

На цьому кроці вимоги ранжуються за критичністю для бізнесу та ризиками їх невиконання. Це дозволяє обрати такі архітектурні рішення, які найкраще відповідають пріоритетним атрибутам якості.

Крок 3. Проєктування архітектури відповідно до ASR/

На цьому етапі формується високорівневий дизайн системи. Для тестової e-commerce платформи доцільним є:

− декомпозиція на спеціалізовані мікросервіси, кожен з яких має чітку відповідальність;

− застосування синхронної взаємодії (HTTP/REST) для критичних запитів типу «запит каталогу», «перевірка користувача», «створення

замовлення»; застосування асинхронної взаємодії на подіях для сценаріїв типу «додавання відгуку», «зміна статусу замовлення», «надсилання сповіщень»; встановлення компонентів підтримки інфраструктури (API Gateway, Service Discovery, централізована аутентифікація).

Ці рішення безпосередньо мають відображати вимоги щодо продуктивності, відмовостійкості та еволюційності архітектури.

Крок 4. Перегляд інших вимог і планування наступної ітерації.

Після формування базового архітектурного каркаса здійснюється перевірка: які ASR вже покриті поточними рішеннями, а які потребують додаткових уточнень. Наприклад, якщо на рівні системи визначено потребу у високій доступності, наступною ітерацією може стати деталізація механізмів масштабування сервісу каталогу або кошика (кешування, реплікація, балансування навантаження).

Крок 5. Повторення циклу до задоволення ключових ASR.

Ітеративність ADD дозволяє поступово деталізувати архітектуру без втрати зв’язку з бізнес-цілями та якісними атрибутами, що особливо важливо в умовах дослідження гнучкої архітектури.

Щоб показати практичність підходу, можна коротко інтерпретувати одну з наступних ітерацій для сервісу кошика:

− об’єкт проектування: сервіс кошика;

− ASR: низька затримка відповіді, коректність даних, масштабованість у пікові періоди;

− архітектурні рішення:

− кешування стану кошика в швидкому сховищі;

− ізоляція даних сервісу від інших модулів;

− чіткий контракт API для взаємодії з сервісом каталогу й замовлень;

− перевірка компромісів: баланс між швидкодією та узгодженістю даних при розподіленій взаємодії [15].

Таблиця 2.3 – Відповідність архітектурно значущих вимог (ASR) архітектурним тактикам та очікуваним ефектам у тестовій EDA-системі

ASR (атрибут якості)	Архітектурні тактики / рішення	Очікуваний ефект у системі
Масштабованість	Декомпозиція на мікросервіси за доменами; асинхронна подійна взаємодія; горизонтальне масштабування продюсерів і консьюмерів; партиціювання потоків подій	Можливість незалежного масштабування критичних доменів (каталог, кошик, замовлення, оплата) без впливу на інші сервіси
Продуктивність (швидкодія)	Мінімізація синхронних ланцюжків викликів; застосування подій для фонового оброблення; кешування часто використовуваних даних; оптимізація ключів партицій для зменшення «гарячих» сегментів	Скорочення середнього часу обробки сценаріїв; підвищення стабільності під піковими навантаженнями
Надійність / відмовостійкість	Ідемпотентні обробники подій; повторні спроби обробки з політиками backoff; ізоляція відмов у межах сервісів; механізми DLQ (для недоставлених/помилкових подій)	Зниження ризику каскадних відмов; коректне відновлення після помилок у частині сервісів
Узгодженість даних	Використання eventual consistency; патерн Saga для розподілених бізнес-операцій; компенсуючі транзакції; чіткі доменні стани і переходи	Кероване виконання складних наскрізних процесів (замовлення–оплата–склад–доставка) без глобальних транзакцій
Модифікованість / еволюційність	Контракти подій з версіонуванням; слабка зв’язність продюсерів і консьюмерів; ізоляція моделей даних за сервісами	Додавання нового функціоналу або сервісів без зміни існуючих модулів і без порушення взаємодії
Спостережуваність	Централізоване логування; метрики брокера і консьюмерів; трасування кореляційних ідентифікаторів подій; аудит подій	Підвищення керованості системи; спрощення пошуку причин інцидентів і аналізу продуктивності
Безпека	Аутентифікація і авторизація сервісів; контроль доступу до топіків; захист конфігурацій; принцип найменших привілеїв	Зниження ризику несанкціонованого доступу до подій і бізнес-даних
Керованість складності	Стандартизація стеку; уніфікація форматів подій; правила іменування топіків; документування подійних сценаріїв	Менше архітектурного «шуму»; прогнозованість розвитку системи та простіший супровід

Таблиця 2.3 відображає логіку застосування ADD, коли архітектурні рішення формуються не «від технологій», а від пріоритетних атрибутів якості. Вона показує, що для тестової e-commerce системи на базі EDA ключовими є масштабованість, продуктивність і відмовостійкість, а також контрольована

узгодженість даних у розподілених бізнес-процесах. Окремо акцентовано на еволюційності через контракти подій і версіонування, що є критичним для систем із частими змінами доменних вимог. Тактики спостережуваності та безпеки винесені як наскрізні, оскільки подійна взаємодія суттєво ускладнює діагностику інцидентів і контроль доступу при зростанні кількості сервісів. У сукупності це формує структуровану основу для подальшого уточнення вимог і обґрунтування вибору технологій реалізації. Таким чином, наступним кроком є формалізація функціональних і нефункціональних вимог тестової системи, виділення архітектурно значущих вимог та їх пріоритизація [16-17].

2.3 Формування вимог до тестової системи

Проєктування системи на базі подійно-орієнтованої архітектури доцільно розпочинати з формування вимог, оскільки саме вони визначають архітектурний стиль, характер міжсервісної взаємодії та вибір інфраструктурних засобів. Подійно-орієнтована архітектура передбачає, що міжсервісна комунікація здійснюється через події, які містять інформацію про зміну стану об’єкта системи або повідомляють про надходження зовнішнього тригера. Такі архітектури включають сервіси-виробники подій, маршрутизатори подій та сервіси-споживачі, причому виробники й споживачі пов’язані між собою переважно лише чергою повідомлень. Це створює умови для незалежного масштабування й оновлення компонентів системи.

Виходячи з обраної предметної області електронної комерції, функціональні вимоги тестової системи мають охоплювати типовий життєвий цикл замовлення та підтримувати узгоджену роботу доменних сервісів замовлень, оплати, складу й доставки. Логіка EDA в цьому випадку дає змогу реалізувати основні бізнес-сценарії через публікацію і оброблення доменних подій, що зменшує зв’язаність модулів та спрощує розширення системи новими компонентами. Перевага такого підходу полягає у можливості інтегрувати нові сервіси без необхідності значного рефакторингу вже реалізованого

функціоналу, оскільки споживачі подій повинні знати лише формат і семантику подій відповідно до контракту.

Нефункціональні вимоги для тестової системи формуються з урахуванням особливостей EDA і мікросервісів. Вимога швидкодії в розподілених системах ускладнюється наявністю мережевих затримок між сервісами, проте подійний підхід частково нівелює цей недолік за рахунок асинхронної природи обробки запитів. Це дозволяє застосовувати модель fire and forget для скорочення критичних затримок у бізнес-ланцюжках. Масштабованість розглядається як один із ключових атрибутів якості, оскільки EDA дозволяє незалежно масштабувати виробників і споживачів подій, а також інфраструктурні компоненти обміну повідомленнями. У випадку різкого зростання трафіку в межах окремого домену план масштабування може передбачати збільшення кількості інстансів продюсерів і консьюмерів, розширення кількості партицій черги та, за потреби, вертикальне масштабування бази даних.

На рисунку 2.1 відображено основних акторів (користувач, адміністратор, платіжний провайдер, служба доставки) та зовнішні системи інтеграції.

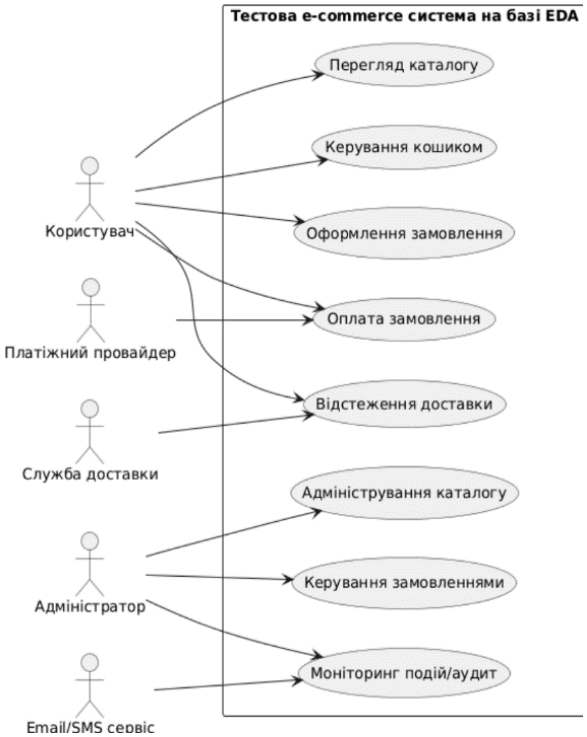

Рисунок 2.1 – Контекстна діаграма тестової e-commerce системи

Рисунок 2.2 фіксує основні функціональні сценарії, які надалі будуть пов’язані з доменними подіями.

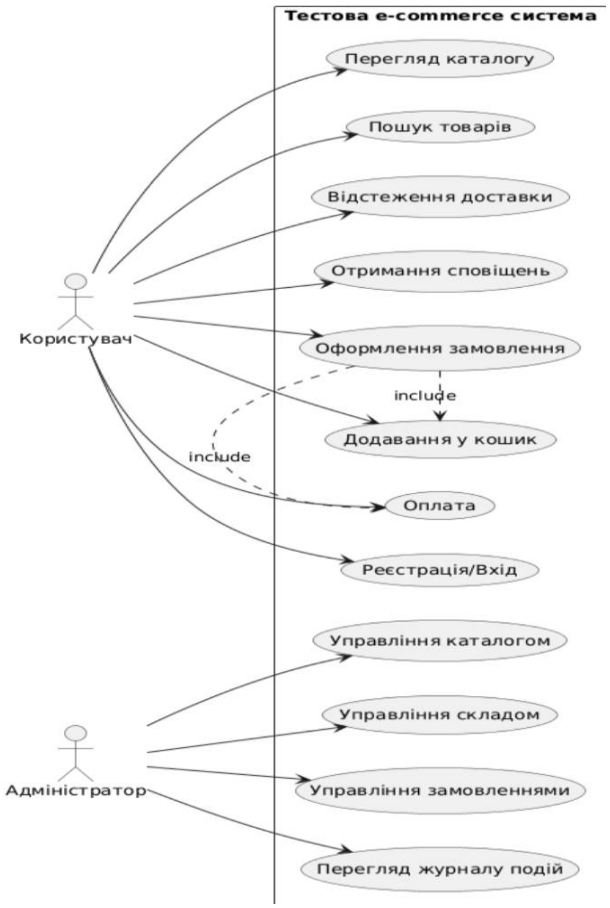

Рисунок 2.2 – Діаграма варіантів використання системи

На рисунку 2.3 показано відповідність ключових бізнес-дій доменним подіям, що ініціюють обробку у відповідних сервісах.

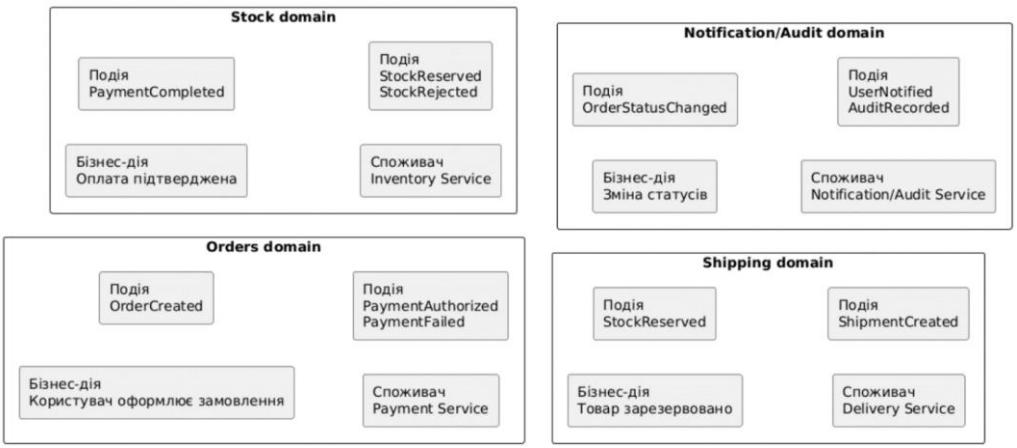

Рисунок 2.3 – Карта подій предметної області

2.4 Проєктування архітектури тестової системи на базі EDA

Архітектурне проєктування тестової системи виконується на підставі сформованих функціональних і нефункціональних вимог та з урахуванням принципів подійно-орієнтованої взаємодії. Основною ідеєю архітектури є розподіл бізнес-функцій за окремими сервісами, які реагують на доменні події та формують наступні події життєвого циклу бізнес-процесу. У такому підході маршрутизатор/брокер подій виступає центральним механізмом асинхронної комунікації, забезпечуючи передачу повідомлень між виробниками і споживачами без прямої залежності між ними.

Проєктована архітектура передбачає виділення щонайменше чотирьох базових доменних сервісів: сервісу замовлень, сервісу оплати, складського сервісу та сервісу доставки. Взаємодія між ними реалізується як послідовність подій, де створення замовлення ініціює потік подій оплати, резервування товару та формування доставки. Така логіка дозволяє перевірити ключові властивості EDA на практиці, зокрема низьку зв’язаність, відмовостійкість і масштабованість у межах окремих доменів. Слабо пов’язані вузли формують цілісний механізм обробки повідомлень, а при виході з ладу одного з них інший може продовжити опрацювання подій з черги.

Важливим аспектом проєктування є забезпечення контролю порядку обробки подій у критичних потоках. Для інфраструктури на базі Kafka порядок повідомлень гарантується в межах однієї партиції, тому при збільшенні кількості партицій необхідно коректно підбирати ключі розподілу подій або використовувати обмежену кількість партицій для сценаріїв, де порядок є принциповим. Це дозволяє зменшити ризики логічних помилок у послідовностях зміни статусів замовлення чи оплати.

У зв’язку з eventual consistency доцільно передбачити механізми обробки розподілених бізнес-операцій із підтримкою компенсуючих дій. Для тестової системи це може бути реалізовано на рівні доменних сценаріїв, коли помилка оплати або відсутність товару на складі ініціює відповідні події відміни чи

компенсації. Такий підхід безпосередньо відповідає необхідності контролю транзакційної природи процесів у EDA, яка ускладнюється через кінцеву синхронізацію даних.

З погляду нефункціональних характеристик архітектура повинна забезпечувати можливість незалежного масштабування компонентів: наприклад, у випадку збільшення навантаження на домен замовлень система повинна дозволяти додавання інстансів відповідного продюсера та консьюмера без втручання в інші сервіси.

Для повноцінного відображення архітектурної моделі зобразимо набір UML-діаграм і схем, які фіксують як статичну, так і динамічну структуру тестової системи.

На рисунку 2.4 показано основні сервіси, брокер подій, канали синхронної взаємодії та зовнішні інтеграції.

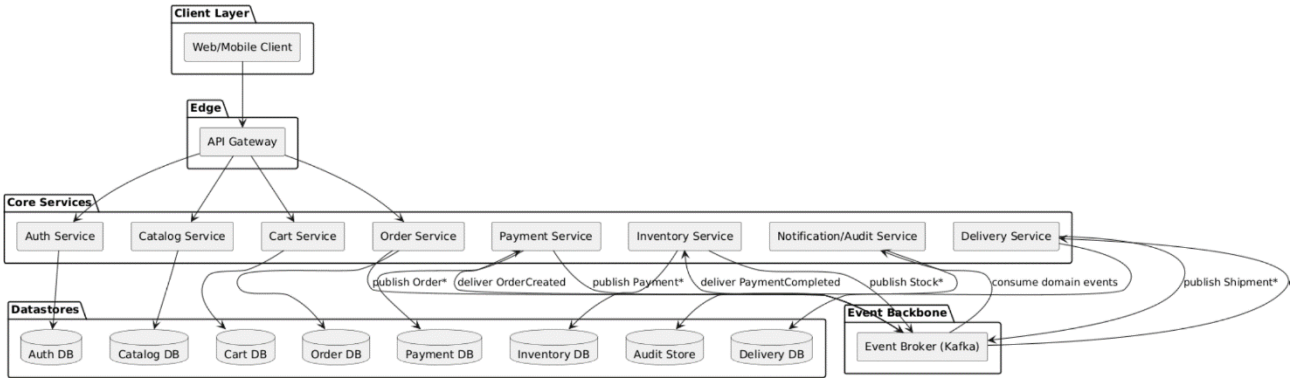

Рисунок 2.4 – Компонентна діаграма тестової системи на базі EDA

Рисунок 2.5 відображає логіку контейнеризації сервісів, розміщення брокера подій та баз даних сервісів.

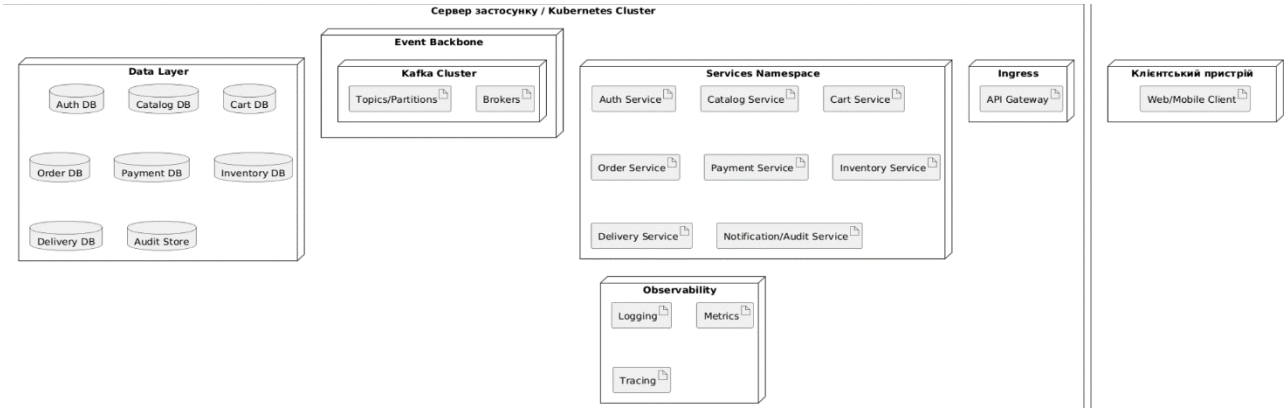

Рисунок 2.5 – Діаграма розгортання (deployment) прототипу

На діаграмі (рис. 2.6) показано ланцюжок доменних подій між сервісом замовлень, оплати, складу та доставки.

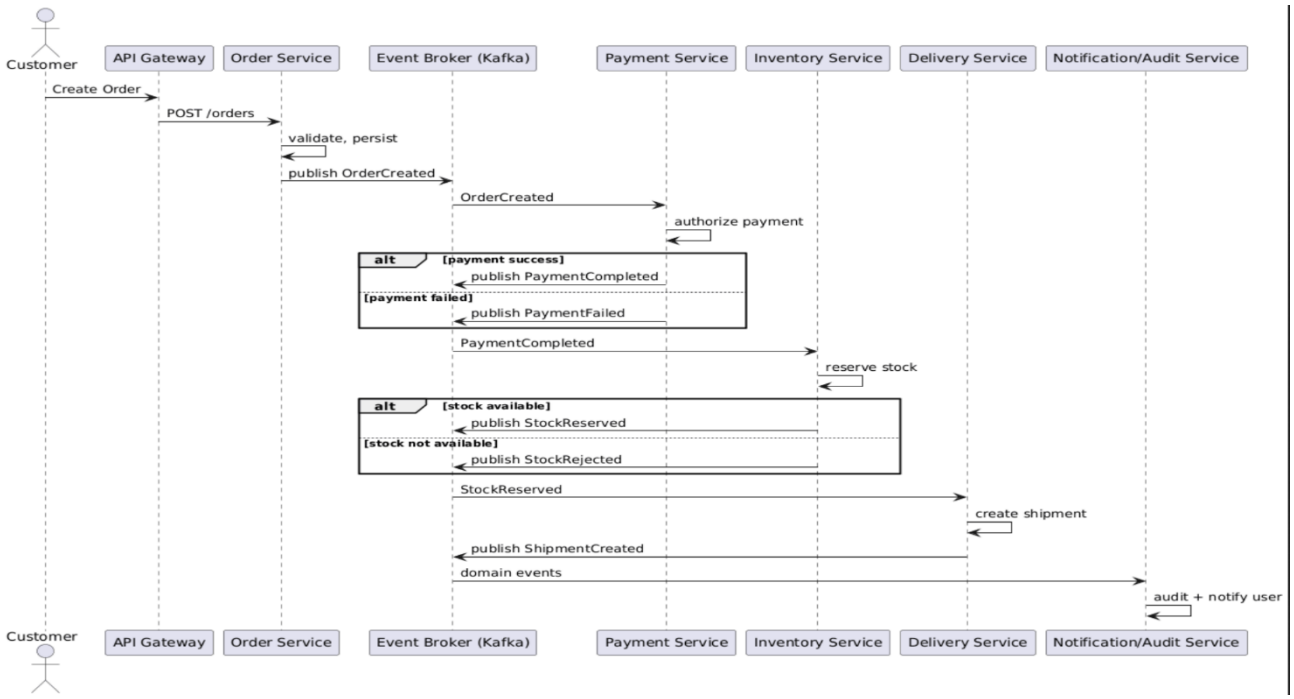

Рисунок 2.6 – Діаграма послідовності обробки сценарію створення замовлення

Відображаємо переходи між станами із зазначенням подій-тригерів на рисунку 2.7.

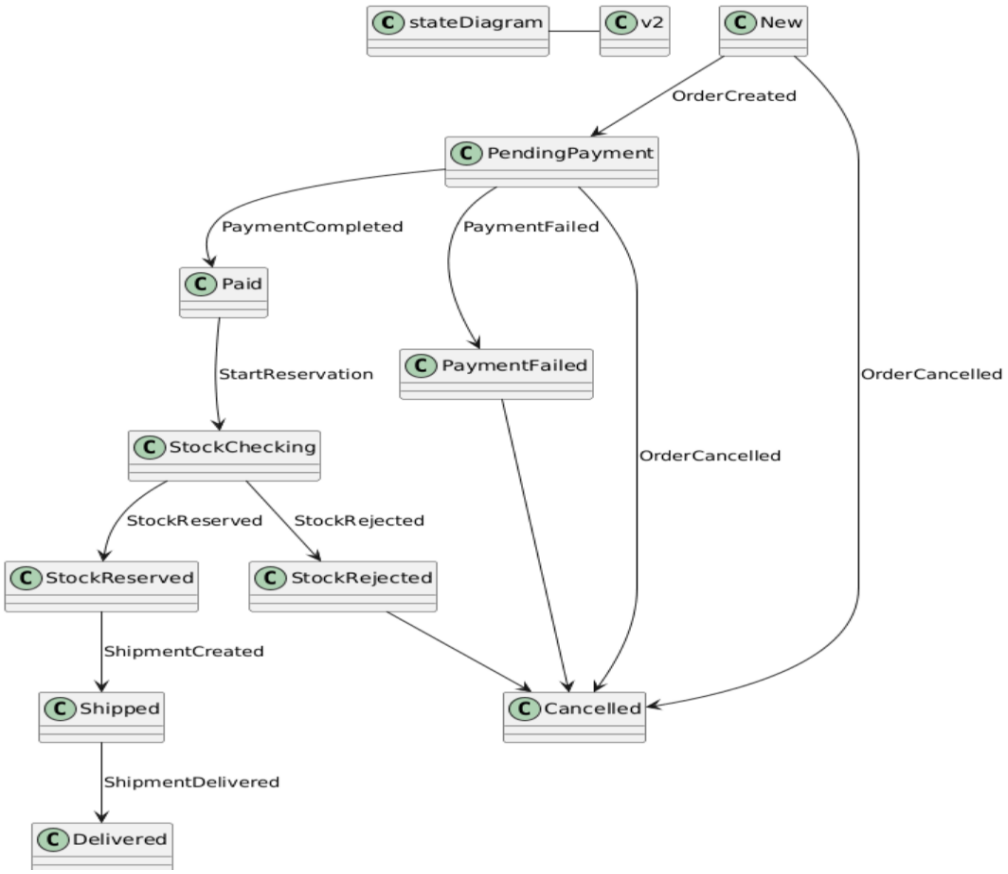

Рисунок 2.7 – Діаграма станів замовлення

Рисунок 2.8 показує підхід масштабування продюсерів, консьюмерів і партицій.

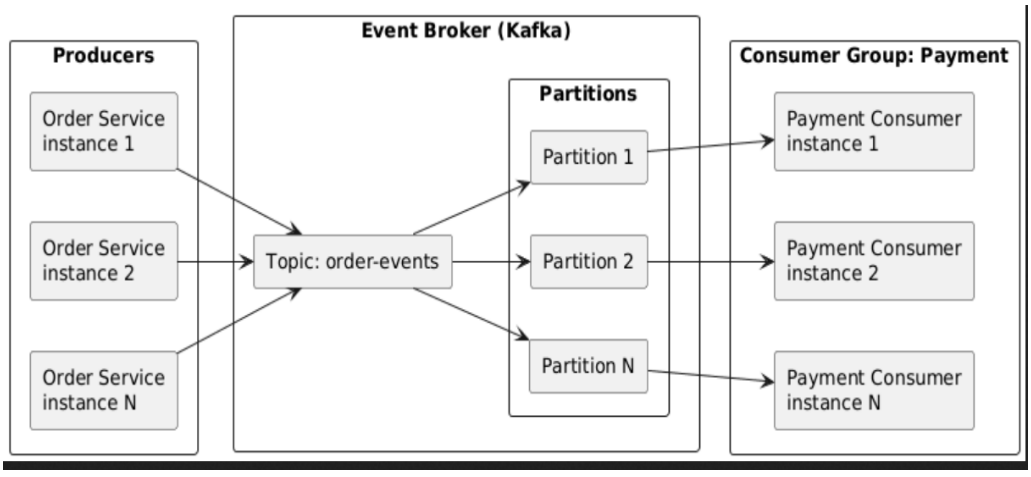

Рисунок 2.8 – Схема масштабування сервісів у межах домену

2.5 Технології для реалізації системи

Формування рекомендованого набору технологій для реалізації тестової системи на базі подійно-орієнтованої архітектури має спиратися на поєднання двох ключових міркувань. З одного боку, необхідно забезпечити підтримку архітектурних рішень, пов’язаних із мікросервісною та подійно-орієнтованою природою системи; з іншого – уникнути надмірної різнорідності технологічного стеку, яка ускладнює супровід та експлуатацію. Мікросервісний підхід формально допускає використання різних мов програмування й фреймворків для окремих сервісів, однак на практиці надмірна варіативність інструментів призводить до зростання операційних витрат, ускладнює залучення розробників до різних компонентів і робить більш дорогими всі зміни в інфраструктурі. Тому доцільним є обмеження кількості технологій і застосування узгодженого стеку, який відповідає потребам предметної області та можливостям команди розробки.

У межах даної роботи як базова платформа для реалізації сервісів обрано мову програмування Python у поєднанні з фреймворком Django. Такий вибір дає змогу отримати уніфікований підхід до реалізації бізнес-логіки для більшості сервісів системи, використати зрілу екосистему бібліотек і засобів розробки, а також забезпечити достатньо високу швидкість побудови прототипів. Django історично орієнтований на створення вебзастосунків із чітким поділом на рівні представлення, логіки та доступу до даних, що добре корелює з потребами сервісів на кшталт сервісу користувачів, каталогу, замовлень, відгуків тощо. Використання спільної технологічної основи для різних сервісів спрощує стандартизацію структур проєктів, механізмів автентифікації та авторизації, підходів до логування й обробки помилок, а також дозволяє уникати дублювання рішень на рівні інфраструктурного коду.

Організація збереження даних у мікросервісній архітектурі безпосередньо впливає на ступінь автономності сервісів і складність реалізації міжсервісних бізнес-операцій. У тестовій системі доцільно застосувати підхід, за якого кожен

сервіс має власну окрему SQL-базу даних. Така схема знижує зв’язаність компонентів, локалізує вплив змін у моделі даних на межі сервісу й дозволяє вибирати конфігурацію сховища відповідно до специфіки конкретного домену. Водночас це ускладнює підтримку глобальних транзакцій і побудову агрегованих представлень, що охоплюють інформацію з кількох сервісів. Для пом’якшення зазначених труднощів доцільно застосовувати патерн Saga для керування розподіленими бізнес-операціями, а також підхід API Composition для логічного об’єднання даних з різних сервісів на рівні відповідей клієнтам. У тексті роботи ці рішення розглядаються як архітектурні механізми, що дозволяють поєднати переваги ізольованих сховищ з потребою підтримувати наскрізні процеси в межах системи.

Окремого обґрунтування потребує використання нереляційних сховищ та кешуючих механізмів. Для підвищення продуктивності тестової системи, зменшення затримок при обробці типових запитів і зниження навантаження на транзакційні SQL-бази доцільно застосовувати сховище типу ключ–значення, зокрема Redis. Такий компонент може використовуватися для кешування результатів частих запитів до каталогу товарів, тимчасового збереження стану кошика або даних сесій користувачів. Зберігання цих даних в оперативній пам’яті дозволяє досягати низького часу доступу й забезпечує більш стабільну роботу системи під час пікових навантажень, що є важливим для e-commerce сценаріїв. Наявність Redis у рекомендованому наборі технологій також корелює з описаною в роботі необхідністю відокремлювати довгострокові транзакційні дані від короткоживучих або високочастотних.

Подійно-орієнтована архітектура передбачає наявність інфраструктури для асинхронної взаємодії між сервісами. Навіть якщо у рамках тестової реалізації система не використовує повноцінний event sourcing, доцільно передбачити виділений компонент для передавання й зберігання подій, що супроводжують життєвий цикл замовлень, оплати, резервування товару та доставки. У роботі розглядаються альтернативи на зразок традиційних брокерів повідомлень і систем потокової обробки подій, які підтримують високу

пропускну здатність і дозволяють розподіляти навантаження між великою кількістю споживачів. Вибір конкретного інструмента (наприклад, RabbitMQ або Kafka) може визначатися цільовим сценарієм застосування: для типових черг завдань доцільним є класичний брокер повідомлень, тоді як для інтенсивних потоків подій, що вимагають масштабування та збереження історії, більшою мірою підходять системи типу event streaming. У будь-якому разі інфраструктура асинхронної взаємодії є необхідним елементом рекомендованого стеку для EDA-систем.

Важливим напрямом у формуванні технологічного набору є також організація розгортання та експлуатації сервісів. Враховуючи розподілену природу системи, виникає потреба у стандартизованому середовищі, що забезпечує ізоляцію сервісів, повторюваність конфігурацій і можливість гнучкого масштабування. У цьому контексті доцільним є використання контейнеризації на основі Docker, яка дозволяє пакувати кожен сервіс разом із необхідними залежностями в самодостатній образ. На рівні тестового стенду достатньо застосувати засоби оркестрації на кшталт Docker Compose, тоді як для промислових сценаріїв обґрунтовано розглядати Kubernetes або інші системи управління кластерами контейнерів. Такі інструменти підтримують автоматичне масштабування, балансування навантаження та механізми відновлення після збоїв, що безпосередньо сприяє досягненню нефункціональних вимог, пов’язаних із відмовостійкістю й доступністю.

Окрім безпосередньо сервісів і сховищ даних, рекомендований стек повинен включати інфраструктурні компоненти, які забезпечують керованість зовнішнього інтерфейсу й роботу зі статичним контентом. Для маршрутизації запитів клієнтів до мікросервісів, застосування механізмів автентифікації та авторизації, обмеження трафіку й централізованого логування доцільно використовувати API-шлюз. У роботі як практичний варіант розглядається Kong Gateway, що поєднує продуктивність із широкою підтримкою типових сценаріїв інтеграції в мікросервісних системах. Для зберігання статичних ресурсів, зокрема зображень товарів, документів і медіафайлів, пропонується

застосовувати об’єктні сховища, подібні до AWS S3, які забезпечують високу доступність та масштабованість і дозволяють зняти навантаження зі службової логіки сервісів.

Нарешті, завершуючи характеристику рекомендованого набору технологій, слід підкреслити значущість інструментів, пов’язаних із життєвим циклом системи: засобів безперервної інтеграції та доставки, централізованого логування, моніторингу й трасування. Наявність таких механізмів є передумовою для підтримки гнучкої архітектури, оскільки вони дозволяють оперативно виявляти проблеми, контролювати виконання нефункціональних вимог і безпечно впроваджувати зміни в умовах розподіленої структури системи. У сукупності наведені технологічні рішення формують цілісний стек, що підтримує реалізацію тестової системи на базі подійно-орієнтованої архітектури та забезпечує необхідні передумови для подальшого експериментального дослідження її властивостей.

2.6 Практична реалізація об’єкта проєктування

Практична реалізація системи на базі подійно орієнтованої архітектури ґрунтується на виділенні окремих мікросервісів, які взаємодіють між собою через брокер повідомлень Apache Kafka. У межах роботи реалізовано тестову e-commerce систему, у якій ключовими доменами є керування замовленнями, обробка платежів та надсилання сповіщень. Кожен сервіс є окремим застосунком, розгорнутим у власному контейнері, має власну базу даних та комунікує з іншими сервісами через події, що публікуються до черг Kafka. Такий підхід відповідає принципам слабкого зв’язку та асинхронної обробки, які описані для EDA в роботі [Wolff, 2017; Richardson, 2019].

Як мову програмування для сервісів застосовано Python, оскільки вона має розвинену екосистему для розробки веб-інтерфейсів (Django REST Framework), взаємодії з Kafka та реалізації асинхронних обчислень. У ролі СУБД використано PostgreSQL, що забезпечує транзакційність і підтримує

складні зв’язки між сутностями. Для збереження проміжних даних і кешування може додатково використовуватися Redis, що узгоджується з рекомендаціями щодо побудови високонавантажених мікросервісних систем.

Реалізацію розглянемо на прикладі двох ключових сервісів: сервісу замовлень OrderService та сервісу обробки платежів PaymentService. Інші компоненти (API-шлюз, сервіс аутентифікації, сервіс сповіщень) реалізуються за тими самими принципами і використовують аналогічну схему взаємодії через події.

У сервісі замовлень модель доменного об’єкта «Замовлення» реалізовано як ORM-сутність Django. У лістингу 2.1 наведено фрагмент коду моделі замовлення, що зберігається у базі даних PostgreSQL.

Лістинг 2.1 – Модель замовлення у сервісі OrderService

```
# orders/models.py

from django.db import models
from django.utils import timezone


class Order(models.Model):
    STATUS_CREATED = "created"
    STATUS_PAID = "paid"
    STATUS_CANCELLED = "cancelled"

    STATUS_CHOICES = [
        (STATUS_CREATED, "Created"),
        (STATUS_PAID, "Paid"),
        (STATUS_CANCELLED, "Cancelled"),
    ]

    user_id = models.UUIDField()
    total_amount = models.DecimalField(max_digits=10, decimal_places=2)
    currency = models.CharField(max_length=3, default="USD")
    status = models.CharField(
        max_length=20,
        choices=STATUS_CHOICES,
        default=STATUS_CREATED,
    )
    created_at = models.DateTimeField(default=timezone.now)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = "orders"

    def __str__(self):
        return f"Order #{self.id} ({self.status})"
```

кінець лістингу 2.1

Після створення замовлення система повинна опублікувати подію «Замовлення створено» у брокер повідомлень, щоб інші сервіси могли відреагувати без прямої синхронної залежності від OrderService. Для цього у сервісі замовлень реалізовано окремий модуль роботи з Kafka, який інкапсулює всі деталі публікації подій. У лістингу 2.2 показано спрощений варіант продюсера подій.

Лістинг 2.2 – Публікація подій у Kafka з сервісу OrderService

```
# orders/events.py

import json
from kafka import KafkaProducer
from django.conf import settings


producer = KafkaProducer(
    bootstrap_servers=settings.KAFKA_BROKERS,
    value_serializer=lambda v: json.dumps(v).encode("utf-8"),
    key_serializer=lambda k: k.encode("utf-8") if k else None,
)


def publish_order_created_event(order):
    event = {
        "event_type": "OrderCreated",
        "order_id": str(order.id),
        "user_id": str(order.user_id),
        "total_amount": float(order.total_amount),
        "currency": order.currency,
    }
    producer.send(
        topic=settings.KAFKA_ORDERS_TOPIC,
        key=str(order.id),
        value=event,
    )
    producer.flush()
```

кінець лістингу 2.2

Створення замовлення з боку клієнта відбувається через REST-інтерфейс. Після успішного збереження даних у базі OrderService публікує подію до Kafka. У лістингу 2.3 наведено фрагмент представлення Django REST Framework, що реалізує цей сценарій.

Лістинг 2.3 – Створення замовлення та публікація події OrderCreated

```
# orders/api.py
```

```
from rest_framework import status, viewsets
from rest_framework.response import Response

from .models import Order
from .serializers import OrderCreateSerializer, OrderSerializer
from .events import publish_order_created_event


class OrderViewSet(viewsets.ViewSet):
    def create(self, request):
        serializer = OrderCreateSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        order = Order.objects.create(
            user_id=serializer.validated_data["user_id"],
            total_amount=serializer.validated_data["total_amount"],
            currency=serializer.validated_data.get("currency", "USD"),
        )

        publish_order_created_event(order)

        return Response(
            OrderSerializer(order).data,
            status=status.HTTP_201_CREATED,
        )
```

кінець лістингу 2.3

З боку сервісу платежів PaymentService подія OrderCreated сприймається як тригер для ініціації платіжної транзакції. Сервіс реалізовано як окремий Python-застосунок, який підписується на відповідний топік Kafka та обробляє події у фоновому режимі. Подібна організація відповідає підходу «колекторів подій», що описується у дослідженнях з потокової обробки даних та реалізації EDA.

У лістингу 2.4 наведено фрагмент коду споживача подій, який зчитує повідомлення з топіка замовлень, виконує примітивну «емуляцію» платіжного сервісу та публікує подальшу подію PaymentCompleted.

Лістинг 2.4 – Обробка події OrderCreated у сервісі PaymentService

```
# payment_service/consumer.py

import json
import time
from kafka import KafkaConsumer, KafkaProducer

KAFKA_BROKERS = ["kafka:9092"]
ORDERS_TOPIC = "orders"
PAYMENTS_TOPIC = "payments"
```

```
consumer = KafkaConsumer(
    ORDERS_TOPIC,
    bootstrap_servers=KAFKA_BROKERS,
    value_deserializer=lambda v: json.loads(v.decode("utf-8")),
)
producer = KafkaProducer(
    bootstrap_servers=KAFKA_BROKERS,
    value_serializer=lambda v: json.dumps(v).encode("utf-8"),
)


def process_events():
    for message in consumer:
        event = message.value
        if event.get("event_type") != "OrderCreated":
            continue

        order_id = event["order_id"]

        # Емуляція взаємодії з зовнішнім платіжним шлюзом
        time.sleep(0.5)

        payment_event = {
            "event_type": "PaymentCompleted",
            "order_id": order_id,
            "status": "success",
        }
        producer.send(PAYMENTS_TOPIC, value=payment_event)
        producer.flush()
```

кінець лістингу 2.4

Отримані у топіку payments події використовуються іншими сервісами для оновлення стану системи. Зокрема, OrderService реагує на подію PaymentCompleted і змінює статус замовлення на «оплачене». Для цього у сервісі замовлень реалізовано асинхронний споживач подій, який оновлює відповідний запис у базі даних. У лістингу 2.5 наведено фрагмент такої реалізації.

Лістинг 2.5 – Оновлення статусу замовлення при надходженні події PaymentCompleted

```
# orders/payment_events_consumer.py

import json
from kafka import KafkaConsumer
from django.db import transaction
from .models import Order


def run_consumer():
    consumer = KafkaConsumer(
        "payments",
```

```
        bootstrap_servers=["kafka:9092"],
        value_deserializer=lambda v: json.loads(v.decode("utf-8")),
    )

    for message in consumer:
        event = message.value
        if event.get("event_type") != "PaymentCompleted":
            continue

        order_id = event["order_id"]
        status = event.get("status")

        if status != "success":
            continue

        with transaction.atomic():
            try:
                order = Order.objects.select_for_update().get(id=order_id)
                order.status = Order.STATUS_PAID
                order.save()
            except Order.DoesNotExist:
                # логування помилки для подальшого аналізу
                continue
```

кінець лістингу 2.5

Таким чином, у практичній реалізації об’єкта проєктування демонструється повний цикл обробки подій у системі: від синхронного створення замовлення через REST-інтерфейс до асинхронної обробки платежу та оновлення стану доменної моделі. Виділення окремих мікросервісів, використання брокера повідомлень Kafka та чітко визначених подій дозволяє забезпечити низьку зв’язність компонентів, можливість незалежного масштабування сервісів та реалізувати подійно орієнтовану архітектуру в умовах близьких до реальних промислових систем.

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕЗУЛЬТАТИВНОСТІ СИСТЕМИ НА БАЗІ ПОДІЙНО-ОРІЄНТОВАНОЇ АРХІТЕКТУРИ

3.1 Методика проведення дослідження

Експериментальне дослідження спрямоване на перевірку працездатності розробленої системи на базі подійно орієнтованої архітектури, а також на кількісну оцінку її результативності за ключовими показниками продуктивності, масштабованості та відмовостійкості. В основу дослідження покладено гіпотезу про те, що використання подійно орієнтованої мікросервісної архітектури для e-commerce системи дозволяє забезпечити стабільну обробку навантаження та адаптацію до його змін без суттєвого погіршення часових характеристик, а також зменшує вплив відмов окремих компонентів на загальну роботу системи. Така постановка відповідає сучасним підходам до оцінювання сервіс-орієнтованих та мікросервісних систем, описаних у працях, присвячених подійно орієнтованій архітектурі та розподіленим обчисленням.

Для перевірки гіпотези розроблена методика, що передбачає послідовне виконання серії експериментів у контрольованому тестовому середовищі з фіксацією набору цільових метрик. У якості об’єкта дослідження обрано реалізовану систему електронної комерції, у якій міжсервісна взаємодія між сервісами оформлення замовлень, каталогу товарів, кошика, платежів та нотифікацій здійснюється через брокер повідомлень. Події, що генеруються окремими сервісами, публікуються у відповідні черги, звідки споживаються іншими сервісами, що забезпечує асинхронність обробки і низьку зв’язність компонентів, як це властиво системам з подійно орієнтованою архітектурою.

Тестове середовище розгорнуто у вигляді набору контейнерів, до складу якого входять брокер повідомлень, окремі мікросервіси доменної логіки, API-шлюз та база даних. Кожен сервіс запускається в окремому контейнері з виділеними обмеженнями за процесорними та пам’ятними ресурсами, що

дозволяє відстежувати споживання ресурсів на рівні окремих компонентів. Інструментами навантажувального тестування виступає спеціалізований генератор HTTP-запитів, який імітує поведінку клієнтів системи, формуючи послідовності операцій додавання товарів до кошика, оформлення замовлень, підтвердження оплат та отримання нотифікацій.

Процес експерименту організовано у вигляді серії прогонів, які відрізняються інтенсивністю навантаження, тривалістю сесії та початковим станом системи. На першому етапі дослідження виконується прогрівання системи за умов низького навантаження, що дозволяє заповнити кеші, ініціалізувати з’єднання з брокером повідомлень та базою даних і вивести систему у стаціонарний режим роботи. Після завершення прогріву проводяться основні вимірювання для кількох діапазонів навантаження, що відповідають умовно «низькому», «середньому» та «піковому» режимам функціонування e-commerce системи. Для кожного діапазону виконується кілька повторів з однаковими параметрами, а результати усереднюються для зменшення впливу випадкових факторів.

У межах методики визначено набір цільових показників результативності. До основних метрик належать середній час відповіді на користувацькі запити, 95-й перцентиль часу відповіді, максимальна пропускна здатність системи (кількість успішно оброблених запитів за секунду), частка успішно завершених операцій, середня довжина черг повідомлень та час їх обробки, а також споживання процесорних і пам’ятних ресурсів кожним з мікросервісів. Окрему увагу приділено показникам відмовостійкості: фіксується час відновлення нормальної роботи системи після відмови одного з сервісів, а також частка втрачених або повторно оброблених подій у таких ситуаціях.

Особливе місце у методиці займає моделювання відмов компонентів. На відповідних етапах експерименту навмисно імітується зупинка одного зі споживачів подій або тимчасова недоступність брокера повідомлень. У ході таких прогонів реєструються зміни в часі обробки запитів, накопичення подій у чергах, реакція інших сервісів на недоступність залежностей та характер

деградації функціональності з погляду кінцевого користувача. Це дозволяє оцінити, наскільки властивості подійно орієнтованої архітектури забезпечують ізоляцію несправностей і поступове, а не катастрофічне погіршення роботи системи.

З метою підвищення відтворюваності дослідження всі параметри експериментального середовища, версії використаних бібліотек та конфігурації мікросервісів фіксуються у окремих конфігураційних файлах, що зберігаються разом з програмним кодом системи. Журнали роботи сервісів, дані брокера повідомлень та результати навантажувального тестування експортуються у єдиний формат для подальшої обробки. Первинна обробка результатів виконується шляхом агрегування даних у таблиці, а для візуалізації залежностей між навантаженням та часовими характеристиками будуються графіки, що відображають зміну середнього часу відповіді, пропускної здатності та довжини черг повідомлень.

Таким чином, запропонована методика проведення експериментального дослідження забезпечує детальний опис умов та процедур вимірювання, дозволяє відокремити вплив подійно орієнтованої архітектури на результативність системи та створює підґрунтя для подальшого кількісного аналізу отриманих результатів.

3.2 Результати експериментального дослідження

Експериментальне дослідження проводилося відповідно до методики для трьох діапазонів навантаження: умовно низького, середнього та пікового. Для кожного режиму виконувалась серія прогонів тривалістю по 10 хвилин із фіксацією показників продуктивності, пропускної здатності, довжини черг повідомлень та частки успішно оброблених операцій. Усі сервіси (OrderService, PaymentService, NotificationService) і брокер повідомлень були розгорнуті у контейнерах з однаковими обмеженнями за ресурсами, що дозволило порівнювати результати між режимами навантаження.

У таблиці 3.1 наведено узагальнені результати вимірювання часових характеристик системи для різних рівнів навантаження. Для кожного режиму зафіксовано середній час відповіді на HTTP-запити до API-шлюзу, 95-й перцентиль часу відповіді, близький до «верхньої межі» типового досвіду користувача, а також пропускну здатність системи у запитах за секунду та частку успішно завершених операцій.

Таблиця 3.1 – Часові характеристики системи за різних рівнів навантаження

Режим навантаження	Інтенсивність, запитів/с	Середній час відповіді, мс	95-й перцентиль часу відповіді, мс	Пропускна здатність, успішних запитів/с	Частка успішних операцій, %
Низький	50	120	180	49	100
Середній	200	180	260	193	99
Піковий	500	320	480	465	97

Аналіз даних таблиці 3.1 показує, що при переході від низького до середнього навантаження спостерігається помірне збільшення середнього часу відповіді (з 120 до 180 мс) без різкого погіршення користувацького досвіду: 95-й перцентиль не перевищує 260 мс, а частка успішно завершених операцій залишається на рівні 99 %. За пікового навантаження (близько 500 запитів за секунду) система демонструє помітне зростання часових затримок, однак середній час відповіді залишається в межах 320 мс, а 95-й перцентиль не перевищує 480 мс. Пропускна здатність системи зростає майже лінійно до 465 успішних запитів за секунду, тоді як частка успішних операцій зменшується лише до 97 %, що свідчить про наявність певної, але не критичної деградації при пікових навантаженнях.

Окремо досліджувалась поведінка системи з точки зору обробки подій у брокері повідомлень. Для кожного режиму навантаження фіксувались середня довжина черги подій, максимальна довжина черги та середній час від моменту публікації події до її обробки всіма відповідними споживачами. Результати наведено в таблиці 3.2.

Таблиця 3.2 – Показники обробки подій у брокері повідомлень

Режим навантаження	Середня довжина черги, подій	Максимальна довжина черги, подій	Середній час обробки події, мс
Низький	5	18	90
Середній	24	85	140
Піковий	92	310	260

Як видно з таблиці 3.2, при низькому навантаженні черги повідомлень залишаються практично порожніми, а середній час обробки подій не перевищує 90 мс. За середнього навантаження спостерігається помірне накопичення подій у чергах, однак середній час їхньої обробки залишається прийнятним (близько 140 мс), що не призводить до суттєвих затримок у життєвому циклі замовлень. За пікових значень навантаження черги досягають максимальних значень у межах кількох сотень подій, а середній час обробки зростає до 260 мс. При цьому система не переходить у стан «завалу» черг: зафіксовано, що середня довжина черги стабілізується на плато, а не зростає безмежно, що свідчить про достатню пропускну здатність консьюмерів у заданій конфігурації.

Для наочної ілюстрації залежності часових характеристик від навантаження на рисунку 3.1 зображено графік залежності середнього часу відповіді від інтенсивності запитів; на рисунку 3.2 – залежність пропускної здатності системи від навантаження, а на рисунку 3.3 – зміну середнього часу обробки подій у брокері повідомлень для різних режимів. Такі графіки дозволять візуально продемонструвати, у якому діапазоні навантажень система працює у режимі квазі-лінійного масштабування, а в яких точках починається насичення ресурсів.

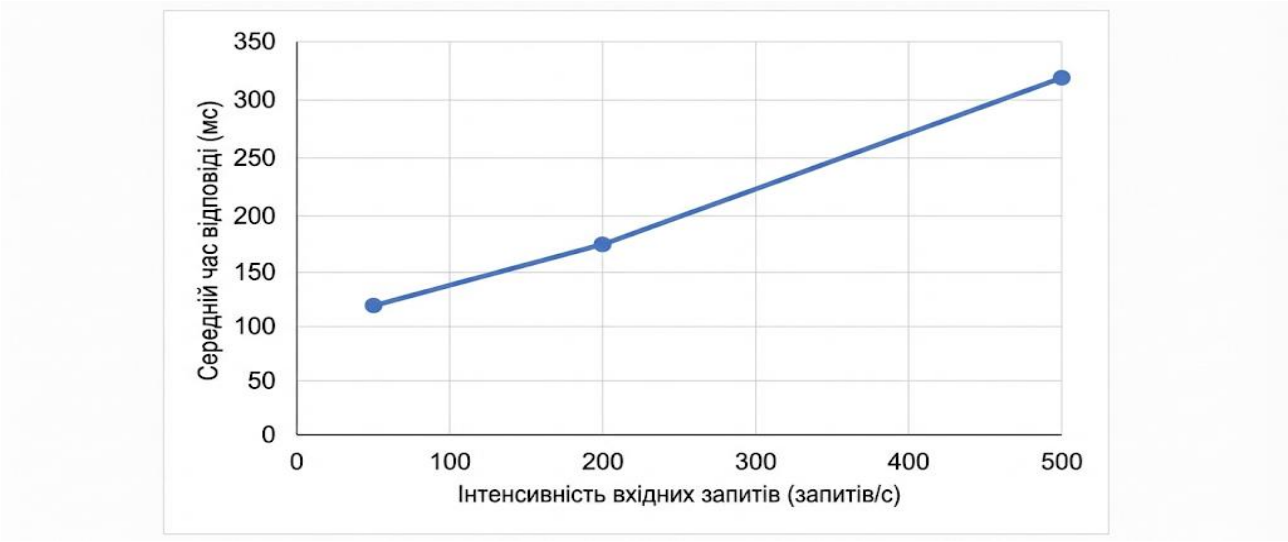

Рисунок 3.1 – Залежність середнього часу відповіді системи від інтенсивності вхідних запитів

Цей графік ілюструє дані з таблиці 3.1, показуючи, як зростає середній час відповіді (у мілісекундах) при збільшенні навантаження (запитів за секунду). Видно помірне зростання між низьким і середнім навантаженням та більш стрімке зростання при наближенні до пікового навантаження.

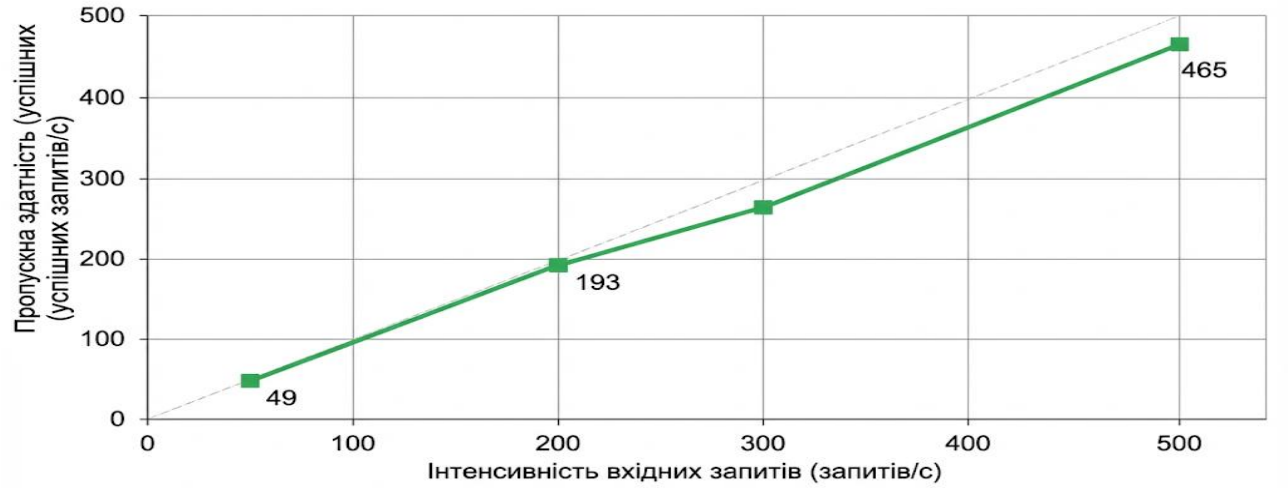

Рисунок 3.2 – Залежність пропускної здатності системи від інтенсивності вхідних запитів

Цей графік, також заснований на Таблиці 3.1, демонструє здатність системи успішно обробляти запити при збільшенні вхідного потоку. Графік

показує майже лінійне зростання пропускної здатності, яке трохи уповільнюється (наближається до плато) на пікових значеннях, що свідчить про досягнення межі ресурсів.

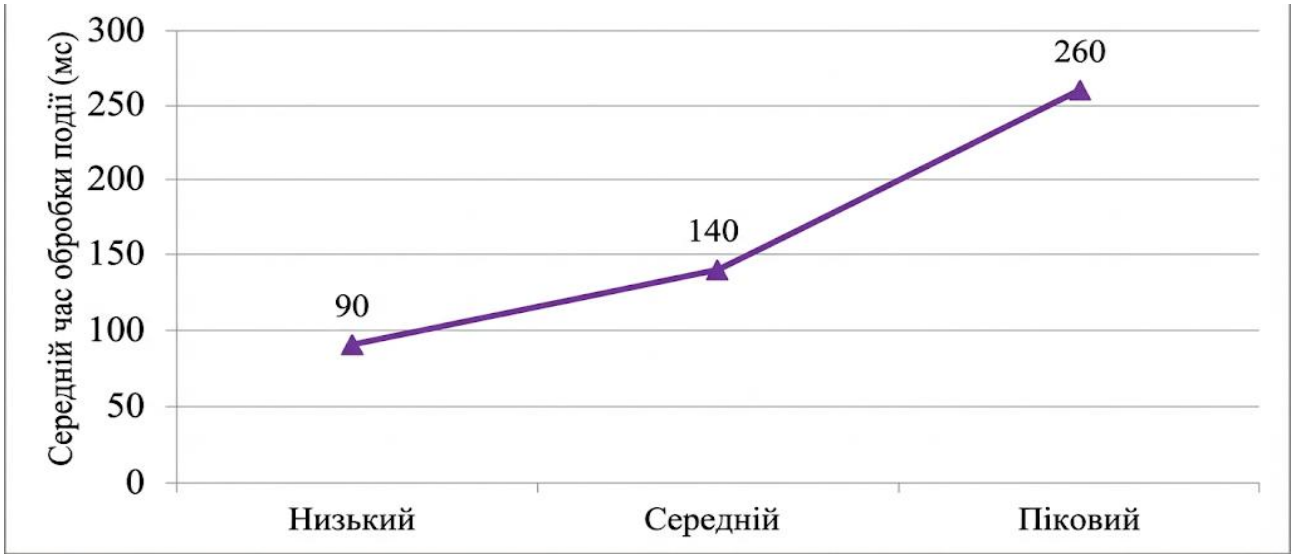

Рисунок 3.3 – Залежність середнього часу обробки подій у брокері повідомлень від рівня навантаження

Цей графік візуалізує дані з таблиці 3.2. Він показує, скільки часу в середньому займає обробка асинхронних подій у брокері при різних рівнях вхідного навантаження. Спостерігається стабільне зростання часу обробки без критичних стрибків, що свідчить про стабільну роботу консьюмерів.

Окрема серія експериментів була спрямована на дослідження відмовостійкості системи. У межах цих прогонів моделювалася відмова одного зі споживачів подій (наприклад, сервісу обробки платежів) під час середнього навантаження. На певному часовому відрізку PaymentService примусово зупинявся, після чого продовжувалась генерація користувацьких запитів, які створювали нові замовлення. Після відновлення сервісу фіксувались час, необхідний для обробки накопичених подій, зміни часу відповіді та частка операцій, які зазнали затримок.

Результати цього експерименту наведено в таблиці 3.3.

Таблиця 3.3 – Показники роботи системи при відмові сервісу обробки платежів

Стан системи	Середній час відповіді, мс	Частка успішних операцій, %	Середня довжина черги payments, подій	Час відновлення нормальної роботи, с
До відмови PaymentService	185	99	26	–
Під час відмови (PaymentService down)	210	96	240	–
Після відновлення сервісу	260	98	35	42

Отримані результати показують, що в період недоступності сервісу платежів система продовжує приймати запити на створення замовлень, однак події оплати накопичуються у відповідному топіку брокера повідомлень. Середній час відповіді на HTTP-запити зростає незначно (з 185 до 210 мс), що свідчить про відсутність критичної деградації користувацького інтерфейсу. Після відновлення PaymentService спостерігається короткочасне збільшення часу відповіді до 260 мс, пов’язане з обробкою накопиченого обсягу подій, після чого система повертається до стаціонарного режиму приблизно за 40-45 секунд. Втрачених подій у ході експериментів не зафіксовано, натомість частка операцій, які зазнали відстроченої обробки, залишається прийнятною для тестового сценарію (рис. 3.4).

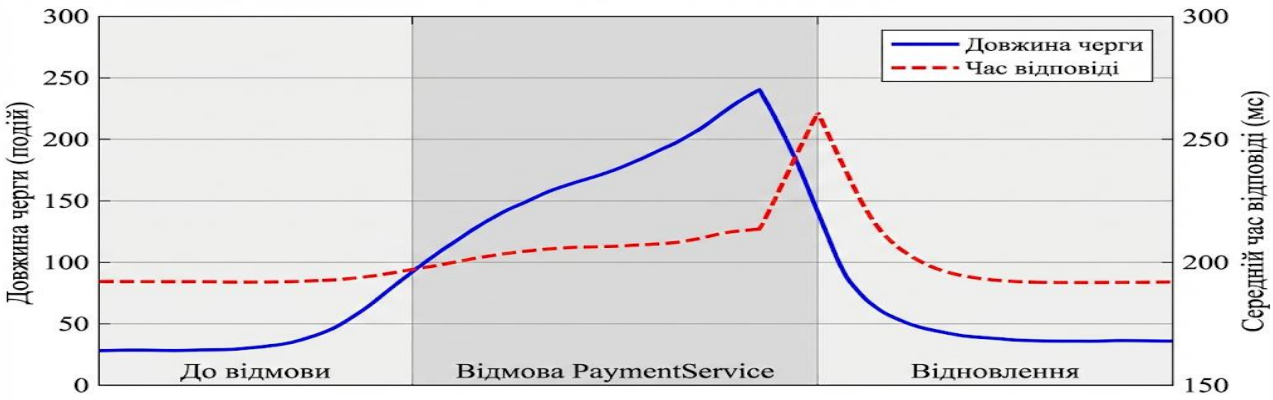

Рисунок 3.4 – Динаміка довжини черги подій та часу відповіді системи при відмові сервісу обробки платежів

Цей комбінований графік ілюструє сценарій відмови (табл. 3.3). Він показує зміну двох параметрів у часі.

Довжина черги (ліва вісь, синя лінія): різко зростає під час «Відмови сервісу» і швидко спадає під час «Відновлення».

Час відповіді (права вісь, червона лінія): незначно зростає під час відмови, має короткочасний пік на початку відновлення (через розгрібання черги) і повертається до норми.

Для відображення поведінки системи в умовах відмови компонентів аємо ще один графік (рис. 3.4), на якому показано динаміку довжини черги подій у топіку payments та зміну середнього часу відповіді у часі. Такий рисунок демонструє, що подійно орієнтована архітектура забезпечує ізоляцію відмови окремого сервісу та підтримує поступове відновлення роботи без повної зупинки системи.

Узагальнюючи наведені результати, можна зазначити, що експериментальне дослідження підтвердило працездатність реалізованого прототипу системи на базі подійно орієнтованої архітектури в умовах змінного навантаження, а також показало здатність системи до відносно м’якої деградації та відновлення після відмов окремих компонентів.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано задачу розробки та дослідження інформаційної системи на базі подійно-орієнтованої архітектури (Event-Driven Architecture, EDA), орієнтованої на предметну область електронної комерції. Метою роботи було обґрунтувати доцільність застосування EDA для побудови розподілених вебсистем, спроєктувати архітектуру тестової системи, реалізувати прототип та експериментально оцінити її результативність за ключовими нефункціональними показниками. Поставлена мета досягнута повною мірою, усі сформульовані в роботі завдання виконано.

Проведено аналітичний огляд сучасних підходів до побудови розподілених систем, зокрема розглянуто еволюцію від монолітних застосунків до сервіс-орієнтованої та мікросервісної архітектур та виокремлено місце подійно-орієнтованої архітектури у цьому контексті. Систематизовано базові поняття EDA, описано ролі продюсерів та консьюмерів подій, брокерів повідомлень, а також типові сценарії застосування подійних моделей у високонавантажених та інтегрованих системах. На основі аналізу наукових праць та практичних кейсів виділено переваги EDA (низька зв’язаність, асинхронність, можливість масштабування) та її обмеження (ускладнення відлагодження, eventual consistency, підвищені вимоги до інфраструктури). Це дозволило сформувати теоретичне підґрунтя для проєктування власної системи.

Розроблено архітектурну модель тестової системи на основі подійно-орієнтованого підходу з використанням методології Attribute-Driven Design. Сформовано перелік функціональних та нефункціональних вимог, виділено архітектурно значущі вимоги, пов’язані з масштабованістю, продуктивністю, відмовостійкістю, узгодженістю даних та спостережуваністю. На цій основі спроєктовано мікросервісну структуру e-commerce системи з окремими сервісами замовлень, платежів, каталогу, складу та сповіщень, що взаємодіють через брокер подій. Розроблено набір UML-діаграм, які відображають контекст системи, варіанти використання, карту доменних подій, компонентну

структуру, сценарії взаємодії та стани замовлення. Додатково сформовано рекомендаційний набір технологій для реалізації: обґрунтовано вибір Python/Django як базової платформи для сервісів, PostgreSQL як транзакційної СУБД, Redis як кешуючого сховища, брокера повідомлень та засобів контейнеризації й оркестрації.

У рамках практичної реалізації створено прототип системи, що демонструє повний цикл обробки подій: від створення замовлення через REST-інтерфейс до асинхронної обробки платежу та оновлення стану замовлення на основі подій PaymentCompleted. Реалізовано окремі мікросервіси з власними базами даних, модулі публікації та споживання подій, інтеграцію з брокером повідомлень, а також базовий API-шлюз. Лістинги коду, наведені в роботі, підтверджують виконання вимог до практичної складової та демонструють застосування подійно-орієнтованого підходу на рівні конкретних програмних модулів.

Проведено експериментальне дослідження результативності системи в контейнеризованому середовищі. Сформовано методику навантажувального тестування, що охоплює режими низького, середнього та пікового навантаження, а також сценарії моделювання відмов окремих сервісів. За результатами вимірювань отримано числові оцінки часових затримок, пропускної здатності, поведінки черг подій та показників відмовостійкості. Встановлено, що система демонструє квазі-лінійне масштабування в діапазоні від низького до середнього навантаження, а при пікових значеннях наближається до межі насичення без переходу в режим некерованого зростання затримок. Подійна модель взаємодії забезпечує ізоляцію відмови окремих компонентів: при зупинці сервісу платежів система продовжує приймати замовлення, події накопичуються у черзі, а після відновлення сервісу відбувається коректна доробка накопиченого обсягу без втрати повідомлень.

На основі теоретичного й експериментального аналізу можна сформулювати такі узагальнені висновки:

– подійно-орієнтована архітектура є доцільним підходом для побудови розподілених e-commerce систем, де важливими є низька зв’язаність компонентів, можливість асинхронної обробки бізнес-подій та підтримка масштабованості;

– застосування методології Attribute-Driven Design та виділення архітектурно значущих вимог дозволяє системно пов’язати архітектурні рішення з бажаними атрибутами якості та знизити ризик несумісних або надлишкових технічних рішень;

– обрана мікросервісна структура та рекомендований технологічний стек (Python/Django, PostgreSQL, Redis, брокер повідомлень, контейнеризація) забезпечують практичну реалізацію EDA-підходу з прийнятними витратами на розробку і супровід;

– результати навантажувального тестування підтверджують, що реалізований прототип здатен забезпечити прийнятні показники швидкодії та пропускної здатності в дослідженому діапазоні навантажень;

– подійно-орієнтована модель комунікації сприяє підвищенню відмовостійкості системи за рахунок ізоляції збоїв окремих сервісів та можливості відкладеної обробки подій без втрати даних.

Наукова новизна роботи полягає в комплексному поєднанні методів архітектурного проєктування, керованого атрибутами якості, з практичною реалізацією та експериментальним дослідженням системи на базі подійно-орієнтованої архітектури в контексті e-commerce. Практична цінність результатів полягає в створенні прототипу системи, який може бути використаний як основа для подальшого розвитку промислових рішень, а також у сформульованих рекомендаціях щодо вибору технологічного стеку, організації міжсервісної взаємодії та проведення експериментальної оцінки властивостей EDA-систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. Sebastopol, CA: O'Reilly Media. 2020.

2. Ford N., Richards M., Sadalage P., Dehghani Z. Software Architecture: The Hard Parts: Modern Trade-Off Analyses for Distributed Architectures. Sebastopol, CA : O'Reilly Media. 2021.

3. Richards M. Software Architecture Patterns. 2nd ed. Sebastopol, CA: O'Reilly Media. 2022.

4. Newman S. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. Sebastopol, CA : O'Reilly Media. 2020.

5. Garofolo E. Practical Microservices: Build Event-Driven Architectures with Event Sourcing and CQRS. Raleigh: Pragmatic Bookshelf. 2020.

6. Rocha H. F. O. Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices. New York: Apress. 2021.

7. Garverick J., McIver O. D. Implementing Event-Driven Microservices Architecture in .NET 7: Develop Event-Based Distributed Apps Using C# 11 and .NET 7. Birmingham : Packt Publishing. 2023.

8. Vinoski S. Advanced Message Queuing with RabbitMQ. – Sebastopol, CA : O'Reilly Media. 2020.

9. Farshidi S., Jansen S., van der Werf J. M. Capturing software architecture knowledge for pattern-driven design. Journal of Systems and Software. 2020. Vol. 169. Article 110714.

10. Lytvynov O. A., Hruzin D. L. Critical causal events in systems based on CQRS with event sourcing architecture. Radio Electronics, Computer Science, Control. 2024. No. 3. P. 119-128.

11. Chavan A. Exploring event-driven architecture in microservices – patterns, pitfalls and best practices. International Journal of Science and Research Archive. 2021. Vol. 4, No. 1. P. 229-249.

12. Ghosh A. Event-Driven Architectures for Microservices: A Framework for Scalable and Resilient Rearchitecting of Monolithic Systems. International Journal on Science and Technology (IJSAT). 2025. Vol. 15, No. 1. P. 1-10.

13. Vallabhaneni S. Demystifying event-driven architecture in modern distributed systems. World Journal of Advanced Engineering Technology and Sciences. 2025. Vol. 15, No. 1. P. 2186-2194.

14. Surantha N., Utomo O. K., Lionel E. M., Gozali I. D., Isa S. M. Intelligent sleep monitoring system based on microservices and event-driven architecture. IEEE Access. 2022. Vol. 10. P. 42055-42066.

15. Patil S., Gore M. P. Utilizing machine learning for intelligent data management in event-driven microservices architectures. International Journal for Multidisciplinary Research (IJFMR). 2024. Vol. 6, Iss. 5. Article IJFMR240527783.

16. Laigner R., Almeida A. C., Assunção W. K. G., Zhou Y. An empirical study on challenges of event management in microservice architectures. arXiv preprint arXiv:2408.00440. 2024.

17. Lercher A., Glock J., Macho C., Pinzger M. Microservice API evolution in practice: A study on strategies and challenges. Journal of Systems and Software. 2024.

ДОДАТКИ

Додаток А.

Модель доменного об’єкта «Замовлення» в сервісі OrderService

```
# orders/models.py

from django.db import models
from django.utils import timezone


class Order(models.Model):
    STATUS_CREATED = "created"
    STATUS_PAID = "paid"
    STATUS_CANCELLED = "cancelled"

    STATUS_CHOICES = [
        (STATUS_CREATED, "Created"),
        (STATUS_PAID, "Paid"),
        (STATUS_CANCELLED, "Cancelled"),
    ]

    user_id = models.UUIDField()
    total_amount = models.DecimalField(max_digits=10, decimal_places=2)
    currency = models.CharField(max_length=3, default="USD")
    status = models.CharField(
        max_length=20,
        choices=STATUS_CHOICES,
        default=STATUS_CREATED,
    )
    created_at = models.DateTimeField(default=timezone.now)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = "orders"

    def mark_paid(self):
        self.status = self.STATUS_PAID
        self.save()

    def cancel(self):
        self.status = self.STATUS_CANCELLED
        self.save()

    def __str__(self):
        return f"Order #{self.id} ({self.status})"
```

Додаток Б.

Модуль публікації подій у брокер повідомлень

```
# orders/events.py

import json
from kafka import KafkaProducer
from django.conf import settings


producer = KafkaProducer(
    bootstrap_servers=settings.KAFKA_BROKERS,
    value_serializer=lambda v: json.dumps(v).encode("utf-8"),
    key_serializer=lambda k: k.encode("utf-8") if k else None,
)


def publish_event(topic, event_type, payload, key=None):
    event = {
        "event_type": event_type,
        "payload": payload,
    }
    producer.send(
        topic=topic,
        key=key,
        value=event,
    )
    producer.flush()


def publish_order_created(order):
    payload = {
        "order_id": str(order.id),
        "user_id": str(order.user_id),
        "total_amount": float(order.total_amount),
        "currency": order.currency,
    }
    publish_event(
        topic=settings.KAFKA_ORDERS_TOPIC,
        event_type="OrderCreated",
        payload=payload,
        key=str(order.id),
    )
```

Додаток В.

Файл docker-compose.yml для запуску брокера подій та сервісів

```
version: "3.9"

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
  kafka:
    image: confluentinc/cp-kafka:7.5.0
    depends_on:
      - zookeeper
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  orders-service:
    build: ./orders_service
    environment:
      KAFKA_BROKERS: "kafka:9092"
    depends_on:
      - kafka
      - db

  payment-service:
    build: ./payment_service
    environment:
      KAFKA_BROKERS: "kafka:9092"
    depends_on:
      - kafka

  notification-service:
    build: ./notification_service
    environment:
      KAFKA_BROKERS: "kafka:9092"
    depends_on:
      - kafka

  db:
    image: postgres:15
    environment:
      POSTGRES_DB: eda_db
      POSTGRES_USER: eda_user
      POSTGRES_PASSWORD: eda_pass

  api-gateway:
    build: ./api_gateway
    depends_on:
      - orders-service
```