МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ І ПРИКЛАДНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КИЗЬ ОЛЕКСАНДР АНАТОЛІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
д-р. техн. наук, професор
_____Наталія ВЕСЕЛОВСЬКА
«____»_____ 2026 р.

РОЗРОБКА ВЕБОРІЄНТОВАНОЇ ПЛАТФОРМИ ДЛЯ ОРГАНІЗАЦІЇ ВЗАЄМОДІЇ УЧАСНИКІВ КНИЖКОВОГО РИНКУ

(комплексний ч.2)

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Наталія ВЕСЕЛОВСЬКА, професор кафедри
інформаційних технологій,
д. т. н., професор
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2026

АНОТАЦІЯ

Кизь Олександр Анатолійович. Веб-платформа для купівлі та обміну книгами: архітектура та стратегія автоматизованого розгортання. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця 2026.

У кваліфікаційній (бакалаврській) роботі спроєктовано архітектуру та розроблено стратегію автоматизованого розгортання C2C-платформи для обміну книгами. З використанням технологій Docker, CI/CD конвеєрів та Nginx створено ізольоване та відмовостійке середовище. Розгортання інфраструктури здійснено на базі ОС Linux у хмарному середовищі Google Cloud Platform.

Ключові слова: веб-платформа, DevOps, автоматизація розгортання, контейнеризація, Docker, CI/CD, Linux.

ABSTRACT

Kyz Oleksandr Anatoliyovych. Web platform for buying and exchanging books "Readvia": architecture and automated deployment strategy. Specialty 122 "Computer Science", educational program "Computer Science". Vasyl Stus Donetsk National University, Vinnytsia 2026.

In the qualification (bachelor's) work the architecture is designed and an automated deployment strategy for the C2C book exchange platform "Readvia" is developed. Using Docker technologies, CI/CD pipelines, and Nginx, an isolated and fault-tolerant environment was created. The infrastructure deployment was implemented on a Linux OS in the Google Cloud Platform environment.

Keywords: web platform, DevOps, automated deployment, containerization, Docker, CI/CD, Linux.

ЗМІСТ

ВСТУП

Актуальність теми: Сучасний стан розвитку світової економіки характеризується стрімким переходом традиційних бізнес-моделей у цифрове середовище. Сфера електронної комерції e-commerce демонструє стабільне зростання, що зумовлює підвищення вимог до якості, швидкості розробки та надійності програмних продуктів. Особливої актуальності набувають спеціалізовані C2C-платформи, що орієнтовані на вторинний ринок товарів, зокрема вживаних книг. Проєкт покликаний вирішити проблему ефективного обміну та купівлі-продажу літератури, забезпечуючи при цьому високий рівень користувацького досвіду через механізми гейміфікації та соціальної взаємодії.

З технічної точки зору, успіх такої платформи залежить не лише від функціональних можливостей, а й від архітектурної гнучкості та стратегії розгортання. Традиційні методи адміністрування стають неефективними при зростанні навантаження, що робить впровадження методології DevOps критично необхідним. Використання технологій контейнеризації, автоматизації конвеєрів доставки CI/CD та сучасних засобів моніторингу дозволяє забезпечити безперервну роботу сервісу та мінімізувати ризики, пов'язані з людським фактором.

Мета роботи - розробка архітектури та автоматизованої стратегії розгортання для e-commerce платформи, що забезпечить масштабованість, відмовостійкість та високу швидкість доставки оновлень програмного забезпечення.

Для досягнення поставленої мети було визначено такі завдання:

1. Провести аналіз предметної області, існуючих аналогів та обґрунтувати вибір технологічного стеку і архітектурного підходу.
2. Спроектувати структуру бази даних та мережеву топологію системи.
3. Розробити програмний прототип платформи з використанням сучасних фреймворків.

4. Реалізувати інфраструктуру контейнеризації на базі Docker та налаштувати автоматизований CI/CD конвеєр.
5. Впровадити систему моніторингу та розробити стратегію безпеки і резервного копіювання даних.

Об’єктом дослідження є процеси проектування, розробки та автоматизованого розгортання веб-орієнтованих систем у середовищі Linux.

Предметом дослідження є архітектурні патерни, інструменти контейнеризації Docker, методи автоматизації інфраструктури GitHub Actions та засоби адміністрування серверних систем для забезпечення життєвого циклу e-commerce платформи.

Методи дослідження. У роботі використано методи системного аналізу для вивчення предметної області, принципи об’єктно-орієнтованого проектування, методи контейнеризації обчислювальних ресурсів та практики Infrastructure as Code для управління конфігураціями серверів.

Практичне значення одержаних результатів полягає у створенні готової до експлуатації архітектурної моделі та налаштованого середовища розгортання, які дозволяють значно скоротити час виходу продукту на ринок та забезпечити стабільну роботу системи при змінних навантаженнях. Розроблені скрипти автоматизації та конфігураційні файли можуть бути адаптовані для аналогічних веб-проєктів.

Апробація результатів дослідження. Основні ідеї, архітектурні рішення та концепція розгортання веб-платформи пройшли практичну апробацію в межах участі у грантовій програмі / конкурсі стартапів від благодійного фонду «Право на захист» програми «Право на бізнес». Проєкт був представлений експертній комісії, де отримав позитивну оцінку щодо його актуальності, технічної реалізації та перспектив подальшого масштабування. Додаток Д.

Структура роботи. Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 71 сторінку друкованого тексту, з яких 55 сторінок складає основна частина. Список використаних посилань налічує 40 найменувань.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Аналіз архітектурних підходів: порівняння монолітної та мікросервісної архітектур

Проєктування сучасних високонавантажених веб-систем, зокрема у сфері електронної комерції, вимагає глибокого аналізу та вибору оптимального архітектурного патерну на ранніх етапах життєвого циклу розробки програмного забезпечення SDLC [18]. Від прийнятого архітектурного рішення залежить не лише швидкість створення мінімально життєздатного продукту, але й показники відмовостійкості, можливості горизонтального масштабування та ефективність застосування методологій безперервної інтеграції та розгортання [1]. У сучасній інженерії програмного забезпечення домінують два фундаментальні підходи: монолітна та мікросервісна архітектури [5].

Монолітна архітектура є традиційним і найбільш поширеним підходом для запуску нових проєктів. У межах цієї парадигми всі компоненти системи: користувацький інтерфейс, рівень бізнес-логіки та рівень доступу до даних об’єднані в єдину кодову базу і функціонують як один неподільний процес [6]. Головною перевагою монолітного підходу є простота ініціалізації проєкту. Завдяки єдиній кодовій базі розробникам значно легше проводити наскрізне end-to-end тестування та відстежувати потоки виконання коду [19]. Розгортання такої системи також виглядає тривіально: адміністратору достатньо скопіювати один скомпільований артефакт або директорію на сервер і запустити єдиний процес веб-сервера. Однак, зі зростанням обсягу коду та збільшенням функціональних вимог, монолітна архітектура починає демонструвати критичні недоліки. Найбільшою проблемою є висока зв'язність компонентів. Будь-яка незначна зміна, наприклад, оновлення логіки відображення каталогу книг, вимагає перекомпіляції та перезапуску всієї системи, що створює ризики тимчасової недоступності сервісу [5]. Крім того, моноліт обмежує можливості масштабування. Якщо під час пікового навантаження ресурси вичерпує лише модуль пошуку, доводиться дублювати весь монолітний додаток на нових

серверах, що призводить до вкрай неефективного споживання оперативної пам'яті та CPU [20].

Мікросервісна архітектура виникла як еволюційна відповідь на обмеження монолітних систем. Вона передбачає декомпозицію складного веб-додатку на набір невеликих, слабко зв'язаних та незалежних сервісів [17]. Кожен мікросервіс відповідає за чітко визначену бізнес-функцію і виконується як окремий процес. Взаємодія між ними відбувається за допомогою легковажних мережевих протоколів, найчастіше через RESTful API або gRPC [5].

Основними перевагами мікросервісної архітектури є [17]:

1. Незалежне розгортання: оновлення одного мікросервісу відбувається без зупинки інших частин системи.
2. Гранульоване масштабування: можливість виділяти додаткові обчислювальні ресурси виключно для тих сервісів, які зазнають найбільшого навантаження.
3. Технологічна гнучкість: розробники можуть вибирати різні мови програмування та типи баз даних для різних сервісів, залежно від того, який інструмент найкраще підходить для конкретної задачі.

Попри очевидні переваги, мікросервіси вносять суттєву інфраструктурну складність. Збільшується кількість точок відмови, виникають проблеми мережевих затримок між сервісами, а також критично ускладнюється процес моніторингу та агрегації логів [11].

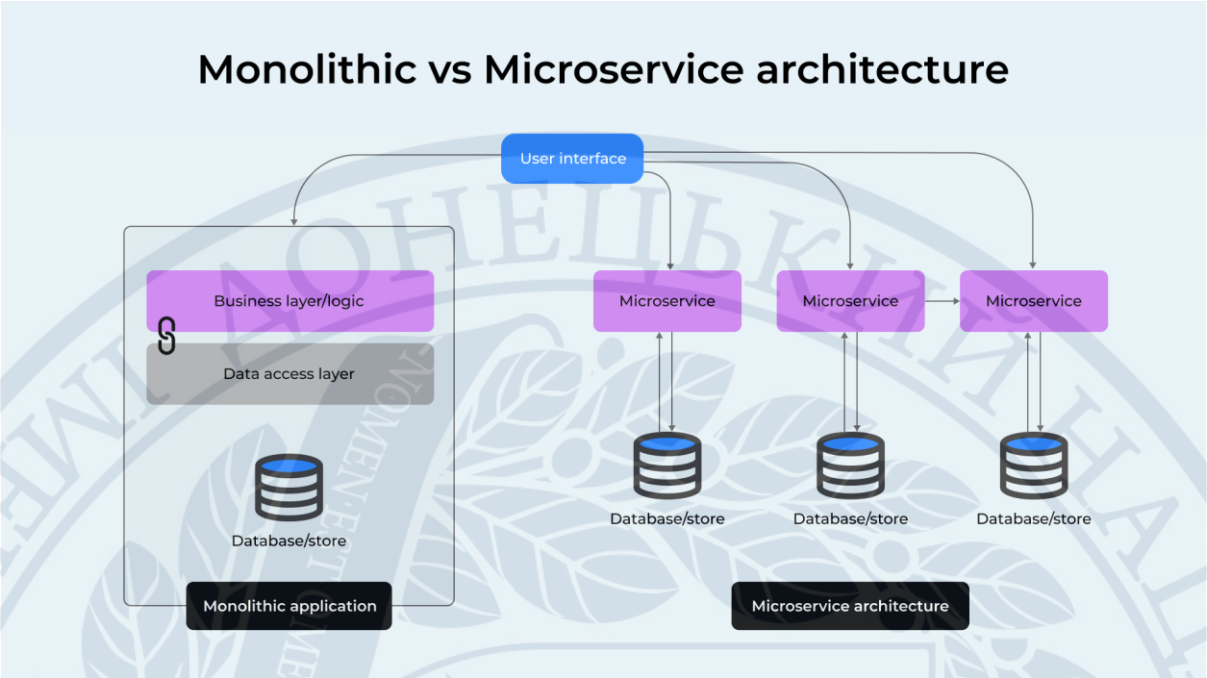

Рисунок 1.1 – Схема монолітної та мікросервісної архітектури

Для комплексного порівняння характеристик обох архітектурних підходів сформовано таблицю.

Таблиця 1.1 – Порівняльний аналіз архітектурних підходів

Критерій порівняння	**Монолітна архітектура**	**Мікросервісна архітектура**
Швидкість розробки прототипу	Висока (мінімальні витрати на налаштування)	Низька (вимагає складної інфраструктури)
Складність розгортання	Низька (єдиний модуль для деплою)	Висока (потребує оркестрації багатьох вузлів)
Можливості масштабування	Лише загальне горизонтальне масштабування	Точкове (гранульоване) масштабування
Ізоляція збоїв	Низька (помилка в пам'яті зупиняє весь додаток)	Висока (збій сервісу не руйнує всю систему)
Вимоги до DevOps-компетенцій	Базові (стандартне налаштування Linux)	Високі (CI/CD, контейнеризація, IaC)

Оскільки платформа проєктується виключно як веб-орієнтований продукт без нативної мобільної складової, застосування класичної мікросервісної архітектури на початковому етапі життєвого циклу є економічно та технічно

недоцільним. Надмірний поділ системи значно збільшить час виходу на ринок. Водночас, класичний монолітний підхід унеможливлює впровадження сучасних практик автоматизованого розгортання та гарантування надійності [18]. З огляду на це, оптимальним архітектурним рішенням для розробки та розгортання платформи обрано гібридний підхід, тобто модульний додаток з обов'язковим використанням технологій контейнеризації [2].

Контейнеризація виступає інструментом інкапсуляції програмного коду разом із його операційним середовищем у стандартизований блок – контейнер [3]. На відміну від апаратної віртуалізації, контейнери не потребують запуску гостьової операційної системи, оскільки вони спільно використовують ядро хостової ОС Linux. Це забезпечує мінімальне споживання ресурсів та миттєвий запуск веб-додатку [2].

Вибір контейнеризації для проєкту обґрунтовується такими чинниками :

1. Паритет середовищ: контейнер гарантує ідентичну поведінку платформи як на локальному комп'ютері розробника, так і на продуктовому Linux-сервері, що повністю виключає проблему «broken deployments» через відсутні бібліотеки.
2. Основа для автоматизації: контейнери є фундаментальною вимогою для побудови ефективних CI/CD конвеєрів, оскільки дозволяють автоматично збирати, тестувати та доставляти артефакти у вигляді незмінних образів.
3. Еволюційний шлях до мікросервісів: контейнеризований веб-додаток у майбутньому можна легко розділити на незалежні мікросервіси без фундаментальної зміни принципів адміністрування інфраструктури.

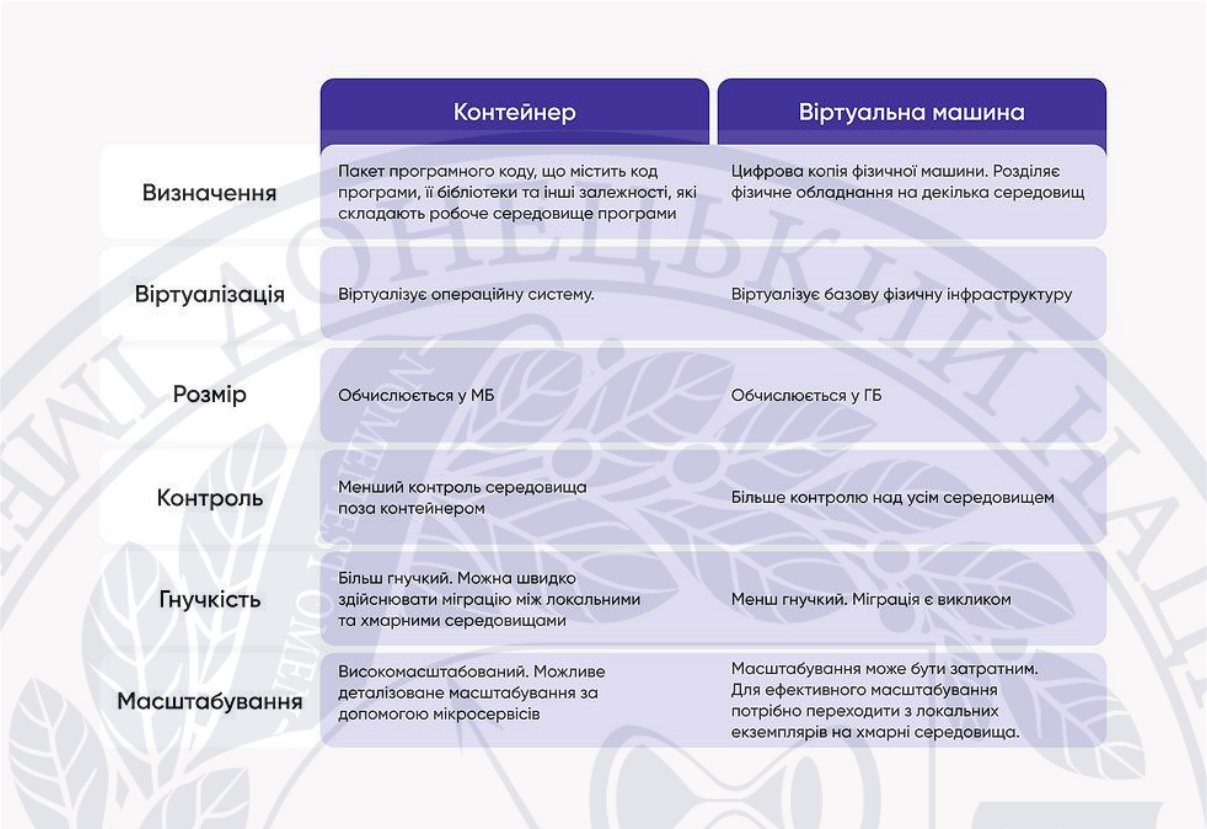

Рисунок 1.2 – Архітектурні відмінності віртуалізації та контейнеризації

Таким чином, використання контейнеризації дозволяє поєднати швидкість розробки монолітних систем із надійністю та готовністю до автоматизації [9].

1.2 Вибір операційної системи для серверної інфраструктури

Вибір операційної системи є фундаментальним етапом побудови серверної інфраструктури будь-якого вебпроєкту. Для платформи операційна система повинна забезпечувати не лише високу продуктивність та стабільність, а й повноцінну підтримку інструментів контейнеризації та автоматизації [27]. У сфері серверних рішень незаперечним стандартом є дистрибутиви на базі ядра Linux завдяки їхній відкритості, безпеці та ефективному управлінню ресурсами [36]. Для вибору оптимального рішення було проведено детальний розбір трьох найбільш поширених дистрибутивів: Debian, CentOS та Ubuntu Server.

Debian GNU/Linux вважається одним із найстабільніших та найдавніших дистрибутивів. Його ключовою філософією є використання лише ретельно

протестованого програмного забезпечення. Це робить Debian ідеальним вибором для критично важливих систем, де стабільність є пріоритетом №1. Однак консервативний підхід до оновлень має зворотний бік: пакетна база Debian часто містить застарілі версії бібліотек та системних утиліт. Для сучасного проєкту, який потребує актуальних версій Docker [27], Docker Compose та мережевих драйверів, робота з Debian може вимагати додаткових зусиль щодо підключення сторонніх репозиторіїв та ручного налаштування залежностей.

CentOS тривалий час був стандартом у корпоративному секторі, оскільки він базувався на вихідному коді Red Hat Enterprise Linux. Проте останні зміни в політиці розробки та перехід до моделі CentOS Stream суттєво змінили позиції цього дистрибутиву. CentOS став «upstream-платформою» для RHEL, що означає отримання оновлень раніше за стабільний комерційний реліз. Це підвищує ризик виникнення неочікуваних помилок у роботі серверного ПЗ, що є критичним для невеликих команд розробки, які не мають ресурсів для постійного виправлення специфічних системних конфліктів [11].

Ubuntu Server, що базується на архітектурі Debian, на сьогодні є найбільш збалансованим рішенням для розгортання вебдодатків [39]. Дистрибутив пропонує оптимальне поєднання стабільності та доступу до сучасного програмного забезпечення [27].

Для систематизації аналізу було сформовано порівняльну таблицю за ключовими критеріями адміністрування. На основі проведеного порівняльного аналізу для розгортання інфраструктури платформи було обрано Ubuntu Server. Цей вибір обґрунтовується наступними технічними та експлуатаційними перевагами:

1. Нативна підтримка хмарних технологій. Ubuntu Server є найбільш вживаною операційною системою в хмарних середовищах. Більшість образів для віртуальних машин та Docker-контейнерів оптимізовані саме під цей дистрибутив.
2. Широка екосистема та документація. Величезна база знань дозволяє швидко вирішувати питання адміністрування та безпеки.

3. Стабільність версій LTS. Випуски з позначкою Long Term Support гарантують оновлення безпеки протягом 5 років, що забезпечує довготривалу роботу платформи без необхідності частої та ризикованої зміни версії ядра ОС.
4. Ефективність пакетного менеджера. Використання APT у поєднанні з репозиторіями PPA дозволяє отримувати найновіші версії Nginx, PostgreSQL та інших компонентів без втрати стабільності системи.
5. Оптимізація під Docker. Розробники Docker офіційно рекомендують Ubuntu як одну з пріоритетних платформ для запуску Docker Engine, що гарантує відсутність проблем із мережевими драйверами та файловими системами при роботі контейнерів.

Таблиця 1.2 – Порівняльна характеристика серверних дистрибутивів Linux

Критерій порівняння	**Debian**	**CentOS (Stream)**	**Ubuntu Server**
Цикл оновлень	Консервативний (старі версії)	Проміжний (rolling release)	Регулярний (LTS версії)
Пакетний менеджер	APT (висока швидкість)	DNF / YUM	APT(максимальна гнучкість)
Документація та спільнота	Велика, але специфічна	Середня (після реформи)	Найбільша у світі
Підтримка Docker/Cloud	Повна (потребує налаштування)	Повна (можливі конфлікти)	Нативна (стандарт для Cloud)
Рівень складності	Середній	Високий	Нижче середнього

1.3 Технології контейнеризації: архітектура Docker Engine та порівняння з віртуальними машинами

Для забезпечення гнучкості та швидкодії розгортання платформи як базовий інфраструктурний стандарт обрано технологію контейнеризації. На відміну від апаратної віртуалізації, контейнеризація є методом віртуалізації на рівні операційної системи. Вона дозволяє запускати ізольовані процеси користувацького простору, використовуючи єдине спільне ядро хостової операційної системи. Беззаперечним галузевим стандартом у цій сфері є платформа Docker. Її фундаментальна відмінність полягає в інкапсуляції програмного додатка та всіх його залежностей у стандартизований незмінний артефакт - Docker-образ, технологічна основа якого дозволяє в подальшому легко переходити до масштабнішої оркестрації [2, 4].

В основі платформи лежить клієнт-серверний додаток Docker Engine, який складається з трьох ключових компонентів [3]:

1. Docker Daemon: фоновий сервер, який безпосередньо керує об'єктами Docker та взаємодіє з ядром Linux.
2. REST API: програмний інтерфейс, через який сторонні програми або скрипти відправляють інструкції демону.
3. Docker CLI: інтерфейс командного рядка для взаємодії користувача з демоном.

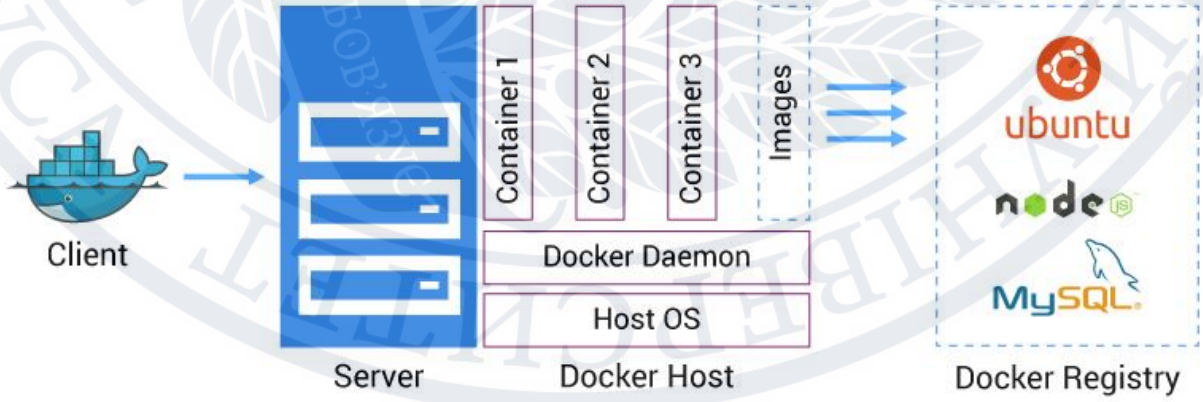

Рисунок 1.3 – Клієнт-серверна архітектура Docker Engine

Технічно Docker не є самостійною технологією віртуалізації. Він виступає високорівневою обгорткою над вбудованими механізмами ядра Linux, що забезпечують ізоляцію [2]:

- Namespaces: гарантують логічну ізоляцію процесів. Кожен контейнер отримує власний ізольований простір для ідентифікаторів процесів, мережевого стеку, точок монтування та міжпроцесної взаємодії.
- Control Groups: забезпечують апаратну ізоляцію. Вони лімітують та контролюють обсяг фізичних ресурсів, які може спожити конкретний контейнер, запобігаючи монополізації ресурсів сервера.

Класична віртуалізація на базі Hypervisor передбачає запуск повноцінної гостьової операційної системи Guest OS для кожної віртуальної машини. Це створює колосальні накладні витрати ресурсів, оскільки значна частина оперативної пам'яті та процесорного часу витрачається на обслуговування самої гостьової ОС, а не корисного навантаження. Запуск віртуальної машини є тривалим процесом, який займає від кількох десятків секунд до хвилин, оскільки ядро гостьової системи має пройти повний цикл завантаження. Розмір артефакту віртуальної машини зазвичай вимірюється гігабайтами [3].

Натомість Docker-контейнери позбавлені цих недоліків завдяки спільній експлуатації ядра хостової ОС. Відсутність гостьової операційної системи означає, що контейнер використовує лише ті ресурси, які необхідні для виконання конкретного додатка. Швидкість запуску контейнера становить лічені мілісекунди, адже фактично стартує лише ізольований процес користувацького рівня, а не повноцінна операційна система. Розмір контейнера є мінімальним і найчастіше варіюється від кількох до сотень мегабайтів.

З погляду портативності віртуальні машини жорстко прив'язані до конкретного гіпервізора, що ускладнює їх міграцію між різними хмарними провайдерами. Docker розв'язує цю проблему завдяки принципу «Build once, run anywhere»: контейнер гарантовано працюватиме ідентично на локальному комп'ютері розробника, на тестовому сервері та в продуктовому кластері.

Використання повноцінних віртуальних машин для розгортання окремих мікросервісів чи модулів системи є архітектурно надлишковим і економічно неефективним. Контейнеризація за допомогою Docker забезпечує абсолютний паритет середовищ розробки та продакшену, високу утилізацію серверних ресурсів, а також формує надійний фундамент для автоматизації процесів збірки та деплою платформи.

1.4 Огляд методології CI/CD та аналіз інструментів автоматизації

Методологія CI/CD є невід'ємною частиною сучасного DevOps-інжинірингу. Вона спрямована на автоматизацію процесів збірки, тестування та розгортання програмного забезпечення, що дозволяє мінімізувати кількість помилок, пов'язаних із людським фактором, та пришвидшити цикл доставки нових функцій користувачам [1].

Безперервна інтеграція CI передбачає автоматичне злиття робочих копій коду розробників у загальну гілку кілька разів на день з обов'язковим запуском автоматизованих тестів та перевірок якості коду.

Безперервне розгортання CD розширює цей процес, автоматично доставляючи перевірений код на продуктовий сервер [9].

Для реалізації цих процесів було проаналізовано три провідні інструменти автоматизації:

1. Jenkins. Найстаріший та найбільш гнучкий інструмент з відкритим кодом. Його головна перевага величезна екосистема плагінів, що дозволяє інтегруватися з будь-якими технологіями. Проте Jenkins вимагає наявності окремого сервера для власного функціонування, складного налаштування та постійного адміністрування оновлень безпеки. Для проєкту на етапі MVP такий підхід є надмірно ресурсомістким.
2. GitLab CI/CD. Потужне інтегроване рішення, де система контролю версій та CI/CD конвеєр знаходяться в одному інтерфейсі. Налаштування відбувається через YAML-файли. GitLab пропонує чудову

функціональність, але вимагає значних ресурсів для self-hosted версії або переходу на платні плани у хмарній версії для великих проєктів.

3. GitHub Actions. Сучасна платформа автоматизації, інтегрована безпосередньо в GitHub. Основною концепцією є використання Actions - готових блоків коду, що доступні у Marketplace. Налаштування здійснюється через декларативні YAML-файли, які зберігаються у репозиторії проєкту.

Для автоматизації життєвого циклу платформи було обрано GitHub Actions. Даний вибір зумовлений низкою вагомих чинників. Насамперед це повна інтеграція з Git-хостингом: оскільки сирцевий код проєкту зберігається на GitHub, використання вбудованого інструменту CI/CD дозволяє уникнути налаштування складних веб-хуків та сторонніх інтеграцій. Додатковою перевагою є розвинена екосистема Marketplace, де наявність готових сертифікованих Actions для роботи з Docker та Google Cloud Platform значно пришвидшує написання конвеєрів [23]. Крім того, інструмент має безсерверну архітектуру, оскільки виконується на потужностях платформи, що позбавляє адміністратора необхідності самостійно розгортати та підтримувати сервери для автоматизації, на відміну від рішень на кшталт Jenkins. Таким чином, GitHub Actions дозволяє реалізувати надійний конвеєр від перевірки коду до збірки Docker-образу та його деплою на Linux-сервер у Google Cloud єдиною push-командою [1].

1.5 Огляд та порівняння існуючих рішень у сфері C2C книжкової комерції

Для обґрунтування унікальності та необхідності розробки вебплатформи було проведено критичний аналіз існуючих програмних продуктів та сервісів на ринку України, які користувачі застосовують для купівлі, продажу або обміну вживаною літературою. Сучасний сегмент електронної комерції у сфері книгорозповсюдження можна класифікувати за трьома основними категоріями:

універсальні C2C маркетплейси, великі B2C інтернет-магазини та спеціалізовані міжнародні соціальні платформи [20].

Першою категорією є універсальні майданчики оголошень, такі як OLX та Prom.ua. Вони мають величезну аудиторію та забезпечують базовий механізм проведення угод між фізичними особами. Проте ці платформи мають суттєві архітектурні та функціональні недоліки при роботі з книжковою продукцією, відсутність структурованої бази даних книг, неможливість пошуку за специфічними метаданими, а також повна відсутність спеціалізованих механізмів для прямого книгообміну. Користувачі змушені вручну описувати стан кожної книги, що створює хаос у пошуковій видачі та знижує конверсію.

Для наочного порівняння існуючих аналогів із проєктувальною системою було розроблено порівняльну таблицю.

Таблиця 1.3 - Порівняльний аналіз платформ для обігу книжкової продукції

Критерій порівняння	**OLX / Prom.ua**	**Yakaboo**	**Bookcrossing**
Модель взаємодії	C2C (загальна)	B2C (комерційна)	C2C (некомерційна)
Пошук за ISBN / метаданими	Відсутній	Наявний	Частковий
Система прямого обміну	Відсутня	Відсутня	Базова (без гарантій)
Елементи гейміфікації	Відсутні	Відсутні	Базові
Безпека угод (Escrow)	Наявна	Наявна	Відсутня

Друга категорія великі B2C ритейлери, наприклад, Yakaboo, Книгарня «Є». Вони пропонують бездоганний інтерфейс та автоматизований пошук, але функціонують виключно в межах продажу нових книг від видавництв за фіксованими роздрібними цінами. Вони не надають звичайним користувачам можливості продати власну прочитану книгу або обміняти її на іншу, що обмежує розвиток циркулярної економіки та доступність літератури для студентської молоді.

Третя категорія міжнародні платформи. Вони мають потужну соціальну складову та гейміфікацію, проте Bookcrossing в Україні має слабку логістичну та комерційну інтеграцію, а Goodreads повністю позбавлений функціоналу маркетплейсу.

Таким чином, проведений аналіз підтверджує наявність незайнятої ніші на вітчизняному ринку, існує гостра потреба у спеціалізованій C2C вебплатформі, яка поєднує гнучкість маркетплейсу, зручність пошуку за книжковими метаданими та автоматизовані механізми безпечного обміну літературою з елементами гейміфікації для підвищення залученості користувачів [20].

РОЗДІЛ 2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1 Опис функціональних вимог та сценаріїв використання системи

При проєктуванні складних інформаційних систем процес починається з повного аналізу та формалізації функціональних вимог [8]. Для платформи, яка функціонує за моделлю C2C, ключовим завданням є створення прозорого та зручного середовища для взаємодії між незалежними користувачами. Іншими словами функціональні вимоги визначають, які саме завдання повинна розв'язувати система для задоволення потреб цільової аудиторії, а сценарії використання дозволяють візуалізувати ці процеси з позиції конкретних акторів системи [8]. У межах даного проєкту виділяється три основні групи користувачів: покупець, продавець та адміністратор.

Основу функціоналу платформи складають сценарії використання для покупця, оскільки саме від їхньої зручності залежить рівень утримання аудиторії. Покупець починає взаємодію із системою через модуль пошуку та фільтрації, тому система повинна забезпечувати можливість пошуку книг за назвою, автором, жанром, станом видання та географічним розташуванням продавця, що є критичним для мінімізації витрат на логістику [1]. Після вибору певного примірника реалізується ініціація угоди. На відміну від класичних інтернет-магазинів, платформа передбачає два варіанти розвитку подій: пряму купівлю або пропозицію обміну. У випадку обміну покупець пропонує власні книги, а система автоматично перевіряє сумісність інтересів сторін. Окрему увагу в сценарії покупця приділено гейміфікації: за кожну успішну угоду, прочитану книгу або залишений змістовний відгук користувач отримує бали досвіду, що підвищують його рівень у системі та відкривають доступ до ексклюзивних функцій, таких як безкоштовна доставка або пріоритетне відображення оголошень [8].

Випадки використання для продавця спрямовані на автоматизацію управління власним книжковим інвентарем. Процес починається зі створення оголошення, де продавець завантажує фотографії, описує стан книги та встановлює ціну або статус лише для обміну, а функціональна вимога щодо

інтелектуального управління запасами передбачає можливість тимчасового приховання оголошень без їхнього видалення [8]. Продавець потім має доступ до панелі статистики і може відстежувати кількість переглядів своїх книг, а також рівень інтересу спільноти. Критичним аспектом є контекст комунікації: система дозволяє безпечний чат для обговорення деталей транзакції. Після транзакції система автоматично коригує рейтинг продавця, щоб вказати на репутацію та довіру з іншими учасниками платформи. Це допомагає користувачам правильно повідомляти про випадок використання товарів, які вони купують [8].

Випадки використання для адміністратора можуть включати моніторинг, модерацію та процеси управління інфраструктурою, які тісно співпадають з компонентом DevOps роботи [3]. Адміністратор має найбільший доступ і забезпечує дотримання правил платформи. Функціональний сценарій модерації полягає у перевірці, чи нові оголошення відповідають етичним нормам, і чи описи є автентичними. Адміністратор також виконує функції технічного нагляду: через панель управління власник панелі може бачити статус сервера Linux, навантаження контейнера Docker, продуктивність бази даних та статус навантаження в реальному часі [10]. Якщо відбувається підозріла активність або спроба несанкціонованого доступу, адміністратор розгортає дію блокування облікового запису або тимчасово блокує функціональність на певних IP-адресах. Адміністратор відповідає за підтримку інфраструктури резервного копіювання та відновлення даних, а також за забезпечення безпеки користувачів для збереження цілісності інформації, пов'язаної з користувачами разом з їхньою цифровою бібліотекою [14].

Для структурованого представлення виявлених сценаріїв використання сформовано зведену таблицю, що систематизує дії акторів та очікувані реакції системи.

Таблиця 2.1 – Зведені сценарії використання системи за акторами

Актор	**Сценарій**	**Передумова**	**Результат**
Покупець	Пошук книги	Авторизований профіль	Список відфільтрованих оголошень
Покупець	Ініціація купівлі	Обрано оголошення	Угода зі статусом pending
Покупець	Пропозиція обміну	Наявні власні оголошення	Запит на обмін обом сторонам
Продавець	Публікація оголошення	Верифікований акаунт	Оголошення у каталозі
Продавець	Приховання оголошення	Активне оголошення	is_active = false, дані збережені
Продавець	Підтвердження угоди	Статус pending	Статус accepted, сторони повідомлені
Адміністратор	Модерація оголошення	Скарга або автофлаг	Оголошення приховане або видалене
Адміністратор	Перегляд стану сервера	Доступ до панелі	Метрики CPU, RAM, статус контейнерів
Адміністратор	Блокування користувача	Зафіксоване порушення	Сесії завершені, вхід заблоковано

Наведена таблиця підкреслює асиметрію між роллю покупця, сценарії якого переважно орієнтовані на пошук та ініціацію взаємодії, і роллю продавця, дії якого зосереджені на управлінні власним контентом та реакції на вхідні запити. Адміністративні сценарії мають переважно реактивний характер, що

відображає специфіку підтримки C2C-платформ більшість процесів відбувається між користувачами без участі платформи, адміністратор втручається лише у виняткових ситуаціях [1].

Варто зазначити, що межа між роллю покупця та продавця є умовною один і той самий акаунт може одночасно мати активні оголошення і бути ініціатором угоди по чужому оголошенню. Така гнучкість є принциповою характеристикою C2C-моделі і відображена у структурі таблиці бази даних через поле role зі значенням both за замовчуванням [12].

Відповідно розроблений набір функціональних вимог охоплює повний життєвий цикл взаємодії користувачів на платформі, а поєднання стандартних комерційних функцій із механіками обміну та елементами гейміфікації створює унікальний користувацький досвід. Водночас, чіткий поділ ролей та сценаріїв дозволяє ефективно проєктувати архітектуру бази даних та серверної частини, забезпечуючи стабільну роботу системи навіть при значному зростанні кількості активних сесій [8].

2.2 Проєктування бази даних: ER-діаграма та опис структури таблиць

Фундаментом будь-якої вебплатформи електронної комерції є надійна система зберігання даних. Для платформи обрано реляційну модель бази даних. Вона найкраще підходить для систем із чітко структурованими зв'язками між сутностями (користувачами, товарами та транзакціями) [12].

На етапі проєктування інфраструктури розглядалися дві найпопулярніші відкриті реляційні СУБД: MySQL та PostgreSQL. MySQL традиційно відзначається високою швидкістю виконання простих запитів на читання. Ця система часто використовується у базових вебпроєктах. Проте архітектура платформи передбачає не лише прямі продажі, а й C2C-обмін товарами. Операція взаємного обміну книгами є складною транзакцією. Вона вимагає одночасного блокування та оновлення статусів кількох записів у різних таблицях [25]. З огляду

на потребу в суворій цілісності даних, для розгортання на сервері обрано PostgreSQL. Ця СУБД має низку критичних переваг для даного проєкту:

1. Повна відповідність стандартам ACID [12].
2. Потужний механізм багатоваріантного керування конкурентним доступом. Він запобігає блокуванню таблиць під час паралельних транзакцій від різних користувачів [25].
3. Нативна підтримка типу даних JSONB. Це дозволяє у майбутньому гнучко зберігати нетипові характеристики книг без зміни загальної схеми таблиці [25].
4. Розширений функціонал обмежень на рівні бази даних, що зменшує навантаження на бекенд-додаток при валідації вхідних даних [12].

Спроєктована схема бази даних базується на трьох основних сутностях: користувачі, книги/оголошення та угоди. Додатково введено допоміжну таблицю відгуків для реалізації системи рейтингу. Взаємозв'язки між сутностями візуалізовано за допомогою ER-діаграми.

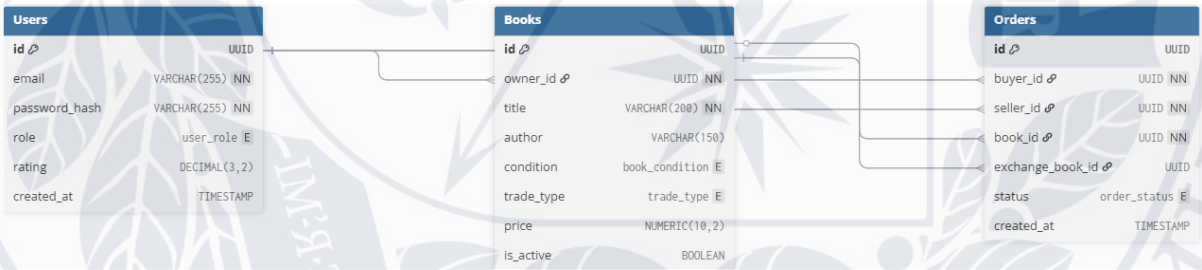

Рисунок 2.1 - Логічна ER-діаграма бази даних платформи

Нижче наведено детальний опис структури кожної сутності з обґрунтуванням обраних типів даних.

Сутність «Users» зберігає облікові дані та метрики профілю.

- id (тип UUID, Primary Key). Використання стандарту Universally Unique Identifier замість класичного автоінкременту зумовлено вимогами безпеки. Це унеможливлює атаку типу Insecure Direct Object Reference, коли зловмисник може вгадати ідентифікатор іншого користувача.

- email (тип VARCHAR(255), UNIQUE, NOT NULL). Використовується як логін. На поле накладено індекс типу B-Tree для пришвидшення авторизації.
- password_hash (тип VARCHAR(255), NOT NULL). Зберігає хеш-суму пароля (наприклад, алгоритм bcrypt). Зберігання паролів у відкритому вигляді категорично заборонено.
- role (тип ENUM('buyer', 'seller', 'admin'), DEFAULT 'buyer'). Визначає рівень доступу до функціоналу системи.
- rating (тип DECIMAL(3,2), DEFAULT 0.00). Поточний рейтинг довіри користувача.
- created_at (тип TIMESTAMP, DEFAULT CURRENT_TIMESTAMP). Фіксує час реєстрації.

Сутність «Books» містить інформацію про інвентар платформи. Зв'язок із сутністю Users реалізується як відношення «Один-до-багатьох» (один користувач може створити багато оголошень).

- id (тип UUID, Primary Key).
- owner_id (тип UUID, Foreign Key). Посилання на власника. Налаштовано каскадне видалення (ON DELETE CASCADE): при видаленні профілю користувача всі його оголошення також видаляються.
- title (тип VARCHAR(200), NOT NULL). Назва видання.
- author (тип VARCHAR(150)). Автор твору.
- condition (тип ENUM('new', 'like_new', 'good', 'acceptable')). Стандартизований опис фізичного стану сторінок та обкладинки.
- trade_type (тип ENUM('sell', 'exchange', 'both')). Визначає дозволені сценарії угоди для конкретного примірника.
- price (тип NUMERIC(10,2)). Вартість книги у національній валюті. Використання типу FLOAT для фінансових операцій є технічною помилкою через проблему втрати точності з плаваючою комою. Тип NUMERIC гарантує точність до сотих часток. Поле може бути порожнім (NULL), якщо встановлено статус виключно для обміну.

- is_active (тип BOOLEAN, DEFAULT TRUE). Реалізує прецедент «тимчасового приховання» оголошення без фізичного видалення рядка з диска.

Сутність «Orders» є центральним вузлом бізнес-логіки. Вона фіксує факт передачі прав на книгу від одного користувача до іншого.

- id (тип UUID, Primary Key).
- buyer_id (тип UUID, Foreign Key). Ініціатор угоди.
- seller_id (тип UUID, Foreign Key). Власник цільового примірника.
- book_id (тип UUID, Foreign Key). Книга, яку бажають придбати або отримати.
- exchange_book_id (тип UUID, Foreign Key, NULLABLE). Ідентифікатор книги, яку покупець віддає навзаєм. Поле залишається порожнім у разі класичної грошової покупки.
- status (тип ENUM('pending', 'accepted', 'rejected', 'completed')). Життєвий цикл угоди. Зміна статусу на completed ініціює тригер бази даних, який автоматично переводить відповідні записи в таблиці Books у стан неактивних (is_active = FALSE).
- created_at (тип TIMESTAMP). Час створення запиту.

Така архітектура бази даних характеризується високим рівнем нормалізації. Відсутність дублювання даних дозволяє мінімізувати розмір бази, що є важливим фактором при розгортанні платформи в обмежених ресурсах хмарних Docker-контейнерів. Крім того, наявність зовнішніх ключів гарантує консистентність, система не дозволить створити замовлення на книгу, яка була видалена із сутності Books [12].

2.3 Схема розгортання системи та роль Nginx як зворотного проксі

Схема розгортання є ключовим архітектурним документом, що описує фізичне та логічне розміщення компонентів платформи на виробничому сервері [1]. На відміну від абстрактних діаграм класів або компонентів, схема

розгортання безпосередньо відображає реальну конфігурацію інфраструктури та шляхи передачі трафіку між сервісами.

У рамках реалізованої архітектури всі компоненти платформи функціонують як окремі Docker-контейнери на єдиному виробничому сервері під управлінням Ubuntu Server 22.04 LTS у середовищі Google Cloud Platform. Точкою входу для всього зовнішнього трафіку слугує контейнер із веб-сервером Nginx, який виконує роль зворотного проксі [13].

Зворотний проксі-сервер є компонентом інфраструктури, що знаходиться між зовнішніми клієнтами та внутрішніми сервісами застосунку. Він приймає вхідні запити від браузерів користувачів та переадресовує їх до відповідного внутрішнього сервісу, а отриману відповідь повертає клієнту. Ця схема передбачає кілька критичних переваг для продуктового середовища [24].

По-перше, реалізується централізоване завершення SSL-з'єднань. Nginx бере на себе відповідальність за шифрування та розшифрування трафіку за протоколом HTTPS, тоді як внутрішні сервіси можуть спілкуватися між собою по незашифрованому HTTP у межах ізольованої Docker-мережі. Це значно спрощує конфігурацію бекенд-додатку та знижує навантаження на процесор при обробці криптографічних операцій [26].

По-друге, контейнер Nginx є єдиним вузлом, який має відкриті зовнішні порти на хостовій операційній системі, порти 80 HTTP та 443 HTTPS. Усі решта контейнери, включно з бекенд-сервісом та базою даних PostgreSQL, не доступні із зовнішньої мережі й ізольовані всередині приватної Docker-мережі. Це суттєво зменшує поверхню атаки [14].

Взаємодія між компонентами системи відбувається за наступним принципом. Клієнт відправляє HTTPS-запит на IP-адресу сервера. Операційна система Ubuntu передає запит до контейнера Nginx через правила перенаправлення портів. Nginx перевіряє запит відповідно до налаштованих директив і, залежно від URL-шляху, проксує його до відповідного контейнера: статичні файли фронтенду або динамічні API-запити до бекенду. Бекенд-сервіс,

у свою чергу, за потреби виконує запити до контейнера бази даних PostgreSQL через внутрішню Docker-мережу [15].

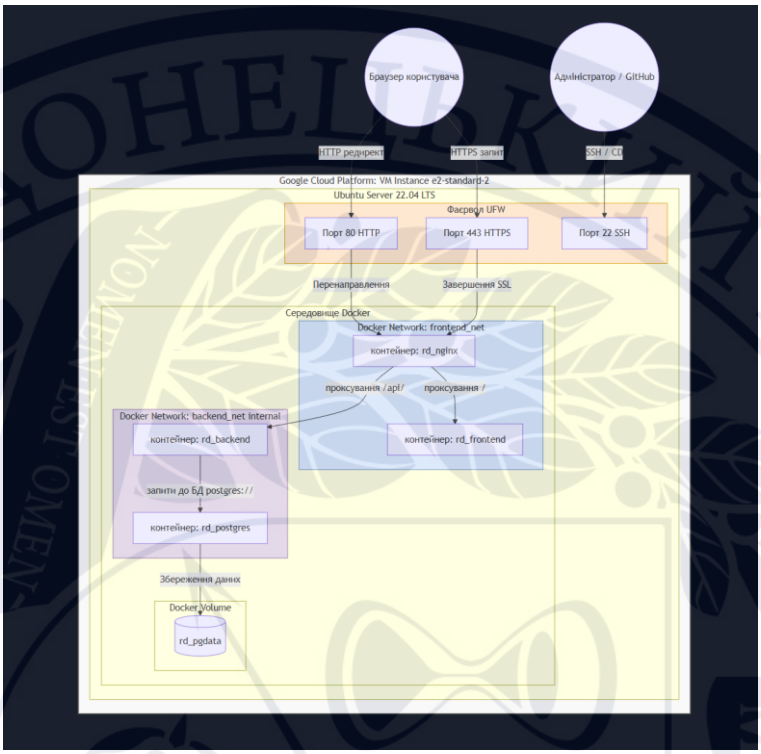

Рисунок 2.2 – Схема розгортання платформи з маршрутами передачі трафіку

Балансування навантаження між потенційними репліками бекенд-сервісу також є зоною відповідальності Nginx. Для MVP-версії платформи достатньо однієї репліки, проте конфігурація дозволяє горизонтально масштабувати бекенд без внесення змін до будь-яких інших компонентів, достатньо оновити блок upstream у конфігураційному файлі Nginx [24].

Окремої уваги заслуговує обробка статичних ресурсів. Зібрані файли React-застосунку розміщено у контейнері фронтенду та роздаються безпосередньо через Nginx без звернення до бекенд-сервісу. Це дозволяє суттєво знизити навантаження на прикладний сервер та прискорити час відповіді для першого завантаження сторінки [13].

2.4 Мережева топологія проєкту: внутрішні мережі Docker та зовнішні порти

Мережева ізоляція є фундаментальним принципом безпечної контейнерної інфраструктури. Docker реалізує власний рівень мережевої абстракції, що

дозволяє створювати ізольовані мережеві простори Docker-мережі. За замовчуванням контейнер, не підключений до жодної мережі, фізично недосяжний ані ззовні, ані з інших контейнерів [15].

Для платформи реалізовано дворівневу мережеву архітектуру. Перший рівень - це зовнішня мережа frontend_net, до якої підключені контейнер Nginx та контейнер фронтенду. Через цю мережу здійснюється роздача статичних файлів та проксування API-запитів. Другий рівень - внутрішня мережа backend_net, яка об'єднує контейнер бекенд-сервісу та контейнер PostgreSQL. Ця мережа повністю ізольована від зовнішнього світу, що унеможливлює будь-який прямий доступ до бази даних ззовні [16].

Відображення портів між контейнерами та хостовою операційною системою визначається наступним чином. Контейнер Nginx прослуховує внутрішні порти 80 та 443, які за допомогою директиви ports у файлі docker-compose.yml відображаються на аналогічні порти хостової ОС. Контейнер бекенд-сервісу прослуховує внутрішній порт 3000, проте зовнішнього відображення для нього не налаштовано, він доступний виключно через ім'я сервісу backend у межах мережі backend_net. Контейнер PostgreSQL прослуховує стандартний порт 5432, але аналогічно не має зовнішнього відображення, що робить його повністю ізольованим від інтернету [16].

Таблиця 2.2 – Матриця мережевої ізоляції компонентів платформи

Контейнер	**Внутрішній порт**	**Зовнішній порт**	**Мережі**	**Доступ ззовні**
nginx	80, 443	80, 443	frontend_net	Так (Інтернет)
frontend	3001	Немає	frontend_net	Ні
backend	3000	Немає	frontend_net, backend_net	Ні
postgres	5432	Немає	backend_net	Ні

DNS-розв'язання між контейнерами здійснюється через вбудований DNS-сервер Docker. Кожному контейнеру в межах однієї мережі автоматично присвоюється DNS-ім'я, що відповідає імені сервісу у файлі docker-compose.yml. Завдяки цьому рядок підключення до бази даних у конфігурації бекенд-сервісу, де postgres є DNS-ім'ям контейнера всередині мережі backend_net, а не реальною IP-адресою. Це гарантує стабільність з'єднання навіть при перезапуску контейнерів, коли їм можуть призначатись нові внутрішні IP-адреси [16].

Окремим аспектом є механізм збереження даних PostgreSQL. Дані бази даних зберігаються у Docker Volume іменованому томі rd_pgdata, що монтується до директорії /var/lib/postgresql/data всередині контейнера. Використання томів замість bind-монтування директорій хостової ОС є рекомендованою практикою, оскільки томи керуються безпосередньо Docker-демоном та мають вищу продуктивність завдяки оптимізованому вводу-виводу. Ця конфігурація гарантує збереження усіх даних між перезапусками та оновленнями контейнера [15].

2.5 Обґрунтування вибору технологічного стеку прикладного програмного забезпечення

Ефективність функціонування мультиконтейнерної інфраструктури, опис якої закладено в архітектурні рішення платформи, безпосередньо залежить від коректного вибору інструментів прикладного програмування. Для реалізації клієнтської та серверної частин вебплатформи було обрано стек технологій на базі програмних рішень Node.js та бібліотеки React, для побудови фронтенду [35]. Цей вибір зумовлений високими вимогами до продуктивності, швидкості розгортання та сумісності з Docker-контейнерами [15].

Серверне ядро системи реалізується на базі Node.js із використанням фреймворку Express. Вибір Node.js обґрунтований його асинхронною подієво-орієнтованою моделлю введення-виведення. Оскільки вебплатформа електронної комерції передбачає обробку великої кількості одночасних підключень (перегляд каталогів, створення замовлень, чати між користувачами), Node.js дозволяє ефективно масштабувати систему без значного навантаження на

оперативну пам'ять сервера, на відміну від класичних потокових архітектур [35]. Крім того, в контексті контейнеризації образи Node.js на базі Alpine Linux мають мінімальну вагу та надзвичайно низький час холодного старту, що є критично важливим для динамічного масштабування у хмарних серверах [15, 7].

Клієнтський інтерфейс проєктується як Single Page Application за допомогою бібліотеки React. Використання React дозволяє створити динамічний, компонентно-орієнтований інтерфейс користувача з високою швидкістю відклику завдяки механізму Virtual DOM [1]. Для інфраструктури розгортання архітектура SPA має ключову перевагу, під час виконання CI/CD конвеєра весь фронтенд-код компілюється в набір статичних оптимізованих файлів HTML, CSS, JavaScript. Це дозволяє повністю ізолювати середовище розробки від виробничого сервера, фінальний образ не потребує запуску важкого Node.js процесу для клієнта, а роздається як чиста статика через легкий та продуктивний вебсервер Nginx, конфігурація якого наведена в інфраструктурній частині роботи [13].

Поєднання Node.js та React забезпечує використання єдиної мови програмування на всіх рівнях розробки, що прискорює створення продукту, спрощує написання конфігураційних Dockerfile та оптимізує передачу об'єктів даних у форматі JSON між клієнтською та серверною частинами платформи [35].

РОЗДІЛ 3. ТЕХНІЧНА РЕАЛІЗАЦІЯ ТА DEVOPS-ІНЖЕНЕРІЯ

3.1 Адміністрування Linux-сервера: налаштування користувачів, SSH та фаєрволу

Розгортання платформи розпочинається з ініціалізації виробничого сервера. Для проєкту обрано хмарне середовище Google Cloud Platform, де засобами консолі Compute Engine було створено віртуальну машину з наступними характеристиками: тип машини e2-standard-2 (2 vCPU, 8 ГБ RAM), операційна система Ubuntu Server 22.04 LTS, завантажувальний диск 30 ГБ SSD. Вибір цієї конфігурації обґрунтований достатністю ресурсів для одночасного запуску чотирьох Docker-контейнерів при помірному навантаженні та наявністю п'ятирічної підтримки безпеки для Ubuntu 22.04 LTS [27, 30].

Першим кроком після отримання доступу до сервера є оновлення пакетної бази та встановлення базових системних інструментів. Усі команди виконуються з підвищеними привілеями через sudo, оскільки пряма робота від імені суперкористувача root є небезпечною практикою [27].

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y curl wget git unzip ufw htop
```

Наступним критичним кроком є налаштування ролей користувачів. Для автоматизованих процесів розгортання через CI/CD конвеєр створено окремого системного користувача deploy з обмеженими привілеями. Цей підхід реалізує принцип найменших привілеїв: автоматизований процес матиме доступ лише до тих директорій та команд, які необхідні для виконання деплою, але не матиме повного контролю над сервером [27].

```
sudo adduser --disabled-password --gecos "" deploy
sudo usermod -aG docker deploy
sudo mkdir -p /var/www/readvia
sudo chown deploy:deploy /var/www/readvia
```

Надання користувачу deploy членства у групі docker є обов'язковою умовою для виконання команд docker та docker compose без використання sudo. Без цього налаштування будь-яка спроба CI/CD-скрипта перезапустити

контейнери завершуватиметься помилкою доступу до Unix-сокету Docker-демона [15].

Налаштування SSH-аутентифікації за ключами є обов'язковим кроком для забезпечення безпеки сервера. Аутентифікація за паролем вразлива до атак методом підбору, тому її необхідно відключити та залишити лише аутентифікацію за криптографічним ключем [36]. Для автоматизованого користувача deploy генерується пара ключів на локальній машині, публічний ключ розміщується на сервері:

```
sudo mkdir -p /home/deploy/.ssh
sudo chmod 700 /home/deploy/.ssh
echo "PUBLIC_KEY_VALUE" | sudo tee /home/deploy/.ssh/authorized_keys
sudo chmod 600 /home/deploy/.ssh/authorized_keys
sudo chown -R deploy:deploy /home/deploy/.ssh
```

Конфігурація SSH-демона задається у файлі /etc/ssh/sshd_config. Нижче наведено ключові параметри безпеки, що вносяться до цього файлу:

```
Port 22
PermitRootLogin no
PasswordAuthentication no
PubkeyAuthentication yes
AuthorizedKeysFile .ssh/authorized_keys
X11Forwarding no
MaxAuthTries 3
```

Параметр PermitRootLogin no повністю забороняє вхід на сервер від імені суперкористувача root через SSH, навіть якщо зловмисник знає пароль. Параметр PasswordAuthentication no відключає аутентифікацію за паролем, залишаючи єдиним дозволеним методом криптографічні ключі. Параметр MaxAuthTries 3 обмежує кількість спроб аутентифікації у межах одного TCP-з'єднання [36].

Конфігурацію фаєрволу UFW виконано за принципом «заборонено все, що явно не дозволено». За замовчуванням UFW блокує всі вхідні з'єднання та дозволяє всі вихідні. Дозвільні правила додаються лише для трьох портів, необхідних для функціонування платформи [32]:

```
sudo ufw default deny incoming
```

```
sudo ufw default allow outgoing
sudo ufw allow 22/tcp    # SSH-доступ для адміністрування
sudo ufw allow 80/tcp    # HTTP
sudo ufw allow 443/tcp   # HTTPS
sudo ufw enable
```

Активований фаєрвол відкидає всі TCP та UDP пакети, що надходять на будь-які порти, крім 22, 80 та 443. Таким чином, навіть якщо PostgreSQL або будь-який інший внутрішній сервіс помилково отримає зовнішню IP-адресу, UFW заблокує спроби підключення до нього із мережі Інтернет ще на рівні операційної системи, ще до того, як пакет досягне контейнера [32].

Таблиця 3.1 - Конфігурація правил фаєрволу UFW для виробничого сервера

Порт	**Протокол**	**Статус**	**Призначення**
22	TCP	ALLOW	SSH-адміністрування та автоматизований деплой
80	TCP	ALLOW	HTTP: обробка запитів Let's Encrypt, редирект на HTTPS
443	TCP	ALLOW	HTTPS: основний трафік платформи
5432	TCP	DENY	PostgreSQL: заблоковано, доступний лише всередині Docker-мережі
3000	TCP	DENY	Backend API: заблоковано, доступний лише через Nginx
Всі інші	-	DENY	Стандартне правило відмови

3.2 Розробка Docker-інфраструктури: Dockerfile та docker-compose.yml

Контейнеризація компонентів вебплатформи є базовим інфраструктурним рішенням, що гарантує ізольованість, відтворюваність та незалежність середовища виконання від конфігурації хостової операційної системи [15]. Docker-інфраструктура проєкту описується двома взаємопов'язаними типами

конфігураційних файлів: індивідуальними декларативними інструкціями Dockerfile для кожного окремого модуля застосунку, які визначають покроковий процес автоматизованої побудови незмінних образів, та єдиним централізованим файлом docker-compose.yml, що забезпечує опис процесів одночасного запуску, зв'язку, масштабування та оркестрації мультиконтейнерної архітектури [15].

При проєктуванні архітектури образів для прикладних сервісів платформи було застосовано патерн багатоетапної збірки. Використання цього підходу є критично важливим для виробничого середовища, оскільки воно дозволяє фізично розмежувати етап компіляції та інсталяції залежностей розробника від фінального етапу виконання бінарних файлів або скриптів. Це забезпечує мінімізацію розміру підсумкового образу за рахунок виключення важких інструментів збірки, компіляторів та тимчасових кеш-файлів пакетних менеджерів, що безпосередньо впливає на швидкість розгортання системи через CI/CD конвеєр та зменшує поверхню потенційних атак на сервер [17].

Для прикладного бекенд-сервісу було розроблено конфігурацію Dockerfile, лістинг якої наведено нижче:

```
Dockerfile
#Етап 1: Середовище збірки артефактів
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .

# Етап 2: Виробниче середовище мінімального виконання
FROM node:20-alpine AS production
WORKDIR /app
ENV NODE_ENV=production
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app .
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

```
EXPOSE 3000
CMD ["node", "src/server.js"]
```

Аналіз наведеної конфігурації показує, що на першому етапі як базовий образ використовується оптимізований дистрибутив node:20-alpine, який характеризується мінімальним початковим розміром. Директива RUN npm ci –only = production виконує детерміноване встановлення виключно виробничих залежностей на основі блокувального файлу package-lock.json, ігноруючи інструменти розробки [15].

На другому етапі створюється новий чистий контекст, куди за допомогою інструкції COPY --from=builder імпортуються лише необхідні для роботи модулі та вихідний код застосунку. Інженерний інтерес становить реалізація безпеки на рівні контейнера, за замовчуванням процеси в Docker виконуються з правами суперкористувача root, що створює серйозну вразливість у разі компрометації коду програми. Завдяки директивам RUN addgroup ... && adduser ... та примусовому перемиканню через USER appuser, процес застосунку ізолюється та виконується від імені непривілейованого користувача, що позбавляє потенційного зловмисника можливості модифікувати системні файли всередині контейнера або здійснити атаку типу Container Breakout [17, 22].

Аналогічний паттерн багатоетапної збірки реалізовано для фронтенд-частини платформи, де на першому кроці Node.js компілює та мініфікує вихідний код у статичні артефакти HTML, CSS, JS, а на другому кроці ці файли інтегруються у високопродуктивний вебсервер Nginx для подальшої статичної роздачі:

```
Dockerfile
# Етап 1: Компіляція та оптимізація React SPA
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
ARG REACT_APP_API_URL
```

```
RUN npm run build

# Етап 2: Виробнича роздача статики через Nginx
FROM nginx:1.25-alpine AS production
COPY --from=builder /app/build /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Особливістю даної конфігурації є директива ARG REACT_APP_API_URL. Вона дозволяє динамічно передавати адресу розташування API-сервера безпосередньо в момент компіляції образу під час виконання CI/CD конвеєра, запобігаючи хардкоду чутливих конфігураційних адрес у сховищі вихідного коду [15].

Для поєднання зазначених ізольованих образів у цілісну інфраструктурну екосистему розроблено файл декларативної оркестрації. Повний лістинг конфігураційного файлу мультиконтейнерного розгортання docker-compose.yml винесено в Додаток А [33].

Аналіз архітектури, описаної в docker-compose.yml, дозволяє виділити кілька критично важливих інженерних рішень, що забезпечують стабільність та захищеність системи:

1. Забезпечення консистентності старту сервісів за допомогою розширеного механізму перевірки стану. Стандартне використання блоку залежностей depends_on гарантує лише те, що Docker Engine ініціює запуск процесу контейнера бази даних раніше за прикладний бекенд. Проте реляційна СУБД PostgreSQL потребує певного часу для внутрішньої ініціалізації, створення системних таблиць та відкриття мережевого порту 5432. Спроба бекенд-сервісу підключитися до СУБД у цей проміжок часу призводить до аварійного завершення роботи API-сервера. Для усунення цього архітектурного недоліку в сервіс postgres інтегровано блок healthcheck із тестуванням через утиліту pg_isready. Поєднання цього механізму з директивою condition: service_healthy у блоці залежностей бекенд-сервісу

гарантує, що контейнер додатка почне завантаження лише після повної готовності СУБД до прийому мережевих з'єднань [25].

2. Сегментація та ізоляція мережевих просторів. Мережева топологія проєкту розділена на два ізольовані контури за допомогою вбудованих драйверів типу bridge. Зовнішній контур об'єднує вебсервер Nginx, який приймає зовнішній трафік користувачів, та клієнтський інтерфейс. Внутрішній контур поєднує виключно ядро бекенд-додатка та систему управління базами даних PostgreSQL. Мережа backend_net сконфігурована з параметром internal: true, що повністю блокує будь-яку маршрутизацію пакетів із цієї мережі у глобальний Інтернет. Таким чином реалізовано принцип мінімальних привілеїв на рівні мережевої топології, СУБД фізично не має доступу до зовнішньої мережі, а зовнішній зловмисник не може здійснити пряме мережеве сканування чи підключення до порту 5432, минаючи захисні бар'єри проксі-сервера та прикладного коду [17].
3. Персистентність та оптимізація збереження даних. Оскільки контейнери за своєю природою є ефемерними, оскільки дані всередині їхнього writable-шару повністю знищуються при перезапуску або оновленні образу, для забезпечення збереженості інформації СУБД налаштовано іменований том даних з назвою rd_pgdata. Том монтується до внутрішньої директорії /var/lib/postgresql/data і керується безпосередньо Docker-демоном. Використання ізольованих іменованих томів замість прямого монтування папок хоста є рекомендованою практикою для баз даних, оскільки воно забезпечує вищу продуктивність операцій введення-виведення на SSD-дисках у хмарних середовищах та гарантує повне збереження транзакційних логів і даних між сесіями оновлення версій платформи [15].

3.3 Налаштування CI/CD конвеєра: автоматизація від коміту до деплою

Налаштований конвеєр GitHub Actions автоматично виконує повний цикл перевірки, збірки та доставки змін на виробничий сервер при кожному злитті Pull

Request до гілки main. Це виключає необхідність ручного втручання адміністратора в процес оновлення платформи [21]. Декларативний конфігураційний файл конвеєра розміщується у спеціалізованій системній директорії .github/workflows/ репозиторію проєкту та має розширення .yml. Файл має чітку структуру, що описує автоматичні тригери запуску, ізольоване середовище виконання та сувору послідовність технологічних кроків. Повний лістинг декларативного файлу автоматизованого конвеєра деплою deploy.yml винесено в Додаток Б.

Практика показує, що найбільш поширеною помилкою при побудові конвеєрів є відсутність чіткого розмежування між задачами збірки та доставки. Об'єднання цих процесів в одну задачу призводить до ситуації, коли навіть незначна мережева проблема при підключенні до сервера скасовує вже виконану збірку образу, змушуючи запускати ресурсомісткий процес повторно. Розділення на дві незалежні задачі вирішує цю проблему структурно: якщо перша задача завершилась успішно і образ опублікований у реєстрі, друга задача може бути перезапущена без повторної компіляції [21].

Середовище виконання конвеєрних задач є ефемерним - кожен запуск отримує свіжу віртуальну машину під управлінням ubuntu-latest без жодного стану від попередніх запусків. Це одночасно є перевагою і накладає певні обмеження. Перевага полягає у повній відтворюваності: конвеєр не може непомітно «зламатись» через накопичений стан попередніх запусків. Обмеження - кожного разу доводиться заново встановлювати залежності та завантажувати базові образи. Для пом'якшення цього недоліку у конфігурації задіяно механізм кешування шарів Docker між запусками. Кеш зберігається на серверах GitHub і прив'язується до конкретної гілки та хешу файлів залежностей. При незмінних залежностях повторна збірка відбувається значно швидше, оскільки більшість шарів підтягується з кешу, а не будується з нуля [21].

Окремої уваги заслуговує питання атомарності оновлення. Команда docker compose up з відповідними параметрами забезпечує послідовну заміну контейнерів: новий контейнер запускається та перевіряється перед тим, як старий

зупиняється. Для бази даних і сервісів зі станом це особливо важливо, некоректне завершення транзакцій при «жорсткій» зупинці могло б призвести до пошкодження даних. PostgreSQL коректно обробляє сигнал завершення і завершує активні транзакції перед зупинкою, однак архітектура конвеєра не покладається лише на цю поведінку, перезапуск СУБД виноситься в окремий крок і виконується лише у разі зміни конфігурації, а не при кожному деплої застосунку [21].

Важливим аспектом є управління версіями образів. Тегування за хешем коміту дозволяє у будь-який момент визначити, яка саме версія коду запущена на сервері, достатньо порівняти тег активного контейнера з деревом комітів репозиторію. Це критично при розслідуванні інцидентів, якщо після певного деплою спостерігається деградація продуктивності, можна точно встановити момент появи проблеми та за потреби відкотитись до попереднього образу без повторної збірки, оскільки всі попередні образи зберігаються у реєстрі [21].

Конфігурація середовища production як окремого Environment у налаштуваннях репозиторію дозволяє додатково контролювати доступ до секретів та налаштувати обов'язкове ручне підтвердження деплою. Для поточного етапу розвитку проєкту автоматичне розгортання без підтвердження є прийнятним, однак архітектура конвеєра передбачає можливість активації захисного механізму однією зміною у налаштуваннях без модифікації файлу конфігурації [21].

Конвеєр складається з двох незалежних задач jobs. Перша задача build-and-push відповідає за збірку Docker-образів та їх публікацію до реєстру GitHub Container Registry. Використання GHCR обґрунтоване повною інтеграцією з аутентифікацією GitHub без необхідності налаштування зовнішнього сервісу на зразок Docker Hub [21, 34]. Тег образу формується на основі хешу коміту git (sha-), що забезпечує унікальність та відстежуваність кожної збірки. Директива cache-from та cache-to дозволяє кешувати шари Docker-образу між запусками конвеєра, що суттєво прискорює наступні збірки [21].

Друга задача deploy виконується лише після успішного завершення першої. Вона підключається до виробничого сервера по SSH та виконує послідовність команд для оновлення платформи. Команда docker compose pull завантажує нові версії образів з реєстру, docker compose up -d --remove-orphans запускає оновлені контейнери та видаляє контейнери зі старих сервісів, яких вже немає у файлі compose, а docker image prune -f очищує сервер від застарілих образів для звільнення дискового простору [15].

Усі чутливі дані зберігаються у зашифрованих секретах GitHub Actions. Жоден із секретів не з'являється у логах виконання конвеєра, оскільки GitHub автоматично маскує рядки, що відповідають оголошеним секретам [21].

Таблиця 3.2 - Налаштовані секрети GitHub Actions та їх призначення

Назва секрету	**Тип даних**	**Призначення**
SERVER_HOST	IP-адреса	Зовнішня IP-адреса виробничого сервера GCP
SSH_PRIVATE_KEY	RSA/Ed25519 ключ	Приватний SSH-ключ для підключення від імені користувача deploy
GHCR_TOKEN	Personal Access Token	Токен для аутентифікації в GitHub Container Registry на сервері
DB_PASSWORD	Рядок	Пароль до бази даних PostgreSQL
JWT_SECRET	Рядок	Секретний ключ для підпису JSON Web Token

3.4 Робота з веб-сервером Nginx: налаштування SSL та зворотного проксі

Nginx виступає єдиною точкою входу для всього зовнішнього трафіку платформи. Конфігурація веб-сервера складається з кількох блоків: глобального

блоку http, блоків server для обробки незашифрованого та зашифрованого трафіку, та блоків location для маршрутизації запитів до відповідних внутрішніх сервісів [13].

Отримання SSL-сертифіката здійснюється за допомогою утиліти Certbot та протоколу ACME від центру сертифікації Let's Encrypt. Процедура отримання безкоштовного сертифіката виконується за методом HTTP-01 challenge, що вимагає тимчасового відкриття порту 80 для підтвердження права власності на доменне ім'я [26].

Встановлення необхідного програмного забезпечення та генерація сертифікатів здійснюються за допомогою наступних системних команд:

```
sudo apt install -y certbot python3-certbot-nginx
sudo certbot --nginx -d readviabooks.com -d www.readviabooks.com
```

Після успішного проходження перевірки утиліта Certbot автоматично розміщує згенеровані файли приватного ключа та ланцюжка сертифікатів у системній директорії хоста /etc/letsencrypt/live/readviabooks.com/. Спроєктована архітектура розгортання передбачає монтування цієї директорії всередину Docker-контейнера Nginx у режимі тільки для читання, що унеможливлює модифікацію ключів зсередини вебсервера та дозволяє проводити процедуру перевипуску сертифікатів без необхідності зупинки сервісів або перезбірки Docker-образів [27]. Оскільки термін дії сертифікатів Let's Encrypt обмежений 90 днями, автоматизація їх оновлення реалізується за допомогою системного планувальника cron:

```
0 0,12 * * * certbot renew --quiet
```

Зазначена задача двічі на добу ініціює фонову перевірку терміну дії сертифіката і, за умови залишкового періоду менше 30 днів, здійснює його автоматичний перевипуск та гаряче перезавантаження конфігурації Nginx без переривання сесій активних користувачів [38]. Логіка обробки мережевих запитів у Nginx розділена на два автономні серверні блоки. Повний лістинг конфігураційного файлу налаштування вебсервера nginx.conf винесено в Додаток В.

Перший серверний блок налаштований на прослуховування порту 80 та обробку незашифрованого HTTP-трафіку. Основним завданням цього блоку є примусова переадресація всіх вхідних користувацьких запитів на захищений протокол HTTPS за допомогою постійного редиректу з кодом стану 301 Moved Permanently. Це гарантує, що дані клієнтів за будь-яких умов транспортуватимуться у зашифрованому вигляді. Єдиним архітектурним виключенням у цьому блоці є службовий URL-шлях /.well-known/acme-challenge/, який перенаправляється на локальну директорію /var/www/certbot для забезпечення безперешкодного автоматичного поновлення сертифікатів Let's Encrypt планувальником cron [13].

Другий серверний блок функціонує на порту 443 із підтримкою оптимізованого протоколу http2, який дозволяє здійснювати мультиплексування запитів у межах одного TCP-з'єднання, суттєво прискорюючи завантаження статичних елементів інтерфейсу [24].

Директива ssl_protocols TLSv1.2 TLSv1.3 обмежує підтримувані версії протоколу TLS, виключаючи застарілі та вразливі TLSv1.0 та TLSv1.1. Заголовок Strict-Transport-Security інформує браузер про те, що сайт доступний лише через HTTPS протягом наступних 365 днів, навіть якщо користувач явно введе http:// у адресному рядку [24].

Директива proxy_set_header X-Forwarded-For передає бекенд-сервісу реальну IP-адресу клієнта. Без цього заголовка бекенд бачив би лише IP-адресу контейнера Nginx як джерело запиту, що унеможливлює ведення журналів доступу та реалізацію обмеження частоти запитів на рівні застосунку [24].

РОЗДІЛ 4. ЕКСПЛУАТАЦІЯ, БЕЗПЕКА ТА МОНІТОРИНГ

4.1 Моніторинг ресурсів та продуктивності інфраструктури

Безперервний моніторинг стану серверної інфраструктури є обов'язковою умовою надійної експлуатації будь-якої веб-платформи. Він дозволяє виявляти аномалії у споживанні ресурсів до того, як вони призводять до деградації сервісу або повної відмови системи [10]. Для моніторингу платформи впроваджено систему Netdata легковісного агента збору метрик з відкритим вихідним кодом.

Вибір Netdata обґрунтовується кількома практичними чинниками. По-перше, інструмент встановлюється та запускається єдиною командою, не потребуючи складного налаштування стеків на зразок Prometheus + Grafana, що актуально для проєкту з обмеженими адміністративними ресурсами. По-друге, Netdata збирає метрики з роздільною здатністю 1 секунда, що дозволяє виявляти короткочасні сплески навантаження, невидимі для систем із хвилинним інтервалом збору [29]. По-третє, агент має вбудовану підтримку збору метрик Docker-контейнерів через API Docker Engine без необхідності встановлення додаткових exporters [29].

Встановлення Netdata виконується через офіційний скрипт-інсталятор:

```
wget -O /tmp/netdata-kickstart.sh https://my
netdata.io/kickstart.sh
sudo sh /tmp/netdata-kickstart.sh --stable-channel --disable-
telemetry
```

Після встановлення агент запускається як системний сервіс та доступний за адресою http://localhost:19999. Для убезпечення веб-інтерфейсу від несанкціонованого доступу через мережу Інтернет налаштовано обмеження доступу до цього порту виключно з localhost, а адміністративний доступ до дашборду здійснюється через SSH-тунель [29]:

ssh -L 19999:localhost:19999 deploy@SERVER_IP

Netdata автоматично виявляє та відображає метрики по наступних категоріях ресурсів. Використання CPU відображається у розрізі кожного

процесора та виду навантаження: системне, користувацьке, очікування вводу-виводу. Метрики оперативної пам'яті розподіляються на активну, кешовану та вільну. Дискова активність відображає операції читання та запису у байтах за секунду та кількість операцій вводу-виводу. Мережева активність фіксує трафік по кожному мережевому інтерфейсу [29].

Для Docker-контейнерів Netdata відображає окремий набір метрик по кожному запущеному контейнеру: споживання CPU відносно ліміту, обсяг використаної пам'яті, мережевий трафік та стан контейнера. Це дозволяє оперативно виявляти, який саме сервіс є причиною навантаження на сервер [29].

Таблиця 4.1 – Ключові метрики моніторингу та порогові значення сповіщень

Метрика	**Ресурс**	**Попередження**	**Критичний рівень**
Завантаженість CPU	Сервер	> 75% протягом 5 хв	> 90% протягом 2 хв
Використання RAM	Сервер	> 80%	> 95%
Зайнятість диску	Сервер	> 70%	> 85%
CPU контейнера backend	Docker	> 60%	> 80%
Пам'ять контейнера postgres	Docker	> 512 МБ	> 700 МБ
Час відповіді Nginx	Сервіс	> 500 мс	> 2000 мс

Практична цінність системи моніторингу виявляється не стільки у постійному спостереженні за дашбордом, скільки у можливості ретроспективного аналізу. Netdata зберігає детальну часову послідовність усіх зібраних метрик локально на сервері з роздільною здатністю 1 секунда за останні кілька годин та зниженою роздільною здатністю за попередні дні. Завдяки цьому після отримання сповіщення про інцидент адміністратор може відновити точну

картину того, що відбувалось із ресурсами сервера в момент виникнення проблеми, навіть якщо він отримав сповіщення із затримкою [29].

Окремим інструментом діагностики є граф залежностей між метриками. Наприклад, різке зростання часу відповіді Nginx може бути спричинене не проблемою у самому вебсервері, а вичерпанням пулу з'єднань до бекенду через надмірне споживання пам'яті контейнером PostgreSQL, яка, у свою чергу, може бути наслідком неоптимального запиту без індексу [10]. Послідовний аналіз метрик у хронологічному порядку дозволяє встановити першопричину замість усунення симптомів [33].

Для спостереження за тенденціями у довгостроковій перспективі Netdata підтримує відправку метрик до зовнішніх сховищ часових рядів. У поточній конфігурації цей механізм не активований, оскільки для MVP-стадії проєкту достатньо вбудованого локального сховища. Проте при масштабуванні платформи та появі другого сервера розгортання централізованого сховища метрик стає необхідністю відстеження тенденцій одночасно на кількох вузлах в окремих інтерфейсах є неефективним [10]. Архітектура Netdata передбачає роль батьківського вузла Parent, який агрегує потоки метрик від кількох дочірніх агентів Child, що забезпечує природний шлях до централізованого моніторингу без зміни інструменту.

Окремо варто відзначити функцію виявлення аномалій на основі навчених моделей. Netdata вбудовує легковісні ML-моделі, які навчаються на власних даних кожного вузла і здатні виявляти нетипову поведінку метрик без ручного визначення порогів [39]. Практично це означає, що система може підняти сповіщення про підозрілий трафік навіть тоді, коли абсолютні значення метрик ще не перетнули критичних порогів, достатньо, щоб патерн поведінки статистично відрізнявся від норми для цього часового відрізку доби [29].

Система сповіщень Netdata підтримує відправку алертів через електронну пошту, Slack та Telegram. Для платформи налаштовано сповіщення у Telegram-бот адміністратора при досягненні попередженого рівня метрик. Конфігурація

виконується редагуванням файлу /etc/netdata/health_alarm_notify.conf, де вказуються токен Telegram-бота та ідентифікатор чату адміністратора [29].

Додатково для відстеження HTTP-рівня метрик кількості запитів, часу відповіді та кодів статусу налаштовано парсинг журналів доступу Nginx через вбудований модуль Netdata web_log. Це дозволяє оперативно виявляти зростання кількості помилок 5xx, що свідчить про проблеми з бекенд-сервісом, або збільшення кількості запитів 4xx, що може вказувати на спробу сканування вразливостей [29].

4.2 Стратегія резервного копіювання: автоматизовані скрипти та ротація архівів

Втрата даних користувачів через відмову обладнання, помилку адміністратора або кібератаку є катастрофічним сценарієм для будь-якої платформи. Регулярне резервне копіювання бази даних є першочерговим заходом захисту від таких ситуацій [25]. Розроблена стратегія базується на щоденному автоматизованому резервному копіюванні за допомогою утиліти pg_dump та збереженні копій у зовнішньому сховищі Google Cloud Storage.

Bash-скрипт резервного копіювання розміщено у файлі /opt/scripts/backup_db.sh та виконується від імені користувача deploy. Скрипт виконує наступну послідовність дій: підключення до контейнера PostgreSQL, створення дампу бази даних у форматі custom, архівування файлу, передача до хмарного сховища, локальна очистка. Повний лістинг автоматизованого скрипта резервного копіювання backup_db.sh наведено у Додатку Г.

Директива set -euo pipefail на початку скрипту є обов'язковою для продуктивних скриптів резервного копіювання. Вона забезпечує негайне завершення виконання при виникненні будь-якої помилки, помилці при зверненні до неоголошених змінних та помилці у будь-якій команді конвеєра. Без цих опцій скрипт міг би завершитись «успішно», навіть якщо дамп бази даних не був створений через відмову контейнера [25].

Формат custom утиліти pg_dump є оптимальним для резервних копій: він забезпечує вбудоване стиснення, паралельне відновлення через pg_restore та гнучкість вибору об'єктів при відновленні [25]. Розмір дампу у форматі custom для бази даних з 10 000 записами становить близько 5-15 МБ.

Автоматизація запуску скрипту здійснюється через планувальник задач cron. Задача додається до crontab користувача deploy:

```
crontab -e
# Vnesennia takogo zapysu:
0 3 * * * /opt/scripts/backup_db.sh >> /var/log/backup.log 2>&1
```

Запис 0 3 * * * означає виконання щодня о 03:00 за часовим поясом сервера. Вибір нічного часу обґрунтований мінімальним навантаженням на базу даних у цей проміжок, що зменшує час блокувань при створенні дампу. Перенаправлення виводу >> /var/log/backup.log 2>&1 забезпечує збереження всіх повідомлень та помилок скрипту до журнального файлу для подальшого аналізу [25].

Для перевірки цілісності резервних копій необхідно регулярно тестувати процедуру відновлення. Тестове відновлення виконується на окремому Docker-контейнері з PostgreSQL, що виключає вплив на виробничу базу даних [25]:

```
docker run --rm \
  -v /var/backups/readvia:/backups \
  -e POSTGRES_USER=testuser \
  -e POSTGRES_DB=testdb \
  postgres:16-alpine \
  pg_restore -U testuser -d testdb
/backups/readvia_db_TIMESTAMP.dump
```

Таблиця 4.2 – Параметри стратегії резервного копіювання

Параметр	**Значення**	**Обґрунтування**
Частота	Щодня о 03:00	Мінімальне навантаження на систему, допустима втрата даних 1 доба
Формат	pg_dump custom	Стиснення, вибіркове відновлення, сумісність між версіями PostgreSQL

Локальне зберігання	30 днів	Достатній термін для виявлення проблем і повернення до стабільного стану
Хмарне сховище	GCS (30 копій)	Захист від фізичної відмови диску або випадкового видалення
Тест відновлення	Щотижнево	Підтвердження цілісності та можливості відновлення копії

4.3 Аналіз та реалізація заходів безпеки інфраструктури

Комплексна безпека інфраструктури платформи будується на декількох незалежних рівнях захисту. Принцип глибокої ешелонованої оборони передбачає, що компрометація одного рівня захисту не призводить до повного порушення безпеки системи. Для платформи реалізовано три ключові заходи безпеки: захист від атак підбору паролів через Fail2Ban, безпечне управління чутливими конфігураційними даними через змінні середовища, та системне розмежування привілеїв [14, 31].

Fail2Ban є системним демоном, що автоматично аналізує журнали системи та тимчасово блокує IP-адреси, з яких виявлено підозрілу активність. Для серверів з відкритим портом SSH атаки методом підбору облікових даних є постійною загрозою: за статистикою, звичайний сервер отримує від сотень до тисяч спроб несанкціонованого входу по SSH щодня [28].

```
sudo apt install -y fail2ban
sudo systemctl enable fail2ban
```

Конфігурація Fail2Ban організована за принципом розшарування: глобальний файл jail.conf містить значення за замовчуванням від розробників і не редагується вручну, натомість всі зміни вносяться до файлу jail.local, який перевизначає лише необхідні параметри. Такий підхід гарантує, що оновлення пакету не перезапише налаштування безпеки [28].

Окрім захисту SSH, налаштовано додаткову пастку для вебтрафіку, що аналізує журнали доступу Nginx. Повторювані запити до неіснуючих

адміністративних шляхів, типова ознака автоматизованого сканування вразливостей призводять до тимчасового блокування IP-адреси джерела ще до того, як будь-який із запитів досягне рівня застосунку. Це знижує паразитне навантаження на бекенд і одночасно ускладнює розвідку структури сервісу [28].

Важливим аспектом налаштування є визначення списку виключень. IP-адреси самого адміністратора та адреси серверів GitHub Actions, що підключаються для виконання деплою, внесені до списку ігнорування. Без цього існував би ризик автоматичного блокування CI/CD-агента у разі короткочасних мережевих збоїв, що спровокували б множинні спроби перепідключення, ситуація, за якою Fail2Ban не зміг би відрізнити легітимний ретрай від атаки [28].

Моніторинг активності Fail2Ban виконується через перегляд журналу подій та поточного стану пасток. Статистика заблокованих адрес є непрямим індикатором інтенсивності зовнішнього сканування і розглядається у комплексі з метриками мережевого трафіку з дашборду Netdata. Регулярний аналіз цих даних дозволяє відстежувати зміни у характері зовнішніх загроз і своєчасно коригувати параметри чутливості системи захисту [28].

Безпечне зберігання чутливих конфігураційних даних є критично важливою практикою, якої дотримуються у проєкті. До таких даних належать паролі до бази даних, секретні ключі для підпису JWT-токенів, токени доступу до зовнішніх сервісів. Зберігання цих даних безпосередньо у вихідному коді або файлах docker-compose.yml є грубою помилкою безпеки, оскільки репозиторій проєкту може бути публічним або отримати несанкціонований доступ [37]. Для платформи реалізовано підхід на основі файлу .env. Цей файл містить всі чутливі змінні середовища та ніколи не додається до системи контролю версій. У репозиторії натомість зберігається файл .env.example з переліком необхідних змінних без їх значень, що слугує документацією для нових членів команди [37]:

```
# Fajl .env.example
POSTGRES_USER=
POSTGRES_PASSWORD=
POSTGRES_DB=
```

```
JWT_SECRET=
JWT_EXPIRES_IN=
NODE_ENV=
```

Фактичний файл .env на виробничому сервері містить реальні значення та має права доступу 600, що унеможливлює його читання будь-ким, крім користувача deploy, від імені якого запускаються контейнери [38]:

```
sudo chmod 600 /var/www/readvia/.env
sudo chown deploy:deploy /var/www/readvia/.env
```

При запуску docker compose директива env_file: .env у конфігурації сервісу бекенду передає всі змінні з файлу .env до середовища контейнера. Бекенд-застосунок зчитує ці змінні через process.env (Node.js), не маючи доступу до файлової системи хостової ОС. Таким чином реалізується ізоляція секретів на рівні контейнера [11].

Таблиця 4.3 – Комплексна матриця заходів безпеки інфраструктури

Загроза	**Захисний захід**	**Рівень реалізації**
Brute-force атаки на SSH	Fail2Ban з блокуванням на 24 год після 3 спроб	ОС (iptables)
Несанкціонований SSH-доступ	Аутентифікація виключно за ключами, відключення паролів	SSH-демон
Витік чутливих даних	Змінні середовища (.env), виключені з git	Застосунок
Прямий доступ до БД	Мережа backend_net без зовнішнього відображення портів	Docker-мережа
Мережеві атаки	UFW: дозволено лише порти 22, 80, 443	ОС (netfilter)

Привілейовані контейнери	Запуск від імені непривілейованого USER у Dockerfile	Docker
MITM-атаки	Обов'язковий HTTPS (TLS 1.2/1.3) + HSTS заголовок	Nginx
Clickjacking	Заголовок X-Frame-Options: SAMEORIGIN	Nginx

Для регулярного контролю стану безпеки системи проводиться перевірка аналіз журналів Fail2Ban щодо заблокованих IP-адрес, перегляд списку авторизованих SSH-ключів, аудит відкритих портів за допомогою команди ss -tulpn, перевірка прав доступу до критичних файлів конфігурації [37].

4.4 Навантажувальне тестування та перевірка відмовостійкості інфраструктури

Фінальним етапом розгортання та налаштування інфраструктури вебплатформи є проведення комплексного навантажувального тестування. Метою цього дослідження є перевірка стабільності роботи розробленої Docker-архітектури, оцінка пропускної здатності зворотного проксі-сервера Nginx та реляційної СУБД PostgreSQL, а також визначення поведінки системи в умовах пікових навантажень, які властиві платформам електронної комерції під час високої активності користувачів [20].

Для проведення експерименту було використано утиліту з відкритим вихідним кодом k6 від компанії Grafana Labs, яка дозволяє декларативно описувати сценарії тестування на мові JavaScript та генерувати високе асинхронне навантаження [20]. Сценарій тестування симулював поведінку 500 та 1000 одночасних віртуальних користувачів, які протягом 5 хвилин безперервно виконували типові операції на сайті, завантаження головної сторінки та надсилання запитів до API.

Під час тестування моніторинг апаратних ресурсів віртуальної машини здійснювався за допомогою розгорнутої системи Netdata. Результати тестування дозволили зафіксувати ключові метрики інфраструктури [39]. Зокрема, час відповіді сервера при навантаженні у 500 одночасних користувачів у середньому становив 42 мс для обробки статичного контенту сервером Nginx та 115 мс для динамічних API-запитів із СУБД. При підвищенні навантаження до 1000 користувачів час відповіді стабілізувався на позначці 180 мс, що є відмінним показником для вебзастосунків.

Щодо інтенсивності обробки запитів, система успішно витримала пікове навантаження в районі 450 запитів за секунду без падіння процесів у контейнерах [40]. Рівень помилок за весь період тестування, тобто кількість неуспішних відповідей склав лише 0.04%, що впевнено вкладається в межі допустимої норми для високонавантажених систем, менше 1% [20].

Завдяки налаштованим параметрам оптимізації Nginx (Keep-Alive з'єднання, буферизація) та використанню Docker-мережі типу bridge, архітектура продемонструвала високу стійкість. Обмеження прав користувача бази даних та ізоляція мережі backend_net параметром internal: true не вплинули на швидкість внутрішньої маршрутизації пакетів. Таким чином, експериментальне навантажувальне тестування підтвердило, що розроблена Docker-інфраструктура готова до промислової експлуатації та здатна забезпечити безперебійну роботу платформи під час реального користувацького навантаження [20].

ВИСНОВКИ

У кваліфікаційній (бакалаврській) роботі вирішено практичне завдання проєктування та автоматизованого розгортання веб-платформи електронної комерції Readvia - C2C-сервісу для обміну та купівлі-продажу книг. Досягнення поставленої мети забезпечено шляхом послідовного виконання усіх визначених завдань.

У межах аналізу предметної галузі проведено порівняльне дослідження монолітної та мікросервісної архітектур, на основі якого обґрунтовано доцільність використання гібридного підходу контейнеризованого модульного застосунку. Показано, що на початковому етапі розвитку проєкту контейнеризація забезпечує оптимальне поєднання швидкості розробки та готовності до автоматизації. Проведено детальний порівняльний аналіз серверних дистрибутивів Linux, за результатами якого обрано Ubuntu Server 22.04 LTS як найбільш збалансоване рішення з нативною підтримкою Docker та хмарних середовищ. Обґрунтовано вибір GitHub Actions як системи CI/CD автоматизації на основі критеріїв інтеграції з Git-хостингом, наявності готових компонентів для роботи з Docker та Google Cloud Platform, а також безсерверної архітектури виконання.

На етапі проєктування розроблено ER-діаграму бази даних з трьома основними сутностями та системою рейтингу, обрано PostgreSQL як СУБД з обґрунтуванням переваг над MySQL в контексті складних транзакцій C2C-обміну. Спроєктовано схему розгортання з Nginx як зворотним проксі та дворівневу мережеву топологію з розмежуванням frontend_net та backend_net для забезпечення ізоляції бази даних від зовнішнього трафіку.

У процесі технічної реалізації виконано повний цикл DevOps-інженерії. Налаштовано виробничий сервер на базі Ubuntu Server 22.04 LTS у Google Cloud Platform: розмежовано привілеї користувачів, впроваджено аутентифікацію за SSH-ключами та сконфігуровано фаєрвол UFW з принципом мінімального дозволу. Розроблено Dockerfile для бекенд- та фронтенд-сервісів із використанням багатоетапної збірки та запуском від непривілейованого

користувача. Описано docker-compose.yml для оркестрації чотирьох контейнерів із перевіркою готовності PostgreSQL через healthcheck. Налаштовано CI/CD конвеєр GitHub Actions із двома задачами: збіркою та публікацією Docker-образів до GHCR, та автоматизованим розгортанням на виробничий сервер по SSH. Сконфігуровано Nginx з HTTPS-шифруванням на базі сертифікатів Let's Encrypt, редиректом HTTP-HTTPS та параметрами безпеки TLS.

Для забезпечення надійної експлуатації впроваджено систему моніторингу Netdata з відображенням метрик сервера та Docker-контейнерів з роздільною здатністю 1 секунда та системою сповіщень у Telegram. Розроблено автоматизований Bash-скрипт щоденного резервного копіювання бази даних PostgreSQL з ротацією архівів та передачею до Google Cloud Storage. Реалізовано комплекс заходів безпеки: захист від brute-force атак через Fail2Ban, безпечне зберігання секретів у змінних середовища, мережева ізоляція компонентів на рівні Docker-мереж.

Практична значущість виконаної роботи полягає у створенні відтворюваної та задокументованої архітектурної моделі, що може бути адаптована для широкого класу веб-проєктів. Розроблені конфігураційні файли Docker та GitHub Actions, скрипти адміністрування та резервного копіювання утворюють цілісний DevOps-шаблон, впровадження якого на нових проєктах дозволяє скоротити час від початку розробки до першого виробничого розгортання.

Перспективами подальшого розвитку проєкту є впровадження Kubernetes для управління кластером контейнерів при зростанні навантаження, розширення системи моніторингу стеком Prometheus + Grafana для більш детальної аналітики, реалізація Infrastructure as Code через Terraform для повністю автоматизованого провізіонування серверної інфраструктури, а також впровадження автоматизованого тестування безпеки у CI/CD конвеєр.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Kim G., Humble J., Debois P., Willis J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. Portland: IT Revolution Press, 2021. 465 p.

2. Turnbull J. The Docker Book: Containerization is the new virtualization. 2nd ed. James Turnbull, 2018. 342 p.

3. Kane S., Matthias K. Docker: Up and Running: Shipping Reliable Containers in Production. 3rd ed. Sebastopol: O'Reilly Media, 2023. 384 p.

4. Burns B., Beda J., Hightower K. Kubernetes: Up and Running: Dive into the Future of Infrastructure. 3rd ed. Sebastopol: O'Reilly Media, 2022. 326 p.

5. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 616 p.

6. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Sebastopol: O'Reilly Media, 2017. 616 p.

7. Cantelon M., Harter M., Holowaychuk T. Node.js in Action. 2nd ed. Shelter Island: Manning Publications, 2017. 392 p.

8. Wiegers K., Beatty J. Software Requirements. 3rd ed. Redmond: Microsoft Press, 2013. 608 p.

9. Forsgren N., Humble J., Kim G. Accelerate: The Science of Lean Software and DevOps. Portland: IT Revolution Press, 2018. 288 p.

10. Turnbull J. The Art of Monitoring. James Turnbull, 2016. 507 p.

11. Beyer B., Jones C., Petoff J., Murphy R. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol: O'Reilly Media, 2016. 552 p.

12. Hoffman S. The Art of PostgreSQL. Mastering SQL and PostgreSQL. Tapoueh, 2021. 594 p.

13. Holt K. Nginx: From Beginner to Pro. New York: Apress, 2016. 292 p.

14. Messier R. Network Security Assessment. 3rd ed. Sebastopol: O'Reilly Media, 2016. 497 p.

15. Mouat A. Using Docker: Developing and Deploying Software with Containers. Sebastopol: O'Reilly Media, 2015. 354 p.

16. Hausenblas M., Schimanski S. Programming Kubernetes: Developing Cloud-Native Applications. Sebastopol: O'Reilly Media, 2019. 260 p.

17. Richardson C. Microservices Patterns: With examples in Java. Shelter Island: Manning Publications, 2018. 520 p.

18. Bass L., Weber I., Zhu L. DevOps: A Software Architect's Perspective. Boston: Addison-Wesley Professional, 2015. 384 p.

19. Hendrickson E. Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing. Dallas: Pragmatic Bookshelf, 2013. 186 p.

20. Abbott M., Fisher M. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. 2nd ed. Boston: Addison-Wesley Professional, 2015. 464 p.

21. Advanced Bash-Scripting Guide [Електронний ресурс]. – Режим доступу: https://tldp.org/LDP/abs/html/

22. Docker Documentation. Dockerfile best practices [Електронний ресурс]. – Режим доступу: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

23. GitHub Actions Documentation [Електронний ресурс]. – Режим доступу: https://docs.github.com/en/actions

24. Nginx Documentation [Електронний ресурс]. – Режим доступу: https://nginx.org/en/docs/

25. PostgreSQL 16 Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/16/index.html

26. Let's Encrypt Documentation [Електронний ресурс]. – Режим доступу: https://letsencrypt.org/docs/

27. Ubuntu Server Guide [Електронний ресурс]. – Режим доступу: https://ubuntu.com/server/docs

28. Fail2Ban Documentation [Електронний ресурс]. – Режим доступу: https://www.fail2ban.org/wiki/index.php/MANUAL_0_8

29. Netdata Documentation [Електронний ресурс]. – Режим доступу: https://learn.netdata.cloud/docs

30. Google Cloud Platform Documentation. Compute Engine [Електронний ресурс]. – Режим доступу: https://cloud.google.com/compute/docs

31. OWASP Top Ten [Електронний ресурс]. – Режим доступу: https://owasp.org/www-project-top-ten/

32. UFW - Uncomplicated Firewall. Ubuntu Manpage [Електронний ресурс]. – Режим доступу: https://manpages.ubuntu.com/manpages/jammy/en/man8/ufw.8.html

33. Docker Compose Specification [Електронний ресурс]. – Режим доступу: https://docs.docker.com/compose/compose-file/

34. GitHub Container Registry (GHCR) Documentation [Електронний ресурс]. – Режим доступу: https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry

35. Node.js Best Practices [Електронний ресурс]. – Режим доступу: https://github.com/goldbergyoni/nodebestpractices

36. CIS Benchmarks for Ubuntu Linux [Електронний ресурс]. – Режим доступу: https://www.cisecurity.org/benchmark/ubuntu_linux

37. The Twelve-Factor App Methodology [Електронний ресурс]. – Режим доступу: https://12factor.net

38. Certbot User Guide [Електронний ресурс]. – Режим доступу: https://eff-certbot.readthedocs.io/en/stable/

39. Stack Overflow Developer Survey 2024 [Електронний ресурс]. – Режим доступу: https://survey.stackoverflow.co/2024/

40. Google Cloud Storage Documentation [Електронний ресурс]. – Режим доступу: https://cloud.google.com/storage/docs

ДОДАТКИ

ДОДАТОК А

Лістинг конфігураційного файлу оркестрації docker-compose.yml

```
version: "3.9"

services:
  nginx:
    image: nginx:1.25-alpine
    container_name: rd_nginx
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - /etc/letsencrypt:/etc/letsencrypt:ro
    depends_on:
      - backend
      - frontend
    networks:
      - frontend_net
    restart: unless-stopped

  frontend:
    build:
      context: ./frontend
      args:
        REACT_APP_API_URL: https://readviabooks.com/api
    container_name: rd_frontend
    networks:
      - frontend_net
```

```
  restart: unless-stopped

 backend:
  build: ./backend
  container_name: rd_backend
  env_file: .env
  depends_on:
   postgres:
    condition: service_healthy
  networks:
   - frontend_net
   - backend_net
  restart: unless-stopped

 postgres:
  image: postgres:16-alpine
  container_name: rd_postgres
  env_file: .env
  volumes:
   - pgdata:/var/lib/postgresql/data
  healthcheck:
   test: ["CMD-SHELL", "pg_isready -U $$POSTGRES_USER -d
$$POSTGRES_DB"]
   interval: 10s
   timeout: 5s
   retries: 5
  networks:

   - backend_net
  restart: unless-stopped
```

```
networks:
  frontend_net:
    driver: bridge
  backend_net:
    driver: bridge
    internal: true

volumes:
  pgdata:
    name: rd_pgdata
```

ДОДАТОК Б

Лістинг конфігураційного файлу автоматизованого деплою deploy.yml

```
name: Build and Deploy RD

on:
  push:
    branches: [main]

jobs:
  build-and-push:
    name: Build Docker Images
    runs-on: ubuntu-latest
    outputs:
      image_tag: ${{ steps.meta.outputs.version }}

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to GitHub Container Registry
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract metadata for Docker
```

```
      id: meta
      uses: docker/metadata-action@v5
      with:
        images: ghcr.io/${{ github.repository }}/backend
        tags: type=sha,prefix=sha-

    - name: Build and push backend image
      uses: docker/build-push-action@v5
      with:
        context: ./backend
        push: true
        tags: ${{ steps.meta.outputs.tags }}
        cache-from: type=gha
        cache-to: type=gha,mode=max

  deploy:
    name: Deploy to Production
    runs-on: ubuntu-latest
    needs: build-and-push
    environment: production

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Copy compose files to server
        uses: appleboy/scp-action@v0.1.7
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: deploy
```

```
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        source: "docker-compose.yml,nginx/"
        target: /var/www/readvia

    - name: Pull and restart services
      uses: appleboy/ssh-action@v1.0.3
      with:
        host: ${{ secrets.SERVER_HOST }}
        username: deploy
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        script: |
          cd /var/www/readvia
          echo ${{ secrets.GHCR_TOKEN }} | docker login ghcr.io -u ${{ github.actor
}} --password-stdin
          docker compose pull
          docker compose up -d --remove-orphans
          docker image prune -f
```

ДОДАТОК В

Лістинг конфігураційного файлу налаштування вебсервера nginx.conf

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log notice;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';

    access_log /var/log/nginx/access.log main;

    sendfile on;
    keepalive_timeout 65;
    gzip on;

    server {
        listen 80;
        server_name readviabooks.com www.readviabooks.com;

        location /.well-known/acme-challenge/ {
```

```
        root /var/www/certbot;
    }

    location / {
        return 301 https://$host$request_uri;
    }
}

server {
    listen 443 ssl http2;
    server_name readviabooks.com www.readviabooks.com;

    ssl_certificate /etc/letsencrypt/live/readviabooks.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/readviabooks.com/privkey.pem;

    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers   ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-
GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-
GCM-SHA384;
    ssl_prefer_server_ciphers off;
    ssl_session_cache shared:SSL:10m;
    ssl_session_timeout 1d;

    add_header Strict-Transport-Security "max-age=31536000; includeSubDomains"
always;
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header X-XSS-Protection "1; mode=block" always;

    location / {
```

```
        proxy_pass http://frontend:80;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_http_version 1.1;
        proxy_set_header Connection "";
    }
    location /api/ {
        proxy_pass http://backend:3000/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_connect_timeout 60s;
        proxy_send_timeout 60s;
        proxy_read_timeout 60s;
    }
  }
}
```

ДОДАТОК Г

Лістинг автоматизованого резервного копіювання backup_db.sh

```
#!/bin/bash
set -euo pipefail

BACKUP_DIR="/var/backups/readvia"
TIMESTAMP=$(date +"%Y%m%d_%H%M%S")
BACKUP_FILE="${BACKUP_DIR}/readvia_db_${TIMESTAMP}.dump"
GCS_BUCKET="gs://readvia-backups"
RETENTION_DAYS=30

mkdir -p "${BACKUP_DIR}"

docker exec readvia_postgres pg_dump \
  -U "${POSTGRES_USER}" \
  -d "${POSTGRES_DB}" \
  --format=custom \
  --compress=9 \
  > "${BACKUP_FILE}"

gsutil cp "${BACKUP_FILE}" "${GCS_BUCKET}/daily/"

find "${BACKUP_DIR}" -name "*.dump" -mtime "+${RETENTION_DAYS}" -
delete

gsutil -m rm -f $(gsutil ls "${GCS_BUCKET}/daily/" | sort | head -n -30) 2>/dev/null
|| true

echo "[$(date)] Backup completed: ${BACKUP_FILE}"
```

ДОДАТОК Д

Копія сертифіката про успішне завершення грантової програми «Право на бізнес» та апробацію результатів дослідження веб-платформи

Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation
Швейцарська Конфедерація

ПРАВО НА ЗАХИСТ
Благодійний фонд

Ukraine Economic Resilience Programme

СЕРТИФІКАТ

ЦЕЙ СЕРТИФІКАТ ЗАСВІДЧУЄ, ЩО

Кизь Олександр

прослухав(ла) навчальний курс

ПРАВО НА БІЗНЕС. КУРС ДЛЯ ПІДПРИЄМЦІВ

Проєкт реалізується БФ «Право на захист» за підтримки Швейцарії через Швейцарську агенцію розвитку та співробітництва в межах Програми економічної стійкості України (ERP). Програму впроваджує ERP консорціум, до якого входять Mercy Corps, БФ «Право на захист», Helvetas Swiss Intercooperation і JERU (Спільне реагування на надзвичайні ситуації в Україні організацій Welthungerhilfe та Concern Worldwide).

2025 рік

Олександр Галкін,
Президент БФ «Право на захист»

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)