

**Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення**

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА ВЕБ СЕРВІСУ ДЛЯ ТРЕКІНГУ ТА УПРАВЛІННЯ
ПОШТОВИМИ ВІДПРАВЛЕННЯМИ НА ОСНОВІ REACT І NESTJS**

**DEVELOPMENT OF A WEB SERVICE FOR TRACKING AND MANAGING
POSTAL SHIPMENTS BASED ON REACT AND NESTJS**

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-42
Пуршак Ілля Андрійович

Керівник:
Андрущак Ігор Євгенович

Кваліфікаційну роботу
допущено до захисту
«___» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Галузь знань: 12 «Інформаційні технології»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

«___» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Пуршака Іллі Андрійовича

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: «Розробка веб сервісу для трекінгу та управління поштовими відправленнями на основі React і NestJS»

Керівник роботи: д.т.н., професор, Андрущак І. Є.

затверджені наказом закладу вищої освіти від «31» грудня 2025 р. №541/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.

3. Вихідні дані до роботи: вимоги до B2B веб-платформи для централізованого управління поштовими відправленнями, документація API поштових операторів Нова Пошта, Укрпошта та Meest, документація фреймворків NestJS і React, документація Prisma ORM та PostgreSQL, довідкові матеріали з проєктування REST API, сучасні підходи до контейнеризації та хмарного розгортання веб-застосунків, методичні вказівки до виконання кваліфікаційної роботи бакалавра

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): обґрунтувати актуальність розробки B2B веб-платформи для управління поштовими відправленнями; проаналізувати існуючі програмні рішення та визначити вимоги до системи; обґрунтувати вибір технологій і спроектувати архітектуру програмного забезпечення та базу даних; реалізувати серверну та клієнтську частини системи й інтеграцію з API поштових операторів; реалізувати механізми захисту даних; провести тестування системи; виконати розгортання програмного забезпечення та налаштувати CI/CD-процес.

5. Перелік графічного матеріалу: 13 рисунків, 8 таблиць, 2 додатки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	<i>Андрущак І. Є.</i>		
Специфікація вимог до розробленої системи	<i>Андрущак І. Є.</i>		
Розробка об'єкта проектування	<i>Андрущак І. Є.</i>		
Нормоконтроль	<i>Суринович О.М.</i>		
Гарант ОП	<i>Ліщина Н. М.</i>		
Показник запозичень тексту	_____ %		
Академічна доброчесність	<i>Андрущак І. Є.</i>		

7. Дата видачі завдання «3» лютого 2026 р.

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедрі	до 29.05.2026 р.	

Здобувач вищої освіти _____

Ілля ПУРШАК

Керівник кваліфікаційної роботи _____

Ігор АНДРУЩАК

АНОТАЦІЯ

Пуршак І. А. Розробка веб сервісу для трекінгу та управління поштовими відправленнями на основі React і NestJS. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності 121 «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається з вступу, трьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі було досліджено ринок поштово-логістичних послуг в Україні. Проведено аналіз API основних поштових операторів: Нової Пошти, Укрпошти та Meest. Результати виявили відсутність на українському ринку мультиоператорної платформи для централізованої роботи з поштовими службами, а закордонні аналоги не підтримують українських перевізників, тим самим підтвердивши актуальність розробки.

У другому розділі сформульовано функціональні та нефункціональні вимоги до системи, на основі яких обрано технологічний стек і спроектовано архітектуру платформи, що охоплює схему бази даних із 26 моделей на базі PostgreSQL і Prisma ORM, рольову модель із трьома рівнями доступу, JWT-автентифікацію з підтримкою двофакторної автентифікації та систему підписок.

У третьому розділі описано практичну реалізацію платформи, а саме агрегацію відправлень різних поштових операторів, мапінг статусів відправлень, клієнтський модуль і панель адміністратора. Висвітлено розгортання системи на AWS з CI/CD-пайплайном через GitHub Actions.

Ключові слова: веб-платформа, агрегатор відправлень, B2B, NestJS, React, PostgreSQL, JWT, AWS, поштові оператори, REST API.

ABSTRACT

Purshak I. A. Development of a web service for tracking and managing postal shipments based on React and NestJS. Manuscript. Qualification work of the bachelor's degree program "Software Engineering", specialty 121 "Software Engineering". Lutsk National Technical University. Lutsk, 2026.

The bachelor's thesis consists of an introduction, three chapters, conclusions, a list of references, and appendices.

The first chapter examines the postal and logistics services market in Ukraine. The APIs of the main postal operators – Nova Poshta, Ukrposhta, and Meest – were analyzed. The findings revealed the absence of a multi-operator platform for centralized postal shipment management on the Ukrainian market, while foreign analogues of this class do not support Ukrainian carriers, thereby confirming the relevance of the development.

The second chapter formulates the functional and non-functional requirements for the system, on the basis of which the technology stack was selected and the platform architecture was designed, encompassing a database schema of 26 models built on PostgreSQL and Prisma ORM, a role-based access model with three permission levels, JWT authentication with two-factor authentication support, and a subscription system.

The third chapter describes the practical implementation of the platform, namely real-time shipment aggregation from multiple postal operators, shipment status mapping, the client module, and the administration panel. The deployment of the system on AWS with a CI/CD pipeline via GitHub Actions is also covered.

Keywords: web platform, shipment aggregator, B2B, NestJS, React, PostgreSQL, JWT, AWS, postal operators, REST API.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1 АНАЛІЗ СУЧАСНОГО СТАНУ УПРАВЛІННЯ ПОШТОВИМИ ВІДПРАВЛЕННЯМИ ТА ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ	9
1.1 Аналіз сучасного стану проблеми	9
1.2 Постанова завдання на кваліфікаційну роботу бакалавра	10
Висновки до розділу 1	11
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЕНОЇ СИСТЕМИ	13
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення	13
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання	17
Висновки до розділу 2	21
РОЗДІЛ 3 РОЗРОБКА В2В ВЕБ-ПЛАТФОРМИ АГРЕГАЦІЇ ПОШТОВИХ ВІДПРАВЛЕНЬ	23
3.1 Практична реалізація об'єкта проектування	23
3.2 Тестування та налагодження інформаційно-комп'ютерної системи.....	33
3.3 Розробка бази даних.....	35
3.4 Розміщення сайту на вибраній платформі хостингу	39
3.5 Захист інформаційно-комп'ютерної системи.....	42
Висновки до розділу 3	48
ВИСНОВКИ.....	50
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	52
ДОДАТКИ.....	54

ВСТУП

Актуальність теми. Стрімкий розвиток електронної комерції в Україні суттєво збільшив навантаження на поштово-логістичну інфраструктуру і загострив операційну проблему бізнесу. Компанії, що відправляють товари через кількох операторів одночасно, змушені вручну зводити дані про відправлення з різних джерел і долати несумісність форматів між системами. Глобальні платформи-агрегатори, що вирішують цю задачу на ринках США та Західної Європи, не підтримують жодного українського оператора. Сукупність цих факторів обумовлює актуальність розробки спеціалізованої B2B-платформи з уніфікованим інтерфейсом управління відправленнями.

Таким чином, метою цієї кваліфікаційної роботи є проектування та розробка B2B веб-платформи-агрегатора «SendIt» для централізованого управління поштовими відправленнями через кількох операторів, що надає організаціям єдиний інтерфейс з використанням власних API-ключів клієнта без необхідності перемикатись між окремими кабінетами перевізників.

Об'єктом роботи є процес розробки B2B веб-платформи для централізованого управління поштовими відправленнями кількох операторів.

Предметом роботи є методи, технології та інструменти створення агрегатора поштових відправлень з використанням сучасного стеку: NestJS, React, PostgreSQL, Prisma ORM.

Для досягнення поставленої мети необхідно реалізувати наступні завдання:

- проаналізувати сучасний стан ринку управління логістикою в Україні, дослідити існуючі рішення-аналоги, визначити їхні переваги та недоліки і обґрунтувати доцільність розробки нової платформи;
- сформулювати функціональні та нефункціональні вимоги до платформи, змодельовати взаємодію акторів та обрати технологічний стек на основі порівняльного аналізу;

- спроектувати трирівневу клієнт-серверну архітектуру із виділеним інтеграційним шаром адаптерів для підключення зовнішніх поштових операторів;
- розробити серверну частину платформи на основі NestJS і TypeScript з використанням Prisma ORM та PostgreSQL, реалізувати REST API з автодокументуванням через Swagger;
- розробити клієнтську частину у вигляді SPA-застосунку на основі React і Vite з TailwindCSS для стилізації інтерфейсу;
- реалізувати систему безпеки платформи: JWT-автентифікацію з ротацією токенів, двофакторну автентифікацію, шифрування API-ключів операторів та рольову модель доступу;
- розробити базу даних платформи та реалізувати систему підписок;
- розгорнути платформу в інфраструктурі AWS і налаштувати CI/CD-пайплайн через GitHub Actions.

Практичне значення роботи полягає в розробці готової до розгортання B2B веб-платформи «SendIt», що дозволяє організаціям централізовано управляти поштовими відправленнями через єдиний інтерфейс замість окремих кабінетів операторів. Реалізована інтеграція з API Нової Пошти, адаптерна архітектура для підключення нових операторів без зміни основної логіки системи, а також механізми безпеки, підписок і CI/CD-розгортання формують повноцінну технічну основу для подальшого розвитку платформи та її комерційного застосування на українському ринку.

Апробація роботи: результати кваліфікаційної роботи бакалавра опубліковані в двох статтях міжнародного наукового журналу (дод. А, Б)

РОЗДІЛ 1

АНАЛІЗ СУЧАСНОГО СТАНУ УПРАВЛІННЯ ПОШТОВИМИ ВІДПРАВЛЕННЯМИ ТА ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Цифровізація кардинально змінила підходи до організації логістики і управління відправленнями. Ключовий чинник цього процесу – масове поширення електронної комерції. За останнє десятиліття обсяг онлайн-продажів у світі зростає у середньому на 15-20 % щорічно, а разом із ним пропорційно збільшувалася кількість поштових відправлень [1]. А в Україні, починаючи з 2020 року, ця тенденція виявилась ще більш вираженою, що призвело до різкого збільшення навантаження на поштову інфраструктуру країни [2].

Нова Пошта, Укрпошта та Meest Express охоплюють переважну частину українського ринку відправлень і кожен із них надає корпоративним клієнтам API для автоматизації роботи. Проте вони все ще різні між собою, що становить проблеми при інтеграції. Так, Нова Пошта повертає HTTP-статус 200 навіть при помилці – перевірити успішність запиту можна лише розібравши тіло відповіді. Стан відправлення «В дорозі» в різних операторів має різні назви, а без додаткового шару нормалізації об'єднати статуси від кількох перевізників в єдиній таблиці фактично неможливо [3].

У процесі дослідження проведено аналіз публічно доступної документації API усіх трьох операторів. За результатами аналізу встановлено, що sandbox-середовище Нової Пошти є відкритим і задокументованим, що дозволило реалізувати повноцінну інтеграцію з її API. Укрпошта та Meest на момент дослідження не надають публічного тестового доступу без укладення договору, тому для цих операторів розроблено адаптери зі змодельованими даними – підхід, що підтверджує коректність обраної архітектури і готовність системи до підключення реальних API цих операторів.

Для компаній, що відправляють товари регулярно і в значних обсягах, це породжує конкретну операційну проблему. Бізнес, що одночасно користується послугами кількох перевізників, змушений підтримувати паралельні процеси створення та відстеження відправлень для кожного оператора і вручну зводити трекінгові дані в цільну картину, а менеджер з обробки замовлень, витрачає час не на роботу з клієнтами, а на перемикання між системами і ручне копіювання даних.

На зрілих ринках таку задачу вирішують спеціалізовані платформи-агрегатори. ShipStation і ShipBob набули широкого поширення в e-commerce-сегменті США та Західної Європи саме тому, що усунули потребу бізнесу окремо працювати з кожним перевізником: обидві платформи надають єдиний інтерфейс для управління відправленнями через десятки операторів одночасно. ShipStation орієнтований на автоматизацію обробки замовлень, тоді як ShipBob поєднує агрегацію перевізників із власною фулфілмент-мережею – різні акценти при однаковій відповіді на той самий операційний запит [4, 5]. Проте жодна з цих платформ не підтримує вітчизняних перевізників.

Таким чином, на українському ринку існує незаповнена ніша. Доступні закордонні агрегатори не підтримують вітчизняних перевізників, а самі оператори не зацікавлені в розробці рішень, що виходять за межі їхньої власної екосистеми. Це обґрунтовує актуальність розробки спеціалізованої платформи, здатної агрегувати відправлення від кількох українських операторів через їхні API, нормалізувати статуси до єдиного формату та надати організаціям єдину точку доступу до всієї поштової логістики.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Метою кваліфікаційної роботи є проєктування та розробка B2B веб-платформи-агрегатора «SendIt» для централізованого управління поштовими відправленнями через кількох операторів. Платформа має надавати організаціям єдиний інтерфейс для роботи з підключеними через API-ключі поштовими

операторами, без необхідності перемикатись між окремими кабінетами.

Для досягнення поставленої мети необхідно реалізувати наступні завдання:

- проаналізувати сучасний стан ринку управління логістикою в Україні, дослідити існуючі рішення-аналоги, визначити їхні переваги та недоліки і обґрунтувати доцільність розробки нової платформи;
- сформулювати функціональні та нефункціональні вимоги до платформи, змодельовати взаємодію акторів та обрати технологічний стек на основі порівняльного аналізу;
- спроектувати трирівневу клієнт-серверну архітектуру із виділеним інтеграційним шаром адаптерів для підключення зовнішніх поштових операторів;
- розробити серверну частину платформи на основі NestJS і TypeScript з використанням Prisma ORM та PostgreSQL, реалізувати REST API з автодокументуванням через Swagger;
- розробити клієнтську частину у вигляді SPA-застосунку на основі React і Vite з TailwindCSS для стилізації інтерфейсу;
- реалізувати систему безпеки платформи: JWT-автентифікацію з ротацією токенів, двофакторну автентифікацію, шифрування API-ключів операторів та рольову модель доступу;
- розробити базу даних платформи та реалізувати систему підписок;
- розгорнути платформу в інфраструктурі AWS і налаштувати CI/CD-пайплайн через GitHub Actions.

Висновки до розділу 1

У першому розділі проведено аналіз сучасного стану ринку управління поштовими відправленнями в Україні. Встановлено, що стрімке зростання електронної комерції суттєво збільшило операційне навантаження на компанії, які працюють із кількома поштовими операторами одночасно. Вони змушені підтримувати паралельні робочі процеси у відокремлених системах кожного

перевізника, оскільки єдиного інструменту для централізованого управління відправленнями на вітчизняному ринку немає.

Встановлено також, що глобальні платформи-агрегатори ShipBob і ShipStation, які вирішують аналогічну задачу на світових ринках, не підтримують жодного українського оператора і практично непридатні для суто українського бізнесу. Отже, на українському ринку зафіксовано незаповнену нішу: відсутність мультиоператорної B2B-платформи для централізованого управління відправленнями.

На основі виявлених проблем у підрозділі 1.2 сформульовано мету кваліфікаційної роботи – проєктування та розробка платформи-агрегатора «SendIt» – і визначено комплекс із восьми завдань, необхідних для її досягнення: від аналізу вимог і вибору технологічного стеку до реалізації системи безпеки, бази даних і розгортання у хмарній інфраструктурі AWS.

Таким чином, результати першого розділу сформулювали теоретичне та аналітичне підґрунтя для переходу до етапу проєктування системи, що розглядається у наступному розділі кваліфікаційної роботи.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЕНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення

Розробка агрегатора відповідає на проблему компаній, що відправляють товари через кількох операторів і змушені вести паралельну роботу в окремих кабінетах і вручну зводити дані. Платформи агрегує відправлення від операторів через їхні API, використовуючи власні ключі клієнта, і дає єдиний інтерфейс для роботи з усіма ними. У системі передбачено три типи акторів: клієнт, адміністратор та суперадміністратор.

Правильний та докладний опис вимог потенційно скорочує кількість переробок у процесі розробки. Основними функціональними вимогами до системи визначено:

- реєстрація та автентифікація користувачів за електронною поштою і паролем із підтримкою двофакторної TOTP-автентифікації
- відновлення пароля через електронну пошту;
- підключення API-ключів поштових операторів із шифруванням на стороні сервера;
- агрегація відправлень від усіх підключених операторів;
- створення відправлень через підключених операторів та збереження шаблонів;
- управління шаблонами типових відправлень та адресною книгою;
- управління тарифними планами;
- модуль внутрішніх сповіщень та тікетна система підтримки;
- онбординговий чеклист для нових користувачів;
- панель адміністратора що включає: управління користувачами, операторами та підписками;

Крім функціональних, до системи сформовано список нефункціональних вимог:

- шифрування API-ключів поштових операторів;
- захист паролів за допомогою алгоритмів хешування;
- захист сесій через короткоживучі JWT access-токени з механізмом ротації refresh-токенів;
- рольова модель доступу до функціоналу платформи;
- стійкість до специфічної поведінки зовнішніх API;
- контейнеризація на основі Docker та хмарне розгортання на інфраструктурі AWS;
- налаштований CI/CD-пайплайн через GitHub Actions;
- захищене з'єднання HTTPS для всіх запитів;
- автодокументування REST API засобами Swagger UI.

Взаємодії між акторами та компонентами системи змодельовано засобами UML. На рисунку 2.1 показано діаграму варіантів використання для клієнта-організації.

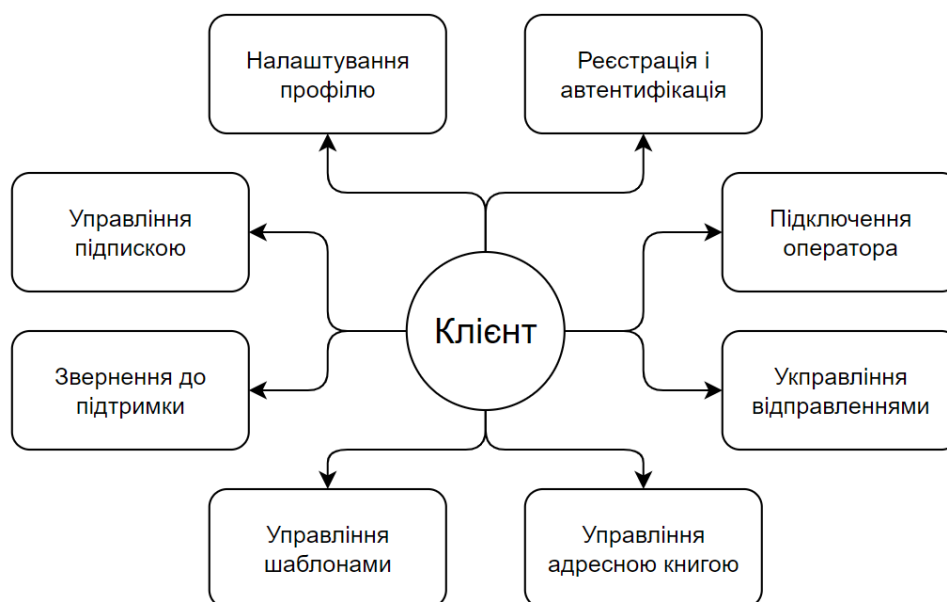


Рисунок 2.1 – Діаграма варіантів використання клієнта

На рисунку 2.2 показано аналогічну діаграму для адміністратора та суперадміністратора з розподілом дій між ролями.



Рисунок 2.2 – Діаграма варіантів використання адміністратора

Окремо варто відзначити архітектурну особливість, що безпосередньо впливає на логіку сценаріїв: відправлення не зберігаються в базі даних платформи, натомість під час кожного запиту система звертається до API оператора з зашифрованим ключем клієнта. Через це успішність відповіді залежить не лише від стану самої платформи, а й від доступності зовнішнього API та актуальності ключа. Такий підхід гарантує актуальність даних без потреби у синхронізації, хоч водночас робить систему залежною від стабільності зовнішніх сервісів.

Логіку роботи платформи зручно розглядати через конкретні сценарії взаємодії. У таблиці 2.1 наведено типові події з боку клієнта й адміністратора та відповідні реакції системи, що відображають практичну поведінку застосунку.

Таблиця 2.1 – Сценарії взаємодії акторів із системою

Актор	Подія	Реакція системи
Клієнт	Організація реєструється на платформі	Створюється обліковий запис і система запитує заповнення профілю компанії
Клієнт	Користувач підключає API-ключ оператора	Ключ шифрується і зберігається; підключенню присвоюється статус ACTIVE

Продовження таблиці 2.1

Актор	Подія	Реакція системи
Клієнт	Користувач відкриває список відправлень	Система паралельно запитує дані від усіх підключених операторів і повертає уніфікований список
Зовнішній API	Оператор повертає помилку автентифікації	Підключенню присвоюється статус INVALID
Зовнішній API	Відправлення отримує нерозпізнаний статус від оператора	Система присвоює normalizedStatus = UNKNOWN
Клієнт	Користувач знижує тарифний план	Зміна відкладається до наступного розрахункового періоду; зайві підключення автоматично блокуються
Клієнт	Користувач підвищує тарифний план	Раніше заблоковані підключення автоматично відновлюються
Система	Розрахунковий період підписки завершується	Cron-задача продовжує підписку або знижує тариф до FREE
Адміністратор	Адміністратор входить у систему	Після перевірки пароля система вимагає підтвердження TOTP-кодом
Супер адміністратор	SUPER_ADMIN запрошує нового адміністратора	Система генерує токен та надсилає посилання-запрошення з терміном дії 7 днів

Наведені сценарії відображають ключові поведінкові особливості платформи: залежність від доступності зовнішніх API, відкладене виконання змін тарифних планів і жорсткий розподіл ролей між клієнтом та адміністративним персоналом. Ці характеристики безпосередньо вплинули на вибір технологічного стеку, що розглядається у наступному підрозділі.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Вибір технологій для платформи-агрегатора відрізняється від вибору для типового CRUD-застосунку. Агрегатор одночасно вирішує кілька важливих задач: зберігає чужі API-ключі в зашифрованому вигляді, паралельно звертається до зовнішніх сервісів під час кожного запиту на список відправлень. Технологічний стек має це покрити без зайвого ускладнення кодової бази.

Коротко розглянуто варіанти BaaS-платформ Firebase і Supabase. Вони виправдані там, де застосунок вписується в стандартні шаблони автентифікації та зберігання даних. Платформа-агрегатор потребує нестандартних рішень – власної системи підписок, шифрування API-ключів операторів і двошарової нормалізації статусів відправлень. BaaS робить таку логіку важко реалізованою або взагалі неможливою в межах платформи. Мікросервісна архітектура цього обмеження не має, але виправдана лише там, де команди налічують десятки розробників, а сервіси масштабуються незалежно. Для цього проєкту це зайву оркестрування і складніше розгортання без видимих переваг. Монолітний бекенд із модульною структурою тримає всю логіку в одній кодовій базі, розгортається одним контейнером і тестується без додаткової інфраструктури.

У межах монолітного підходу для Node.js-екосистеми порівняно Express.js і NestJS. Додатково розглянуто ASP.NET Core як альтернативу поза межами JavaScript-стеку.

Express.js тривалий час залишався стандартним вибором для будь-якого Node.js серверного застосунку. Він надає маршрутизатор і middleware-пайплайн, тоді як структура папок, підхід до залежностей і спосіб валідації залишаються на розсуд розробника. Для невеликого проєкту це перевага. Для платформи-агрегатора з понад десятьма модулями ситуація інша. Відсутність вбудованої ін'єкції залежностей означає необхідність ручного передавання сервісів між шарами, оскільки немає жодного структурного механізму, який гарантував би правильне їх підключення. Без модульної системи межі між модулями умовні,

будь-який з них може імпортувати будь-що звідусіль. Рольова модель реалізується через middleware-функції на кожній групі маршрутів, і правильність їх порядку та повнота покриття залежать виключно від знань та уважності розробника, а не від архітектурних гарантій фреймворку [6].

ASP.NET Core – фреймворк із вбудованою ін'єкцією залежностей, ORM (Entity Framework Core) і готовою системою авторизації [7]. Для проєктів із повним .NET-стеком він природний. Але є принциповий мінус: використовувати C# суто для серверної частини означає підтримувати розробку двома мовами з різними IDE, залежностями і підходами до типізації в одному проєкті.

NestJS є фреймворком для Node.js, що запозичив у Angular декоратори, ін'єкцією залежностей і модульну систему [8]. Він не обмежує свободу, але надає готову структуру, де кожен модуль ізольований і має власні сервіси, контролери і провайдери. Для платформи-агрегатора з понад модулями це принципово важливо. TypeScript-first підхід нативно поєднується з Prisma і Zod. Завдяки підтримці Guards і Interceptors рольова модель доступу може бути реалізована засобами фреймворку без побудови складних ланцюжків middleware. Недоліком такого підходу є підвищений поріг входу і збільшення кількості шаблонного коду в простих сценаріях використання. Узагальнений порівняльний аналіз розглянутих фреймворків наведено у таблиці 2.2.

Таблиця 2.2 – Порівняння backend-фреймворків за критеріями

Критерій	Nest.js	Express.js	ASP.NET Core
Вбудована модульна структура	+	-	+
TypeScript-підтримка	+	±	-
Ін'єкція залежностей	+	-	+
Swagger без конфігурації	+	-	+
Guards / Middleware для авторизації	+	±	+
Єдина мова з фронтом	+	+	-
Швидкість старту	±	+	-
Підтримуваність великих проєктів	+	±	+

Платформа-агрегатор є закритим B2B-застосунком, де всі сторінки доступні лише після авторизації. Пошукові боти не матимуть доступу до контенту, тому SEO і серверний рендеринг тут практично не потрібні. Порівняння обмежено трьома React-рішеннями, що дає змогу оцінити вплив архітектурного підходу фреймворку, а не різниці між бібліотеками.

TanStack Start – повностековий React-фреймворк на базі TanStack Router із підтримкою SSR [9]. На відміну від Next.js із його Suspense-межами, TanStack Start рендерить HTML лише після завершення серверної функції – без проміжного скелетон-стану. Порівняльне дослідження продуктивності (дод. А) підтвердило, що блокувальна стратегія дає нижчий LCP: 67-107 мс проти 97-113 мс у Next.js при 25-100 записах. Однак ця перевага актуальна для відкритих сторінок із SEO-вимогами – для закритої платформи, де всі дані завантажуються після авторизації, вона не має практичного значення. Додатково фреймворк молодший за конкурентів – екосистема готових компонентів значно вужча.

Next.js – фреймворк на базі React із вбудованим SSR, SSG і файловою маршрутизацією [11]. Для публічних сайтів це вагома перевага. Дослідження (дод. А) показало, що FCP у Next.js (50-55 мс) нижчий, ніж у TanStack Start, проте відображає не дані, а loading-оболонку `loading.tsx`: реальний вміст з'являється через 42-63 мс після FCP. Для закритої B2B-платформи без індексації ні нижчий FCP, ні складність Suspense-межі не дають реальної користі – дані все одно завантажуються після авторизації, статичних сторінок для індексації немає. Обидва SSR-фреймворки забезпечують CLS = 0 завдяки серверному формуванню розмітки до її відправки клієнту (дод. А) – перевага, що має практичне значення для публічних сторінок з індексацією, але не для закритої B2B-платформи.

React, Vite – React як бібліотека для UI і Vite як збирач із ESM-підходом [12]. Vite забезпечує швидкий dev-сервер без попередньої компіляції модулів. React має найбільшу екосистему серед JS-фреймворків: TanStack для серверного стану, react-hook-form і Zod для форм, широкий вибір компонентних бібліотек. Дослідження (дод. А) підтверджує цей вибір: Vite CSR забезпечує

найнижчий TTFB серед трьох реалізацій (33-35 мс) без накладних витрат серверного рендерингу. Водночас CSR демонструє CLS = 0.027 через зсув розмітки в момент отримання даних у браузері – компроміс, прийнятний для закритого застосунку без SEO-індексації, де показник CLS не впливає на ранжування в пошукових системах. Для SPA-dashboard без SEO-вимог це оптимальний вибір. Узагальнене порівняння розглянутих фреймворків наведено у таблиці 2.3.

Таблиця 2.3 – Порівняння фреймворків за критеріями

Критерій	Vite, React	Next.js	TanStack Start
Підходить для закритого SPA	+	±	±
SSR/SEO	-	+	+
Швидкість dev-сервера	+	±	+
Розмір екосистеми	+	+	+
TypeScript-підтримка	+	+	+
Простота конфігурації	+	±	±
Потреба в SSR для B2B-dashboard	±	±	±

Розглянуто три рішення, що представляють різні підходи до зберігання та організації даних. MongoDB реалізує документну модель, MySQL і PostgreSQL – реляційну, проте відрізняються рівнем підтримки сучасних інструментів і гнучкістю роботи з типами даних.

MongoDB найкраще розкривається там, де схема нестабільна або дані не мають чітких зв'язків – каталоги, логи, документи з різнорідними атрибутами [13]. Предметна область платформи влаштована інакше. Кожен зв'язок між користувачем, підпискою, планом і підключеннями до операторів закріплений зовнішніми ключами, нативної підтримки яких у MongoDB немає. Цілісність у цьому разі переноситься на рівень застосунку, а Mongoose не компенсує відсутність TypeScript-типізації.

Серед реляційних варіантів MySQL – природний вибір за замовчуванням, підтримується практично всіма хостинг-провайдерами і добре задокументована [14]. За функціональністю вона повністю покриває потреби

проєкту, проте у роботі з JSON-полями і підтримці розширень, на які спирається Prisma, поступається PostgreSQL.

PostgreSQL обирають там, де реляційні зв'язки складні і цілісність критична [15]. Транзакції, зовнішні ключі, JSON-поля і широка екосистема розширень реалізовані безпосередньо на рівні СУБД. Prisma генерує типізований клієнт зі схеми – моделі, поля і зв'язки стають частиною TypeScript-системи типів з автодоповненням. Це усуває цілий клас помилок ще на етапі компіляції.

Додатково, проведений авторський бенчмарк Prisma 7 та нативного pg-драйвера підтвердив, що після переходу Prisma на TypeScript-рушій накладні витрати ORM-абстракції становлять 0.5-1.1 мс залежно від типу операції – різниця статистично достовірна, але практично незначуща для B2B-платформи такого масштабу (дод. Б).

Узагальнене порівняння розглянутих СУБД наведено у таблиці 2.4.

Таблиця 2.4 – Порівняння СУБД за критеріями

Критерій	PostgreSQL	MongoDB	MySQL
Реляційна модель із зовнішніми ключами	+	-	+
Транзакційна цілісність	+	±	+
TypeScript-типізація через Prisma	+	±	+
JSON-поля для гнучких структур	+	+	+
Підтримка складних запитів	+	±	+
Горизонтальне масштабування	±	+	±

Висновки до розділу 2

У другому розділі визначено вимоги та спроектовано основні компоненти платформи.

Сформульовано тринадцять функціональних вимог, що охоплюють автентифікацію організацій із TOTP-захистом, підключення та шифрування API-ключів поштових операторів, агрегацію відправлень у реальному часі,

управління підпискою з трьома тарифними планами та адміністративну панель. Нефункціональні вимоги охоплюють шифрування для API-ключів, хешування для паролів, JWT із ротацією токенів, трирівневу рольову модель і автоматизований CI/CD.

Проведено порівняльний аналіз технологій за трьома категоріями. Для серверної частини порівняно Express.js, ASP.NET Core і NestJS – обрано NestJS завдяки вбудованій модульній системі, ін'єкції залежностей і Guards, що відповідають потребам. Для клієнтської частини порівняно React + Vite, Next.js і TanStack Start – обрано React + Vite як оптимальне SPA-рішення для закритої B2B-платформи без SEO-вимог; вибір підкріплено порівняльним дослідженням продуктивності серверних компонентів, яке показало, що для закритого B2B-застосунку без SEO-вимог накладні витрати серверного рендерингу не дають практичної переваги. Для рівня даних порівняно PostgreSQL, MongoDB і MySQL – обрано PostgreSQL з Prisma ORM завдяки реляційній структурі схеми, типобезпечному TypeScript-клієнту та підтвердженій прийнятності накладних витрат ORM-абстракції для застосунків подібного класу [16].

Обраний стек повністю покриває сформовані вимоги: NestJS забезпечує модульну серверну архітектуру з рольовим захистом, React із Vite – SPA-клієнт без SEO-накладних витрат, PostgreSQL із Prisma ORM – надійне реляційне сховище з точними типами для монетарних значень і TypeScript-типізацією запитів. Конкретні рішення щодо схеми даних, механізмів безпеки та системи підписок описано в розділі 3.

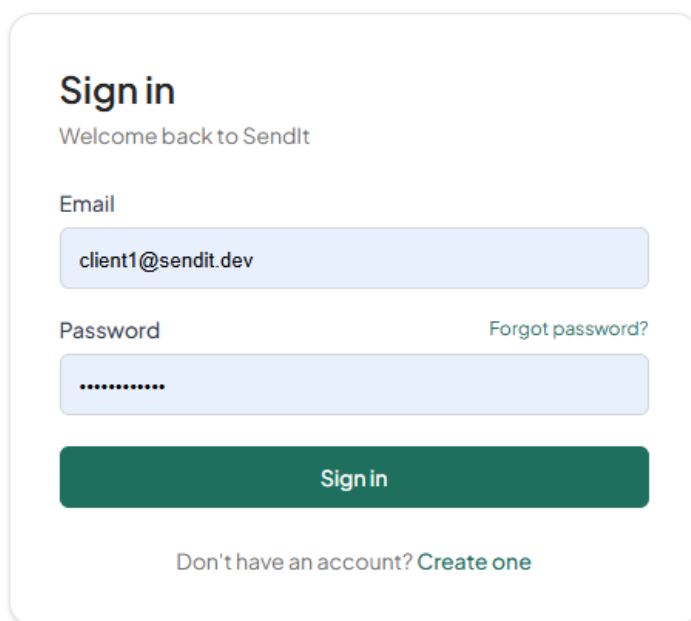
РОЗДІЛ 3

РОЗРОБКА B2B ВЕБ-ПЛАТФОРМИ АГРЕГАЦІЇ ПОШТОВИХ ВІДПРАВЛЕНЬ

3.1 Практична реалізація об'єкта проєктування

Платформу реалізовано у вигляді клієнт-серверного застосунку: бекенд на NestJS 11 з TypeScript, фронтенд на React 18 з Vite, TypeScript і TailwindCSS. На клієнтській стороні TanStack Query відповідає за кешування відповідей API і синхронізацію серверного стану, «react-hook-form» і «zod» – за форми і валідацію. Схеми «zod» на фронтенді повторюють обмеження серверу, тому більшість помилок вводу перехоплюється до відправки запиту. Реалізацію розбито на шість функціональних модулів: автентифікація, підключення операторів, агрегація відправлень, підписки, панель адміністратора і підтримка.

Розробку розпочато з модуля автентифікації – без нього неможливо тестувати жодний захищений маршрут. Клієнт входить через форму з полями email і пароль; помилки валідації відображаються inline під кожним полем через реактивний стан форми (рис. 3.1).



Sign in
Welcome back to SendIt

Email
client1@sendit.dev

Password
.....

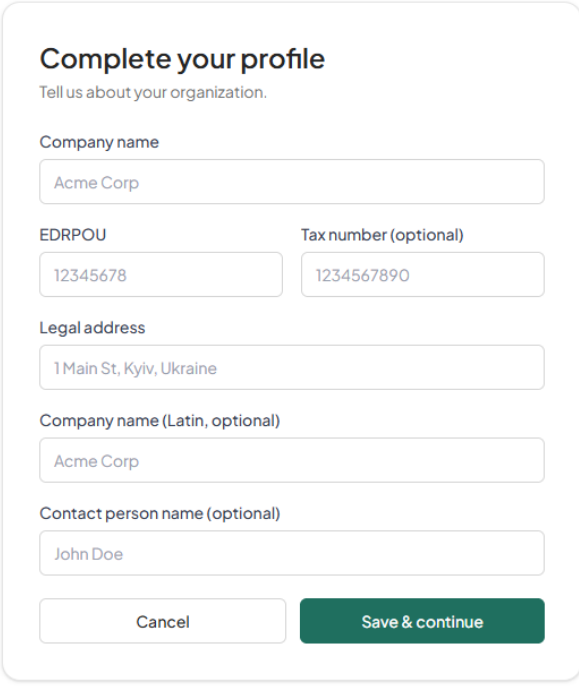
[Forgot password?](#)

Sign in

Don't have an account? [Create one](#)

Рисунок 3.1 – Сторінка входу до платформи

Реєстрація двоетапна: спочатку email і пароль, потім профіль організації – назва компанії, ЄДРПОУ, юридична адреса. Після першого кроку сервер повертає «profileSetupToken», і клієнт перенаправляється на форму профілю (рис. 3.2).



The screenshot shows a web form titled "Complete your profile" with the subtitle "Tell us about your organization." The form contains several input fields: "Company name" (filled with "Acme Corp"), "EDRPOU" (filled with "12345678"), "Tax number (optional)" (filled with "1234567890"), "Legal address" (filled with "1 Main St, Kyiv, Ukraine"), "Company name (Latin, optional)" (filled with "Acme Corp"), and "Contact person name (optional)" (filled with "John Doe"). At the bottom, there are two buttons: "Cancel" and "Save & continue".

Рисунок 3.2 – Сторінка реєстрації організації

При вході сервер перевіряє пароль і видає пару токенів: access-токен і refresh-токен. Відповідь при невірних даних – HTTP 401 без уточнення, чи email не знайдено, чи пароль хибний – щоб не розкривати зловмиснику стан системи. Реалізацію механізмів хешування, зберігання токенів і шифрування детально описано в підрозділі 3.5.

Якщо 2FA увімкнено, сервер після перевірки пароля повертає pendingToken замість access-токена, а фронтенд показує форму введення TOTP-коду (рис. 3.3). Для клієнтів 2FA опціональна. Адміністратор без налаштованої 2FA отримує setup_required-токен і перенаправляється на первинне налаштування – доступу до панелі без завершення цього кроку немає, суперадмін не потребує цього кроку.

Two-factor authentication

Inactive Add an extra layer of security to your account.

Scan this QR code with your authenticator app:



Or enter manually: `VQWJBU4LFFVYA0XTDYWJQKG40UNKJ8GE`

Verification code

Confirm **Cancel**

Рисунок 3.3 – Сторінка введення TOTP-коду

Після проходження автентифікації користувач отримує доступ до сторінки керування підключеними поштовими операторами, де може додавати нові інтеграції, переглядати їх поточний стан та керувати наявними підключеннями. Як зазначено в підрозділі 1.1, повноцінну взаємодію з зовнішнім API реалізовано для Нової Пошти; адаптери Укрпошти та Meest використовують змодельовані дані. Інтерфейс цієї сторінки наведено на рисунку 3.4.

PROFILE WORKSPACE

Postal operators

Use the fixed account menu to switch between profile sections.

Postal operators

Nova Poshta	Connect
Ukrposhta Connected	Edit Disconnect
Meest Express	Connect

Рисунок 3.4 – Сторінка підключення поштового оператора

Нова Пошта відрізняється від інших операторів поведінкою API: сервіс повертає HTTP 200 для всіх відповідей, зокрема й для помилок автентифікації.

Перевірка HTTP-статусу не відрізняє успіх від помилки – потрібно аналізувати тіло відповіді. Поле `success: false` з кодом `AUTH_ERROR` переводить підключення в статус `INVALID` і повертає клієнту HTTP 500 (ліст. 3.1). (згадати що API нової пошти єдине вдалось підключити, а інші штучні)

Лістинг 3.1 – Обробка відповіді Нової Пошти з урахуванням HTTP 200 при помилці авторизації (`nova-post.service.ts`)

```
const response = await firstValueFrom(
  this.httpService.post(NOVA_POST_API_URL, payload),
);
if (!response.data.success) {
  const errorCode = response.data.errorCodes?.[0];
  if (errorCode === 'AUTH_ERROR') {
    await this.postalConnectionsService.markAsInvalid(connectionId);
    throw new UnauthorizedException('Nova Post API key is invalid');
  }
  throw new BadRequestException(
    response.data.errors?.[0] ?? 'Nova Post API error',
  );
}
```

кінець лістингу 3.1

Повноцінна взаємодія з реальним API реалізована виключно для Нової Пошти через відкрите `sandbox`-середовище, описане в підрозділі 1.1. Укрпошта та Meest на момент розробки не надають публічного тестового доступу без договору, тому їхні адаптери повертають змодельовані дані, що підтверджує коректність обраної архітектури та готовність системи до підключення реальних API цих операторів.

Для ізоляції специфіки кожного оператора від решти системи реалізовано патерн адаптера: «`NovaPostShipmentsService`», «`UkrposhtaShipmentsService`», «`MeestShipmentsService`» реалізують спільний інтерфейс, а «`ShipmentReadService`» взаємодіє виключно з ним. Нові оператори підключаються написанням окремого сервісу-адаптера без змін у центральному сервісі (рис. 3.5).

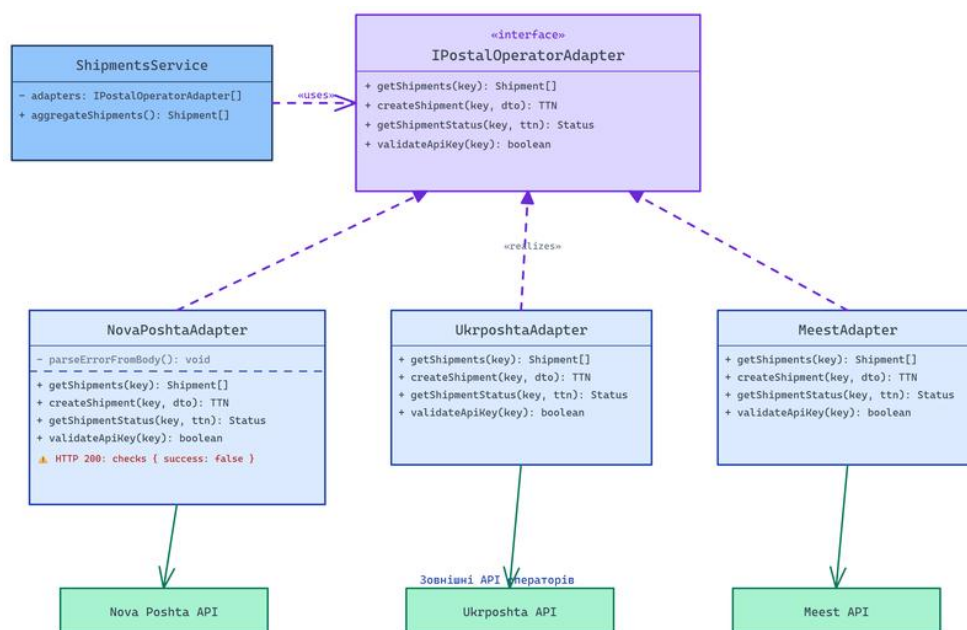


Рисунок 3.5 – UML-діаграма адаптерів поштових операторів

Кожен оператор описує стани відправлень власними термінами – Нова Пошта використовує англійські рядки («ReadyToShip», «Delivered»), детальна історія переміщень – числові коди. Уніфікована фільтрація і відображення без нормалізації неможливі, тому платформа використовує двошаровий підхід. Поле «rawStatus» зберігає оригінальний рядок без змін, «normalizedStatus» – значення внутрішнього enum «ShipmentStatus» після маппінгу через модуль нормалізації. Enum охоплює вісім станів: «DRAFT», «CREATED», «PREPARING», «IN_TRANSIT», «DELIVERED», «CANCELLED», «RETURNED», «UNKNOWN». Відповідність документних статусів Нової Пошти наведено в таблиці 3.1.

Таблиця 3.1 – Маппінг документних статусів Нової Пошти на normalizedStatus

rawStatus	normalizedStatus
Draft	DRAFT
ReadyToShip	CREATED
Accepted	PREPARING
Issued	PREPARING
Delivered	DELIVERED
Returned	RETURNED

Продовження таблиці 3.1

rawStatus	normalizedStatus
Utilized	RETURNED
Deleted	CANCELLED

Числові коди з детальної історії переміщень: коди 100+ переважно мапяться на «IN_TRANSIT», код 2 – на «CANCELLED», коди 102-103, 105-106 і 155 – на «RETURNED». Будь-яке нерозпізнане значення дає «UNKNOWN». На основі «normalizedStatus» модуль нормалізації обчислює три прапорці дій: «canEdit» та «canCancel» – «CREATED», «DRAFT», і «PREPARING», «canDuplicate» – відправлення з будь-яким статусом. Прапорці не зберігаються в базі, а обчислюються під час кожного запиту і передаються клієнту разом з об’єктом відправлення (ліст. 3.2).

Лістинг 3.2 – Модуль нормалізації документних статусів Нової Пошти

```
const novaPostStatusMap: Record<string, ShipmentStatus> = {
  Draft: ShipmentStatus.DRAFT,
  ReadyToShip: ShipmentStatus.CREATED,
  Accepted: ShipmentStatus.PREPARING,
  Issued: ShipmentStatus.PREPARING,
  Delivered: ShipmentStatus.DELIVERED,
  Returned: ShipmentStatus.RETURNED,
  Utilized: ShipmentStatus.RETURNED,
  Deleted: ShipmentStatus.CANCELLED,
};
export function mapNovaPostDocumentStatus(rawStatus: string):
ShipmentStatus {
  return novaPostStatusMap[rawStatus] ?? ShipmentStatus.UNKNOWN;
}
```

кінець лістингу 3.2

При запиті «GET /api/v1/shipments» сервіс отримує всі «ACTIVE»-підключення для поточного користувача і запускає виклики до адаптерів паралельно через «Promise.all». Якщо один із операторів недоступний, опція «suppressMissingConnection: true» повертає порожній масив замість помилки –

часткова відмова оператора не перериває весь запит. Результати об'єднуються з локальними чернетками і сортуються за датою створення за спаданням (ліст. 3.3).

Лістинг 3.3 – Паралельна агрегація відправлень з опцією придушення помилок підключення

```
const activeConnections = await
this.prisma.db.userPostalConnection.findMany({
  where: { userId, status: PostalConnectionStatus.ACTIVE },
});
const results = await Promise.all(
  activeConnections.map((conn) =>
    this.getShipmentsForConnection(conn, {
      suppressMissingConnection: true,
    }),
  ),
);
const shipments = results.flat();
const drafts = await this.shipmentDraftsService.findAllByUser(userId);
return [...drafts, ...shipments].sort(
  (a, b) => b.createdAt.getTime() - a.createdAt.getTime(),
);
```

кінець лістингу 3.3

Фронтенд відображає отриманий масив у таблиці з фільтрацією за нормалізованим статусом, оператором і датою (рис. 3.6). TanStack Query кешує результат і під час завантаження показує скелетон-заглушки, що резервують місце для рядків таблиці без зсуву компонування.

Shipments						
Track and manage all your shipments.						
TTN / ID	OPERATOR	RECIPIENT	STATUS	DATE	VALUE, ₴	ACTIONS
UP000000000000050	Укрпошта	Кравченко Сергій	Preparing	29.05.2026	100	   
ME000000000000050	Meest Express	Woźniak Agnieszka	Preparing	29.05.2026	100	   
UP000000000000049	Укрпошта	Павленко Юлія	Created	28.05.2026	2500	   
ME000000000000049	Meest Express	Szymański Robert	Created	28.05.2026	2500	   
UP000000000000048	Укрпошта	Савченко Олег	Returned	27.05.2026	2400	   
ME000000000000048	Meest Express	Zielińska Katarzyna	Returned	27.05.2026	2400	   

Рисунок 3.6 – Сторінка списку відправлень

Платформа підтримує три тарифні плани, що відрізняються єдиним параметром – «maxOperators»: «FREE» – 1 оператор, «PRO» – кілька операторів і «ULTRA» – необмежена кількість. Зміна плану набирає чинності на початку наступного розрахункового циклу: запит записує в «UserSubscriptionBalance» поля «scheduledSwitchTo» і «scheduledSwitchAt», а поточна підписка залишається активною. Фронтенд відразу показує запланований перехід і дату набрання чинності (рис. 3.7).

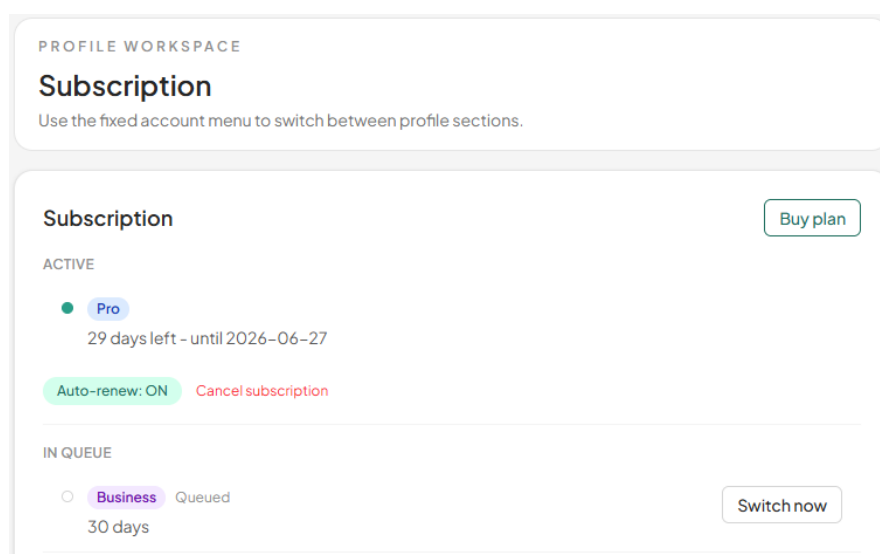


Рисунок 3.7 – Сторінка управління підпискою

При пониженні плану надлишкові підключення переводяться у «BLOCKED»: клієнт вибирає які залишити, або система автоматично зберігає найстаріші за. Щонічні cron-завдання продовжують підписки, активують заплановані переходи і переводять скасовані підписки в «FREE» (ліст. 3.4).

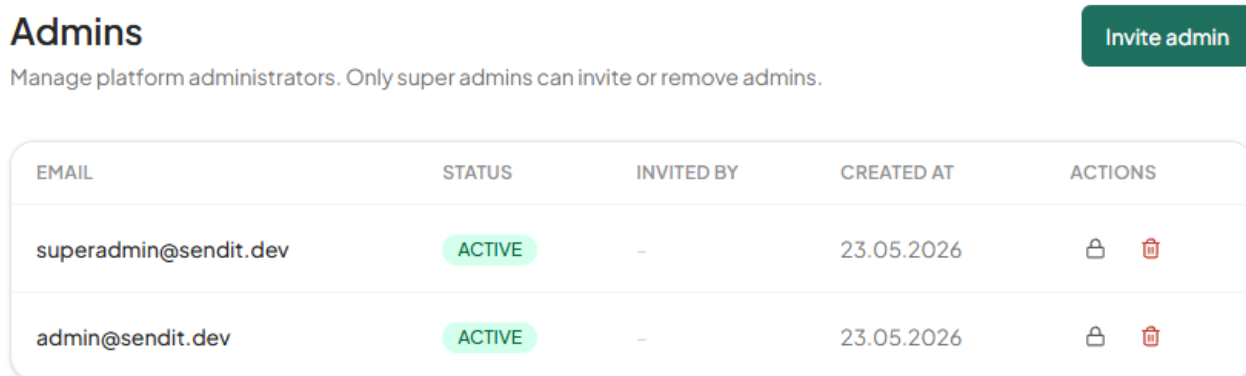
Лістинг 3.4 – Відкладена зміна тарифного плану

```
const switchAt = new Date(Date.now() + getSwitchDelayMs());
await this.prisma.db.userSubscriptionBalance.update({
  where: { id: activeBalance.id },
  data: {
    scheduledSwitchTo: dto.scheduledSwitchTo,
    scheduledSwitchAt: switchAt,
  },
});
```

кінець лістингу 3.4

Адміністратори зберігаються в окремій таблиці «Admin» і не перетинаються з таблицею клієнтів. Самостійна реєстрація користувача з цією роллю неможлива. Для створення нового «Admin», користувач «SUPER_ADMIN» надсилає запрошення на email, система генерує токен і зберігає його SHA-256-хеш у таблиці «AdminInvite» з терміном дії 7 днів. Адміністратор переходить за посиланням, встановлює пароль і налаштовує TOTP. Реалізацію безпечного зберігання токена запрошення описано в підрозділі 3.5.

Доступ до маршрутів управління адміністраторами захищає «SuperAdminGuard», що перевіряє поле «isSuperAdmin» у JWT-payload: звичайний «ADMIN» отримує HTTP 403 (рис. 3.8). Панель охоплює перегляд і м'яке видалення клієнтів, ручне редагування підписок, реєстр операторів, статистику платформи і відповіді на тікети підтримки



EMAIL	STATUS	INVITED BY	CREATED AT	ACTIONS
superadmin@sendit.dev	ACTIVE	-	23.05.2026	 
admin@sendit.dev	ACTIVE	-	23.05.2026	 

Рисунок 3.8 – Панель управління адміністраторами

Система підтримки зберігає листування в таблиці «SupportMessage», прив'язаній до «SupportTicket». Замість маркування кожного повідомлення окремо «TicketReadStatus» зберігає один запис на учасника тікету з полем «lastReadAt». Непрочитаними вважаються повідомлення з «createdAt > lastReadAt», тому таблиця «TicketReadStatus» зберігає фіксовану кількість рядків (по одному на учасника тікету) незалежно від обсягу листування. Такий підхід мінімізує навантаження на базу даних порівняно з підходом індивідуального маркування кожного повідомлення. При першій відповіді адміністратора статус

тікету змінюється з «WAITING» на «IN_PROGRESS» автоматично. Клієнт бачить список своїх тікетів з лічильниками непрочитаних повідомлень і датою останньої активності (рис. 3.9).

Support

All support tickets across the platform, starting with the oldest waiting requests.

Reset filters

Filters 1

Columns

CLIENT	SUBJECT	CATEGORY	STATUS	ASSIGNED TO	CREATED AT	ACTIONS
ТОВ «Тестова Компанія» client@sendit.dev	Ticket about billing	Billing	Waiting	Unassigned	29.05.2026, 11:51:50	Assign to me Open
ТОВ «Тестова Компанія» client@sendit.dev	12323	Question	Waiting	Unassigned	29.05.2026, 11:51:09	Assign to me Open

2 rows

<<

<

1

>

>>

Рисунок 3.9 – Список тікетів підтримки на сторінці клієнта

При відкритті тікету клієнт і адміністратор бачать повний ланцюжок повідомлень у форматі чату. Поле введення нового повідомлення знаходиться на тій самій сторінці (рис. 3.10).

My tickets

Assigned conversations with read state, close flow, and leave-ticket controls.

ActiveClosedAll

ASSIGNED CHATS

ТОВ «Тестова Компанія»
12323
QuestionIn Progress

ТОВ «Тестова Компанія»
Hello
QuestionIn Progress

QuestionIn Progress

12323

ТОВ «Тестова Компанія» - Updated 29.05.2026, 11:53:46

ТОВ «Тестова Компанія»
12312412
29.05.2026, 11:51:09

Reply to client...

Send

Рисунок 3.10 – Інтерфейс тікету підтримки

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Тестування платформи «SendIt» проводилося у чотири етапи: ручне тестування, модульне тестування, аудит продуктивності за допомогою Lighthouse та тестування API через Swagger.

Ручне тестування відтворює користувацький досвід, починаючи з введення даних у форму і до отримання відповіді сервера. Включає 15 сценаріїв, що покривають основні функціональні вимоги системи, а саме: реєстрацію і автентифікацію з можливістю приєднання 2FA, підключення операторів, агрегацію та фільтрацію відправлень, управління підпискою і роботу панелі адміністратора.

Для кожного тест-кейсу зіставлено очікуваний і фактичний результат. Виявлені у процесі дрібні невідповідності поведінки інтерфейсу: відсутність inline-помилки під полями форми входу при хибних даних та відсутність блокера кнопок «submit» у формах у процесі обробки запиту – усунені до завершення тестування. Результати ручного тестування наведено у таблиці в додатку В.

Swagger-тестування передбачало послідовну перевірку кожного ендпоінта з контрольованими вхідними даними та прямим спостереженням відповідей. Окрім функціональної перевірки, Swagger UI слугував одночасно інструментом документування REST API, що дозволило суміщати тестування з формуванням актуальної технічної документації.

Запит «GET /api/v1/shipments» виконувався під обліковим записом із двома активними підключеннями – Нова Пошта та Укрпошта. Swagger повернув об'єднаний масив відправлень із нормалізованими статусами та прапорцями дій для кожного об'єкта.

Для перевірки обробки некоректного API-ключа оператора надіслано «POST /api/v1/postal-connections» із навмисно хибним значенням ключа. Сервер повернув помилку з кодом 500 (рис. 3.11).

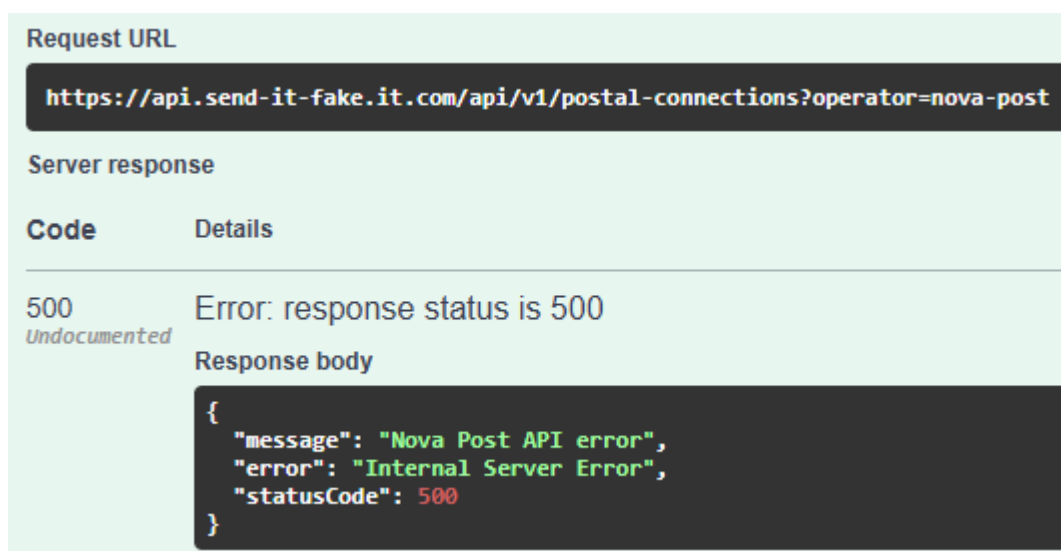


Рисунок 3.11 – Відповідь на запит підключення з некоректним API-ключем у Swagger UI

Юніт-тестування на базі Jest охоплює ізольовані сервіси та утиліти без залежності від бази даних і зовнішніх API. Юніт-тести написано з використанням бібліотеки Jest разом із пакетами Nest.js для тестування. Тестові набори охоплюють модулі: криптографічних утиліт, модуль нормалізації статусів відправлень Нова Пошта, сервісу автентифікації та сервісу підписок. Зовнішні залежності замінені на mock-об'єкти для ізоляції від бази даних і зовнішніх API. Тести перевіряють симетричність шифрування та унікальність шифрів при повторному шифруванні одного і того ж рядка даних. Такий підхід гарантує коректність криптографічної логіки незалежно від середовища виконання. Результати запуску наведено у таблиці в додатку Г.

Продуктивність клієнтської частини виміряно через Lighthouse у десктопному режимі без обмеження мережі та ЦП і вимкненим кешуванням. Перевірка запускала на сторінці шипментів, оскільки це найважча сторінка, після повного завантаження та відображення агрегованого списку відправлень із трьох поштових операторів. Для виключення похибок заміри зібрано тричі. У таблиці 3.2 зафіксовано медіанні значення.

Таблиця 3.2 – Результати Lighthouse-аудиту сторінки шипментів

Метрика	Ітерації			Медіана
	№1	№2	№3	
Performance Score	97	97	98	97
First Contentful Paint (FCP)	913мс	941мс	856мс	913мс
Largest Contentful Paint (LCP)	1118мс	1020мс	948мс	1020мс
Speed Index (SI)	1031мс	1065мс	900мс	1031мс
Total Blocking Time (TBT)	0мс	35мс	17мс	17мс
Cumulative Layout Shift (CLS)	0.0213	0.0212	0.0213	0.0213
Time to First Byte (TTFB)	1118мс	1243мс	1043мс	1118мс

Показник TBT 17 мс свідчить про те, що основний потік залишається практично незаблокованим під час завантаження сторінки. CLS 0,0213 – залишковий зсув після впровадження скелетона таблиці: без явного резервування простору початкове значення сягало 0,08 через зсув компонування при lazy-loading; залишковий зсув перебуває в межах порогу «добре» ($< 0,1$). Загальна оцінка 97 балів підтверджує відповідність платформи вимогам виробничої експлуатації.

3.3 Розробка бази даних

Платформа «SendIt» використовує PostgreSQL як реляційну систему керування базами даних, а Prisma ORM спрощує опис структури бази даних і взаємодію з нею на рівні платформи. Такий підхід робить роботу з даними більш зрозумілою та зручною під час розробки.

Структура бази даних описана у файлі schema.prisma, який містить 26 моделей і 14 enum-типів. Для забезпечення її стабільності та консистентності у різних середовищах використовується Prisma Migrate. Цей інструмент автоматизує процес версіонування схеми бази даних, що суттєво спрощує розробку та підтримку структури бази даних.

Усі чутливі короткоживучі токени зберігаються виключно у вигляді хешів – відкрита версія видається клієнту один раз і до бази даних не потрапляє. Refresh-токени та токени запрошення адміністратора зберігаються як `hashSha256(rawToken)`. Токени для скидання пароля використовують патерн із двома хешами. Поле `tokenLookupHash = hashSha256('lookup:' + raw)` використовується як унікальний індекс, щоб забезпечити швидкий та ефективний пошук записів. Поле `tokenHash = hashSha256('verify:' + raw)` служить для автономної перевірки відповідності значення. Такий поділ унеможливорює отримання активного токена навіть за умови витоку одного поля (ліст. 3.5).

Лістинг 3.5 – Модель `ResetPasswordToken` з двохешовим патерном зберігання

```
model ResetPasswordToken {
  id          Int          @id @default(autoincrement())
  userId      Int
  tokenHash   String
  tokenLookupHash String    @unique
  usedAt      DateTime?
  expiresAt   DateTime
  createdAt   DateTime     @default(now())

  user User @relation(fields: [userId], references: [id], onDelete:
Cascade)}
```

кінець лістингу 3.5

Перед збереженням у базу даних API-ключі операторів і TOTP-секрети шифруються алгоритмом AES-256-CBC. Для кожного шифрування формується окремий вектор ініціалізації, тому навіть однакові ключі щоразу мають різний зашифрований вигляд, що унеможливорює зіставлення записів за збіжністю зашифрованих значень. В базі зберігається єдиний рядок, що містить і IV, і шифротекст. Ключ шифрування задається через змінну оточення і не міститься у вихідному коді.

База даних запобігає створенню дублікатів підключень за допомогою унікального індексу на полях (`userId`, `postalServiceId`), що унеможливорює підключення одного оператора для користувача більше ніж один раз (ліст. 3.6).

Лістинг 3.6 – Модель UserPostalConnection з унікальним обмеженням і статусами

```

model UserPostalConnection {
  id          Int          @id @default(autoincrement())
  userId      Int
  postalServiceId Int
  apiKey      String
  status      PostalConnectionStatus @default(ACTIVE)
  connectedAt DateTime    @default(now())
  updatedAt   DateTime    @updatedAt
  user        User        @relation(fields: [userId], references:
[id], onDelete: Cascade)
  postalService PostalService @relation(fields: [postalServiceId],
references: [id])

  @@unique([userId, postalServiceId])
  @@index([userId])
}
enum PostalConnectionStatus {
  ACTIVE
  BLOCKED
  INVALID
}

```

кінець лістингу 3.6

Кожне підключення може перебувати в одному з трьох станів:

- ACTIVE – активне підключення, яке функціонує належним чином;
- BLOCKED – тип підключення, що встановлюється при переході на нижчий тариф. Ключ із цим статусом не дозволяє взаємодіяти із відповідним оператором, проте ключ залишається збереженим;
- INVALID – тип підключення, що встановлюється, за умови відхилення ключа оператором при API-запиті.

Якщо користувач сам відкликає ключ, то відповідний запис у базі даних видаляється. Система перевіряє актуальність підключення під час його використання.

Підписка клієнта зберігається у моделі UserSubscriptionBalance. Відкладена зміна тарифного плану реалізована через два поля: scheduledSwitchTo (nullable Int – ідентифікатор цільового плану) та scheduledSwitchAt (nullable DateTime – дата активації переходу). До настання цієї

дати поточна підписка залишається активною; перехід виконується нічним cron-завданням планувальника. Поле `customAmount` дозволяє адміністратору встановити персональну ціну підписки без зміни базового плану. Enum `DiscountType` розрізняє одноразову знижку (`ONE_TIME`, що автоматично очищується після першого продовження циклу) і постійну (`PERMANENT`). Усі грошові поля схеми – `SubscriptionPlan.price`, `BillingHistory.amount`, `UkrposhtaShipment.declaredValue` та аналогічні – використовують тип `Decimal` `@db.Decimal(10, 2)`, що відповідає PostgreSQL-типу `NUMERIC(10,2)` з точним десятковим поданням замість `Float` (ліст. 3.7).

Лістинг 3.7 – Ключові поля моделі `UserSubscriptionBalance`

```

model UserSubscriptionBalance {
  id                Int                @id
  @default(autoincrement())
  userId            Int
  planId            Int
  periodType        SubscriptionPeriodType @default(MONTHLY)
  daysTotal         Int
  periodEnd         DateTime?
  status            SubscriptionBalanceStatus @default(ACTIVE)
  autoRenew         Boolean             @default(true)
  scheduledSwitchTo Int?
  scheduledSwitchAt DateTime?
  customAmount      Decimal?            @db.Decimal(10, 2)
  discountType      DiscountType?
  createdAt         DateTime             @default(now())
  updatedAt         DateTime             @updatedAt

  @@index([userId, status])
}

```

кінець лістингу 3.7

Для моделей із гнучкою або еволюційною структурою даних застосовуються поля типу `Json`. `ShipmentDraft.draftData` та `ShipmentTemplate.templateData` зберігають специфічні для оператора параметри відправлення у вигляді JSON-блобів: набір і формат полів суттєво різняться між Нова Пошта, Укрпоштою та Meest, тому фіксована реляційна схема призвела б або до великої кількості `NULL`-полів, або до окремих таблиць на кожного

оператора. `Recipient.address` також зберігається як `Json` з тієї ж причини – структура адреси залежить від оператора доставки.

Для відстеження прочитаних повідомлень у тикетах підтримки обрано патерн часового курсора замість маркування кожного повідомлення окремо. Модель `TicketReadStatus` зберігає один запис на пару (тікет, сторона) з полем `lastReadAt`. Непрочитаними вважаються повідомлення, для яких виконується умова `createdAt > lastReadAt`. Такий підхід мінімізує кількість рядків у таблиці – $O(1)$ записів на учасника тикету замість $O(n)$ – і дозволяє підраховувати непрочитані повідомлення одним запитом із фільтром по часу без сканування всіх рядків `SupportMessage`.

Схема бази даних містить набір складених індексів, оптимізованих під найпоширеніші сценарії роботи системи. Вони використовуються для прискорення пошуку активних сесій користувачів, вибірки підписок, обробки тикетів підтримки та отримання непрочитаних сповіщень. Окремі індекси також забезпечують ефективну фільтрацію тикетів за адміністратором, статусом і датою створення, що особливо важливо для пагінованого перегляду великих обсягів даних.

Механізм м'якого видалення реалізований через спеціальні поля стану та дати запланованого видалення. Такий підхід дозволяє зберігати дані протягом визначеного `grace-period` і спрощує відновлення акаунтів у разі потреби. Для більшості пов'язаних сутностей використовується каскадне видалення, що допомагає автоматично очищувати залежні записи та підтримувати цілісність даних.

3.4 Розміщення сайту на вибраній платформі хостингу

Для розгортання та забезпечення роботи серверної частини застосунку було використано низку хмарних та інфраструктурних сервісів. Основні компоненти доцільно розглядати відповідно до шляху, який проходить користувацький запит під час взаємодії із системою.

Спочатку запит надходить на відповідний субдомен, «app.send-it-fake.it.com» або «api.send-it-fake.it.com», потім спрямовується до AWS Application Load Balancer, де виконується завершення SSL-з'єднання. Після чого передається далі на EC2-інстанс, де Nginx маршрутизує його до відповідного контейнера на порту 3000 до API або до фронту на 3001.

Окрім сервісів, які безпосередньо обробляють запити, у проєкті також є Namecheap – сервіс реєстрації доменного імені, керування субдоменами і налаштування DNS-записів.

Бекенд, як наведено в лістингу 3.12, розгортається у контейнері за принципом multi-stage білда:

- builder – встановлює dev залежності, генерує Prisma-клієнт і компілює TypeScript у JavaScript;
- runner – заново встановлює залежності, цього разу всі, і запускає скомпільований код з першого етапу

На EC2-інстансі сервіси платформи керуються через docker-compose.prod.yml. У виробничому середовищі розгорнуто два контейнери: backend-сервіс api та база даних PostgreSQL. Конфігурація backend-контейнера визначається через файл .env – секрети та змінні середовища у такий спосіб не потрапляють ні до кодової бази, ні до Compose-файлу. Дані PostgreSQL зберігаються у іменованому Docker-томі postgres_data, що унеможлиблює їх втрату при перезапуску контейнерів (ліст. 3.8).

Лістинг 3.8 – Файл docker-compose.prod.yml для виробничого середовища

```
services:
  postgres:
    image: postgres:16-alpine
    restart: unless-stopped
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    volumes:
      -postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ['CMD-SHELL', 'pg_isready -U ${POSTGRES_USER}']
```

```

    interval: 5s
    timeout: 5s
    retries: 10

api:
  build: .
  restart: unless-stopped
  ports:
    - '3000:3000'
  env_file: .env
  depends_on:
    postgres:
      condition: service_healthy

volumes:
  postgres_data:

```

кінець лістингу 3.8

CI/CD-пайплайн бекенду описано у файлі `.github/workflows/deploy.yml`; він запускається автоматично після кожного `push` у гілку `main`. Розгортання – автоматичне: пайплайн підключається по `SSH`, отримує актуальну версію репозиторію командою `git pull`, перезбирає контейнери та видаляє застарілі `Docker`-образи. Для підключення використовуються три `GitHub Secrets`: `EC2_HOST`, `EC2_USER` та `EC2_SSH_KEY`.

Фронтенд-контейнер описаний окремим `docker-compose.yml`. Сервіс `sendit` збирається локально на сервері при кожному деплої та доступний на порті `3001`. `API_BASE_URL` передається як `build-argument` під час збірки образу. `Nginx` на рівні хоста проксіює зовнішні запити до цього порту (ліст. 3.9).

Лістинг 3.9 – Файл конфігурації `Nginx`

```

server {
    listen 80;
    server_name api.send-it-fake.it.com;

    location / {
        proxy_pass http://127.0.0.1:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}

```

```
server {
    listen 80;
    server_name app.send-it-fake.it.com;

    location / {
        proxy_pass http://127.0.0.1:3001;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

кінець лістингу 3.9

CI/CD-пайплайн фронтенду складається з двох послідовних кроків:

- `ci` – встановлює залежності, запускає лінтер і виконує Vite-збірку з `API_BASE_URL` зі `secrets`, перевіряючи коректність збірки до деплою;
- `deploy` – запускається лише за умови успішного `ci`. Під час виконання підключається до сервера взявши ключ із GitHub Secrets, тоді стягує актуальну версію репозиторія, зупиняє контейнери, ребілдить і запускає їх заново.

Міграції бази даних не автоматизовані в CI/CD пайплайні, оскільки зміна схеми під навантаженням може призвести до нестабільної роботи сервісу. Тому в разі змін в схемі бази даних треба вручну застосувати міграцію перед деплоєм. Секрети та змінні середовища зберігаються окремо від репозиторію у файлі `.env` за шляхом `/home/ubuntu/send-it-api` і оновлюються вручну.

3.5 Захист інформаційно-комп'ютерної системи

Платформа «SendIt» оперує трьома категоріями чутливих даних: облікові дані користувачів (паролі), токени сеансів і запрошень, а також API-ключі та TOTP-секрети поштових операторів. Для кожної категорії застосовано окремий криптографічний підхід відповідно до природи даних: паролі необоротно хешуються, токени зберігаються як односпрямовані хеші, а ключі і секрети – оборотно шифруються, оскільки сервер повинен мати змогу їх відновити для звернення до зовнішніх API.

Хешування паролів. Паролі користувачів обробляються алгоритмом `bcrypt` із коефіцієнтом складності 12 раундів. `Bcrypt` є адаптивною хеш-функцією на

основі шифру Blowfish: кожне підвищення коефіцієнта складності подвоює обчислювальне навантаження, що дозволяє підтримувати стійкість до підбору зі зростанням обчислювальних потужностей. При 12 раундах час хешування становить приблизно 250-400 мс на сучасному сервері – достатньо для захисту від атаки перебором, але прийнятно для користувача. Хеш разом із вбудованою сіллю зберігається в окремій таблиці UserCredentials; відкрита версія пароля не з'являється ні в базі даних, ні в логах.

При вході bcrypt.compare порівнює введений пароль із хешем: функція витягує сіль із наявного хешу, повторно хешує введений рядок і порівнює результат. Відповідь при невірних даних повертається як HTTP 401 без уточнення, яке саме поле неправильне, – щоб не розкривати зломиснику інформацію про стан системи (ліст. 3.10).

Лістинг 3.10 – Хешування пароля при реєстрації та верифікація при вході

```

async register(dto: RegisterDto) {
  const existing = await this.prisma.db.user.findUnique({
    where: { email: dto.email },
  });
  if (existing) throw new ConflictException('Email already in use');
  const passwordHash = await bcrypt.hash(dto.password, BCRYPT_ROUNDS);
  const user = await this.prisma.db.user.create({ data: { email:
dto.email } });
  await this.prisma.db.userCredentials.create({
    data: { userId: user.id, passwordHash },
  });
  const profileSetupToken = this.issueProfileSetupToken(user.id);
  return { requiresProfileCompletion: true, profileSetupToken };
}
async login(dto: LoginDto) {
  const user = await this.prisma.db.user.findUnique({
    where: { email: dto.email },
    include: { credentials: true, twoFactorAuth: true },
  });
  if (!user || !user.credentials)
    throw new UnauthorizedException('Invalid credentials');
  const passwordValid = await bcrypt.compare(
    dto.password,
    user.credentials.passwordHash,
  );
  if (!passwordValid) throw new UnauthorizedException('Invalid
credentials');

```

```

    if (user.twoFactorAuth?.isEnabled) {
      const pendingToken = this.issuePendingToken(user.id);
      return { requires2FA: true, pendingToken };
    }
    return { requires2FA: false, ...await this.issueTokenPair(user) };
  }

```

кінець лістингу 3.10

Керування сеансами через JWT. Після успішної автентифікації сервер видає два токени: короткоживучий access-токен (15 хвилин) для авторизації API-запитів і refresh-токен (7 днів) для відновлення сеансу без повторного введення пароля. Refresh-токен зберігається у таблиці RefreshToken виключно у вигляді SHA-256-хешу: клієнт отримує відкрите значення один раз, і воно ніколи не потрапляє до бази даних. Відкликання токена реалізовано через виставлення поля revokedAt без видалення запису – це дозволяє зберігати аудит виданих токенів і запобігати повторному використанню відкликаних значень.

Для підтримки багатокрокової автентифікації система використовує додаткові типи токенів із коротким терміном дії: pendingToken – для очікування ТОТР-коду після перевірки пароля, setup_required – для нового адміністратора, що ще не налаштував 2FA, profileSetupToken – для нового клієнта, що ще не заповнив профіль організації. Кожен тип підписується окремим секретом з оточення і має обмежений набір дозволених маршрутів, що унеможлиблює його використання поза передбаченим сценарієм.

Двофакторна автентифікація. Для захисту адміністративних облікових записів реалізовано ТОТР-двофакторну автентифікацію на основі бібліотеки otplib 13.x відповідно до стандарту RFC 6238. ТОТР-код обчислюється як HMAC-SHA1 від поточного часового кроку (30 секунд) і спільного секрету: $\text{TOTP} = \text{HOTP}(\text{secret}, [t / 30])$, де t – поточний час в секундах. Це забезпечує одноразовість кожного коду без необхідності синхронізації стану між сервером і застосунком-автентифікатором.

ТОТР-секрет генерується сервером, шифрується алгоритмом AES-256-CBC і зберігається в таблиці TwoFactorAuth. QR-код для первинного

налаштування автентифікатора генерується бібліотекою qrcode у форматі otpauth://totp/... і передається клієнту у вигляді Data URL – секрет у відкритому вигляді покидає сервер лише один раз, під час налаштування (ліст. 3.11).

Лістинг 3.11 – Генерація TOTP-секрету та QR-коду для підключення автентифікатора

```

async setup2fa(user: JwtUser): Promise<{ qrCodeUrl: string; secret:
string }> {
  const secret = generateSecret();
  const encryptedSecret = encryptTotp(secret);
  await this.prisma.db.twoFactorAuth.upsert({
    where: { userId: user.id },
    create: { userId: user.id, secret: encryptedSecret },
    update: { secret: encryptedSecret, isEnabled: false },
  });
  const otpAuthUrl = generateURI({
    issuer: 'SendIt',
    label: user.email ?? «user-${user.id}»,
    secret,
  });
  const qrCodeUrl = await QRCode.toDataURL(otpAuthUrl);
  return { qrCodeUrl, secret };
}

```

кінець лістингу 3.11

Шифрування API-ключів і TOTP-секретів. API-ключі поштових операторів і TOTP-секрети є оборотно зашифрованими даними: сервер повинен мати змогу відновити їх у відкритому вигляді для звернення до зовнішніх API та верифікації TOTP-кодів відповідно. Для цього застосовано алгоритм AES-256-CBC (Advanced Encryption Standard у режимі Cipher Block Chaining з 256-бітним ключем). CBC-режим шифрує кожен 16-байтовий блок відкритого тексту за формулою 3.1:

$$C_i = \{E_k\}(P_i \oplus C_{i-1}), \quad (3.1)$$

де P_i – блок відкритого тексту,

C_i – блок шифротексту,

E_k – функція шифрування AES з ключем k ,

$C_0 = IV$.

Вектор ініціалізації (IV) генерується криптографічно стійким генератором псевдовипадкових чисел для кожного виклику функції шифрування незалежно. Завдяки цьому однаковий відкритий текст при повторному шифруванні дає різний шифротекст, що унеможливорює зіставлення записів у базі даних за збігом зашифрованих значень.

IV і шифротекст зберігаються разом у полі бази даних у форматі `<ivHex>:<encryptedHex>`. Ключ шифрування задається через змінну оточення `TOTP_ENCRYPTION_KEY` (64 шістнадцяткових символи = 32 байти) і не міститься у вихідному коді (ліст. 3.12).

Лістинг 3.12 – Шифрування та дешифрування чутливих значень

```
export function encryptValue(value: string, key: string): string {
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv(
    'aes-256-cbc',
    Buffer.from(key, 'hex'),
    iv,
  );
  const encrypted = Buffer.concat([
    cipher.update(value, 'utf8'),
    cipher.final(),
  ]);
  return `${iv.toString('hex')}:${encrypted.toString('hex')}`;
}
export function decryptValue(stored: string, key: string): string {
  const [ivHex, encryptedHex] = stored.split(':');
  const decipher = crypto.createDecipheriv(
    'aes-256-cbc',
    Buffer.from(key, 'hex'),
    Buffer.from(ivHex, 'hex'),
  );
  const decrypted = Buffer.concat([
    decipher.update(Buffer.from(encryptedHex, 'hex')),
    decipher.final(),
  ]);
  return decrypted.toString('utf8');
}
```

кінець лістингу 3.12

Безпечне зберігання токенів запрошення. Токен запрошення адміністратора є одноразовим доказом того, що `SUPER_ADMIN` видав дозвіл на реєстрацію конкретного email. Сервер генерує криптографічно стійкий токен і

зберігає у таблиці AdminInvite виключно його SHA-256-хеш; відкрите значення включається до посилання-запрошення і більше не зберігається. Для скидання пароля застосовано двохешовий патерн: `tokenLookupHash = SHA-256('lookup:' + raw)` використовується як унікальний індекс для ефективного пошуку запису, а `tokenHash = SHA-256('verify:' + raw)` – для незалежної перевірки відповідності токена. Поділ на два хеші унеможливорює компрометацію активного токена навіть у разі витоку одного з полів (ліст. 3.13).

Лістинг 3.13 – Генерація токена запрошення адміністратора

```

async inviteAdmin(actor: AdminJwtUser, dto: AdminInviteAdminDto) {
  const newAdmin = await this.prisma.db.admin.create({
    data: {
      email: dto.email,
      isSuperAdmin: false,
      status: 'PENDING',
      invitedById: actor.id,
    },
  });
  const rawToken = generateToken();
  const tokenHash = hashSha256(rawToken);
  const expiresAt = new Date(Date.now() + INVITE_TTL_DAYS * 24 * 60 *
60 * 1000);
  await this.prisma.db.adminInvite.create({
    data: { adminId: newAdmin.id, token: tokenHash, invitedById:
actor.id, expiresAt },
  });
  return { adminId: newAdmin.id, email: newAdmin.email, inviteToken:
rawToken, expiresAt };
}

```

кінець лістингу 3.13

Рольова модель і авторизація. Система розмежування доступу реалізована через механізм Guards фреймворку NestJS. Глобально застосований JwtAuthGuard перехоплює кожен вхідний запит, верифікує підпис JWT і встановлює об'єкт поточного користувача у контексті запиту. Маршрути, що потребують перевірки ролі, додатково захищаються спеціалізованими guards: AdminJwtAuthGuard обмежує доступ до маршрутів адміністративної панелі власниками адмін-токенів, а SuperAdminGuard перевіряє поле isSuperAdmin у

JWT-payload і повертає HTTP 403 для звичайного ADMIN. Відкриті маршрути (реєстрація, вхід) позначаються декоратором `@Public()`, що обходить глобальний guard. Такий декларативний підхід гарантує, що права доступу перевіряються архітектурою фреймворку, а не покладаються на дисципліну розробника.

Усі запити між клієнтом і сервером передаються виключно по захищеному з'єднанню HTTPS із завершенням TLS на рівні AWS Application Load Balancer. Nginx на рівні хосту маршрутизує трафік між контейнерами вже по HTTP у межах локальної мережі EC2-інстанса, недоступної ззовні. Таким чином, чутливі дані – включно з паролями і токенами у заголовках запитів – ніколи не передаються у відкритому вигляді через публічну мережу.

Висновки до розділу 3

У третьому розділі описано практичну реалізацію платформи «SendIt»: зведено повний технічний стек, спроектовано схему бази даних, налаштовано виробниче розгортання та проведено тестування готової системи.

Серверну частину реалізовано на NestJS 11, фронтенд – на React 18 з Vite. Шість функціональних модулів охоплюють автентифікацію, підключення операторів, агрегацію відправлень, підписки, адміністративну панель і підтримку. Для захисту облікових даних застосовано диференційований підхід залежно від природи даних: паролі хешуються bcrypt, токени зберігаються як SHA-256-хеш, а API-ключі операторів шифруються AES-256-CBC з унікальним вектором ініціалізації на кожне шифрування, що унеможливорює зіставлення записів за збіжністю зашифрованих значень. Агрегація відправлень виконується паралельно з відмовостійкістю до недоступності окремого оператора, а зміни тарифів відкладаються до початку наступного розрахункового циклу.

Схему бази даних із 26 моделей і 14 enum-типів спроектовано на PostgreSQL; керування версіями схеми виконується через Prisma Migrate. Клієнтська та адмінська ієрархії таблиць ізольовані одне від одного, щоб унеможливити зміну ролей на рівні бази даних. Грошові значення зберігаються

типом `Decimal(10,2)`, гнучкі параметри відправлень – JSON-полями; стратегічні складені індекси оптимізують найчастіші запити системи.

Розгортання автоматизовано через GitHub Actions із доставкою на AWS. Docker-образ backend-у збирається двоетапно – до фінального образу потрапляють лише скомпільований код і виробничі залежності, що суттєво зменшує його розмір. Інфраструктура включає EC2 із Docker, AWS Application Load Balancer для балансування та маршрутизації трафіку.

Тестування проведено у чотири етапи: усі 15 ручних тест-кейсів пройдено успішно, Усі 157 юніт-тестів пройдено успішно за 30,7 секунд, Lighthouse-аудит дашборда показав Performance Score дев'яносто сім із ста при FCP 913 мс та CLS 0,0213. Два дефекти, виявлені в процесі тестування, усунені до завершення розробки.

Таким чином, у третьому розділі реалізовано повнофункціональну платформу «SendIt» із захищеною автентифікацією, агрегацією даних від кількох операторів і гнучким управлінням підписками, та підтверджено її працездатність у межах визначених вимог.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи досягнуто поставленої мети – спроектовано та розроблено B2B веб-платформу «SendIt» для централізованого управління поштовими відправленнями через єдиний інтерфейс із використанням API-ключів клієнта.

Для досягнення поставленої мети було виконано всі визначені завдання. Проаналізовано сучасний стан ринку управління логістикою в Україні, досліджено існуючі рішення-аналоги, визначено їхні переваги та недоліки і обґрунтовано доцільність розробки нової платформи.

Сформульовано функціональні та нефункціональні вимоги до платформи, змодельовано взаємодію акторів та обрано технологічний стек на основі порівняльного аналізу.

Спроектовано трирівневу клієнт-серверну архітектуру із виділеним інтеграційним шаром адаптерів для підключення зовнішніх поштових операторів. У межах реалізації інтеграційного функціоналу виконано повноцінне підключення API Нової Пошти та розроблено тестові адаптери для Укрпошти і Meest, що підтвердило працездатність обраного архітектурного підходу та можливість подальшого розширення системи.

Розроблено серверну частину платформи на основі NestJS і TypeScript з використанням Prisma ORM та PostgreSQL, реалізовано REST API з автодокументуванням через Swagger. Розроблено клієнтську частину у вигляді SPA-застосунку на основі React і Vite з TailwindCSS для стилізації інтерфейсу.

Реалізовано систему безпеки платформи: JWT-автентифікацію з ротацією токенів, двофакторну автентифікацію, шифрування API-ключів операторів та рольову модель доступу.

Реалізовані, але не деталізовані в тексті з огляду на обсяг пояснювальної записки, допоміжні компоненти системи – аналітичний дашборд, адресна книга отримувачів, шаблони відправлень та онбординговий чеклист – забезпечують повноту виконання всіх тринадцяти визначених функціональних вимог.

Розроблено базу даних платформи та реалізовано систему підписок. Розгорнуто платформу в інфраструктурі AWS та налаштовано CI/CD-пайплайн через GitHub Actions.

Працездатність розробленої платформи підтверджено результатами комплексного тестування. Проведені ручні перевірки, модульне тестування, тестування API та аудит продуктивності засвідчили коректність роботи системи й відповідність реалізованого функціоналу визначеним вимогам.

Таким чином, усі завдання кваліфікаційної роботи були успішно виконані, а розроблена платформа «SendIt» може використовуватися як основа для централізованого управління поштовими відправленнями та подальшого розширення інтеграції з іншими поштовими операторами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Global ecommerce sales growth (2022-2028) [updated oct 2024]. Oberlo. URL: <https://www.oberlo.com/statistics/global-ecommerce-sales-growth> (дата звернення: 28.02.2026).
2. Ukraine ecommerce trends: market insights & forecast. Promodo Performance Marketing Agency. URL: <https://www.promodo.com/blog/research-of-the-ukrainian-ecommerce-market> (дата звернення: 25.03.2026).
3. How to properly operate an online store with delivery services: Nova Poshta, Ukrposhta, Meest, CityHost. URL: <https://cityhost.ua/en/blog/obzor-sluzhbdostavki-novaya-pochta-ukrpochta-meest-justin.html> (дата звернення: 25.03.2026).
4. ShipStation. Про продукт, інтеграцію. URL: <https://hellip.com/ua/product/shipstation.html> (дата звернення: 26.03.2026).
5. ShipBob Review 2026: 3PL Fulfillment Built for Shopify DTC Brands. ATTN Agency. URL: <https://www.attnagency.com/blog/shipbob-shopify-review> (дата звернення: 28.03.2026).
6. Express.js · Node.js web application framework. Express.js. URL: <https://expressjs.com/en/> (дата звернення: 10.04.2026).
7. ASP.NET documentation. Microsoft Learn: Build with answers in reach. URL: <https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-10.0> (дата звернення: 14.04.2026).
8. Documentation. NestJS – A progressive Node.js framework. URL: <https://docs.nestjs.com/> (дата звернення: 15.04.2026).
9. TanStack Start. TanStack The open-source application stack for the web. URL: <https://tanstack.com/start/latest> (дата звернення: 17.04.2026).
10. Google Web Dev. (2024). Largest Contentful Paint (LCP). web.dev. URL: <https://web.dev/articles/lcp> (дата звернення: 20.04.2026).
11. Vercel. Rendering: Server-side Rendering (SSR) Next.js. Next.js by Vercel – The React Framework. URL: <https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering> (дата звернення: 24.04.2026).

12. React. React. URL: <https://react.dev/> (дата звернення: 25.04.2026).
13. What is MongoDB? – Database Manual –MongoDB Docs. MongoDB: The World's Leading Modern Data Platform MongoDB. URL: <https://www.mongodb.com/docs/manual/> (дата звернення: 26.04.2026).
14. MySQL – MySQL 8.4 Reference Manual. fw_error_www. URL: <https://dev.mysql.com/doc/refman/8.4/en/> (дата звернення: 27.04.2026).
15. PostgreSQL 18.4 Documentation. PostgreSQL Documentation. URL: <https://www.postgresql.org/docs/current/> (дата звернення: 01.05.2026).

ДОДАТКИ

Додаток А

Наукова стаття «Порівняльна характеристика підходів до рендерингу в React: серверні компоненти (RSC) у TanStack Start і Next.js та клієнтський рендеринг (CSR)»

International Science Journal of Engineering & Agriculture

2022; X(X): XX-XX

<https://isg-journal.com/XXXXXX/>

doi: 10.46299/j.xxx.xxxxxxxx.xx

ISSN: XXXX-XXXX



Інформаційні технології

Порівняльна характеристика підходів до рендерингу в React: серверні компоненти (RSC) у TanStack Start і Next.js та клієнтський рендеринг (CSR).

Ігор Андрушак

Луцький національний технічний університет, Луцьк, Україна

ORCID 0000-0002-8751-4420

Електронна адреса: 9000@lntu.edu.ua

Ілля Пуршак

Луцький національний технічний університет, Луцьк, Україна

ORCID 0009-0007-0407-2625

Електронна адреса: purshak.illa@gmail.com

Анотація: вибір стратегії рендерингу у React-застосунках безпосередньо визначає швидкість відображення контенту та якість користувацького досвіду. Серверні компоненти React (RSC), стабілізовані у React 19, реалізовані у двох фреймворках із принципово різною архітектурою: Next.js App Router надсилає HTML-оболонку одразу та передає вміст через Suspense-межу потоком, тоді як TanStack Start блокує рендеринг маршруту до завершення route loader-а і відправляє повністю сформований HTML за один крок. На сьогодні відсутні рецензовані порівняльні дослідження продуктивності цих фреймворків за метриками Core Web Vitals, що ускладнює обґрунтований вибір при розробці. У дослідженні проведено контрольоване тестування TanStack Start і Next.js App Router; Vite + React (CSR) використано як контрольну реалізацію для виявлення CLS-компромісу клієнтського рендерингу. Тестування проводилось на ізольованому середовищі AWS EC2; набір метрик сформовано з Google Lighthouse: LCP, FCP, TBT, CLS, Speed Index та TTFB. Виміри виконувались при чотирьох обсягах табличних даних – 25, 50, 75 та 100 записів – в умовах «теплої» бази даних PostgreSQL. За результатами дослідження TanStack Start демонструє стабільно нижчі значення LCP (67–107 мс проти 97–113 мс у Next.js), TTFB (44–49 мс проти 77–120 мс) та Speed Index (114–144 мс проти 291–330 мс) на всіх рівнях навантаження. Обидва SSR-фреймворки забезпечують нульові значення CLS, тоді як Vite CSR демонструє CLS = 0.027 через ререндер при підвантаженні даних. Встановлено, що агрегований бал Lighthouse Performance стоїть із стабілізацією в обох SSR-фреймворках і не відображає відмінностей у стратегії рендерингу. Отримані результати формують практичну основу для обґрунтованого вибору фреймворку рендерингу під час розробки React-застосунків із вимогами до продуктивності за Core Web Vitals.

Ключові слова: серверні компоненти React, TanStack Start, Next.js, клієнтський рендеринг, продуктивність вебзастосунків, Lighthouse, Largest Contentful Paint, Time to First Byte, бенчмарк.

1. Вступ

React веб-застосунки зазнали суттєвих змін за час свого існування. Перехід від традиційних сторінок із серверним рендерингом до односторінкових застосунків (SPA) призвів до приросту рівня інтерактивності, але спричинив проблеми надмірно великих JavaScript-бандлів, збільшення часу до першого байту та погіршення показників Core Web Vitals на пристроях з обмеженими апаратними ресурсами. Як часткове рішення було введено

серверний рендеринг (SSR), що згодом еволюціонував у серверні компоненти (RSC). Цей підхід передбачає рендеринг компонентів на сервері та їх передавання клієнту у вигляді серіалізованого віртуального DOM, що позбавляє необхідності відправляти їхній JavaScript-код у браузер.

Упродовж кількох років Next.js залишається де-факто стандартним фреймворком для розгортання RSC. App Router, зробив RSC моделлю рендерингу за замовчуванням і запропонував комплексне рішення для маршрутизації, кешування та отримання даних. Проте такі проблеми, як залежність фреймворку від інфраструктури Vercel, обмеження щодо того, де можна використовувати серверні та клієнтські компоненти та загальна складність розуміння, викликана власним runtime-шаром для RSC, стали об'єктом постійної критики. Це сформувало запит на альтернативний підхід до розробки на React. Побудований на основі TanStack Router та Vite, TanStack Start розглядає RSC як додаткову можливість, а не основний підхід, віддаючи перевагу явним налаштуванням над усталеними правилами та забезпечуючи легший runtime з меншою кількістю шарів абстракції [1]. Ця архітектурна філософія формує відмінні показники продуктивності, які на сьогодні не досліджувались в академічній літературі.

Розробники, які приймають рішення щодо вибору фреймворку, наразі змушені покладатися на публікації, досвід спільноти або маркетингові матеріали. Про цю ж проблему згадує офіційна документація TanStack та прямо застерігає, що будь-які порівняння продуктивності без чіткої методології позбавлені наукової цінності [2]. Саме це і покликає вирішити дане дослідження.

Для встановлення базової лінії порівняння до дослідження включено третю реалізацію – Vite + React (CSR), яка дозволяє кількісно виявити CLS-компроміс клієнтського рендерингу та відокремити затримки, характерні для SPA-архітектури, від затримок серверного рендерингу.

Для оцінювання продуктивності використовувався стандартизований набір метрик від Google – Core Web Vitals. Головною метрикою для цього дослідження є LCP – час появи найбільшого видимого елемента сторінки. Також враховано дані CLS та TBT. Додаткові метрики – FCP, SI та TTFB – доповнюють зібраний набір даних, охоплюючи різні етапи завантаження сторінки [3]. Дослідження зосереджено навколо вибору між підходами до серверних компонентів від фреймворків TanStack Start і Next.js. Проблема полягає в тому, що цей вибір зазвичай робиться без реальних вимірювань.

Структура статті складається з 8 розділів, включаючи вступ. Розділ 2 описує об'єкт та предмет дослідження. У розділі 3 сформульовано мету та задачі. Розділ 4 містить огляд літератури щодо продуктивності RSC та порівняння фреймворків. У розділі 5 описується експериментальна методика. Розділ 6 представляє результати досліджень. У розділі 7 здійснюється аналіз та інтерпретація отриманих даних. Розділ 8 містить висновки та визначає напрями подальших досліджень.

2. Об'єкт і предмет дослідження

Об'єкт дослідження – продуктивність React-застосунків при різних стратегіях рендерингу, а саме: як вибір між серверним і клієнтським рендерингом та конкретна реалізація фреймворку впливають на час появи контенту на сторінках із табличними даними. Від рішення рендерити компоненти на сервері чи на клієнті безпосередньо залежить, як швидко користувач побачить вміст. Серверні компоненти React (RSC) виконуються виключно на сервері й передаються браузеру як серіалізований результат, а не JavaScript-код. Браузер не запускає їхній код – бандл менший, а момент появи контенту залежить від швидкості сервера, а не клієнта.

Від того, як саме фреймворк реалізує RSC-конвеєр, залежить момент, коли браузер отримує перший байт, коли з'являється будь-який вміст і коли відображається основний контент. Ці відношення між TTFB, FCP та LCP не однакові навіть для двох фреймворків, що обидва реалізують RSC: різні підходи до блокування відповіді, потокової передачі Suspense-

меж (точок розбиття HTML-відповіді на потокові фрагменти) дають різні числа за тих самих умов. Один лише показник Lighthouse Performance стоїть із ста нічого не говорить про те, скільки часу насправді проходить до появи даних – тому важливо вимірювати не загальну оцінку, а конкретні метрики.

Тестовий сценарій – однакова таблиця користувачів із живого бекенд-маршруту при чотирьох обсягах: 25, 50, 75 і 100 записів; усі три реалізації – TanStack Start, Next.js App Router та Vite + React (CSR) – звертаються до спільного бекенду та відображають ідентичну таблицю. PostgreSQL працювала в «теплого» стані – записи вже в буферному кеші, дискових затримок немає.

Предметом дослідження є відмінності у метриках Core Web Vitals між трьома реалізаціями рендерингу: TanStack Start і Next.js App Router – двома підходами до серверних компонентів React (RSC) з принципово різною архітектурою формування HTTP-відповіді – та Vite + React як контрольною CSR-реалізацією, що встановлює базову лінію клієнтського рендерингу.

Предмет включає дослідження того, як архітектурне рішення – чекати завершення route loader-а перед рендерингом маршруту, де createServerFn викликається з loader-а (TanStack Start), чи надсилати HTML-оболонку одразу з подальшою потоковою передачею вмісту через Suspense-межу (Next.js App Router) – впливає на момент появи першого байту, першого контенту та найбільшого видимого елемента сторінки. Предмет також включає дослідження того, як співвідношення між FCP і LCP змінюється залежно від обраної стратегії рендерингу та як обсяг переданих даних впливає на динаміку зростання LCP у кожному з фреймворків. Окремо досліджується, як клієнтська стратегія Vite + React – де дані з'являються лише після завершення fetch у браузері – формує відмінний патерн CLS і TTFB порівняно з обома SSR-реалізаціями.

У межах дослідження також розглядається необхідність формалізації критеріїв вибору стратегії рендерингу для React-застосунків із вимогами до Core Web Vitals – включно з питанням, за яких умов CSR-підхід залишається конкурентоспроможним попри CLS-компроміс. Для таких систем важливо визначити, яке архітектурне рішення забезпечує нижчий LCP при різних обсягах даних, як інтерпретувати розбіжність між FCP і LCP як індикатор обраного підходу до рендерингу, а також за яких умов перевага одного фреймворку нівелюється зі зростанням навантаження.

Таким чином, предмет дослідження охоплює як теоретичні аспекти – архітектурне пояснення причинно-наслідкових зв'язків між стратегією рендерингу та числовими показниками метрик – так і практичні: числове порівняння вимірювань, отриманих у контрольованому середовищі, та формулювання рекомендацій щодо вибору стратегії рендерингу та фреймворку залежно від пріоритетних критеріїв продуктивності.

3. Мета та задачі дослідження

Метою дослідження є порівняльний аналіз метрик Core Web Vitals у TanStack Start і Next.js App Router при відображенні серверних компонентів із реальними табличними даними. Центральне питання – чому два фреймворки, що реалізують RSC, дають різні патерни LCP, FCP і TTFB за ідентичних умов: у реалізації TanStack Start createServerFn викликається з route loader-а, і router не рендерить маршрут до завершення loader-а, тоді як Next.js App Router відправляє HTML-оболонку одразу й передає вміст через Suspense-межу потоком. Вимірювання спираються на відтворювану методологію, яка ізолює вплив стратегії рендерингу від зовнішніх апаратних і мережових чинників.

У межах роботи також аналізується поведінка Vite CSR як контрольної реалізації, що дозволяє відокремити затримки клієнтського рендерингу від серверних і відстежити, як зміна обсягу даних (25, 50, 75 і 100 записів) впливає на зростання LCP у кожному фреймворку. Окремо розглядається CLS: єдина метрика, де CSR-підхід показує ненульове значення – індикатор стрибка розмітки при клієнтській гідратації.

Досягнення мети передбачає вирішення комплексу взаємопов'язаних завдань:

✓ **реалізація тестових застосунків:**

розробити ідентичний серверний компонент відображення табличних даних у трьох реалізаціях: TanStack Start через createServerFn, Next.js App Router через асинхронний Server Component із Suspense-межею та Vite + React як CSR-контроль; забезпечити однакову структуру запиту до спільного NestJS-бекенду з PostgreSQL-базою для всіх трьох реалізацій; унеможливити розбіжності у логіці відображення, щоб різниця метрик відображала виключно стратегію рендерингу;

✓ **підготовка контрольованого тестового середовища:**

розгорнути всі три застосунки та спільний бекенд на єдиному інстансі AWS EC2 t3.micro (Ubuntu 22.04 LTS) для виключення мережових затримок між сервісами; підготувати базу даних у «теплову» стані – всі тестові записи заздалегідь у буферному кеші PostgreSQL – для ізоляції вимірювань від затримок дискового введення-виведення; зафіксувати чотири обсяги вибірок: 25, 50, 75 і 100 записів;

✓ **автоматизоване вимірювання метрик:**

реалізувати конвеєр збору метрик через Google Lighthouse у режимі headless Chrome (Puppeteer) у конфігурації desktop без симуляції мережевого або CPU-throttling; виконати три незалежних вимірювання на кожну конфігурацію «фреймворк × обсяг»; підсумковим значенням обрати середнє трьох запусків; зафіксувати шість показників – LCP, FCP, TBT, CLS, Speed Index та TTFB;

✓ **порівняльний аналіз результатів:**

порівняти числові значення метрик між TanStack Start, Next.js і Vite CSR при усіх чотирьох обсягах даних; встановити, при яких обсягах і для яких метрик різниця між фреймворками є практично значущою; визначити, як співвідношення FCP і LCP змінюється залежно від стратегії рендерингу і чи нівелюється перевага одного фреймворку зі зростанням обсягу даних;

✓ **формування рекомендацій:**

на основі отриманих даних розробити рекомендації щодо вибору стратегії рендерингу – між TanStack Start, Next.js та CSR-підходом – для React-застосунків із вимогами до Core Web Vitals; визначити сценарії, в яких блокувальна стратегія TanStack Start забезпечує нижчий LCP, та сценарії, де потоковий підхід Next.js дає перевагу за FCP; окреслити межі узагальнення результатів з огляду на умови тестового стенду.

Загалом поставлені завдання охоплюють як архітектурний аналіз відмінностей між фреймворками, так і їх емпіричну верифікацію в числових метриках, що дозволяє сформулювати практичні висновки з опорою на відтворювані вимірювання, а не на документальні твердження виробників.

4. Аналіз літератури

Дослідження підходів до рендерингу у React-застосунках є порівняно молодим напрямом, який активно розвивається паралельно зі змінами у специфікаціях самого фреймворку. Базовий поділ між клієнтським і серверним рендерингом зафіксований у роботі Iskandar et al. [4], де шляхом вимірювань за метриками Lighthouse – зокрема Speed Index, First Contentful Paint і Time to Interactive – встановлено, що серверний рендеринг забезпечує вищу продуктивність початкового відображення сторінки порівняно з клієнтським підходом. Ця робота стала методологічним підґрунтям для подальших порівняльних досліджень і регулярно цитується у суміжних публікаціях. Аналогічну методику використали Nordström та Dixelius [5], які порівняли SSR і CSR засобами Google Lighthouse та Chrome DevTools Performance Insights і дійшли висновку, що на сторінках із невеликою кількістю елементів вибір стратегії рендерингу не має суттєвого впливу на швидкість завантаження. Цей висновок є важливим контекстом для поточного дослідження: він припускає, що значущі відмінності між

фреймворками можуть виявлятися лише за певних умов навантаження – саме тому у цій роботі тестування проводилося при чотирьох рівнях обсягу табличних даних.

Наближення серверного рендерингу до специфіки Next.js здійснено у роботі Prabowo et al. [6], де побудовано ідентичні версії застосунків на React і Next.js та виміряно їхню поведінку за метриками FCP і Time to Interactive в умовах різної якості мережевого з'єднання. Автори підтвердили перевагу Next.js за показниками продуктивності початкового завантаження, а також продемонстрували, що CSR-підхід систематично поступається SSR за умов обмеженої пропускної здатності. Водночас ця стаття залишає відкритим питання порівняння декількох SSR-реалізацій між собою, оскільки зосереджена виключно на протиставленні CSR та одного SSR-фреймворку.

Архітектурна основа серверних компонентів React задокументована командою React у серії публікацій у офіційному блозі [7, 1]. У них описано еволюцію моделі рендерингу від класичного SSR до RSC: серверні компоненти виконуються виключно на сервері й передаються клієнту у вигляді серіалізованого віртуального DOM, що усуває необхідність надсилати їхній JavaScript-код у браузер. Стабільна реалізація RSC увійшла до складу React 19, випущеного у грудні 2024 року [1]. Відповідно до офіційної документації, Next.js App Router є першою публічною реалізацією RSC-специфікації, де маршрутизація, кешування та отримання даних побудовані навколо серверних компонентів як примітива [2]. TanStack Start, своєю чергою, позиціонується як альтернативний підхід: побудований на основі TanStack Router та Vite, він розглядає RSC як факультативну можливість і надає перевагу явним налаштуванням над усталеними правилами [8, 3].

Ключовим архітектурним розходженням між двома фреймворками є спосіб формування HTTP-відповіді. У Next.js App Router реалізовано потокову передачу через Suspense-межу: сервер надсилає HTML-оболонку одразу, а компоненти, що очікують на дані, транслюються в потоці у міру їхньої готовності. Механізм вибіркової гідратації (selective hydration), введений у React 18, дозволяє розпочинати гідратацію окремих частин сторінки незалежно одна від одної [9, 10]. Imoh [11] і LogRocket [12] детально описують цю архітектуру та вказують, що Suspense розбиває застосунок на HTML-фрагменти за ознакою залежності від даних, що дозволяє клієнту отримувати та відображати контент поступово. Saeloun [13] розглядає внутрішній механізм нової SSR-архітектури React 18 та наголошує, що поєднання потокової передачі й вибіркової гідратації усуває необхідність чекати завершення отримання всіх даних перед відправленням HTML клієнту. TanStack Start застосовує інший принцип: router не рендерить маршрут до завершення route loader-a, а createServerFn виконується на сервері та повертає вже готові дані [14, 15]. Це означає, що клієнт отримує повністю сформований HTML за один крок, без проміжних завантажувальних станів.

Щодо метрик оцінювання, дослідження спирається на систему показників Google Lighthouse та Core Web Vitals. Офіційна документація Google визначає Largest Contentful Paint як міру часу від початку завантаження сторінки до відображення найбільшого видимого елемента у viewport, що є найбільш репрезентативним показником моменту появи основного контенту [16, 17]. First Contentful Paint фіксує час появи першого будь-якого видимого елемента і, відповідно до документації web.dev, слугує раннім індикатором швидкості завантаження, але не є показником корисного контенту [17]. Time to First Byte є первинним драйвером LCP: дослідження показують, що сторінки з поганим LCP мають середній TTFB 2 270 мс, і саме TTFB необхідно оптимізувати в першу чергу [18]. Наявне практичне порівняння TanStack Start, React Router та Next.js, проведене Platformatic [19], підтверджує, що при однаковому SSR-навантаженні 1 000 запитів на секунду фреймворки демонструють різні характеристики часу відповіді. Ця публікація є єдиним відомим авторам публічним бенчмарком, що включає TanStack Start, однак її методологія відрізняється від поточного дослідження: вона вимірює серверну пропускну здатність, а не клієнтські метрики Lighthouse.

Таким чином, аналіз наявних досліджень виявляє системну прогалину: жодна з рецензованих робіт не порівнює метрики Core Web Vitals між TanStack Start і Next.js App

Router у контрольованих умовах. Публікації або зосереджуються на протиставленні SSR та CSR загалом [1, 2], або порівнюють Next.js з React без SSR [3], або описують архітектуру окремого фреймворку [7, 8, 13]. Пропонована робота заповнює цю прогалину, вимірюючи шість метрик Lighthouse при чотирьох рівнях навантаження та надаючи архітектурне пояснення причинно-наслідкових зв'язків між стратегією рендерингу й числовими результатами.

5. Методи досліджень

Методологія дослідження поєднує три складові: архітектурний аналіз відмінностей між фреймворками у підходах до формування HTTP-відповіді, кількісне вимірювання метрик у контрольованому середовищі та статистичне порівняння результатів за формулою відносного відхилення. Числа без архітектурного пояснення не відповідають на питання «чому», а пояснення без чисел не доводять. Саме це поєднання дозволяє пов'язати технічне рішення фреймворку зі спостережуваним результатом у браузері. Загальну структуру тестового стенду наведено на рис. 1.

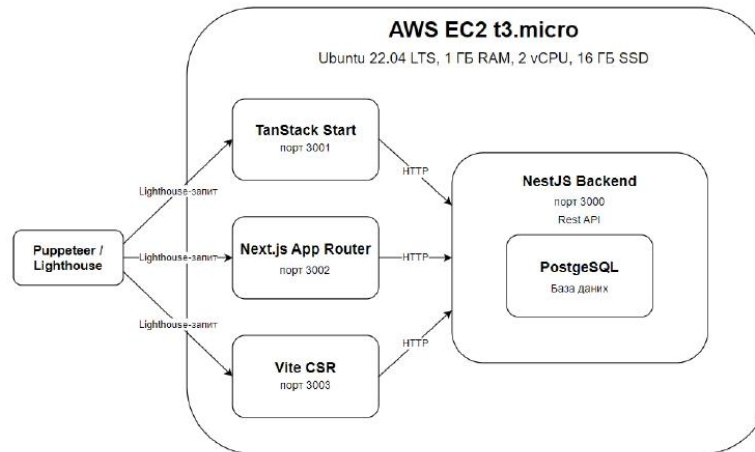


Рис. 1. Структура тестового стенду

Шість зібраних метрик не є довільним набором – кожна фіксує конкретну точку в конверсії рендерингу. TTFB вимірює час обробки запиту на сервері до першого байту відповіді. FCP – перший намальований контент: у Next.js це може бути оболонка `loading.tsx`, у TanStack Start – одразу таблиця з даними. LCP фіксує появу найбільшого видимого елемента, в нашому сценарії – таблиці. Розрив між FCP і LCP є архітектурним індикатором: нульовий розрив означає блокувальний підхід (TanStack Start), ненульовий – потоковий (Next.js). Speed Index інтегрує швидкість заповнення viewport і чутливий до Suspense-скелетонів. TBT і CLS фіксують стабільність розмітки – характерну проблему CSR-підходу. Загальна оцінка Lighthouse Performance стоїть із ста в усіх трьох реалізацій підтверджує, що вона не здатна розрізняти стратегії рендерингу – саме тому аналіз спирається на окремі метрики.

Підсумкове значення кожної метрики – середнє трьох вимірювань за формулою (1):

$$M = \left(\frac{1}{3}\right) \sum_{i=1}^3 x_i,$$

де x_i – значення i -го вимірювання метрики для конфігурації «фреймворк × обсяг»;

M – підсумкове значення, що використовується для порівняння.

Для кількісного порівняння між фреймворками застосовується формула відносного відхилення (2):

$$\Delta = \frac{M_A - M_B}{M_B} \times 100\%,$$

де M_A , M_B – середні значення однойменної метрики фреймворків А та В відповідно.

Ця формула обґрунтовує відсоткові твердження в §7 (наприклад, «31 % при 25 записях»).

Тестовий стенд розгорнуто на інстансі AWS EC2 t3.micro під управлінням Ubuntu 22.04 LTS: 2 vCPU, 1 ГБ RAM, 16 ГБ дискового простору. Обмеження у 1 ГБ RAM впливає на абсолютні значення метрик, але не на відносне порівняння між фреймворками – всі реалізації тестувалися за ідентичних умов. Розміщення всіх сервісів на одній машині виключає зовнішні мережеві затримки між фронтендом і бекендом.

Конфігурацію всіх трьох тестових реалізацій наведено в таблиці 1.

Таблиця 1. Конфігурація тестових реалізацій

Фреймворк	Версія	React	Механізм отримання даних
TanStack Start	1.168.6	19.2.4	createServerFn всередині route loader
Next.js App Router	16.2.6	19.2.6	Async Server Component + app/loading.tsx
Vite + React (CSR)	8.0.13	19.2.6	useEffect + fetch на клієнті

Тестувалося чотири фіксовані обсяги вибірок: 25, 50, 75 і 100 записів. Усі три фронтенди звертаються до спільного маршруту GET /api/v1/admin/users/test?pageSize=N та відображають ідентичну таблицю – відмінності в метриках зумовлені виключно стратегією рендерингу. База підтримувалася в «теплого» стані (warm state): перед вимірюваннями всі тестові записи вже перебували в буферному кеші PostgreSQL. Це виключає дискові затримки холодного старту й дозволяє виміряти вплив рендерингу, а не I/O-варіативність.

Google Lighthouse запускався через Puppeteer у режимі headless Chrome, конфігурація desktop, без симуляції мережевого або CPU-throttling – щоб ізолювати серверні затримки від апаратних. Усього 36 вимірювань: 3 фреймворки × 4 обсяги × 3 повторення. Підсумкове значення – середнє за формулою (1). Зібрані показники: LCP, FCP, TBT, CLS, Speed Index, TTFB. Конвеєр автоматизованого вимірювання метрик проілюстровано на рис. 2.



Рис. 2. Конвеєр автоматизованого вимірювання метрик

Описана методологія ізолює досліджувану змінну від апаратних і мережевих чинників: одна машина, ідентичні дані, «теплий» стан бази, відсутність throttling. Числові відмінності між фреймворками з §6 відображають виключно архітектурні рішення щодо формування HTTP-відповіді. Поєднання теоретично обґрунтованого набору метрик, контрольованого стенду і відтворюваного конвеєра вимірювань закладає основу для рекомендацій у §8.

6. Результати досліджень

Для порівняльного оцінювання TanStack Start і Next.js App Router виконано 36 вимірювань Lighthouse – по три повторення для кожної конфігурації «фреймворк × обсяг даних» при трьох

реалізаціях і чотирьох рівнях навантаження. Тестові обсяги: 25, 50, 75 і 100 записів із живого NestJS/PostgreSQL бекенду. Підсумкове значення – середнє трьох запусків за формулою (1) з §5.

Для кожної конфігурації зафіксовано:

- LCP – час появи найбільшого видимого елемента (у цьому сценарії – таблиця з даними); основна метрика корисного вмісту;
- FCP – перший намальований контент: у Next.js це може бути оболонка loading.tsx, а не власне дані;
- розрив FCP–LCP – різниця між двома попередніми; 0 = блокувальна стратегія, ненуль = потокова;
- Speed Index – інтегральна швидкість заповнення viewport; чутливий до скелетонів і анімацій завантаження;
- TTFB – час до першого байту; відображає затримку серверної обробки;
- TBT – час блокування головного потоку після завантаження.

Загальна оцінка Lighthouse Performance сто із ста збіглася в обох фреймворках – агрегований бал не розрізняє стратегій рендерингу. Окремі метрики дають іншу картину. На рисунку 3 показано динаміку LCP при зміні обсягу даних: лінії TanStack Start і Next.js зростають лінійно, але з різним нахилом – перевага TanStack Start за LCP звужується зі збільшенням навантаження. Найбільш відмінним результатом виявився Speed Index Next.js: у 2.3–2.5 рази вище, ніж у TanStack Start.

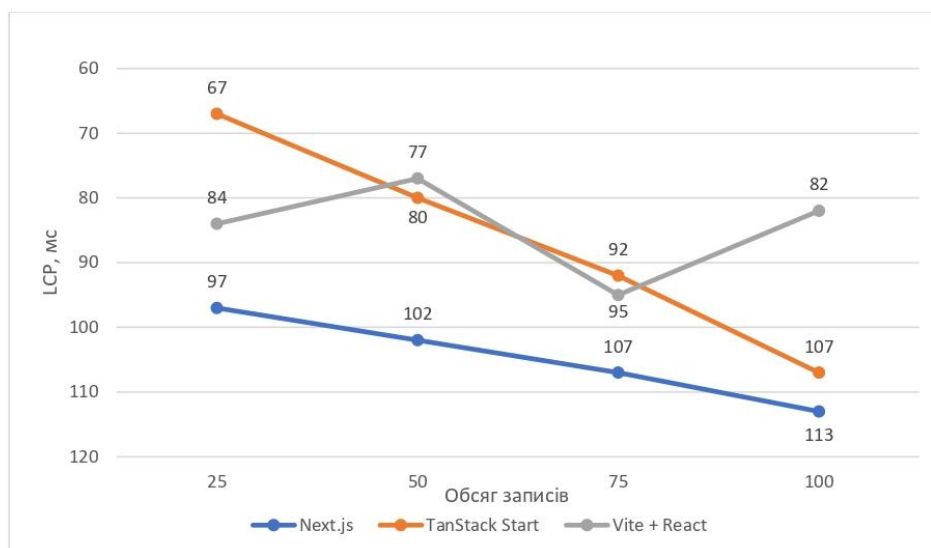


Рис. 3. LCP залежно від обсягу даних

Результати вимірювань усіх шести метрик для трьох реалізацій наведено в таблиці 2.

Таблиця 2. Показники продуктивності фреймворків за метриками Lighthouse

Фреймворк	Записів	LCP, мс	FCP, мс	TBT, мс	CLS	SI, мс	TTFB, мс
TanStack Start	25	67	67	0	0,000	114	45
	50	80	80	0	0,000	121	44
	75	92	92	0	0,000	134	48
	100	107	107	0	0,000	144	49
Next.js	25	97	55	0	0,000	291	77
	50	102	50	0	0,000	303	79
	75	107	50	0	0,000	330	120
	100	113	50	2	0,000	317	87
Vite + React	25	84	41	0	0,027	147	35
	50	77	35	0	0,027	132	34
	75	95	54	0	0,027	138	33
	100	82	36	0	0,027	152	33

Таблиця 2 показує такі закономірності:

✓ TanStack Start – нижчий LCP на всіх рівнях навантаження (67–107 мс); FCP = LCP у всіх чотирьох вимірюваннях (нульовий розрив); TTFB стабільно 44–49 мс без аномальних стрибків;

✓ Next.js App Router – FCP стабільно 50–55 мс незалежно від обсягу; LCP 97–113 мс із постійним розривом FCP–LCP 42–63 мс; Speed Index 291–330 мс; TTFB 77–120 мс із аномальним стрибком при 75 записах;

✓ Vite + React (CSR) – найнижчий TTFB (33–35 мс) завдяки відсутності серверного рендерингу; варіативний LCP (77–95 мс) без чіткої лінійної залежності від обсягу; CLS = 0,027 в усіх чотирьох вимірюваннях – єдина реалізація з ненульовим зсувом розмітки.

TBT та CLS – нульові в обох фреймворках (виняток: Next.js 2 мс TBT при 100 записах, що відповідає похибці вимірювання); серверний рендеринг гарантує стабільність розмітки.

7. Перспективи подальшого розвитку досліджень

Проведене дослідження охоплює обсяги табличних даних до 100 записів включно в умовах «теплої» бази даних і конфігурації desktop без мережевого та CPU-throttling, що визначає межі його узагальнення й окреслює напрями подальшої роботи. Першочерговим напрямом є розширення діапазону вимірювань до 500–1000 і більше записів: лінії LCP TanStack Start і Next.js зростають з різним нахилом (+40 мс і +16 мс відповідно на 100 записах), і точка їхнього сходження – де перевага блокувального підходу нівелюється – залишається невстановленою. Знаходження цієї точки перетину дозволить сформулювати кількісний поріг, за яким вибір між фреймворками набуває практичної значущості.

Другим перспективним напрямом є повторення вимірювань в умовах mobile throttling – зі зниженою пропускною здатністю мережі та обмеженою потужністю CPU. За умов обмеженого клієнта поведінка Suspense-стрімінгу Next.js і блокувальної стратегії TanStack Start може суттєво відрізнятися від результатів, отриманих на desktop-конфігурації без throttling: прогресивний рендеринг здатен виявити перевагу саме там, де кожна додаткова мілісекунда затримки є відчутною для користувача. Окремий інтерес становить перевірка поведінки при «холодній» базі даних – з дисковими затримками замість буферного кешу – що дозволить оцінити, наскільки стабільна відносна перевага TanStack Start за TTFB в умовах, ближчих до реального виробничого середовища.

Третім напрямом є дослідження впливу явного Suspense-стрімінгу в TanStack Start. У поточній роботі TanStack Start тестувався у блокувальному режимі через createServerFn у route loader-i; використання явних Suspense-меж у TanStack Start може суттєво змінити

співвідношення FCP і LCP, наблизивши його до поведінки Next.js App Router, – і цей варіант конфігурації потребує окремого вимірювання. Нарешті, розширення порівняння на інші фреймворки з підтримкою RSC – зокрема Remix та Astro – дозволить перевірити, чи є виявлені патерни LCP, TTFB і Speed Index специфічними для TanStack Start і Next.js, чи відображають більш загальну закономірність між блокувальними та потоковими підходами до серверного рендерингу.

8. Висновки

У дослідженні проведено контрольоване порівняння TanStack Start, Next.js App Router та Vite + React (CSR) за шістьма метриками Lighthouse при чотирьох рівнях навантаження. Архітектурна відмінність між блокувальним підходом TanStack Start і потоковим Next.js – числово підтверджена в кожній конфігурації таблиці 2. Lighthouse Performance стоїть із стабілізацією в обох фреймворках, що свідчить про неспроможність агрегованого балу розрізняти стратегії рендерингу, і для порівняння потрібні конкретні метрики, а не загальна оцінка.

На основі отриманих даних встановлено:

- ✓ FCP Next.js (50–55 мс) – нижчий, ніж у TanStack Start, але відображає не дані: це час появи loading-оболонки loading.tsx. Реальний вміст (LCP) у Next.js з'являється через 42–63 мс після FCP. У TanStack Start FCP = LCP в усіх чотирьох вимірюваннях – перший видимий елемент одразу містить таблицю.

- ✓ LCP TanStack Start нижчий на всіх рівнях навантаження (67–107 мс проти 97–113 мс); перевага становить 31 % при 25 записах і зростає до 5 % при 100. Обидва фреймворки демонструють лінійне зростання, але з різним нахилом: TanStack Start +40 мс, Next.js +16 мс – при значно більших обсягах перевага може нівелюватися.

- ✓ Speed Index – найбільш значуща відмінність: TanStack Start 114–144 мс проти 291–330 мс у Next.js (у 2.3–2.5 рази). Suspense-скелетон заповнює viewport повільніше, ніж одноразова поява повного вмісту: «прогресивний» підхід інтегрально повільніший.

- ✓ TTFB TanStack Start стабільний (44–49 мс); Next.js – 77–120 мс із аномальним стрибком при 75 записах, що вказує на менш передбачувану поведінку під змінним навантаженням.

- ✓ TBT та CLS нульові в обох SSR-фреймворках – серверний рендеринг гарантує стабільність розмітки; Vite CSR є єдиною реалізацією з ненульовим CLS = 0.027, що підтверджує характерний компроміс клієнтського рендерингу.

TanStack Start підходить для застосунків, де LCP, TTFB і Speed Index є визначальними – зокрема в контексті SEO та Core Web Vitals. Next.js залишається обґрунтованим вибором там, де архітектура проекту спирається на Suspense і прогресивне завантаження, або де потрібна розвинена екосистема фреймворку. Vite CSR залишається доцільним для внутрішніх SPA без SEO-вимог, де простота архітектури важливіша за CLS-стабільність.

Межі узагальнення: вимірювання виконувалися до 100 записів включно – поведінка при 500–1000+ записів, зокрема точка сходження LCP-ліній, не досліджена. Тестування проводилося на «теплій» базі, у режимі desktop, без network/CPU-throttling. Абсолютні значення залежать від апаратного обмеження t3.micro (1 ГБ RAM), хоча відносне порівняння при ідентичних умовах залишається дійсним. Перспективні напрями: знайти точку перетину LCP-ліній при 500–1000 записів; перевірити відносну перевагу в умовах mobile throttling; порівняти поведінку при «холодній» базі даних; оцінити, як зміниться FCP–LCP gap у TanStack Start із явним Suspense-стрімінгом.

Список літератури:

1) React Team. (2024, December 5). React 19. React Developer Blog. <https://react.dev/blog/2024/12/05/react-19>

2) Vercel. (2024). *Next.js documentation: App Router*. <https://nextjs.org/docs/app>

- 3) TanStack. (2025). *TanStack Start overview*. <https://tanstack.com/start/latest/docs/framework/react/overview>
- 4) Iskandar, T. F., Lubis, M., Kusumasari, T. F., & Lubis, A. R. (2020). Comparison between client-side and server-side rendering in the web development. *IOP Conference Series: Materials Science and Engineering*, 801(1), 012136. <https://doi.org/10.1088/1757-899X/801/1/012136>
- 5) Nordström, C., & Dixelius, A. (2023). *Comparisons of server-side rendering and client-side rendering for web pages* [Bachelor's thesis, Uppsala University]. Uppsala University DiVA Portal. <https://uu.diva-portal.org/smash/get/diva2:1775555/FULLTEXT02.pdf>
- 6) Prabowo, A. S., Pati, S., Alam, M., & Zaki, Y. (2025). Evaluating the efficacy of Next.js: A comparative analysis with React.js on performance, SEO, and global network equity. *Companion Proceedings of the ACM on Web Conference 2025*. <https://doi.org/10.1145/3701716.3715565>
- 7) React Team. (2023, March 22). React Labs: What we've been working on – March 2023. React Developer Blog. <https://react.dev/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023>
- 8) Linsley, T. (2024, December 3). Why choose TanStack Start and Router? *TanStack Blog*. <https://tanstack.com/start/latest/docs/blog>
- 9) Google Chrome Developers. (2024). Lighthouse performance scoring. Chrome for Developers. <https://developer.chrome.com/docs/lighthouse/performance/performance-scoring>
- 10) Google Web Dev. (2024). Largest Contentful Paint (LCP). web.dev. <https://web.dev/articles/lcp>
- 11) Google Web Dev. (2024). Top Core Web Vitals recommendations for 2024. web.dev. <https://web.dev/articles/top-cwv>
- 12) Imoh, I. (2023, February 7). Suspense on the server in React 18. Telerik Blog. <https://www.telerik.com/blogs/suspense-server-react-18>
- 13) Saeloun. (2022, January 20). Deep dive into the new Suspense server-side rendering (SSR) architecture in React 18. Saeloun Blog. <https://blog.saeloun.com/2022/01/20/new-suspense-ssr-architecture-in-react-18/>
- 14) LogRocket. (2024, June 4). A guide to streaming SSR with React 18. LogRocket Blog. <https://blog.logrocket.com/streaming-ssr-with-react-18/>
- 15) LogRocket. (2025, April 24). An introduction to the TanStack Start framework. LogRocket Blog. <https://blog.logrocket.com/tanstack-start-overview/>
- 16) Platformatic. (2026, March 24). React SSR benchmark: TanStack Start, React Router, Next.js. Platformatic Blog. <https://blog.platformatic.dev/react-ssr-framework-benchmark-tanstack-start-react-router-nextjs>

Comparative Analysis of Rendering Approaches in React: Server Components (RSC) in TanStack Start and Next.js, and Client-Side Rendering (CSR)

Ihor Andrushchak

Lutsk National Technical University, Lutsk, Ukraine

ORCID 0000-0002-8751-4420

Email address: 9000@lntu.edu.ua

Illia Purshak

Lutsk National Technical University, Lutsk, Ukraine

ORCID 0009-0007-0407-2625

Email address: purshak.illa@gmail.com

Abstract: The choice of rendering strategy in React applications directly determines content display speed and user experience quality. React Server Components (RSC), stabilized in React 19, are implemented in two frameworks with fundamentally different architectures: Next.js App Router sends an HTML shell immediately and streams content through a Suspense boundary, while TanStack Start blocks route rendering until the route loader completes and sends fully formed HTML in a single step. To date, no peer-reviewed comparative studies of these frameworks' performance based on Core Web Vitals metrics exist, which complicates informed decision-making during development. The study conducted controlled testing of TanStack Start and Next.js App Router; Vite + React (CSR) was used as a control implementation to identify the CLS trade-off of client-side rendering. Testing was performed in an isolated AWS EC2 environment; the metric set was composed from Google Lighthouse: LCP, FCP, TBT, CLS, Speed Index, and TTFB. Measurements were taken at four data volumes – 25, 50, 75, and 100 records – under warm PostgreSQL database conditions. The results show that TanStack Start consistently demonstrates lower values for LCP (67–107 ms vs. 97–113 ms in Next.js), TTFB (44–49 ms vs. 77–120 ms), and Speed Index (114–144 ms vs. 291–330 ms) across all load levels. Both SSR frameworks achieve zero CLS values, while Vite CSR exhibits CLS = 0.027 due to re-rendering during data loading. It was established that the aggregate Lighthouse Performance score of 100/100 is identical in both SSR frameworks and does not reflect differences in rendering strategy. The obtained results provide a practical basis for informed selection of a rendering framework when developing React applications with Core Web Vitals performance requirements.

Keywords: React Server Components, TanStack Start, Next.js, client-side rendering, web application performance, Lighthouse, Largest Contentful Paint, Time to First Byte, benchmark.

Додаток Б

Наукова стаття «Порівняльний аналіз продуктивності Prisma ORM та нативного pg-драйвера при роботі з PostgreSQL у Node.js-застосунках»

International Science Journal of Engineering & Agriculture

2022; X(X): XX-XX

<https://isg-journal.com/XXXXXX/>

doi: 10.46299/j.xxx.xxxxxxx.xx

ISSN: XXXX-XXXX



Інформаційні технології

Порівняльний аналіз продуктивності Prisma ORM та нативного pg-драйвера при роботі з PostgreSQL у Node.js-застосунках

Ігор Андрушак

Луцький національний технічний університет, Луцьк, Україна

ORCID 0000-0002-8751-4420

Електронна адреса: 9000@lntu.edu.ua

Ілля Пуршак

Луцький національний технічний університет, Луцьк, Україна

ORCID 0009-0007-0407-2625

Електронна адреса: purshak.illa@gmail.com

Анотація: Вибір інструменту доступу до бази даних є одним із базових архітектурних рішень при розробці серверних застосунків на Node.js. ORM-інструменти, зокрема Prisma, спрощують взаємодію з реляційними базами даних через типобезпечний інтерфейс, однак вносять додатковий рівень абстракції, вплив якого на продуктивність запитів залишається недостатньо дослідженим у контексті актуальних версій інструменту. У 2025 році Prisma здійснила архітектурний перехід від внутрішнього Rust-рушія до нативного TypeScript-рушія, що принципово змінило профіль продуктивності інструменту і зробило результати попередніх порівняльних досліджень некоректними для екстраполяції на поточний стан екосистеми. У дослідженні проведено контрольований бенчмарк Prisma ORM версії 7 та нативного pg-драйвера версії 8 при роботі з PostgreSQL 17 у Node.js-середовищі. Вимірювання виконувались ізольовано від HTTP-шару як автономний Node.js-процес поза контекстом веб-фреймворку на реальній схемі бази даних TypeScript-застосунку з 10 000 записів. Для кожного з чотирьох типів операцій – простого SELECT, фільтрованого SELECT із пагінацією, INSERT та JOIN-запиту з об'єднанням чотирьох таблиць – виконано 1000 ітерацій із прогрівом у 50 запитів. Зафіксовано середнє значення, медіану, мінімум, максимум та стандартне відхилення латентності. За результатами дослідження pg-драйвер демонструє нижчу середню латентність у всіх чотирьох сценаріях – від ~25% для INSERT до ~47% для JOIN-запиту. Зафіксований розрив є суттєво меншим порівняно з результатами попередніх досліджень, що підтверджує якісну зміну продуктивності Prisma після переходу на TypeScript-рушій. Абсолютна різниця між підходами становить 0.5–1.1 мс, що є статистично достовірним, але практично незначущим для переважної більшості продуктивних застосунків. Стабільність результатів підтверджена повторним запуском на наборі з 100 записів. Отримані результати формують практичну основу для обґрунтованого вибору інструменту доступу до бази даних при розробці TypeScript-застосунків на Node.js із PostgreSQL.

Ключові слова: Prisma ORM, pg-драйвер, PostgreSQL, Node.js, TypeScript, продуктивність бази даних, бенчмарк, латентність запитів, ORM-абстракція, порівняльний аналіз.

1. Вступ

Сучасна розробка серверних застосунків на Node.js нерозривно пов'язана з вибором інструменту для роботи з реляційними базами даних. Серед доступних підходів особливе місце займають ORM-інструменти – бібліотеки, що забезпечують об'єктно-орієнтований

інтерфейс до реляційних даних і дозволяють розробнику взаємодіяти зі схемою бази даних засобами мови програмування без написання SQL вручну. PostgreSQL залишається одною з найпопулярніших реляційних СУБД у світі, стабільно посідаючи провідні позиції серед інструментів, що використовуються у виробничих середовищах [1]. У TypeScript-екосистемі Node.js одним із найбільш поширених ORM-інструментів є Prisma – інструмент із декларативною схемою, автоматично згенерованим типобезпечним клієнтом та вбудованою системою міграцій [2, 3]. Строга типізація запитів, яку забезпечує Prisma, дозволяє виявляти помилки на етапі компіляції, що суттєво підвищує надійність коду при роботі зі складними схемами даних [4].

Попри очевидні переваги з точки зору зручності розробки, використання ORM-інструментів неминуче вносить додатковий шар абстракції між застосунком і базою даних. Це ставить під питання, наскільки суттєвими є накладні витрати цієї абстракції порівняно з прямим зверненням до бази даних через нативний драйвер – у випадку Node.js та PostgreSQL це бібліотека pg. Відповідь на це питання є практично значущою: у застосунках із високим навантаженням або з вимогами до мінімальної латентності вибір між ORM та нативним драйвером може безпосередньо впливати на продуктивність системи в цілому.

Додаткової актуальності цьому питанню надає архітектурна еволюція самої Prisma. До виходу версії 7 інструмент використовував внутрішній Rust-рушій, який вносив суттєві накладні витрати, задокументовані в низці досліджень. У 2025 році Prisma здійснила перехід на нативний TypeScript-рушій, що принципово змінило характеристики продуктивності інструменту. Результати попередніх порівняльних досліджень, таким чином, більше не відображають актуального стану речей і потребують переосмислення в контексті нової архітектури.

Метою цієї роботи є проведення контрольованого бенчмарку Prisma 7 та нативного pg-драйвера на рівні запиту до бази даних, аналіз отриманих результатів та формулювання практичних рекомендацій щодо вибору інструменту для роботи з PostgreSQL у Node.js-застосунках на TypeScript.

2. Об'єкт і предмет дослідження

Об'єктом дослідження є продуктивність інструментів доступу до реляційних баз даних у серверних Node.js-застосунках на TypeScript, а саме: як вибір між ORM-абстракцією та нативним драйвером впливає на латентність запитів при роботі з PostgreSQL. Сучасні серверні застосунки на Node.js взаємодіють із базою даних через один із двох принципово різних підходів: або через ORM-інструмент, що надає типобезпечний об'єктно-орієнтований інтерфейс і приховує SQL за рівнем абстракції, або через нативний драйвер, що виконує SQL-запити безпосередньо без проміжних перетворень. Вибір між цими підходами є одним із базових архітектурних рішень при проектуванні бекенду, і від нього залежить не лише продуктивність, а й зручність розробки, типова безпека та підтримуваність коду.

Prisma ORM є одним із найпоширеніших інструментів у TypeScript-екосистемі Node.js. Він надає декларативну схему бази даних, автоматично згенерований типобезпечний клієнт та вбудовану систему міграцій. Нативний pg-драйвер є низькорівневим інтерфейсом до PostgreSQL, що дозволяє виконувати довільні SQL-запити з мінімальними накладними витратами. Принципово важливим контекстом є те, що у Prisma 7 обидва підходи спираються на один і той самий низькорівневий механізм з'єднань – pg використовується як транспортний рівень через PrismaPg-адаптер, – що дозволяє ізолювати вимірювання виключно на рівні абстракції ORM.

Предметом дослідження є відмінності у латентності запитів між Prisma ORM версії 7 та нативним pg-драйвером версії 8 при виконанні чотирьох типів операцій – простого SELECT за первинним ключем, фільтрованого SELECT із пагінацією, INSERT та JOIN-запиту з

об'єднанням чотирьох таблиць – на реальній схемі бази даних TypeScript-застосунку з використанням PostgreSQL 17.

Предмет включає дослідження того, як накладні витрати ORM-абстракції – мапінг результатів у типізовані TypeScript-об'єкти, генерація SQL із декларативного опису запиту, управління пулом з'єднань – розподіляються між різними типами операцій і чи є ця різниця практично значущою в контексті реальних серверних застосунків. Окремо розглядається питання стабільності латентності: чи відрізняється варіативність результатів між підходами, і що це означає для систем із вимогами до передбачуваності часу відповіді.

У межах дослідження також розглядається версійний контекст: архітектурний перехід Prisma від Rust-рушія до нативного TypeScript-рушія у версії 7 змінив профіль продуктивності інструменту, і результати попередніх досліджень потребують переосмислення з урахуванням цієї зміни. Таким чином, предмет дослідження охоплює як кількісний аспект – числове порівняння латентності у контрольованих умовах – так і якісний: інтерпретацію отриманих результатів у контексті архітектурної еволюції інструменту та їх практичне значення для прийняття рішень при розробці TypeScript-застосунків.

3. Мета та задачі дослідження

Метою дослідження є кількісне порівняння продуктивності Prisma ORM версії 7 та нативного pg-драйвера при роботі з PostgreSQL у Node.js-середовищі на основі контрольованого бенчмарку, а також формулювання практичних рекомендацій щодо вибору інструменту доступу до бази даних для TypeScript-застосунків.

Досягнення мети передбачає вирішення таких взаємопов'язаних задач:

- проектування методології бенчмарку: визначити набір операцій, що репрезентують основні сценарії взаємодії з базою даних у серверних застосунках; обґрунтувати протокол вимірювання, що забезпечує ізоляцію досліджуваної змінної від зовнішніх чинників – HTTP-шару, серіалізації та мережевої варіативності; сформулювати вимоги до статистичної достовірності результатів;
- реалізація тестового середовища: розробити автономний бенчмарк-скрипт поза контекстом веб-фреймворку; реалізувати дзеркальні імплементації чотирьох операцій через Prisma та pg з ідентичною семантикою запитів; забезпечити симетричність умов порівняння шляхом використання спільного механізму пулу з'єднань для обох підходів через PrismaPg-адаптер;
- проведення вимірювань: виконати 1000 ітерацій із прогрівом у 50 запитів для кожної операції на наборі з 10 000 записів; зафіксувати середнє значення, медіану, мінімум, максимум та стандартне відхилення латентності; підтвердити стабільність результатів повторним запуском на наборі з 100 записів;
- аналіз та інтерпретація результатів: встановити відносну різницю між підходами для кожного типу операцій; пояснити природу накладних витрат ORM-абстракції через призму архітектурних змін Prisma 7; зіставити отримані результати з висновками попередніх досліджень у контексті версійної еволюції інструменту;
- формулювання рекомендацій: визначити сценарії, у яких різниця між підходами є практично значущою, та сценарії, де вона не впливає на вибір інструменту; надати обґрунтовану рекомендацію щодо вибору між Prisma та pg для типових TypeScript-застосунків на Node.js.

4. Аналіз літератури

Питання продуктивності ORM-інструментів у порівнянні з прямими підходами до роботи з базами даних досліджувалося в різних технологічних контекстах, однак специфіка Node.js-

екосистеми та TypeScript-орієнтованих застосунків залишається недостатньо висвітленою в академічній літературі.

Серед найбільш релевантних робіт слід виділити дослідження, опубліковане у збірнику наукових праць «SCIENTIA» (Гельсінкі, 2023), у якому Жихарський П. та Губарева О. провели порівняльний аналіз підходів до роботи з базами даних у контексті API-застосунків [5]. Автори зафіксували суттєву різницю в часі виконання запитів між нативним SQL та ORM-рішеннями, зазначивши, що накладні витрати ORM особливо відчутні при складних запитах із з'єднаннями таблиць. Разом із тим дослідження не охоплювало Node.js-стек і Prisma ORM зокрема, що залишає відкритим питання застосовності цих висновків у сучасних TypeScript-застосунках.

Загальнішу картину продуктивності ORM у середовищі .NET окреслює робота Гуверчіна А. та Авеноглу Б., опублікована у «Bilişim Teknolojileri Dergisi» (2022), де автори дослідили поведінку різних ORM-інструментів у середовищі .NET 6 [6]. Дослідження підтвердило, що ORM-рішення стабільно поступаються нативному SQL за абсолютними показниками швидкодії, однак ступінь цього відставання суттєво залежить від типу операції та конфігурації інструменту. Аналогічні висновки отримали Змаранда Д. та співавтори (2020), які зафіксували переваги нативного підходу при операціях CRUD у середовищі .NET, хоч і відзначили, що різниця в продуктивності скорочується з ростом складності бізнес-логіки [7]. Обидві роботи стосуються платформ, архітектурно відмінних від Node.js, що ускладнює перенесення їхніх результатів на досліджуваний контекст.

Яннелі Р. у роботі «Object Relational Mapping Framework Performance Impact» (2021) досліджував вплив ORM-фреймворків на продуктивність запитів у широкому класі застосунків [8]. Автор зробив висновок, що накладні витрати ORM пов'язані не стільки з абстракцією як такою, скільки з кількістю SQL-запитів, що генеруються для однієї операції на рівні застосунку – особливо у сценаріях із вибіркою пов'язаних сутностей. Цей висновок безпосередньо стосується Prisma ORM, оскільки інструмент за замовчуванням виконує окремі запити для кожного включеного відношення замість JOIN. Практичне підтвердження застосовності Prisma ORM у реальних проектах із складною схемою бази даних представлено також у роботі Мутіари Аулії Хадіджи та співавторів (2024), де інструмент використовувався для реалізації модуля управління ресурсами у виробничій системі з кількома взаємопов'язаними сутностями [9].

З боку самої екосистеми Prisma заслуговують уваги офіційні бенчмарки, опубліковані командою розробників у 2024 році [10, 11]. У цих матеріалах Prisma ORM порівнювалася з TypeORM та Drizzle ORM на реальній інфраструктурі AWS RDS та Supabase, з використанням 500 ітерацій на кожну операцію. Результати засвідчили, що різниця в медіанній латентності між досліджуваними ORM-рішеннями є незначною для простих операцій вибірки, натомість суттєво зростає при складних запитах із вкладеними відношеннями.

Комплексне дослідження, опубліковане на ScienceDirect (2025), присвячене порівнянню Prisma ORM та оптимізованого нативного SQL на PostgreSQL у контейнеризованому середовищі, представляє найближчу до цієї роботи постановку задачі [12]. Автори зафіксували, що нативний SQL виявився до п'яти разів швидшим за часом виконання, а також споживав у 6–9 разів менше CPU та у 2–3 рази менше пам'яті порівняно з Prisma. Робота Йокубжонова Д. та Гібадуліна Р. (2024) досліджувала схожу проблематику у контексті Microsoft SQL Server і підтвердила загальну тенденцію переваги нативного підходу, вказавши водночас, що для простих CRUD-операцій різниця може бути статистично незначущою [13].

Однак критично важливим контекстом для інтерпретації наведених результатів є версійна історія Prisma ORM. До виходу версії 7 Prisma використовувала внутрішній Rust-рушій для виконання запитів, що вносило суттєві накладні витрати, особливо помітні при роботі з великими наборами результатів. У 2025 році команда Prisma здійснила повний архітектурний перехід на нативний TypeScript-рушій, офіційно зафіксувавши триразове прискорення виконання запитів та 90-відсоткове скорочення розміру бандлу порівняно з Rust-реалізацією [14]. Це означає, що результати всіх досліджень, проведених до виходу Prisma 7,

характеризують інструмент у принципово іншому технічному стані, ніж той, що є актуальним на момент написання цієї роботи.

Таким чином, аналіз наявних публікацій виявляє прогалину, яку заповнює це дослідження: більшість порівняльних робіт або орієнтовані на платформи поза Node.js-екосистемою, або ґрунтуються на версіях Prisma з Rust-рушієм, результати яких не можуть бути коректно екстрапольовані на поточний стан інструменту. Крім того, більшість існуючих бенчмарків вимірюють час відповіді на рівні HTTP-запиту, що поєднує в одному показнику затримку бази даних, серіалізацію, мережевий стек та обробку у фреймворку. Це дослідження спрямоване на заповнення зазначеної прогалини шляхом проведення ізольованого бенчмарку Prisma 7 та нативного pg-драйвера безпосередньо на рівні запиту до бази даних, на реальній схемі TypeScript-застосунку з використанням PostgreSQL.

5. Методи досліджень

Для досягнення мети дослідження було розроблено контрольований бенчмарк, що дозволяє ізольовано виміряти латентність запитів до бази даних при використанні двох підходів – Prisma ORM та нативного pg-драйвера – в ідентичних умовах виконання.

Середовище виконання. Експеримент проводився локально на одній машині з метою виключення мережевої варіативності між застосунком і базою даних. База даних PostgreSQL запущена у Docker-контейнері. Версії компонентів: Node.js 22, Prisma 7, pg 8, PostgreSQL 17. З'єднання з базою даних в обох підходах здійснювалося через пул з'єднань – у Prisma це вбудований пул, у pg використовувався Pool з бібліотеки node-postgres [15]. Це забезпечило симетричність умов, оскільки обидва підходи виконували запити через постійні з'єднання без накладних витрат на їх встановлення при кожній ітерації.

Схема даних. Бенчмарк виконувався на реальній схемі бази даних TypeScript-застосунку, що включає таблицю користувачів із пов'язаними сутностями: обліковими даними, профілем організації та налаштуваннями двофакторної автентифікації. Схема налічує понад 18 полів у межах одного запиту з об'єднаннями. База даних попередньо заповнена фіксованим набором із 10 000 записів користувачів, згенерованих за допомогою seed-скрипту. Набір даних не змінювався між запусками.

Операції тестування. Для порівняння обрано чотири типи операцій, що охоплюють основні сценарії взаємодії з базою даних у серверних застосунках: простий SELECT за первинним ключем (отримання одного запису користувача за ідентифікатором), фільтрований SELECT із пагінацією (вибір активних користувачів із обмеженням у 20 записів), INSERT із подальшим видаленням створеного запису (для забезпечення ізольованості між ітераціями) та JOIN-запит із об'єднанням чотирьох таблиць (користувач разом із обліковими даними, профілем та налаштуваннями двофакторної автентифікації).

Протокол вимірювання. Кожна операція виконувалася в 1000 ітерацій із попереднім прогрівом у 50 ітерацій, результати яких не враховувалися. Прогрів необхідний для того, щоб пул з'єднань встановився, а перші аномально повільні запити не впливали на статистику. Час кожної ітерації фіксувався засобами performance.now() безпосередньо навколо виклику до бази даних, без включення будь-якої іншої логіки застосунку. HTTP-шар, серіалізація відповіді та middleware фреймворку були повністю виключені з вимірювання. За результатами 1000 вимірювань для кожної операції обчислювались такі статистичні показники: середнє значення (mean), медіана (p50), мінімум, максимум та стандартне відхилення.

Реалізація. Обидва підходи реалізовані як окремі модулі в межах одного бенчмарк-скрипту, що виконується поза контекстом веб-сервера. Операції Prisma та pg чергувалися між собою в рамках кожної ітерації, а не виконувалися блоками – це унеможливило вплив теплового стану процесора та кешу бази даних на відносні результати одного з підходів.

6. Результати досліджень

За результатами проведеного експерименту отримано статистичні показники латентності для чотирьох типів операцій при двох підходах до роботи з базою даних. Основні результати наведено в таблиці 1, що містить дані для набору з 10 000 записів.

Таблиця 1 . Результати бенчмарку Prisma 7 vs pg 8 (10 000 записів, 1000 ітерацій)

Операція	Інструмент	Mean, мс	p50, мс	Min, мс	Max, мс	Stddev, мс	Перевага pg, %
Simple SELECT	Prisma	1,449	1,344	0,877	23,322	1,159	–
Simple SELECT	pg	0,846	0,786	0,534	17,557	0,611	41,6%
Filtered SELECT	Prisma	1,493	1,445	0,988	21,206	0,954	–
Filtered SELECT	pg	0,917	0,872	0,594	17,464	0,738	38,6%
INSERT	Prisma	2,766	2,130	1,760	25,658	3,285	–
INSERT	pg	2,063	1,533	1,200	24,803	2,973	25,4%
JOIN	Prisma	2,366	2,287	1,655	20,903	1,020	–
JOIN	pg	1,248	1,176	0,880	21,844	0,863	47,3%

У всіх чотирьох операціях pg демонструє нижче середнє значення латентності порівняно з Prisma. Найбільша відносна різниця зафіксована для операції простого SELECT – середня латентність pg становить 0.846 мс проти 1.449 мс у Prisma, що відповідає перевазі у ~41%. Схожу картину демонструє JOIN-запит: 1.248 мс проти 2.366 мс, різниця ~47%. Для фільтрованого SELECT різниця за середнім значенням становить ~39%. Найменша відносна різниця зафіксована для операції INSERT – 2.063 мс проти 2.766 мс, що відповідає ~25%. При цьому стандартні відхилення для INSERT є найвищими серед усіх операцій і близькими між собою для обох підходів (3.285 мс у Prisma проти 2.973 мс у pg), що свідчить про те, що варіативність латентності INSERT визначається переважно поведінкою самої бази даних при операціях запису, а не рівнем абстракції інструменту.

Окремої уваги заслуговує порівняння стабільності результатів. Незважаючи на те, що pg стабільно швидший за середнім значенням, Prisma демонструє зріване або менше стандартне відхилення в більшості операцій – зокрема для простого SELECT (1.159 мс проти 0.611 мс у pg) та JOIN (1.020 мс проти 0.863 мс у pg). Це означає що перевага pg за швидкістю супроводжується дещо вищою варіативністю латентності в окремих сценаріях.

З метою перевірки стабільності отриманих результатів при різних розмірах набору даних експеримент було повторено на базі з 100 записів за ідентичним протоколом. Скорочені результати наведено в таблиці 2.

Таблиця 2. Порівняння mean та p50 при 100 та 10 000 записах

Операція	Інструмент	100 записів		10 000 записів	
		Mean, мс	p50, мс	Mean, мс	p50, мс
Simple SELECT	Prisma	1,399	1,366	1,449	1,344
Simple SELECT	pg	0,814	0,772	0,846	0,786
Filtered SELECT	Prisma	1,342	1,207	1,493	1,445
Filtered SELECT	pg	0,868	0,732	0,917	0,872
INSERT	Prisma	2,649	2,025	2,766	2,130
INSERT	pg	2,080	1,457	2,063	1,533
JOIN	Prisma	2,302	2,144	2,366	2,287
JOIN	pg	1,210	1,040	1,248	1,176

Результати двох запусків є практично ідентичними – різниця між середніми значеннями не перевищує 10% для жодної з операцій. Відносна перевага pg над Prisma залишилась незмінною в обох конфігураціях, що підтверджує стабільність і відтворюваність отриманих результатів незалежно від розміру набору даних у дослідженому діапазоні. Ця стабільність пояснюється тим, що всі запити спираються на індекси – пошук за первинним ключем та фільтрація за індексованим полем – тому планувальник PostgreSQL обирає однаковий план виконання незалежно від кількості рядків у таблиці.

7. Перспективи подальшого розвитку досліджень

Проведене дослідження охоплює чотири типи операцій на схемі з 10 000 записів в умовах локального підключення без мережевої затримки між застосунком і базою даних, що визначає межі його узагальнення та окреслює напрями подальшої роботи. Першочерговим напрямом є дослідження поведінки обох підходів в умовах конкурентного навантаження з паралельними запитами. Поточний бенчмарк виконує операції послідовно, що не відображає реального профілю навантаження серверного застосунку, де пул з'єднань обслуговує одночасні запити від різних клієнтів. В умовах конкуренції за з'єднання накладні витрати управління пулом можуть по-різному проявлятися у Prisma та pg, і відносна різниця в латентності може суттєво змінитись порівняно з результатами, отриманими при послідовному виконанні.

Другим перспективним напрямом є повторення вимірювань в умовах реальної мережевої затримки між застосунком і базою даних – коли PostgreSQL розгорнуто на окремому сервері, а не в Docker-контейнері на тій самій машині. У поточному експерименті абсолютні значення латентності є нижчими, ніж у типових виробничих умовах, саме через відсутність мережевого round-trip. При збільшенні базової мережевої затримки абсолютна різниця 0.5–1.1 мс між підходами може набувати іншого практичного значення: відносна перевага pg залишиться незмінною, але питома вага ORM-накладних витрат у загальному часі відповіді зменшиться, що додатково обґрунтовує вибір Prisma для більшості продуктивних сценаріїв.

Третім напрямом є оцінка впливу Prisma Accelerate – хмарного рівня кешування та пулу з'єднань від команди Prisma – на продуктивність у виробничому середовищі. Поточне дослідження порівнює Prisma та pg за умови прямого підключення до бази даних; введення Prisma Accelerate як проміжного рівня може суттєво змінити профіль латентності для операцій читання, де кешування є найбільш ефективним. Нарешті, розширення набору операцій на сценарії з вкладеними відношеннями – де Prisma за замовчуванням виконує окремі запити замість JOIN – дозволить кількісно оцінити проблему N+1 запитів у контексті нового TypeScript-рушія і перевірити, чи зберігається ця поведінка після архітектурного переходу від Rust-реалізації.

8. Висновки

У дослідженні проведено контрольований бенчмарк Prisma 7 та нативного pg-драйвера на рівні запиту до бази даних PostgreSQL у Node.js-середовищі. Вимірювання виконувались ізольовано від HTTP-шару на реальній схемі TypeScript-застосунку з використанням 1000 ітерацій та прогрівом у 50 запитів для кожної з чотирьох операцій. Стабільність результатів підтверджена повторним запуском на наборі з 100 записів, який показав відхилення не більше 10% від основних значень при незмінній відносній різниці між підходами.

На основі отриманих даних встановлено:

1. pg-драйвер демонструє нижчу середню латентність у всіх чотирьох сценаріях – від ~25% для INSERT до ~47% для JOIN-запиту. Найменша різниця зафіксована для операції INSERT, де стандартні відхилення обох підходів практично збігаються (3.285 мс у Prisma проти 2.973 мс у pg), що свідчить про домінування затримок на рівні самої бази даних при операціях запису. Найбільша відносна різниця зафіксована для JOIN-запиту та простого

SELECT, де накладні витрати мапінгу результатів у типізовані TypeScript-об'єкти є найбільш відчутними.

2. Зафіксований розрив між підходами є суттєво меншим порівняно з результатами попередніх досліджень, де перевага нативного SQL над Prisma сягала п'яти разів. Це підтверджує що архітектурний перехід Prisma 7 від Rust-рушія до нативного TypeScript-рушія принципово змінив профіль продуктивності інструменту – і попередні порівняльні результати не можуть бути коректно екстрапольовані на поточний стан екосистеми.

3. Абсолютна різниця між підходами становить 0.5–1.1 мс залежно від типу операції. У контексті реального серверного застосунку, де загальний час обробки HTTP-запиту вимірюється десятками мілісекунд, ця різниця є статистично достовірною, але практично незначущою для переважної більшості продуктивних сценаріїв. Prisma 7 є обґрунтованим вибором за замовчуванням для TypeScript-застосунків на Node.js – перехід на нативний pg виправданий лише тоді, коли профілювання підтверджує що рівень ORM-абстракції є вузьким місцем продуктивності, а не мережа, серіалізація чи бізнес-логіка.

4. Межі узагальнення: експеримент проводився локально з підключенням до бази даних через Docker без мережевої затримки між застосунком і сервером бази даних, що дає нижчі абсолютні значення латентності порівняно з виробничими умовами, хоча відносне порівняння між підходами залишається дійсним. Результати отримані на схемі з 10 000 записів – поведінка при значно більших наборах даних зі складними індексами не досліджувалась. Перспективні напрями: дослідити поведінку при конкурентному навантаженні з паралельними запитами; порівняти результати в умовах реальної мережевої затримки між застосунком і базою даних; оцінити вплив Prisma Accelerate як рівня кешування на продуктивність у виробничому середовищі.

Список літератури

- 1) Akhtar, A. (2023). Popularity ranking of database management systems. <https://arxiv.org/abs/2301.00847>
- 2) Prisma. (n.d.). Prisma ORM | Type-safe ORM for Node.js and TypeScript. <https://www.prisma.io/orm>
- 3) Strapi. (2024). Top 5 ORMs for developers in 2025. <https://strapi.io/blog/orms-for-developers>
- 4) Prisma. (n.d.). Type safety. <https://www.prisma.io/docs/orm/prisma-client/type-safety>
- 5) Zhykharskyi, P., & Gubaryeva, O. (2023). Pure SQL vs ORM libraries: A comparative analysis for API applications. Collection of Scientific Papers «SCIENTIA» (June 30, 2023; Helsinki, Finland), 97–98.
- 6) Güvercin, A. E., & Avenoglu, B. (2022). Performance analysis of object-relational mapping (ORM) tools in .NET 6 environment. Bilişim Teknolojileri Dergisi, 15(4), 453–465. <https://doi.org/10.17671/gazibtd.1059516>
- 7) Zmărănda, D., Pop-Fele, L.-L., Györödi, C., Györödi, R., & Pecherle, G. (2020). Performance comparison of CRUD methods using .NET object relational mappers: A case study. International Journal of Advanced Computer Science and Applications, 11(1). <https://www.ijacsa.thesai.org>
- 8) Jannali, R. (2021). Object relational mapping framework performance impact. Turkish Journal of Computer and Mathematics Education, 12(7).
- 9) Mutiara Auliya Khadija et al. (2024). Backend programming techniques in the development of resource manager features in VRMS systems based on ORM Prisma. Decode: Jurnal Pendidikan Teknologi Informasi, 4(3), 830–843. <https://doi.org/10.51454/decode.v4i3.792>
- 10) Prisma. (2024). Performance benchmarks: Comparing query latency across TypeScript ORMs & databases. <https://www.prisma.io/blog/performance-benchmarks-comparing-query-latency-across-typescript-orms-and-databases>
- 11) Prisma. (n.d.). Query performance benchmarks. <https://benchmarks.prisma.io/>
- 12) Yusmita, J. C., Arya, R., Wijaya, J. M., Suryaningrum, K. M., & Siswanto, R. R. (2025). Optimizing database access strategy: A performance analysis comparison of raw SQL and

- Prisma ORM. Procedia Computer Science.
<https://www.sciencedirect.com/science/article/pii/S187705092502722X>
- 13) Yoqubjonov, D. I., & Gibadullin, R. F. (2024). A performance study of object-relational mapping technologies when interacting with Microsoft SQL Server. International Research Journal.
- 14) Prisma. (2025). Prisma ORM without Rust: Latest performance benchmarks. <https://www.prisma.io/blog/prisma-orm-without-rust-latest-performance-benchmarks>
- 15) node-postgres. (n.d.). Connection pooling. <https://node-postgres.com/features/pooling>

Comparative Analysis of the Performance of Prisma ORM and the Native pg Driver When Working with PostgreSQL in Node.js Applications

Ihor Andrushchak

Lutsk National Technical University, Lutsk, Ukraine

ORCID 0000-0002-8751-4420

Email address: 9000@lntu.edu.ua

Illa Purshak

Lutsk National Technical University, Lutsk, Ukraine

ORCID 0009-0007-0407-2625

Email address: purshak.illa@gmail.com

Abstract: The choice of database access tool is one of the fundamental architectural decisions in Node.js server-side application development. ORM tools, including Prisma, simplify interaction with relational databases through a type-safe interface, yet introduce an additional abstraction layer whose impact on query performance remains insufficiently studied in the context of current tool versions. In 2025, Prisma underwent an architectural transition from an internal Rust engine to a native TypeScript engine, which fundamentally changed the performance profile of the tool and rendered the results of previous comparative studies invalid for extrapolation to the current state of the ecosystem. This study presents a controlled benchmark of Prisma ORM version 7 and the native pg driver version 8 operating against PostgreSQL 17 in a Node.js environment. Measurements were performed in isolation from the HTTP layer as a standalone Node.js process outside the context of any web framework, on a real TypeScript application database schema with 10,000 records. For each of four operation types – simple SELECT, filtered SELECT with pagination, INSERT, and a JOIN query across four tables – 1,000 iterations were executed with a 50-query warm-up. Mean, median, minimum, maximum, and standard deviation of latency were recorded. The results show that the pg driver demonstrates lower mean latency across all four scenarios – ranging from ~25% for INSERT to ~47% for the JOIN query. The observed gap is substantially smaller than reported in previous studies, confirming a qualitative shift in Prisma's performance following the transition to the TypeScript engine. The absolute difference between the approaches is 0.5–1.1 ms, which is statistically significant but practically negligible for the vast majority of production applications. Result stability was confirmed by a repeated run on a dataset of 100 records. The findings provide a practical basis for informed selection of a database access tool when developing TypeScript applications on Node.js with PostgreSQL.

Keywords: Prisma ORM, pg driver, PostgreSQL, Node.js, TypeScript, database performance, benchmark, query latency, ORM abstraction, comparative analysis.

Додаток В

Таблиця В.1 – Тест-кейси ручного функціонального тестування

Модуль	Тест-кейс	Очікуваний результат	Результат
Реєстрація	Реєстрація нового клієнта з валідними даними	Обліковий запис створено, повернуто profileSetupToken, перехід до заповнення профілю	Відповідає очікуваному
Реєстрація	Реєстрація з уже існуючим email	HTTP 409, повідомлення «Email already in use»	Відповідає очікуваному
Автентифікація	Вхід клієнта без 2FA	Повернуто accessToken і refreshToken, метадані підписки	Відповідає очікуваному
Автентифікація	Вхід клієнта з увімкненою 2FA	Повернуто pendingToken, запит TOTP-коду; після коректного коду – пара токенів	Відповідає очікуваному
Автентифікація	Вхід з неправильним паролем	HTTP 401 Unauthorized	Відповідає очікуваному
Оператори	Підключення до Нової Пошти з валідним API-ключем	З'єднання зі статусом ACTIVE збережено	Відповідає очікуваному
Оператори	Підключення до Нової Пошти з недійсним API-ключем	HTTP 500, підключення не створено	Відповідає очікуваному
Відправлення	Запит агрегованого списку (два активних оператори)	Об'єднаний масив від Нової Пошти і Укрпошти, відсортований за датою	Відповідає очікуваному
Відправлення	Фільтрація за normalizedStatus = DELIVERED	Лише доставлені відправлення від усіх операторів	Відповідає очікуваному
Підписка	Запит на підвищення плану FREE → PRO	Зміна запланована: scheduledSwitchTo = PRO; поточний план активний до кінця розрахункового циклу	Відповідає очікуваному
Підписка	Пониження плану при перевищенні ліміту операторів	Надлишкові підключення переведено у BLOCKED; ключі збережено	Відповідає очікуваному
Підтримка	Створення тикету клієнтом	Тікет зі статусом OPEN збережено; у відповіді повернуто id	Відповідає очікуваному
Підтримка	Перегляд списку клієнтів з роллю ADMIN	Пагінований список клієнтів із фільтрами	Відповідає очікуваному
Адмін	Ручне оновлення підписки клієнта	Підписка оновлена, зміна набрала чинності негайно (обхід планувальника)	Відповідає очікуваному
Адмін	Запрошення нового адміністратора (роль SUPER_ADMIN)	AdminInvite зі SHA-256-хешем токена і терміном дії 7 днів збережено; токен повернуто у відповіді	Відповідає очікуваному

Додаток Г

Таблиця Г.1 – Результати юніт-тестування

Модуль	Тестовий набір	Тестів	Статус
Nova Post API клієнт	NovaPostApiClient	11	Пройдено
Контролер Nova Post авторизації	NovaPostAuthController	6	Пройдено
Контролер відправлень	ShipmentsController	11	Пройдено
Контролер автентифікації (клієнт)	AuthController	12	Пройдено
Контролер автентифікації (адмін)	AdminAuthController	10	Пройдено
Контролер профілю (клієнт)	ProfileController	5	Пройдено
Контролер профілю (адмін)	AdminProfileController	3	Пройдено
Контролер підписок (клієнт)	SubscriptionController	5	Пройдено
Контролер підписок (адмін)	AdminSubscriptionController	6	Пройдено
Контролер планів (адмін)	AdminPlansController	5	Пройдено
Контролер поштових операторів	PostalConnectionsController	7	Пройдено
Сервіс поштових операторів	PostalConnectionsService	13	Пройдено
Контролер одержувачів	RecipientsController	5	Пройдено
Контролер шаблонів	TemplatesController	6	Пройдено
Контролер чернеток	DraftsController	5	Пройдено
Контролер підтримки (клієнт)	SupportController	5	Пройдено
Контролер підтримки (адмін)	AdminSupportController	6	Пройдено
Сервіс підтримки (адмін)	AdminSupportService	2	Пройдено
Контролер сповіщень	NotificationsController	6	Пройдено
Контролер білінгу	BillingController	5	Пройдено
Контролер онбордингу	OnboardingController	1	Пройдено
Контролер операторів (адмін)	AdminServicesController	5	Пройдено
Контролер адміністраторів	AdminAdminsController	6	Пройдено
Контролер користувачів (адмін)	AdminUsersController	3	Пройдено
Сервіс статистики (адмін)	AdminStatisticsService	2	Пройдено
Контролер статистики (адмін)	AdminStatisticsController	1	Пройдено
Guard доступу адміна	ActiveAdminAccessGuard	4	Пройдено
Кореневий контролер	AppController	1	Пройдено
Разом	28 тестових наборів	157	Пройдено