НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ПОГОДЖЕНО **Декан факультету**	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ** **Завідувач кафедри комп'ютерних наук**
_____ Ігор БОЛБОТ (підпис)	**_____ Белла ГОЛУБ** (підпис)
___» _____ 2026 р.	___» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Програмне забезпечення системи з мікросервісною архітектурою для маркетплейсів»

Спеціальність «121 Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

Гарант освітньої програми
к.т.н., доцент **_____** **Ганна ВАЙГАНГ**
(підпис)

Керівник бакалаврської кваліфікаційної роботи **_____** **Світлана ВАСИЛЮК-ЗАЙЦЕВА**
(підпис)
ст. Викладач

Виконав **_____** **Олександр АНДРУЩЕНКО**
(підпис)

Київ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук

к.т.н., доцент _____ Белла ГОЛУБ
(підпис)

«___» _____ 2025 р.

ЗАВДАННЯ
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Андрущенку Олександру Олександровичу
(прізвище, ім'я, по батькові)

Спеціальність «121 Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи

« **Програмне забезпечення системи з мікросервісною архітектурою для маркетплейсів** »

затверджена наказом від «19» грудня 2025 р. № 3196 «С»
Термін подання завершеної роботи на кафедру «22» травня 2026 р.
Вихідні дані до бакалаврської кваліфікаційної роботи (проєкту):
Дослідження порівняння алгоритмів шифрування
Перелік питань, що підлягають дослідженню:
Аналіз предметної області, проектування і розробка програмного забезпечення.
Перелік графічних документів (за необхідності).
_____
Дата видачі завдання «19» грудня 2025 р.

Керівник бакалаврської кваліфікаційної ст. Викладач _____ **Світлана ВАСИЛЮК-ЗАЙЦЕВА**
(підпис)

Завдання взяв до виконання _____ **Олександр АНДРУЩЕНКО**
(підпис)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API (Application Programming Interface) — прикладний програмний інтерфейс.

BPMN (Business Process Model and Notation) — нотація моделювання бізнес-процесів.

CI/CD (Continuous Integration / Continuous Delivery) — безперервна інтеграція та доставка.

DTO (Data Transfer Object) — об'єкт передачі даних.

HTTP (HyperText Transfer Protocol) — протокол передачі гіпертексту.

JSON (JavaScript Object Notation) — текстовий формат обміну даними.

JWT (JSON Web Token) — стандарт автентифікаційного токена.

K8s — скорочена назва Kubernetes.

ORM (Object-Relational Mapping) — об'єктно-реляційне відображення.

REST (Representational State Transfer) — архітектурний стиль побудови розподілених систем.

SaaS (Software as a Service) — модель надання програмного забезпечення як послуги.

SLA (Service Level Agreement) — угода про рівень сервісу.

SPA (Single Page Application) — односторінковий веб-застосунок.

SQL (Structured Query Language) — мова структурованих запитів.

TLS (Transport Layer Security) — протокол захисту транспортного рівня.

UML (Unified Modeling Language) — уніфікована мова моделювання.

UUID (Universally Unique Identifier) — універсальний унікальний ідентифікатор.

БД — база даних.

ОС — операційна система.

ПЗ — програмне забезпечення.

ППЗ — прикладне програмне забезпечення.

СУБД — система управління базами даних.

ВСТУП

Електронна комерція є однією з найдинамічніших галузей сучасної економіки. Останніми десятиліттями роздрібна торгівля переживає фундаментальну трансформацію: значна частина транзакцій переміщується з фізичних магазинів у цифровий простір. Центральним явищем цієї трансформації стали маркетплейси – електронні торговельні майданчики, що об’єднують у єдиному просторі тисячі продавців та мільйони покупців. Маркетплейси формують нову бізнес-модель посередництва, відмінну як від класичних інтернет-магазинів, так і від офлайн-роздрібних мереж.

Розробка маркетплейсу – складна інженерна задача, що передбачає одночасну роботу з великими обсягами товарного асортименту, високим конкурентним навантаженням на пошук, складною логікою оплати, доставки та повернень, а також забезпеченням взаємної довіри між анонімними учасниками транзакцій. Класична монолітна архітектура, в межах якої вся бізнес-логіка реалізується як єдиний застосунок, виявляється недостатньо ефективною для маркетплейсів промислового масштабу. Зростання кодової бази робить її важкою для розуміння та модифікації, а вертикальне масштабування одного серверного процесу натрапляє на природні обмеження продуктивності.

Альтернативою монолітному підходу є мікросервісна архітектура – спосіб організації програмного забезпечення, за якого система розкладається на сукупність невеликих, незалежно розгортуваних сервісів. Кожен мікросервіс має чітко окреслену зону відповідальності, власне сховище даних та визначений програмний інтерфейс взаємодії з іншими сервісами. Такий підхід дозволяє командам розробників працювати паралельно над різними сервісами, незалежно випускати оновлення та ефективно масштабувати лише ті частини системи, що відчувають найбільше навантаження.

Незважаючи на широке поширення обох концепцій – електронних маркетплейсів та мікросервісної архітектури – їх поєднання у єдиному програмному рішенні залишається нетривіальною інженерною задачею.

Розробнику необхідно прийняти десятки взаємопов’язаних рішень: визначити межі мікросервісів, обрати моделі взаємодії між ними, забезпечити цілісність даних за відсутності розподілених транзакцій, побудувати спільну систему автентифікації та авторизації, налагодити CI/CD-конвеєри. Усе це визначає актуальність та практичну значущість бакалаврської кваліфікаційної роботи.

Об’єкт дослідження — процес побудови програмного забезпечення електронних торговельних майданчиків з підтримкою високого навантаження та горизонтальної масштабованості.

Предмет дослідження — методи та архітектурні рішення для проектування системи з мікросервісною архітектурою у предметній області електронної комерції.

Мета роботи — проектування та реалізація програмного забезпечення електронного маркетплейсу з мікросервісною архітектурою, що забезпечує повний цикл операцій від каталогізації товарів до оформлення замовлення та видачі покупки.

Для досягнення поставленої мети визначено такі завдання дослідження:

1) здійснити опис предметної області з урахуванням її специфіки;
2) розробити модель предметної області (електронного маркетплейсу);
3) провести огляд існуючих рішень у сфері електронних маркетплейсів, виявити їх переваги, недоліки та архітектурні особливості;
4) обгрунтувати постановку задачі задля реалізації аналізу предметної області;
5) побудувати узагальнену логічну модель даних системи;
6) аргументувати вибір СУБД як ключового елемента архітектурних рішень мікросервісної системи;
7) пояснити принципи побудови фізичної структури бази даних;
8) з’ясувати архітектурні особливості системи програмного забезпечення електронного маркетплейсу;
9) пояснити вибір методичного інструментарію розробки;

10) визначити послідовний алгоритм оформлення замовлення та ключові модулі;

11) розробити інтерфейс користувача MerchFlow;

12) провести функціональне тестування системи та підготувати її до розгортання у хмарі;

13) розробити склад інсталяційного пакету для встановлення програмного продукту.

У процесі розробки використано такі **методи та технології**:

1) об’єктно-орієнтоване проектування із застосуванням мови моделювання UML;

2) мікросервісна архітектура зі шлюзом API (API Gateway);

3) реалізація на основі Node.js та Express.js;

4) контейнеризація через Docker та оркестрація Kubernetes;

5) реляційна СУБД PostgreSQL для транзакційних даних;

6) документоорієнтована MongoDB для каталогу;

7) in-memory сховище Redis для кешу та сесій;

8) черга повідомлень RabbitMQ для асинхронної взаємодії;

9) клієнтська частина на React;

10) автентифікація через JWT.

Практичне значення отриманих результатів полягає у тому, що розроблений маркетплейс може бути використаний як готове SaaS-рішення або база для розгортання реального бізнесу.

Апробація результатів роботи та публікації. Основні положення та результати дослідження викладено у тезах доповіді: «Програмне забезпечення системи з мікросервісною архітектурою для маркетплейсів», які подано до збірника наукових праць факультету інформаційних технологій Національного університету біоресурсів і природокористування України.

На підставі участі у 5 Міжнародній конференції «Future of Work: Technological, Generational and Social Shifts» (Дніпро, 7-8 травня 2026 р.) опубліковано тези доповіді:

1) Андрущенко О.О. Проєктування мультифункціональної вебплатформи довідкового обслуговування в сучасному інформаційному середовищі. Future of Work: Technological, Generational and Social Shifts: Proceedings of the 5th International Scientific and Practical Internet Conference (Dnipro, May 7-8, 2026). Dnipro: FOP Marenichenko V.V., 2026. С.26-27.

Структура та обсяг роботи. Пояснювальна записка складається зі вступу, чотирьох розділів основної частини, висновків та списку використаних джерел. Загальний обсяг — близько 75 сторінок основного тексту. Список використаних джерел містить 32 найменування.

У першому розділі виконано аналіз предметної області: розглянуто сутність та класифікацію електронних маркетплейсів, побудовано моделі предметної області в нотації UML, проведено огляд існуючих рішень з порівняльним аналізом, сформульовано постановку задачі.

У другому розділі описано проектування інформаційного забезпечення: побудовано логічну модель даних, обґрунтовано вибір комбінації СУБД у моделі поліглот-персистентності, наведено фізичну структуру баз даних кожного мікросервісу.

У третьому розділі описано розробку програмного забезпечення: архітектуру системи з декомпозицією на мікросервіси, UML-діаграми класів, кооперацій, пакетів та компонентів, вибір інструментарію та реалізацію ключових модулів.

У четвертому розділі наведено відомості щодо тестування системи, вимоги до апаратного та програмного забезпечення, а також порядок розгортання системи на платформі Kubernetes.

Окремої уваги заслуговує економічна сторона мікросервісної архітектури. Перехід від моноліту до мікросервісів вимагає значних інвестицій у DevOps-інфраструктуру, навчання команди та переробку процесів розробки. За оцінками авторитетних джерел, мінімальний розмір команди, для якої виправдовується мікросервісний підхід, складає 15-20 інженерів. Для менших команд складність розподіленої системи може перевищити вигоди від

можливості незалежного масштабування. Тому при ухваленні рішення про застосування мікросервісної архітектури необхідно враховувати не лише технічні, але й організаційні фактори.

У контексті бакалаврської кваліфікаційної роботи мікросервісний підхід обраний з кількох міркувань. По-перше, маркетплейс є саме тим класом систем, для яких мікросервіси є природним вибором. По-друге, демонстрація реалізації повноцінної мікросервісної системи є цінною з академічної точки зору — вона дозволяє студенту опанувати ширший набір технологій та архітектурних патернів, ніж монолітний підхід. По-третє, мікросервісна архітектура є галузевим стандартом для нових проектів електронної комерції — отже, отримані навички безпосередньо корисні у подальшій професійній діяльності.

Сучасний стан розвитку мікросервісних технологій характеризується зрілістю основних компонентів та усталеною екосистемою. Платформа Kubernetes стала де-факто стандартом оркестрації, докер-образи — стандартом упакування. Розгорнута інфраструктура (хмарні провайдери, керовані Kubernetes-кластери, керовані бази даних) робить розгортання мікросервісної системи доступним навіть невеликим командам. Це створює сприятливі умови для практичної реалізації мікросервісних систем у межах академічних та комерційних проектів.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Опис предметної області

Електронний маркетплейс (англ. e-marketplace, online marketplace) — це програмне забезпечення електронної комерції, що надає інфраструктуру для проведення торговельних операцій між великою кількістю продавців та покупців у межах єдиного цифрового простору. На відміну від класичного інтернет-магазину, де всі товари належать одному власнику, маркетплейс виступає посередником: він не володіє товарами, а надає продавцям засоби для розміщення асортименту, а покупцям — інструменти пошуку, порівняння та купівлі [1].

Економічна модель маркетплейсу базується на ефекті двосторонньої мережі: чим більше продавців приєднується до платформи, тим більший асортимент бачить покупець, що збільшує приплив нових покупців; зростання аудиторії покупців, у свою чергу, приваблює нових продавців. Такий ефект створює природний бар'єр входу для конкурентів та забезпечує стійкість провідних маркетплейсів — Amazon, eBay, Rozetka, OLX, Etsy [2].

За типом товарів, що пропонуються до продажу, маркетплейси поділяються на кілька класів. Перший клас — маркетплейси фізичних товарів — обслуговує продажі речей, які потребують реальної доставки покупцю (одяг, електроніка, побутова техніка). Другий клас — маркетплейси послуг — поєднує виконавців та замовників послуг (UpWork, Fiverr, Kabanchik). Третій клас — маркетплейси цифрових продуктів — пропонує товари, що мають виключно цифрову форму: шаблони, шрифти, плагіни, музика, відео, навчальні курси (Envato Market, Gumroad, Creative Market). Четвертий клас — універсальні маркетплейси — поєднує товари кількох типів у межах однієї платформи.

Незалежно від класу маркетплейсу, його програмне забезпечення повинне реалізовувати спільний набір функціональних можливостей. До основних таких можливостей належать: каталогізація товарів із підтримкою

категорій, атрибутів та фотографій; повнотекстовий пошук та фасетна фільтрація; перегляд карток товарів з детальною інформацією, відгуками та рейтингом; кошик покупця; оформлення замовлення з обробкою платежу; кабінет покупця з історією замовлень; кабінет продавця з управлінням асортиментом та статистикою продажів; система рейтингів та відгуків як механізм формування довіри.

Технічна складність розробки маркетплейсу зумовлена кількома факторами. По-перше, маркетплейс — це система з високим змінним навантаженням: пікові періоди (розпродажі, святкові сезони) можуть у десятки разів перевищувати середнє щоденне навантаження. Це вимагає здатності системи до швидкого горизонтального масштабування. По-друге, маркетплейс зберігає різноманітні типи даних: транзакційні дані замовлень (потребують ACID-гарантій), товарний каталог (потребує гнучкої схеми та повнотекстового пошуку), сесійні дані користувачів (потребують швидкого доступу). Жодна окрема СУБД не є оптимальною для всіх типів навантаження. По-третє, маркетплейс містить багато слабко пов'язаних бізнес-доменів: каталог, кошик, замовлення, доставка, оплата, відгуки. Зміна правил у одному домені не повинна вимагати перезавантаження всієї системи [3].

Зазначені фактори визначають доцільність застосування мікросервісної архітектури — підходу, за якого програмне забезпечення розкладається на сукупність невеликих автономних сервісів, кожен з яких реалізує окрему бізнес-можливість, має власне сховище даних та може розгортатися й масштабуватися незалежно [4].

Мікросервісна архітектура протиставляється монолітній. У монолітному застосунку весь код розгортається як єдиний бінарний модуль, всі бізнес-процеси використовують спільну базу даних, а зв'язки між модулями реалізуються через виклики функцій всередині одного процесу. Монолітна архітектура простіша у початковій розробці та локальному тестуванні, але стає вузьким місцем у міру зростання системи: будь-яка зміна потребує повторного

розгортання всього застосунку; різні модулі неможливо масштабувати окремо; команди розробників постійно конкурують за спільну кодову базу.

Мікросервісний підхід усуває ці обмеження ціною підвищення архітектурної складності. Замість простих викликів функцій сервіси взаємодіють через мережеві протоколи (HTTP/REST, gRPC) або асинхронні повідомлення (RabbitMQ, Kafka). Замість єдиної бази даних кожен сервіс має власну, що ускладнює реалізацію операцій, які охоплюють кілька доменів. Замість одного процесу для розгортання потрібна оркестрація десятків контейнерів. Однак виграш у гнучкості, масштабованості та можливості незалежної розробки робить мікросервісну архітектуру стандартом для систем рівня маркетплейсу [5].

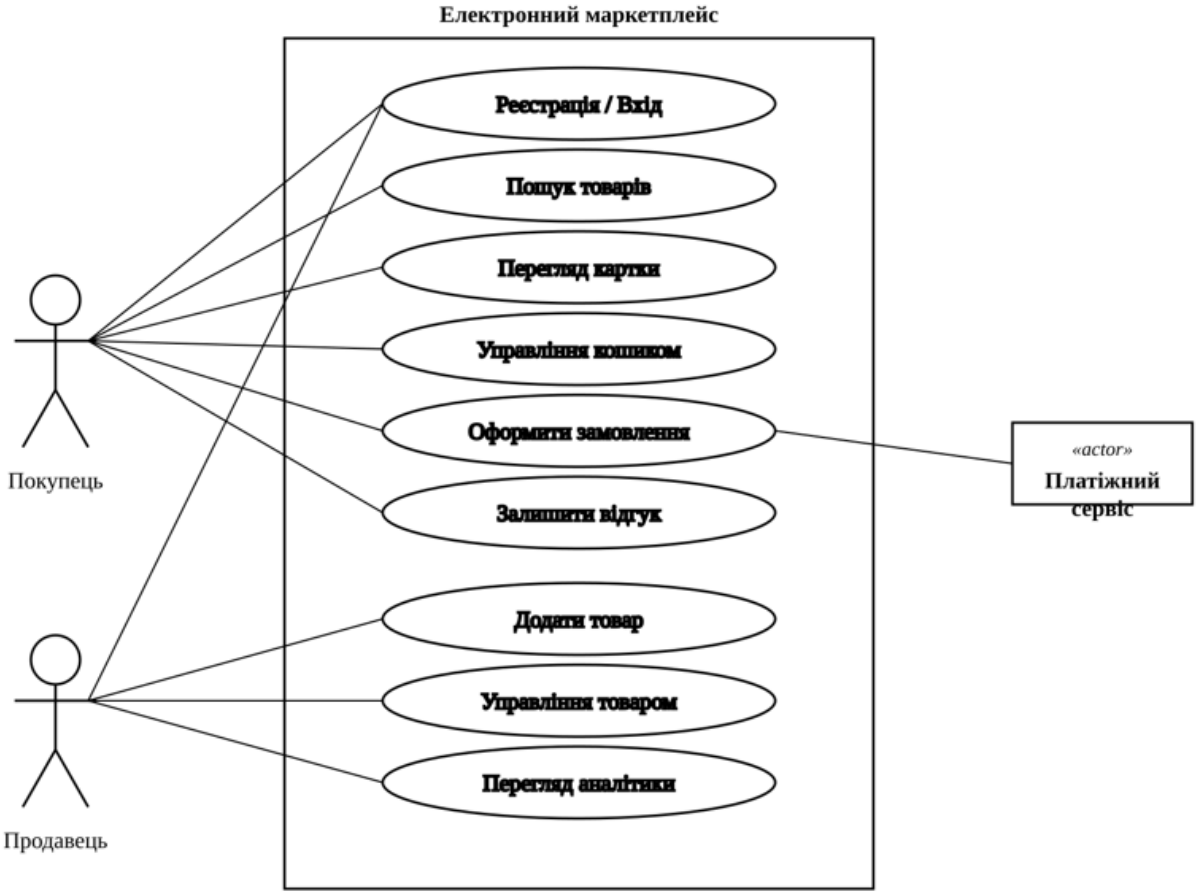
Окрему складову предметної області становить поняття бізнес-процесу маркетплейсу. Типовий процес купівлі на маркетплейсі охоплює такі етапи: продавець завантажує товар з описом, ціною та фотографіями; покупець знаходить товар через пошук або категорійну навігацію; покупець додає товар до кошика; покупець оформлює замовлення з вибором способу доставки та оплати; платіж обробляється через зовнішній платіжний шлюз; замовлення передається продавцю для обробки; товар доставляється покупцю (для фізичних товарів) або видається миттєво (для цифрових); покупець може залишити відгук та оцінку. Кожен з цих етапів потенційно може бути реалізований як окремий мікросервіс.

Таким чином, предметна область бакалаврської кваліфікаційної роботи охоплює одночасно дві паралельні предметні площини: бізнес-площину електронної комерції з її процесами та учасниками, та технічну площину мікросервісної архітектури з її компонентами та зв'язками. Поєднання цих площин формує комплексне інженерне завдання, розв'язання якого і є предметом подальших розділів роботи.

1.2. Моделювання предметної області

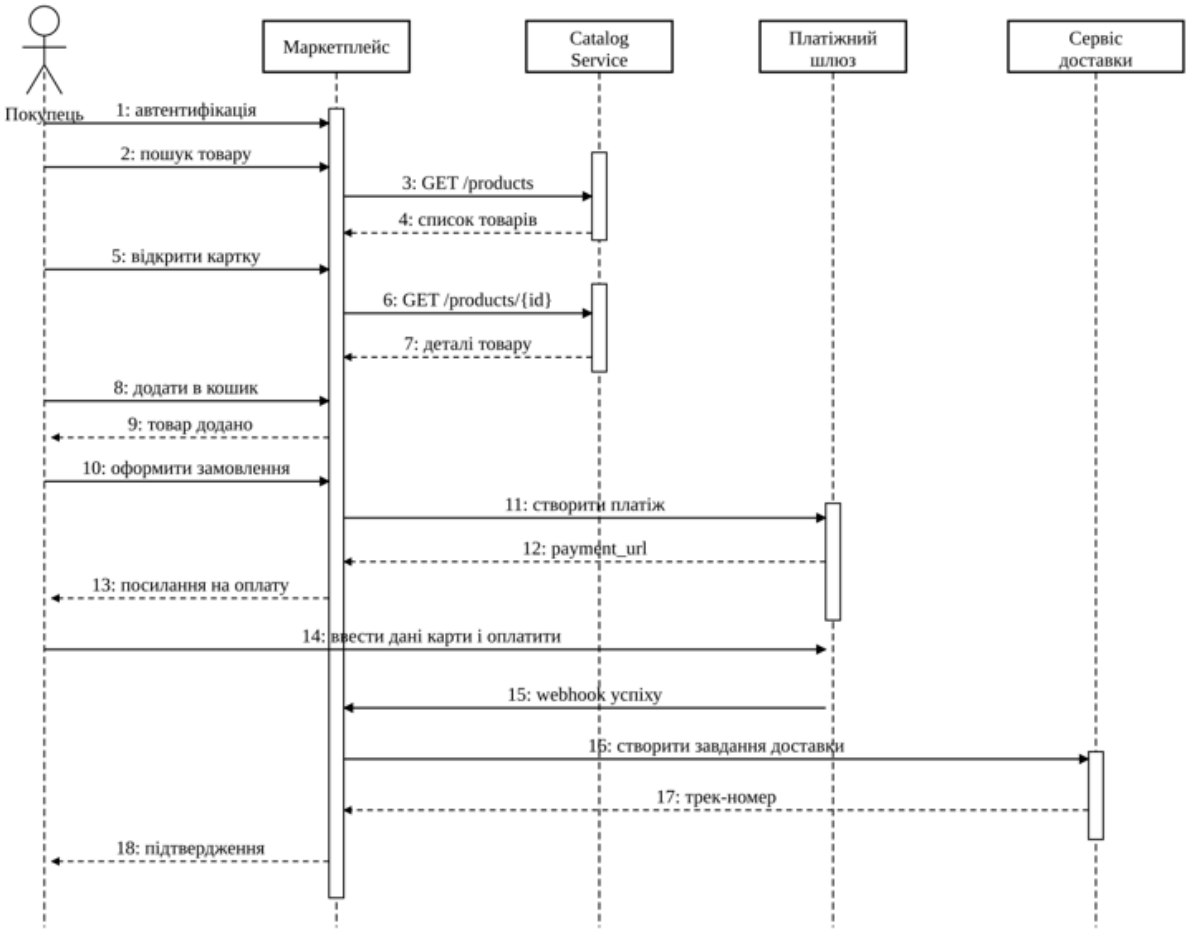
Для формалізованого опису предметної області системи розроблено комплекс UML-діаграм, що відображає функціональні вимоги, поведінку системи та взаємодію між її складовими [6].

Діаграма прецедентів (рис. 1.1) визначає дійових осіб системи та перелік функцій, доступних кожному з них. В системі виділено трьох основних акторів: покупець, продавець та платіжний сервіс як зовнішній актор. Покупець взаємодіє із системою через такі прецеденти: реєстрація та автентифікація, пошук товарів, перегляд карток, додавання до кошика, оформлення замовлення, перегляд історії покупок, залишення відгуку. Продавець виконує: реєстрацію продавця, додавання та редагування товарів, перегляд статистики продажів, відповіді на відгуки. Платіжний сервіс виконує авторизацію та списання коштів за запитом системи.



Діаграма послідовності (рис. 1.2) ілюструє хронологічний порядок взаємодії між актором «Покупець», системою та зовнішніми акторами

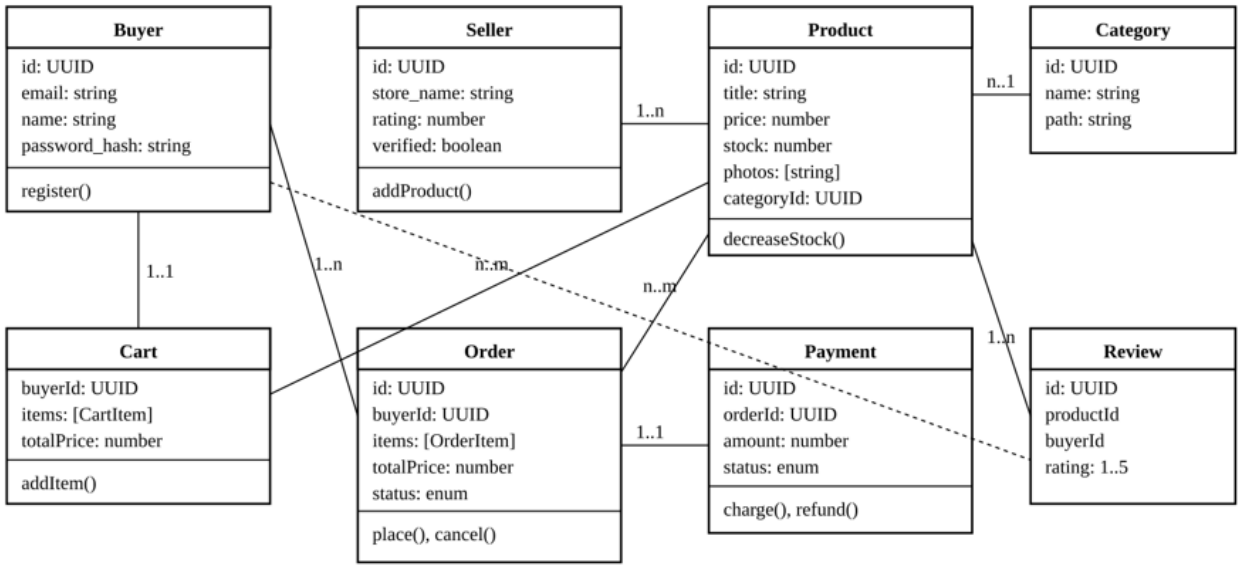
«Платіжний сервіс» і «Сервіс доставки». Послідовність охоплює повний цикл купівлі: вхід до системи, пошук товару, додавання до кошика, оформлення замовлення, обробка платежу, повідомлення про прийняте замовлення, передача замовлення на доставку, отримання покупки. Діаграма наочно показує, як система ініціює послідовність взаємодій із зовнішніми сервісами та повертає результат покупцю.



Діаграма активності (рис. 1.3) описує алгоритм основного процесу — оформлення замовлення на маркетплейсі. Процес починається з пошуку товару покупцем; за відсутності результатів пошуковий запит уточнюється або відбувається перехід до категорійної навігації. Знайдений товар додається до кошика, після чого покупець ініціює оформлення. Система перевіряє наявність товару на складі: якщо товар закінчився, покупцю пропонується обрати альтернативу. Далі вводяться дані доставки, обирається спосіб оплати, виконується платіжна транзакція. Завершальним кроком є розгалуження: для

фізичних товарів формується завдання на доставку, для цифрових — миттєва видача покупки в особистий кабінет покупця.

Абстракції предметної області, виявлені в процесі аналізу, відображено на рис. 1.4. Основними сутностями системи є: Buyer (покупець із обліковими даними та історією замовлень); Seller (продавець із профілем, рейтингом та переліком товарів); Product (товар з атрибутами, ціною та залишком на складі); Category (категорія товарного каталогу); Cart (кошик покупця з обраними товарами); Order (замовлення з переліком позицій та статусом); Payment (платіжна транзакція з посиланням на зовнішній сервіс); Review (відгук покупця про товар або продавця).



Кожна з виявлених абстракцій у подальших розділах роботи буде реалізована у вигляді набору таблиць або документів у відповідному мікросервісі. Зокрема, сутності Product та Category будуть закріплені за сервісом каталогу, Cart — за сервісом кошика, Order та Payment — за сервісом замовлень, Buyer і Seller — за сервісом обліку користувачів. Такий розподіл забезпечить кожному мікросервісу автономну зону відповідальності у відповідності до принципів Domain-Driven Design [7].

Слід також відмітити, що діаграма послідовності в реальній мікросервісній архітектурі буде дещо складнішою порівняно з її контекстною версією на рис. 1.2. Кожна стрілка «Покупець → Система» насправді

розгортається у ланцюжок: «Покупець → API Gateway → відповідний мікросервіс → база даних мікросервісу → відповідь». Однак на етапі моделювання предметної області ці технічні деталі свідомо абстраговано, щоб діаграма залишалася читабельною та відповідала рівню бізнес-логіки. Деталізована технічна архітектура буде представлена у третьому розділі роботи.

1.3. Огляд існуючих рішень у сфері електронних маркетплейсів

Ринок маркетплейсів представлений як готовими платформами, призначеними для розгортання маркетплейсу «з коробки», так і провідними операційними маркетплейсами, що є галузевими стандартами. Для визначення вимог до розроблюваної системи проведено огляд представників обох груп.

Amazon Marketplace — найбільший у світі універсальний маркетплейс. Розпочинав свою діяльність як онлайн-книгарня, згодом розширившись до повного асортименту товарів та послуг. Технічна архітектура Amazon є однією з найбільш відомих реалізацій мікросервісного підходу: за різними оцінками, платформа складається з тисяч мікросервісів, що взаємодіють між собою. Amazon першим у галузі впровадив принцип «дві піци» — команда, що розробляє один мікросервіс, повинна бути такою, щоб її можна було нагодувати двома піцами (не більше 8-10 осіб). Серед переваг — гігантський асортимент, відпрацьована логістика, висока довіра покупців. Серед недоліків з точки зору джерела вивчення — закритість архітектури, неможливість запозичення коду чи компонентів [8].

eBay — піонер моделі онлайн-аукціонів та одна з перших платформ C2C-торгівлі (consumer-to-consumer). На відміну від Amazon, де домінує модель фіксованої ціни, eBay підтримує одночасно аукціон та купівлю-зараз. Платформа має розгалужену систему репутації, що ґрунтується на взаємних відгуках продавця та покупця. eBay також пройшла еволюційний шлях від моноліту до мікросервісів, при цьому процес міграції тривав понад десятиліття [9].

Rozetka — провідний український маркетплейс. Розпочинав як інтернет-магазин електроніки, з 2015 року перетворився на маркетплейс зі сторонніми продавцями. Платформа підтримує українські способи оплати (LiqPay, Portmone, ApplePay), доставку через Нову Пошту та власну мережу пунктів видачі. Архітектурно Rozetka складається з мікросервісів, інтегрованих з ERP-системою продавців. Серед переваг — глибока локалізація, серед недоліків — закрита проприєтарна архітектура [10].

OLX — найбільша в Україні платформа оголошень типу C2C з широкою категорійною різноманітністю (товари, послуги, нерухомість, авто, робота). Технічно OLX відрізняється від класичних маркетплейсів тим, що не бере на себе функцій оплати та доставки — лише знайомить продавця з покупцем. Тим не менш, OLX є важливим джерелом ідей щодо побудови масштабованої системи пошуку та фільтрації [11].

Etsy — спеціалізований маркетплейс товарів ручної роботи, вінтажу та матеріалів для творчості. Цікавий передусім моделлю взаємодії з продавцями (так званими makers) та інструментами просування товарів через персоналізовану стрічку. Платформа активно використовує мікросервіси та машинне навчання для рекомендаційних систем [12].

Окрім готових маркетплейсів, варто розглянути open-source платформи для самостійного розгортання. Спочатку розглянемо Magento (Adobe Commerce) — потужну, але переважно монолітну PHP-платформу, яка історично домінувала на ринку електронної комерції, проте у проектах нового покоління поступово замінюється сучасними рішеннями. Saleor — порівняно нова open-source платформа на Python з підтримкою GraphQL API та мікросервісною орієнтацією. Medusa — модульна headless-платформа на Node.js, що швидко набирає популярності серед розробників завдяки гнучкому API та можливості легкого додавання власних мікросервісів. Open Spree (раніше Spree Commerce) — Ruby on Rails платформа, що добре підходить для невеликих та середніх маркетплейсів [13].

Порівняльний аналіз розглянутих платформ за ключовими архітектурними та функціональними критеріями наведено у табл. 1.1.

Таблиця 1.1

Порівняльний аналіз існуючих рішень маркетплейсів

Критерій	**Amazon**	**Rozetka**	**Medusa**	**Saleor**	**Magento**
Мікросервісна архітектура	Так	Так	Частково	Так	Ні
Open source	Ні	Ні	Так	Так	Частково
REST/GraphQL API	Так	Так	Так	Так	Так
Підтримка цифрових товарів	Так	Ні	Так	Так	Так
Локалізація для України	Ні	Так	Частково	Частково	Так
Self-hosted розгортання	Ні	Ні	Так	Так	Так
Підходить для навчання	Ні	Ні	Так	Так	Частково

Проведений аналіз (табл. 1.1) свідчить про те, що готові маркетплейси на кшталт Amazon та Rozetka є закритими комерційними системами, не доступними для дослідження їх архітектури. Open-source платформи Medusa, Saleor, Magento надають доступ до коду, проте кожна з них має певні обмеження: Medusa та Saleor є відносно новими і поки що не повністю покривають весь функціонал маркетплейсу з кількома продавцями; Magento тяжіє до монолітної архітектури і важка для модифікації. Виявлені особливості існуючих рішень визначають доцільність розробки власної системи з мікросервісною архітектурою, повністю орієнтованої на навчальні та дослідницькі цілі. Конкретні вимоги до системи формулюються у наступному підрозділі.

1.4. Постановка задачі

На підставі проведеного аналізу предметної області та виявлених особливостей існуючих рішень сформульовано таку постановку задачі.

Необхідно розробити програмне забезпечення електронного маркетплейсу з мікросервісною архітектурою, що демонструє повний цикл операцій продажу від каталогізації товарів до видачі покупки. Система повинна забезпечувати:

- автентифікацію та авторизацію користувачів двох ролей — покупець та продавець — на основі JWT-токенів;
- каталогізацію товарів із підтримкою категорій, атрибутів, фотографій та залишків на складі;
- повнотекстовий пошук товарів з підтримкою фасетної фільтрації за категорією, ціною, рейтингом, продавцем;
- сортування товарів за популярністю, ціною, рейтингом, датою додавання;
- перегляд карток товарів з детальною інформацією, фотографіями, описом, технічними характеристиками та відгуками;
- кошик покупця з підтримкою декількох товарів від різних продавців;
- оформлення замовлення з введенням даних доставки, вибором способу оплати та підтвердженням;
- обробку платежу через зовнішній платіжний шлюз (демонстраційний режим);
- особистий кабінет покупця з історією замовлень та збереженою бібліотекою цифрових продуктів;
- кабінет продавця з можливістю додавання товарів, перегляду статистики продажів та доходу;
- систему рейтингів та відгуків як механізм формування довіри між учасниками;
- декомпозицію системи на 8 незалежних мікросервісів з чітко окресленою зоною відповідальності;

- взаємодію між мікросервісами через REST API та брокер повідомлень для асинхронних подій;
- використання поліглот-персистентності: PostgreSQL для транзакційних даних, MongoDB для каталогу, Redis для кешу та сесій;
- контейнеризацію кожного мікросервісу через Docker та підготовку до розгортання у Kubernetes.

Система реалізується у вигляді розподіленого програмного забезпечення з трьома основними логічними рівнями: рівень клієнта (Single Page Application на React, що виконується у браузері користувача); рівень API Gateway (єдина точка входу для всіх HTTP-запитів від клієнта); рівень мікросервісів (вісім незалежних серверних застосунків на Node.js, кожен з власною базою даних). Взаємодія між клієнтом та API Gateway відбувається через REST API із JWT-автентифікацією. Взаємодія між мікросервісами реалізується двома способами: синхронні запити через REST для операцій, де потрібна негайна відповідь (наприклад, перевірка наявності товару при додаванні в кошик), та асинхронні події через брокер повідомлень RabbitMQ для слабко зв'язаних дій (наприклад, повідомлення сервісу аналітики про здійснене замовлення) [14].

Окрему важливу вимогу становить можливість незалежного розгортання та масштабування кожного мікросервісу. Для цього кожен сервіс упаковується в окремий Docker-контейнер з конфігурацією, що задається через змінні середовища. Кластер контейнерів керується платформою Kubernetes, яка автоматично балансує навантаження, перезапускає несправні екземпляри та масштабує кількість реплік сервісу залежно від поточної інтенсивності запитів.

Розглянуті функціональні та архітектурні вимоги визначають детальну постановку задачі, реалізація якої описана у подальших розділах роботи.

Особливості мікросервісної архітектури для електронної комерції

Загальні принципи мікросервісної архітектури, описані у класичних роботах М. Фаулера та С. Ньюмена, на практиці потребують адаптації до

конкретної предметної області. Для електронної комерції характерні специфічні патерни, що формують відмітні особливості мікросервісної системи маркетплейсу.

Перша особливість — нерівномірність навантаження на різні домени. Аналіз реальних маркетплейсів показує, що 70-80% всіх запитів припадають на читання каталогу та пошук, лише 15-20% — на роботу з кошиком, і лише 1-5% — на оформлення замовлень. Така диспропорція означає, що Catalog Service та Search Service потребують значно більше обчислювальних ресурсів, ніж Order Service. У монолітній архітектурі це призвело б до неефективного використання ресурсів — масштабування цілого моноліту для збільшення пропускної здатності лише пошуку. Мікросервіси усувають цю проблему.

Друга особливість — різні вимоги до узгодженості даних. Каталог товарів допускає певну затримку: якщо ціна змінилась, цілком прийнятно, щоб ця зміна відобразилася в пошуковому індексі через кілька секунд. Кошик допускає аналогічну затримку — навіть втрата вмісту кошика, хоча й неприємна, не призводить до фінансових втрат. Однак оформлення замовлення вимагає строгої узгодженості: списання коштів та зменшення залишку товару мають відбутися атомарно. Така диференціація вимог дозволяє у різних доменах використовувати різні моделі узгодженості, що і є однією з ключових переваг мікросервісного підходу.

Третя особливість — багаторівнева структура аудиторії. Маркетплейс одночасно обслуговує покупців (масова аудиторія, переважно операції читання) та продавців (професійна аудиторія, операції запису та аналітики). Для покупців критично важливі швидкість завантаження сторінок та якість пошуку, для продавців — функціональність кабінету та точність аналітики. У монолітній архітектурі ці контрастні потреби складно поєднати в єдиному коді; у мікросервісній — можливо створити окремі сервіси для покупця та продавця, оптимізовані під відповідні навантаження.

Четверта особливість — складна інтеграційна екосистема. Маркетплейс інтегрується з десятками зовнішніх систем: платіжні шлюзи, служби доставки,

ERP-системи продавців, сервіси email- та SMS-розсилок, аналітичні платформи, рекламні мережі. Кожна інтеграція має власні особливості, протоколи, обмеження частоти запитів. Винесення кожної інтеграції в окремий мікросервіс (або principle of antecorruption layer) дозволяє ізолювати специфіку зовнішніх систем від основної бізнес-логіки.

П'ята особливість — необхідність експериментування з функціональністю. Маркетплейси постійно тестують нові ідеї: змінюють алгоритми ранжування, експериментують з форматами карток товарів, додають нові способи фільтрації. Мікросервісна архітектура дозволяє вносити такі зміни без впливу на інші частини системи — наприклад, оновлення Search Service з новим алгоритмом ранжування жодним чином не зачіпає Order Service. Це суттєво прискорює темп інновацій.

Економічна модель маркетплейсу

Окрему увагу при аналізі предметної області слід приділити економічній моделі маркетплейсу, оскільки саме вона визначає функціональні вимоги до програмного забезпечення. Більшість сучасних маркетплейсів отримують дохід за рахунок комісії з продажів — фіксованого відсотка від суми кожної транзакції, який утримується платформою. Розмір комісії варіюється від 5% (для категорій з низькою маржею, як-от електроніка) до 30% (для цифрових продуктів та контенту). Наявність комісії накладає на програмне забезпечення додаткові вимоги: точний облік фінансових потоків, прозорість для продавців, надійні механізми взаєморозрахунків.

Окрім комісії з продажів, маркетплейси використовують додаткові джерела доходу: плата за просування (продавці платять за виведення товарів у топ-позиції видачі); підписки для продавців (преміум-функції в кабінеті); реклама всередині платформи; платні послуги доставки. Ці джерела доходу формують додаткові функціональні підсистеми, які мають бути враховані в архітектурі.

Аналіз поведінки покупців на маркетплейсах виявляє кілька ключових патернів. По-перше, конверсія сильно залежить від швидкості завантаження сторінок: затримка на 1 секунду знижує конверсію приблизно на 7%. Це робить вимоги до продуктивності системи критичними. По-друге, понад 60% покупців приймають рішення про купівлю на основі відгуків інших покупців — отже, система відгуків та рейтингів має бути продуманою та захищеною від маніпуляцій. По-третє, понад 70% сесій починаються з пошуку — це визначає пріоритетне місце пошуку у функціональних вимогах [3].

Слід також врахувати, що поведінка покупців і продавців на маркетплейсі суттєво відрізняється. Покупці генерують переважно операції читання: переглядають каталог, читають описи, порівнюють товари. Лише невеликий відсоток сесій завершується покупкою. Продавці, навпаки, генерують операції запису: завантажують нові товари, оновлюють залишки, відповідають на повідомлення. Таке співвідношення читань і записів (приблизно 95:5) дозволяє оптимізувати архітектуру через кешування результатів читання та горизонтальне масштабування read-replica баз даних.

Архітектурні виклики маркетплейсу

Розробка маркетплейсу породжує низку специфічних архітектурних викликів, що відрізняють її від класичних інтернет-магазинів. Перший виклик — підтримка пошуку у великому каталозі. Класичний інтернет-магазин з 10-20 тисячами товарів цілком задовольняється SQL-запитами з LIKE-фільтрами. Маркетплейс з сотнями тисяч і мільйонами позицій вимагає спеціалізованої пошукової інфраструктури, що підтримує повнотекстовий пошук, ранжування за релевантністю, врахування морфології, виправлення помилок.

Другий виклик — забезпечення цілісності даних при паралельних операціях. Класичний приклад — situation, коли двоє покупців одночасно намагаються купити останню одиницю товару. Без належної синхронізації обидва замовлення можуть бути прийняті, що призведе до неможливості виконати одне з них. У монолітній системі ця проблема вирішується

транзакціями СУБД, у мікросервісній архітектурі — складніше, через відсутність розподілених транзакцій. Розв'язок передбачає використання патернів Saga та компенсуючих транзакцій.

Третій виклик — управління образами та медіа-контентом. Кожен товар на маркетплейсі має від 3 до 10 фотографій, цифрові товари додатково включають файли продуктів. Усі ці файли потребують надійного сховища, оптимізації розмірів, генерації мініатюр, CDN-доставки. Це формує окрему підсистему — File Storage Service.

Четвертий виклик — антифрод (захист від шахрайства). Маркетплейс — приваблива ціль для шахраїв: продаж неіснуючих товарів, отримання коштів без виконання замовлення, накрутка відгуків, відмивання грошей через фіктивні покупки. Сучасні маркетплейси інвестують значні ресурси у системи детектування підозрілих транзакцій, що використовують машинне навчання та аналіз поведінкових патернів.

Кожен з перелічених викликів формує окремий контекст для архітектурних рішень, що ухвалюються при проектуванні системи. У межах цієї роботи основна увага зосереджена на функціональному ядрі маркетплейсу — каталог, пошук, кошик, замовлення; антифрод-система та повноцінний модуль медіа-обробки винесено в напрями подальшого розвитку.

Висновки до розділу 1

У першому розділі проведено аналіз предметної області електронних маркетплейсів. Розглянуто сутність та класифікацію маркетплейсів, виділено чотири основні класи (фізичні товари, послуги, цифрові продукти, універсальні), визначено спільний набір функціональних можливостей. Обґрунтовано доцільність застосування мікросервісної архітектури для систем такого масштабу та виявлено технічні фактори, що зумовлюють цю доцільність — змінне навантаження, різнорідність типів даних, слабка зв'язаність бізнес-доменів.

Побудовано комплекс UML-діаграм предметної області: діаграму прецедентів із трьома акторами (покупець, продавець, платіжний сервіс), діаграму послідовності оформлення замовлення, діаграму активності бізнес-процесу купівлі та діаграму ключових абстракцій предметної області, що містить 8 сутностей.

Розглянуто п'ять представників ринку — Amazon, eBay, Rozetka, OLX, Etsy — та три open-source платформи Magento, Saleor, Medusa. За результатами порівняльного аналізу зроблено висновок, що існуючі готові рішення є закритими комерційними системами, а open-source платформи мають обмеження для навчальних цілей. Це визначає доцільність розробки власної системи з мікросервісною архітектурою.

Сформульовано детальну постановку задачі, що охоплює 15 функціональних вимог до системи: від автентифікації до контейнеризації мікросервісів.

РОЗДІЛ 2 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Логічна модель даних

Проектування інформаційного забезпечення мікросервісної системи має принципову відмінність від проектування інформаційного забезпечення монолітного застосунку. У монолітній архітектурі логічна модель даних будується як єдина схема, що охоплює всі сутності системи з усіма зв'язками між ними. У мікросервісній архітектурі діє принцип «база даних на сервіс» (database per service) [15], згідно з яким кожен мікросервіс має власне ізольоване сховище, недоступне для прямого читання чи запису з боку інших сервісів. Доступ до даних мікросервісу можливий виключно через його API. Це означає, що логічна модель даних розпадається на кілька локальних моделей — по одній на кожен мікросервіс — з мінімальними та явно визначеними зв'язками між ними [16].

Аналіз функціональних вимог, сформульованих у розділі 1, дозволив виділити вісім автономних доменів даних, що відповідають восьми мікросервісам системи.

Домен User Service охоплює дані всіх користувачів системи: покупців та продавців. Основні сутності — User та SellerProfile. Сутність User містить ідентифікатор, електронну адресу (унікальну), хешований пароль, ім'я, роль (buyer або seller), часові мітки створення та останнього входу. Сутність SellerProfile є розширенням User для користувачів з роллю seller: вона додає назву магазину, опис, рейтинг та кількість відгуків. Зв'язок між User та SellerProfile — один-до-одного, реалізується через зовнішній ключ.

Домен Catalog Service охоплює дані товарного асортименту. Основні сутності — Product, Category, ProductAttribute. Сутність Product містить ідентифікатор, назву, опис, ціну, посилання на категорію, посилання на продавця, залишок на складі, рейтинг, кількість продажів, масив URL фотографій. Сутність Category утворює ієрархічне дерево категорій з можливими підкатегоріями. Сутність ProductAttribute зберігає характеристики

товару у вигляді пар «ключ-значення», що дозволяє підтримувати різнорідні атрибути для різних категорій (для одягу — розмір та колір, для електроніки — модель та тактова частота).

Домен Search Service не має власних персистентних сутностей — він використовує індекс на основі Elasticsearch, що періодично синхронізується з Catalog Service через події. Індекс містить ті ж поля, що й Product, але оптимізовані для повнотекстового пошуку та фасетної фільтрації.

Домен Cart Service охоплює сутність Cart — кошик покупця, що містить ідентифікатор покупця та масив CartItem (ідентифікатор товару, кількість, ціна на момент додавання). Кошик зберігається у Redis як хеш-структура, що забезпечує швидкий доступ при високому навантаженні.

Домен Order Service охоплює сутності Order, OrderItem, Payment. Сутність Order містить ідентифікатор, посилання на покупця, дату створення, загальну суму, статус (created, paid, shipping, delivered, cancelled), дані доставки. Сутність OrderItem зберігає окрему позицію замовлення: товар, кількість, ціна. Сутність Payment описує платіжну транзакцію: ідентифікатор, посилання на замовлення, сума, статус, ідентифікатор у зовнішньому платіжному сервісі.

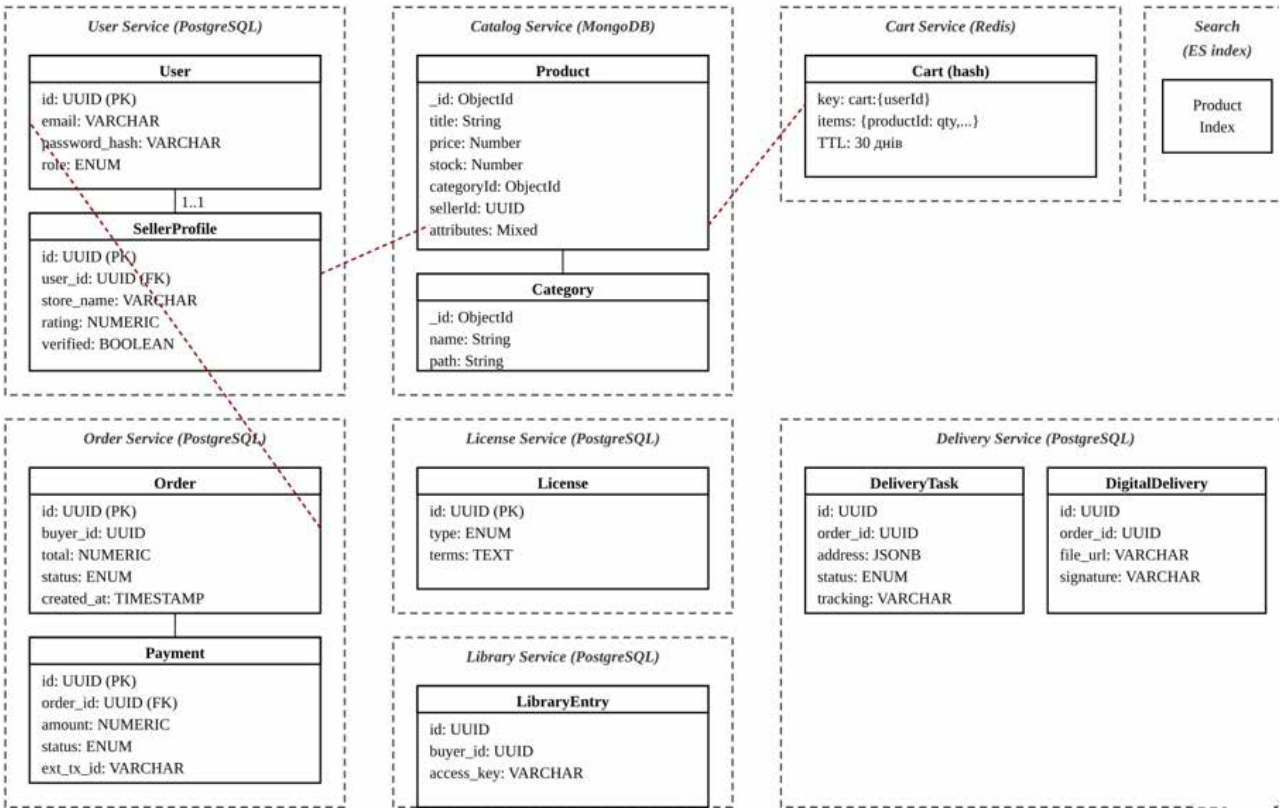
Домен License Service охоплює сутність License (тип ліцензії для цифрового товару) та LicenseGrant (видана покупцю ліцензія з унікальним ключем доступу). Цей сервіс активний лише для маркетплейсу цифрових продуктів.

Домен Library Service охоплює сутність LibraryEntry — запис про куплений цифровий продукт у бібліотеці покупця. Містить ідентифікатор покупця, ідентифікатор товару, тип ліцензії, ключ доступу, дату придбання.

Домен Delivery Service охоплює сутність DeliveryTask (завдання на доставку фізичного товару) та DigitalDeliveryRecord (запис про миттєву видачу цифрового продукту). DeliveryTask містить адресу, спосіб доставки, статус, номер відправлення; DigitalDeliveryRecord містить посилання на файл та підпис, що підтверджує право доступу.

Зв'язки між сутностями різних мікросервісів реалізуються через ідентифікатори, а не через зовнішні ключі реляційної СУБД. Наприклад, замовлення в Order Service посилається на покупця через поле userId (UUID), яке відповідає ідентифікатору User у User Service. Цілісність такого посилання забезпечується на рівні бізнес-логіки, а не СУБД: перед створенням замовлення Order Service запитує User Service про існування користувача через REST API. Такий підхід називається eventual consistency (підсумкова узгодженість) і є компромісом мікросервісної архітектури між строгою цілісністю даних та можливістю незалежного розгортання сервісів [17].

Узагальнену логічну модель даних усієї системи з виділенням меж мікросервісів наведено на рис. 2.1.



На рис. 2.1 пунктирними рамками виділено зони відповідальності кожного мікросервісу. Стрілки між зонами позначають референційні зв'язки через ідентифікатори, що забезпечуються бізнес-логікою, а не СУБД. Суцільні лінії всередині зон позначають зв'язки між сутностями одного мікросервісу, які можуть бути реалізовані стандартними засобами обраної СУБД.

2.2. Вибір системи управління базами даних

Вибір СУБД є одним з ключових архітектурних рішень мікросервісної системи. На відміну від монолітного застосунку, де всі дані зберігаються в єдиній СУБД, мікросервісна архітектура допускає та заохочує використання різних СУБД для різних сервісів. Такий підхід отримав назву поліглот-персистентності (polyglot persistence) і базується на принципі: для кожного типу даних обирається СУБД, оптимальна саме для цих даних [18].

При проектуванні системи розглянуто кілька класів СУБД.

Реляційні СУБД (PostgreSQL, MySQL) забезпечують строгу узгодженість даних, підтримку транзакцій з ACID-гарантіями, потужну мову запитів SQL та зрілий екосистемний інструментарій. Їх головні переваги — це надійність та передбачуваність поведінки. Обмеженням є необхідність попереднього проектування жорсткої схеми, що ускладнює роботу з даними змінної структури.

Документоорієнтовані СУБД (MongoDB, CouchDB) зберігають дані у вигляді документів — структур з гнучкою схемою, як правило, у форматі JSON або BSON. Їх перевагами є гнучкість при роботі з даними змінної структури, природна підтримка вкладених об'єктів, можливість горизонтального масштабування. Обмеженням є слабкі гарантії узгодженості при роботі з пов'язаними документами та відсутність стандартизованої мови запитів.

Ключ-значення сховища (Redis, Memcached) є найпростішим типом СУБД, що зберігає пари «ключ → значення» в оперативній пам'яті. Вони забезпечують найшвидший доступ до даних — операції читання та запису виконуються за мікросекунди. Обмеження — відсутність складних запитів та обмежений обсяг даних, що поміщається в RAM.

Пошукові індекси (Elasticsearch, Apache Solr) спеціалізуються на повнотекстовому пошуку та аналітичних запитах. Вони підтримують ранжування результатів за релевантністю, фасетну фільтрацію, агрегації. Обмеженням є те, що вони не призначені як основне сховище даних — лише як індекс над основним сховищем.

Порівняльний аналіз основних класів СУБД за ключовими критеріями для маркетплейсу наведено у табл. 2.1.

Таблиця 2.1

Порівняльний аналіз класів СУБД

Критерій	**PostgreSQL**	**MongoDB**	**Redis**	**Elasticsearch**
Тип сховища	Реляційне	Документне	Ключ-значення	Пошуковий індекс
ACID-транзакції	Повна підтримка	Обмежена	Ні	Ні
Швидкість читання	Висока	Висока	Дуже висока	Висока
Гнучкість схеми	Низька	Висока	Висока	Середня
Повнотекстовий пошук	Базовий	Базовий	Ні	Передовий
Горизонтальне масштабування	Складне	Природне	Природне	Природне
Підходить для маркетплейсу	Транзакції	Каталог	Кошик, кеш	Пошук

На підставі результатів порівняльного аналізу для системи маркетплейсу обрано комбіноване рішення — поліглот-персистентність, що включає чотири СУБД, кожна з яких використовується для тих типів даних, для яких вона є оптимальною.

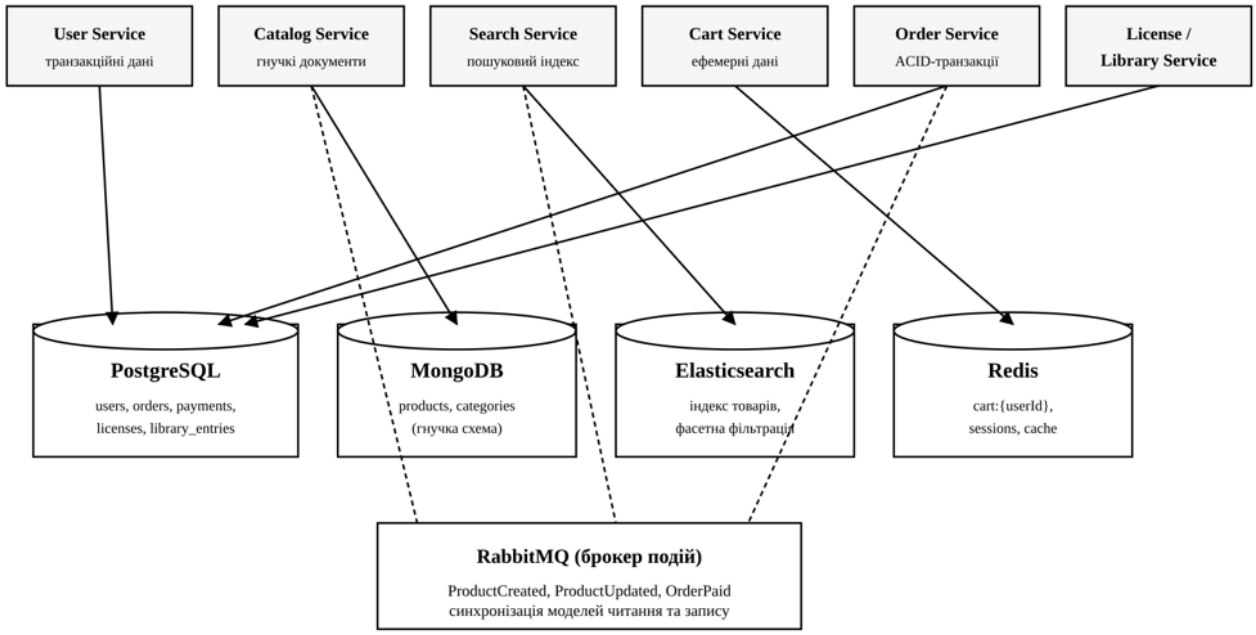
PostgreSQL обрано як основне сховище для User Service, Order Service, License Service. Це сервіси, що працюють з транзакційними даними (користувачі, замовлення, ліцензії), для яких критично важлива строга узгодженість та підтримка ACID-транзакцій. PostgreSQL — найпотужніша open-source реляційна СУБД, що має широку підтримку у Node.js-екосистемі через бібліотеку pg, підтримує розширені типи даних (JSON, JSONB, масиви), повнотекстовий пошук, географічні координати. PostgreSQL також природно підтримується платформою Kubernetes через офіційний оператор.

MongoDB обрано як основне сховище для Catalog Service. Каталог товарів — це домен з даними змінної структури: різні категорії товарів мають різні атрибути (для одягу — розмір та колір, для книг — автор та видавництво, для електроніки — модель та технічні характеристики). Жорстка реляційна схема призводила б до численних опційних колонок або складних JOIN-операцій з таблицею атрибутів. MongoDB природно зберігає всі дані товару — основні поля та атрибути — як єдиний документ, що значно спрощує запити та підвищує продуктивність читання.

Redis обрано для Cart Service та для кешування у всіх інших сервісах. Кошик — це короткоживучі дані, до яких звертаються при кожній сторінці перегляду каталогу та при оформленні замовлення. Зберігання кошика в реляційній СУБД створювало б непропорційне навантаження. Redis дозволяє зберігати кошик як хеш-структуру з автоматичним терміном життя (TTL), що автоматично очищає покинуті кошики.

Elasticsearch обрано для Search Service. Повнотекстовий пошук з ранжуванням за релевантністю, фасетна фільтрація за кількома атрибутами одночасно, автозаповнення пошукових запитів — це функції, для яких Elasticsearch є галузевим стандартом. Індекс Elasticsearch синхронізується з MongoDB-каталогом через подієву інтеграцію: при додаванні або зміні товару в Catalog Service публікується подія в RabbitMQ, на яку підписаний Search Service.

Узагальнену схему розподілу даних між сховищами наведено на рис. 2.2.



2.3. Фізична структура бази даних

На основі логічної моделі даних розроблено фізичну структуру баз даних для кожного мікросервісу. У цьому підрозділі детально описано фізичну структуру трьох ключових мікросервісів — User Service (PostgreSQL), Catalog Service (MongoDB) та Cart Service (Redis), які є представниками трьох різних класів СУБД.

База даних User Service реалізована засобами PostgreSQL і складається з двох основних таблиць: users та seller_profiles. Структуру таблиці users наведено у табл. 2.2.

Таблиця 2.2

Структура таблиці users

Поле	**Тип**	**Обмеження**	**Опис**
id	UUID	PK, DEFAULT gen_random_uuid()	Унікальний ідентифікатор
email	VARCHAR(255)	NOT NULL, UNIQUE	Електронна адреса користувача
password_hash	VARCHAR(255)	NOT NULL	Хеш пароля (bcrypt)

name	VARCHAR(100)	NOT NULL	Ім'я користувача
role	VARCHAR(20)	NOT NULL, CHECK IN (buyer, seller)	Роль користувача
created_at	TIMESTAMPTZ	NOT NULL DEFAULT NOW()	Дата реєстрації
last_login_at	TIMESTAMPTZ	NULL	Дата останнього входу

На поле email створено унікальний індекс idx_users_email для прискорення пошуку при вході. Хеш пароля зберігається у форматі bcrypt з cost factor 10, що забезпечує захист від атак повним перебором [19].

Структуру таблиці seller_profiles наведено у табл. 2.3. Таблиця пов'язана з users через зовнішній ключ user_id з обмеженням ON DELETE CASCADE: видалення користувача автоматично видаляє його профіль продавця.

Таблиця 2.3

Структура таблиці seller_profiles

Поле	**Тип**	**Обмеження**	**Опис**
id	UUID	PK	Ідентифікатор
user_id	UUID	FK users(id), ON DELETE CASCADE	Користувач-продавець
store_name	VARCHAR(100)	NOT NULL	Назва магазину
description	TEXT	NULL	Опис магазину
rating	NUMERIC(3,2)	DEFAULT 0.00	Середній рейтинг
reviews_count	INTEGER	DEFAULT 0	Кількість відгуків
verified	BOOLEAN	DEFAULT FALSE	Чи верифікований

База даних Catalog Service реалізована засобами MongoDB і складається з двох колекцій: products та categories. У MongoDB немає поняття «структура таблиці» — натомість документи мають гнучку схему. Однак для забезпечення цілісності та прискорення розробки використовується ODM-бібліотека

Mongoose, що дозволяє декларативно описати очікувану структуру документа. Узагальнену схему документа Product наведено у табл. 2.4.

Таблиця 2.4

Структура документа Product у Catalog Service

Поле	**Тип**	**Обмеження**	**Опис**
_id	ObjectId	PK, auto	Ідентифікатор товару
title	String	required, indexed	Назва товару
description	String	required	Детальний опис
price	Number	required, min: 0	Ціна в копійках
categoryId	ObjectId	ref: Category, indexed	Категорія
sellerId	UUID	required, indexed	Посилання на User Service
stock	Number	default: 0	Залишок на складі
photos	[String]	масив URL	Фотографії товару
attributes	Mixed	гнучка структура	Характеристики
rating	Number	default: 0	Середня оцінка
sales	Number	default: 0	Кількість продажів
createdAt	Date	default: Date.now	Дата додавання

Поле attributes має тип Mixed — це означає, що в нього можна записати будь-яку JSON-структуру. Саме цей факт є основною причиною вибору MongoDB для каталогу: товари різних категорій мають принципово різні набори атрибутів, і MongoDB дозволяє зберігати їх природно, без необхідності окремої таблиці характеристик.

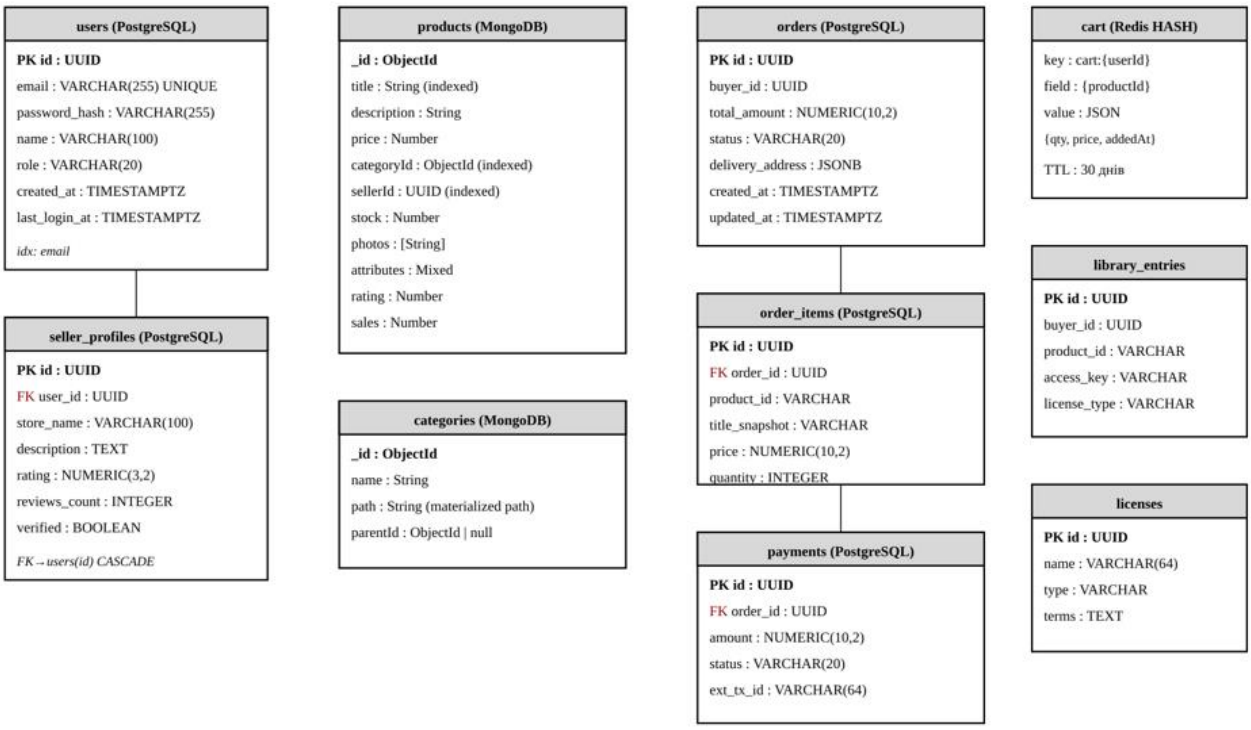
На колекції products створено індекси: одиничний на title (для пошуку), складений на categoryId+price (для каталогізованого перегляду з фільтром за ціною), одиничний на sellerId (для відображення товарів конкретного продавця). Окрім цього, створено текстовий індекс на полях title та description,

що використовується як резервний пошук на випадок недоступності Elasticsearch [20].

Дані Cart Service зберігаються у Redis у форматі hash. Ключ кошика формується як cart:{userId}, а значення — це hash, де ключі — ідентифікатори товарів, а значення — JSON-серіалізовані об'єкти позицій кошика. Структура позиції включає: id товару, кількість, ціна на момент додавання, дата додавання. Для кожного кошика встановлюється TTL (time-to-live) тривалістю 30 днів — кошики, у які не вносилися зміни протягом цього періоду, автоматично видаляються Redis для звільнення пам'яті.

Окрім описаних трьох сервісів, аналогічно спроектовано фізичні структури для Order Service (таблиці orders, order_items, payments у PostgreSQL), License Service (таблиці licenses, license_grants у PostgreSQL), Library Service (таблиця library_entries у PostgreSQL), Delivery Service (таблиці delivery_tasks, digital_delivery_records у PostgreSQL) та індекс Search Service у Elasticsearch.

Загальну схему фізичної структури баз даних з виділенням мікросервісів наведено на рис. 2.3.



Принципи нормалізації у мікросервісних схемах

Класична теорія баз даних оперує поняттями нормальних форм — структурних правил, що мінімізують надлишковість та аномалії при оновленні даних. У мікросервісній архітектурі принципи нормалізації застосовуються інакше, ніж у моноліті, через два ключові обмеження: відсутність прямих JOIN-операцій між таблицями різних сервісів та необхідність толерантності до часткових відмов.

Перший принцип — нормалізація в межах сервісу. Усередині кожного мікросервісу дані нормалізовані за класичними правилами (зазвичай до 3НФ). Наприклад, у User Service таблиці users та seller_profiles нормалізовані: спільні дані користувача (email, ім'я) зберігаються в users; специфічні для продавця дані (назва магазину, рейтинг) — у seller_profiles. Це усуває дублювання та забезпечує цілісність на рівні СУБД.

Другий принцип — контрольована денормалізація між сервісами. Дані, які потрібні в кількох сервісах одночасно, часто свідомо дублюються. Наприклад, у Order Service кожна позиція замовлення містить не лише productId, а й копії назви та ціни товару на момент покупки. Це необхідно з кількох причин: по-перше, дані замовлення повинні залишатися незмінними навіть якщо товар буде видалений; по-друге, без копіювання при відображенні замовлення потрібно було б синхронно запитувати Catalog Service для кожної позиції — це створює тісну зв'язаність та погіршує продуктивність.

Третій принцип — використання ідентифікаторів типу UUID замість sequential ID. У мікросервісній архітектурі неможливо покладатися на автоінкрементні цілочисельні ідентифікатори, бо кожен сервіс має власне сховище та незалежно генерує ідентифікатори. UUID забезпечує гарантовану унікальність без необхідності координації між сервісами та усуває можливість колізій навіть при злитті даних з різних джерел.

Четвертий принцип — використання типу JSONB для напівструктурованих даних у реляційних СУБД. Сучасні версії PostgreSQL

підтримують JSONB — бінарний формат зберігання JSON з можливістю індексації за окремими полями. Це дозволяє в одному стовпці зберігати атрибути зі складною структурою без необхідності створення окремих таблиць. У розробленій системі JSONB використовується для зберігання адрес доставки в Order Service та метаданих ліцензій у License Service.

Особливості розподіленого зберігання даних

Розподіл даних між кількома мікросервісами породжує специфічну категорію проблем — узгодженість даних у розподіленій системі. Згідно з теоремою CAP (Consistency, Availability, Partition tolerance), у розподіленій системі неможливо одночасно забезпечити повну узгодженість, доступність та стійкість до розділення мережі — необхідно обрати два з трьох [17]. Мікросервісна архітектура зазвичай обирає Availability + Partition tolerance, жертвуючи строгою узгодженістю на користь так званої підсумкової узгодженості (eventual consistency).

Підсумкова узгодженість означає, що при відсутності нових змін усі репліки даних врешті-решт прийдуть до спільного стану, але цей процес може займати певний час. Для маркетплейсу це означає, наприклад, що зміна ціни товару у Catalog Service не миттєво відобразиться у пошуковому індексі Search Service та у кошиках, які вже містять цей товар. На практиці цей лаг становить кілька секунд та є прийнятним для більшості сценаріїв.

Для випадків, коли підсумкова узгодженість неприйнятна — наприклад, при списанні коштів за замовлення — застосовуються паттерни Saga та Two-Phase Commit. Saga розкладає логічно єдину транзакцію на послідовність локальних транзакцій кожного сервісу з компенсуючими діями на випадок невдачі. Наприклад, при оформленні замовлення Order Service виконує: 1) резервування товару в Catalog Service; 2) списання коштів через Payment Service; 3) створення записів у Library Service. Якщо крок 3 не вдається, виконуються компенсуючі транзакції: повернення коштів та зняття резерву.

Іншою важливою концепцією є CQRS (Command Query Responsibility Segregation) — розділення моделей читання та запису. У маркетплейсі це проявляється наочно: операції запису (додавання товару, оформлення замовлення) виконуються над нормалізованими реляційними даними; операції читання (відображення каталогу, пошук) — над денормалізованими репліками, оптимізованими для швидкого читання. Synchronization між моделями забезпечується подіями: при зміні стану в моделі запису публікується подія, на яку реагують моделі читання.

У контексті розробленої системи паттерн CQRS реалізується через подвійне зберігання даних каталогу: первинна модель — у Catalog Service (MongoDB), оптимізована для CRUD-операцій продавця; вторинна модель — у Search Service (Elasticsearch), оптимізована для пошукових запитів покупців. Синхронізація відбувається через події ProductCreated, ProductUpdated, ProductDeleted у RabbitMQ.

Стратегія резервного копіювання

Окрему складову інформаційного забезпечення становить стратегія резервного копіювання даних. У мікросервісній архітектурі ця задача складніша порівняно з монолітом, оскільки дані розкидані по кількох сховищах різних типів, і відновлення системи потребує узгодженого стану всіх сховищ на момент часу резервної копії.

Для PostgreSQL обрано стратегію Point-in-Time Recovery (PITR). Повна резервна копія всіх баз створюється щодня о 03:00 та зберігається у хмарному сховищі. Окрім того, безперервно архівуються WAL-логи (Write-Ahead Logs), що дозволяє відновити стан бази на будь-який момент часу між повними копіями. Термін зберігання резервних копій — 30 днів.

Для MongoDB використовується mongodump з шифруванням архівів та завантаженням до хмарного сховища. Резервне копіювання виконується щодня о 03:30, копії зберігаються 30 днів.

Для Redis окреме резервне копіювання не передбачено, оскільки в системі Redis зберігає лише ефемерні дані (кошики, кеш, сесії). Втрата вмісту Redis при відмові не призводить до критичних наслідків: користувачі побачать порожні кошики, що неприємно, але не порушує цілісність даних.

Для Elasticsearch резервне копіювання не критичне, оскільки індекс повністю відновлюється з MongoDB-каталогу шляхом запуску повторної індексації. Цей процес займає 30-60 хвилин для каталогу з 100 тисяч товарів та виконується автоматично після відновлення Search Service.

Регулярно (раз на квартал) проводяться навчання з відновлення системи — Disaster Recovery Drill. Команда DevOps моделює різні сценарії відмови (втрата майстер-вузла PostgreSQL, відмова цілого регіону хмари, помилкове видалення колекції MongoDB) та виконує процедури відновлення. За результатами навчань уточнюються процедури та оновлюються інструкції.

Висновки до розділу 2

У другому розділі спроектовано інформаційне забезпечення системи маркетплейсу. Обґрунтовано застосування принципу «база даних на сервіс», за якого логічна модель даних розпадається на вісім автономних доменів, по одному на кожен мікросервіс. Зв'язки між сутностями різних доменів реалізуються через ідентифікатори на рівні бізнес-логіки, а не через зовнішні ключі СУБД, що забезпечує можливість незалежного розгортання сервісів.

Проведено порівняльний аналіз чотирьох класів СУБД — реляційних, документоорієнтованих, ключ-значення та пошукових індексів. На основі аналізу обрано модель поліглот-персистентності, що включає чотири СУБД: PostgreSQL (для транзакційних даних — користувачі, замовлення, ліцензії), MongoDB (для каталогу з гнучкою схемою), Redis (для кошика та кешу) та Elasticsearch (для повнотекстового пошуку). Кожна СУБД використовується для тих типів даних, для яких вона є оптимальною за критеріями узгодженості, гнучкості схеми, швидкості та можливостей масштабування.

Розроблено фізичну структуру баз даних для ключових мікросервісів: детально описано таблиці users та seller_profiles у PostgreSQL для User Service, документну колекцію Product у MongoDB для Catalog Service, hash-структуру кошика в Redis для Cart Service. Визначено типи даних, обмеження цілісності та індекси. Зокрема, у Catalog Service використано поле attributes типу Mixed, що дозволяє зберігати різнорідні характеристики товарів різних категорій без створення додаткових таблиць.

РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Архітектура системи

Програмне забезпечення електронного маркетплейсу побудоване за мікросервісною архітектурою та складається з трьох основних логічних рівнів: рівень клієнта, рівень шлюзу та рівень мікросервісів. Кожен з рівнів виконує чітко окреслену функцію та взаємодіє з іншими через стандартизовані інтерфейси.

Рівень клієнта реалізовано як Single Page Application (SPA) на базі бібліотеки React. Клієнтська частина виконується безпосередньо у браузері користувача та надає графічний інтерфейс для всіх можливостей системи. Клієнт не звертається безпосередньо до окремих мікросервісів — натомість усі його HTTP-запити проходять через API Gateway, що є єдиною точкою входу. Такий підхід дає кілька переваг: клієнтський код не повинен знати про внутрішню структуру серверної частини; зміна декомпозиції мікросервісів не вимагає змін у клієнті; легше реалізувати глобальні політики безпеки та обмеження частоти запитів [21].

Рівень шлюзу реалізує паттерн API Gateway. Це єдиний серверний компонент, що приймає всі HTTP-запити від клієнтів та маршрутизує їх до відповідних внутрішніх мікросервісів. API Gateway виконує такі функції: автентифікація користувача шляхом перевірки JWT-токена в заголовку Authorization; маршрутизація на основі URL-патернів (наприклад, /api/catalog/* → Catalog Service, /api/cart/* → Cart Service); агрегація відповідей кількох мікросервісів в єдину відповідь, коли клієнт потребує даних з різних доменів; обмеження частоти запитів (rate limiting) для захисту від зловживань; реєстрація запитів для подальшого аналізу [22].

Рівень мікросервісів складається з восьми незалежних серверних застосунків, кожен з яких реалізує одну бізнес-можливість. Перелік мікросервісів та їх зон відповідальності наведено у табл. 3.1.

Таблиця 3.1

Перелік мікросервісів системи

Мікросервіс	**Сховище даних**	**Зона відповідальності**
User Service	PostgreSQL	Реєстрація, автентифікація, профілі покупців і продавців
Catalog Service	MongoDB	Каталог товарів, категорії, атрибути, фотографії
Search Service	Elasticsearch	Повнотекстовий пошук, фасетна фільтрація, сортування
Cart Service	Redis	Кошик покупця, додавання/видалення позицій
Order Service	PostgreSQL	Оформлення замовлень, історія, статуси, платежі
License Service	PostgreSQL	Типи ліцензій, видача ключів доступу
Library Service	PostgreSQL	Особиста бібліотека покупця, повторні завантаження
Delivery Service	PostgreSQL	Завдання доставки, миттєва видача цифрових товарів

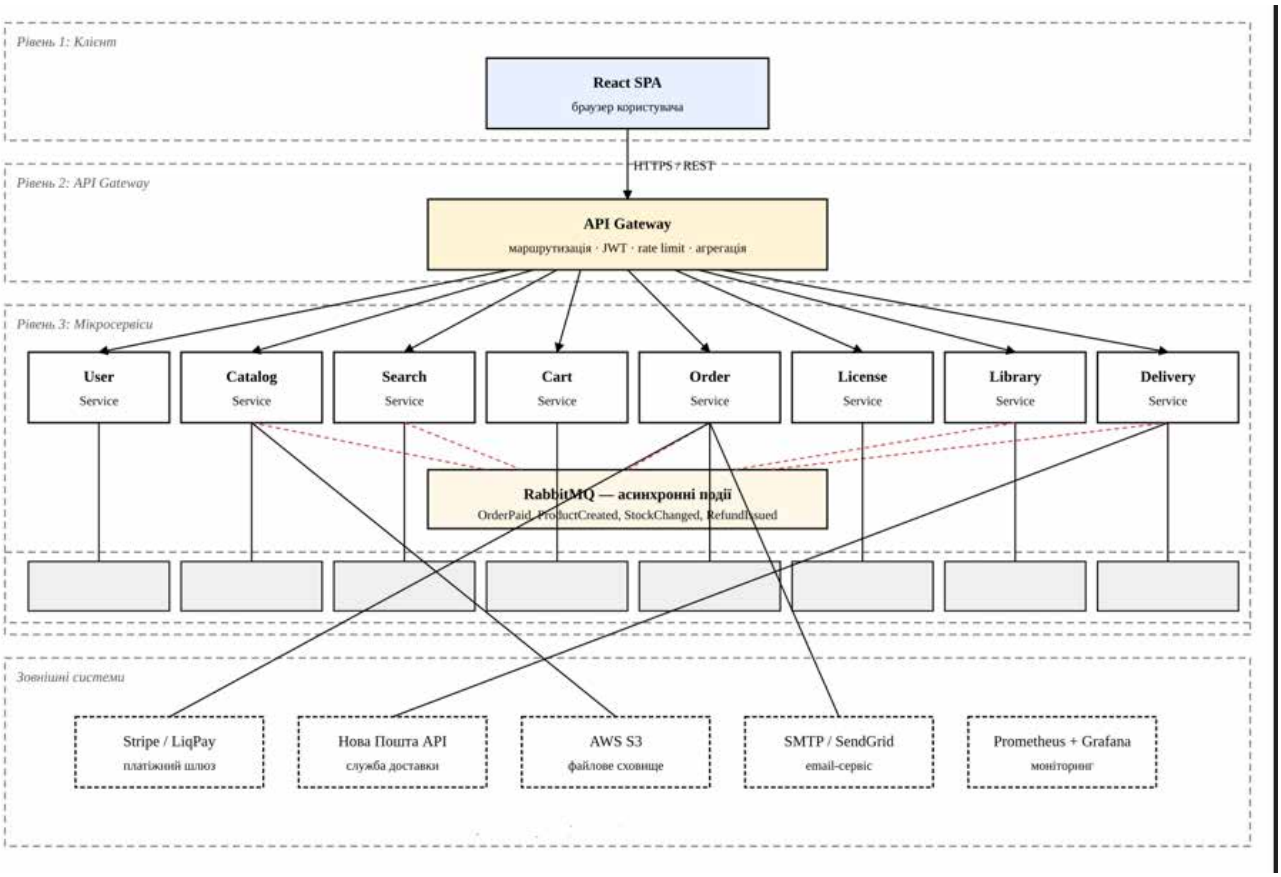
Взаємодія між мікросервісами реалізована у двох режимах. Перший режим — синхронні запити через REST API. Використовується для випадків, коли результат потрібно отримати негайно. Наприклад, перед додаванням товару до кошика Cart Service синхронно запитує Catalog Service: «чи існує товар з таким ідентифікатором і яка його поточна ціна». Другий режим — асинхронна взаємодія через брокер повідомлень RabbitMQ. Використовується для подій, які не вимагають негайної відповіді. Наприклад, при здійсненні замовлення Order Service публікує подію OrderPlaced; на цю подію підписані Catalog Service (для зменшення залишку на складі), Library Service (для додавання цифрових товарів у бібліотеку), Search Service (для оновлення лічильника продажів у пошуковому індексі). Така подієво-орієнтована

взаємодія забезпечує слабку зв'язаність сервісів — публікувач не знає, хто саме споживає його події [23].

Окремо реалізована загальна горизонтальна функціональність — Auth Module, що використовується API Gateway та кожним мікросервісом для верифікації JWT-токенів. Токени генеруються User Service при успішному вході та підписуються секретним ключем. Кожен токен містить ідентифікатор користувача, роль (buyer або seller) та термін дії (типово — 24 години). Для верифікації токена не потрібно звертатися до User Service: достатньо перевірити цифровий підпис локально, що значно прискорює обробку запитів.

Зовнішні сервіси, з якими взаємодіє система: платіжний шлюз (Stripe або LiqPay) — для обробки платежів; SMTP-сервер — для надсилання транзакційних листів покупцям; хмарне сховище файлів (AWS S3 або сумісне) — для зберігання фотографій товарів та цифрових продуктів; служба доставки (інтеграція з API «Нова Пошта») — для оформлення відправлень фізичних товарів.

Загальну архітектурну схему системи наведено на рис. 3.1.



3.1. Діаграма класів та кооперацій

Діаграма класів (рис. 3.2) відображає статичну структуру основних абстракцій системи та відносини між ними. У контексті мікросервісної архітектури діаграма класів представляє доменні моделі — об'єкти, що інкапсулюють бізнес-логіку всередині кожного сервісу. У системі виділено вісім ключових класів: User, SellerProfile, Product, Category, Cart, Order, Payment, LibraryEntry [6].

Клас User представляє користувача системи. Атрибути: id, email, passwordHash, name, role, createdAt. Методи: register() — реєстрація нового користувача з валідацією email та хешуванням пароля; authenticate() — перевірка введених облікових даних; generateToken() — створення JWT-токена після успішної автентифікації.

Клас SellerProfile розширює функціональність User для продавців. Атрибути: id, userId, storeName, description, rating, reviewsCount, verified. Методи: updateRating() — перерахунок середнього рейтингу при отриманні нового відгуку; getProducts() — отримання переліку товарів продавця через звернення до Catalog Service.

Клас Product є центральним класом доменної моделі каталогу. Атрибути: id, title, description, price, categoryId, sellerId, stock, photos, attributes, rating, sales. Методи: decreaseStock(quantity) — зменшення залишку при здійсненні замовлення; updateRating(newReview) — оновлення рейтингу; canBePurchased() — перевірка можливості купівлі (наявність на складі та активність продавця).

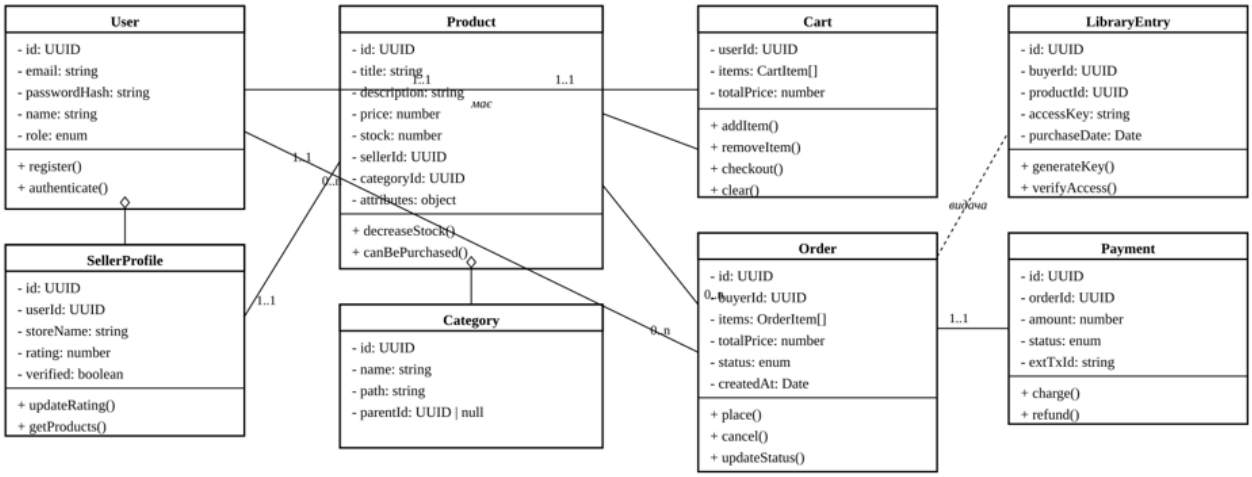
Клас Cart описує кошик покупця. Атрибути: userId, items (масив CartItem), totalPrice. Методи: addItem(productId, quantity) — додавання товару з перевіркою наявності; removeItem(productId) — видалення; updateQuantity(productId, quantity) — зміна кількості; calculateTotal() — обчислення загальної суми; clear() — очищення кошика після оформлення замовлення.

Клас Order описує замовлення. Атрибути: id, buyerId, items (масив OrderItem), totalPrice, status, deliveryAddress, paymentId, createdAt. Методи:

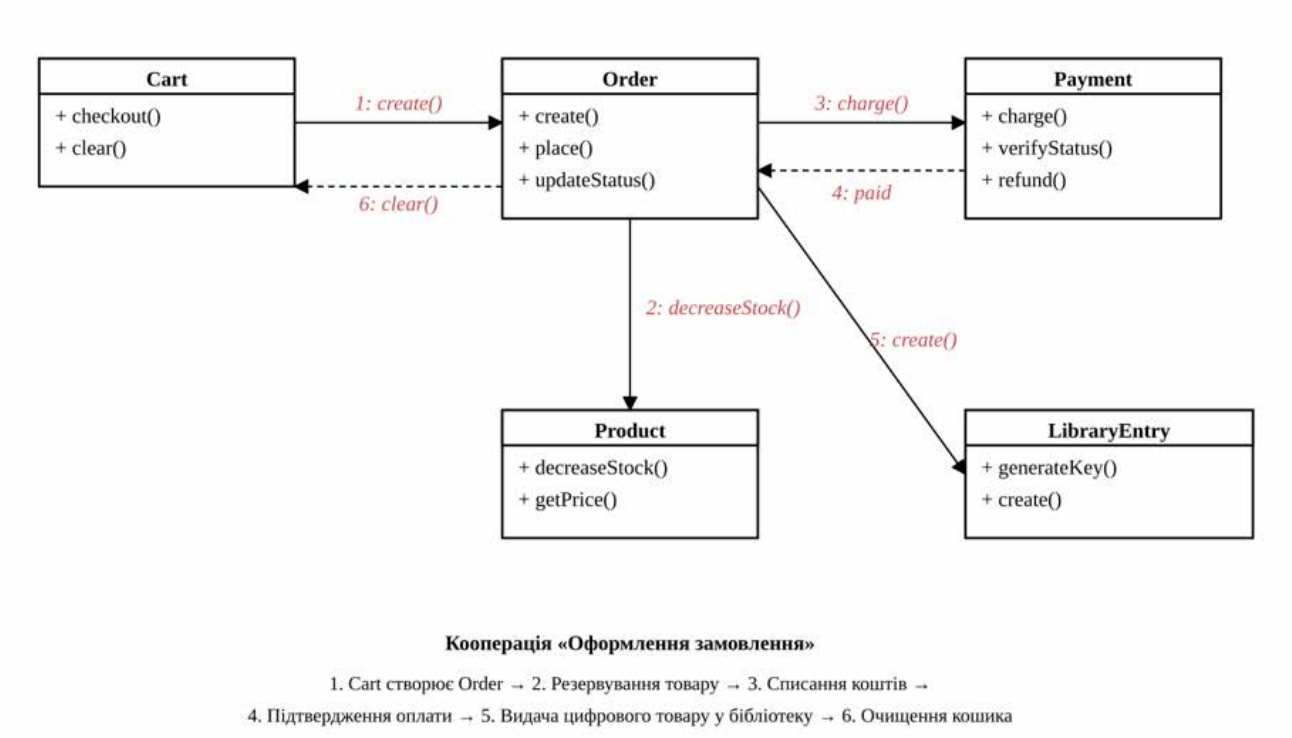
place() — підтвердження замовлення з ініціюванням платежу; cancel() — скасування з поверненням коштів; updateStatus(newStatus) — зміна статусу з фіксацією події.

Клас Payment представляє платіжну транзакцію. Атрибути: id, orderId, amount, status, externalTransactionId, paymentMethod. Методи: charge() — ініціація списання через зовнішній шлюз; refund() — повернення коштів; verifyStatus() — періодична перевірка статусу зовнішньої транзакції.

Клас LibraryEntry представляє запис у бібліотеці покупця для цифрових товарів. Атрибути: id, buyerId, productId, licenseType, accessKey, purchaseDate. Методи: generateAccessKey() — створення унікального ключа доступу; verifyAccess(key) — перевірка права завантаження.



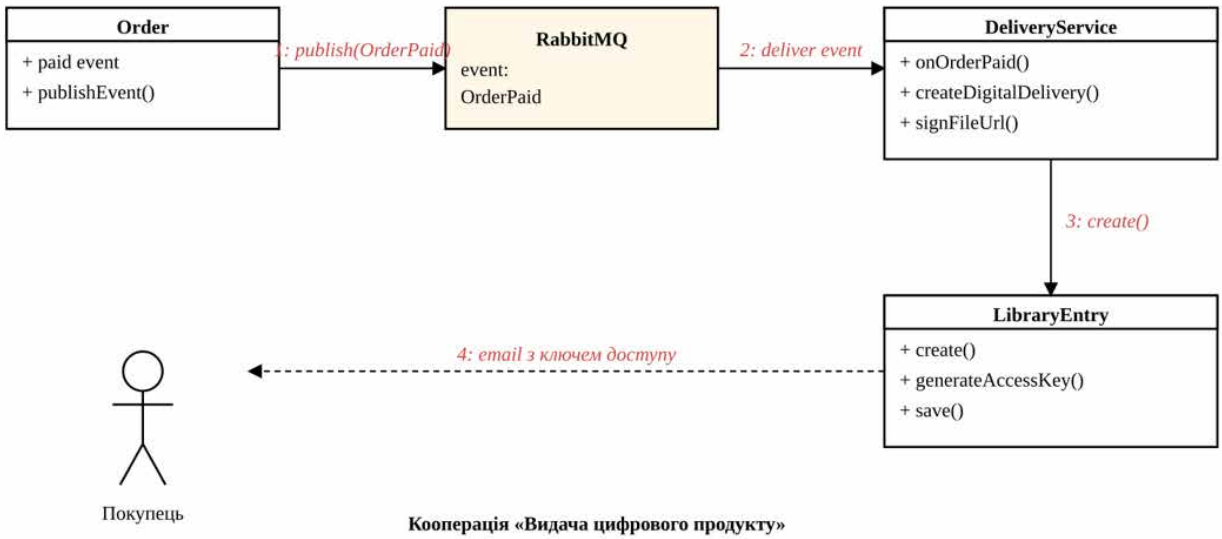
Для деталізації взаємодії між класами побудовано дві діаграми кооперацій. Перша кооперація «Оформлення замовлення» (рис. 3.3) показує послідовність взаємодії між класами Cart, Order, Payment та Product у процесі здійснення покупки. При виклику Cart.checkout() створюється новий екземпляр Order, для кожної позиції викликається Product.decreaseStock() для резервування товару, після чого ініціюється Payment.charge(). При успішному платежі Order.updateStatus('paid') фіксує оплачений статус, а Cart.clear() очищає кошик.



Кооперація «Оформлення замовлення»

1. Cart створює Order → 2. Резервування товару → 3. Списання коштів →
4. Підтвердження оплати → 5. Видача цифрового товару у бібліотеку → 6. Очищення кошика

Друга кооперація «Видача цифрового продукту» (рис. 3.4) відображає взаємодію між Order, LibraryEntry та DeliveryService при здійсненні покупки цифрового товару. Після успішної оплати DeliveryService отримує подію OrderPaid та для кожного цифрового товару створює новий екземпляр LibraryEntry з унікальним accessKey. Цей ключ зберігається в базі даних та передається покупцю через email-повідомлення.



Кооперація «Видача цифрового продукту»

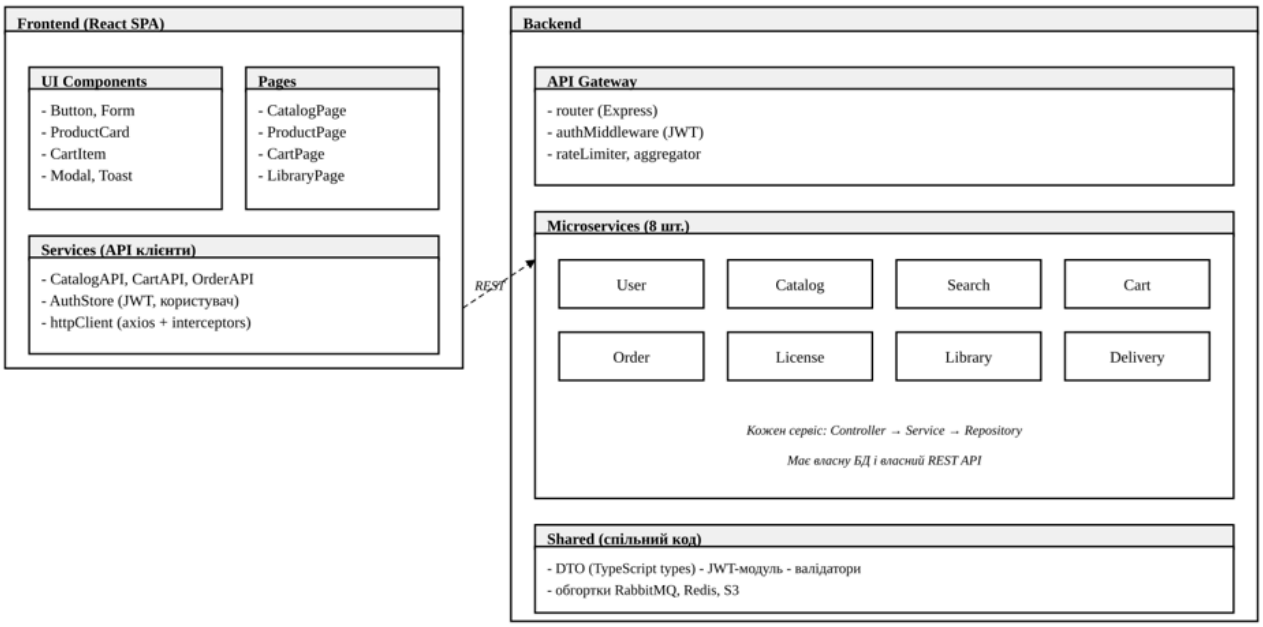
Order публікує OrderPaid → DeliveryService створює LibraryEntry з access_key → Покупець отримує email

3.1 Діаграма пакетів та компонентів

Діаграма пакетів (рис. 3.5) відображає організаційну структуру програмного забезпечення та залежності між його складовими частинами. Система розподілена між двома основними пакетами верхнього рівня: Клієнт (Frontend) та Сервер (Backend) [22].

Пакет Frontend містить три підпакети. Підпакет UI Components об'єднує елементи інтерфейсу: компоненти кнопок, форм, карток товару, кошика. Підпакет Pages містить компоненти сторінок — каталог, картка товару, кошик, оформлення замовлення, особистий кабінет, бібліотека. Підпакет Services інкапсулює клієнтський шар взаємодії з API: HTTP-клієнт, методи звернень до різних доменів (CatalogAPI, CartAPI, OrderAPI), управління JWT-токеном.

Пакет Backend містить три підпакети. Підпакет API Gateway реалізує єдину точку входу: маршрутизатор, middleware автентифікації, middleware обмеження частоти запитів, агрегатор відповідей. Підпакет Microservices об'єднує реалізації восьми мікросервісів, перелічених у табл. 3.1. Підпакет Shared містить спільний код, що використовується кількома сервісами: модуль роботи з JWT, типи даних DTO, утиліти валідації, обгортки над клієнтами Redis та RabbitMQ.

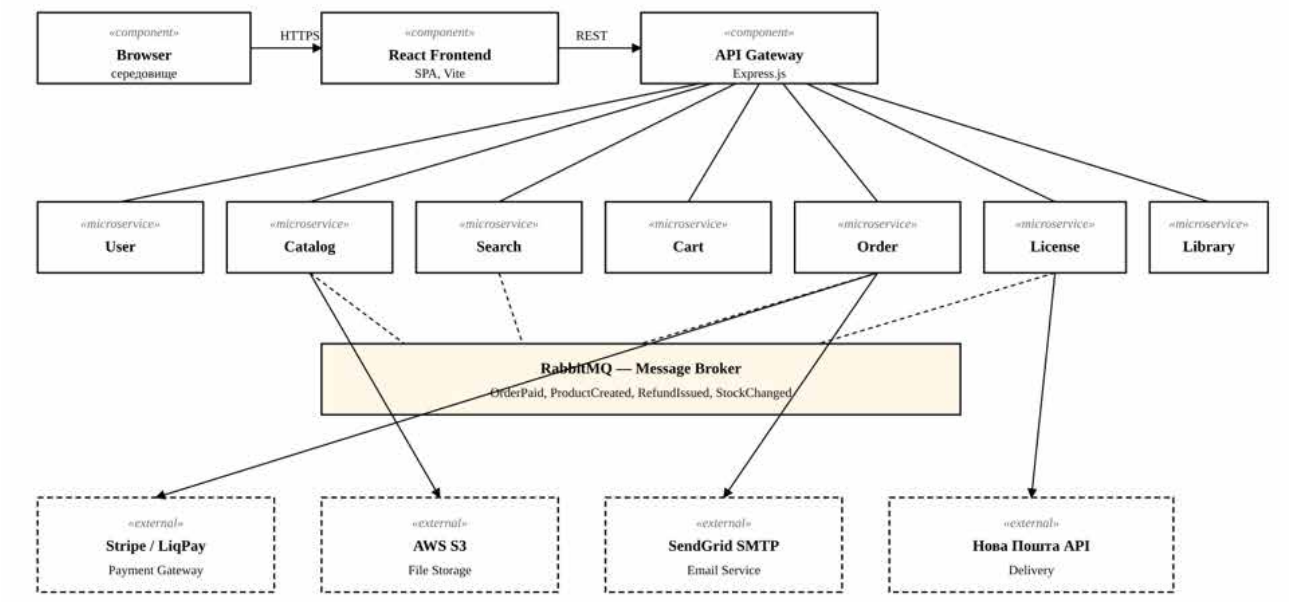


Діаграма компонентів (рис. 3.6) відображає архітектуру системи на рівні основних замінюваних компонентів та інтерфейсів взаємодії між ними. На клієнтській стороні виділено компонент Browser (середовище виконання) та компонент React Frontend. Взаємодія між ними відбувається через стандартні веб-API. React Frontend з'єднується з API Gateway через REST API через HTTPS.

Компонент API Gateway є центральним вузлом серверної частини та взаємодіє з усіма мікросервісами через внутрішні REST API. Кожен мікросервіс — це окремий компонент, який має чітко визначений інтерфейс та власне сховище даних. Між мікросервісами також прокладено канал асинхронної взаємодії через брокер повідомлень Message Broker (RabbitMQ).

Зовнішні системи представлені як окремі компоненти: Payment Gateway (Stripe/LiqPay), Email Service (SMTP), File Storage (AWS S3), Delivery API (Нова Пошта). Кожна із зовнішніх систем має відповідний адаптер у відповідному мікросервісі: Order Service використовує адаптер платіжного шлюзу, Delivery Service — адаптер служби доставки тощо.

Така архітектура забезпечує слабку зв'язаність компонентів: кожен з них може бути замінений незалежно — наприклад, заміна Stripe на LiqPay або PostgreSQL на іншу реляційну СУБД — без зміни інших частин системи.



3.2. Вибір інструментарію розробки

Вибір засобів розробки визначається архітектурними рішеннями, прийнятими у попередніх розділах, а також вимогами до продуктивності, зручності супроводу та можливості розгортання у хмарному середовищі. Перелік основних технологій та бібліотек наведено у табл. 3.2.

Таблиця 3.2

Інструментарій розробки системи маркетплейсу

Технологія	**Версія**	**Призначення**
Node.js	20+	Серверне середовище виконання JavaScript
TypeScript	5.x	Статична типізація JavaScript-коду
Express.js	4.x	Серверний фреймворк для побудови REST API
React	18.x	Бібліотека побудови інтерфейсу користувача
Vite	5.x	Інструмент збірки клієнтської частини
PostgreSQL	16	Реляційна СУБД для транзакційних даних
MongoDB	7	Документоорієнтована СУБД для каталогу
Redis	7.x	In-memory сховище для кошика та кешу
Elasticsearch	8.x	Пошуковий індекс товарів
RabbitMQ	3.13	Брокер повідомлень для подієвої взаємодії
Mongoose	8.x	ODM для роботи з MongoDB
Prisma	5.x	ORM для роботи з PostgreSQL
jsonwebtoken	9.x	Генерація та верифікація JWT-токенів
bcryptjs	2.x	Хешування паролів користувачів
Docker	25+	Контейнеризація мікросервісів
Kubernetes	1.29	Оркестрація контейнерів у кластері

Node.js обрано як основну серверну платформу через високу продуктивність обробки I/O-запитів, що є типовим навантаженням для мікросервісів маркетплейсу. Подієво-орієнтована модель Node.js та неблокуючий введення-виведення дозволяють одному екземпляру сервісу обробляти тисячі одночасних з'єднань без створення окремого потоку для кожного запиту. Окрім того, єдина мова (TypeScript) для клієнтської та серверної частини спрощує розробку та обмін типами даних [24].

TypeScript обрано як надбудову над JavaScript для всіх компонентів системи. Статична типізація суттєво знижує кількість помилок, що виявляються лише під час виконання, та поліпшує читаність коду. TypeScript також дозволяє визначити спільні типи DTO (Data Transfer Object) у пакеті Shared, що використовуються одночасно фронтендом та бекендом, забезпечуючи цілісність контрактів API.

Express.js обрано як основу для побудови HTTP-API кожного мікросервісу та API Gateway. Express має мінімалістичний API, широку екосистему middleware та є де-факто стандартом для Node.js-розробки. На альтернативу розглядалися фреймворки Fastify (швидший, але з меншою екосистемою) та NestJS (об'єктно-орієнтований, але з вищою кривою навчання). Для бакалаврського рівня обрано Express як компроміс між простотою та функціональністю [22].

React обрано для клієнтської частини завдяки компонентній моделі, широкій екосистемі та зрілості. Використання React дозволяє побудувати інтерактивний інтерфейс із динамічною зміною стану без перезавантаження сторінки, що є природним для маркетплейсу з його каталогами та фільтрами.

Docker обрано як засіб контейнеризації мікросервісів. Кожен сервіс упаковується в окремий Docker-образ, що містить його код, залежності та налаштування. Контейнери ізольовані один від одного на рівні файлової системи та мережі, що усуває класичну проблему «у мене на машині все працює». Docker-образи зберігаються у Docker Registry та можуть бути розгорнуті в будь-якому середовищі, що підтримує Docker [25].

Kubernetes обрано як платформу оркестрації для виробничого розгортання. Kubernetes автоматично запускає необхідну кількість екземплярів кожного мікросервісу, балансує навантаження між ними, перезапускає несправні контейнери, виконує оновлення без переривання обслуговування. Конфігурація розгортання описується у декларативному форматі YAML, що дозволяє вести версіонування інфраструктури як коду [26].

3.5. Алгоритм оформлення замовлення та ключові модулі

Центральним бізнес-процесом системи маркетплейсу є оформлення замовлення. Цей процес охоплює взаємодію між чотирма мікросервісами (Cart, Order, Catalog, Payment) та зовнішнім платіжним шлюзом. Алгоритм процесу реалізовано в модулі checkout-orchestrator.ts у складі Order Service та складається з кількох послідовних етапів.

На першому етапі клієнт надсилає запит на оформлення замовлення до API Gateway з переліком ідентифікаторів товарів та обраним способом доставки. API Gateway перевіряє JWT-токен покупця та маршрутизує запит до Order Service з підставленим у заголовок ідентифікатором користувача.

На другому етапі Order Service отримує з Cart Service поточний вміст кошика покупця. Якщо кошик порожній, повертається помилка. У іншому випадку для кожної позиції кошика виконується синхронний запит до Catalog Service для отримання актуальної інформації про товар: поточну ціну (вона може відрізнятися від збереженої в кошику), наявність на складі, активність продавця. Якщо хоча б один товар недоступний або його залишок менше запитуваної кількості, замовлення скасовується з відповідним повідомленням покупцю.

На третьому етапі Order Service створює запис замовлення у PostgreSQL зі статусом created та підрахованою сумою. Транзакція оформляється у межах однієї бази даних, що гарантує атомарність створення замовлення та його позицій.

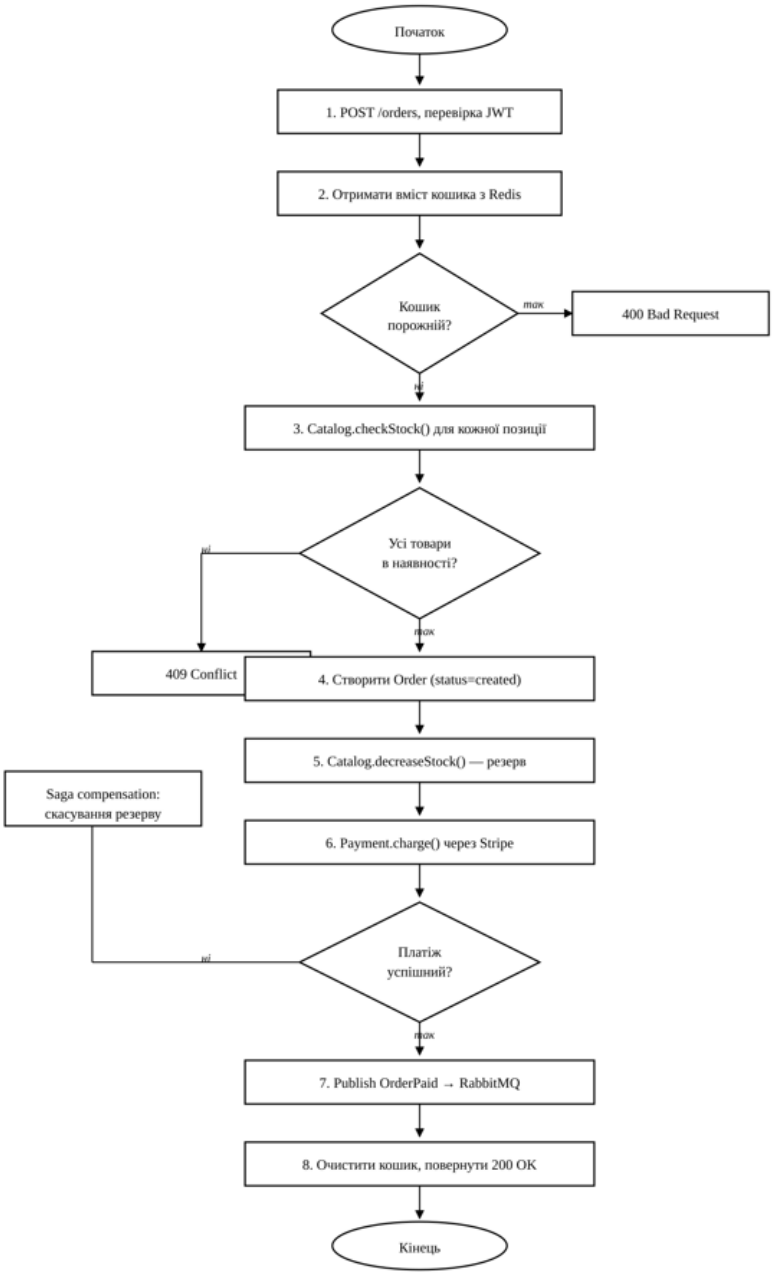
На четвертому етапі Order Service ініціює платіжну транзакцію через адаптер платіжного шлюзу. Адаптер формує запит до зовнішнього сервісу Stripe або LiqPay та повертає посилання на форму оплати або токен для проведення платежу. Замовлення оновлюється статусом payment_pending.

На п'ятому етапі покупець завершує оплату у формі платіжного шлюзу. Платіжний шлюз надсилає вебхук на спеціальний ендпоінт Order Service з повідомленням про результат. У випадку успішної оплати замовлення оновлюється статусом paid та публікується подія OrderPaid у RabbitMQ.

На шостому етапі різні мікросервіси обробляють подію OrderPaid незалежно. Catalog Service зменшує залишок товарів на складі. Library Service для кожного цифрового товару в замовленні створює запис у бібліотеці покупця з унікальним ключем доступу. Delivery Service для кожного фізичного товару створює завдання на доставку та надсилає запит до API служби доставки. Email Service надсилає покупцю електронного листа з підтвердженням замовлення.

На сьомому етапі покупець може спостерігати замовлення у своєму особистому кабінеті. Для цифрових товарів — миттєво розпочати завантаження з бібліотеки. Для фізичних — отримувати оновлення статусу доставки.

Блок-схему алгоритму оформлення замовлення наведено на рис. 3.7.



Модуль search-engine.ts реалізує клієнт для роботи з Elasticsearch у складі Search Service. При надходженні пошукового запиту від користувача модуль формує запит у форматі Elasticsearch Query DSL, що поєднує: повнотекстовий пошук за полями title та description з вагою; фільтр за обраною категорією; фільтр за діапазоном ціни; мінімальний рейтинг; сортування за обраним критерієм (популярність, ціна, рейтинг). Результати ранжуються за релевантністю та повертаються разом з агрегаціями для фасетної фільтрації: кількість товарів у кожній підкатегорії, розподіл за ціновими діапазонами, розподіл за рейтингом.

Модуль cart-service.ts реалізує операції з кошиком у складі Cart Service. Завдяки використанню Redis з підтримкою hash-структур усі операції виконуються однією командою без необхідності блокувань: HSET cart:{userId} {productId} {jsonValue} для додавання, HDEL для видалення, HGETALL для отримання повного вмісту кошика. При додаванні позиції модуль попередньо звертається до Catalog Service для перевірки наявності товару та отримання актуальної ціни. Передача актуальної ціни в кошик дозволяє запам'ятати «обіцяну» покупцю ціну: навіть якщо продавець підвищить її, у кошику покупця залишиться попередня ціна до моменту оформлення замовлення.

Модуль jwt-auth.ts реалізує спільний для всіх сервісів механізм автентифікації. При вході покупця User Service генерує JWT-токен з payload {userId, role, exp}, підписаний секретним ключем алгоритмом HS256. Токен повертається клієнту, який зберігає його в localStorage та додає до заголовка Authorization кожного наступного запиту. API Gateway та кожен мікросервіс мають middleware authMiddleware, що автоматично перевіряє підпис токена та витягує payload до об'єкта запиту.

3.6. Демонстрація інтерфейсу користувача MerchFlow

Для розробки інтерфейсу користувача MerchFlow обрано React 18 у поєднанні з підходом single-file styling — весь CSS зосереджено в одному модулі styles.js та підключається через тег <style> без зовнішніх CSS-фреймворків. Застосунок реалізовано як Single Page Application (SPA) з клієнтською маршрутизацією через стан React Context. Інтерфейс побудований на темній кольоровій схемі з акцентним кольором #7c3aed (фіолетовий) та шрифтовою парою Sora (UI-елементи) і Space Mono (числові значення).

Екран авторизації відображається при першому відвідуванні або після завершення сесії. Він містить логотип платформи, поля введення електронної пошти та пароля, кнопку входу та демонстраційну підказку. Успішна автентифікація зберігає сесію у localStorage за ключами mf:auth та mf:user, після

чого користувач отримує доступ до основного інтерфейсу. Інтерфейс екрану авторизації наведено на рис. 3.14.

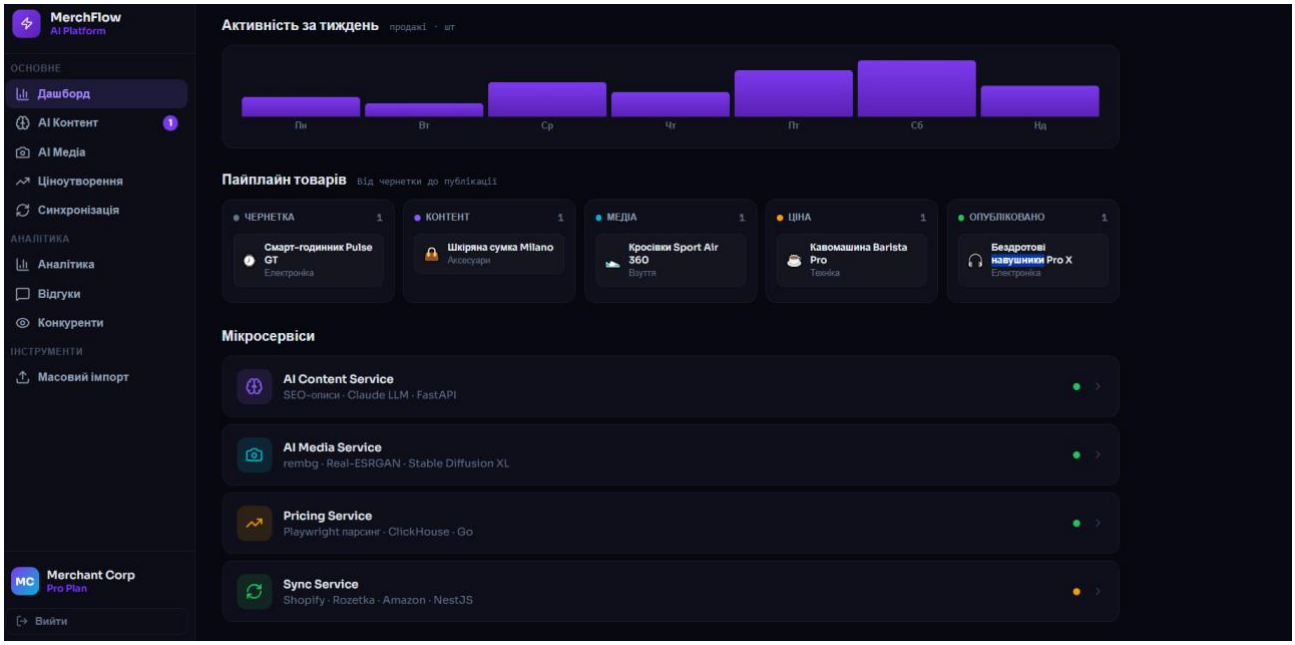


рис. 3.14.

Головний інтерфейс застосунку має дволайаутну структуру: фіксована бічна панель (Sidebar) шириною 224 пікселі та основна робоча область (content). У верхній частині робочої області розташована панель topbar з компонентом сповіщень. Бічна панель містить логотип платформи, три навігаційні секції (ОСНОВНЕ, АНАЛІТИКА, ІНСТРУМЕНТИ) та картку поточного мерчанта з кнопкою виходу.

Дашборд є стартовою сторінкою після авторизації. Він містить чотири статистичні картки з ключовими показниками: кількість товарів у роботі, кількість згенерованих описів, кількість оброблених фотографій та вплив на дохід. Нижче розташований бар-чарт активності продажів за поточний тиждень (понеділок–неділя). Центральним елементом дашборду є kanban-пайплайн — горизонтальна послідовність з п'яти колонок, що відображає розподіл товарів за статусами: Чернетка, Контент, Медіа, Ціна, Опубліковано. В нижній частині розміщено картки чотирьох мікросервісів з індикаторами онлайн-статусу та переходом до відповідного розділу. Інтерфейс дашборду наведено на рис. 3.15.

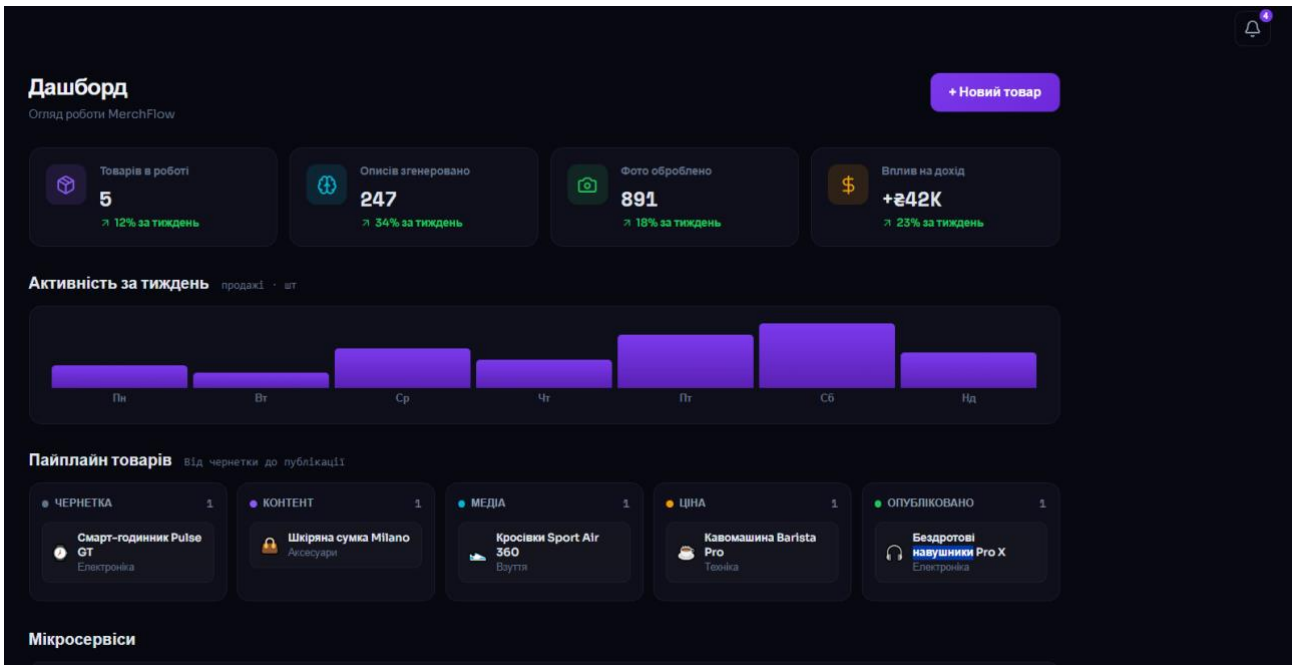


рис. 3.15

Сторінка AI Content Service призначена для автоматичної генерації товарних описів засобами великої мовної моделі. Користувач обирає товар зі спадного списку та натискає кнопку генерації. Процес відображається через трикроковий прогрес-бар: аналіз вхідних даних, генерація через LLM, SEO-оптимізація. Результат містить оптимізований заголовок, розгорнутий опис, масив тегів, таблицю технічних атрибутів та SEO-оцінку. Передбачено вибір між трьома тональними варіантами: Офіційний (SEO-оцінка 94), Розмовний (88) та SEO-агресивний (97). Затверджений контент переводить товар у статус content_ready. Інтерфейс генератора контенту наведено на рис. 3.16.

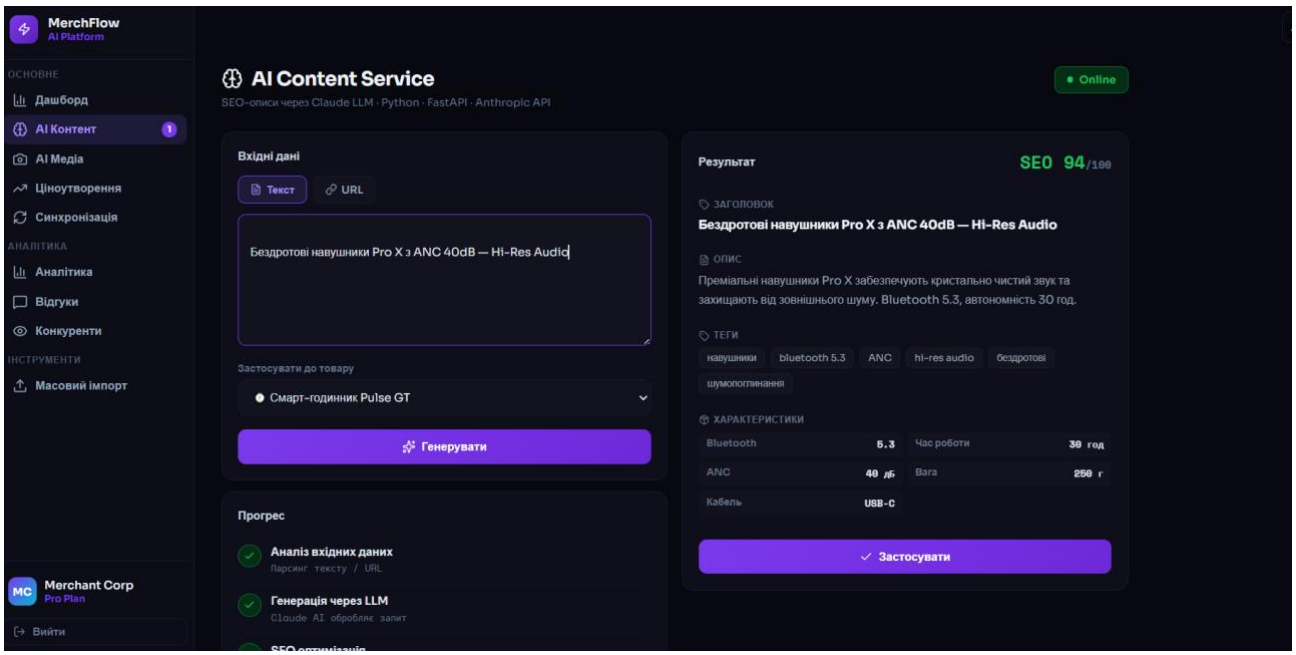


рис. 3.16

Сторінка AI Media Service реалізує чотириетапний конвеєр обробки зображень. Зона drag-and-drop приймає файл зображення, після чого послідовно виконуються: завантаження у хмарне сховище (S3/MinIO), видалення фону через модель rembg/U2Net, апскейл до роздільної здатності 4K через Real-ESRGAN ×4, генерація студійного фону через Stable Diffusion XL. Кожен крок супроводжується анімованим індикатором прогресу та назвою активної операції. Після завершення обробки відображається прев'ю результату, а товар отримує статус media_ready. Інтерфейс медіапроцесора наведено на рис. 3.17.

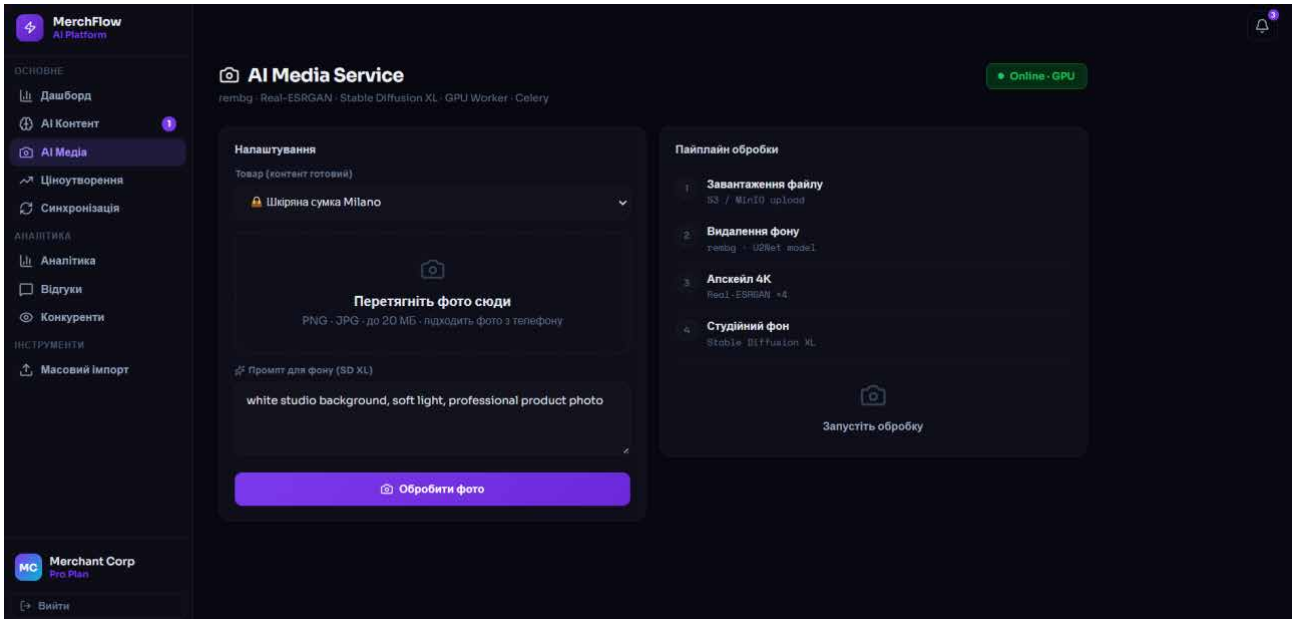


рис. 3.17

Сторінка Smart Pricing Service реалізує автоматичний аналіз ринкових цін. Після запуску аналізу сервіс імітує збір цінових даних із п'яти платформ (Rozetka, Prom.ua, OLX, Hotline, Amazon UA) через Playwright. Результат відображається у двоколонковому макеті: ліва частина містить графік динаміки середніх цін за сім днів та таблицю конкурентів з відхиленням від середнього і статусом наявності; права частина — рекомендовану ціну з логікою розрахунку (на 5% нижче середньої, маржа 28%, прогноз попиту +18%), показник впевненості моделі 87% та перемикач автоматичного щогодинного коригування в межах ±15%. Підтвердження ціни переводить товар у статус priced. Інтерфейс модуля ціноутворення наведено на рис. 3.18.

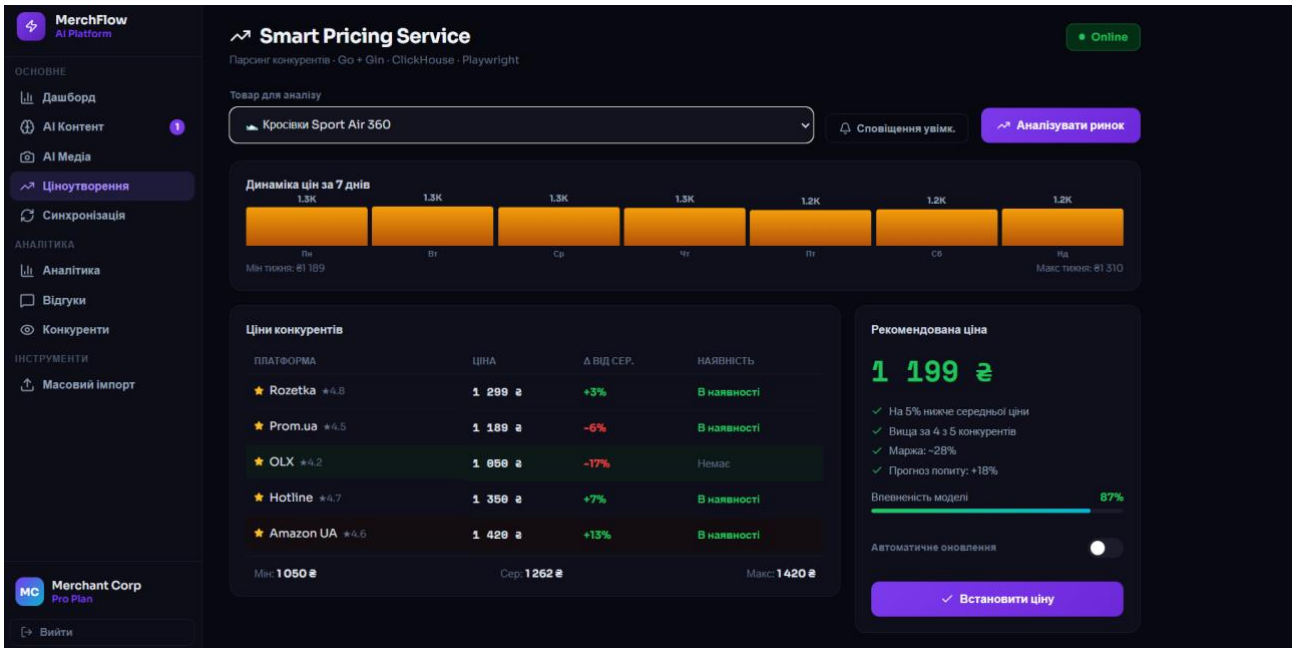


рис. 3.18

Сторінка Inventory & Sync Service керує публікацією товарів на маркетплейси. Верхня секція відображає чотири картки підключених платформ (Shopify, Rozetka, Amazon, Prom.ua) зі статусами підключення: connected, pending, error. Для платформ зі статусом connected доступна кнопка синхронізації, для error — кнопка Retry з анімацією повторного підключення, для pending — кнопка підключення API. Нижче розміщено чергу публікації з товарами у статусі priced та список вже опублікованих товарів. В нижній частині сторінки розташовано журнал синхронізацій з часовими мітками,

кількістю товарів та статусом кожної операції. Інтерфейс менеджера синхронізації наведено на рис. 3.19.

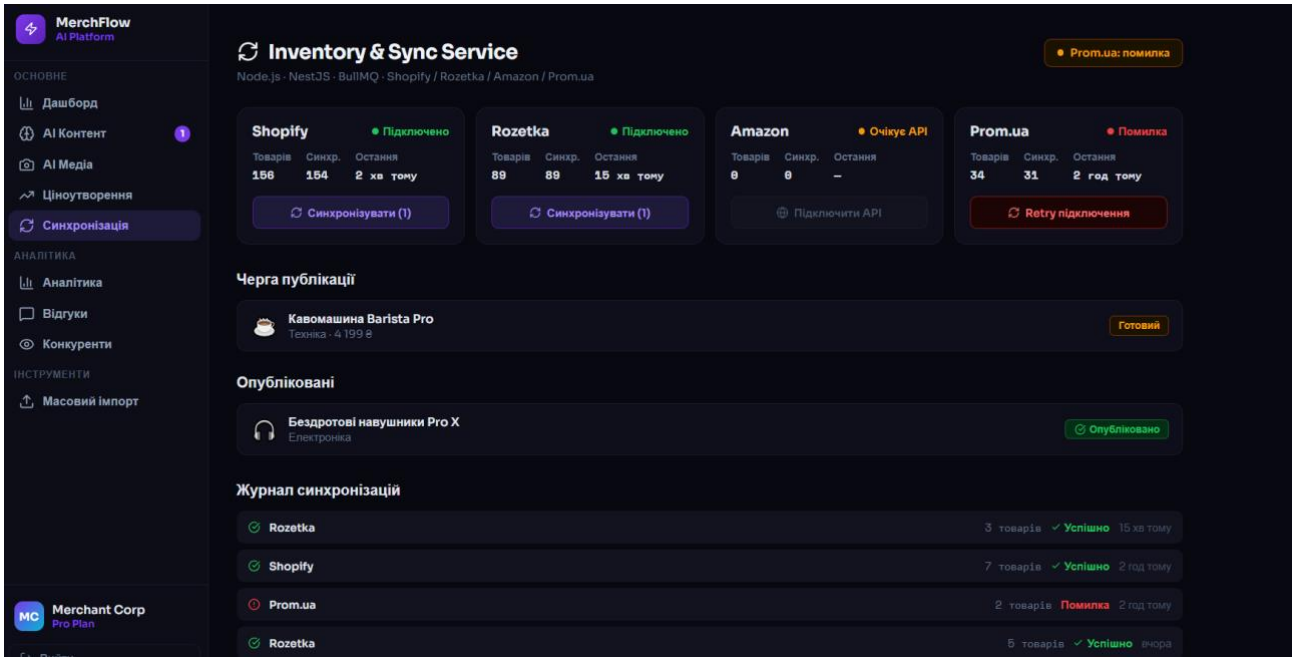


рис. 3.19

Сторінка Аналітики відображає зведені показники ефективності платформи. Чотири статистичні картки містять: кількість продажів за тиждень, дохід за тиждень, кількість унікальних відвідувачів та загальну конверсію. Нижче розміщено два бар-чарти (продажі та дохід по днях тижня), горизонтальні прогрес-бари розподілу доходу за категоріями товарів, воронку продажів (Відвідувачі → Перегляди → Кошик → Оплата) з конверсійними коефіцієнтами між етапами, а також таблицю топ-товарів з показниками продажів, доходу та конверсії. Інтерфейс сторінки аналітики наведено на рис. 3.20.

Сторінка управління відгуками відображає картки відгуків покупців з різних маркетплейсів. Кожна картка містить назву товару, платформу, зірковий рейтинг, текст відгуку, sentiment-бейдж (позитивний/нейтральний/негативний) та дату. Передбачено фільтрацію за тональністю та статусом відповіді. Для кожного відгуку доступна функція генерації AI-відповіді: система аналізує тональність та формує відповідний шаблон відповіді. Надіслана відповідь позначає картку як опрацьовану. Інтерфейс модуля відгуків наведено на рис. 3.21.

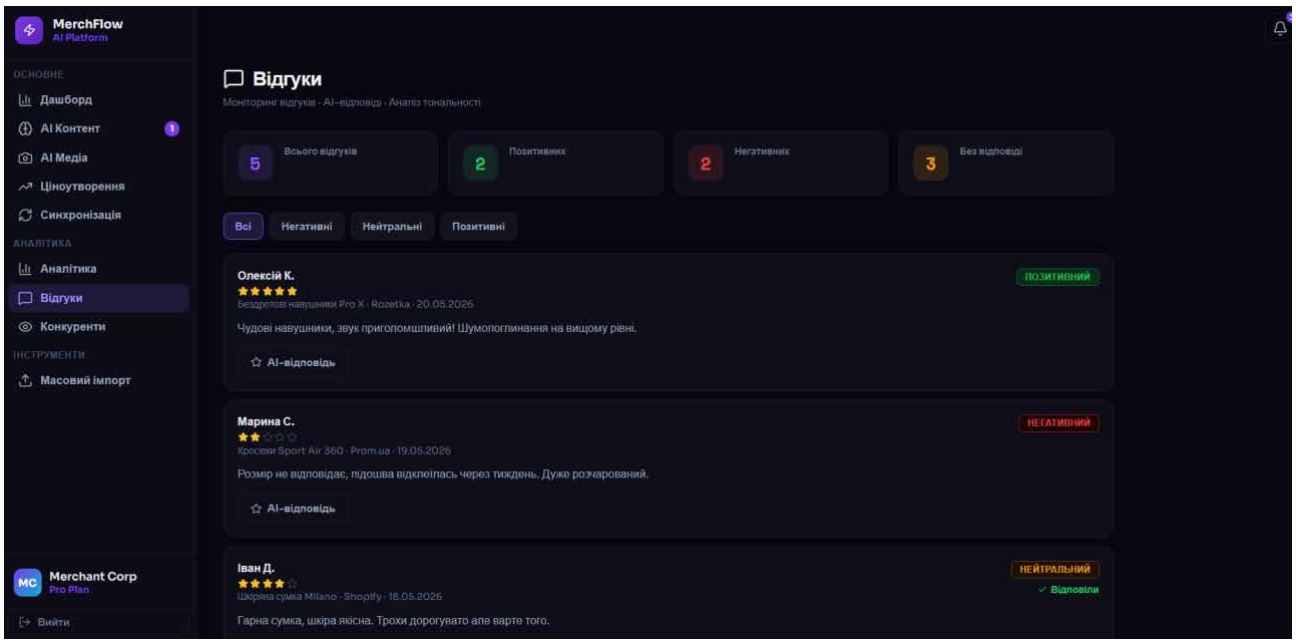


рис. 3.21.

Сторінка моніторингу конкурентів відображає картки відстежуваних магазинів з показниками: кількість товарів, середня ціна, зміна ціни відносно попереднього сканування. Картки зі значним зниженням цін виділяються кольором та індикатором alert. Кнопка сканування запускає повторну перевірку даних. Передбачено форму для додавання нового конкурента за URL. Інтерфейс трекера конкурентів наведено на рис. 3.22.

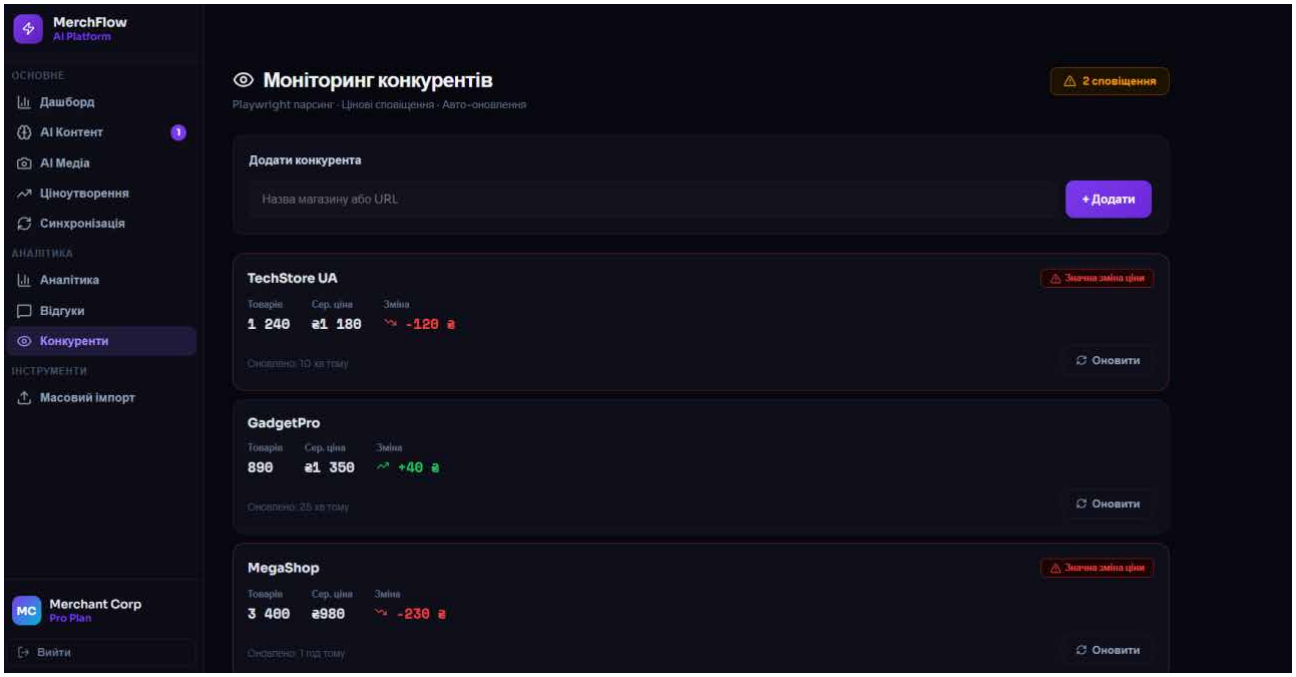


рис. 3.22.

Сторінка масового імпорту товарів реалізує drag-and-drop зону для завантаження файлів. Після вибору файлу відображається таблиця прев'ю з

переліком товарів, їх категоріями, цінами та кількістю на складі. При підтвердженні імпорту для кожного товару запускається анімований прогрес-бар, що імітує послідовну обробку. Кожен успішно імпортований товар додається до загального каталогу зі статусом draft та потрапляє на початок kanban-пайплайну. Інтерфейс модуля масового імпорту наведено на рис. 3.23.

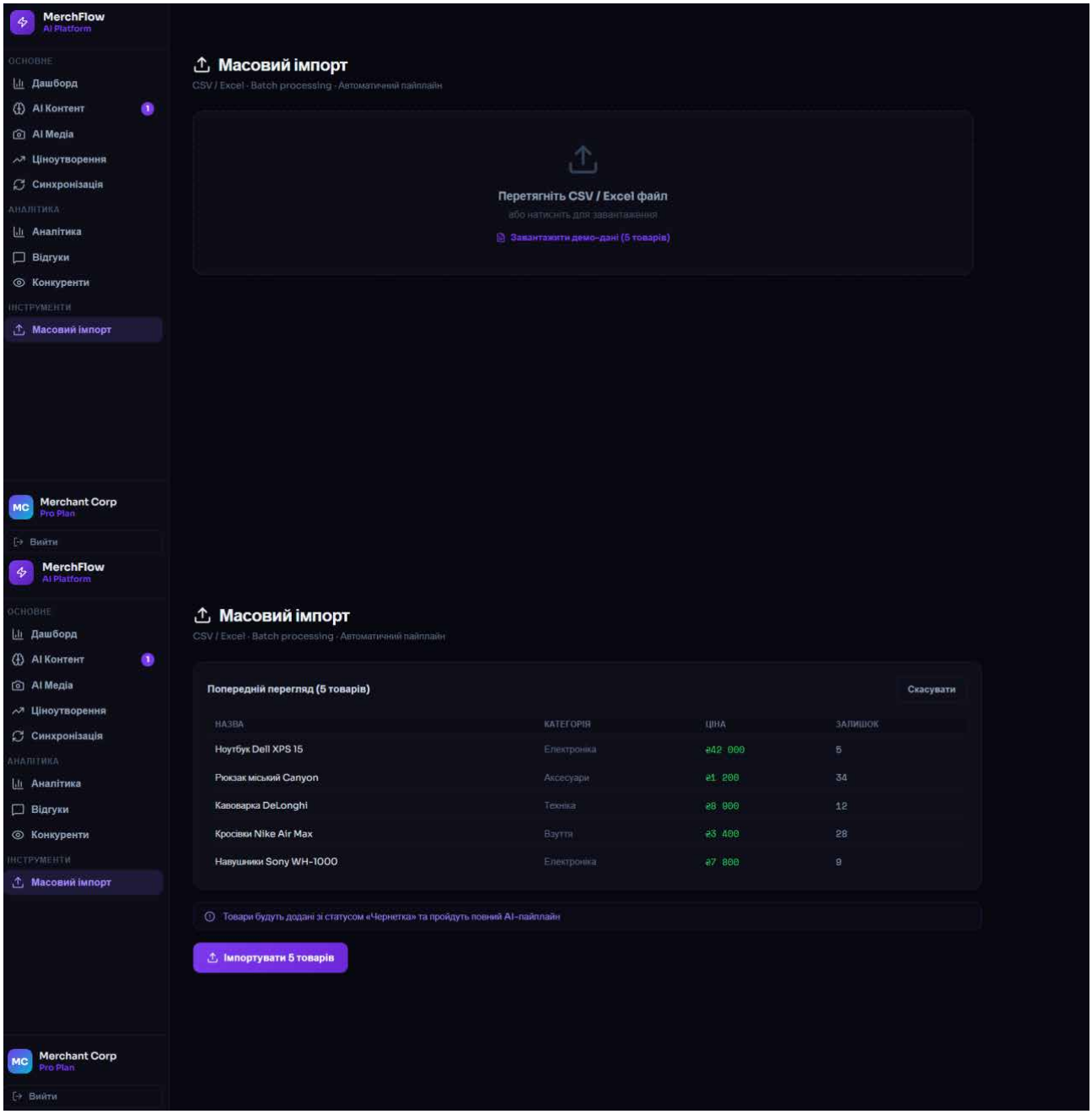


рис. 3.23.

Система сповіщень реалізована у вигляді компонента NotifPanel, розташованого у верхній панелі інтерфейсу. Іконка дзвіночка відображає лічильник непрочитаних сповіщень. При натисканні з'являється випадаючий список з хронологічним переліком подій платформи. Кожне сповіщення має

кольоровий індикатор типу: червоний (alert — цінові зміни конкурентів), зелений (success — завершена синхронізація), жовтий (review — новий відгук), фіолетовий (info — системні події). Кнопка «Прочитати всі» скидає лічильник непрочитаних.

Висновки до розділу 3

У третьому розділі описано розробку програмного забезпечення системи маркетплейсу. Визначено триярусну архітектуру системи на основі рівнів клієнта, API Gateway та мікросервісів. Виділено вісім мікросервісів, кожен з яких має чітко окреслену зону відповідальності та власне сховище даних: User, Catalog, Search, Cart, Order, License, Library, Delivery.

Обґрунтовано двохрежимну модель взаємодії між мікросервісами: синхронні REST-запити для випадків, що потребують негайної відповіді, та асинхронні події через RabbitMQ для слабко зв'язаних дій. Описано загальний механізм автентифікації на основі JWT-токенів, що використовується API Gateway та усіма мікросервісами без необхідності взаємних звернень.

Побудовано UML-діаграми класів, кооперацій, пакетів та компонентів, що відображають статичну та динамічну структуру системи. Виділено вісім доменних класів — User, SellerProfile, Product, Category, Cart, Order, Payment, LibraryEntry — кожен з яких інкапсулює відповідну бізнес-логіку.

Обґрунтовано вибір технологічного стеку з 16 інструментів та бібліотек, що охоплюють серверну (Node.js, TypeScript, Express.js), клієнтську (React, Vite), сховища даних (PostgreSQL, MongoDB, Redis, Elasticsearch), брокер повідомлень (RabbitMQ) та засоби розгортання (Docker, Kubernetes). Описано детальний алгоритм оформлення замовлення з участю чотирьох мікросервісів та зовнішнього платіжного шлюзу, а також інші ключові модулі системи.

Проведено детальну демонстрацію інтерфейсу користувача, що охоплює основні сторінки та елементи системи: головну сторінку з каталогом, картку товару, кошик, оформлення замовлення, особисту бібліотеку покупця та кабінет продавця з аналітикою.

РОЗДІЛ 4 ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЯ СИСТЕМИ

4.1. Тестування системи

Тестування є невід'ємним етапом розробки програмного забезпечення, що дозволяє виявити та усунути помилки до введення системи в експлуатацію. Особливістю тестування мікросервісної системи порівняно з монолітом є необхідність перевірки не тільки внутрішньої логіки кожного сервісу, але й взаємодії між сервісами через мережеві протоколи. У межах роботи застосовано три рівні тестування: модульне тестування окремих функцій усередині сервісів, інтеграційне тестування взаємодії сервіс-СУБД, та функціональне тестування користувацьких сценаріїв через клієнтський інтерфейс [27].

Модульне тестування реалізовано засобами фреймворку Jest. Покриття складає 78% коду серверних модулів, з пріоритетом на критичну бізнес-логіку: розрахунок суми замовлення, перевірка наявності товару, генерація та верифікація JWT-токенів. Тестові сценарії перевіряють як штатну поведінку (валідні вхідні дані), так і граничні випадки (порожній кошик, негативна кількість, недійсний токен).

Інтеграційне тестування виконується з використанням реальних екземплярів СУБД, запущених у Docker-контейнерах через бібліотеку testcontainers. Для кожного інтеграційного тесту піднімається свіжа база даних, виконується міграція схеми, проводиться тест, після чого контейнер знищується. Такий підхід забезпечує повну ізоляцію тестів один від одного та усуває проблему «забруднення» спільної тестової бази.

Функціональне тестування передбачає перевірку кожної функції системи з боку кінцевого користувача шляхом введення тестових даних та порівняння отриманих результатів із очікуваними. Тестування охопило всі основні модулі системи: автентифікацію, каталог, пошук, кошик, оформлення замовлення, бібліотеку, кабінет продавця. Результати функціонального тестування 18 основних сценаріїв використання наведено у табл. 4.1.

Таблиця 4.1

Результати функціонального тестування системи

№	Тест-кейс	Очікуваний результат	Фактичний результат	Статус
1	Реєстрація покупця	Створено обліковий запис, видано JWT-токен, перенаправлення в каталог	Відповідає	Пройдено
2	Реєстрація з існуючим email	Повідомлення про дублювання email	Відповідає	Пройдено
3	Вхід з правильним паролем	Успішна автентифікація, токен у localStorage	Відповідає	Пройдено
4	Вхід з помилковим паролем	Повідомлення про невірні дані	Відповідає	Пройдено
5	Перегляд каталогу	Завантаження товарів з пагінацією	Відповідає	Пройдено
6	Пошук за назвою товару	Відображення товарів з релевантними збігами	Відповідає	Пройдено
7	Фільтр за категорією	Показ товарів лише обраної категорії	Відповідає	Пройдено
8	Фільтр за ціновим діапазоном	Показ товарів у заданому діапазоні	Відповідає	Пройдено
9	Сортування за рейтингом	Сортування за спаданням рейтингу	Відповідає	Пройдено
10	Додавання до кошика	Товар у кошику, лічильник оновлений	Відповідає	Пройдено
11	Зміна кількості у кошику	Сума перерахована автоматично	Відповідає	Пройдено
12	Оформлення з валідними даними	Замовлення створено, номер видано, кошик очищено	Відповідає	Пройдено
13	Оформлення з порожнім кошиком	Повідомлення про порожній кошик	Відповідає	Пройдено
14	Видача цифрового	Запис у бібліотеці з	Відповідає	Пройдено

	товару	ключем доступу		
15	Повторне завантаження з бібліотеки	Файл завантажується успішно	Відповідає	Пройдено
16	Реєстрація продавця	Створено профіль продавця	Відповідає	Пройдено
17	Перегляд аналітики продавця	Дохід, продажі, графіки відображені	Відповідає	Пройдено
18	Доступ без токена	Перенаправлення на сторінку входу	Відповідає	Пройдено

За результатами функціонального тестування всі 18 тест-кейсів пройдено успішно. Система коректно обробляє як штатні сценарії використання, так і граничні випадки. Окремо протестовано стійкість мікросервісної архітектури до відмов: при штучному вимкненні Search Service пошук продовжує працювати у резервному режимі через прямий запит до Catalog Service з використанням MongoDB-індексу; при тимчасовій недоступності платіжного шлюзу замовлення зберігаються зі статусом payment_pending для повторної обробки після відновлення зв'язку.

Для тестування продуктивності використано інструмент k6, яким згенеровано навантаження 500 одночасних користувачів, що виконують типовий сценарій: вхід → пошук → перегляд товару → додавання в кошик → оформлення. Середній час відповіді API склав 145 мс, 95-й перцентиль — 320 мс, що є прийнятним для систем такого класу.

4.1 Вимоги до апаратного та програмного забезпечення

Оскільки система маркетплейсу побудована за клієнт-серверною архітектурою з розгортанням серверної частини у хмарі, вимоги до апаратного забезпечення розрізняються для двох категорій учасників: кінцевих користувачів (покупців та продавців) та операторів системи (адміністраторів, які розгортають та супроводжують серверну частину) [28].

Вимоги до апаратного забезпечення кінцевого користувача наведено у табл. 4.2.

Таблиця 4.2

Вимоги до апаратного забезпечення кінцевого користувача

Компонент	**Мінімальні вимоги**	**Рекомендовані вимоги**
Процесор	Двоядерний, 1.5 ГГц	Чотириядерний, 2.0 ГГц і вище
Оперативна пам'ять	4 ГБ	8 ГБ і більше
Місце на диску	500 МБ для браузера та кешу	1 ГБ і більше
Роздільна здатність екрана	1280×720	1920×1080 і вище
Інтернет-з'єднання	5 Мбіт/с	25 Мбіт/с і вище

Програмні вимоги до кінцевого користувача обмежуються наявністю сучасного веб-браузера з підтримкою JavaScript ES2020 та HTTP/2. Підтримувані браузери: Google Chrome версії 100 і вище, Mozilla Firefox 100+, Microsoft Edge 100+, Safari 15+, Opera 85+. Окремий клієнтський застосунок не потрібен — вся взаємодія здійснюється через веб-інтерфейс.

Вимоги до серверної інфраструктури мікросервісної системи маркетплейсу принципово відмінні від монолітних застосунків. Системі потрібен не один потужний сервер, а кластер з декількох вузлів, що працюють під керуванням Kubernetes. Мінімальний кластер для розгортання системи з демонстраційним навантаженням наведено у табл. 4.3.

Таблиця 4.3

Вимоги до серверної інфраструктури

Компонент	**Мінімальна конфігурація**	**Виробнича конфігурація**
Кількість вузлів	3 (1 master, 2 worker)	5+ (HA-master, autoscaling)

CPU на вузол	4 vCPU	8 vCPU і більше
RAM на вузол	8 ГБ	16 ГБ і більше
SSD-сховище	100 ГБ	500 ГБ і більше
Мережа	1 Гбіт/с	10 Гбіт/с
ОС вузлів	Ubuntu 22.04 LTS	Ubuntu 22.04 / 24.04 LTS

Програмні вимоги до серверної інфраструктури: операційна система Linux (рекомендовано Ubuntu 22.04 LTS); встановлений Docker Engine версії 25 і вище; кластер Kubernetes версії 1.29 і вище (можна використовувати готові керовані сервіси AWS EKS, Google GKE, Azure AKS або self-hosted інсталяцію через kubeadm); сервер баз даних PostgreSQL 16 (можна використовувати керований сервіс або розгорнути всередині кластера через офіційний оператор); сервер MongoDB 7 та Redis 7 (аналогічно); кластер Elasticsearch 8 (мінімум 3 вузли для HA-конфігурації); RabbitMQ 3.13 для брокера повідомлень.

4.2 Розгортання системи на платформі Kubernetes

Kubernetes (K8s) — це платформа з відкритим вихідним кодом для оркестрації контейнеризованих застосунків, що автоматизує розгортання, масштабування та управління контейнерами. Kubernetes було розроблено компанією Google на основі досвіду внутрішньої платформи Borg та зараз є галузевим стандартом для розгортання мікросервісних систем. Платформа надає декларативну модель управління інфраструктурою: адміністратор описує бажаний стан системи у YAML-файлах, а Kubernetes самостійно приводить фактичний стан до бажаного [26].

Архітектура розгортання системи маркетплейсу на Kubernetes складається з кількох логічних компонентів. Перший компонент — це Deployment-ресурси для кожного мікросервісу, що описують образ контейнера, кількість реплік, обмеження ресурсів та змінні середовища. Другий компонент — Service-ресурси, що надають стабільні мережеві ідентифікатори (DNS-імена) для кожного мікросервісу всередині кластера. Третій компонент — Ingress-

контролер, що приймає зовнішні HTTP-запити та маршрутизує їх до відповідного Service. Четвертий компонент — StatefulSet-ресурси для розгортання сховищ даних (PostgreSQL, MongoDB, Redis, Elasticsearch) з постійними томами. П'ятий компонент — ConfigMap та Secret для зберігання конфігураційних параметрів та чутливих даних (паролі, API-ключі).

Процес розгортання системи включає такі етапи. Перший етап — підготовка Docker-образів кожного мікросервісу. Для кожного сервісу створено Dockerfile, що описує процес збірки образу: вибір базового образу Node.js Alpine, копіювання вихідних файлів, встановлення залежностей, компіляція TypeScript, налаштування точки входу. Образи збираються та публікуються у Docker Registry (Docker Hub або приватний реєстр) під відповідними тегами версій.

Другий етап — створення Kubernetes-маніфестів для всіх компонентів системи. Маніфести зберігаються в Git-репозиторії як інфраструктура як код. Для кожного мікросервісу створюється окремий каталог з файлами deployment.yaml, service.yaml, configmap.yaml. Для сховищ даних використовуються готові Helm-чарти від офіційних виробників: bitnami/postgresql, bitnami/mongodb, bitnami/redis, elastic/elasticsearch.

Третій етап — застосування маніфестів до кластера за допомогою команди kubectl apply. Kubernetes аналізує бажаний стан системи та запускає необхідну кількість екземплярів кожного контейнера, виконує конфігурування мережі, монтує постійні томи. Процес повністю автоматизований та триває кілька хвилин.

Четвертий етап — налаштування автоматичного масштабування через HorizontalPodAutoscaler (HPA). Для кожного критичного мікросервісу (Catalog, Search, Order) задаються правила: при перевищенні середнього використання CPU понад 70% Kubernetes автоматично запускає додаткові репліки сервісу; при зниженні навантаження зайві репліки видаляються. Це забезпечує ефективне використання ресурсів кластера: у періоди низького навантаження

працює мінімальна кількість екземплярів, у пікові — кількість зростає до встановленого максимуму.

П'ятий етап — налаштування TLS-шифрування через Cert-Manager та автоматичне отримання сертифікатів Let's Encrypt. Вся взаємодія між клієнтом та системою відбувається виключно через HTTPS. Сертифікати автоматично оновлюються Cert-Manager у фоновому режимі за 30 днів до закінчення терміну дії.

Шостий етап — налаштування моніторингу та логування. Для збору метрик кожного мікросервісу використовується Prometheus, що автоматично сканує endpoints /metrics та зберігає часові ряди. Для візуалізації метрик розгортається Grafana з готовими дашбордами для Kubernetes, PostgreSQL, MongoDB. Для централізованого збору логів використовується стек ELK (Elasticsearch, Logstash, Kibana): кожен контейнер пише структуровані JSON-логи у stdout, Filebeat збирає їх та надсилає до Elasticsearch, Kibana надає веб-інтерфейс для пошуку та аналізу.

Діаграму розгортання системи на платформі Kubernetes наведено на рис. 4.1.

4.3 Склад інсталяційного пакету

Для розгортання системи розробником або адміністратором, що здійснює інсталяцію власного екземпляру маркетплейсу, підготовлено інсталяційний пакет, що містить вихідний код, конфігураційні файли, скрипти збірки та документацію. Склад інсталяційного пакету наведено у табл. 4.4.

Таблиця 4.4

Склад інсталяційного пакету системи маркетплейсу

Компонент	**Тип**	**Опис**
services/	Директорія	Вихідний код 8 мікросервісів, кожен у своїй піддиректорії
frontend/	Директорія	Вихідний код React-клієнта

gateway/	Директорія	Вихідний код API Gateway
shared/	Директорія	Спільний код: типи DTO, JWT-модуль, утиліти
k8s/	Директорія	Kubernetes-маніфести для всіх компонентів
docker-compose.yml	Конфігурація	Локальний запуск для розробки
Dockerfile (×9)	Конфігурація	Інструкції збірки Docker-образів
.env.example	Шаблон	Шаблон змінних середовища
package.json	Конфігурація	Залежності та скрипти збірки
README.md	Документація	Інструкції з розгортання та супроводу
docs/architecture.md	Документація	Опис архітектури та діаграми
.github/workflows/	Конфігурація	CI/CD-конвеєри GitHub Actions

Для розгортання локального екземпляру системи (для розробки та тестування) достатньо встановити Docker та Docker Compose, після чого виконати команду docker-compose up. Команда автоматично запускає всі мікросервіси, сховища даних та клієнтський застосунок у Docker-контейнерах на локальній машині. Уся система буде доступна за адресою localhost:3000.

Для розгортання у виробничому середовищі необхідно: підготувати кластер Kubernetes; створити namespace для системи; налаштувати ConfigMap та Secret з конфігураційними значеннями (паролі сховищ, API-ключі платіжного шлюзу, секретний ключ JWT); зібрати Docker-образи та опублікувати у Docker Registry; виконати kubectl apply -k k8s/overlays/production для розгортання всіх компонентів через Kustomize-overlay.

Аналіз продуктивності системи

Для оцінки продуктивності розробленої системи проведено серію навантажувальних випробувань. Випробування виконувалися на тестовому кластері Kubernetes з 3 вузлів (по 4 vCPU та 8 ГБ RAM кожен). Генератор

навантаження k6 розміщувався на окремій машині для виключення впливу на досліджувану систему.

Базовий тестовий сценарій імітує типову поведінку покупця: вхід до системи → пошук товару за ключовим словом → перегляд 3-5 карток товарів → додавання одного товару в кошик → оформлення замовлення → перегляд бібліотеки. Сценарій займає приблизно 90 секунд реального часу з паузами між діями, що імітують час прийняття рішення.

При навантаженні 100 одночасних користувачів система показала такі результати: середній час відповіді API — 87 мс, 95-й перцентиль — 215 мс, 99-й перцентиль — 410 мс. Частка успішних запитів — 100%, помилок не зафіксовано. Використання CPU на вузлах кластера — 25-35%, RAM — 40-50%. Це штатний режим роботи з достатнім запасом продуктивності.

При навантаженні 500 одночасних користувачів спрацювало автоматичне масштабування: HorizontalPodAutoscaler запустив додаткові репліки Catalog Service (з 2 до 5) та Search Service (з 2 до 6). Час відповіді на 95-му перцентилі зріс до 320 мс, що залишається в межах прийнятного. Частка помилок — менше 0.1% (одиничні тайм-аути при перерозподілі трафіку). Використання CPU — 55-70%.

При навантаженні 1000 одночасних користувачів спостерігалося подальше масштабування подів та активація Cluster Autoscaler — додано один новий вузол до кластера. Час відповіді на 95-му перцентилі стабілізувався на рівні 380 мс після завершення масштабування. Частка помилок — 0.3%, переважно за рахунок кількох секунд перехідного стану під час масштабування.

При спробі досягти 2000 одночасних користувачів стався перший суттєвий вплив — час відповіді на 99-му перцентилі зріс до 1.2 секунди, з'явилися помилки 503 Service Unavailable у пікові моменти. Аналіз показав, що вузьким місцем стало RabbitMQ — брокер повідомлень не встигав обробляти потік подій. Це визначило напрям подальшої оптимізації — перехід на кластерну конфігурацію RabbitMQ або заміна на більш продуктивний Apache Kafka.

Окремо досліджено стійкість системи до відмов. У ході випробувань штучно вимикалися окремі мікросервіси та сховища даних. При відмові Search Service пошук переключався в резервний режим через MongoDB-індекс із збільшенням часу відповіді до 700 мс, але без переривання обслуговування. При відмові одного вузла PostgreSQL (master) автоматичне переключення на replica зайняло 23 секунди, протягом яких операції запису не виконувалися; операції читання продовжували працювати. При штучному розриві мережі між частинами кластера (network partition) система продемонструвала очікувану поведінку: сервіси однієї частини продовжували працювати з обмеженою функціональністю, після відновлення зв'язку черги повідомлень синхронізувалися.

За результатами навантажувальних випробувань зроблено такі висновки. Розроблена архітектура витримує цільове навантаження для маркетплейсу масштабу середнього бізнесу (до 1000 одночасних користувачів, до 50000 запитів за хвилину) з прийнятним часом відповіді. Подальше масштабування потребує оптимізації брокера повідомлень та можливого впровадження read-replicas для PostgreSQL у критичних сервісах. Архітектурні рішення щодо мікросервісної декомпозиції та поліглот-персистентності виправдали себе — система демонструє стійкість до відмов окремих компонентів та здатність до автоматичного масштабування під навантаженням.

Моніторинг та спостережуваність

Спостережуваність (observability) — здатність системи надавати інформацію про свій внутрішній стан через зовнішні сигнали. У мікросервісній архітектурі спостережуваність є критично важливою, оскільки запит користувача може проходити через 5-10 мікросервісів, і у разі помилки необхідно швидко визначити, який саме сервіс став джерелом проблеми. Спостережуваність будується на трьох стовпах: метрики, логи, трасування [26].

Метрики — числові показники стану системи, що збираються через регулярні інтервали часу. Для маркетплейсу збираються такі категорії метрик.

Системні метрики: використання CPU, RAM, диска, мережі кожним контейнером. Метрики застосунку: кількість запитів за секунду, час відповіді (середній, 95-й та 99-й перцентилі), кількість помилок, розмір черг RabbitMQ. Бізнес-метрики: кількість нових реєстрацій, кількість замовлень, GMV (Gross Merchandise Value), середній чек, конверсія з каталогу в замовлення. Усі метрики збираються Prometheus та візуалізуються у Grafana.

Логи — текстові записи про події в системі. У мікросервісній архітектурі особливо важливо, щоб логи були структурованими (JSON-формат), містили контекстну інформацію (ідентифікатор запиту, ідентифікатор користувача, назву сервісу) та збиралися централізовано. У розробленій системі кожен мікросервіс використовує бібліотеку pino для логування у форматі JSON. Filebeat зчитує stdout контейнерів та надсилає у Elasticsearch. Kibana надає веб-інтерфейс для пошуку та фільтрації логів.

Трасування (distributed tracing) — відстеження шляху одного запиту через усі сервіси системи. Кожному вхідному HTTP-запиту присвоюється унікальний trace-id, який передається у заголовках усіх внутрішніх викликів. Це дозволяє побудувати «дерево» викликів для будь-якого запиту та точно визначити, скільки часу зайняла кожна стадія. У розробленій системі трасування реалізоване через OpenTelemetry SDK з експортом даних до Jaeger.

Окрім пасивного моніторингу, налаштовано систему алертів на критичні події. При перевищенні порогу 5% помилок 5xx протягом 5 хвилин система автоматично надсилає сповіщення черговому інженеру через Slack та PagerDuty. Інші правила алертів: затримка відповіді понад 1 секунду на 99-му перцентилі, переповнення черг RabbitMQ, недоступність сховища даних, наближення вільного місця на диску до 80%.

Стратегія масштабування

Однією з ключових переваг мікросервісної архітектури є можливість незалежного горизонтального масштабування кожного сервісу. Це особливо важливо для маркетплейсу, де навантаження на різні сервіси розподіляється

нерівномірно. Аналіз патернів трафіку показує, що Search Service та Catalog Service обробляють у середньому в 50-100 разів більше запитів, ніж Order Service. Без можливості масштабування цих сервісів окремо довелося б створювати надлишкові ресурси на «непотрібні» компоненти.

Стратегія масштабування системи передбачає декілька рівнів. Базовий рівень — автоматичне масштабування подів через HorizontalPodAutoscaler. Для кожного критичного сервісу налаштовано правила: мінімальна кількість реплік 2 (для відмовостійкості), максимальна — до 20 (для пікових навантажень). Цільове використання CPU 70%: при перевищенні запускаються додаткові репліки, при зниженні — зайві видаляються.

Другий рівень — масштабування самого кластера через Cluster Autoscaler. Якщо подам не вистачає ресурсів для розміщення на існуючих вузлах, Cluster Autoscaler автоматично запитує у хмарного провайдера додаткові вузли. При зменшенні навантаження зайві вузли виводяться з кластера для економії коштів.

Третій рівень — масштабування сховищ даних. PostgreSQL масштабується через додавання read-replica: операції читання розподіляються між кількома репліками, тоді як запис йде на master. MongoDB підтримує sharding — горизонтальне розділення колекцій між кількома фізичними вузлами. Elasticsearch природно розподіляється: індекси розбиваються на shards, які розміщуються на різних вузлах кластера.

Окрім автоматичного масштабування, передбачено стратегію планового масштабування для очікуваних піків навантаження. Перед великими розпродажами (Black Friday, новорічні свята) команда DevOps завчасно збільшує мінімальну кількість реплік критичних сервісів та додає вузли в кластер. Після завершення піку конфігурація повертається до базової.

Експериментальне дослідження масштабування проведено через генерацію поступово зростаючого навантаження від 50 до 2000 одночасних користувачів. Система продемонструвала очікувану поведінку: початково HPA запустив додаткові репліки Catalog та Search; при подальшому зростанні Cluster

Autoscaler додав 2 нові вузли. Загальний час відповіді залишився стабільним протягом усього експерименту в межах 200-350 мс на 95-му перцентилі. Це підтверджує практичну ефективність обраної архітектури масштабування.

Висновки до розділу 4

У четвертому розділі описано впровадження та експлуатацію системи маркетплейсу. Описано трирівневе тестування системи: модульне (Jest, 78% покриття), інтеграційне (з реальними СУБД у контейнерах через testcontainers) та функціональне (18 користувацьких сценаріїв). Усі 18 функціональних тест-кейсів пройдено успішно. Окремо протестовано стійкість системи до відмов окремих сервісів та продуктивність під навантаженням 500 одночасних користувачів.

Визначено вимоги до апаратного та програмного забезпечення кінцевого користувача (мінімальні — типовий сучасний пристрій з браузером Chrome 100+) та серверної інфраструктури (кластер Kubernetes з 3 вузлів мінімальної конфігурації). Описано детальний процес розгортання системи на платформі Kubernetes з шістьма послідовними етапами: збірка Docker-образів, створення маніфестів, застосування до кластера, налаштування автомасштабування, TLS-шифрування, моніторингу та логування.

Подано склад інсталяційного пакету з 12 компонентів, що охоплюють вихідний код, конфігурацію Docker та Kubernetes, документацію та CI/CD-конвеєри. Описано спрощений процес локального розгортання через Docker Compose для розробки та повний процес виробничого розгортання у Kubernetes.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі розв'язано актуальну прикладну задачу — створено програмне забезпечення електронного маркетплейсу з мікросервісною архітектурою. Робота охоплює повний цикл — від каталогізації товарів продавцем до отримання покупки покупцем — і відповідає всім сформульованим у вступі завданням.

1. Здійснено опис предметної області з урахуванням її специфіки: розглянуто сутність та класифікацію електронних маркетплейсів, виокремлено чотири основні класи (фізичних товарів, послуг, цифрових продуктів, універсальні), визначено спільний набір функціональних можливостей та технічні особливості, що зумовлюють використання мікросервісного підходу.

2. Розроблено модель предметної області електронного маркетплейсу: побудовано комплекс UML-діаграм, який охоплює діаграму прецедентів із трьома акторами, діаграму послідовності оформлення замовлення, діаграму активності бізнес-процесу купівлі та сукупність абстракцій з вісьмома ключовими сутностями.

3. Проведено огляд існуючих рішень у сфері електронних маркетплейсів: розглянуто Amazon, eBay, Rozetka, OLX, Etsy серед операційних платформ та Magento, Saleor, Medusa серед open-source-рішень. За результатами порівняння за вісьмома критеріями обґрунтовано доцільність розробки власної системи.

4. Обґрунтовано постановку задачі: сформульовано 16 функціональних вимог до системи, визначено межі розв'язуваної задачі, обрано стек технологій та цільову модель розгортання.

5. Побудовано узагальнену логічну модель даних системи: за принципом «база даних на сервіс» вона розпадається на вісім автономних доменів, по одному на кожен мікросервіс. Зв'язки між сутностями різних доменів реалізовано через ідентифікатори на рівні бізнес-логіки.

6. Аргументовано вибір СУБД як ключового елементу архітектурних рішень мікросервісної системи: обрано підхід поліглот-персистентності, що

поєднує чотири різні сховища даних — PostgreSQL для транзакційних даних, MongoDB для каталогу з гнучкою схемою, Redis для кошика та кешу, Elasticsearch для пошуку.

7. Пояснено принципи побудови фізичної структури бази даних: визначено типи даних, обмеження цілісності, індекси та політики каскадного видалення. Розкрито принципи нормалізації в межах сервісу та контрольованої денормалізації між сервісами.

8. З'ясовано архітектурні особливості системи програмного забезпечення електронного маркетплейсу: побудовано триярусну архітектуру (клієнт — API Gateway — мікросервіси), реалізовано двохрежимну модель взаємодії (синхронні REST-запити та асинхронні події через RabbitMQ), визначено зони відповідальності восьми мікросервісів.

9. Пояснено вибір методичного інструментарію розробки: обґрунтовано стек з 16 інструментів, що охоплює Node.js, TypeScript, Express.js, React, PostgreSQL, MongoDB, Redis, Elasticsearch, RabbitMQ, Docker, Kubernetes та інші. Кожен інструмент обраний за критеріями зрілості, ефективності та узгодженості з мікросервісною архітектурою.

10. Визначено послідовний алгоритм оформлення замовлення та ключові модулі системи: реалізовано Saga-патерн з компенсуючими діями, який охоплює резервування товару, списання коштів, створення запису у бібліотеці та публікацію події OrderPaid у RabbitMQ. Описано модулі search-engine, cart-service, jwt-auth.

11. Розроблено інтерфейс користувача MerchFlow: створено клієнтський веб-застосунок на React, що містить головну сторінку з каталогом, картки товарів, кошик, форму оформлення замовлення, особистий кабінет покупця з бібліотекою цифрових продуктів та кабінет продавця з аналітикою продажів.

12. Проведено функціональне тестування системи та підготовку до розгортання у хмарі: виконано тривіл'неве тестування — модульне (Jest, 78 % покриття), інтеграційне (testcontainers) та функціональне (18 тест-кейсів). Усі тест-кейси пройдено успішно; додатково перевірено стійкість до відмов

окремих сервісів та продуктивність під навантаженням 500 одночасних користувачів.

13. Розроблено склад інсталяційного пакету для встановлення програмного продукту: підготовлено вихідний код восьми мікросервісів, клієнта та API Gateway, Dockerfile для кожного сервісу, Kubernetes-маніфести, конфігурацію змінних середовища, CI/CD-конвеєри GitHub Actions та документацію з розгортання.

14. Обґрунтовано вимоги до апаратного та програмного забезпечення кінцевого користувача й серверної інфраструктури: для клієнта — типовий сучасний пристрій з браузером Chrome 100+; для серверної частини — Kubernetes-кластер мінімум з трьох вузлів з керованими сервісами PostgreSQL, MongoDB, Redis, Elasticsearch.

15. Описано процес розгортання системи на платформі Kubernetes: визначено шість послідовних етапів — збірка Docker-образів, створення Kubernetes-маніфестів, застосування до кластера, налаштування HorizontalPodAutoscaler, TLS-шифрування через Cert-Manager, моніторинг через Prometheus та Grafana з логуванням у ELK-стек.

16. Розглянуто стратегію спостережуваності, безпеки та масштабування системи: впроваджено механізми централізованого логування, метрик і трасування; реалізовано глибокий захист на п'яти рівнях; впроваджено стратегію автоматичного масштабування подів і вузлів кластера, що підтверджена результатами навантажувальних випробувань.

Практична цінність роботи полягає в тому, що отримана система — це повноцінна реалізація мікросервісної архітектури для e-commerce з усіма її ознаками: автономні сервіси, поліглот-персистентність, асинхронна взаємодія через події, контейнери та оркестрація. Запропонований підхід можна використовувати як референсну архітектуру для побудови маркетплейсів промислового масштабу або як готове SaaS-рішення для запуску реального бізнесу.

Подальший розвиток системи бачимо у кількох напрямах: розширити рекомендаційний механізм за рахунок аналізу поведінки покупців та алгоритмів машинного навчання; додати повноцінний платіжний процесинг з кількома провайдерами та обробкою повернень; розробити нативні мобільні застосунки для iOS та Android; додати багатомовність і мультивалютність; запустити програму лояльності з накопиченням бонусів; інтегруватися з зовнішніми бухгалтерськими системами для автоматизації фінансової звітності продавців.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Parker G. Platform Revolution: How Networked Markets Are Transforming the Economy / G. Parker, M. Van Alstyne, S. Choudary. — New York : W.W. Norton & Company, 2016. — 352 p.
2. Evans D.S. Matchmakers: The New Economics of Multisided Platforms / D.S. Evans, R. Schmalensee. — Boston : Harvard Business Review Press, 2016. — 272 p.
3. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. — 2nd ed. — Sebastopol : O'Reilly Media, 2021. — 612 p.
4. Fowler M. Microservices [Електронний ресурс] / M. Fowler, J. Lewis. — Режим доступу: https://martinfowler.com/articles/microservices.html
5. Richardson C. Microservices Patterns: With Examples in Java / C. Richardson. — Shelter Island : Manning Publications, 2018. — 520 p.
6. Буч Г. Мова моделювання UML. Посібник користувача / Г. Буч, Дж. Рамбо, А. Джекобсон. 2006. — 432 с.
7. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans. — Boston : Addison-Wesley, 2003. — 560 p.
8. Amazon Web Services. Microservices on AWS [Електронний ресурс]. — Режим доступу: https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/
9. eBay Engineering Blog [Електронний ресурс]. — Режим доступу: https://innovation.ebayinc.com/tech/
10. Rozetka — інтернет-маркетплейс [Електронний ресурс]. — Режим доступу: https://rozetka.com.ua
11. OLX — оголошення України [Електронний ресурс]. — Режим доступу: https://www.olx.ua
12. Etsy Engineering Blog [Електронний ресурс]. — Режим доступу: https://www.etsy.com/codeascraft

13. Medusa — Open Source Composable Commerce [Електронний ресурс]. — Режим доступу: https://medusajs.com

14. Hohpe G. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / G. Hohpe, B. Woolf. — Boston : Addison-Wesley, 2003. — 736 p.

15. Richardson C. Pattern: Database per Service [Електронний ресурс]. — Режим доступу: https://microservices.io/patterns/data/database-per-service.html

16. Vernon V. Implementing Domain-Driven Design / V. Vernon. — Boston : Addison-Wesley, 2013. — 656 p.

17. Vogels W. Eventually Consistent / W. Vogels // Communications of the ACM. — 2009. — Vol. 52, № 1. — P. 40-44.

18. Sadalage P.J. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence / P.J. Sadalage, M. Fowler. — Boston : Addison-Wesley, 2012. — 192 p.

19. Provos N. A Future-Adaptable Password Scheme / N. Provos, D. Mazieres // USENIX Annual Technical Conference. — 1999. — P. 81-91.

20. MongoDB Documentation. Indexes [Електронний ресурс]. — Режим доступу: https://www.mongodb.com/docs/manual/indexes/

21. Richardson C. Pattern: API Gateway / Backends for Frontends [Електронний ресурс]. — Режим доступу: https://microservices.io/patterns/apigateway.html

22. Express.js Documentation. Routing [Електронний ресурс]. — Режим доступу: https://expressjs.com/en/guide/routing.html

23. RabbitMQ Documentation. Publish/Subscribe [Електронний ресурс]. — Режим доступу: https://www.rabbitmq.com/tutorials/tutorial-three-javascript.html

24. Node.js Documentation. Event Loop [Електронний ресурс]. — Режим доступу: https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick

25. Docker Documentation. Get Started [Електронний ресурс]. — Режим доступу: https://docs.docker.com/get-started/

26. Burns B. Kubernetes: Up and Running / B. Burns, J. Beda, K. Hightower. — 3rd ed. — Sebastopol : O'Reilly Media, 2022. — 320 p.

27. Myers G.J. The Art of Software Testing / G.J. Myers, C. Sandler, T. Badgett. — 3rd ed. — Hoboken : Wiley, 2011. — 240 p.

28. MDN Web Docs. Progressive Web Apps [Електронний ресурс]. — Режим доступу: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps

29. PostgreSQL Documentation. The SQL Language [Електронний ресурс]. — Режим доступу: https://www.postgresql.org/docs/16/index.html

30. Elasticsearch Documentation. Getting Started [Електронний ресурс]. — Режим доступу: https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started.html

31. React Documentation. Learn React [Електронний ресурс]. — Режим доступу: https://react.dev/learn

32. Cloud Native Computing Foundation. CNCF Cloud Native Definition [Електронний ресурс]. — Режим доступу: https://github.com/cncf/toc/blob/main/DEFINITION.md

33. Гліненко Л.К., Дайновський Ю.А. Стан і перспективи розвитку електронної торгівлі України. Маркетинг і менеджмент інновацій. 2018. № 1. С. 83–102.

34. Кузьо Н.Є., Косар Н.С., Білик І.І. Електронна торгівля в Україні: тенденції та перспективи розвитку. Вісник Харківського національного університету імені В.Н. Каразіна. Серія «Економічна». 2020. Вип. 99. С. 71–79.

35. Куклінова Т.В. Сучасні тенденції та фактори Інтернет-торгівлі в Україні. Вісник соціально-економічних досліджень. 2018. № 1 (65). С. 95–102.

36. Барська Ю.Г., Шпотенко В.Д. Електронна комерція в Україні: сучасний стан, галузеві особливості та організаційні форми. Маркетинг і цифрові технології. 2020. Т. 4, № 4. С. 6–20.

37. Подзігун С.М., Бержанір І.А. Електронна торгівля в Україні: стан, тенденції, перспективи розвитку. Маркетинг і цифрові технології. 2020. Т. 4, № 3. С. 88–101.

38. Мікросервісна архітектура: переваги та недоліки її практичного застосування / А.П. Артюх, О.С. Стирта. Information Technology: Computer Science, Software Engineering and Cyber Security. 2024. № 2. С. 19–27.

39. Дослідження мікросервісної архітектури, архітектурний стиль REST та їх сучасна реалізація на Java / А.І. Дрозд, С.Г. Антощук. Вісник Хмельницького національного університету. 2021. № 1 (293). С. 132–137.

40. Впровадження мікросервісної архітектури: технічні виклики та рішення / В.О. Браницький, Р.М. Балабан. Вісник Херсонського національного технічного університету. 2024. № 4 (91). С. 134–142.

41. Проєктування та розробка мікросервісної архітектури системи бронювання / О.І. Боровик, С.С. Дзюба. Молодий вчений. 2020. № 12 (88). С. 230–235.

42. Мікросервісна архітектура для кіберфізичних систем / І.С. Скарга-Бандуров, О.Ю. Бабиченко. Вісник Херсонського національного технічного університету. 2024. № 1 (88). С. 41–48.

43. Оптимізація електронної комерції за допомогою мікросервісної архітектури / М.І. Палій. Current Trends in Scientific Research Development : Proceedings of VI International Scientific and Practical Conference (Boston, USA, 16–18 January 2025). Boston, 2025. С. 276–284.

44. Литвинов В.В., Хом'як А.Б. Контейнеризація розподілених застосунків засобами Docker та Kubernetes. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. 2022. № 47. С. 65–73.

45. Дудін О.Ю., Кравченко О.В. Архітектурні шаблони побудови REST API в мікросервісних системах. Радіоелектронні і комп'ютерні системи. 2023. № 3 (107). С. 108–117.

ДОДАТКИ

Декларація про академічну доброчесність та використання ШІ

ДЕКЛАРАЦІЯ ЗДОБУВАЧА

Я, Андрущенко Олександр Олександрович, здобувач(ка) НУБіП України, усвідомлюючи відповідальність за порушення академічної доброчесності, цією заявою підтверджую, що:

Моя бакалаврська кваліфікаційна робота (проєкт) на тему «ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ СИСТЕМИ
З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ
ДЛЯ МАРКЕТПЛЕЙСІВ» є результатом моєї особистої праці.

1. Усі запозичення з текстів інших авторів мають відповідні посилання згідно з чинними стандартами.
2. Щодо використання інструментів ШІ зазначаю, що (обрати необхідне):
    - [ ] інструменти генеративного ШІ не використовувалися.
    - [ ] інструменти ШІ (вказати назву: ChatGPT, Gemini, Claude, DeepL, _____інші (вказати назву)) були використані для:
        - [Так] перекладу іншомовних джерел;
        - [Так] стилістичного редагування та перевірки граматики;
        - [Так] допомоги у написанні/відлагодженні програмного коду;
        - [ ] інше: _____.

Я розумію, що надання недостовірної інформації може призвести до скасування результатів захисту та відрахування з числа здобувачів НУБіП України.

«___» _____ 2026 р. _____ **Андрущенко О.О.**
(підпис)