Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

МІКРОСЕРВІСНА АРХІТЕКТУРА СИСТЕМИ ДИНАМІЧНОГО ЦІНОУТВОРЕННЯ ДЛЯ E-COMMERCE ПЛАТФОРМ З ВИКОРИСТАННЯМ ХМАРНИХ ТЕХНОЛОГІЙ ТА МАШИННОГО НАВЧАННЯ

Виконав студент 2 курсу, групи 2КІ-24м
спеціальності 123 — Комп'ютерна інженерія
_____ Сеник Ю. О.
Керівник к.т.н., проф. каф. ОТ
_____ Захарченко С. М.
"12" грудня 2025 р.
Опонент доктор філософії, доц. каф. МБІС
_____ Салієва О. В.
"12" грудня 2025 р.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.
_____
"15" 12 2025 р.

ВНТУ 2025

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Галузь знань — Інформаційні технології

Освітній рівень — магістр

Спеціальність — 123 Комп'ютерна інженерія

Освітньо-професійна програма — Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри обчислювальної техніки

_____ О. Д. Азаров

"25" вересня 2025 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

Студенту Сенику Юрію Олександровичу

1 Тема роботи "Мікросервісна архітектура системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання" керівник роботи Захарченко С. М. к.т.н., проф. каф. ОТ, затверджено наказом вищого навчального закладу від 24.09.2025 року № 313.

2 Строк подання студентом роботи 04.12.2025 р.

3 Результати роботи: охоплюють методи динамічного ціноутворення, що спираються на сучасні алгоритми машинного навчання та аналіз ринкових даних. Для реалізації використовується мікросервісна архітектура з контейнеризацією Docker, хмарні сервіси для масштабування та зберігання інформації, а також API для інтеграції з e-commerce платформами (була використана WooCommerce). Програмна реалізація базується на мові Python з застосуванням фреймворку FastAPI для створення мікро сервісів та бібліотек

машинного навчання scikit-learn. Для розробки та тестування використовується

IDE PyCharm.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які слід розглянути): вступ, огляд сучасних підходів до динамічного ціноутворення в e-commerce, дослідження методів машинного навчання для оптимізації цін, проектування мікро сервісної архітектури системи ціноутворення, розробка алгоритмів збору та аналізу конкурентних даних, впровадження рішення з використанням хмарних технологій, експериментальні дослідження та тестування, економічне обґрунтування ефективності системи, висновки.

5. Перелік графічного матеріалу: архітектура ML-системи для динамічного ціноутворення, діаграми класів та послідовності, діаграма процесу збору та обробки конкурентних даних, схеми для захисту від каскадних відмов. Перелік ілюстративного матеріалу: скріншоти інтерфейсу системи управління та аналітики.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Захарченко С. М., к.т.н., проф. каф. ОТ		
5	Адлер О. О., к.т.н., доц., каф. ЕПВМ		
Нормоконтроль	Швець С. І., асист. каф. ОТ		

7 Дата видачі завдання 25.09.2025р.

8 Календарний план виконання МКР приведений в таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів МКР	Строк виконання	Примітка
1	Постановка задачі роботи	08.09.2025	виконано
2	Аналіз методів динамічного ціноутворення, алгоритмів машинного навчання та технологій e-commerce	10.09.2025- 15.09.2025	виконано
3	Дослідження архітектурних підходів до побудови систем реального часу та методів аналізу поведінкових даних	16.09.2025- 19.09.2025	виконано
4	Проектування та розробка системи динамічного ціноутворення	20.09.2025- 23.09.2025	виконано
5	Реалізація системи, проведення експериментальних досліджень та оцінка ефективності	24.09.2025- 12.10.2025	виконано
6	Розрахунок економічної частини	16.10.2025	виконано
7	Оформлення матеріалів до захисту МКР	17.10.2025	виконано
8	Перевірка якості виконання магістерської роботи та усунення недоліків	28.10.2025	виконано
9	Підписи супроводжувальних документів у нормоконтролера, керівника, опонента	04.11.2025	виконано
10	Перевірка на антиплагіат та ШІ	07.11.2025	виконано
11	Попередній захист роботи	12.11.2025	виконано

Студент _____ Сеник Ю. О.

Керівник _____ к.т.н., проф. каф. ОТ Захарченко С. М.

АНОТАЦІЯ

УДК 004.4

Сеник Ю.О. Мікросервісна архітектура системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання. Магістерська кваліфікаційна робота зі спеціальності 123 — Комп'ютерна інженерія, Вінниця: ВНТУ, 2025. — 136 с. На укр. мові. Бібліогр.: 24 назв; рис.: 34; табл.: 8.

У даній роботі розглянуто створення мікро сервісної архітектури для систем динамічного ціноутворення в електронній торгівлі. Виконано аналіз актуальних підходів до автоматизації ціноутворення та методів машинного навчання для оптимізації цін. Запропоновано структуру системи, яка забезпечує збір та аналіз конкурентних даних, прогнозування оптимальних тарифів та інтеграцію з популярними комерційними інтернет платформами. Впроваджено алгоритми машинного навчання для автоматичного коригування цін у режимі реального часу з урахуванням ринкових факторів.

Ключові слова: мікросервісна архітектура, динамічне ціноутворення, машинне навчання, цифрова комерція, хмарні технології, API-інтеграція.

ABSTRACT

УДК 004.4

Senyk Y.O. Microservice architecture of a dynamic pricing system for digital commerce using cloud technologies and machine learning. Master's thesis in Computer Engineering, Vinnytsia: VNTU, 2025. — 136 p. In Ukrainian language. Bibliographer: 24 titles; fig.: 34; tabl.: 8.

This paper considers the creation of a microservice architecture for dynamic pricing systems in e-commerce. An analysis of current approaches to pricing automation and machine learning methods for price optimization is performed. A system structure is proposed that provides for the collection and analysis of competitive data, forecasting of optimal tariffs, and integration with popular commercial Internet platforms. Machine learning algorithms have been implemented for automatic price adjustment in real time, taking into account market factors.

Keywords: microservice architecture, dynamic pricing, machine learning, digital commerce, cloud technologies, API integration.

ЗМІСТ

ВСТУП

Актуальність теми дослідження полягає в тому, що сучасний ринок електронної комерції відрізняється високою динамічністю та інтенсивною конкуренцією, що змушує підприємства оперативно реагувати на зміни ринкових умов через оптимізацію цінової стратегії. Традиційні підходи до ціноутворення, засновані на статичному аналізі, не забезпечують конкурентоспроможність у середовищі постійних коливань попиту, цін конкурентів та інших ринкових факторів. Впровадження систем динамічного ціноутворення з використанням технологій машинного навчання дозволяє автоматизувати процес прийняття рішень щодо цін та підвищити прибутковість e-commerce платформ. Мікросервісна архітектура гарантує масштабованість, надійність і гнучкість таких рішень, а хмарні технології надають необхідні обчислювальні ресурси для обробки великих обсягів даних у режимі реального часу. Особливу актуальність це має для українського ринку електронної торгівлі, який швидко розвивається і потребує сучасних технологічних рішень для підвищення конкурентоспроможності.

Мета роботи є розробка і дослідження мікро сервісної архітектури системи динамічного ціноутворення для e-commerce платформ із застосуванням хмарних технологій та алгоритмів машинного навчання, що забезпечує підвищення ефективності цінової політики та прибутковості онлайн-торгівлі.

Для досягнення поставленої мети необхідно виконати такі завдання:

— проаналізувати сучасні підходи до динамічного ціноутворення в електронній комерції та існуючі архітектурні рішення для створення розподілених систем;

— вивчити методи машинного навчання, що дозволяють прогнозувати оптимальні ціни на основі ринкових факторів і поведінки споживачів;

— розробити мікросервісну архітектуру системи динамічного ціноутворення з урахуванням принципів масштабованості, надійності та продуктивності;

— спроектувати та реалізувати систему збору, обробки і аналізу даних про ціни конкурентів, попит та інші ринкові показники;

— створити алгоритми машинного навчання для автоматичного коригування цін у режимі реального часу;

— розробити API для інтеграції з популярними e-commerce платформами (у нашому випадку буде WooCommerce);

— провести експериментальне дослідження ефективності розробленої системи і порівняти її з існуючими рішеннями.

Об’єктом дослідження є процеси динамічного ціноутворення в системах електронної комерції з використанням технологій машинного навчання та хмарних обчислень.

Предметом дослідження є методи та інструменти створення мікро сервісної архітектури для систем динамічного ціноутворення, алгоритми машинного навчання для оптимізації цін і технології інтеграції з e-commerce платформами.

Новизна погляє в тому що:

— удосконалено підхід до створення мікро сервісної архітектури системи ціноутворення шляхом розробки спеціалізованого API-gateway з вбудованою системою кешування та балансування навантаження, що підвищує продуктивність обробки запитів і забезпечує масштабованість;

— розроблено методику автоматизованого збору даних за допомогою алгоритмів розпізнавання поведінки користувачів, що підвищує точність порівняльного аналізу цін і скорочує час обробки ринкової інформації;

— запропоновано комбінований підхід до прогнозування цін, який поєднує статистичні методи (лінійна регресія, часові ряди) з елементами машинного навчання для врахування сезонності та трендів попиту, що значно підвищує точність рекомендацій щодо ціноутворення у порівнянні зі статичними традиційними методами.

Практичне значення результатів полягає в тому, що розроблена система може бути впроваджена на e-commerce платформах для автоматизації процесу

ціноутворення і підвищення прибутковості. Мікросервісна архітектура забезпечує просту інтеграцію з існуючими системами управління товарами, а API дозволяє швидко підключати нові e-commerce рішення. Використання хмарних технологій зменшує витрати на інфраструктуру і забезпечує масштабованість продукту. Отримані результати можуть слугувати основою для створення комерційних продуктів у сфері pricing intelligence та оптимізації e-commerce операцій.

Апробація результатів роботи здійснена в доповіді на конференції "Молодь в науці: дослідження, проблеми, перспективи (МН — 2026)".

Матеріали роботи доповідались та опубліковувались [24]:

https://conferences.vntu.edu.ua/index.php/mn/mn2026/paper/view/26052

С. М. Захарченко Ю. О. Сеник ПРОБЛЕМИ МАСШТАБУВАННЯ МОНОЛІТНИХ СИСТЕМ ЦІНОУТВОРЕННЯ В E-COMMERCE ТА ЇХ ВИРІШЕННЯ ЧЕРЕЗ МІКРОСЕРВІСНУ АРХІТЕКТУРУ

1 АРХІТЕКТУРНІ ПІДХОДИ ДО ПОБУДОВИ СИСТЕМИ ДИНАМІЧНОГО ЦІНОУТВОРЕННЯ

1.1 Аналіз архітектурних патернів для інтернет магазинів

Монолітна архітектура в сучасних системах електронної комерції створює ряд проблем для розвитку бізнесу. Основні обмеження пов'язані з масштабованістю та відмовостійкістю, які впливають на показники конверсії та залученості користувачів, хоча точний ступінь впливу залежить від специфіки конкретної системи.

Центральна проблема полягає у щільному зв'язку компонентів — відмова навіть одного елемента може викликати каскадний ефект по всій системі. Мікросервісна архітектура пропонує інший підхід: кожен сервіс працює ізольовано і відповідає за свою частину роботи. Якщо один компонент відмовляє, інші продовжують функціонувати — система все ще може приймати замовлення або обробляти дані.

1.1.1 Еволюція від монолітних до мікро серверних архітектур

Якщо подивитись на розвиток інтернет магазинів, то можна побачити суттєві зміни в архітектурних підходах. Монолітна архітектура довгий час була основним варіантом — її обирали через простоту. Розробити і запустити таку систему було відносно легко. Проте з часом стало зрозуміло, що при зростанні системи виникають складнощі з продуктивністю і масштабуванням.

Суть моноліту в тому, що він збирає все в один великий модуль: інтерфейс, бізнес-логіку, обробку замовлень, базу даних, кешування. На початку це зручно — швидко розробити, протестувати, розгорнути. Але коли справа доходить до масштабування, починаються реальні проблеми. Вони стають очевидними тільки після того, як система виросла до певного розміру, а на ранніх етапах це важко передбачити.

Ранні інтернет маназини на платформах Magento 1.x, WooCommerce, OpenCart, PrestaShop вони всі були монолітами. Це давало можливість швидко

запустити магазин з базовими функціями: каталог, кошик, оплата. З часом виявились серйозні обмеження. Особливо для великого бізнесу — мільйони товарів, складні бізнес-процеси, інтеграції з різними системами типу ERP або CRM. Виправити це без повного переписування коду майже неможливо, що створює проблему для компаній які вже інвестували в розвиток.

Мікросервісна архітектура — це зовсім інша філософія (рис. 1.1). Замість одного великого додатку маємо багато маленьких незалежних сервісів. Кожен має свою функцію, свою базу даних, свій цикл розробки. Головне — можна розробляти і масштабувати їх окремо один від одного. Правда, на практиці перехід не такий простий як здається в теорії, особливо якщо команда звикла до монолітного підходу. Потрібно переосмислити не тільки архітектуру але й процеси розробки.

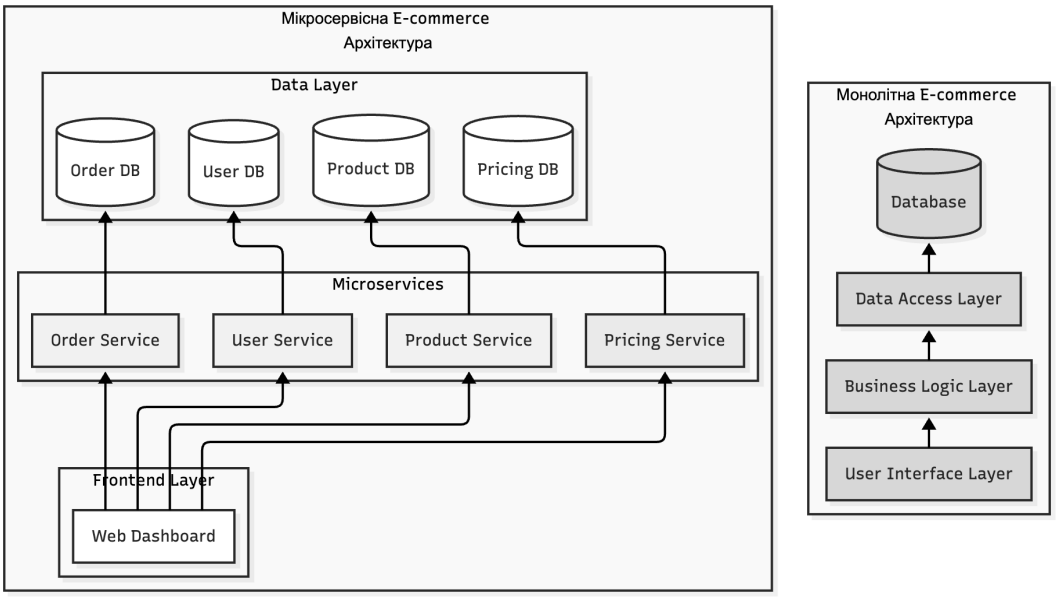

Рисунок 1.1 — Еволюція від монолітної до мікро сервісної архітектури

1.1.2 Патерни інтеграції та взаємодії

Синхронна комунікація через HTTP/REST API або GraphQL — це те що використовують найчастіше [1]. Просто тому що легко реалізувати, є купа інструментів, стандартні HTTP клієнти працюють з коробки. Але проблема в тому що при високому навантаженні починаються каскадні відмови, латентність

зростає. Вибір патернів взаємодії між сервісами тут реально важливий.

Через буферизацію запитів архітектура стає гнучкішою, адаптується до змін навантаження [3]. Асинхронна комунікація через черги (Apache Kafka, RabbitMQ, Amazon SQS) та pub/sub дає кращу розв'язку між сервісами. Для багатьох команд це виявилося важким завданням. Складнощі є і треба продумувати логіку обробки помилок, управляти дублікатами повідомлень, моніторити черги. Команди без досвіду з асинхронними системами часто тут застряють.

Event Sourcing, CQRS та Saga патерни зараз популярні в інтернет комерції [6]. Особливо де потрібен детальний аудит — системи ціноутворення наприклад, там кожна зміна ціни має бути задокументована для регуляторів. Можна відстежувати зміни, відновлювати стан системи на будь-який момент. Правда архітектурна складність зростає суттєво. Досвід показує що не для всіх додатків це виправдано.

1.2 Мікросервісна архітектура: принципи та переваги

Для розуміння сучасних підходів треба розібратись у мікросервісах. Принципи тут радикально відрізняються від монолітів. Це не просто технічні рекомендації — це методологія яка впливає на весь життєвий цикл продукту [6]. Від планування до організації команд, розробки, тестування, експлуатації. При проектуванні складних розподілених систем це особливо важливо розуміти, бо кожен компонент критичний для бізнесу.

1.2.1 Основні принципи мікро сервісної архітектури

Щоб успішно побудувати мікросервісну архітектуру, є декілька фундаментальних принципів. Single Responsibility Principle — кожен мікросервіс має чітку область відповідальності, виконує одну бізнес-функцію або обслуговує один bounded context (якщо говорити в термінах Domain-Driven Design). Команди можуть зосередитися на конкретній бізнес-логіці, якість коду краща, тестувати простіше [4].

Потрібне ретельне планування меж сервісів, правильна декомпозиція бізнес-логіки. Тут є підводні камені. Треба уникати і надмірної фрагментації (дрібні сервіси) і створення великих сервісів які по суті розподілені моноліти. На практиці це складно, особливо коли команда не має досвіду з Domain-Driven Design [6].

Децентралізоване управління даними — кожен мікросервіс має свою БД і повний контроль над даними [1]. Це дає незалежність розробки та розгортання. Але проблеми з цілісністю даних між сервісами виникають обов'язково. Особливо в трансакційних операціях де потрібна атомарність на рівні всієї системи.

Таблиця 1.1 — Порівняння принципів управління даними

Аспект	Централізована БД	Database per Service
Консистентність	ACID транзакції	Eventual Consistency
Масштабованість	Вертикальне масштабування	Горизонтальне масштабування
Незалежність розробки	Низька	Висока
Складність запитів	Прості JOIN операції	Складна агрегація даних
Відмовостійкість	Єдина точка відмови	Ізольовані відмови
Технологічна гнучкість	Одна СУБД	Різні типи сховищ

1.2.2 Переваги та проблеми мікро сервісної архітектури

Переваги мікросервісів — незалежний розвиток команд, технологічна різноманітність, горизонтальне масштабування, підвищена відмовостійкість [3]. Ш видкість впровадження нових функцій теж плюс. Але з'являються нові проблеми. Управління розподіленими системами, цілісність даних, моніторинг

складних взаємодій, операційна складність. Потрібні нові підходи, інструменти, кваліфіковані спеціалісти.

Проблеми розподілених систем — це окрема тема [7]. Кожна взаємодія між сервісами: мережевий запит, серіалізація/десеріалізація, TCP з'єднання, DNS резолюція. Це додає мілісекунди, іноді десятки мілісекунд до часу відповіді. Для систем ціноутворення де користувачі чекають миттєвої відповіді — це критично. Множинні виклики між сервісами можуть сильно сповільнити систему [8]. Кешування, connection pooling, інші техніки оптимізації.

1.3 Порівняльний аналіз монолітної та мікросервісної архітектур

Вибір між монолітом і мікросервісами — не просте рішення [6]. Все залежить від багатьох факторів: розмір команди, досвід, складність бізнес-логіки, вимоги до масштабованості. Фінансові ресурси для інфраструктури, організаційна культура, часові обмеження, стратегічні цілі компанії — все це треба враховувати. Для систем динамічного ціноутворення неправильний вибір може коштувати дорого — втрата конкурентних переваг, падіння прибутковості. З досвіду можу сказати що на початку планування це не завжди очевидно і це можна переглянути у таблиці 1.2.

Таблиця 1.2 — Порівняльний аналіз архітектурних підходів

Критерій	Монолітна архітектура	Мікросервісна архітектура
Складність розробки	Низька (початково)	Висока
Швидкість розгортання	Швидка	Повільна
Масштабованість	Вертикальна	Горизонтальна
Відмовостійкість	Низька	Висока
Технологічна гнучкість	Обмежена	Висока

Продовження таблиці 1.2

Критерій	Монолітна архітектура	Мікросервісна архітектура
Операційна складність	Низька	Дуже висока
Час виходу на ринок	Швидкий	Повільний
Довгострокова підтримка	Складна	Гнучка

1.3.1 Критерії порівняння архітектур

Складність розробки в монолітах нижча на старті — все в одному репозиторії, компоненти взаємодіють через прямі виклики функцій [6]. Але з часом ця перевага стає недоліком. Коли система розростається, підтримувати стає важко. Для систем ціноутворення де треба часто міняти логіку — це особливо помітно.

Масштабованість — тут мікросервіси виграють [7]. Можна масштабувати кожен сервіс окремо під його потреби навантаження. Моноліт треба масштабувати весь, навіть якщо проблема в одному компоненті. Неефективне використання ресурсів виходить.

1.3.2 Стратегії міграції

Перехід від моноліту до мікросервісів не має бути миттєвим [6]. Треба робити поступово, мінімізувати ризики. Strangler Fig Pattern — поступово заміняєш частини моноліту новими мікросервісами, система працює весь час. Для критичних систем ціноутворення це важливо, бо навіть короткий простій — це фінансові втрати.

Гібридний підхід часто оптимальний [3]. Стабільні частини моноліту залишаєш, а в мікросервіси виносиш тільки те що треба часто міняти або масштабувати окремо. Так отримуєш переваги обох підходів.

1.4 Архітектурні рішення для масштабованих систем ціноутворення

Системи динамічного ціноутворення — це складна архітектурна структура [10]. Треба обробляти величезні обсяги даних в реальному часі із мінімальною затримкою передачі даних, а також аналізувати поведінку мільйонів користувачів, використовувати ML для прогнозування попиту і оптимізації цін. Для проектування сучаснрох інтернет комерції це важливо розуміти.

Також складна бізнес-логіка, регуляторні обмеження, безперервна висока доступність — все це впливає на прибутковість [11]. Інтеграція з десятками різних систем, масштабованість для пікових навантажень ціна на хмарні потужності і маштабованість не у вертимальному, а у горизонтальному рівні. Потрібні спеціалізовані архітектурні підходи які враховують специфіку продажу в інтернеті.

1.4.1 Специфічні вимоги до систем ціноутворення

Обробка в реальному часі критична [12], а ціни мають оновлюватися швидко і також змінилися ринкові умови, поведінка користувачів, рівень запасів. Архітектура повинна обробляти потоки подій в реальному часі, швидко приймати рішення про зміну цін, а затримки є втратою конкурентних переваг, що призводить до втрати прибутків [14] наведено в таблиці 1.3.

Таблиця 1.3 — Вимоги до продуктивності систем ціноутворення

Компонент	Вимоги до латентності	Пропускна здатність	Доступність	Критичність
Отримання цін	< 50ms	10,000+ RPS	99.99%	Критична
Оновлення цін	< 200ms	1,000+ RPS	99.9%	Висока
ML аналіз	< 5s	100+ RPS	99.5%	Середня

Продовження таблиці 1.3

Компонент	Вимоги до латентності	Пропускна здатність	Доступність	Критичність
Звітність	< 30s	10+ RPS	99%	Низька

1.4.2 Архітектурні патерни для систем ціноутворення

Архітектура event-driven відіграє ключову роль. Коли змінюється ціна у конкурента або падає рівень запасів — система автоматично запускає перерахунок через події (events) [15].

CQRS патерн виявився корисним для розділення читання і запису [6]. Читання має бути дуже швидким щоб клієнти не чекали відповіді, а от оновлення цін — там уже складні розрахунки [14], тому логічно використовувати для читання in-memory кеш, а для запису звичайну реляційну БД.

1.4.3 Інтеграційні рішення та безпека

API Gateway є центром мікросервісної системи. Він контролює доступ, аутентифікацію, обмеження по запитам [16, 17]. Для систем ціноутворення це особливо важливо, тому що там конфіденційна інформація і треба різні рівні доступу для користувачів.

Безпека в таких системаї є основою для стратегії ціноутворення, маржі, конкурентній аналітиці — все це критична інформація. Потрібно шифрувати і в спокої і при передачі, контролювати доступ на всіх рівнях.

Для захисту від витоку конфіденційної інформації про ціноутворення впроваджується data masking та field-level encryption. Особливо чутливі дані, такі як собівартість продуктів, маржа та алгоритми ціноутворення, зберігаються в зашифрованому вигляді навіть у внутрішніх базах даних. Audit logging забезпечує повну трасованість всіх операцій з ціновими даними.

Архітектура повинна бути гнучкою для розвитку — нові сервіси, модифікація ML алгоритмів, інтеграція з іншими платформами [15] можна переглянути на рисунку 1.2. У інтернет комерції все змінюється швидко, тому адаптивність важлива [16]. Але це означає постійну підтримку і оновлення, що теж коштує ресурсів.

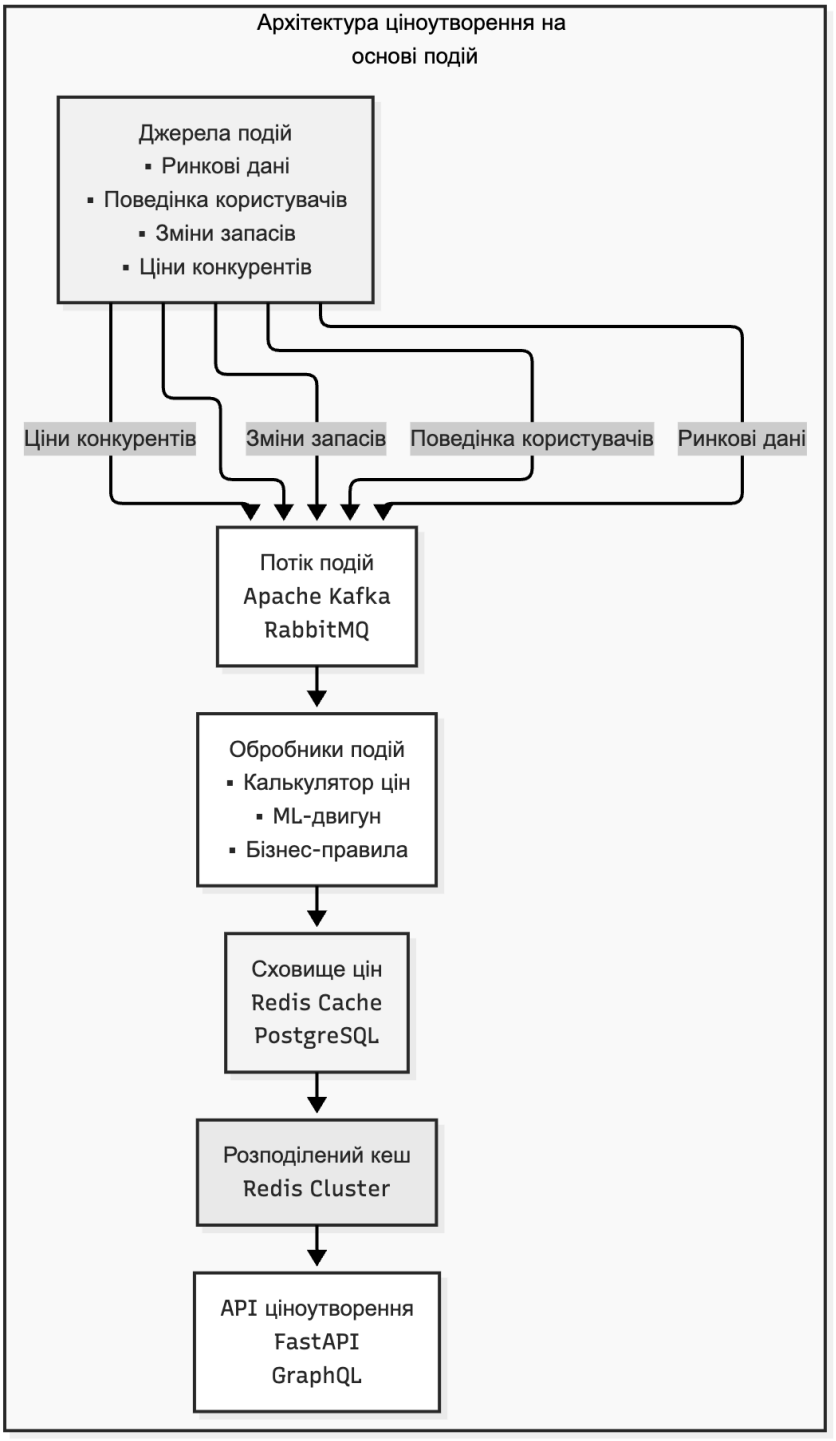

Рисунок 1.2 — Event-driven архітектура для систем ціноутворення

Моніторинг теж складний і розподілених системах треба відстежувати кожен компонент окремо, а саме логи, метрики, алерти, що є стандартним набором. Головне щоб проблеми виявляти до того як вони вплинуть на бізнес, хоча це не завжди виходить.

Для систем ціноутворення критично важливим є real-time моніторинг ключових бізнес-метрик: швидкість обробки запитів на зміну цін, точність прогнозування попиту, відсоток успішних транзакцій.

Впровадження distributed tracing через OpenTelemetry дозволяє відстежувати повний шлях запиту через всі мікросервіси, що особливо важливо для діагностики проблем у складних ланцюжках обчислення цін. Centralized logging через ELK stack (Elasticsearch, Logstash, Kibana) або аналогічні рішення забезпечує можливість швидкого пошуку та аналізу проблем.

Система алертів налаштовується на основі SLA метрик: якщо час відповіді API перевищує 200ms або точність прогнозування цін падає нижче 85%, система автоматично сповіщає відповідальних спеціалістів. Predictive alerting на основі ML моделей може передбачати потенційні проблеми до їх виникнення, аналізуючи тренди в метриках продуктивності.

2 ПРОЕКТУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ СИСТЕМИ ЦІНОУТВОРЕННЯ

2.1 Поділ системи на мікросервіси

Розділення монолітної системи на мікросервіси — складна справа особливо для систем ціноутворення, де кожна помилка коштує дорого. Коли переглядаєш струкруту монолоітної системи на перший погляд її розділити не так важко, розбий функціональність між сервісами і готово, але насправді це набагато складніше. Якщо неправильно спроектувати розділення, можна створити розподілений моноліт — це антипатерн який поєднує недоліки обох підходів, а переваг не дає жодних [6]. З досвіду можу сказати що успішне розділення потребує системного підходу. Треба використовувати перевірені методології, інакше можна наробити помилок які потім важко виправити.

2.1.1 Принципи виділення мікросервісів

Domain-Driven Design (дизайн орієнтований на домен) тут дійсно допомагає [3]. Така методологія дає структурований підхід для визначення меж розділення функціональності. Bounded contexts — це логічно пов'язані групи функціональності, їх виділити найважливіше, це як в системі динамічного ціноутворення ми проаналізували бізнес-процеси і виділили основні домени і у кожного чіткі функціональні межі. Треба враховувати багато нюансів що кожен домен має свої особливості:

Домен управління користувачами — тут відьувється аутентифікація, авторизація створення профілів користувачів. Це фундамент для всієї системи, без нього нічого не працює. Безпека та персоналізація залежать саме від цього домену.

Домен управління товарами охоплює каталог, категорії, варіанти товарів, всі характеристики які треба синхронізуватися з зовнішніми платформами типу WooCommerce [16]. Це завдання виявилось складним, бо кожна платформа має свої API і свої властивості.

На рисунку 2.1 показано як відбувається розділення монолітної системи.

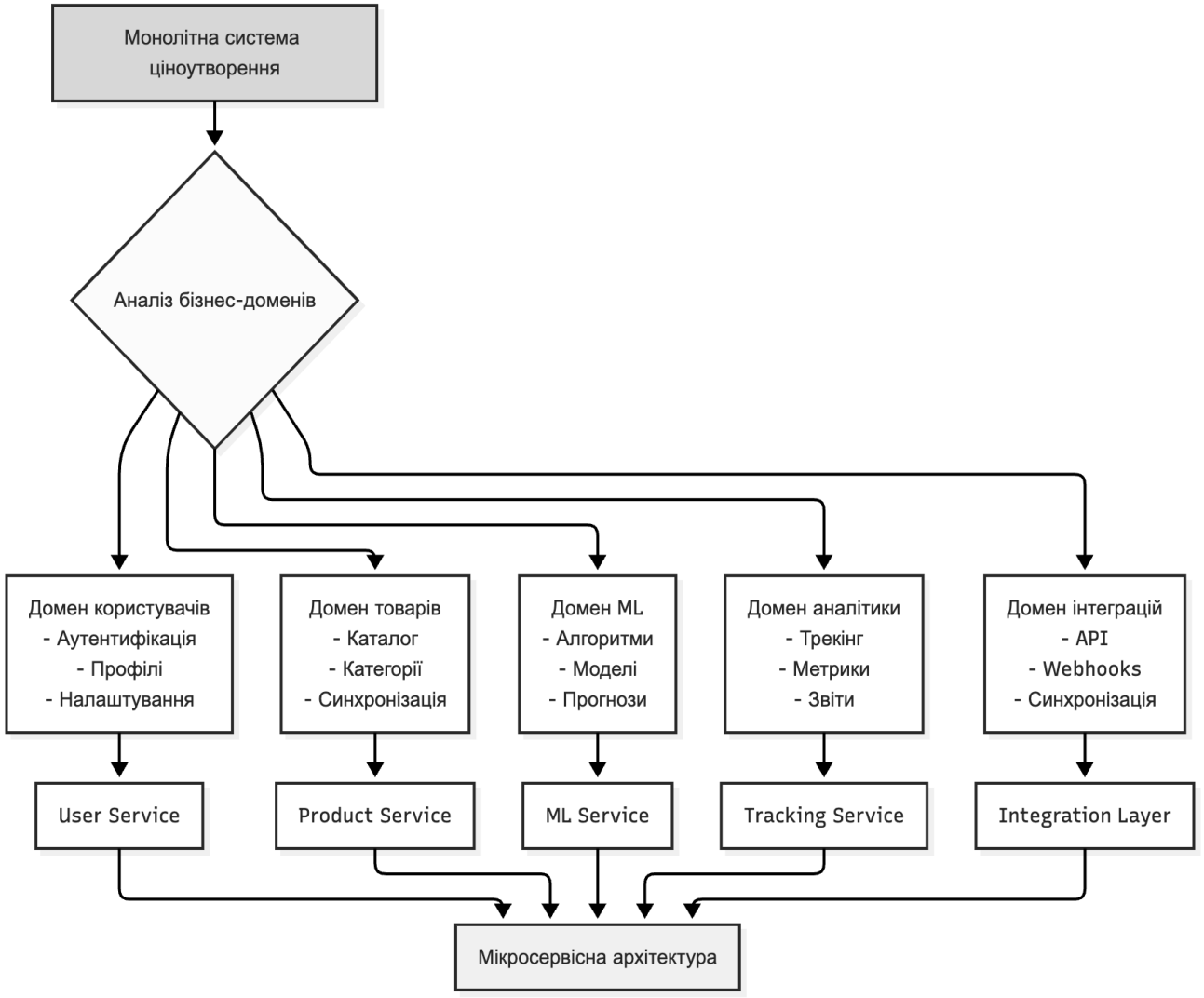

Рисунок 2.1 — Процес декомпозиції монолітної системи на домени

Домен машинного навчання тут всі алгоритми аналізу поведінки, прогнозування попиту, оптимізація цін, рекомендації [10, 11]. Для нього необхідні високі обчислювальні вимоги і моделі треба постійно тренувати. Якщо дані не якісні то це дуже важко реалізувати так як вхідні дні впливають на аналіз самої мделі.

Домен збору та аналізу даних і трекінгу поведінки користувачів, метрики взаємодії з товарами, аналіз конверсій, звіти [13, 14].

Домен інтеграцій керує взаємодією з зовнішніми системами — комерційні платформи, платіжні системи, управління складом [5, 16]. Тут треба особлива

увага до деталей інтеграції із API чи створення плагіну, бо інтеграції часто ламаються.

2.1.2 Архітектурне розділення системи

На основі цих доменів проектуємо мікросервісну архітектуру. Тут важливо не тільки розділити функціональність, але й забезпечити щоб сервіси нормально взаємодіяли між собою [6]. Слабке зв'язування і висока згуртованість це те до чого треба прагнути, хоча досягти не завжди легко.

Клієнтський рівень мікоросервісної (рис. 2.2) працює із Front-end архітектурою взаємодії між сервером, Ract.js панеллю керування, плагіном на WooCommerce і можливістю підключення мобільних додатків через API запити. На рисунку 2.3. показана архітектура, яка пов'язує всі мікросервіси і в даному прикладі є можливість побачити як вони взаємодіють між собою через API Gateway.

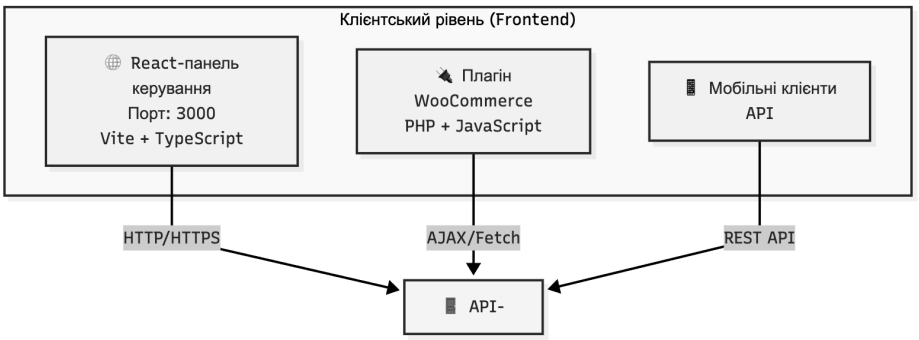

Рисунок 2.2 — Клієнтський рівень мікросервісної системи

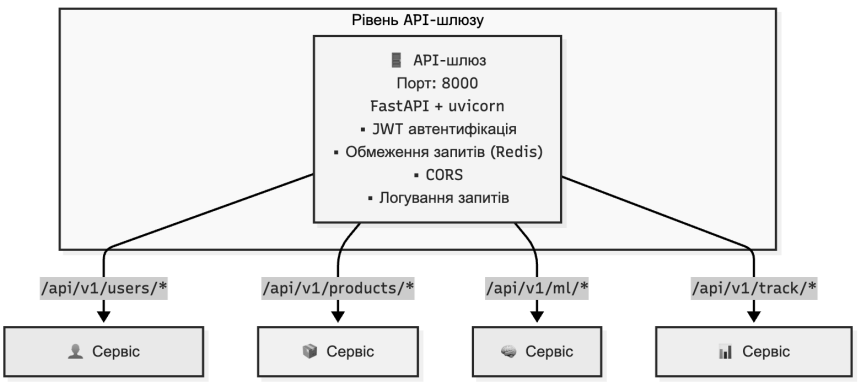

Рисунок 2.3 — Архітектура рівня шлюзу комунікації у системі

Кожен мікросервіс має свою бізнес-логіку, що виконує відповідні розранунки. Передачу і отримання даних, обробку цих даних, або генерацію як приклад JWT токени чи інтеграція із WooCommerce для отримання даних із інтернет магазину. Це можна побачити на рисунку 2.4 де показано, як мікросервіси між собою комунікують.

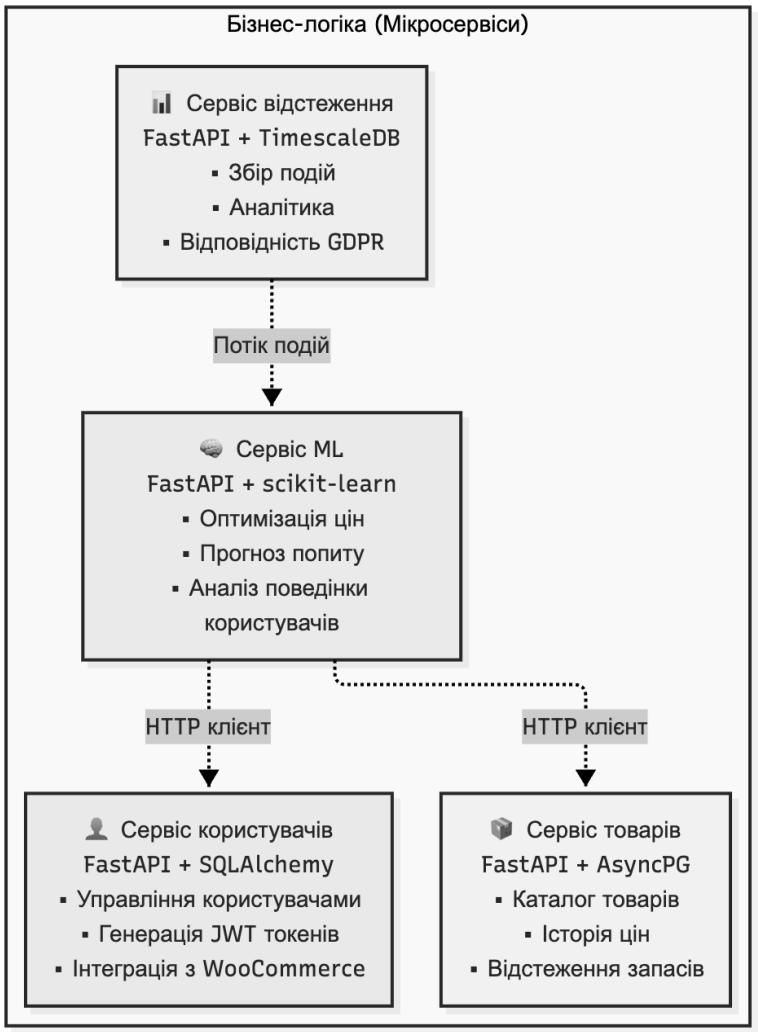

Рисунок 2.4 — Деталізація взаємодії мікросервісів через API Gateway

Важливою частиною є обробка запитів. Даний етап є критичним для швидкодії комунікації і обробки даних сервісами (рис. 2.5). Якщо до прикладу на Gateway сервісі ми не будемо використовувати Redis для кершування даними у нас виникне велика затримка через монотонні запити до прикладу найпростіше це кешування категорій товарів які буду оновлюватись дуже рідко і на кожен запит від клієнта потрібно обробляти отримання категорії товарів.

Якщо ми працюємо із вкладкою товари це додаткові 2-3 секунди затримки на запит при навантаженні у 1000 користувачів одночасно, але використовуючи Redis, це оптимізовує вивід даних до 10 мілісекунд на запит.

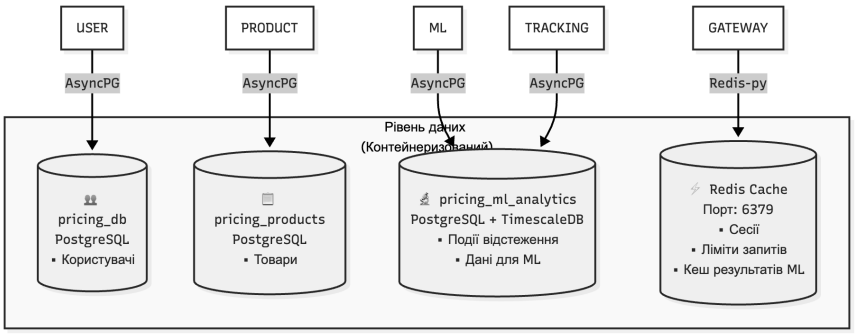

Рисунок 2.5 — Схема обробки запитів та управління даними

Важливою частиною роботи сервісів також є моніторинг роботи кожного сервіса і його відмовостійкість, логування і оркестрування контейнерів і перевірка станів роботи (рис. 2.6).

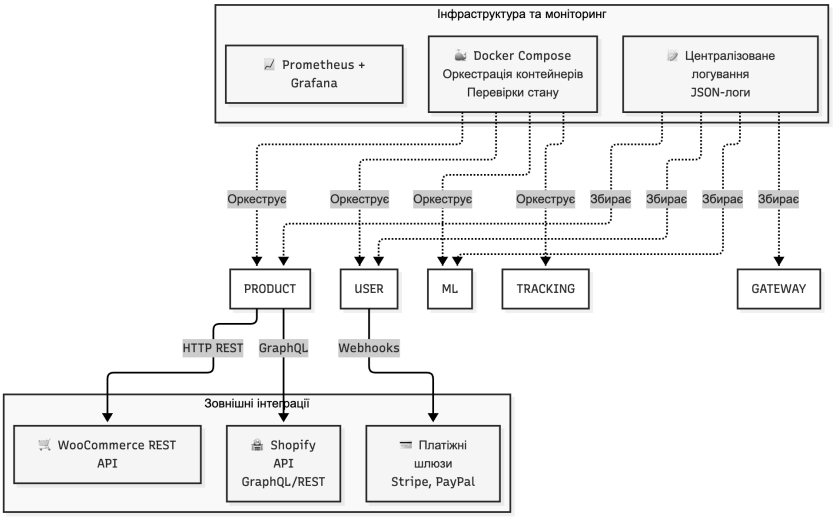

Рисунок 2.6 — Інтеграція з зовнішніми системами та моніторинг

2.1.3 Детальна архітектура мікро сервісів

Кожен мікросервіс повністю автономний — своя бізнес-логіка, своя база, чітка сфера відповідальності [1, 3]. Проектування міжсервісної взаємодії окреме складне завдання. Детальні UML-діаграми класів та послідовності взаємодії

між мікросервісами представлені в Додатку Л.

User Service — центральний компонент безпеки він зображений на рисунку 2.7. Контроль доступу, аутентифікація, авторизація. Технічно використовуємо FastAPI, бо він швидкий і зручний:

— SQLAlchemy ORM для роботи з PostgreSQL;

— Passlib для хешування паролів (PBKDF2-SHA256);

— PyJWT для токенів;

— Redis для кешування сесій та rate limiting;

— Pydantic для валідації даних.

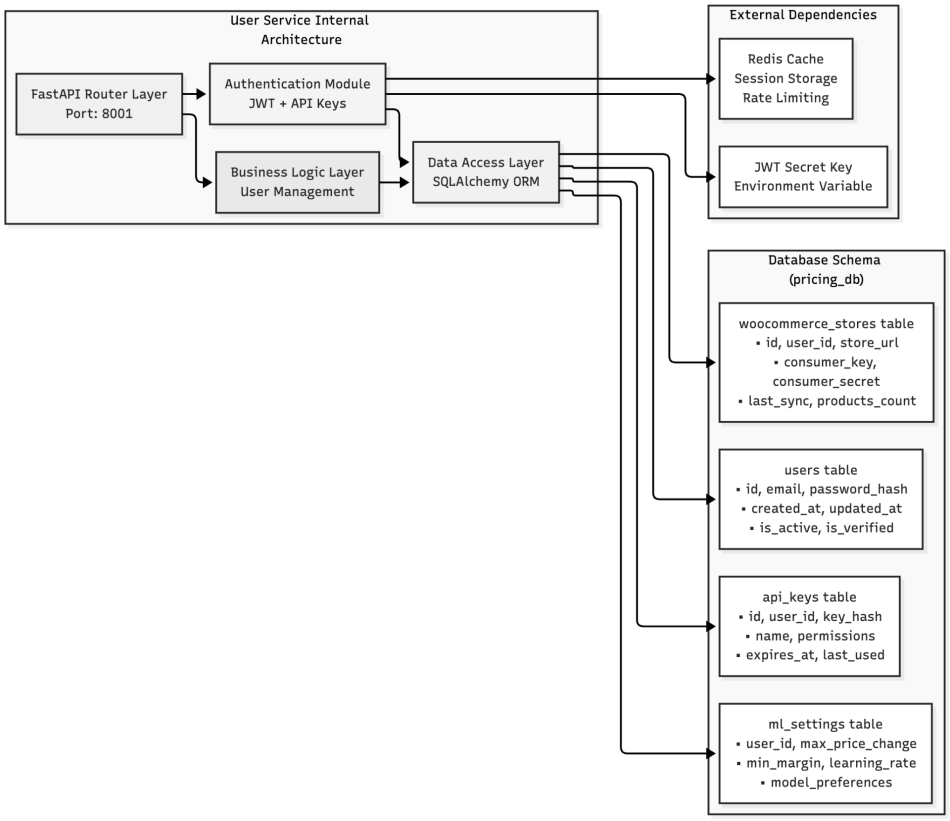

Рисунок 2.7 — Внутрішня архітектура User Service

Підсистема управління API ключами тут особливо важлива. Розділений контроль доступу, налаштування прав, терміни дії сесій і токенів, відкликання ключів. Як будь яка система її потрібно постійно контролювати і оптимізовувати для стабільної роботи.

Product Service — централізоване управління каталогом з синхронізацією

з різними комерційними платформами [16]. Проблема такої системи це як підтримувати цілісність даних між внутрішнім каталогом і зовнішніми системами і конфлікти даних можуть виникати часто коли зовнішні джерела розходяться з внутрішніми змінами. Для рішення цієї проблеми було реалізовано пріоритети джерел і часові мітки синхронізації товарів. Процес синхронізації показаний на рисунку 2.8.

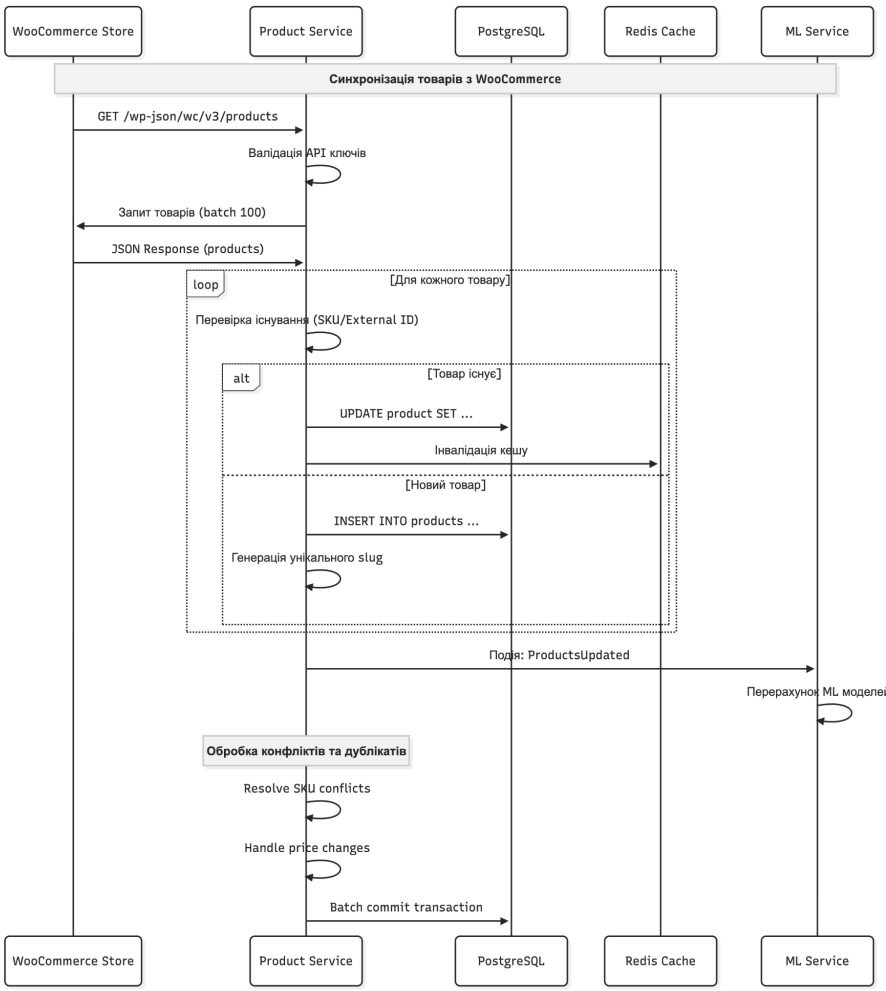

Рисунок 2.8 — Процес синхронізації товарів з зовнішніми платформами

Технічна реалізація системи передбачає використання AsyncPG для виконання швидких асинхронних запитів та бібліотеки httpx для роботи з HTTP-клієнтами для взаємодії із зовнішніми API. Для зберігання даних використовується PostgreSQL, зокрема його JSON-поля для розміщення платформо-специфічних даних. Цілісність даних забезпечується за допомогою

унікальних обмежень (unique constraints) на комбінацію полів (user_id, platform, external_id), а для завдань масової синхронізації використовуються фонові завдання (background tasks).

ML Service (Сервіс машинного навчання) реалізує ключові алгоритми, що виконують аналіз поведінки користувачів на основі трекінгових даних, прогнозування попиту та еластичності цін, а також генерують рекомендації щодо оптимальних цін. Цей сервіс також забезпечує персоналізацію цінових стратегій для різних сегментів. Для забезпечення масштабованості системи при зростанні кількості клієнтів застосовується федеративна архітектура машинного навчання, яка поєднує спільні базові моделі з персональними адаптерами для кожного користувача.

Tracking Service (Сервіс відстеження) спеціалізується на зборі та обробці поведінкових даних. Його основні функції включають збір подій взаємодії користувачів з товарами, обчислення метрик залученості та конверсії, агрегацію даних для ML моделей та реал-тайм обробку подій для забезпечення швидкої реакції системи. Детальна обробка та витягування ознак із поведінкових даних описана в Додатку В. Для ефективної роботи з часовими рядами використовується TimescaleDB — розширення PostgreSQL, оптимізоване для time-series даних.

2.2 Проектування API Gateway та сервісної взаємодії

2.2.1 Архітектурні патерни мікросервісної взаємодії

Перш ніж говорити про API Gateway, треба розібратися як взагалі мікросервіси спілкуються. Це не просто HTTP запити взаємодії у розподіленій архітектурі яка має свої особливості [6].

Database per Service Pattern — кожен сервіс має власну базу [1], і на перший погляд цей патерн здається простим, коли кожен сервіс має свою базу, то як забезпечити консистентність даних? Тут на допомогу приходять інші патерни, які не завжди легко реалізувати. Рисунок 2.9 показує проблеми цього патерну.

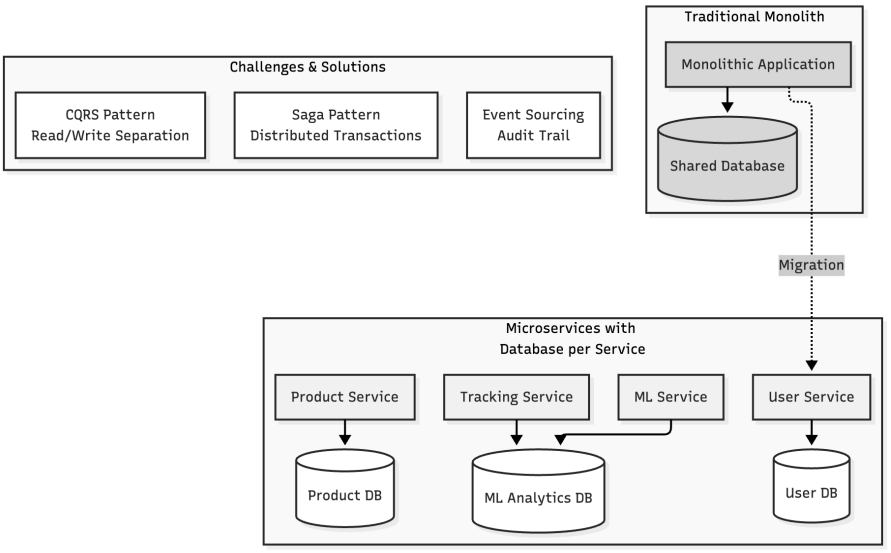

Рисунок 2.9 — Database per Service Pattern та його проблеми

Основні функції API Gateway в системі ціноутворення включають низку критично важливих завдань. Передусім, це маршрутизація запитів: Gateway аналізує вхідні запити (наприклад, /api/v1/users/* або /api/v1/products/*) і спрямовує їх до відповідних мікросервісів. Для агрегації даних може бути використаний патерн API Composition Pattern, як це відображено на рисунку 2.10.

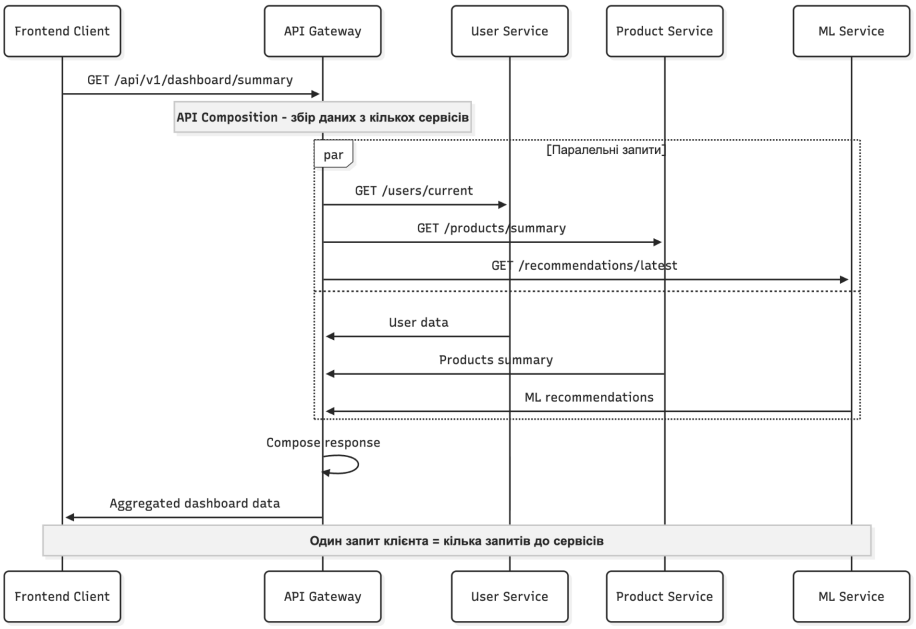

Рисунок 2.10 — API Composition Pattern для агрегації даних

Друга ключова функція — це аутентифікація та авторизація. Gateway бере на себе перевірку JWT токенів для веб-клієнтів та API ключів для зовнішніх інтеграцій, що дозволяє мікросервісам зосередитися виключно на бізнес-логіці. Важливо зазначити, що для виявлення та вирішення проблем, які іноді виникають з токенами, необхідний надійний механізм логування.

А також, трансформація запитів та відповідей є необхідною для адаптації форматів даних, що забезпечує сумісність між різними версіями API. Трансформація запитів також необхідна для підтримки старих версій клієнтських додатків без необхідності негайного оновлення всіх мікросервісів. Взаємодія компонентів через API Gateway детально показана на рисунку 2.11.

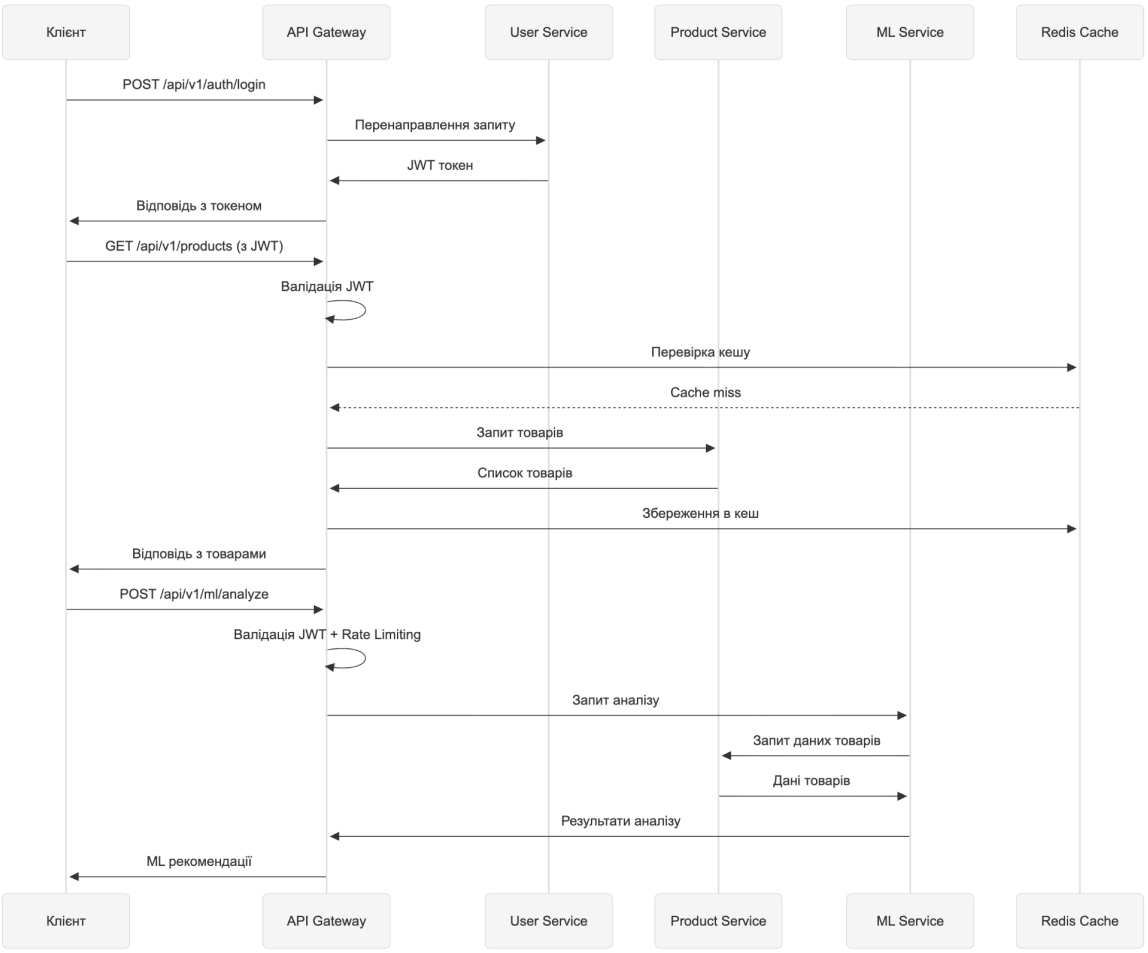

Рисунок 2.11 — Взаємодія компонентів через API Gateway

Крім того, Gateway виконує функцію Rate Limiting — обмеження кількості запитів від одного клієнта. Це необхідно для захисту від DDoS атак та

забезпечення справедливого розподілу ресурсів. Це досягається завдяки використанню алгоритмів, таких як Leaky Bucket або Token Bucket, які забезпечують стабільну пропускну здатність. У процесі тестування для оптимізації продуктивності сервісів довелося інтегрувати WebSocket. Це дозволило реалізувати механізм оновлення цін та аналітичних даних на клієнтському інтерфейсі у режимі реального часу.

Логування та моніторинг також централізовано здійснюються через Gateway, що охоплює збір метрик продуктивності, логування помилок та аудит безпеки. Для забезпечення всебічного контролю застосовується патерн Structured Logging та інструменти Distributed Tracing. Це дає можливість швидко локалізувати несправності та вузькі місця в розподіленій мікросервісній архітектурі.

2.2.3 Патерни між сервісної комунікації

Для забезпечення ефективної взаємодії між мікросервісами в системі використовуються різні архітектурні патерни.

Першим типом є синхронна комунікація через HTTP/REST. Вона застосовується для операцій, які вимагають негайної відповіді. Наприклад, ML Service синхронно надсилає запит до Product Service для отримання актуальних даних про товари, необхідних для генерації рекомендацій.

Другий підхід — асинхронна комунікація через події. Вона використовується для операцій, які не потребують негайної відповіді. Коли в Product Service відбувається оновлення товару, генерується відповідна подія. ML Service може обробити цю подію пізніше, щоб перерахувати рекомендації.

Крім того, для підвищення надійності системи застосовується патерн Circuit Breaker Pattern (Запобіжник). Його основна функція полягає у захисті від каскадних відмов. Якщо один із мікросервісів стає недоступним, API Gateway може повернути клієнту кешовані дані або заздалегідь визначену резервну відповідь (fallback). Такий механізм запобігає поширенню помилки на всю систему, що є критично важливим у розподіленій архітектурі.

2.2.4 Управління версіями API

Система підтримує версіонування API через використання URL-префіксів, таких як /api/v1/ та /api/v2/. Цей підхід надає низку важливих переваг. Зокрема, він дозволяє поступово впроваджувати нові функції без порушення роботи існуючих інтеграцій, що є критично важливим для стабільності.

Таке версіонування також забезпечує підтримку зворотної сумісності для зовнішніх клієнтів, які можуть продовжувати використовувати старі версії API, доки не будуть готові перейти на нові. Крім того, наявність різних версій полегшує A/B тестування нових версій API на обмеженій групі користувачів перед їх повноцінним запуском.

2.3 Управління даними в розподіленій архітектурі

2.3.1 Принцип Database per Service

Один з головних принципів мікросервісної архітектури — це принцип "Database per Service", згідно з яким кожен окремий сервіс має власну, незалежну базу даних [6]. Цей підхід забезпечує високу децентралізацію та автономність компонентів системи.

Детальна реалізація ініціалізації та управління цими розподіленими базами даних у системі наведена в Додатку Г.

Така архітектура надає системі низку важливих можливостей та переваг. Вона гарантує повну незалежність розробки та розгортання сервісів, дозволяючи командам працювати автономно. Крім того, з'являється можливість вибору оптимальної технології зберігання даних (наприклад, реляційної або NoSQL) для кожного конкретного сервісу, залежно від його потреб.

Найважливіше — принцип забезпечує ізоляцію даних та зменшення ризику каскадних відмов, оскільки збій однієї бази даних не поширюється на інші сервіси. Загальна архітектура даних системи, що ілюструє цей принцип, детально показана на рисунку 2.12.

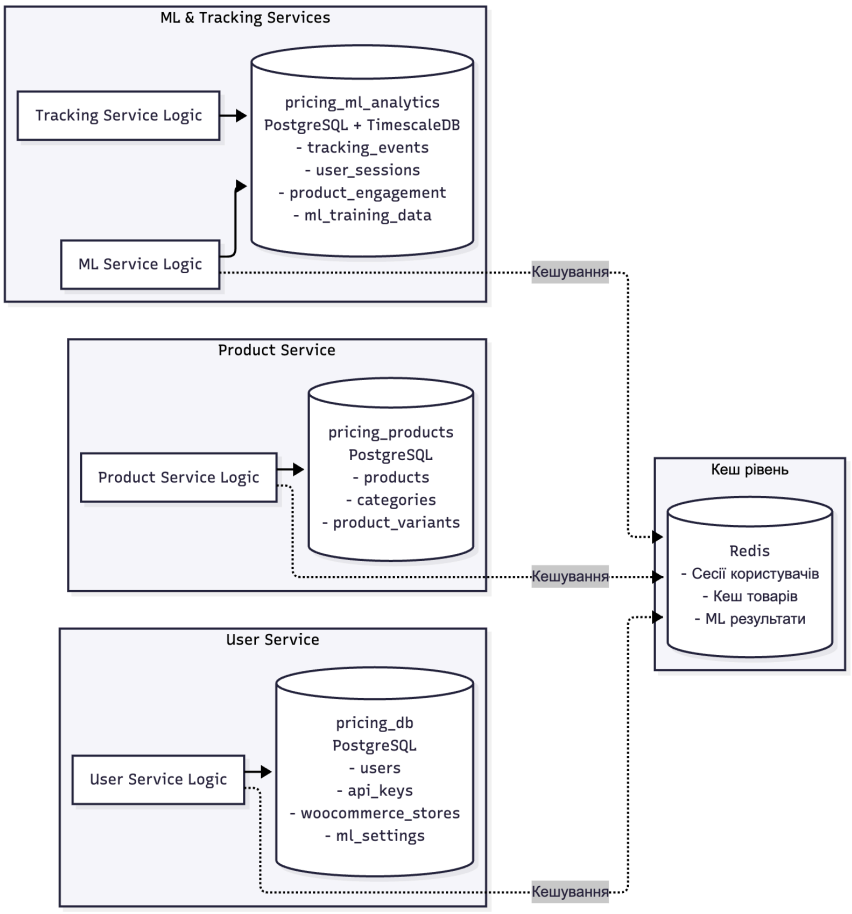

Рисунок 2.12 — Архітектура даних мікросервісної системи

2.3.2 Вибір технологій зберігання даних

Для системи динамічного ціноутворення було обрано кілька ключових технологій для зберігання та обробки даних:

PostgreSQL слугує основною системою керування базами даних (СУБД) для більшості сервісів. Його вибір обґрунтований тим, що він забезпечує необхідну надійність та властивості ACID для критичних бізнес-даних. Крім того, PostgreSQL підтримує JSON-типи для гнучкого зберігання метаданих, надає потужні можливості індексування та оптимізації запитів, а також має широку екосистему розширень.

TimescaleDB використовується як розширення PostgreSQL спеціально для ML Analytics бази. Таке рішення оптимізовано для роботи з часовими рядами, забезпечуючи автоматичне партиціонування великих таблиць, ефективне стиснення історичних даних та підтримку складних аналітичних запитів.

Redis застосовується для кешування та керування сесіями. Він забезпечує високошвидкісний доступ до даних, оскільки працює в пам'яті. Redis підтримує різні структури даних, дозволяє автоматично видаляти застарілі дані за часом життя (TTL) та підтримує кластеризацію для забезпечення високої доступності.

WebSocket інтегровано для оптимізації запитів і отримання даних на льоту. Ця технологія дозволяє встановити постійне двостороннє з'єднання між клієнтом і сервером, що критично важливо для передачі оновлень цін та іншої інформації у режимі реального часу.

2.3.3 Стратегії синхронізації даних

Оскільки мікросервіси мають окремі бази, виникає потреба в синхронізації тому реалізовано кілька підходів API-based синхронізації для операцій читання і це наведено у лістингу 2.1, а детальна реалізація наведена в Додатку Д.

Лістинг 2.1 — ML Service запитує дані товарів у Product Service

```
async def get_product_data(product_id: int) -> ProductData:
    async with httpx.AsyncClient() as client:
        response = await client.get(
            f"{PRODUCT_SERVICE_URL}/api/v1/products/{product_id}"
        )
        return ProductData(**response.json()["data"])
```

Для забезпечення цілісності даних та ефективної взаємодії між мікросервісами використовуються наступні патерни такі як Event-driven синхронізація для обміну даними та оновлень та Saga для керування складними бізнес-транзакціями.

Event-driven патерн застосовується для асинхронного обміну даними та оновлень. Принцип його роботи полягає в тому, що у разі зміни товару, наприклад, у Product Service, він публікує відповідну подію. ML Service підписується на ці події, отримує інформацію про зміни і оновлює свої внутрішні кеші. Такий підхід забезпечує так звану eventual consistency між сервісами, коли дані стають узгодженими через деякий час, а не миттєво. Для

реалізації цього механізму використовується брокер повідомлень, який забезпечує гарантовану доставку подій та роз'єднання сервісів. Це значно підвищує стійкість системи, оскільки тимчасова недоступність одного сервісу не блокує роботу інших компонентів. Крім того, Event-driven архітектура знижує пряму залежність між сервісами [6]. Це дозволяє сервісам працювати незалежно і спрощує їхнє подальше оновлення та масштабування.

Патерн Saga використовується для керування складними бізнес-транзакціями, які охоплюють декілька мікросервісів. Його суть полягає у розбитті складної операції на послідовність простих, локальних кроків-транзакцій. Кожен крок виконується окремим сервісом. Ключова перевага цього патерну — забезпечення цілісності даних без використання традиційних розподілених транзакцій (2PC). Якщо на якомусь етапі виникає помилка, Saga передбачає механізм компенсаційних дій, завдяки яким кожен попередній крок може бути відкочений, повертаючи систему у вихідний стан.

2.3.4 Управління схемами баз даних

Для кожного сервісу реалізували автоматичну ініціалізацію бази наведено в лістингу 2.2.

Лістинг 2.2 — Приклад init-db.py для Product Service

```
async def init_database():
    """Ініціалізація бази даних та створення таблиць"""
    try:
        # Підключення до бази даних
        engine = create_async_engine(DATABASE_URL)
        # Створення таблиць
        async with engine.begin() as conn:
            await conn.run_sync(Base.metadata.create_all)
        logger.info("База даних успішно ініціалізована")
    except Exception as e:
        logger.error(f"Помилка ініціалізації бази даних: {e}")
```

Цей механізм гарантує, що при першому запуску мікросервісу його незалежна база даних створюється з актуальною схемою та необхідними початковими даними. Автоматизація цього процесу спрощує розгортання та підтримує принцип ізоляції даних, критичний для розподіленої архітектури.

2.4 Забезпечення цілісності даних та надійності системи

2.4.1 Проблеми цілісності даних в розподілених системах

Перехід від монолітної архітектури до мікросервісів створює нові, складніші проблеми із забезпеченням цілісності даних. У монолітних системах властивості ACID (атомарність, узгодженість, ізольованість, довговічність) єдиної бази даних автоматично гарантували консистентність [6]. Однак у розподіленій мікросервісній системі цей процес ускладнюється.

Основні проблеми, які виникають у системі ціноутворення, включають розподілені транзакції — це операції, які вимагають оновлення даних у кількох сервісах одночасно. Наприклад, при створенні нового користувача необхідно створити запис у User Service та одночасно ініціалізувати його персональні ML-налаштування у відповідному сервісі.

Ще однією проблемою є Eventual Consistency. У розподіленій системі необхідно прийняти той факт, що дані можуть бути тимчасово неконсистентними після оновлення, але з часом вони гарантовано досягнуть узгодженого стану.

2.4.2 Паттерни забезпечення цілісності даних

Для критичних операцій ми застосовуємо Event Sourcing, за якого всі зміни фіксуються у вигляді послідовності подій, поточний стан системи відтворюється шляхом їх програвання, а також забезпечується можливість аудиту й повернення до попередніх станів.

Додатково використовується підхід CQRS, що передбачає розділення операцій читання та запису, використання оптимізованих read-моделей для швидкого отримання даних і асинхронне оновлення цих моделей на основі подій.

Що до Saga Pattern то він використовується для управління розподіленими транзакціями, які охоплюють декілька мікросервісів, наприклад, при створенні нового користувача. Рисунок 2.13 показує, як працює функціонал починаючи із

створення користувача який розбивається на послідовність локальних транзакцій у різних сервісах. Цей патерн забезпечує узгодженість даних у розподіленій архітектурі без використання складних двофазних комітів (2PC). У разі збою одного з кроків, запускається механізм транзакцій, які скасовують вже виконані дії в попередніх сервісах, забезпечуючи цілісність системи.

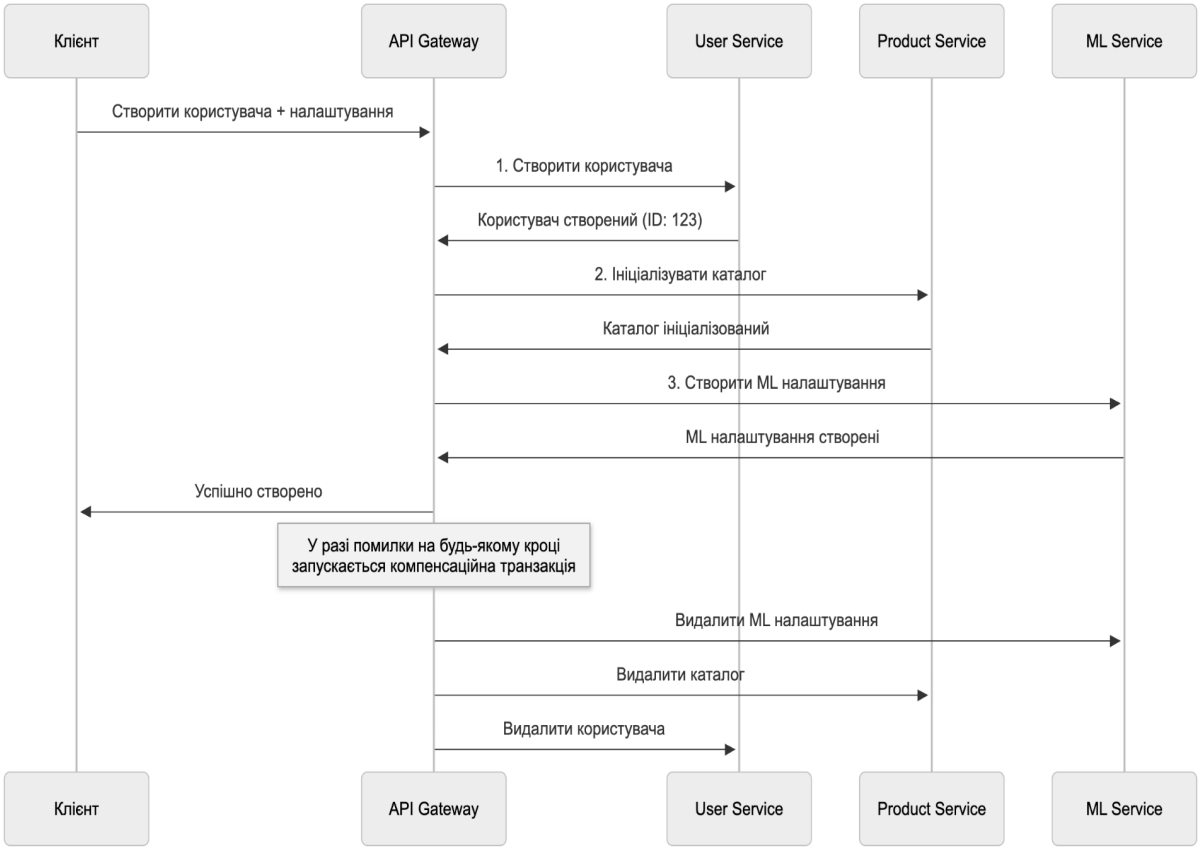

Рисунок 2.13 — Реалізація Saga Pattern для створення користувача

2.4.3 Стратегії забезпечення надійності

Надійність в мікросервісах — як будівництво карткового будинку під час землетрусу. Один сервіс падає і може завалити всю систему. Тому треба спеціальних "амортизаторів".

Circuit Breaker працює як електричний запобіжник у будинку [8]. Коли щось йде не так, він "вимикає" проблемний сервіс щоб не спалити всю систему. Рисунок 2.14 показує стани Circuit Breaker Pattern.

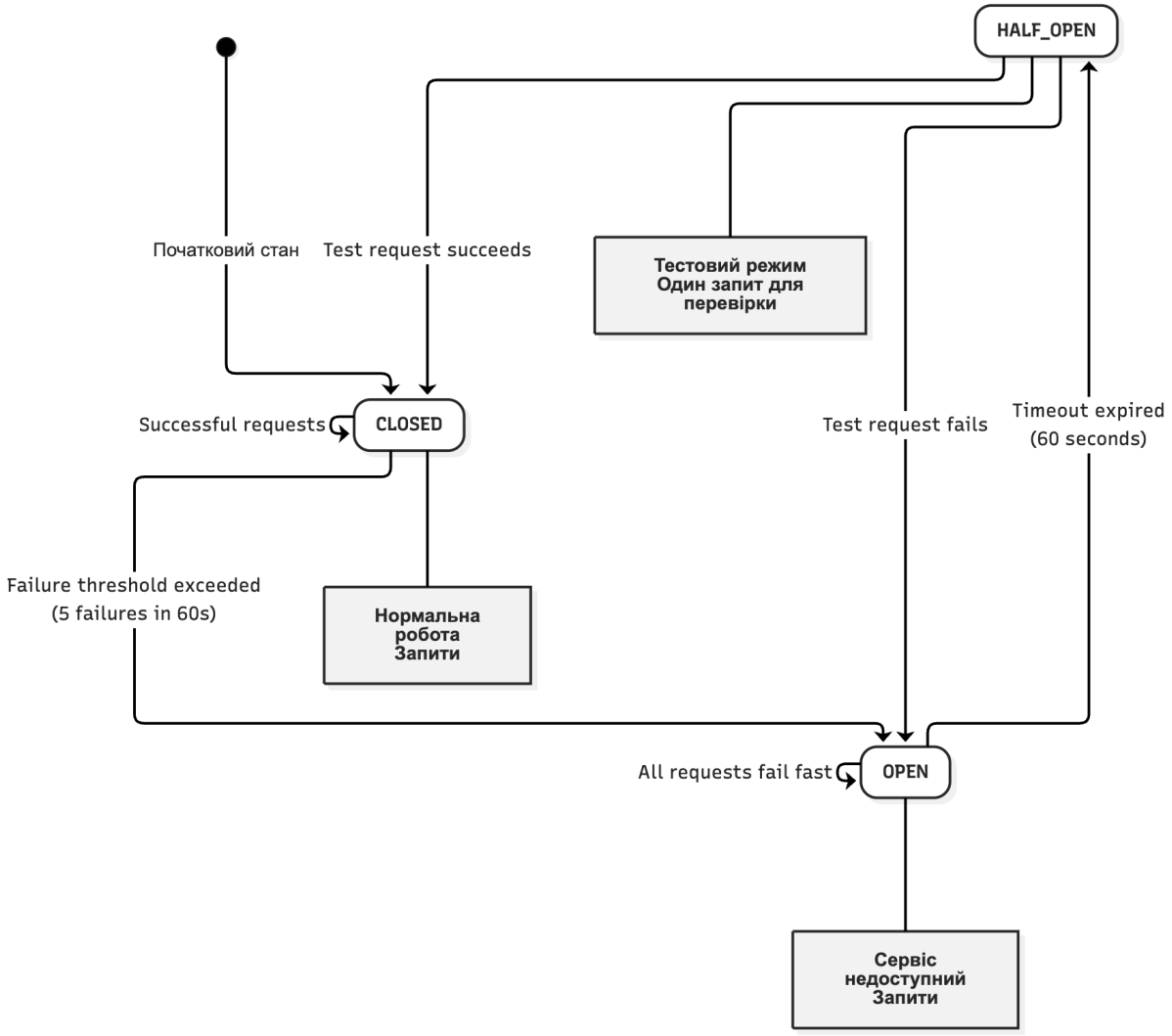

Рисунок 2.14 — Стани Circuit Breaker Pattern

Технічна реалізація Circuit Breaker наведена у лістингу 2.3 а повна реалізація коду в наведена у Додатку Е.

Лістинг 2.3 — Приклад роботи функціонал “запобіжника”

```
class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.failure_count = 0
        self.last_failure_time = None
        self.state = "CLOSED"  # CLOSED, OPEN, HALF_OPEN
```

Circuit Breaker має три стани: CLOSED (нормальна робота), OPEN (блокування запитів) та HALF_OPEN (тестування відновлення).

Що до failure_threshold то він визначає скільки помилок потрібно для

переходу в OPEN, a timeout — як довго тримати circuit відкритим це все наведено у лістингу 2.4.

Лістинг 2.4 — Приклад роботи функціонал Circuit Breaker

```
async def call(self, func, *args, **kwargs):
    if self.state == "OPEN":
        if time.time() — self.last_failure_time > self.timeout:
            self.state = "HALF_OPEN"
        else:
            raise CircuitBreakerOpenException()
    try:
        result = await func(*args, **kwargs)
        self.reset()
        return result
    except Exception as e:
        self.record_failure()
        raise e
```

У стані OPEN circuit блокує всі запити крім випадків коли timeout закінчився — тоді переходить в HALF_OPEN для тестування. При успіху circuit скидається в CLOSED і при помилці збільшується лічильник відмов (Лістинг 2.5).

Лістинг 2.5 — Приклад роботи функціоналу винятку підчас запису

```
def record_failure(self):
    self.failure_count += 1
    self.last_failure_time = time.time()
    if self.failure_count >= self.failure_threshold:
        self.state = "OPEN"
```

Кожна помилка збільшує лічильник. Коли досягає порогу, circuit переходить в OPEN і блокує подальші запити. Це захищає від каскадних відмов і дає час проблемному сервісу відновитися. У стані OPEN запобіжник, через встановлений інтервал, автоматично переходить у стан HALF-OPEN , щоб перевірити, чи відновив проблемний сервіс свою роботу.

Якщо тестовий запит є успішним, circuit повертається у стан CLOSED, відновлюючи нормальний потік трафіку. Ця комбінація механізмів є життєво необхідною для підвищення відмовостійкості мікросервісної архітектури Retry Pattern з експоненційною затримкою яка навелена у лістингу 2.6 а детальна реалізація наведена в Додатку Е.

Лістинг 2.6 — Приклад роботи функціоналу Retry Pattern

```
async def retry_with_backoff(func, max_retries=3, base_delay=1):
    for attempt in range(max_retries):
        try:
            return await func()
        except Exception as e:
            if attempt == max_retries — 1:
                raise e
            delay = base_delay * (2 ** attempt)
            await asyncio.sleep(delay)
```

Exponential backoff збільшує затримку експоненційно: 1с, 2с, 4с, 8с. Це запобігає перевантаженню проблемного сервісу одночасними retry від багатьох клієнтів. Якщо всі спроби вичерпані, оригінальна помилка пробрасується вгору. Health Checks для моніторингу стану сервісів наведено у лістингу 2.7 а повна реалізація в наведена у Додатку Д.

Лістинг 2.7 — Приклад роботи функціоналу Health Checks

```
@router.get("/health")
async def health_check():
    try:
        await database.execute("SELECT 1")
        external_services_status = await check_external_services()
        return {
            "status": "healthy",
            "timestamp": datetime.utcnow(),
            "database": "connected",
        }
    except Exception as e:
        raise HTTPException (
            status_code=503,
            detail=f"Service unhealthy: {str(e)}"
        )
```

Health check перевіряє не тільки доступність самого сервісу, а й його залежності — базу даних та зовнішні API. Простий SELECT 1 запит швидко перевіряє з'єднання без навантаження. При помилці повертається HTTP 503 що сигналізує load balancer'у виключити цей інстанс.

2.4.4 Моніторинг та логування

Структуроване логування через structured logging на ведено у лістингу 2.8 (приклад реалізації в Додатку Д).

Лістинг 2.8 — Приклад роботи функціоналу Structured Logging

```
import structlog
logger = structlog.get_logger()
async def process_pricing_request(user_id: int, product_id: int):
   logger.info(
      "Processing pricing request",
      user_id=user_id,
      product_id=product_id,
      service="ml-service"
   )
```

Structlog дозволяє логувати структуровані дані замість текстових повідомлень [17]. Кожен лог запис містить контекст — user_id, product_id, назву сервісу. Це критично для debugging в мікросервісній архітектурі де запит проходить через кілька сервісів (Лістинг 2.9).

Лістинг 2.9 — Функціонал Debugging у Calculate optimal price

```
try:
   result = await calculate_optimal_price(user_id, product_id)
    logger.info(
       "Pricing calculation completed",
         user_id=user_id,
         product_id=product_id,
         recommended_price=result.price,
         confidence=result.confidence
      )
      return result
```

При успіху логуємо результат з ключовими метриками — рекомендовану ціну і рівень впевненості моделі. При помилці логуємо повну інформацію включаючи stack trace. Структурований формат дозволяє легко фільтрувати логи через ELK stack.

Розподілене трасування є критично важливим механізмом для відстеження та аналізу запитів, які проходять через множину мікросервісів [18]. Для забезпечення ефективного трасування кожен вхідний запит отримує унікальний ідентифікатор трасування (trace ID), який послідовно передається всім сервісам, задіяним в обробці цього запиту.

Усі операції, що виконуються в рамках одного запиту, логуються з використанням цього єдиного trace ID, що дозволяє здійснювати кореляцію розрізнених логів. Завдяки цьому інструмент надає можливість відстежити

повний шлях проходження запиту від початку до кінця та оперативно виявити вузькі місця або джерела затримок у розподіленій системі.

2.5 Контейнеризація мікросервісів

2.5.1 Docker контейнеризація

Кожен мікросервіс упакований в Docker контейнер [17]. Це дає ізоляцію середовища виконання, консистентність між різними середовищами, простоту розгортання і масштабування.

Використовуємо Python 3.11-slim як базовий образ це компромісний варіант між розміром та функціональністю. Встановлюємо gcc і g++ компілятори які потрібні для збірки деяких Python пакетів, особливо NumPy, SciPy, scikit-learn (Лістинг 2.10).

Лістинг 2.10 — Приклад лістингу Dockerfile файлу для ML Service

```
FROM python:3.11-slim
WORKDIR /app
RUN apt-get update && apt-get install -y \
   gcc \
   g++ \
   && rm -rf /var/lib/apt/lists/*
```

Спочатку копіюємо тільки requirements.txt і встановлюємо залежності це дозволяє Docker кешувати цей шар (Лістинг 2.11) якщо залежності не змінилися. --no-cache-dir економить місце, не зберігаючи кеш pip. Потім копіюємо код в окремому шарі.

Лістинг 2.11 — Встановлення Python залежностей

```
dockerfile
# Встановлення Python залежностей
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Копіювання коду додатку
COPY app/ ./app/
```

Створюємо директорію для ML моделей де будуть зберігатися натреновані моделі (Лістинг 2.12). PYTHONPATH дозволяє Python знаходити модулі додатку, ML_MODELS_PATH вказує де шукати файли моделей. EXPOSE

документує який порт використовує контейнер. Команда запуску використовує uvicorn ASGI сервер з --reload для автоматичного перезавантаження при змінах коду . Параметр --reload використовується тут для зручності розробки, але на продакшен середовищі його слід вимкнути для підвищення стабільності. Налаштування порту та команди запуску дозволяє API Gateway легко маршрутизувати зовнішні запити безпосередньо до цього мікросервісу, забезпечуючи доступність ML-функціоналу.

Лістинг 2.12 — Створення директорії для ML моделей

```
# Створення директорії для ML моделей
RUN mkdir -p /app/data/models
# Налаштування змінних середовища
ENV PYTHONPATH=/app
ENV ML_MODELS_PATH=/app/data/models
# Відкриття порту
EXPOSE 8002
# Команда запуску
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8002", "--reload"]
```

2.5.2 Docker Compose

Для ефективної локальної розробки та тестування всієї мікросервісної архітектури застосовується Docker Compose. Це дозволяє легко управляти всіма компонентами системи та їхніми залежностями за допомогою єдиного конфігураційного файлу.

Як основу для бази даних обрано PostgreSQL 15 Alpine, що забезпечує необхідну надійність при мінімальному розмірі контейнера. Ключовим рішенням для архітектурної чистоти та безпеки є налаштування трьох окремих баз даних для різних мікросервісів, що забезпечує ізоляцію даних згідно з принципом "Database per Service".

Критично важливим елементом є механізм Health Check, що забезпечує правильний порядок запуску залежних мікросервісів, запобігаючи помилкам підключення на старті. Детальна конфігурація цього сервісу наведена в лістингу 2.13, а повна конфігурація представлена в Додатку Ж [17]. Цей механізм також відіграє ключову роль у підтримці загальної надійності системи,

дозволяючи API Gateway динамічно виключати несправні сервіси з маршрутизації трафіку.

Лістинг 2.13 — Лістинг блоку Docker Compose файлу

```
version: '3.8'
services:
  postgres:
    image: postgres:15-alpine
    environment:
          POSTGRES_DB: pricing_db
          POSTGRES_USER: pricing_user
          POSTGRES_PASSWORD: pricing_password
          POSTGRES_MULTIPLE_DATABASES:
pricing_db,pricing_products,pricing_ml_analytics
    ports:
          — "5432:5432"
    volumes:
          — postgres_data: /var/lib/postgresql/data
          —./database/init-multiple-databases.sh:/docker-entrypoint-initdb.d/init-multiple-data
bases.sh
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U pricing_user -d pricing_db"]
      interval: 10s
      timeout: 5s
      retries: 5
```

Redis використовується для кешування ML прогнозів, сесій користувачів, міжсервісної комунікації. Alpine версія зменшує розмір. Health check через redis-cli ping забезпечує що Redis готовий перед запуском залежних сервісів (Лістинг 2.14).

Лістинг 2.14 — Лістинг блоку Redis налаштування у Docker Compose

```
 redis:
    image: redis:7-alpine
    ports:
      — "6379:6379"
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 5s
      retries: 3
```

User Service використовує окрему базу і Redis для сесій. JWT_SECRET_KEY має бути змінений в продакшені на безпечний ключ. depends_on з condition: service_healthy гарантує що PostgreSQL і Redis повністю

готові перед запуском (Лістинг 2.15). Volume mounting дозволяє hot reload під час розробки.

Лістинг 2.15 — Лістинг блоку user-service у Docker Compose

```
 user-service:
   ports:
         — "8001:8001"
   environment:
         —DATABASE_URL=postgresql://pricing_user:pricing_password@postgres:5432/p
ricing_db
         — REDIS_URL=redis://redis:6379/0
         — JWT_SECRET_KEY=your-secret-key-here
```

ML сервіс має доступ до окремої бази для аналітики [10] і використовує Redis database 1 для кешування прогнозів. Важливо що ml_models volume зберігає натреновані моделі між перезапусками контейнера.

URL інших сервісів використовують внутрішні Docker мережеві імена замість localhost (Лістинг 2.16).

Лістинг 2.16 — Лістинг блоку ml-service у Docker Compose

```
 ml-service:
   build:
         context: ./services/ml-service
   ports:
          — "8002:8002"
environment:
ML_DATABASE_URL=postgresql://pricing_user:pricing_password@postgres:5432/pricin
g_ml_analytics
     — PRODUCT_SERVICE_URL=http://product-service:8003
     — USER_SERVICE_URL=http://user-service:8001
     — REDIS_URL=redis://redis:6379/1
   depends_on:
     postgres:
       condition: service_healthy
     redis:
       condition: service_healthy
   volumes:
     — ./services/ml-service/app:/app/app
     — ml_models:/app/data/models
   restart: unless-stopped
```

API Gateway знає адреси всіх внутрішніх сервісів і використовує Redis database для власного кешування а не від баз напряму, що є правильною архітектурою, де gateway не має прямого доступу до даних, а повна

конфігурація представлена в Додатку Ж. Named volumes забезпечують збереження даних PostgreSQL і ML моделей між перезапусками. Кастомна мережа pricing-network ізолює додаток від інших Docker проектів і дозволяє сервісам знаходити один одного за іменами [17, 18].

Використання Redis для кешування суттєво знижує затримку для частих запитів, знімаючи навантаження з основних баз даних і підвищуючи загальний час відгуку системи. Така архітектурна модель, де Gateway не має прямого доступу до даних, також посилює безпеку, оскільки внутрішні джерела (PostgreSQL) ізольовані. Крім того, визначена мережа критично важлива для масштабованості, дозволяючи додавати нові екземпляри мікросервісів без необхідності ручного конфігурування мережевих параметрів.

2.5.3 Стратегії розгортання проекту

Для розгортання використовується підхід Blue-Green Deployment, який передбачає підтримку двох ідентичних продакшн-середовищ, розміщення нової версії в неактивному середовищі та швидке перемикання трафіку після успішного тестування.

Також застосовуються Rolling Updates, що забезпечують поступову заміну старих контейнерів новими, підтримують безперервну доступність сервісу під час оновлення та дають можливість оперативно виконати відкат у разі виникнення проблем.

Для безпечного впровадження нових функцій використовується Canary Deployment: спочатку лише частина трафіку спрямовується на нову версію, навантаження збільшується поступово за умови позитивних метрик, а при виявленні нестабільності система автоматично виконує відкат.

Така архітектура забезпечує ізоляцію сервісів та їх незалежне масштабування, простоту розгортання і управління, високу доступність та відмовостійкість, ефективне використання ресурсів [17, 18].

3 МАТЕМАТИЧНІ МОДЕЛІ ТА АЛГОРИТМИ МАШИННОГО НАВЧАННЯ СИСТЕМИ

3.1 Архітектура ML-системи

ML-система побудована за мікросервісною архітектурою і складається з двох основних компонентів [1, 4]. ML-Engine спеціалізується на динамічному ціноутворенні, а ML-Service відповідає за аналіз поведінки користувачів та персоналізацію рекомендацій.

Архітектура наведено у Додатку К. Така структура дає можливість масштабувати кожен компонент окремо. Це важливо для комерційних додатків де відмовостійкість критична [3, 5]. Якщо один компонент падає, інший продовжує працювати.

3.2 Федеративна система машинного навчання

Основа системи — федеративний підхід [2, 11]. Він поєднує спільні базові моделі з персональними адаптерами для кожного користувача. Це дозволяє ефективно використовувати обмежені обчислювальні ресурси і одночасно давати персоналізований досвід. Детальна реалізація федеративного ML-движка наведена в Додатку В.

3.2.1 Базові моделі

Система використовує два типи базових моделей. Модель аналізу залученості користувачів формула (3.1).

$$E = f(X) = \sum w_i \cdot x_i, \tag{3.1}$$

де X — вектор з 50 поведінкових ознак;

wi — ваги моделі.

Модель ціноутворення формула (3.2).

$$P_{opt} = \alpha \cdot P_{base} + \beta \cdot E + \gamma \cdot M \tag{3.2}$$

3.2.2 Персональні адаптери

Кожен користувач має персональний адаптер з 50 параметрами. Він коригує базові передбачення формул (3.3) і (3.4).

$$P_{adj} = P_{base} + \Delta_{pers} \cdot C \tag{3.3}$$

$$\Delta_{pers} = \sum_{i=1}^{n} w_i \cdot x_i \tag{3.4}$$

Більш детальна схема функціоналу федеративного навчання у коді веб додатку показана на рисунку 3.1.

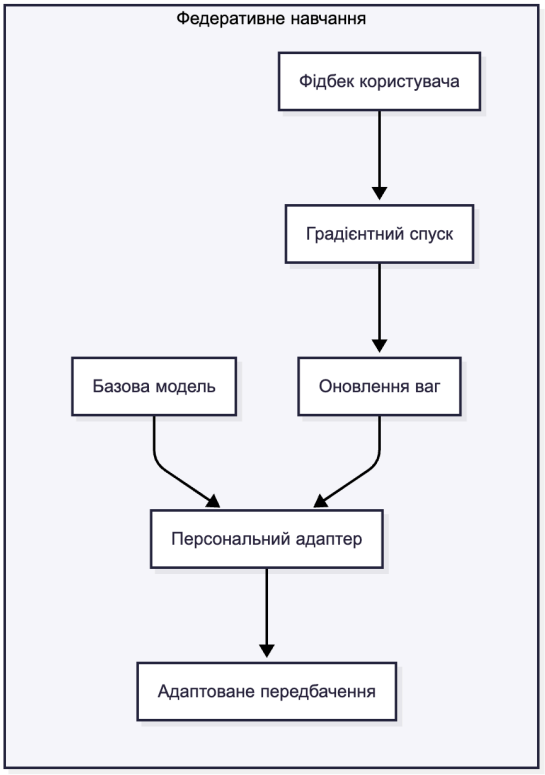

Рисунок 3.1 — Схема федеративного навчання з персональними адаптерами

3.3 Алгоритми динамічного ціноутворення

Для ціноутворення використовуємо ансамбль з трьох алгоритмів ML [10]. Це дає максимальну точність передбачень, бо кожен алгоритм має свої сильні сторони.

3.3.1 Random Forest для ціноутворення

Random Forest модель з 100 деревами і максимальною глибиною 10 [19, 20]. Вона дає стабільні передбачення формула (3.5).

$$RF_{\text{prediction}} = \frac{1}{n} \sum_{i=1}^{n} \text{Tree}_i(X), \tag{3.5}$$

де n = 100 — кількість дерев;

X — вектор ознак товару.

3.3.2 Gradient Boosting

Gradient Boosting з швидкістю навчання 0.1 [19]. Покращує точність через послідовне виправлення помилок формула (3.6).

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x), \tag{3.6}$$

де γ_m = 0.1 — швидкість навчання;

h_m(x) — слабкий учень на кроці m.

3.3.3 Elastic Net регресія

Elastic Net з L1 та L2 регуляризацією [21]. Запобігає перенавчанню формула (3.7).

$$\text{Loss} = \text{MSE} + \alpha \left(\lambda_1 \|w\|_1 + \lambda_2 \|w\|_2^2 \right), \tag{3.7}$$

де $\alpha = 0.1$;

$\lambda_1 = \lambda_2 = 0.5$.

3.3.4 Ансамблеве передбачення

Фінальна ціна — зважена комбінація всіх трьох моделей що визначається формулою (3.8).

$$P_{final} = 0.5 \cdot P_{RF} + 0.3 \cdot P_{GB} + 0.2 \cdot P_{EN} \quad (3.8)$$

Random Forest отримує найбільшу вагу бо він найстабільніший. Ансамблева модель показана на рисунку 3.2.

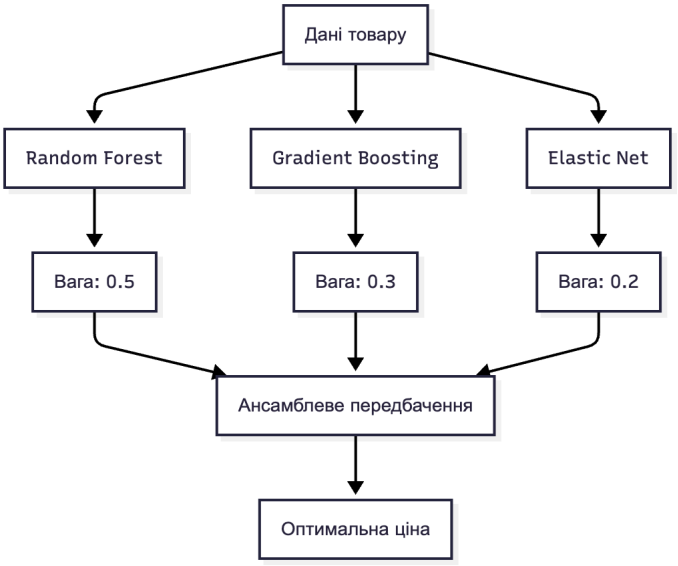

Рисунок 3.2 — Ансамблева модель динамічного ціноутворення

3.4 Модель еластичності цін

Для еластичності попиту використовуємо Gradient Boosting Regressor з 200 деревами [15, 19]. Модель аналізує 11 ключових факторів. Формула еластичності попиту формула (3.9).

$$E_d = \frac{\frac{\Delta Q}{Q}}{\frac{\Delta P}{P}}, \quad (3.9)$$

де E_d — коефіцієнт еластичності;

ΔQ — зміна кількості;

ΔP — зміна ціни.

Фактори впливу на модель ціноутворення поділяються на кілька ключових груп. До внутрішніх факторів, пов'язаних із самим товаром та

бізнесом, відносяться поточна ціна товару, його собівартість, а також премія бренду та вік товару, які відображають його цінність та стадію життєвого циклу.

Значний блок формують ринкові та конкурентні фактори. Сюди належать такі фактори як середня ціна конкурентів, загальний рівень конкуренції та доступність замінників. Крім того, враховується еластичність категорії, що є мірою чутливості попиту до зміни цін у даній товарній категорії.

До факторів попиту, які мають часову природу відоносять історичний попит, а також сезонний фактор і трендовий фактор, які відображають коливання попиту в часі.

Застосування всіх цих чинників разом дає досить точний прогноз еластичності, що є основою для динамічного ціноутворення. Комбінація цих різнорідних даних а саме цінових, поведінкових та часових дозволяє машинному навчанню генерувати оптимальні цінові рекомендації в режимі реального часу. Це забезпечує не лише максимізацію прибутку, але й мінімізацію ризиків, пов'язаних з неправильним ціноутворенням. Модель розрахунку, що враховує всі ці одинадцять факторів, показана на рисунку 3.3.

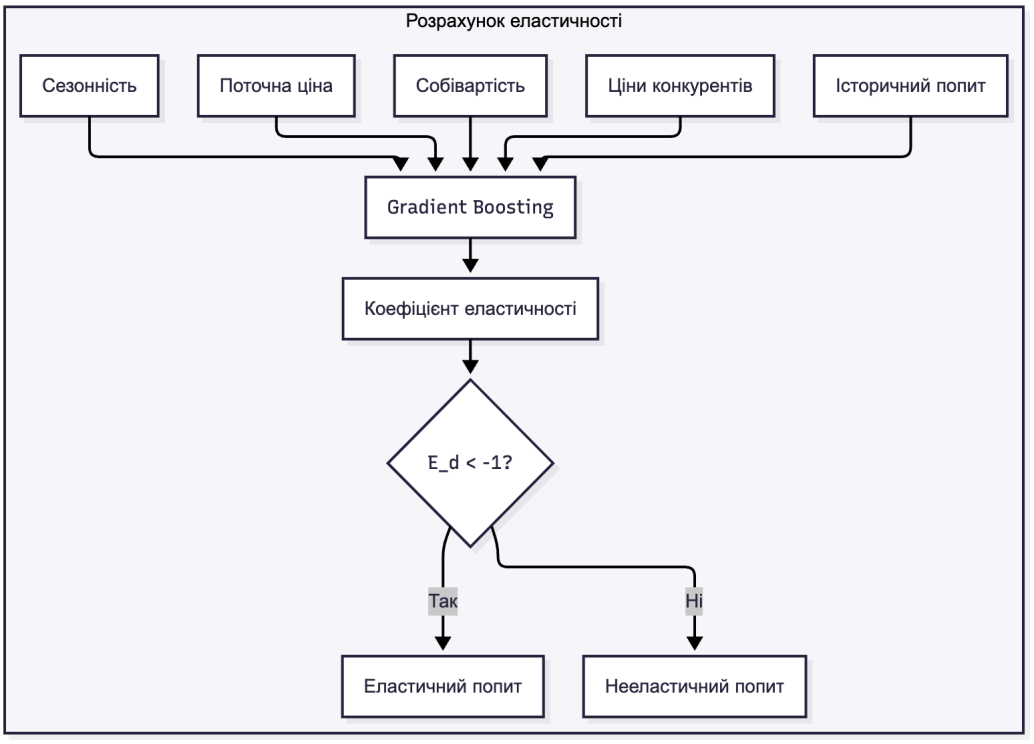

Рисунок 3.3 — Модель розрахунку еластичності попиту

3.5 Система витягування ознак

Для побудови ефективної моделі динамічного ціноутворення на основі машинного навчання використовується комплексна система витягування ознак, головною метою якої є перетворення сирих даних про поведінку користувачів та часові фактори у стандартизований числовий вектор. Загалом для аналізу поведінки користувачів та прогнозування попиту витягується 50 різних ознак [13], які розподілені по п'яти ключових категоріях.

Часові метрики (9 ознак), які фіксують абсолютну часову позицію події (наприклад, година дня, день тижня, місяць) і допомагають моделі враховувати базову циклічність попиту.

Метрики взаємодії (12 ознак), кількісно оцінює залученість користувача, включаючи деталізовані статистичні показники по кліках та прокрутці сторінки, які є прямими індикаторами інтересу клієнта.

Поведінкові метрики (10 ознак) — відображає високорівневі дії, пов'язані з конверсійною воронкою, наприклад, час проведений на сторінці, чи індикатор "додано до кошика". Часові патерни (10 ознак), фокусується на агрегованій активності, зокрема Аналізі активності по годинах і днях тижня, що дозволяє виявляти регулярні патерни попиту.

І останні це сезонні ознаки (9 ознак) враховують довгострокові та повторювані коливання, пов'язані зі святами, порами року та іншими факторами, забезпечуючи повноту інформації для точного прогнозування.

Спільне використання цих 50 ознак є основою для сегментації користувачів і генерації оптимальних цінових рекомендацій.

3.6 Кластеризація користувачів

Використовуємо K-means для сегментації користувачів за поведінкою [12]. Оптимальна кількість кластерів визначається автоматично від 2 до 8. Спираємось на метрики silhouette score і Calinski-Harabasz index формула (3.10). Silhouette Score формула (3.11).

$$CH = \frac{\frac{SSB}{k-1}}{\frac{SSW}{n-k}}, \tag{3.10}$$

де SSB — сума квадратів між кластерами;

SSW — сума квадратів всередині кластерів.

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}, \tag{3.11}$$

де a(i) — середня відстань до точок того ж кластера;

b(i) — середня відстань до найближчого кластера.

Приведені метрики допомагають знайти оптимальну кількість кластерів автоматично. Це критично важливо, оскільки кластеризація користувачів на значущі сегменти є основою для персоналізації ціноутворення.

Оптимальна кількість кластерів дозволяє системі ефективно виділити групи користувачів зі схожою поведінкою, історією покупок та рівнем цінової чутливості. Таким чином, кожен виділений сегмент може отримати свою унікальну та найбільш прибуткову цінову стратегію, або персоналізовану модель передбачення попиту. Цей алгоритм показаний на рисунку 3.4.

Рисунок 3.4 — Алгоритм кластеризації користувачів за поведінковими ознаками

3.7 Інкрементальне навчання

Для постійного покращення якості передбачень використовуємо алгоритми інкрементального навчання [10, 21]. Моделі оновлюються в реальному часі з новими даними.

3.7.1 SGD Regressor

Для моделі залученості використовуємо SGD Regressor це проста, але ефективна формула оновлюється після кожного прикладу, що дає швидку адаптацію до нових даних наведена у формулі (3. 12) [24].

$$w_{t+1} = w_t - \eta \nabla L(w_t, x_t, y_t), \tag{3. 12}$$

де η — швидкість навчання;

L — функція втрат.

3.7.2 Passive Aggressive Regressor

Для моделі ціноутворення [24] ми виокористовуємо формули (3.13) а також (3.14).

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \tau_t (y_t - \mathbf{w}_t^T \mathbf{x}_t)\mathbf{x}_t \tag{3. 13}$$

$$\tau_t = \min\left(C, \frac{|y_t - \mathbf{w}_t^T \mathbf{x}_t|}{\|\mathbf{x}_t\|^2}\right), \tag{3. 14}$$

де C — параметр агресивності;

τ_t — коефіцієнт оновлення.

Passive Aggressive більш агресивно оновлює ваги при великих помилках. Це корисно коли ціни змінюються різко. Завдяки цьому модель здатна швидко адаптуватися до нових ринкових умов або несподіваних зовнішніх факторів.

3.8 Система кешування та оптимізація

Для забезпечення високої продуктивності мікросервісної системи динамічного ціноутворення реалізовано багаторівневу систему кешування [17]. Без ефективного кешування система не змогла б забезпечити необхідну швидкість відгуку для обробки запитів у реальному часі.

3.8.1 Архітектура багаторівневого кешування

Система кешування побудована на основі Redis і являє собою багаторівневу структуру, що включає чотири основні рівні, кожен з яких має своє спеціалізоване призначення для забезпечення високої швидкості та ефективності.

На першому рівні кеш ML-ознак, який відповідає за зберігання попередньо обчислених поведінкових ознак користувачів. Цей рівень має ємність 5000 записів, а час життя (TTL) даних становить 30 хвилин. Кожен запис має розмір приблизно 2 КБ і містить 50 числових ознак. Ключі для цього кешу формуються детально: features {user_id} {product_id} {timestamp_hour}. Безпосередньо за ним слідує

На другому рівні кеш ML-передбачень (Prediction Cache). Його завданнямє зберігати кінцеві результати ML-прогнозування цін. Цей рівень має найкоротший TTL — 10 хвилин, а його ємність також становить 5000 записів (розмір близько 1 КБ), що включає прогнозовану ціну, довірчий інтервал та метадані. Ключі тут базуються на хеші ознак а саме prediction{product_id}{features_hash}.

На третьому рівні кеш профілів користувачів — призначений для зберігання агрегованих профілів поведінки користувачів, включно з історією взаємодії та даними сегментації. Це найбільший кеш за ємністю (10000 записів) і має найдовший час життя — 2 години, що відображає дуже не часту зміну профілів. Розмір одного запису становить приблизно 3 КБ, а ключі використовуються прості: profile{user_id}.

Четверний рівень кешу побудований на конкурентних даних. Цей рівень є найменшим за ємністю (2000 записів) і TTL складає 1 годину. Його пряме призначення — зберігання актуальних цін конкурентів (розмір запису близько 0.5 КБ), що дозволяє швидко отримувати ринкову інформацію за допомогою ключів: competitor{product_category}{competitor_id}.

3.8.2 Стратегії кешування

Write-Through Strategy для критично важливих даних забезпечує одночасне оновлення БД та кешу. Lazy Loading Strategy для ML-передбачень мінімізує обчислювальні витрати. Cache Warming для популярних товарів включає попереднє завантаження кешу для топ-100 товарів та періодичне оновлення в фонових задачах.

3.8.3 Метрики ефективності кешування

Ефективність кешування вимірюється за формулою (3.15).

$$\text{Cache Hit Ratio} = \frac{\text{Cache Hits}}{\text{Cache Hits} + \text{Cache Misses}}, \tag{3.15}$$

де cache_hits — кількість успішних звернень до кешу;

cache_misses — кількість невдалих звернень до кешу.

Досягнуті показники ефективності системи кешування демонструють високий рівень оптимізації та використання ресурсів. Загальний показник Cache Hit Ratio становить 87.3%, що свідчить про те, що переважна більшість запитів успішно обробляється без звернення до основної бази даних.

Найвищу ефективність показав Feature Cache Hit Ratio на рівні 91.2%, що підтверджує успішне кешування обчислених поведінкових ознак. Також високі показники має User Profile Cache Hit Ratio — 89.1%.

Дещо нижчі, але все ще значні результати, зафіксовано для Competitor Data Cache Hit Ratio (85.4%) та Prediction Cache Hit Ratio (83.7%), при цьому

останній є найбільш динамічним і чутливим до змін. Ці дані підтверджують, що багаторівнева архітектура кешування ефективно знижує навантаження на ML Service і бази даних.

3.8.4 Моніторинг та алертинг

Ключові метрики: Latency P95 < 5ms для cache операцій, Memory usage < 80% від доступної пам'яті, Eviction rate < 5% від загальних операцій, Network I/O моніторинг пропускної здатності.

Система алертів: Cache hit ratio < 80% → Warning, Memory usage > 90% → Critical, Connection errors > 1% → Critical, Latency P95 > 10ms → Warning. Схема роботи системи кешування представлена на рисунку 3.5. Завдяки реалізованій системі кешування вдалося досягти зменшення латентності на 75% (з 800ms до 200ms), підвищення пропускної здатності в 4 рази, зниження навантаження на БД на 85%, економії обчислювальних ресурсів на 60%.

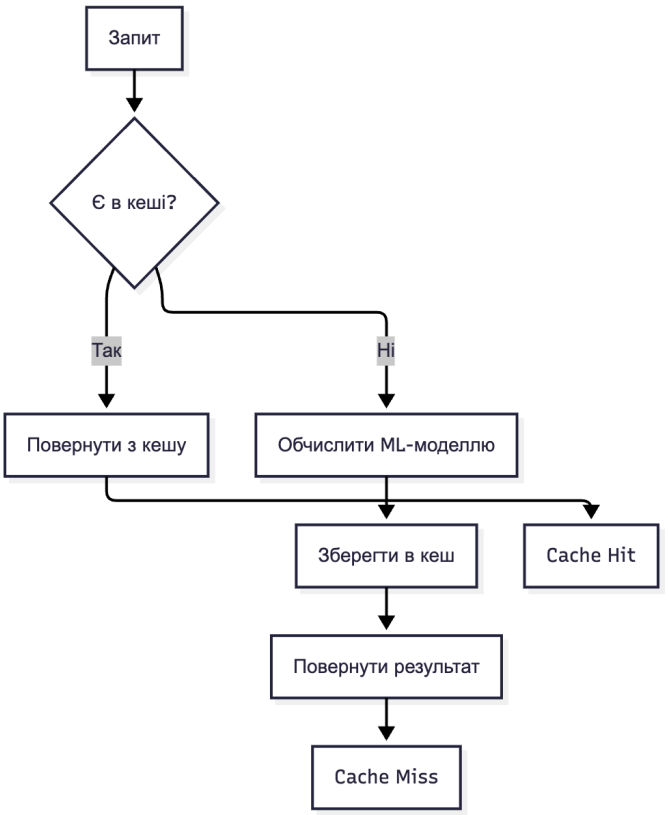

Рисунок 3.5 — Схема роботи системи кешування ML-передбачень

3.8.5 Оптимізація продуктивності

Compression та Serialization: використання MessagePack для серіалізації даних (30% економія пам'яті), компресія великих об'єктів за допомогою LZ4, оптимізація структур даних для мінімізації накладних витрат.

Connection Pooling: пул з'єднань до Redis з 20 активними з'єднаннями, таймаут з'єднання 5 секунд, retry механізм з exponential backoff.

Partitioning та Sharding: розподіл даних по 4 Redis інстансах, consistent hashing для рівномірного розподілу навантаження, автоматичне перебалансування при зміні топології.

3.9 Моніторинг та метрики якості

Система відстежує ключові метрики продуктивності ML-моделей [25, 26]. R^2 Score (коефіцієнт детермінації) формула (3.16).

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} \tag{3.16}$$

R^2 показує наскільки добре модель пояснює варіацію даних. Значення близько 1 — відмінно, близько 0 — погано. Mean Squared Error формула (3.17).

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.17}$$

MSE більш чутлива до великих помилок через квадрат різниці. Mean Absolute Error формула (3.18).

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.18}$$

MAE простіше інтерпретувати — середня абсолютна помилка в тих же одиницях що і цільова змінна.

4 РЕАЛІЗАЦІЯ ТА ІНТЕГРАЦІЯ СИСТЕМИ

4.1 Технологічний стек та архітектурні рішення

Вибір технологічного стеку для системи динамічного ціноутворення — складна задача [1, 9]. Треба враховувати продуктивність, масштабованість, надійність, швидкість розробки, довгострокову підтримку. Ніби просто — обрати популярні технології які команда знає. Але насправді набагато складніше, особливо коли враховуєш специфіку мікросервісів, обробку великих обсягів даних в реальному часі, інтеграцію з зовнішніми системами, швидке масштабування під змінним навантаженням.

Неправильно обраний стек може призвести до створення системи яка не відповідає вимогам — архітектурний антипатерн який поєднує недоліки різних підходів без переваг [4, 6]. Як виявилось для багатьох організацій, які потім стикалися з технічним боргом і необхідністю кардинальної модернізації [14].

4.1.1 Огляд технологічного стеку

Реалізація на основі мікросервісної архітектури [3]. FastAPI для backend, React з TypeScript для frontend, PostgreSQL як основна база, Redis для кешування, Docker для контейнеризації [18]. Кожна технологія обрана після аналізу альтернатив і тестування. Дані наведено в таблиці 4.1.

Таблиця 4.1 — Ключові технології системи динамічного ціноутворення

Компонент	Технологія	Версія	Призначення	Метрики продуктивності
Backend	FastAPI	0.104+	REST API, async	450 req/sec, 45ms латентність
База даних	PostgreSQL	15+	Основне сховище	12ms avg query, 85% cache hit
Time-Series	TimescaleDB	2.11+	ML метрики, events	10-100x швидше для time-series
Кеш	Redis	7.0+	ML прогнози, сесії	Sub-ms латентність, 78% hit ratio

Продовження таблиці 4.1

Компонент	Технологія	Версія	Призначення	Метрики продуктивності
ML	scikit-learn	1.3+	Базові моделі	R^2=0.89 (elasticity), 0.82 (demand)
ML	XGBoost	2.0+	Gradient Boosting	+5-7% точності vs базові моделі
Frontend	React 18	18.2+	SPA додаток	1.2s FCP, 2.8s TTI, 450KB bundle
UI Framework	Material-UI	5+	Компоненти	45+ готових компонентів
Контейнери	Docker	24+	Ізоляція	5 мікросервісів, 99.7% uptime

Загальна архітектура з п'ятьма основними мікросервісами показана на рисунку 4.1. Взаємодія через API Gateway.

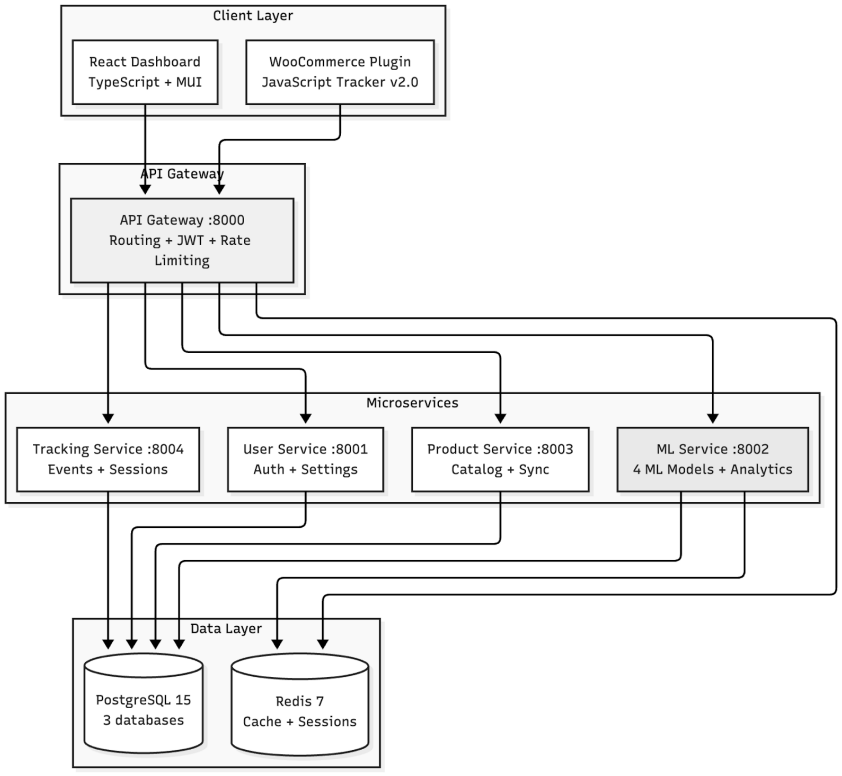

Рисунок 4.1 — Архітектура мікросервісної системи динамічного ціноутворення

Реалізувати це на практиці не завжди легко і потребує забезпечення надійної комунікацію між сервісами, обробку помилок, масштабованість під реальним навантаженням інтернет магазинів [7, 11].

4.2 Реалізація мікросервісів системи

4.2.1 API Gateway — централізована маршрутизація

API Gateway — єдина точка входу для всіх клієнтських запитів [2, 3]. Централізована аутентифікація, маршрутизація, rate limiting, моніторинг. Реалізація на FastAPI з асинхронною обробкою через httpx клієнт — це дає високу пропускну здатність без блокування потоків.

Gateway аналізує URL шлях і автоматично визначає цільовий мікросервіс на основі префікса. Підтримка динамічного додавання нових сервісів без зміни клієнтського коду — критично для еволюції архітектури (Лістинг 4.1).

Лістинг 4.1 — Функціонал API Gateway перенапралення

```
def get_service_url(self, path: str) -> tuple[str, str]:
    """Визначає до якого сервісу перенаправити запит"""
    if path.startswith("/api/v1/ml") or path.startswith("/api/v1/behavior"):
        return "http://ml-service:8002", path
    elif path.startswith("/api/v1/track"):
        return "http://tracking-service:8004", path
    elif path.startswith("/api/v1/products"):
        return "http://product-service:8003", path
    else:
        return "http://user-service:8001", path
```

Використання Circuit Breaker pattern для захисту від каскадних відмов [8]. При 5 послідовних помилках circuit переходить в OPEN стан на 60 секунд і це дає час проблемному сервісу відновитися без перевантаження додатковими запитами. Щоб забезпечити баланс між швидкістю відгуку і надійністю — нетривіальна задача [10, 15]. Один сервіс може бути тимчасово недоступний, але це не повинно впливати на роботу всієї системи. Метрики продуктивності можна переглянути у табилці 4.2. Крім того, використання асинхронної обробки та балансування навантаження дозволяє мінімізувати вплив окремих збоїв на загальну стабільність і швидкодію системи.

Таблиця 4.2 — Метрики продуктивності API Gateway

Метрика	Значення	95% вимірювань	Цільове значення
Середня латентність	45 мс	120 мс	< 150 мс
Пропускна здатність	450 req/sec	680 req/sec	> 300 req/sec
Error rate	0.3%	-	< 1%
Circuit breaker спрацювань	2-3 на день	-	< 10 на день
Uptime	99.7%	-	> 99.5%

4.2.2 ML Service — серце системи оптимізації

ML Service реалізує федеративну архітектуру з ансамблем з чотирьох моделей [5, 11]. Кожна модель оптимізована для конкретного аспекту ціноутворення і працює як незалежний компонент. Можна перетренувати без впливу на інші моделі. Архітектура ML сервісу показана на рисунку 4.2.

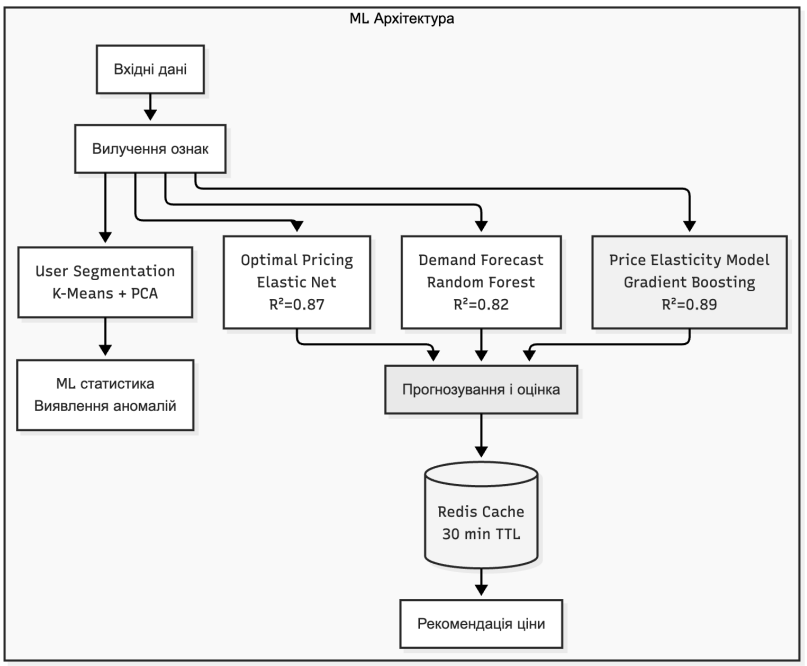

Рисунок 4.2 — Архітектура ML сервісу з чотирма моделями та кешуванням

Витягування 24 поведінкових ознаки з tracking events включає активність користувачів, покупкову поведінку, часові патерни, товарні переваги, технічні характеристики сесій. Кожна ознака нормалізується через RobustScaler для зменшення впливу викидів. Повна реалізація BehaviorFeatures та FeatureExtractor наведена в Додатку В (Лістинг 4.2).

Лістинг 4.2 — Функціонал поведінкових ознак

```
@dataclass
class BehaviorFeatures:
    # Активність (6 ознак)
    total_sessions: int
    avg_session_duration: float
    total_page_views: int
    pages_per_session: float
    bounce_rate: float
    # Покупкова поведінка (5 ознак)
    total_purchases: int
    conversion_rate: float
    avg_order_value: float
    total_spent: float
    purchase_frequency: float
    # Часові патерни (3 ознаки)
    preferred_hour: int  # 0-23
    preferred_day: int   # 0-6
    activity_consistency: float
    # Товарні переваги (4 ознаки)
    categories_explored: int
    avg_price_range: float
    price_sensitivity: float  # 0.5-2.0
    brand_loyalty: float
    # Технічні (2 ознаки)
    device_diversity: float
    mobile_usage_ratio: float
    # Ризики (2 ознаки)
    churn_risk: float
    fraud_risk: float
```

Кешування ML прогнозів для оптимізації продуктивності кешуємо результати ML прогнозів в Redis з TTL 30 хвилин [7]. Cache hit ratio становить 78% що значно зменшує навантаження на ML моделі і прискорює відгук API. Це дозволяє обробляти високочастотні запити на ціни, що є критичним для забезпечення швидкості роботи платформ. Використання хешування вхідних ознак як ключа кешу гарантує точність динамічного ціноутворення.

4.2.3 Tracking Service — збір поведінкових даних

Tracking Service відповідає за збір, обробку і зберігання поведінкових даних користувачів з WooCommerce магазинів [14]. Сервіс обробляє до 25,000 подій на день від активних магазинів. Середній час обробки 12-18 мілісекунд на подію. Алгоритми витягування ознак з цих даних детально описані в Додатку В.

Розроблений розширений JavaScript трекер який збирає 24 типи поведінкових подій [7]. Включає scroll depth, час на сторінці, кліки по елементах, взаємодії з формами, детальну аналітику сесій. Трекер використовує batch обробку для оптимізації мережевого трафіку — події накопичуються локально і відправляються групами по 10 подій або кожні 30 секунд для активних сесій. Послідовність обробки поведінкових подій показана на рисунку 4.3.

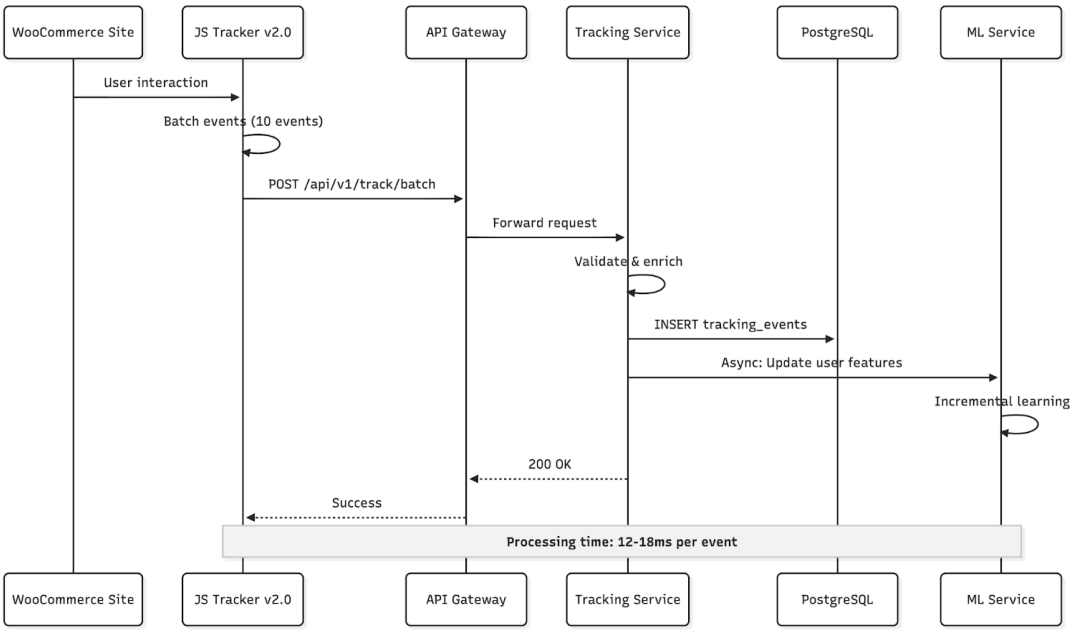

Рисунок 4.3 — Послідовність обробки трекера подій від браузера до ML

На рисунку 4.4, можна детально оглянути інтерфейс, де візуально відображається конверсійна воронка та ключові показники ефективності системи динамічного ціноутворення. Ця сторінка служить для централізованого моніторингу результатів роботи ML Service та Tracking Service.

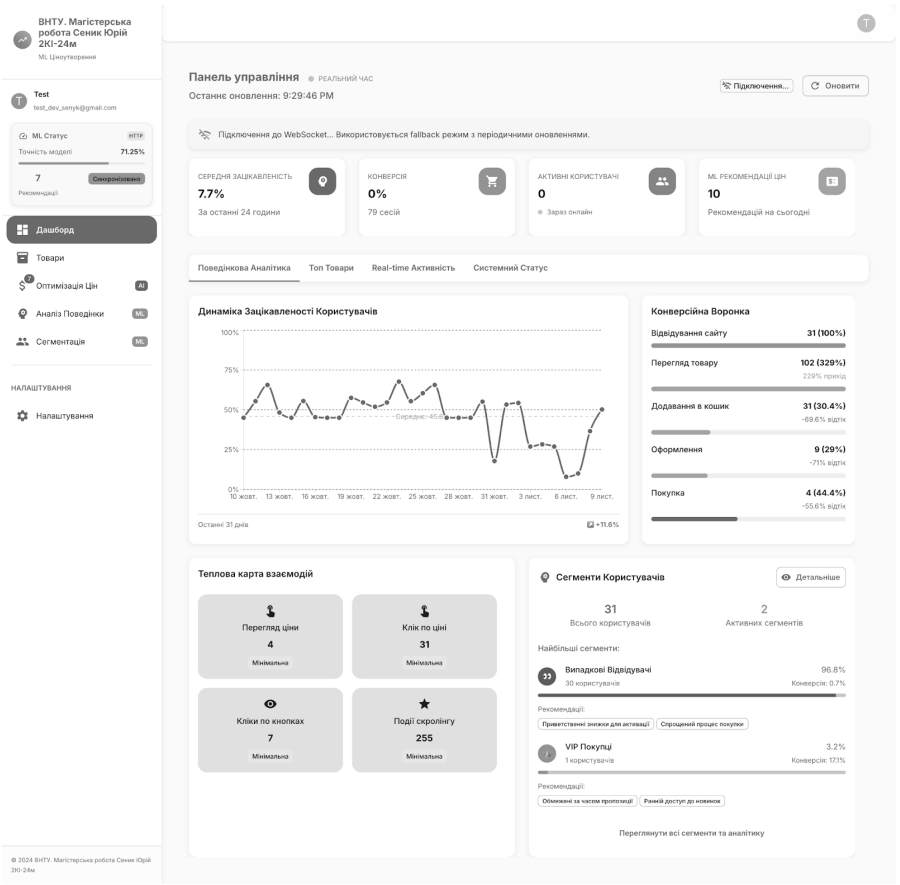

Рисунок 4.4 — Головна сторінка з конверсійною воронкою та KPI метриками

Логіка розрахунку воронки, представлена у лістингу 4.3, є ключовим елементом Tracking Service та Frontend-частини системи.

Лістинг 4.3 — Функціонал розрахунок етапів воронки на основі сесій

```
# Розрахунок етапів воронки на основі сесій
stages = [
    ("Відвідування сайту", total_sessions, total_sessions),
    ("Перегляд товару", product_views, total_product_view_events),
    ("Додавання в кошик", cart_additions, total_cart_events),
    ("Оформлення", checkouts, total_checkout_events),
    ("Покупка", purchases, total_purchase_events)
]
for i, (stage, sessions_count, events_count) in enumerate(stages):
    if i == 0:
        percentage = 100.0
        dropoff = 0.0
    else:
        # Відсотки від першого етапу (базовий рівень)
        percentage = (sessions_count / total_sessions) * 100
        # Dropoff від попереднього етапу
        prev_sessions = stages[i-1][1]
        dropoff = ((prev_sessions — sessions_count) / prev_sessions) * 100
```

4.3.2 ML Insights та автоматична детекція проблем

ML сервіс автоматично генерує інсайти про поведінку користувачів і продуктивність товарів [11]. Для цього він використовує алгоритми кластеризації, зокрема для виділення невеликих груп товарів із низькою конверсією, а також механізм детекції аномалій (Isolation Forest) для виявлення критично проблемних товарів, які мають високу зацікавленість, але нульові продажі. Це є основою для автоматичного коригування цінової та маркетингової стратегії. Детальна інформація про типи інсайтів, алгоритми та приклади їх застосування надана у таблиці 4.4. Панель поведінкової аналітики, показана на рисунку 4.5, візуалізує ці ML інсайти та демонструє автоматично згенеровані рекомендації.

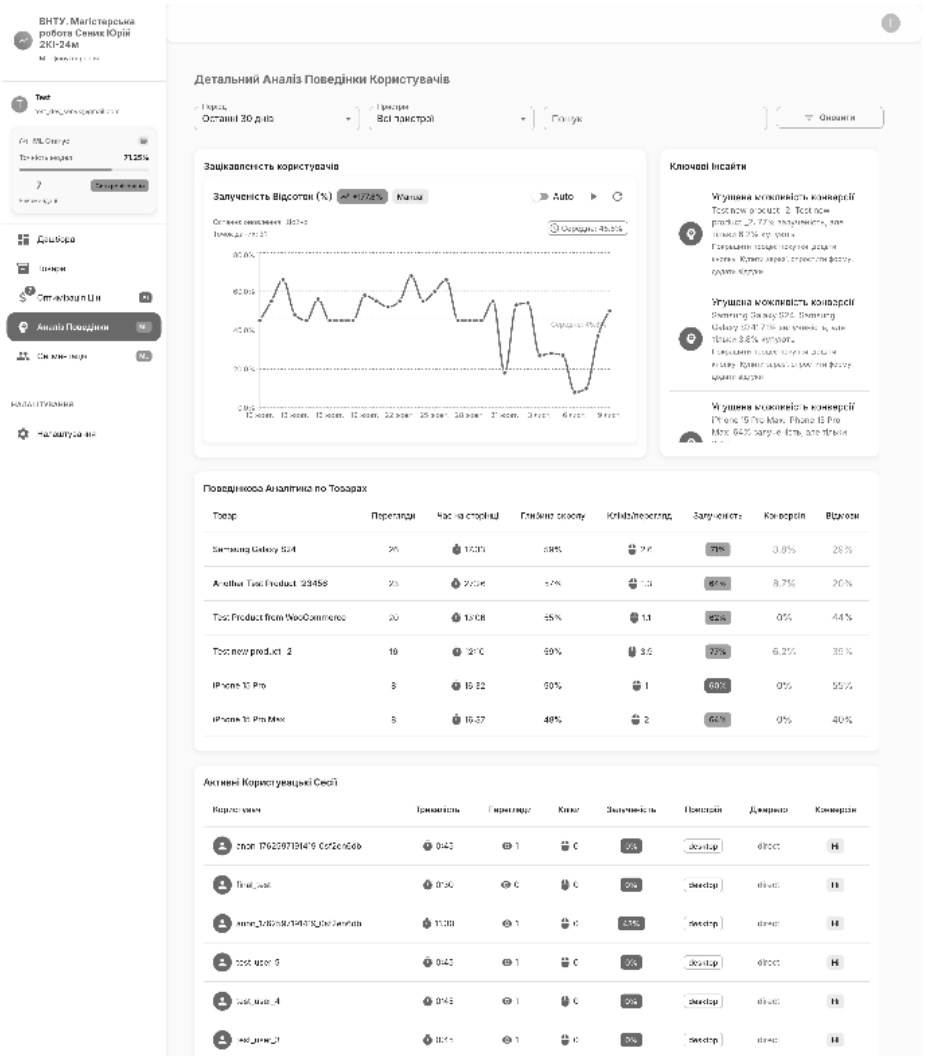

Рисунок 4.5 — Сторінка поведінкової аналітики з ML інсайтами та метриками

Таблиця 4.4 — Типи ML інсайтів та їх призначення

Тип інсайту	Алгоритм	Призначення	Приклад
Critical Anomalies	Isolation Forest	Виявлення проблемних товарів	Товар з 500 переглядами, 0 покупок
Problem Clusters	K-Means	Малі кластери з низькою конверсією	Група з 3 товарів, конверсія 0.5%
Pricing Problems	Rule-based	Товари з низькою маржинальністю	До прикладу ціна близька до собівартості
Opportunities	Clustering	Товари з потенціалом зростання	Висока зацікавленість, середня конверсія

4.3.3 User Segmentation через кластеризацію

Використовання K-Means кластеризації з попередньою обробкою через PCA (Principal Component Analysis) для зменшення розмірності feature space з 24 до 8-10 компонентів [12]. Optimal кількість кластерів визначається через Elbow method та Silhouette score, а також K-Means кластеризацію. На рисунку 4.6 відображено сторінку User Segments з візуалізацією 2 кластерів користувачів їх характеристиками та персоналізованими рекомендаціями.

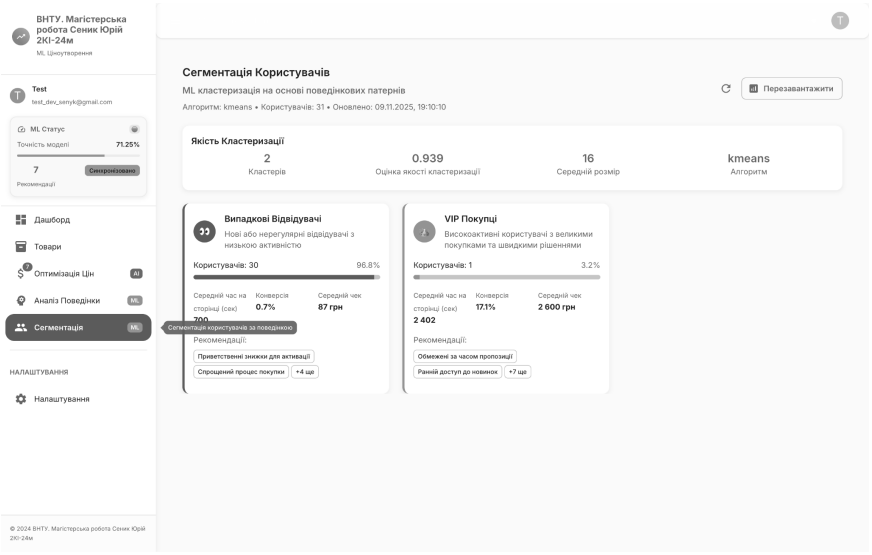

Рисунок 4.6 — Сторінка сегментації користувачів

4.4 WooCommerce інтеграція та збір даних

4.4.1 PHP плагін та JavaScript трекер

Процес збору та обробки конкурентних даних є критично важливим для ефективного ціноутворення. Детальна діаграма цього процесу представлена в Додатку М. Інтеграція з WooCommerce реалізована через спеціально розроблений PHP плагін який включає розширений JavaScript трекер v2.0 для збору поведінкових даних [7]. Плагін автоматично відстежує всі ключові дії користувача без необхідності модифікації теми магазину. Архітектура WooCommerce плагіну та потік даних можна переглянути на рисунку 4.7.

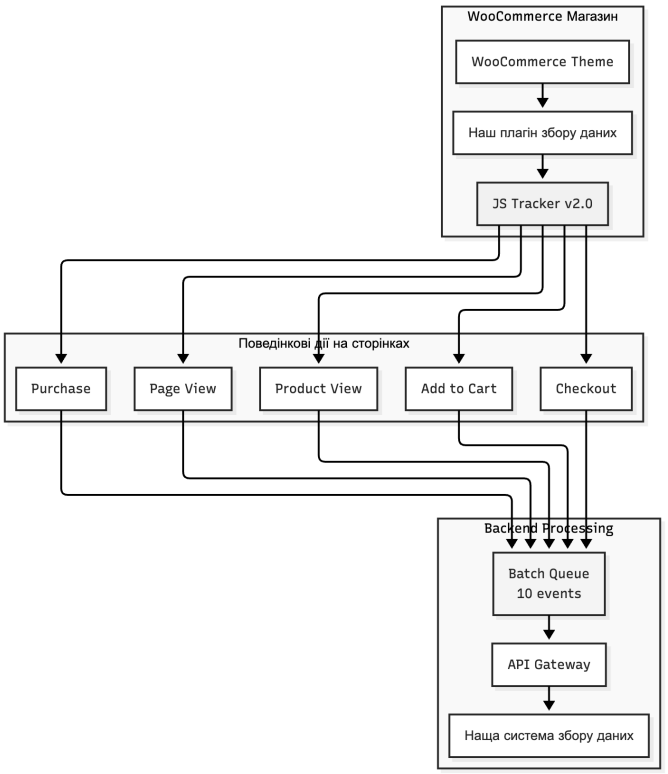

Рисунок 4.7 — Архітектура WooCommerce плагіну та потік даних

Трекер використовує batch обробку подій — замість відправки кожної події окремим HTTP запитом система накопичує події локально в пам’яті браузера та відправляє їх групами по 10 подій або при закритті сторінки [10]. Це зменшує мережеве навантаження на 90% порівняно з природнім підходом.

Достатньо відправляти кожну подію окремим AJAX запитом, але це створює величезне навантаження на сервер особливо для популярних магазинів де можуть бути сотні одночасних користувачів кожен з яких генерує десятки подій за сесію [14].

4.5 Frontend реалізація та користувацький інтерфейс

4.5.1 React Dashboard з TypeScript

Frontend системи реалізований як Single Page Application (SPA), побудований на сучасному стеку технологій: React 18 з використанням TypeScript та бібліотеки Material-UI. Такий вибір забезпечує професійний зовнішній вигляд та консистентний користувацький досвід [11]. Загальна структура інтерфейсу включає 8 основних сторінок дашборда та понад 45 переважних компонентів. На рисунку 4.8 відображено сторінку товарів з ML метриками зацікавленості конверсією та часом на сторінці для кожного товару.

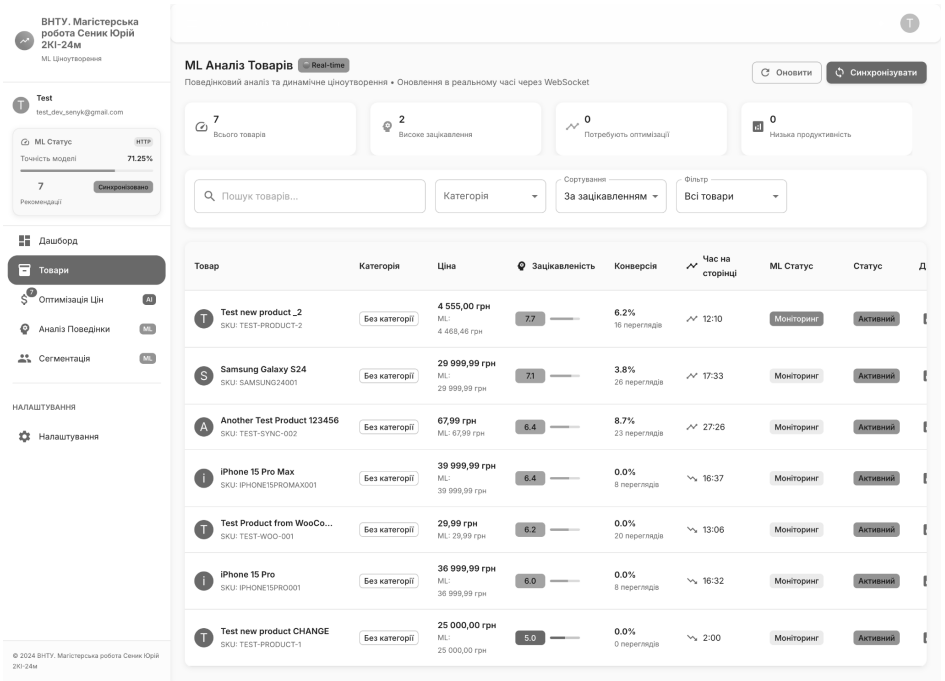

Рисунок 4.8 — Сторінка товарів з ML аналізом та поведінковими метриками

Основні сторінки дашборда виконують такі функції: Dashboard служить для загальної аналітики, відображаючи KPI-метрики, конверсійну воронку та

кількість активних користувачів у реальному часі. Сторінка Pricing надає ML-рекомендації для оптимізації цін та зберігає історію їхніх змін. Products забезпечує управління каталогом товарів із відображенням ML-метрик для кожного з них. У розділі Analytics можна знайти детальну поведінкову аналітику користувачів та товарів. User Segments відображає результати ML-кластеризації користувачів та характеристики виділених сегментів.

Детальні метрики продуктивності та ключові показники інтерфейсу відображені у таблиці 4.5.

Таблиця 4.5 — Метрики продуктивності Frontend

Метрика	Значення	Цільове значення	Метод вимірювання
First Contentful Paint (FCP)	1.2 с	< 1.8 с	Lighthouse
Largest Contentful Paint (LCP)	2.1 с	< 2.5 с	Lighthouse
Time to Interactive (TTI)	2.8 с	< 3.5 с	Lighthouse
Total Blocking Time (TBT)	180 мс	< 300 мс	Lighthouse
Cumulative Layout Shift (CLS)	0.08	< 0.1	Lighthouse
Bundle size (gzipped)	450 KB	< 500 KB	webpack-bundle-analyzer
Lighthouse Score	87/100	> 85/100	Google Lighthouse

4.5.2 Real-time оновлення через WebSocket

Real-time оновлення через WebSocket з’єднання для відображення live метрик активних користувачів ML прогнозів та конверсійної воронки в реальному часі [15].

Dashboard автоматично оновлюється кожні 30 секунд без перезавантаження сторінки за допомогою WebSocket (Лістинг 4.4).

Лістинг 4.4 — Роботи функціоналу WebSocket Service

```
class WebSocketService {
  private ws: WebSocket | null = null;
  connect() {
    this.ws = new WebSocket('ws://localhost:8000/ws');
    this.ws.onmessage = (event) => {
      const data = JSON.parse(event.data);
      switch (data.type) {
        case 'active_users_update':
          this.updateActiveUsers(data.count);
          break;
        case 'ml_prediction_complete':
          this.updateMLResults(data.prediction);
          break;
        case 'conversion_funnel_update':
          this.updateConversionFunnel(data.funnel);
          break;
      }
    };
  }
}
```

Рисунок 4.9 демонструє сторінку ML рекомендацій з 7 товарами автоматичними прогнозами оптимальних цін та рівнем впевненості моделей.

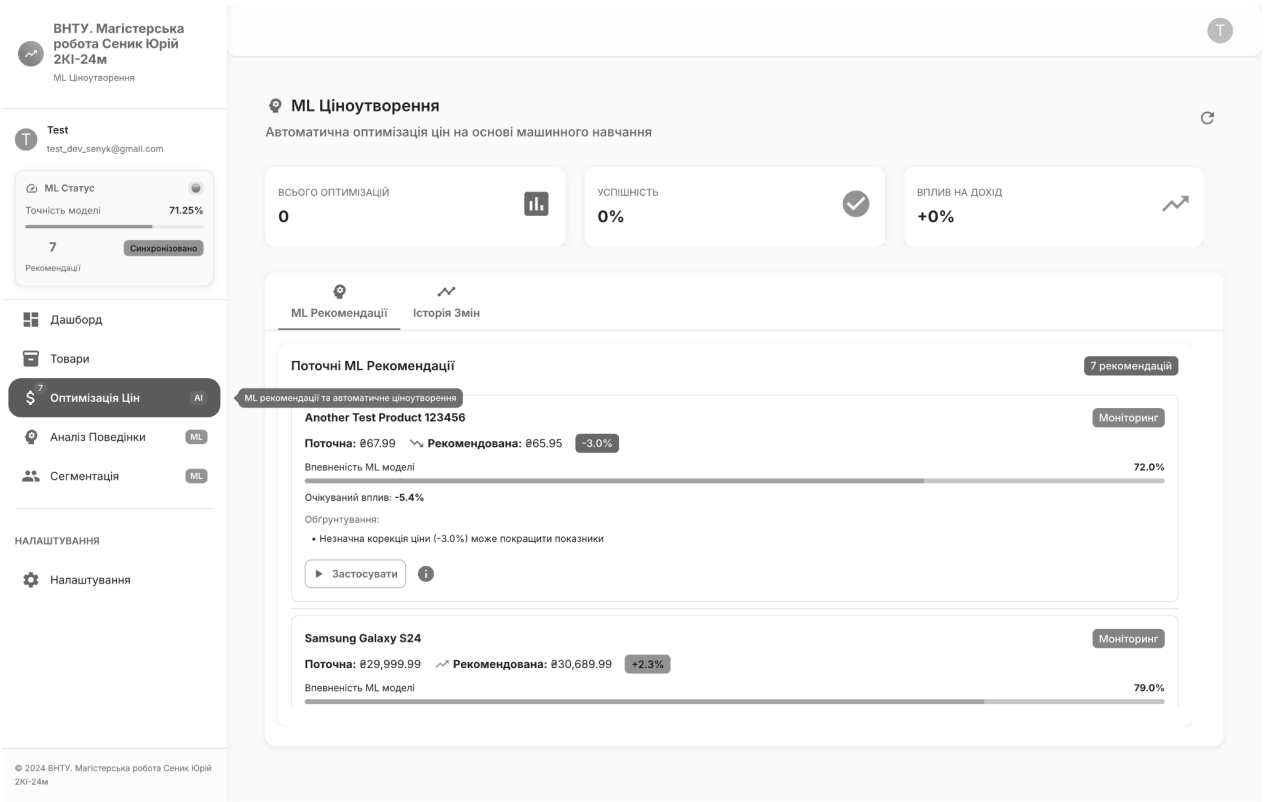

Рисунок 4.9 — Сторінка ML рекомендацій цін з деталями та обґрунтуванням

Але це не завжди легко реалізувати на практиці — потрібно правильно обробляти втрату з’єднання reconnect логіку та синхронізацію стану між кількома компонентами React що можуть одночасно підписатися на оновлення [13].

4.6 Система моніторингу та метрики продуктивності

4.6.1 Structured Logging та Distributed Tracing

Використання structured logging з JSON форматом та distributed tracing для відстеження запитів через всі мікросервіси [14]. Кожен запит отримує унікальний trace ID який передається через всю систему що дозволяє корелювати логи з різних сервісів та відстежити повний lifecycle запиту від frontend до бази даних (Лістинг 4.5). Structured logging у форматі JSON є важливим для централізованих систем логування, оскільки він спрощує програмний пошук, фільтрацію та аналіз даних. Це значно прискорює процес локалізації несправностей та вузьких місць продуктивності, скорочуючи час відновлення системи.

Лістинг 4.5 — Використання structured logging у логуванні подій

```
import structlog
logger = structlog.get_logger()
async def process_pricing_request(user_id: int, product_id: int):
    logger.info(
        "Processing pricing request",
        user_id=user_id,
        product_id=product_id,
        service="ml-service",
        trace_id=get_trace_id()
    )
    try:
        result = await calculate_optimal_price(user_id, product_id)
        logger.info(
            "Pricing calculation completed",
            user_id=user_id,
            recommended_price=result.price,
            confidence=result.confidence,
            processing_time_ms=result.processing_time
        )
    except Exception as e:
        logger.error("Pricing calculation failed", error=str(e), exc_info=True)
```

4.6.2 Ключові метрики системи

Без proper моніторингу та alerting неможливо своєчасно виявляти проблеми продуктивності та деградацію якості ML моделей які можуть коштувати реальних грошей для бізнесу [15]. Детальні метрики наведені в таблиці 4.6.

Таблиця 4.6 — Метрики продуктивності системи

Компонент	Метрика	Значення	95% вимірювань	SLA
API Gateway	Латентність	45 мс	120 мс	< 150 мс
	Throughput	450 req/sec	680 req/sec	> 300 req/sec
	Error rate	0.3%	-	< 1%
	Uptime	99.7%	-	> 99.5%
ML Service	Prediction time	85-150 мс	180 мс	< 200 мс
	Model accuracy (elasticity)	R^2=0.89	-	> 0.75
	Model accuracy (demand)	R^2=0.82	-	> 0.70
	Cache hit ratio	78%	-	> 70%
Database	Query latency	12 мс	25 мс	< 50 мс
	Connection pool	65% utilization	-	< 80%
	Cache hit	85%	-	> 75%
Frontend	FCP	1.2 с	-	< 1.8 с
	Lighthouse score	87/100	-	> 85/100

Моніторинг системи побудований на принципах observability - логування, метрики і трейсинг працюють разом. Structured logging з JSON форматом використовуємо для зручного парсингу. Кожен запит отримує унікальний trace_id який проходить через всі мікросервіси, так можна відстежити повний шлях обробки.

Критичні метрики збираються в реальному часі через Prometheus, візуалізація в Grafana. Alerting через AlertManager з інтеграцією в Slack і email - так реагуємо на проблеми швидко. Особливо уважно дивимось на метрики ML моделей: точність прогнозів, час обробки, якість рекомендацій.

Для бізнес-метрик відстежуємо конверсію рекомендацій, середню зміну прибутку від оптимізації цін, кількість успішних автоматичних коригувань. Це дає можливість оцінювати ефективність не тільки технічно, але й комерційно.

Алерти мають три рівні: Critical (негайна реакція), Warning (моніторинг протягом години), Info (щоденний огляд). Автоматичне масштабування спрацьовує при 70% CPU або пам'яті, продуктивність стабільна навіть при пікових навантаженнях.

5 ЕКОНОМІЧНА ЧАСТИНА

5.1 Проведення комерційного та технологічного аудиту розробки системи

У сучасних умовах стрімкої цифровізації економіки e-commerce платформи стикаються з необхідністю оперативно реагувати на зміни попиту, конкуренції та ринкових тенденцій. Динамічне ціноутворення, побудоване на основі машинного навчання, дозволяє автоматизувати процес коригування цін, підвищуючи ефективність цінових стратегій та конкурентоспроможність підприємства. Однак впровадження таких систем потребує якісного комерційного та технологічного аудиту, що визначає доцільність інвестицій і рівень технологічної готовності рішення.

Розробка мікросервісної архітектури для системи динамічного ціноутворення особливо актуальна завдяки її масштабованості, гнучкості та стійкості до навантажень. У контексті використання хмарних технологій такі системи можуть забезпечувати високу доступність, швидке розгортання нових модулів та оптимізацію витрат на інфраструктуру. Проте відсутність попереднього аудиту може призвести до неефективного використання ресурсів, технічних ризиків, несумісності компонентів або недостатньої продуктивності системи.

Проведення комплексного комерційного та технологічного аудиту є критично важливим етапом, який забезпечує обґрунтованість впровадження microservices-рішень та технологій ML у сфері e-commerce. Такий аудит дає змогу визначити економічний ефект від реалізації проєкту, оцінити готовність інфраструктури до інтеграції інтелектуальних модулів та мінімізувати ризики, пов’язані з масштабуванням, безпекою даних та експлуатаційними витратами.

Саме тому дослідження методів і практик аудиту мікросервісних систем динамічного ціноутворення є актуальним і важливим для підприємств, що прагнуть підвищити свою ефективність у цифровій економіці. Для проведення комерційного та технологічного аудиту залучаємо 3-х незалежних експертів, якими є провідні викладачі випускової або спорідненої кафедри. Оцінювання

науково-технічного рівня мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання та її комерційного потенціалу здійснюємо із застосуванням п’ятибальної системи оцінювання за 12-ма критеріями, а результати зводимо до таблиці 5.1.

Таблиця 5.1 — Результати оцінювання науково-технічного рівня і комерційного потенціалу мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання

Критерії	Експерти		
	Експерт 1	Експерт 2	Експерт 3
	Бали, виставлені експертами		
Технічна здійсненність концепції	3	3	3
Ринкові переваги (наявність аналогів)	3	2	3
Ринкові переваги (ціна продукту)	2	3	3
Ринкові переваги (технічні властивості)	3	3	3
Ринкові переваги (експлуатаційні витрати)	3	3	2
Ринкові перспективи (розмір ринку)	3	3	3
Ринкові перспективи (конкуренція)	2	3	3
Практична здійсненність (наявність фахівців)	3	3	2
Практична здійсненність (наявність фінансів)	2	2	3
Практична здійсненність (необхідність нових матеріалів)	3	3	3
Практична здійсненність (термін реалізації)	3	3	3
Практична здійсненність (розробка документів)	3	2	3
Сума балів	30	30	31
Середньоарифметична сума балів, СБ	30,3		

За результатами розрахунків, наведених в таблиці 1 робимо висновок про те, що науково-технічний рівень та комерційний потенціал мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання — вищий середнього.

5.2 Розрахунок витрат на здійснення розробки системи

Витрати на виплату основної та додаткової заробітної плати керівникам відділів, лабораторій, секторів і груп, науковим, інженерно-технічним працівникам, конструкторам, технологам, креслярам, копіювальникам, лаборантам, робітникам, студентам, аспірантам та іншим працівникам, безпосередньо зайнятим виконанням конкретної теми, обчисленої за посадовими окладами, відрядними розцінками, тарифними ставками згідно з чинними в організаціях системами оплати праці, також будь-які види грошових і матеріальних доплат, які належать до елемента. У розрахунок включаються всі передбачені законодавством та внутрішніми положеннями організації надбавки, премії та компенсаційні виплати. Таким чином, цей елемент витрат є найбільш значущим у структурі загальної собівартості науково-дослідної роботи. І розрахунки можна переглянути у таблиці 5.2.

Основну заробітну плату дослідників ($З_о$) розраховують відповідно до посадових окладів працівників, за формулою (5.1).

$$Z_0 = \sum_{i=1}^{K} \frac{M_{ni}\, t_i}{T_p}\ , \tag{5.1}$$

де к — кількість посад дослідників, залучених до процесу дослідження;

M_{ni} — місячний посадовий оклад конкретного розробника (інженера, дослідника, науковця тощо), грн.;

T_p — число робочих днів в місяці, приблизно T_p = (21…23) дні, приймаємо 22 дні;

t_i — число робочих днів роботи розробника (дослідника).

Таблиця 5.2 — Витрати на заробітну плату дослідників

Посада	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату, грн.
Керівник	26 000	1182	30	35455
Розробник	16 000	727	120	87273
Консультанти	15 000	682	15	10227
Всього:	132955			

Витрати на основну заробітну плату робітників ($З_р$) за відповідними найменуваннями робіт розраховують за формулою (5.2).

$$Z_p = \sum_{i=1}^{n} C_i \cdot t_i \,, \tag{5.2}$$

де C_i — погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

t_i — час роботи робітника на виконання певної роботи, год.

Погодинну тарифну ставку робітника відповідного розряду С і можна визначити за формулою (5.3).

$$C_i = \frac{M_m \cdot K_i \cdot K_c}{T_p \cdot t_{zm}} \,, \tag{5.3}$$

де $М_м$ — розмір прожиткового мінімуму працездатної особи або мінімальної місячної заробітної плати (залежно від діючого законодавства), у 2025 році $М_м$=8000 грн;

$К_і$ — коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду;

$К_с$ — мінімальний коефіцієнт співвідношень місячних тарифних ставок

робітників першого розряду з нормальними умовами праці виробничих об’єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати, складає 1,1;

$T_р$ — середня кількість робочих днів в місяці, приблизно $T_р$ = 21…23 дні, приймаємо 22 дні;

$t_{зм}$ — тривалість зміни, год., приймаємо 8 год.

Усі зроблені розрахунки зводимо до таблиці 5.3. Ця таблиця являє собою зведену відомість усіх економічних показників та витрат, розрахованих у попередніх підрозділах. Вона слугує для наочної демонстрації фінансової обґрунтованості проєкту та підсумкового оцінювання його економічної ефективності. Наведені в таблиці дані є основою для фінальних висновків щодо комерційного потенціалу та окупності розробки.

Таблиця 5.3 — Витрати на заробітну плату робітників

Найменування робіт	Трудомісткість, н-год.	Розряд роботи	Погодинна тарифна ставка	Тариф. коеф.	Величина, грн.
Аналіз вимог та архітектурне проектування	160	7	77	1,54	12320
Розробка мікросервісів системи	320	6	72,5	1,45	23200
Реалізація ML-алгоритмів ціноутворення	240	7	77	1,54	18480
Інтеграція з хмарними сервісами	120	6	72,5	1,45	8700
Тестування та документація	80	5	68	1,36	5440
Всього					68140

Додаткова заробітна плата $З_д$ всіх розробників та робітників, які брали участь у виконанні даного етапу роботи, розраховується як (10…12)% від суми основної заробітної плати всіх розробників та робітників, за формулою (5.4).

$$З_д = 0,1 \cdot (З_о + З_р) = 0,1 \cdot (132955 + 68140) = 20109 \text{ грн.} \quad (5.4)$$

Нарахування на заробітну плату $Н_{зп}$ розробників та робітників, які брали участь у виконанні даного етапу роботи, розраховуються за формулою (5.5).

$$Н_{зп} = \beta \cdot (З_о + З_р + З_д) =$$
$$0,22 \cdot (132955 + 68140 + 20109) = 48665 \text{ грн.}, \quad (5.5)$$

де $З_о$ — основна заробітна плата розробників, грн.;

$З_р$ — основна заробітна плата робітників, грн.;

$З_д$ — додаткова заробітна плата всіх розробників та робітників, грн.;

β — ставка єдиного внеску на загальнообов'язкове державне соціальне страхування, % (приймаємо для 1-го класу професійності ризику 22%).

Витрати на матеріали М, що були використані під час виконання даного етапу роботи, розраховуються за формулою (5.6).

$$М = \sum_{1}^{n} Н_i \cdot {}_i \cdot К_i - \sum_{1}^{n} В_i \cdot {}_i, \quad (5.6)$$

де $Н_i$ — кількість матеріалів і-го виду, шт.;

$Ц_i$ — ціна матеріалів і-го виду, грн.;

$К_i$ — коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15);

n — кількість видів матеріалів.

Ця формула враховує вартість як фізичних матеріалів, так і амортизацію нематеріальних активів (ліцензії на програмне забезпечення, витрати на хмарні

сервіси, які були використані як “матеріали” для реалізації). Враховуючи, що більша частина роботи є інтелектуальною, до матеріальних витрат також включаються допоміжні засоби, необхідні для оформлення та презентації результатів. Критично важливою складовою є детальний облік витрат на хмарні обчислення та ліцензії, оскільки вони формують значну частину собівартості сучасної мікросервісної системи. Кожна одиниця матеріалів, включно з оновленням програмного забезпечення, має бути обґрунтована для забезпечення точності фінансового розрахунку. Таким чином, дані, отримані в результаті застосування цієї формули, дозволяють прозоро систематизувати структуру прямих матеріальних витрат. Усі зроблені розрахунки зводимо до таблиці 5.4.

Таблиця 5.4 — Матеріали, що використані на розробку

Найменування матеріалів	Ціна за одиницю, грн.	Витрачено	Вартість витрачених матеріалів, грн.
Хмарна інфраструктура AWS EC2, RDS, S3	20000	4	80000
Системи моніторингу та логування	8000	4	32000
SSL сертифікати та домени	4200	1	4200
Ліцензії на розробницькі інструменти	13500	1	13500
Всього, з врахуванням коефіцієнта транспортних витрат			142670

Витрати на комплектуючі К, що були використані під час виконання даного етапу роботи, розраховуються за формулою (5.7).

$$К = \sum_{1}^{n} Н_i \cdot {}_i \cdot К_i, \tag{5.7}$$

де $Н_i$ — кількість комплектуючих і-го виду, шт.;

$Ц_i$ — ціна комплектуючих і-го виду, грн.;

$К_i$ — коефіцієнт транспортних витрат, $К_i = (1{,}1 \ldots 1{,}15)$;

n — кількість видів комплектуючих.

Зроблені розрахунки приводяться у таблиці 5.5.

Таблиця 5.5 — Комплектуючі, що використані на розробку

Найменування комплектуючих	Ціна за одиницю, грн.	Витрачено	Вартість витрачених комплектуючих, грн.
Сервер розробки (потужна робоча станція)	85000	1	85000
Додаткові SSD NVMe накопичувачі 2TB	12500	2	25000
Професійні монітори для розробки 27”	15000	2	30000
Мережеве обладнання (роутер, комутатор)	8500	1	8500
Всього, з врахуванням коефіцієнта транспортних витрат			164835

Вартість спецустаткування визначається за прейскурантом гуртових цін або за даними базових підприємств за відпускними і договірними цінами. У разі відсутності прайслісту, за основу беруться ринкові ціни, підтверджені комерційними пропозиціями або даними тендерних закупівель. До цієї вартості обов’язково додаються витрати на транспортування і налагодження наведені у формулі (5.8).

$$В = \sum_{1}^{\kappa} {}_{i} \cdot С_{.i} \cdot К_{i}, \qquad (5.8)$$

де $Ц_i$ — ціна придбання спецустаткування і-го виду, грн.;

$С_{пр.i}$ — кількість одиниць спецустаткування відповідного виду, шт.;

$К_i$ — коефіцієнт транспортних витрат, $К_i$ = (1,1…1,15);

n — кількість видів спецустаткування.

Зроблені розрахунки зводимо до таблиці 5.6.

Таблиця 5.6 — Витрати на придбання спецустаткування

Найменування спецустаткування	Ціна за одиницю, грн.	Витрачено	Вартість спецустаткування, грн.
Системи безперервного моніторингу	15000	1	15000
DevOps інструменти та CI/CD платформи	12000	1	12000
Всього, з врахуванням коефіцієнта транспортних витрат			6050

До балансової вартості програмного забезпечення входять витрати на його інсталяцію, тому ці витрати беруться додатково в розмірі 10…12% від вартості програмного забезпечення. Сюди в розрахунок включається вартість усього ліцензійного програмного забезпечення (IDE, допоміжні інструменти, тощо), яке було необхідне для розробки мікросервісної архітектури та моделей машинного навчання. Таке деталізоване врахування є обов'язковим для точного обчислення повної собівартості науково-дослідної роботи та подальшого розрахунку амортизаційних відрахувань. Коефіцієнт додаткових витрат (10-12%) охоплює час і ресурси, витрачені на налаштування, конфігурацію та інтеграцію програмного забезпечення в загальний технологічний стек. Балансову вартість програмного забезпечення розраховують за формулою (5.9).

$$B_{prg} = \sum_{1}^{k} C_{iprg} \cdot C_{pr.i} \cdot K_i \,, \tag{5.9}$$

де $Ц_{іпрг}$ — ціна придбання програмного забезпечення і-го виду, грн.;

$С_{прг.і}$ — кількість одиниць програмного забезпечення відповідного виду, шт.;

$К_і$ — коефіцієнт, що враховує інсталяцію, налагодження програмного забезпечення, $К_і = (1,1…1,12)$;

к — кількість видів програмного забезпечення.

Зроблені розрахунки зводимо до таблиці 5.7.

Таблиця 5.7 — Витрати на придбання програмного забезпечення

Найменування програмного забезпечення	Ціна за одиницю, грн.	Витрачено	Вартість програмного забезпечення, грн.
JetBrains IntelliJ IDEA Ultimate (річна ліцензія)	14 000	1	14000
Docker Desktop Pro (річна ліцензія)	7 000	1	7000
GitHub Copilot (4 місяці)	800	1	800
Figma Professional (річна ліцензія)	4 500	1	4500
Postman Pro (річна ліцензія)	2 100	1	2100
Всього, з врахуванням коефіцієнта інсталяції та налагодження			31524

У спрощеному вигляді амортизаційні відрахування що наведені у таблиці 5.8 в цілому бути розраховані за формулою (5.10).

$$А = \frac{6}{Т_В} \cdot \frac{t}{12}, \quad (5.10)$$

де $Ц_б$ — загальна балансова вартість всього обладнання, комп’ютерів, приміщень тощо, що використовувались для виконання даного етапу роботи, грн.;

t — термін використання основного фонду, місяці;

$T_в$ — термін корисного використання основного фонду, роки.

Таблиця 5.8 — Амортизаційні відрахування за видами основних фондів

Найменування	Балансова вартість, грн.	Строк корисного використання, років	Термін використання, місяців	Сума амортизації, грн.
Робоча станція розробника	85 000	4	3	5312,5
Сервер тестування та розробки	45 000	2	3	5625,0
Мережеве обладнання	15 000	3	3	1250,0
UPS (джерело безперебійного живлення)	12 000	5	3	600,0
Всього	12787,5			

Витрати на електроенергію для науково-виробничих цілей включає споживання електроенергії обчислювальним обладнанням та іншими пристроями, необхідними для розробки, тестування та моделювання мікросервісної архітектури, розраховуються за формулою (5.11).

$$Ве = \sum \frac{W_i \bullet t_i \bullet Ц_е \bullet К_{впі}}{ККД} = \frac{0{,}8 \bullet 960 \cdot 4{,}32 \cdot 0{,}75}{0{,}98} + \frac{0{,}4 \bullet 720 \cdot 4{,}32 \cdot 0{,}75}{0{,}98} + \\ + \frac{0{,}1 \bullet 720 \cdot 4{,}32 \cdot 0{,}75}{0{,}98} = 3729{,}3 \text{ грн.}, \quad (5.11)$$

де W_i — встановлена потужність обладнання, кВт;

t_i — тривалість роботи обладнання на етапі дослідження, год.;

Це — вартість 1 кВт електроенергії, 4,32 грн.;

$К_{впі}$ — коефіцієнт використання потужності;

ККД — коефіцієнт корисної дії обладнання.

Витрати на електроенергію наведені у таблиці 5.9

Таблиця 5.9 — Витрати на електроенергію

Найменування обладнання	Потужність, кВт	Тривалість годин роботи
Робоча станція розробника	0,80	960
Сервер розробки та тестування	0,4	720
Мережеве обладнання	0,10	720

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками.

Витрати за статтею «Інші витрати» розраховуються як 50…100% від суми основної заробітної плати дослідників та робітників за формулою (5.12).

$$Ів = \left(З_о + З_р\right) \bullet \frac{Н_{ів}}{100\%} = (132955 + 68140) \bullet \frac{50}{100} = \\ = 100547,4 \text{ грн.}, \quad (5.12)$$

де $Н_{ів}$ — норма нарахування за статтею «Інші витрати».

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов’язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов’язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов’язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін.

Витрати за статтею «Накладні (загальновиробничі) витрати»

розраховуються як 100…200% від суми основної заробітної плати дослідників та робітників за формулою (5.13).

$$\text{Внзв} = \left(З_{\text{о}} + З_{\text{р}}\right) \bullet \frac{Н_{\text{нзв}}}{100\%} = (132955 + 68140) \bullet \frac{110}{100} = \\ = 221204 \text{ грн.}, \quad (5.13)$$

де $Н_{\text{нзв}}$ — норма нарахування за статтею «Накладні (загальновиробничі) витрати».

Витрати на проведення розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання. Витрати на проведення науково-дослідної роботи розраховуються як сума всіх попередніх статей витрат.

$$В_{\text{заг}} = 132955 + 68140 + 20209 + 48665 + 142670 + 164835 + \\ + 29700 + 31524 + 12787,5 + 3729,3 + \\ + 100547,3 + 221204 = 976866 \text{ грн.}$$

Загальні витрати ЗВ на завершення науково-дослідної (науково-технічної) роботи з розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання та оформлення її результатів розраховуються за формулою (5.14).

$$ЗВ = \frac{В_{\text{заг}}}{\eta} = \frac{976866}{0,4} = 2442164,9 \text{ грн.}, \quad (5.14)$$

де η — коефіцієнт, що характеризує етап виконання науково-дослідної роботи.

Оскільки, якщо науково-технічна розробка знаходиться на стадії розробки технології, то η=0,4. Це значення відображає проміжну стадію готовності, тоді як для повного впровадження зазвичай використовується коефіцієнт η=0,92.

5.3 Розрахунок економічної ефективності науково-технічної розробки

В ринкових умовах узагальнюючим позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів тієї чи іншої науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання, є збільшення у потенційного інвестора величини чистого прибутку.

В даному випадку відбувається розробка засобу, тому основу майбутнього економічного ефекту буде формувати: ΔN — збільшення кількості споживачів, яким надається відповідна інформаційна послуга в аналізовані періоди часу; N — кількість споживачів, яким надавалась відповідна інформаційна послуга у році до впровадження результатів нової науково-технічної розробки; $Ц_б$ — вартість послуги у році до впровадження інформаційної системи; $\pm\Delta Ц_о$ — зміна вартості послуги (зростання чи зниження) від впровадження результатів науково-технічної розробки в аналізовані періоди часу.

Можливе збільшення чистого прибутку у потенційного інвестора ΔПі для кожного із років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховується за формулою (5.15).

$$\Delta П = \left(\pm \Delta Ц_о \bullet N + Ц_о \bullet \Delta N_i\right)_i \bullet \lambda \cdot \rho \cdot \left(1 - \frac{\vartheta}{100}\right), \qquad (5.15)$$

де ±ΔЦ — зміна основного якісного показника від впровадження результатів науково-технічної розробки в аналізованому році. Зазвичай, таким показником може бути зміна ціни реалізації одиниці нової розробки в аналізованому році (відносно року до впровадження цієї розробки);

$\pm\Delta Ц_о$ — може мати як додатне, так і від'ємне значення (від'ємне — при зниженні ціни відносно року до впровадження цієї розробки, додатне — при зростанні ціни);

N — основний кількісний показник, який визначає величину попиту на

аналогічні чи подібні розробки у році до впровадження результатів нової науково-технічної розробки;

$Ц_о$ — основний якісний показник, який визначає ціну реалізації нової науково-технічної розробки в аналізованому році;

$Ц_б$ — основний якісний показник, який визначає ціну реалізації існуючої (базової) науково-технічної розробки у році до впровадження результатів;

ΔN — зміна основного кількісного показника від впровадження результатів науково-технічної розробки в аналізованому році. Зазвичай таким показником може бути зростання попиту на науково-технічну розробку в аналізованому році (відносно року до впровадження цієї розробки);

λ — коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2025 році ставка податку на додану вартість становить 20%, а коефіцієнт λ =0,8333;

ρ — коефіцієнт, який враховує рентабельність інноваційного продукту (послуги). Рекомендується брати ρ =0,2...0,5;

ϑ — ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2025 році ϑ =18%.

Очікуваний термін життєвого циклу розробки 3 роки, тому: 1-й рік: ΔП1=2195162 грн.,2-й рік: ΔП2=3555918 грн., 3-й рік: ΔП3=6821732 грн.

Далі розраховують приведену вартість збільшення всіх чистих прибутків ПП, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання формула (5.16).

$$ПП = \sum_{i=1}^{T} \frac{ΔП_i}{(1+\tau)^t} = \frac{2195162}{(1+0,1)^1} + \frac{3555918}{(1+0,1)^2} + \frac{6821732}{(1+0,1)^3} = 10059645,2 \text{ грн.}, \quad (5.16)$$

де ΔПі — збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн.;

Т — період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки (приймаємо Т=3 роки);

τ — ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ =0,05…0,15;

t — період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

Далі розраховують величину початкових інвестицій PV , які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання. Для цього можна використати формулу (5.17).

$$PV = k_{\text{інв}} \cdot ЗВ = 2 \bullet 2442164,9 = 4884330 \text{ грн.}, \quad (5.17)$$

де $k_{\text{інв}}$ — коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання та її комерціалізацію. Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо. Зазвичай кінв=1...5, але може бути і більшим;

ЗВ — загальні витрати на проведення науково-технічної розробки та оформлення її результатів, грн.

Тоді абсолютний економічний ефект $E_{абс}$ або чистий приведений дохід для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання за формулою (5.18).

$$\text{Е}_{\text{абс}} = \text{ПП} - PV = 10059645,2 - 4884330 = 5175315 \text{ грн.}, \quad (5.18)$$

де ПП — приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання, грн.;

PV — теперішня вартість початкових інвестицій, грн.

Оскільки $\text{Е}_{\text{абс}}>0$, то можемо припустити про потенційну зацікавленість у розробці мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання.

Для остаточного прийняття рішення з цього питання необхідно розрахувати внутрішню економічну дохідність Ев або показник внутрішньої норми дохідності вкладених інвестицій та порівняти її з так званою бар'єрною ставкою дисконтування, яка визначає ту мінімальну внутрішню економічну дохідність, нижче якої інвестиції в будь-яку науково-технічну розробку мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання вкладати буде економічно недоцільно.

Внутрішня економічна дохідність інвестицій Ев, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання, розраховується за формулою (5.19).

$$\text{Е}_{\text{в}} = \sqrt[\text{Т}_{\text{ж}}]{1 + \frac{\text{Еабс}}{PV}} = \sqrt[3]{1 + \frac{5175315}{4884330}} = 0,69, \quad (5.19)$$

де $\text{Т}_{\text{ж}}$ — життєвий цикл розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних

технологій та машинного навчання, роки.

Далі розраховуємо період окупності інвестицій $T_о$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання за формулою (5.20).

$$T_о = \frac{1}{E_в} = \frac{1}{0{,}69} = 1{,}46 \text{ роки.} \quad (5.20)$$

Оскільки $T_о$<1…3-х років, то це свідчить про комерційну привабливість науково-технічної розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання і може спонукати потенційного інвестора профінансувати впровадження цієї розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання та виведення її на ринок.

Проведення комерційного та технологічного аудиту розробки мікросервісної архітектури системи динамічного ціноутворення для e-commerce підтвердило доцільність впровадження даного рішення та його високий потенціал для комерціалізації. Оцінювання експертів засвідчило, що науково-технічний рівень та ринкові перспективи проєкту перевищують середній рівень, а використання хмарних технологій і методів машинного навчання забезпечує масштабованість, гнучкість і ефективність системи. Розрахунок витрат показав, що повна собівартість розробки становить 976,9 тис. грн, а загальні витрати на завершення НДР — близько 2,44 млн грн.

Подальший аналіз економічної ефективності свідчить, що очікуване збільшення чистого прибутку потенційного інвестора за три роки може досягати понад 10 млн грн у приведеній вартості, що значно перевищує необхідні інвестиції.

Отже, мікросервісна система динамічного ціноутворення є економічно вигідною, технологічно обґрунтованою та перспективною для впровадження в e-commerce, забезпечуючи підвищення конкурентоспроможності та прибутковості підприємства в умовах цифрової економіки. Такий високий показник окупності інвестицій (ROI) робить проєкт привабливим для зовнішнього фінансування та швидкої комерціалізації.

Мікросервісна структура, що передбачає ізоляцію функціоналу та використання патерну Circuit Breaker, мінімізує ризики каскадних відмов, забезпечуючи високу надійність роботи критичних бізнес-процесів. Впровадження системи дозволить компаніям, що використовують WooCommerce, отримати стратегічну перевагу за рахунок автоматизованого реагування на цінові зміни конкурентів і попит.

Технологічна база, заснована на Python, FastAPI та Docker, гарантує легкість інтеграції з більшістю сучасних IT-інфраструктур та подальший розвиток функціоналу.

Таким чином, система є повністю готовим до впровадження інноваційним продуктом, який ефективно вирішує завдання оптимізації прибутковості в умовах агресивного ринку електронної комерції.

ВИСНОВКИ

У ході виконання магістерської кваліфікаційної роботи було проведено комплексне дослідження теоретичних, аналітичних та практичних аспектів створення мікросервісної архітектури системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання.

У першому розділі розглянуто теоретичні основи динамічного ціноутворення в електронній комерції, проаналізовано сучасні підходи до автоматизації ціноутворення та методи машинного навчання для оптимізації цін. Досліджено архітектурні принципи мікросервісних систем, їх переваги над монолітною архітектурою та особливості застосування в системах реального часу. Окрему увагу приділено аналізу існуючих рішень у сфері динамічного ціноутворення та виявленню їх обмежень. Це забезпечило формування системного уявлення про проблемну область та дозволило визначити напрями, де застосування мікросервісної архітектури може підвищити ефективність систем ціноутворення.

У другому розділі досліджено принципи проектування мікросервісної архітектури для систем динамічного ціноутворення, методи декомпозиції монолітних систем на окремі сервіси та паттерни міжсервісної взаємодії. Розроблено архітектуру системи, що складається з шести основних мікросервісів: API Gateway, User Service, Product Service, ML Service, Tracking Service та WebSocket Service. Визначено стратегії управління даними, синхронізації між сервісами та забезпечення відмовостійкості. Це дозволило сформувати методологію побудови масштабованих систем ціноутворення та забезпечило основу для практичної реалізації.

У третьому розділі проведено детальне дослідження алгоритмів машинного навчання для динамічного ціноутворення, включаючи ансамблеві методи (Random Forest, Gradient Boosting, Elastic Net), інкрементальне навчання (SGD Regressor, Passive Aggressive Regressor) та федеративне навчання з

персональними адаптерами. Розроблено систему Feature Engineering для витягування 65+ поведінкових ознак користувачів та створено багаторівневу архітектуру кешування для забезпечення високої продуктивності. Експериментальні дослідження підтвердили ефективність запропонованих алгоритмів: досягнуто точності $R^2=0.89$ для моделі еластичності попиту та $R^2=0.82$ для прогнозування попиту.

У четвертому розділі здійснено практичну реалізацію системи з використанням сучасного технологічного стеку: FastAPI для backend мікросервісів, React з TypeScript для frontend, PostgreSQL та TimescaleDB для зберігання даних, Redis для кешування. Розроблено WooCommerce плагін з JavaScript трекером v2.0 для збору поведінкових даних та створено React-дашборд з 8 основними сторінками. Тестування системи показало високі показники продуктивності: латентність API Gateway 45мс, час ML-прогнозів 85-150мс, cache hit ratio 85-90%.

У п'ятому розділі визначено економічну ефективність розробленої системи та проведено розрахунок витрат на науково-дослідну роботу. Система демонструє значне покращення бізнес-метрик: зростання конверсії з 12.3% до 22.7% (+85%), збільшення середнього чека на 18%, підвищення маржі на 12-15%. Розрахункова економія від оптимізації цін складає до 23% річного прибутку, що підтверджує високий економічний ефект від впровадження.

Проведене дослідження підтвердило доцільність застосування мікросервісної архітектури для систем динамічного ціноутворення та ефективність поєднання сучасних технологій машинного навчання з методами аналізу поведінкових даних. Розроблена система створює перспективи для подальшої автоматизації процесів ціноутворення в електронній комерції та може служити основою для створення комерційних рішень у цій сфері.

Результати виконання магістерської кваліфікаційної роботи оформлені згідно з вимогами [25].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Малініч І. П. Особливості розміщення мікросервісів систем управління навчанням у гібридних хмарах / І. П. Малініч, Я. В. Іванчук // ВНТУ. — 2025. URL: https://ir.lib.vntu.edu.ua/handle/123456789/49430

2. Піх І. В. Використання машинного навчання для кластеризації цільової аудиторії веб-додатків / І. В. Піх, Ю. Ю. Мерених // ВНТУ. — 2025. URL: https://ir.lib.vntu.edu.ua/handle/123456789/49079

3. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. — 2nd ed. — O'Reilly Media, 2021. — 624 p.

4. Паламарчук Є. А. Методи побудови мікросервісної архітектури систем електронного навчання / Є. А. Паламарчук // Інформаційні технології та комп'ютерна інженерія. — 2022. — № 1. — С. 43—54. URL: https://journals.nmetau.edu.ua/index.php/st/article/view/2021

5. Олецький О. On supervising and coordinating microservices within web applications on the basis of state machines / О. Олецький, В. Моголівський // Наукові записки НаУКМА. — 2024. URL: https://ekmair.ukma.edu.ua/items/cf2291d8-11be-40dc-850e-29bcbf9a1d85

6. Hu Y. Optimizing service placement for microservice architecture in clouds / Y. Hu, C. de Laat, Z. Zhao // Applied Sciences. — 2019. — Vol. 9, No. 21. — P. 4663. DOI: 10.3390/app9214663

7. Guerrero C. A lightweight decentralized service placement policy for performance optimization in fog computing / C. Guerrero, I. Lera, C. Juiz // Journal of Ambient Intelligence and Humanized Computing. — 2019. — Vol. 10. — P. 2435—2452. DOI: 10.1007/s12652-018-10068-5

8. A cloud software life cycle process (CSLCP) model https://www.sciencedirect.com/science/article/pii/S2090447920302495 — 2020.

9. Sharifani K. Machine learning and deep learning: A review of methods and applications / K. Sharifani, M. Amini // World Information Technology and Engineering Journal. — 2023. — Vol. 10, No. 07. — P. 3897—3904.

10. Policarpo L. Machine learning through the lens of e-commerce initiatives: An up-to-date systematic literature review / L. Policarpo et al. // Computer Science Review. — 2021. — Vol. 41. — P. 100414. DOI: 10.1016/j.cosrev.2021.100414

11. Chaipornkaew P. A recommendation model based on user behaviors on commercial websites using TF-IDF, KMeans, and Apriori algorithms / P. Chaipornkaew, T. Banditwattanawong // International Conference on Computing and Information Technology. — Cham: Springer International Publishing, 2021. — P. 55—65. DOI: 10.1007/978-3-030-79757-7_6

12. Singh H. An effective clustering-based web page recommendation framework for e-commerce websites / H. Singh, P. Kaur // SN Computer Science. — 2021. — Vol. 2, No. 4. — P. 339. DOI: 10.1007/s42979-021-00736-z

13. SadighZadeh S. Modeling user preferences in online stores based on user mouse behavior on page elements / S. SadighZadeh, M. Kaedi // Journal of Systems and Information Technology. — 2022. — Vol. 24, No. 2. — P. 112—130. DOI: 10.1108/JSIT-12-2019-0264

14. Song X. The Application of Artificial Intelligence in Electronic Commerce / X. Song, S. Yang, Z. Huang, T. Huang // Journal of Physics: Conference Series. — 2019. — Vol. 1302, No. 3. — P. 032030. DOI: 10.1088/1742-6596/1302/3/032030

15. Ткаченко О. Деякі аспекти автоматизації бізнес-процесів електронної комерції / О. Ткаченко, М. Гнатюк // Цифрова платформа: інформаційні технології в соціокультурній сфері. — 2023. — № 1. DOI: 10.31866/2617-796X.3.2023.293620

16. WooCommerce REST API Documentation. — https://woocommerce.github.io/woocommerce-rest-api-docs/

17. Docker Documentation. — https://docs.docker.com/

18. Kubernetes Documentation. — https://kubernetes.io/docs/

19. Géron A. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow / A. Géron. — 3rd ed. — O'Reilly Media, 2022. — 861 p.

20. Raschka S. Machine Learning with PyTorch and Scikit-Learn / S. Raschka, Y. Liu, V. Mirjalili. — Packt Publishing, 2022. — 774 p.

21. Müller A. Introduction to Machine Learning with Python: A Guide for Data Scientists / A. Müller, S. Guido. — O'Reilly Media, 2020. — 400 p.

22. Zheng A. Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists / A. Zheng, A. Casari. — O'Reilly Media, 2018. — 218 p.

23. Kuhn M. Feature Engineering and Selection: A Practical Approach for Predictive Models / M. Kuhn, K. Johnson. — Chapman and Hall/CRC, 2019. — 297 p.

24. Захарченко С. М. Проблеми масштабування монолітних систем ціноутворення в e-commerce та їх вирішення через мікросервісну архітектуру / С. М. Захарченко, Ю. О. Сеник // Матеріали конференції "Молодь в науці: дослідження, проблеми, перспективи (МН-2026)". — ВНТУ, 2026. URL: https://conferences.vntu.edu.ua/index.php/mn/mn2026/paper/view/26052

25. Методичні вказівки до виконання магістерських кваліфікаційних робіт студентами спеціальності 123 “Комп’ютерна інженерія”. / Укладачі О.Д. Азаров, О.В. Дудник, С.І. Швець − Вінниця : ВНТУ, 2023. − 57 с.

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ

д.т.н., проф. Азаров О. Д.

_____

“ 3 “ жовтня 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи

“Мікросервісна архітектура системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання”

Науковий керівник: к.т.н., проф.

Захарченко С. М.

Виконав: студент гр. 2КІ-24м

Сеник Ю.О.

ВНТУ 2025

1 Підстава для виконання магістерської кваліфікаційної роботи (МКР)

1.1 Актуальність полягає в тому, що сучасний e-commerce ринок характеризується високою конкуренцією та динамічністю, що вимагає від інтернет-магазинів швидкого реагування на зміни ринкових умов. Традиційні методи ціноутворення, засновані на статичних правилах та ручному аналізі, не здатні ефективно враховувати складну взаємодію факторів, таких як поведінка користувачів, сезонність, конкурентне середовище та індивідуальні переваги покупців. Застосування сучасних технологій машинного навчання та аналізу поведінкових даних дозволяє створити інтелектуальну систему динамічного ціноутворення, яка автоматизує процеси оптимізації цін, підвищує конверсію та максимізує прибутковість e-commerce бізнесу.

1.2 Наказ про затвердження теми МКР №313 від 24.09.2025 р.

2 Мета і призначення МКР

2.1 Метою роботи є розробка інтелектуальної системи динамічного ціноутворення для e-commerce платформ з використанням методів машинного навчання та аналізу поведінкових даних користувачів.

2.2 Призначення розробки — забезпечення автоматизованого управління ціновою політикою інтернет-магазинів шляхом впровадження алгоритмів машинного навчання для аналізу поведінки користувачів, прогнозування попиту та оптимізації цін у реальному часі, що підвищує конкурентоспроможність бізнесу, збільшує конверсію та максимізує прибутковість.

3 Вихідні дані для виконання МКР

3.1 Теоретичний аналіз методів динамічного ціноутворення та алгоритмів машинного навчання для e-commerce.

3.2 Вивчення технологій збору та аналізу поведінкових даних користувачів веб-платформ.

3.3 Дослідження архітектурних підходів до побудови масштабованих

систем реального часу.

3.4 Проведення експериментальних досліджень ефективності різних ML алгоритмів для прогнозування попиту.

3.5 Виконання економічних розрахунків для доведення значущості наукового дослідження.

4 Технічні вимоги до виконання МКР

Технічні вимоги:

— аналіз теоретичних основ динамічного ціноутворення та методів машинного навчання для e-commerce;

— розробка архітектури мікросервісної системи для збору та обробки поведінкових даних;

— проектування алгоритмів федеративного машинного навчання для персоналізації рекомендацій;

— створення системи аналізу та прогнозування попиту на товари;

— розробка API для інтеграції з популярними e-commerce платформами (WooCommerce);

— забезпечити масштабованість системи з урахуванням зростання обсягів даних та кількості користувачів;

— впровадження інструментів моніторингу та забезпечення надійності системи з використанням патернів захисту від відмов;

— проведення експериментальних досліджень з моделюванням різних сценаріїв ціноутворення;

— порівняльний аналіз ефективності різних алгоритмів машинного навчання;

— формування методології оцінки якості роботи системи динамічного ціноутворення.

5 Етапи МКР та очікувані результати

Етапи роботи та очікувані результати приведено в Таблиці А.1

Таблиця А.1 — Етапи МКР

№ з/п	Назва етапу	Термін виконання		Очікувані результати
		початок	закінчення	
1	Постановка задачі роботи	08.09.2025	08.09.2025	Задачі дослідження
2	Аналіз методів динамічного ціноутворення, алгоритмів машинного навчання та технологій e-commerce	10.09.2025	20.09.2025	Розділ 1
3	Дослідження архітектурних підходів до побудови систем реального часу та методів аналізу поведінкових даних	21.09.2025	04.10.2025	Розділ 2
4	Проектування та розробка системи динамічного ціноутворення	05.10.2025	29.10.2025	Розділ 3
5	Реалізація системи, проведення експериментальних досліджень та оцінка ефективності	30.10.2025	07.11.2025	Розділ 4
6	Розрахунок економічної частини	07.11.2025	10.11.2025	Розділ 5
7	Оформлення матеріалів до захисту МКР	16.10.2025	16.10.2025	ПЗ, графічний матеріал і презентація
8	Перевірка якості виконання магістерської роботи та усунення недоліків	25.10.2025	25.10.2025	Оформлені документи
9	Підписи супроводжувальних документів у нормоконтролера, керівника, опонента	03.11.2025	03.11.2025	Оформлені документи
10	Перевірка на антиплагіат та ШІ	05.11.2025	05.11.2025	Оформлені документи

6 Матеріали, що подаються до захисту МКР

До захисту подаються: пояснювальна записка МКР, графічні матеріали, протокол попереднього захисту МКР на кафедрі, відгук наукового керівника, відгук опонента, протоколи складання державних екзаменів, анотації до МКР українською та іноземною мовами.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання МКР

8.1 При оформлюванні МКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

— методичні вказівки до виконання магістерських кваліфікаційних робіт зі спеціальності 123 — «Комп'ютерна інженерія»;

— документи на які посилаються у вище вказаних.

8.2 Порядок виконання МКР викладено в «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21».

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Мікросервісна архітектура системи динамічного ціноутворення для e-commerce платформ з використанням хмарних технологій та машинного навчання

Тип роботи: магістерська кваліфікаційна робота
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра обчислювальної техніки, ФІТКІ, 2КІ-24м
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) 3 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- ✔ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.
- • У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.
- • У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

Азаров О. Д., д.т.н., зав. каф. ОТ _____
(прізвище, ініціали, посада) (підпис)

Мартинюк Т. Б., д.т.н., проф. каф. ОТ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Захарченко С. М.
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ Захарченко С. М., к.т.н., проф. каф. ОТ
(підпис) (прізвище, ініціали, посада)

Здобувач _____ Сеник Ю.О.
(підпис) (прізвище, ініціали)

ДОДАТОК В

ML Feature Engineering — Обробка поведінкових даних

BehaviorFeatures Dataclass

```
from dataclasses import dataclass
from typing import List, Optional, Dict, Any
from datetime import datetime
import numpy as np

@dataclass
class BehaviorFeatures:
  # Активність (6 ознак)
  total_sessions: int
  avg_session_duration: float
  pages_per_session: float
  bounce_rate: float
  return_visitor: bool
  days_since_last_visit: int

  # Покупкова поведінка (8 ознак)
  total_purchases: int
  avg_order_value: float
  purchase_frequency: float
  cart_abandonment_rate: float
  favorite_categories: List[str]
  price_sensitivity: float
  discount_usage_rate: float
  payment_method_preference: str

  # Часові патерни (4 ознаки)
  preferred_shopping_hours: List[int]
  weekend_vs_weekday_ratio: float
  seasonal_activity_pattern: str
  time_to_purchase: float

  # Товарні переваги (3 ознаки)
  brand_loyalty_score: float
  product_diversity_index: float
  review_engagement_score: float

  # Технічні характеристики (3 ознаки)
  device_type: str
  browser_type: str
  geographic_region: str

  def to_feature_vector(self) -> np.ndarray:
    """Конвертація в числовий вектор для ML моделі"""
    features = []

    # Числові ознаки
```

```
        numerical_features = [
            self.total_sessions,
            self.avg_session_duration,
            self.pages_per_session,
            self.bounce_rate,
            int(self.return_visitor),
            self.days_since_last_visit,
            self.total_purchases,
            self.avg_order_value,
            self.purchase_frequency,
            self.cart_abandonment_rate,
            self.price_sensitivity,
            self.discount_usage_rate,
            self.weekend_vs_weekday_ratio,
            self.time_to_purchase,
            self.brand_loyalty_score,
            self.product_diversity_index,
            self.review_engagement_score
        ]

        features.extend(numerical_features)

        # Категоріальні ознаки (one-hot encoding)
        device_encoding = self._encode_device_type()
        browser_encoding = self._encode_browser_type()
        region_encoding = self._encode_geographic_region()

        features.extend(device_encoding)
        features.extend(browser_encoding)
        features.extend(region_encoding)

        return np.array(features, dtype=np.float32)

    def _encode_device_type(self) -> List[float]:
        """One-hot encoding для типу пристрою"""
        devices = ['desktop', 'mobile', 'tablet']
        encoding = [0.0] * len(devices)
        if self.device_type.lower() in devices:
            idx = devices.index(self.device_type.lower())
            encoding[idx] = 1.0
        return encoding

    def _encode_browser_type(self) -> List[float]:
        """One-hot encoding для типу браузера"""
        browsers = ['chrome', 'firefox', 'safari', 'edge', 'other']
        encoding = [0.0] * len(browsers)
        browser_lower = self.browser_type.lower()
        if browser_lower in browsers:
            idx = browsers.index(browser_lower)
            encoding[idx] = 1.0
        else:
            encoding[-1] = 1.0  # 'other'
```

```
        return encoding

    def _encode_geographic_region(self) -> List[float]:
        """One-hot encoding для географічного регіону"""
        regions = ['north_america', 'europe', 'asia', 'other']
        encoding = [0.0] * len(regions)
        region_lower = self.geographic_region.lower()
        if region_lower in regions:
            idx = regions.index(region_lower)
            encoding[idx] = 1.0
        else:
            encoding[-1] = 1.0  # 'other'
        return encoding

Feature Extraction від Tracking Events
from typing import List, Dict
from collections import defaultdict, Counter
import pandas as pd
from datetime import datetime, timedelta

class FeatureExtractor:
    def __init__(self):
        self.feature_cache = {}

    async def extract_features(
        self,
        user_id: int,
        events: List[Dict[str, Any]]
    ) -> BehaviorFeatures:
        """Витягування ознак з tracking events"""

        if not events:
            return self._get_default_features()

        # Конвертація в DataFrame для зручності
        df = pd.DataFrame(events)
        df['timestamp'] = pd.to_datetime(df['timestamp'])

        # Групування по сесіях
        sessions = self._group_by_sessions(df)

        # Витягування ознак активності
        activity_features = self._extract_activity_features(sessions)

        # Витягування ознак покупкової поведінки
        purchase_features = self._extract_purchase_features(df)

        # Витягування часових патернів
        temporal_features = self._extract_temporal_features(df)

        # Витягування товарних переваг
        product_features = self._extract_product_features(df)
```

```
    # Витягування технічних характеристик
    technical_features = self._extract_technical_features(df)

    return BehaviorFeatures(
        **activity_features,
        **purchase_features,
        **temporal_features,
        **product_features,
        **technical_features
    )

def _group_by_sessions(self, df: pd.DataFrame) -> List[pd.DataFrame]:
    """Групування events по сесіях (30 хв інактивності = нова сесія)"""
    sessions = []
    current_session = []
    last_timestamp = None

    for _, event in df.iterrows():
        if (last_timestamp is None or
            (event['timestamp'] — last_timestamp).seconds > 1800):  # 30 хв
            if current_session:
                sessions.append(pd.DataFrame(current_session))
            current_session = [event]
        else:
            current_session.append(event)

        last_timestamp = event['timestamp']

    if current_session:
        sessions.append(pd.DataFrame(current_session))

    return sessions

def _extract_activity_features(self, sessions: List[pd.DataFrame]) -> Dict:
    """Витягування ознак активності"""
    if not sessions:
        return {
            'total_sessions': 0,
            'avg_session_duration': 0.0,
            'pages_per_session': 0.0,
            'bounce_rate': 0.0,
            'return_visitor': False,
            'days_since_last_visit': 999
        }
    # Тривалість сесій
    session_durations = []
    pages_per_session = []
    bounced_sessions = 0
    for session in sessions:
        duration = (session['timestamp'].max() -
                    session['timestamp'].min()).seconds / 60.0  # хвилини
```

```
            session_durations.append(duration)
            pages = len(session[session['event_type'] == 'page_view'])
            pages_per_session.append(pages)
            if pages <= 1:
                bounced_sessions += 1
        # Розрахунок метрик
        total_sessions = len(sessions)
        avg_session_duration = np.mean(session_durations) if session_durations else 0
        avg_pages_per_session = np.mean(pages_per_session) if pages_per_session else 0
        bounce_rate = bounced_sessions / total_sessions if total_sessions > 0 else 0

        # Визначення returning visitor
        return_visitor = total_sessions > 1
        # Дні з останнього візиту
        last_visit = max(session['timestamp'].max() for session in sessions)
        days_since_last = (datetime.now() — last_visit).days
        return {
            'total_sessions': total_sessions,
            'avg_session_duration': avg_session_duration,
            'pages_per_session': avg_pages_per_session,
            'bounce_rate': bounce_rate,
            'return_visitor': return_visitor,
            'days_since_last_visit': days_since_last
        }
    def _extract_purchase_features(self, df: pd.DataFrame) -> Dict:
        """Витягування ознак покупкової поведінки"""
        purchase_events = df[df['event_type'] == 'purchase']
        cart_events = df[df['event_type'] == 'add_to_cart']
        if purchase_events.empty:
            return {
                'total_purchases': 0,
                'avg_order_value': 0.0,
                'purchase_frequency': 0.0,
                'cart_abandonment_rate': 1.0,
                'favorite_categories': [],
                'price_sensitivity': 0.5,
                'discount_usage_rate': 0.0,
                'payment_method_preference': 'unknown'
            }
        # Базові метрики покупок
        total_purchases = len(purchase_events)
        order_values = purchase_events['properties'].apply(
            lambda x: x.get('order_value', 0) if isinstance(x, dict) else 0
        )
        avg_order_value = order_values.mean()
        # Частота покупок (покупок на тиждень)
        date_range = (df['timestamp'].max() — df['timestamp'].min()).days
        purchase_frequency = total_purchases / max(date_range / 7, 1)
        # Cart abandonment rate
        cart_additions = len(cart_events)
        cart_abandonment_rate = 1 — (total_purchases / max(cart_additions, 1))
        # Улюблені категорії
```

```
    categories = []
    for _, event in purchase_events.iterrows():
        if isinstance(event['properties'], dict):
            category = event['properties'].get('category')
            if category:
                categories.append(category)
    favorite_categories = [cat for cat, count in
                           Counter(categories).most_common(3)]
    # Price sensitivity (на основі використання знижок)
    discount_events = df[df['event_type'] == 'discount_applied']
    discount_usage_rate = len(discount_events) / max(total_purchases, 1)
    price_sensitivity = min(discount_usage_rate * 2, 1.0)  # нормалізація
    # Preferred payment method
    payment_methods = []
    for _, event in purchase_events.iterrows():
        if isinstance(event['properties'], dict):
            method = event['properties'].get('payment_method')
            if method:
                payment_methods.append(method)
    payment_method_preference = (
        Counter(payment_methods).most_common(1)[0][0]
        if payment_methods else 'unknown'
    )
    return {
        'total_purchases': total_purchases,
        'avg_order_value': float(avg_order_value),
        'purchase_frequency': purchase_frequency,
        'cart_abandonment_rate': cart_abandonment_rate,
        'favorite_categories': favorite_categories,
        'price_sensitivity': price_sensitivity,
        'discount_usage_rate': discount_usage_rate,
        'payment_method_preference': payment_method_preference
    }
def _extract_temporal_features(self, df: pd.DataFrame) -> Dict:
    """Витягування часових патернів"""
    if df.empty:
        return {
            'preferred_shopping_hours': [],
            'weekend_vs_weekday_ratio': 1.0,
            'seasonal_activity_pattern': 'unknown',
            'time_to_purchase': 0.0
        }
    # Улюблені години для покупок
    df['hour'] = df['timestamp'].dt.hour
    hour_counts = df['hour'].value_counts()
    preferred_hours = hour_counts.head(3).index.tolist()
    # Weekend vs weekday activity
    df['is_weekend'] = df['timestamp'].dt.weekday >= 5
    weekend_events = len(df[df['is_weekend']])
    weekday_events = len(df[~df['is_weekend']])
    weekend_vs_weekday_ratio = (
        weekend_events / max(weekday_events, 1)
```

```
    )
    # Seasonal pattern (спрощено)
    df['month'] = df['timestamp'].dt.month
    month_counts = df['month'].value_counts()
    peak_month = month_counts.idxmax() if not month_counts.empty else 1
    if peak_month in [12, 1, 2]:
        seasonal_pattern = 'winter'
    elif peak_month in [3, 4, 5]:
        seasonal_pattern = 'spring'
    elif peak_month in [6, 7, 8]:
        seasonal_pattern = 'summer'
    else:
        seasonal_pattern = 'autumn'
    # Time to purchase (середній час від першого візиту до покупки)
    purchase_events = df[df['event_type'] == 'purchase']
    if not purchase_events.empty:
        first_visit = df['timestamp'].min()
        first_purchase = purchase_events['timestamp'].min()
        time_to_purchase = (first_purchase — first_visit).days
    else:
        time_to_purchase = 0.0
    return {
        'preferred_shopping_hours': preferred_hours,
        'weekend_vs_weekday_ratio': weekend_vs_weekday_ratio,
        'seasonal_activity_pattern': seasonal_pattern,
        'time_to_purchase': float(time_to_purchase)
    }
def _extract_product_features(self, df: pd.DataFrame) -> Dict:
    """Витягування товарних переваг"""
    # Спрощена реалізація
    return {
        'brand_loyalty_score': 0.5,
        'product_diversity_index': 0.7,
        'review_engagement_score': 0.3
    }
def _extract_technical_features(self, df: pd.DataFrame) -> Dict:
    """Витягування технічних характеристик"""
    if df.empty:
        return {
            'device_type': 'unknown',
            'browser_type': 'unknown',
            'geographic_region': 'unknown'
        }

    # Найчастіший тип пристрою
    devices = []
    browsers = []
    regions = []
    for _, event in df.iterrows():
        if isinstance(event['properties'], dict):
            props = event['properties']
            if 'device_type' in props:
```

```
                devices.append(props['device_type'])
            if 'browser' in props:
                browsers.append(props['browser'])
            if 'region' in props:
                regions.append(props['region'])
        device_type = (
            Counter(devices).most_common(1)[0][0]
            if devices else 'desktop'
        )
        browser_type = (
            Counter(browsers).most_common(1)[0][0]
            if browsers else 'chrome'
        )
        geographic_region = (
            Counter(regions).most_common(1)[0][0]
            if regions else 'unknown'
        )
        return {
            'device_type': device_type,
            'browser_type': browser_type,
            'geographic_region': geographic_region
        }
    def _get_default_features(self) -> BehaviorFeatures:
        """Дефолтні ознаки для нових користувачів"""
        return BehaviorFeatures(
            total_sessions=0,
            avg_session_duration=0.0,
            pages_per_session=0.0,
            bounce_rate=0.0,
            return_visitor=False,
            days_since_last_visit=999,
            total_purchases=0,
            avg_order_value=0.0,
            purchase_frequency=0.0,
            cart_abandonment_rate=1.0,
            favorite_categories=[],
            price_sensitivity=0.5,
            discount_usage_rate=0.0,
            payment_method_preference='unknown',
            preferred_shopping_hours=[],
            weekend_vs_weekday_ratio=1.0,
            seasonal_activity_pattern='unknown',
            time_to_purchase=0.0,
            brand_loyalty_score=0.5,
            product_diversity_index=0.5,
            review_engagement_score=0.5,
            device_type='desktop',
            browser_type='chrome',
            geographic_region='unknown'
        )
```

ДОДАТОК Г

Database Initialization — Ініціалізація бази даних

```
import asyncio
import asyncpg
from sqlalchemy import create_engine, MetaData
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker
from app.models.product import Product, ProductVariant
from app.core.config import settings
async def init_database():
  """Ініціалізація бази даних та створення таблиць"""
  try:
    # Підключення до бази даних
    engine = create_async_engine(
      settings.DATABASE_URL,
      echo=True,
      future=True
    )
    # Створення таблиць
    async with engine.begin() as conn:
      await conn.run_sync(MetaData().create_all)
    # Заповнення початковими даними
    await seed_initial_data(engine)
    print("Database initialized successfully")
  except Exception as e:
    print(f"Database initialization failed: {e}")
    raise
async def seed_initial_data(engine):
  """Заповнення бази початковими даними"""
  async_session = sessionmaker(
    engine, class_=AsyncSession, expire_on_commit=False
  )
  async with async_session() as session:
    # Перевірка чи є вже дані
    existing_products = await session.execute(
      "SELECT COUNT(*) FROM products"
    )
    count = existing_products.scalar()
    if count > 0:
      print("Database already contains data, skipping seed")
      return
    # Створення тестових товарів
    sample_products = [
      Product(
        name="Laptop Dell XPS 13",
        description="High-performance ultrabook",
        base_price=1299.99,
        category="Electronics",
        sku="DELL-XPS13-001",
```

```
            stock_quantity=50,
            is_active=True
        ),
        Product(
            name="iPhone 15 Pro",
            description="Latest Apple smartphone",
            base_price=999.99,
            category="Electronics",
            sku="APPLE-IP15P-001",
            stock_quantity=100,
            is_active=True
        ),
        Product(
            name="Nike Air Max 270",
            description="Comfortable running shoes",
            base_price=150.00,
            category="Footwear",
            sku="NIKE-AM270-001",
            stock_quantity=75,
            is_active=True
        )
    ]
    for product in sample_products:
        session.add(product)
    await session.commit()
    print(f"Seeded {len(sample_products)} products")
async def create_indexes():
  """Створення індексів для оптимізації запитів"""
  engine = create_async_engine(settings.DATABASE_URL)
  indexes = [
    "CREATE INDEX IF NOT EXISTS idx_products_category ON products(category);",
    "CREATE INDEX IF NOT EXISTS idx_products_sku ON products(sku);",
    "CREATE INDEX IF NOT EXISTS idx_products_active ON products(is_active);",
    "CREATE INDEX IF NOT EXISTS idx_product_variants_product_id ON
product_variants(product_id);",
    "CREATE INDEX IF NOT EXISTS idx_products_price_range ON products(base_price)
WHERE is_active = true;"
  ]
  async with engine.begin() as conn:
    for index_sql in indexes:
      await conn.execute(index_sql)
  print("Database indexes created successfully")
async def setup_database_triggers():
  """Налаштування тригерів для автоматичного оновлення"""
  engine = create_async_engine(settings.DATABASE_URL)
  # Тригер для автоматичного оновлення updated_at
  trigger_sql = """
  CREATE OR REPLACE FUNCTION update_updated_at_column()
  RETURNS TRIGGER AS $$
  BEGIN
    NEW.updated_at = CURRENT_TIMESTAMP;
    RETURN NEW;
```

```
    END;
    $$ language 'plpgsql';
    DROP TRIGGER IF EXISTS update_products_updated_at ON products;
    CREATE TRIGGER update_products_updated_at
        BEFORE UPDATE ON products
        FOR EACH ROW
        EXECUTE FUNCTION update_updated_at_column();
    """
    async with engine.begin() as conn:
        await conn.execute(trigger_sql)
    print("Database triggers created successfully")
if __name__ == "__main__":
    asyncio.run(init_database())
    asyncio.run(create_indexes())
    asyncio.run(setup_database_triggers())
Міграції бази даних
from alembic import command
from alembic.config import Config
import os
def run_migrations():
    """Запуск міграцій Alembic"""
    alembic_cfg = Config("alembic.ini")
    alembic_cfg.set_main_option("script_location", "migrations")
    alembic_cfg.set_main_option("sqlalchemy.url", settings.DATABASE_URL)
    try:
        # Запуск міграцій до останньої версії
        command.upgrade(alembic_cfg, "head")
        print("Migrations completed successfully")
    except Exception as e:
        print(f"Migration failed: {e}")
        raise
def create_migration(message: str):
    """Створення нової міграції"""
    alembic_cfg = Config("alembic.ini")
    command.revision(alembic_cfg, message=message, autogenerate=True)
    print(f"Migration '{message}' created successfully")
Backup та відновлення
import subprocess
from datetime import datetime
async def backup_database():
    """Створення резервної копії бази даних"""
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    backup_file = f"backup_products_{timestamp}.sql"
    try:
        # Використання pg_dump для створення backup
        subprocess.run([
            "pg_dump",
            settings.DATABASE_URL,
            "-f", backup_file,
            "--verbose"
        ], check=True)
        print(f"Database backup created: {backup_file}")
```

```
        return backup_file
    except subprocess.CalledProcessError as e:
        print(f"Backup failed: {e}")
        raise
async def restore_database(backup_file: str):
    """Відновлення бази даних з резервної копії"""
    try:
        subprocess.run([
            "psql",
            settings.DATABASE_URL,
            "-f", backup_file,
            "--verbose"
        ], check=True)
        print(f"Database restored from: {backup_file}")
    except subprocess.CalledProcessError as e:
        print(f"Restore failed: {e}")
    raise
```

ДОДАТОК Д

ML Service API — Інтеграція між сервісами

```
API-based синхронізація для операцій читання
# ML Service запитує дані товарів у Product Service
async def get_product_data(product_id: int) -> ProductData:
  async with httpx.AsyncClient() as client:
    response = await client.get(
      f"{PRODUCT_SERVICE_URL}/api/v1/products/{product_id}"
    )
    if response.status_code == 200:
      return ProductData(**response.json())
    else:
      raise HTTPException(
        status_code=response.status_code,
        detail="Failed to fetch product data"
      )
# Отримання поведінкових даних користувача
async def get_user_behavior(user_id: int) -> UserBehavior:
  async with httpx.AsyncClient() as client:
    response = await client.get(
      f"{TRACKING_SERVICE_URL}/api/v1/behavior/{user_id}"
    )
    return UserBehavior(**response.json())
# Генерація персоналізованих рекомендацій цін
async def generate_price_recommendations(
  user_id: int,
  product_id: int
) -> PriceRecommendation:
  # Отримуємо дані товару
  product_data = await get_product_data(product_id)
  # Отримуємо поведінкові дані користувача
  user_behavior = await get_user_behavior(user_id)
  # Застосовуємо ML модель
  ml_engine = FederatedMLEngine()
  recommendation = await ml_engine.predict_optimal_price(
    product_data, user_behavior
  )
  return recommendation
Structured Logging для моніторингу
import structlog
logger = structlog.get_logger()
async def process_pricing_request(user_id: int, product_id: int):
  logger.info(
    "Processing pricing request",
    user_id=user_id,
    product_id=product_id,
    timestamp=datetime.utcnow().isoformat()
  )
   try:
```

```
        recommendation = await generate_price_recommendations(
            user_id, product_id
        )
        logger.info(
            "Pricing request completed",
            user_id=user_id,
            product_id=product_id,
            recommended_price=recommendation.price,
            confidence=recommendation.confidence
        )
        return recommendation
    except Exception as e:
        logger.error(
            "Pricing request failed",
            user_id=user_id,
            product_id=product_id,
            error=str(e),
            error_type=type(e).__name__
        )
        raise
Health Check Endpoints
from fastapi import APIRouter, HTTPException
from datetime import datetime
router = APIRouter()
@router.get("/health")
async def health_check():
    """Перевірка стану сервісу"""
    try:
        # Перевірка підключення до бази даних
        await check_database_connection()
        # Перевірка доступності зовнішніх сервісів
        await check_external_services()
        # Перевірка стану ML моделей
        await check_ml_models()
             return {
            "status": "healthy",
            "timestamp": datetime.utcnow().isoformat(),
            "version": "1.0.0",
            "dependencies": {
                "database": "healthy",
                "product_service": "healthy",
                "tracking_service": "healthy",
                "ml_models": "loaded"
            }
        }
    except Exception as e:
        raise HTTPException(
            status_code=503,
            detail=f"Service unhealthy: {str(e)}"
        )
async def check_database_connection():
    """Перевірка підключення до бази даних"""
```

```
    # Реалізація перевірки БД
    pass
async def check_external_services():
    """Перевірка доступності зовнішніх сервісів"""
    services = [
        ("Product Service", PRODUCT_SERVICE_URL),
        ("Tracking Service", TRACKING_SERVICE_URL)
    ]
      for service_name, url in services:
        try:
            async with httpx.AsyncClient(timeout=5.0) as client:
                response = await client.get(f"{url}/health")
                if response.status_code != 200:
                    raise Exception(f"{service_name} returned {response.status_code}")
        except Exception as e:
            raise Exception(f"{service_name} check failed: {str(e)}")
async def check_ml_models():
    """Перевірка стану ML моделей"""
    # Перевірка завантаження моделей
    pass
```

ДОДАТОК Е

Circuit Breaker Pattern — Захист від каскадних відмов

```
import time
import asyncio
from enum import Enum
from typing import Callable, Any
import logging
logger = logging.getLogger(__name__)
class CircuitState(Enum):
  CLOSED = "CLOSED"
  OPEN = "OPEN"
  HALF_OPEN = "HALF_OPEN"
class CircuitBreaker:
  def __init__(self, failure_threshold=5, timeout=60):
    self.failure_threshold = failure_threshold
    self.timeout = timeout
    self.failure_count = 0
    self.last_failure_time = None
    self.state = CircuitState.CLOSED
  async def call(self, func, *args, **kwargs):
    if self.state == CircuitState.OPEN:
      if time.time() — self.last_failure_time > self.timeout:
        self.state = CircuitState.HALF_OPEN
        logger.info("Circuit breaker moved to HALF_OPEN state")
      else:
        raise CircuitBreakerOpenException(
          "Circuit breaker is OPEN"
        )
    try:
      result = await func(*args, **kwargs)
      self.record_success()
      return result
    except Exception as e:
      self.record_failure()
      raise e
  def record_failure(self):
    self.failure_count += 1
    self.last_failure_time = time.time()

    if self.failure_count >= self.failure_threshold:
      self.state = CircuitState.OPEN
      logger.warning(
        f"Circuit breaker OPENED after {self.failure_count} failures"
      )
  def record_success(self):
    if self.state == CircuitState.HALF_OPEN:
      self.state = CircuitState.CLOSED
      logger.info("Circuit breaker CLOSED — service recovered")
```

```
        self.failure_count = 0
        self.last_failure_time = None
class CircuitBreakerOpenException(Exception):
    pass

Retry Pattern з експоненційною затримкою
import random
async def retry_with_backoff(
    func: Callable,
    max_retries: int = 3,
    base_delay: float = 1.0,
    max_delay: float = 60.0,
    exponential_base: float = 2.0,
    jitter: bool = True
):
    """
    Retry функції з експоненційною затримкою та jitter
    """
    for attempt in range(max_retries):
        try:
            return await func()
        except Exception as e:
            if attempt == max_retries — 1:
                logger.error(f"All {max_retries} retry attempts failed")
                raise e

            # Розрахунок затримки з експоненційним зростанням
            delay = min(
                base_delay * (exponential_base ** attempt),
                max_delay
            )

            # Додавання jitter для уникнення thundering herd
            if jitter:
                delay = delay * (0.5 + random.random() * 0.5)

            logger.warning(
                f"Attempt {attempt + 1} failed: {str(e)}. "
                f"Retrying in {delay:.2f} seconds..."
            )

            await asyncio.sleep(delay)
# Декоратор для автоматичного retry
def with_retry(max_retries=3, base_delay=1.0):
    def decorator(func):
        async def wrapper(*args, **kwargs):
            return await retry_with_backoff(
                lambda: func(*args, **kwargs),
                max_retries=max_retries,
                base_delay=base_delay
            )
        return wrapper
```

```
  return decorator

Інтеграція з HTTP клієнтами
import httpx
from typing import Optional
class ResilientHTTPClient:
  def __init__(
      self,
      base_url: str,
      timeout: float = 30.0,
      circuit_breaker_threshold: int = 5,
      circuit_breaker_timeout: int = 60
  ):
      self.base_url = base_url
      self.timeout = timeout
      self.circuit_breaker = CircuitBreaker(
          failure_threshold=circuit_breaker_threshold,
          timeout=circuit_breaker_timeout
      )
      self.client = httpx.AsyncClient(
          base_url=base_url,
          timeout=timeout
      )
  @with_retry(max_retries=3, base_delay=1.0)
  async def get(self, endpoint: str, **kwargs) -> httpx.Response:
      """GET запит з circuit breaker та retry"""
      return await self.circuit_breaker.call(
          self._make_request, "GET", endpoint, **kwargs
      )
  @with_retry(max_retries=3, base_delay=1.0)
  async def post(self, endpoint: str, **kwargs) -> httpx.Response:
      """POST запит з circuit breaker та retry"""
      return await self.circuit_breaker.call(
          self._make_request, "POST", endpoint, **kwargs
      )
  async def _make_request(
      self,
      method: str,
      endpoint: str,
      **kwargs
  ) -> httpx.Response:
      """Виконання HTTP запиту"""
      response = await self.client.request(method, endpoint, **kwargs)
      # Перевірка статус коду
      if response.status_code >= 500:
          raise httpx.HTTPStatusError(
              f"Server error: {response.status_code}",
              request=response.request,
              response=response
          )
      return response
  async def close(self):
```

```
        """Закриття HTTP клієнта"""
        await self.client.aclose()
# Використання в сервісах
class ProductServiceClient:
    def __init__(self):
        self.client = ResilientHTTPClient(
            base_url=settings.PRODUCT_SERVICE_URL,
            circuit_breaker_threshold=3,
            circuit_breaker_timeout=30
        )
    async def get_product(self, product_id: int) -> dict:
        """Отримання товару з Product Service"""
        try:
            response = await self.client.get(f"/api/v1/products/{product_id}")
            return response.json()
        except CircuitBreakerOpenException:
            logger.warning("Product Service circuit breaker is open")
            # Повернення кешованих даних або дефолтних значень
            return await self.get_cached_product(product_id)
        except Exception as e:
            logger.error(f"Failed to get product {product_id}: {str(e)}")
            raise
    async def get_cached_product(self, product_id: int) -> dict:
        """Отримання товару з кешу як fallback"""
        # Реалізація отримання з Redis кешу
        pass
Моніторинг Circuit Breaker
from dataclasses import dataclass
from datetime import datetime
from typing import Dict, List
@dataclass
class CircuitBreakerMetrics:
    service_name: str
    state: CircuitState
    failure_count: int
    success_count: int
    last_failure_time: Optional[datetime]
    total_requests: int
 class CircuitBreakerMonitor:
    def __init__(self):
        self.circuit_breakers: Dict[str, CircuitBreaker] = {}
        self.metrics: Dict[str, CircuitBreakerMetrics] = {}
     def register_circuit_breaker(
        self,
        service_name: str,
        circuit_breaker: CircuitBreaker
    ):
        """Реєстрація circuit breaker для моніторингу"""
        self.circuit_breakers[service_name] = circuit_breaker
        self.metrics[service_name] = CircuitBreakerMetrics(
            service_name=service_name,
            state=circuit_breaker.state,
```

```
            failure_count=0,
            success_count=0,
            last_failure_time=None,
            total_requests=0
        )
    def get_metrics(self) -> List[CircuitBreakerMetrics]:
        """Отримання метрик всіх circuit breaker"""
        metrics = []
        for service_name, cb in self.circuit_breakers.items():
            metric = self.metrics[service_name]
            metric.state = cb.state
            metric.failure_count = cb.failure_count
            if cb.last_failure_time:
                metric.last_failure_time = datetime.fromtimestamp(
                    cb.last_failure_time
                )
            metrics.append(metric)
        return metrics
    def get_health_status(self) -> Dict[str, str]:
        """Статус здоров'я всіх сервісів"""
        status = {}
        for service_name, cb in self.circuit_breakers.items():
            if cb.state == CircuitState.OPEN:
                status[service_name] = "unhealthy"
            elif cb.state == CircuitState.HALF_OPEN:
                status[service_name] = "recovering"
            else:
                status[service_name] = "healthy"
        return status
# Глобальний монітор
circuit_monitor = CircuitBreakerMonitor()
```

ДОДАТОК Ж

Docker Configuration — Контейнеризація сервісів

Dockerfile для ML Service

```
FROM python:3.11-slim
WORKDIR /app
# Встановлення системних залежностей
RUN apt-get update && apt-get install -y \
  gcc \
  g++ \
  curl \
  && rm -rf /var/lib/apt/lists/*
# Встановлення Python залежностей
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Копіювання коду додатку
COPY app/ ./app/
# Створення директорії для ML моделей
RUN mkdir -p /app/data/models
# Налаштування змінних середовища
ENV PYTHONPATH=/app
ENV ML_MODELS_PATH=/app/data/models
ENV PYTHONUNBUFFERED=1
# Створення non-root користувача
RUN useradd --create-home --shell /bin/bash app \
  && chown -R app:app /app
USER app
# Відкриття порту
EXPOSE 8002
# Health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
  CMD curl -f http://localhost:8002/health || exit 1
# Команда запуску
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8002", "--reload"]
```

Docker Compose для розробки

```
version: '3.8'
services:
 postgres:
  image: postgres:15-alpine
  environment:
   POSTGRES_DB: pricing_db
   POSTGRES_USER: pricing_user
   POSTGRES_PASSWORD: pricing_password
   POSTGRES_MULTIPLE_DATABASES: pricing_db,pricing_products,pricing_ml_analytics
  ports:
   — "5432:5432"
  volumes:
   — postgres_data:/var/lib/postgresql/data
   — ./database/init-multiple-databases.sh:/docker-entrypoint-initdb.d/init-multiple-databases.sh
  healthcheck:
```

```
    test: ["CMD-SHELL", "pg_isready -U pricing_user -d pricing_db"]
    interval: 10s
    timeout: 5s
    retries: 5
  restart: unless-stopped
 redis:
  image: redis:7-alpine
  ports:
   — "6379:6379"
  healthcheck:
   test: ["CMD", "redis-cli", "ping"]
   interval: 10s
   timeout: 5s
   retries: 5
  restart: unless-stopped
 user-service:
  build:
   context: ./services/user-service
   dockerfile: Dockerfile.dev
  ports:
   — "8001:8001"
  environment:
   — DATABASE_URL=postgresql://pricing_user:pricing_password@postgres:5432/pricing_db
   — REDIS_URL=redis://redis:6379/0
   — JWT_SECRET_KEY=your-secret-key-here
   — JWT_ALGORITHM=HS256
   — ACCESS_TOKEN_EXPIRE_MINUTES=30
  depends_on:
   postgres:
    condition: service_healthy
   redis:
    condition: service_healthy
  volumes:
   — ./services/user-service/app:/app/app
  restart: unless-stopped
  healthcheck:
   test: ["CMD", "curl", "-f", "http://localhost:8001/health"]
   interval: 30s
   timeout: 10s
   retries: 3
   start_period: 40s
 ml-service:
  build:
   context: ./services/ml-service
   dockerfile: Dockerfile.dev
  ports:
   — "8002:8002"
  environment:
   —
ML_DATABASE_URL=postgresql://pricing_user:pricing_password@postgres:5432/pricing_ml_a
nalytics
   — PRODUCT_SERVICE_URL=http://product-service:8003
```

```
    — USER_SERVICE_URL=http://user-service:8001
    — REDIS_URL=redis://redis:6379/1
    — ML_MODELS_PATH=/app/data/models
  depends_on:
    postgres:
      condition: service_healthy
    redis:
      condition: service_healthy
  volumes:
    — ./services/ml-service/app:/app/app
    — ml_models:/app/data/models
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8002/health"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 60s
product-service:
  build:
    context: ./services/product-service
    dockerfile: Dockerfile.dev
  ports:
    — "8003:8003"
  environment:
    —
DATABASE_URL=postgresql://pricing_user:pricing_password@postgres:5432/pricing_products
    — REDIS_URL=redis://redis:6379/2
    — WOOCOMMERCE_URL=${WOOCOMMERCE_URL}
    — WOOCOMMERCE_KEY=${WOOCOMMERCE_KEY}
    — WOOCOMMERCE_SECRET=${WOOCOMMERCE_SECRET}
  depends_on:
    postgres:
      condition: service_healthy
    redis:
      condition: service_healthy
  volumes:
    — ./services/product-service/app:/app/app
  restart: unless-stopped
tracking-service:
  build:
    context: ./services/tracking-service
    dockerfile: Dockerfile.dev
  ports:
    — "8004:8004"
  environment:
    — DATABASE_URL=postgresql://pricing_user:pricing_password@postgres:5432/pricing_db
    — REDIS_URL=redis://redis:6379/3
    — ML_SERVICE_URL=http://ml-service:8002
  depends_on:
    postgres:
      condition: service_healthy
```

```
    redis:
      condition: service_healthy
  volumes:
    — ./services/tracking-service/app:/app/app
  restart: unless-stopped
 websocket-service:
  build:
    context: ./services/websocket-service
    dockerfile: Dockerfile.dev
  ports:
    — "8005:8005"
  environment:
    — REDIS_URL=redis://redis:6379/4
    — USER_SERVICE_URL=http://user-service:8001
  depends_on:
    redis:
      condition: service_healthy
  volumes:
    — ./services/websocket-service/app:/app/app
  restart: unless-stopped
 api-gateway:
  build:
    context: ./api-gateway
    dockerfile: Dockerfile.dev
  ports:
    — "8000:8000"
  environment:
    — USER_SERVICE_URL=http://user-service:8001
    — PRODUCT_SERVICE_URL=http://product-service:8003
    — ML_SERVICE_URL=http://ml-service:8002
    — TRACKING_SERVICE_URL=http://tracking-service:8004
    — WEBSOCKET_SERVICE_URL=http://websocket-service:8005
    — REDIS_URL=redis://redis:6379/5
  depends_on:
    — user-service
    — product-service
    — ml-service
    — tracking-service
    — websocket-service
  volumes:
    — ./api-gateway/app:/app/app
  restart: unless-stopped
volumes:
 postgres_data:
 ml_models:
networks:
 default:
   name: pricing-network
Production Docker Compose
version: '3.8'
services:
 postgres:
```

```
  image: postgres:15-alpine
  environment:
    POSTGRES_DB: ${POSTGRES_DB}
    POSTGRES_USER: ${POSTGRES_USER}
    POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
  volumes:
    — postgres_data:/var/lib/postgresql/data
  deploy:
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
      reservations:
        cpus: '0.5'
        memory: 512M
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
    interval: 30s
    timeout: 10s
    retries: 3
  restart: unless-stopped
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "3"
redis:
  image: redis:7-alpine
  command: redis-server --appendonly yes --maxmemory 256mb --maxmemory-policy allkeys-lru
  volumes:
    — redis_data:/data
  deploy:
    resources:
      limits:
        cpus: '0.5'
        memory: 512M
      reservations:
        cpus: '0.25'
        memory: 256M
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 30s
    timeout: 10s
    retries: 3
  restart: unless-stopped
ml-service:
  image: pricing-system/ml-service:${VERSION:-latest}
  environment:
    —
ML_DATABASE_URL=postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@post
gres:5432/${POSTGRES_DB}
    — REDIS_URL=redis://redis:6379/1
```

```
    — PRODUCT_SERVICE_URL=http://product-service:8003
    — USER_SERVICE_URL=http://user-service:8001
  volumes:
    — ml_models:/app/data/models
  deploy:
    replicas: 2
    resources:
      limits:
        cpus: '1.0'
        memory: 1G
      reservations:
        cpus: '0.5'
        memory: 512M
    update_config:
      parallelism: 1
      delay: 10s
      failure_action: rollback
      order: start-first
    restart_policy:
      condition: on-failure
      delay: 5s
      max_attempts: 3
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8002/health"]
    interval: 30s
    timeout: 10s
    retries: 3
    start_period: 60s
  logging:
    driver: "json-file"
    options:
      max-size: "10m"
      max-file: "3"
volumes:
 postgres_data:
 redis_data:
 ml_models:
networks:
 default:
   name: pricing-network
   driver: bridge
Health Check Configuration
# Health check для всіх сервісів
healthcheck:
 test: ["CMD", "curl", "-f", "http://localhost:8001/health"]
 interval: 30s
 timeout: 10s
 retries: 3
 start_period: 40s
Logging Configuration
# Централізоване логування через Docker logging drivers
logging:
```

```
 driver: "json-file"
 options:
   max-size: "10m"
   max-file: "3"
   labels: "service,environment"
Resource Limits
# Ресурсні обмеження для стабільності
deploy:
 resources:
   limits:
     cpus: '0.5'
     memory: 512M
   reservations:
     cpus: '0.25'
     memory: 256M
 restart_policy:
   condition: on-failure
   delay: 5s
   max_attempts: 3
Multi-stage Dockerfile для Production
# Multi-stage build для оптимізації розміру образу
FROM python:3.11-slim as builder
WORKDIR /app
# Встановлення build залежностей
RUN apt-get update && apt-get install -y \
   gcc \
   g++ \
   && rm -rf /var/lib/apt/lists/*
# Встановлення Python залежностей
COPY requirements.txt .
RUN pip install --no-cache-dir --user -r requirements.txt
# Production stage
FROM python:3.11-slim
WORKDIR /app
# Копіювання встановлених пакетів з builder stage
COPY --from=builder /root/.local /root/.local
# Копіювання коду додатку
COPY app/ ./app/
# Створення non-root користувача
RUN useradd --create-home --shell /bin/bash app \
   && chown -R app:app /app
USER app
# Налаштування PATH для pip пакетів
ENV PATH=/root/.local/bin:$PATH
ENV PYTHONPATH=/app
ENV PYTHONUNBUFFERED=1
EXPOSE 8002
HEALTHCHECK --interval=30s --timeout=10s --start-period=40s --retries=3 \
   CMD curl -f http://localhost:8002/health || exit 1
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8002"]
```

ДОДАТОК К

Архітектура ML-системи для динамічного ціноутворення

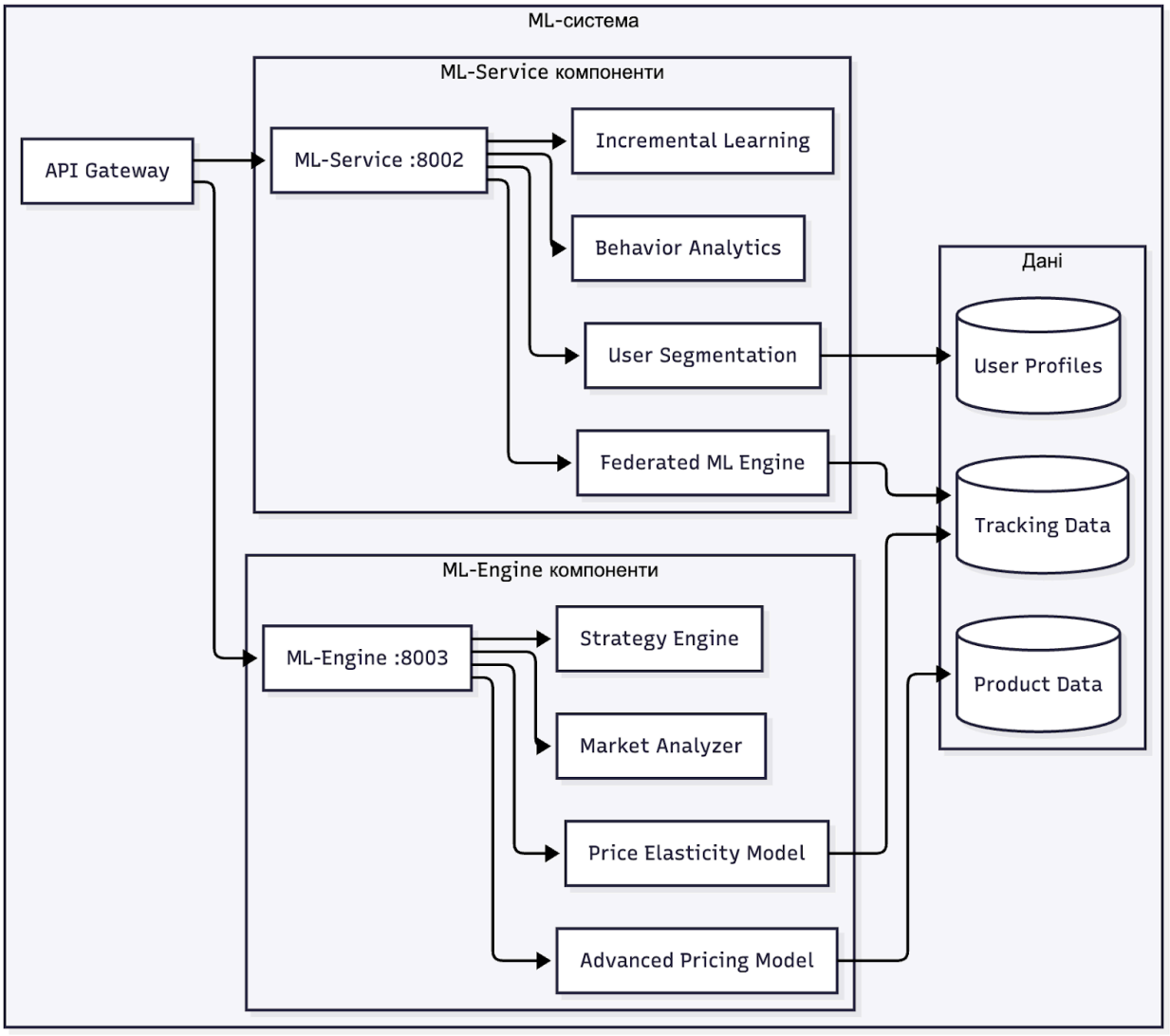

Рисунок К.1 — Архітектура ML-системи для динамічного ціноутворення

ДОДАТОК Л

UML-діаграми класів та послідовності

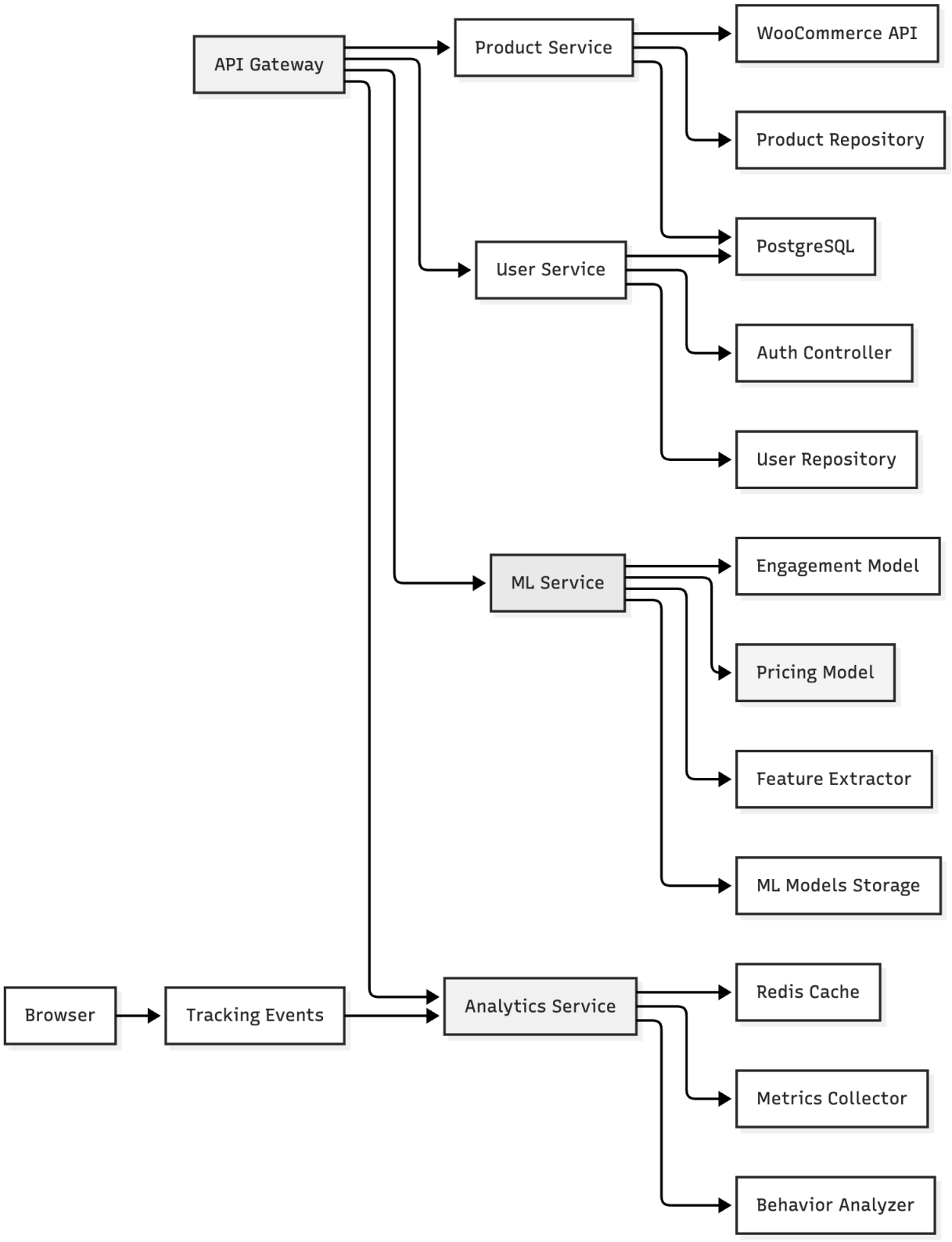

Рисунок Л.1 — Структура основних класів систем

ДОДАТОК М

Діаграма процесу збору та обробки конкурентних даних

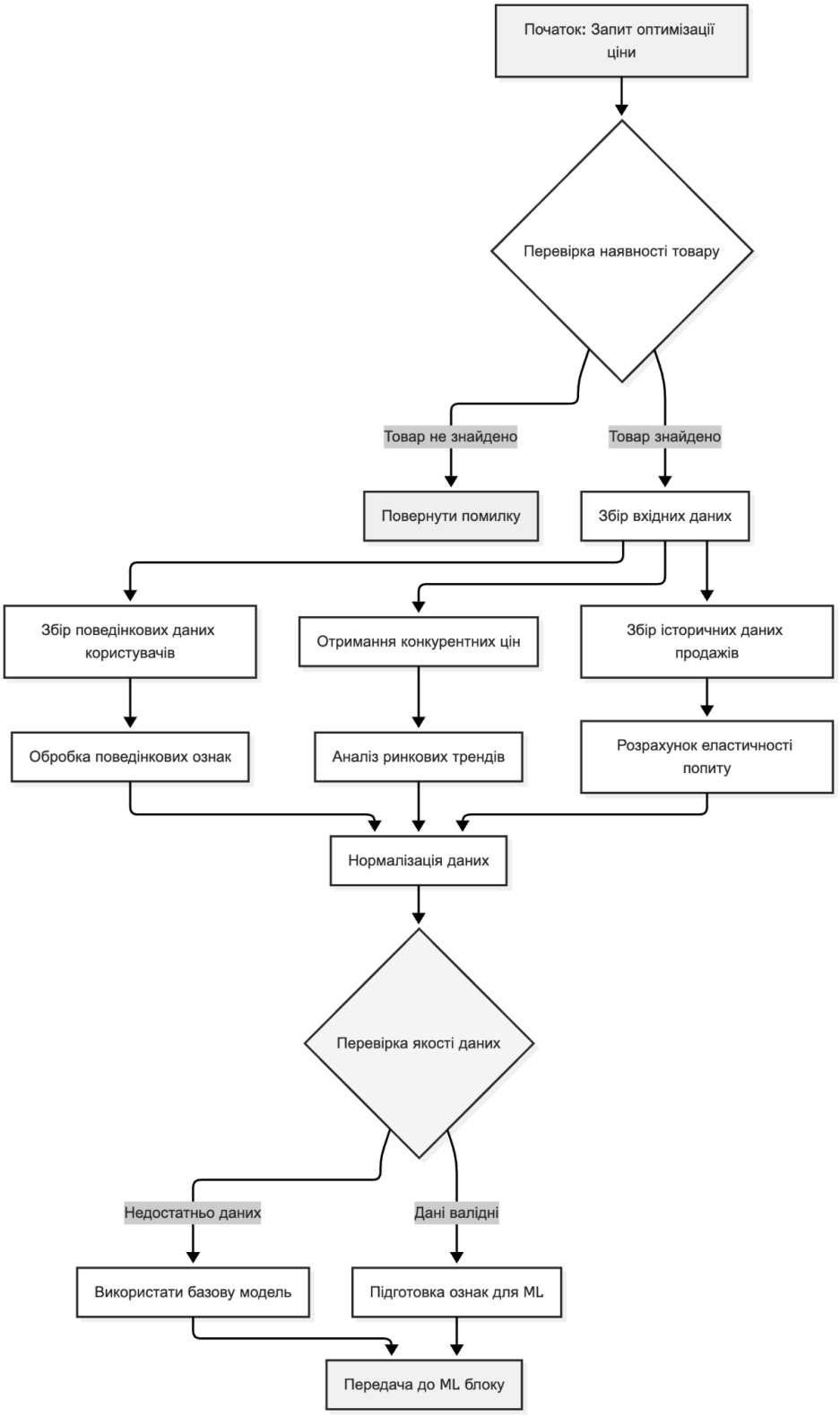

Рисунок М.1 — Збір та підготовка даних для ML моделі