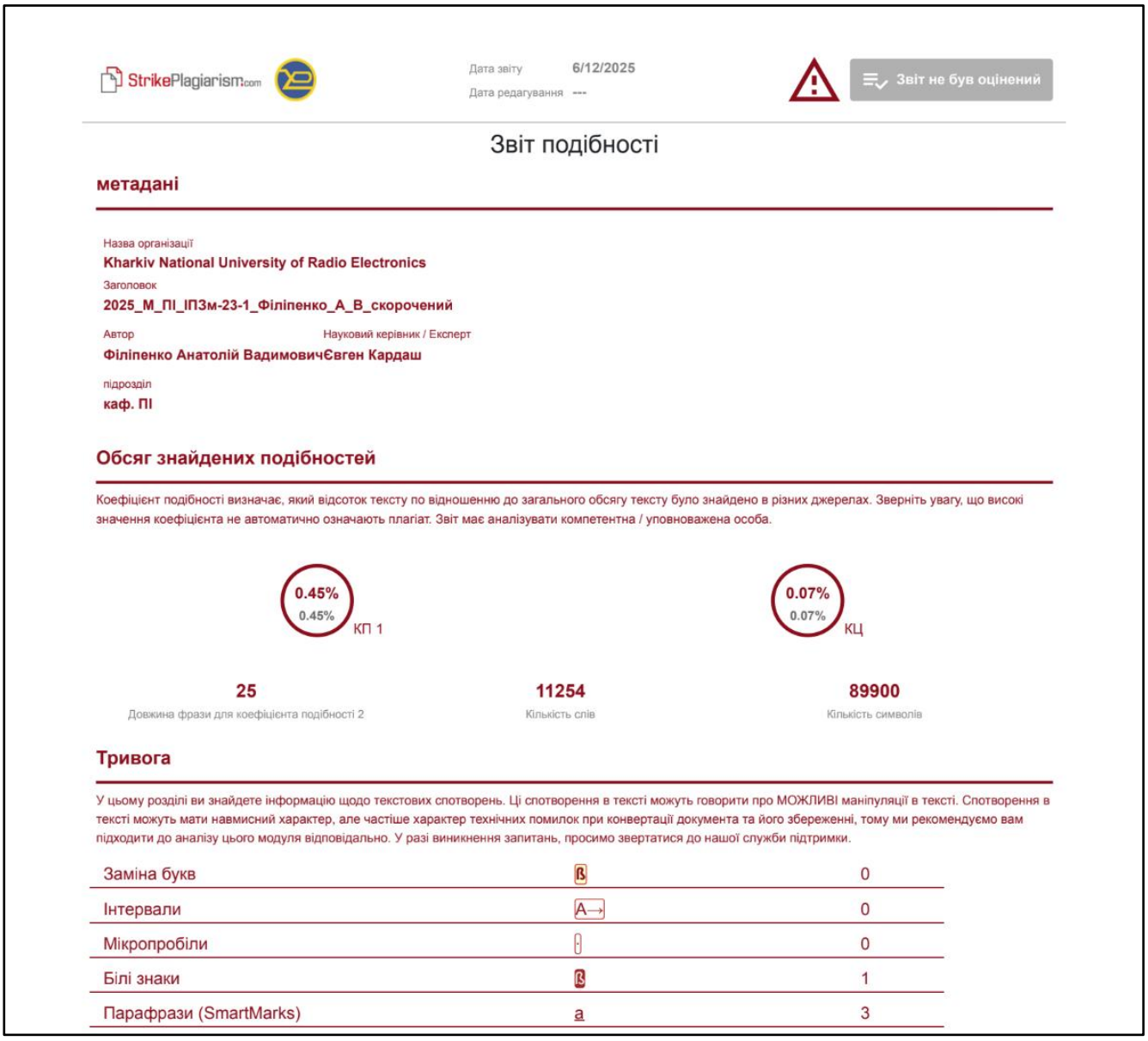
ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ



StrikePlagiarism.com

Дата звіту 6/12/2025
Дата редагування ---

Звіт не був оцінений

Звіт подібності

метадані

Назва організації
Kharkiv National University of Radio Electronics
Заголовок
2025_М_ПІ_ІПЗм-23-1_Філіпенко_А_В_скорочений
Автор
Філіпенко Анатолій Вадимович
Науковий керівник / Експерт
Євген Кардаш
підрозділ
каф. ПІ

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

0.45% 0.45% КП 1

0.07% 0.07% КЦ

25	11254	89900
Довжина фрази для коефіцієнта подібності 2	Кількість слів	Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли		0
Білі знаки		1
Парафрази (SmartMarks)	a	3

Рисунок А. 1 – Результати перевірки на унікальність тексту(рисунок виконаний самостійно)

ДОДАТОК Б

Слайди презентації

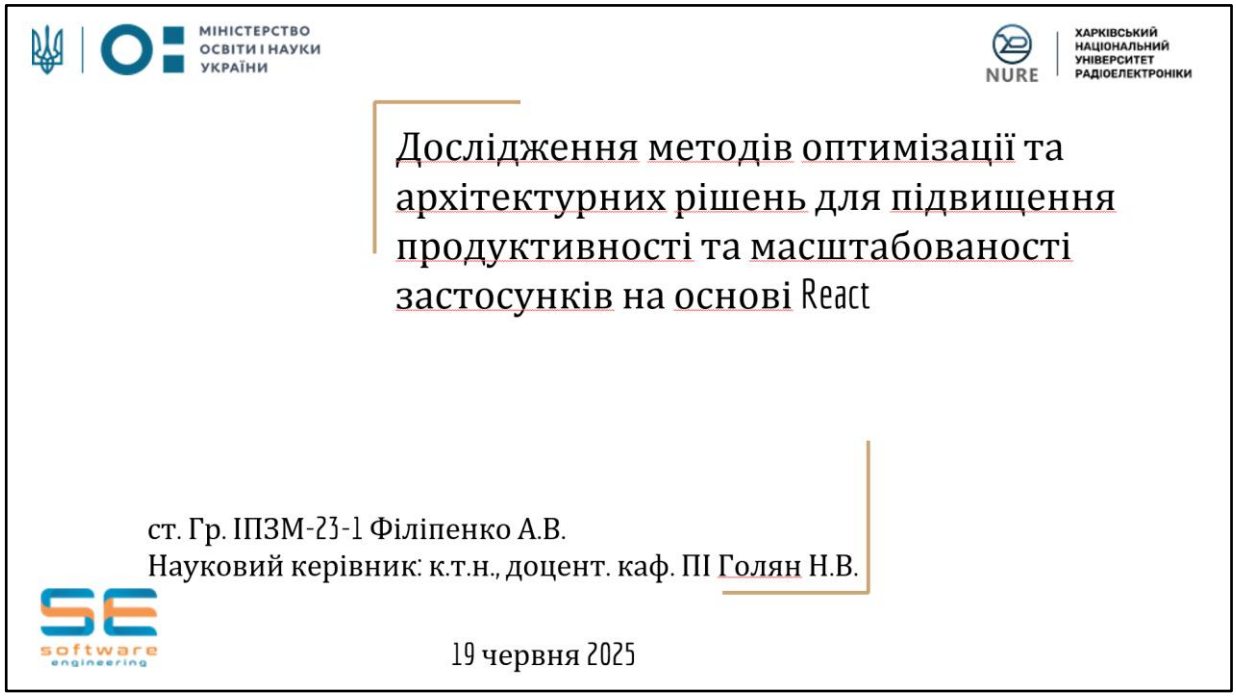


Рисунок А. 1 – 1 слайд (рисунок виконаний самостійно)

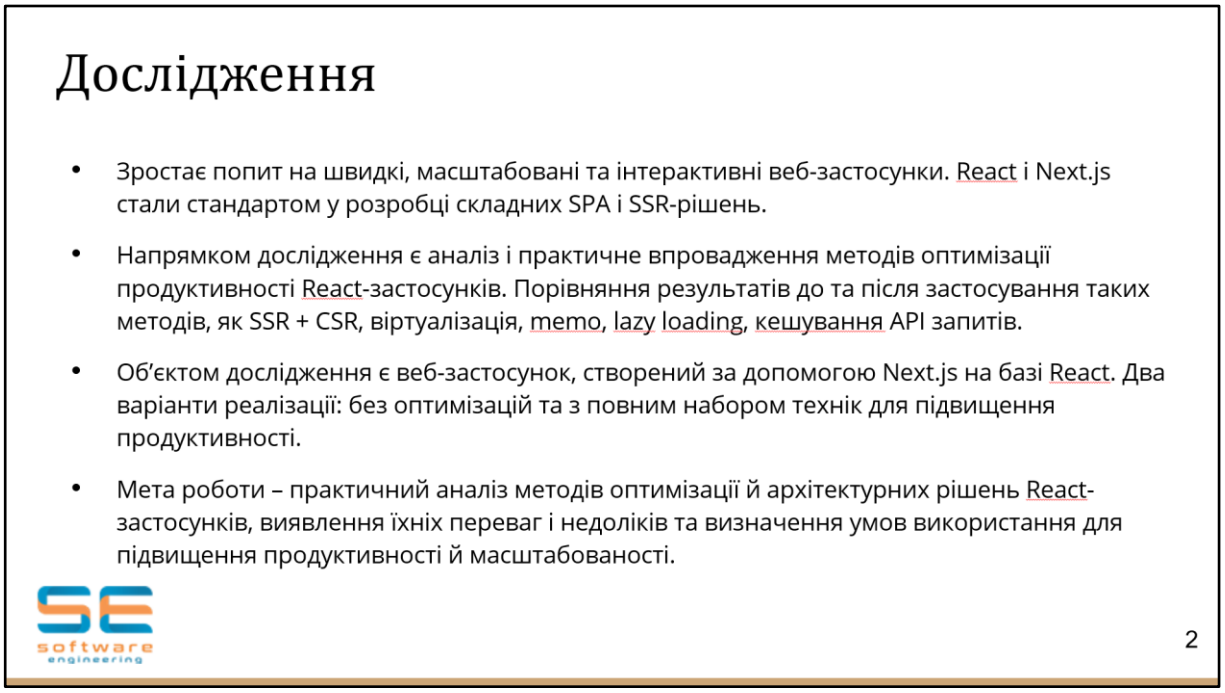


Рисунок А. 2 – 2 слайд (рисунок виконаний самостійно)

Огляд літератури (аналогів)

- Основні джерела: офіційна документація React і Next.js, статті з технічних блогів.
- O. I. Bezverhiy and O. I. Kutsenko, «Шляхи оптимізації кросплатформенних додатків із використанням бібліотеки React та фреймворку React Native», стр. 30 – 35, 2024, doi: 10.32782/2521-6643-2024-1-67.5.
- Виявлені прогалини: відсутність production-кейсів, кількісних метрик FCP/LCP/TTI до/після оптимізації та глибинного SSR/SSG-аналізу.

Рисунок А. 3 – 3 слайд (рисунок виконаний самостійно)

Постановка задачі

Задача полягає у систематичному дослідженні, порівнянні та застосуванні ефективних методів оптимізації й архітектурних рішень для застосунків на основі React з метою підвищення їхньої продуктивності та поліпшення користувацького досвіду. На практичному рівні передбачається реалізація прототипу застосунку, побудованого на базі React з використанням фреймворку Next.js, а також створення його альтернативної версії без оптимізаційних покращень. Подальше порівняння отриманих результатів дозволить визначити ефективність застосованих підходів.

Рисунок А. 4 – 4 слайд (рисунок виконаний самостійно)

Методологія

Експериментальний дизайн: два прототипи: контрольний та оптимізований.

Операційні змінні: Показники Web Vitals (FCP, LCP, TTI, TBT).

Умови та середовище: Chrome Lighthouse в режимі «Incognito», стабільне з'єднання.

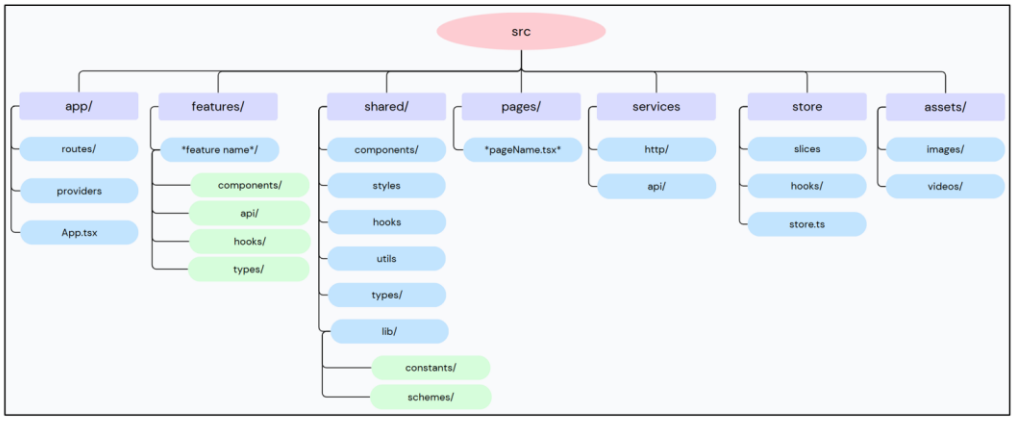
Процедура вимірювань: Три прогони для кожної конфігурації, отримання усереднення результатів.

Методи обробки даних: Статистичне усереднення, порівняльний аналіз.

5

Рисунок А. 5 – 5 слайд (рисунок виконаний самостійно)

Архітектура система для проведення експериментального дослідження



6

Рисунок А. 6 – 6 слайд (рисунок виконаний самостійно)

Опис програмного забезпечення, що було використано у дослідженні

- **Опис процесу розробки:** підготовка тестових даних, створення двох версій додатку (базова та оптимізована), інтеграція інструментів вимірювання продуктивності.
- **Вибрані мови, фреймворки та бібліотеки:** TypeScript, React 19, Redux, Next.js 15, react-window, TanStack Query.

7

Рисунок А. 7 – 7 слайд (рисунок виконаний самостійно)

Зміст проведеного експерименту

Методи оптимізації:

- Віртуалізація списків (react-window).
- Мемоізація компонентів (React.memo, useMemo, useCallback).
- Динамічний імпорт компонентів (Next.js dynamic).
- Lazy loading зображень.
- Кешування API-запитів (TanStack Query).
- Використання SSR та CSR.

Вхідні дані:

- Списки зі 200+ продуктів та 40+ різних зображень.
- Штучні API-відповіді зі списками користувачів до 225-ти елементів.

8

Рисунок А. 8 – 8 слайд (рисунок виконаний самостійно)

Зміст проведеного експерименту

Критерії оцінки:

- FCP (First Contentful Paint).
- LCP (Largest Contentful Paint).
- TTI (Time To Interactive).
- Загальний бал продуктивності за Lighthouse.

Послідовність:

- Створення просто серверного застосунку для імітації отримання даних.
- Контрольний прототип (без оптимізацій).
- Послідовне застосування кожного методу.
- Фінальна версія прототипу(з усіма оптимізаціями).

Вимірювання:

- Chrome Lighthouse у режимі «Incognito».
- Три запуску для кожної конфігурації для дестопної та мобільної версій.
- Отримання усередненого результату який зустрічається найчастіше.

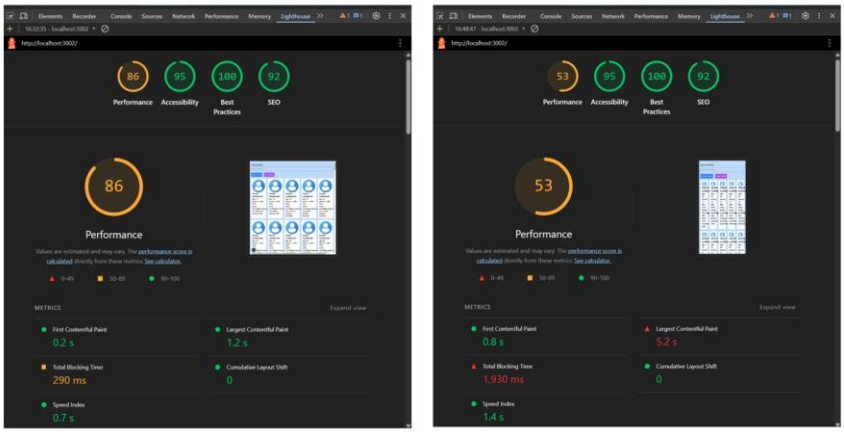
software engineering

9

Рисунок А. 9 – 9 слайд (рисунок виконаний самостійно)

Результати експерименту

Неоптимізована версія



Desktop device

Mobile Device

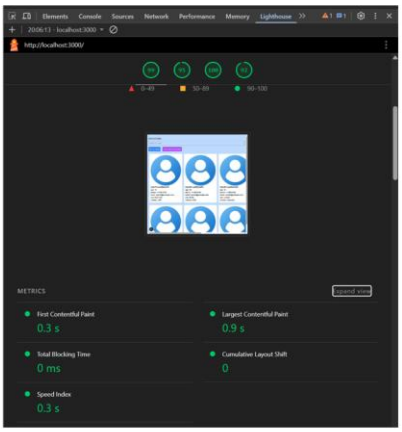
software engineering

10

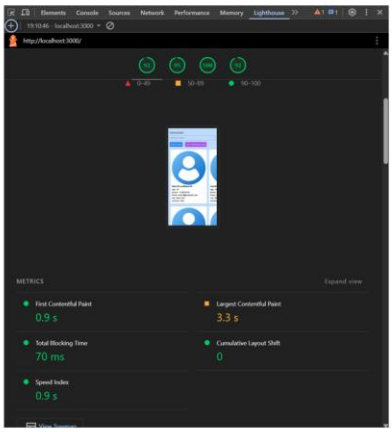
Рисунок А. 10 – 10 слайд (рисунок виконаний самостійно)

Результати експерименту

Оптимізована фінальна версія



Desktop device



Mobile Device

11

Рисунок А. 11 – 11 слайд (рисунок виконаний самостійно)

Результати експерименту

Версія	Score	FCP (First Contentful Paint)	Speed Index	LCP (Largest Contentful Paint)	TBT (Total Blocking Time)
Неоптимізована (десктоп)	86	0.2 s	0.7 s	1.2 s	290 ms
Неоптимізована (мобільна)	53	0.8 s	1.4 s	5.2 s	1930 ms
Оптимізована SSR + мемо (десктоп)	98	0.3 s	1.1 s	1.0 s	50 ms
Оптимізована SSR + мемо (мобільна)	88	0.9 s	1.9 s	3.3 s	230 ms
+ TanStack Query (десктоп)	99	0.2 s	0.7 s	0.9 s	10 ms
+ TanStack Query (мобільна)	89	0.9 s	1.8 s	3.6 s	130 ms
+ Віртуалізація списків (десктоп)	99	0.3 s	0.3 s	0.9 s	0 ms
+ Віртуалізація списків (мобільна)	92	0.9 s	0.9 s	3.3 s	70 ms

12

Рисунок А. 12 – 12 слайд (рисунок виконаний самостійно)

Аналіз отриманих результатів

- Базова версія дала хороші десктоп-показники, але на мобільному Score був критично низьким через високий LCP і TBT.
- Додавання SSR + memo підвищило Score обох версій, особливо мобільної, та значно зменшило затримки рендеру.
- Кешування API через TanStack Query дало змогу миттєво отримувати та відображати дані при здійсненні навігації між сторінками.
- Віртуалізація списків практично усунула блокування основного потоку (TBT ≈ 0).
- Lazy loading допоміг усунути завантаження відразу усіх зображень.
- Оптимальним виявилося комбіноване застосування усіх методів.

software engineering

13

Рисунок А. 13 – 13 слайд (рисунок виконаний самостійно)

Публікація результатів

software engineering

14

Рисунок А. 14 – 14 слайд (рисунок виконаний самостійно)

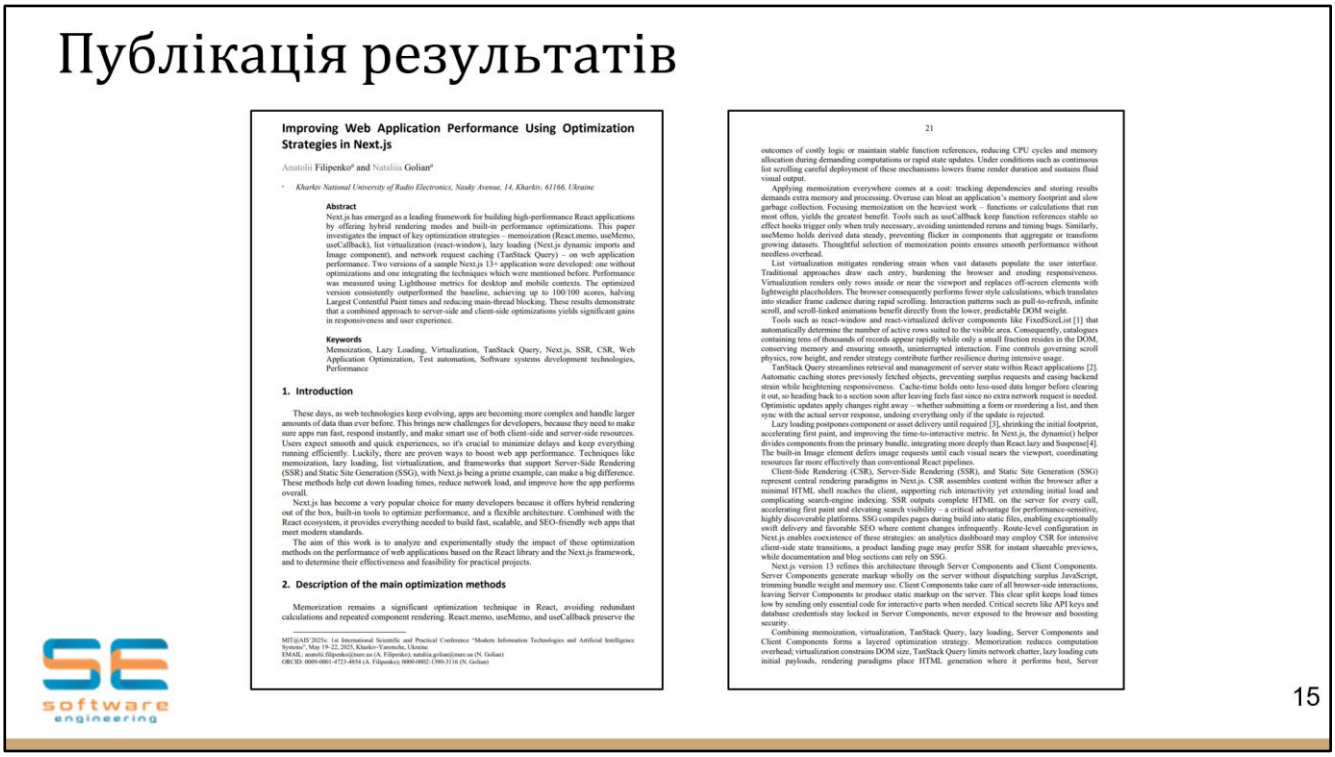


Рисунок А. 15 – 15 слайд (рисунок виконаний самостійно)

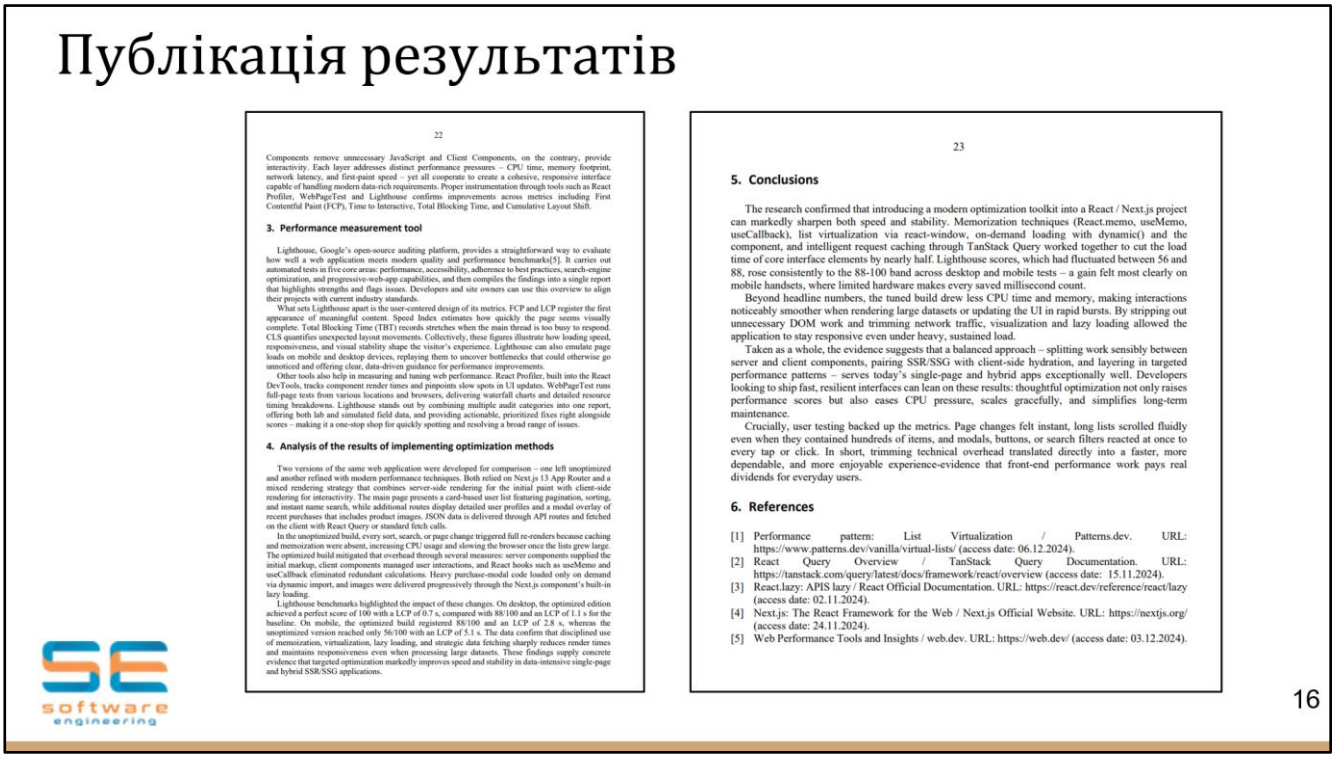


Рисунок А. 16 – 16 слайд (рисунок виконаний самостійно)

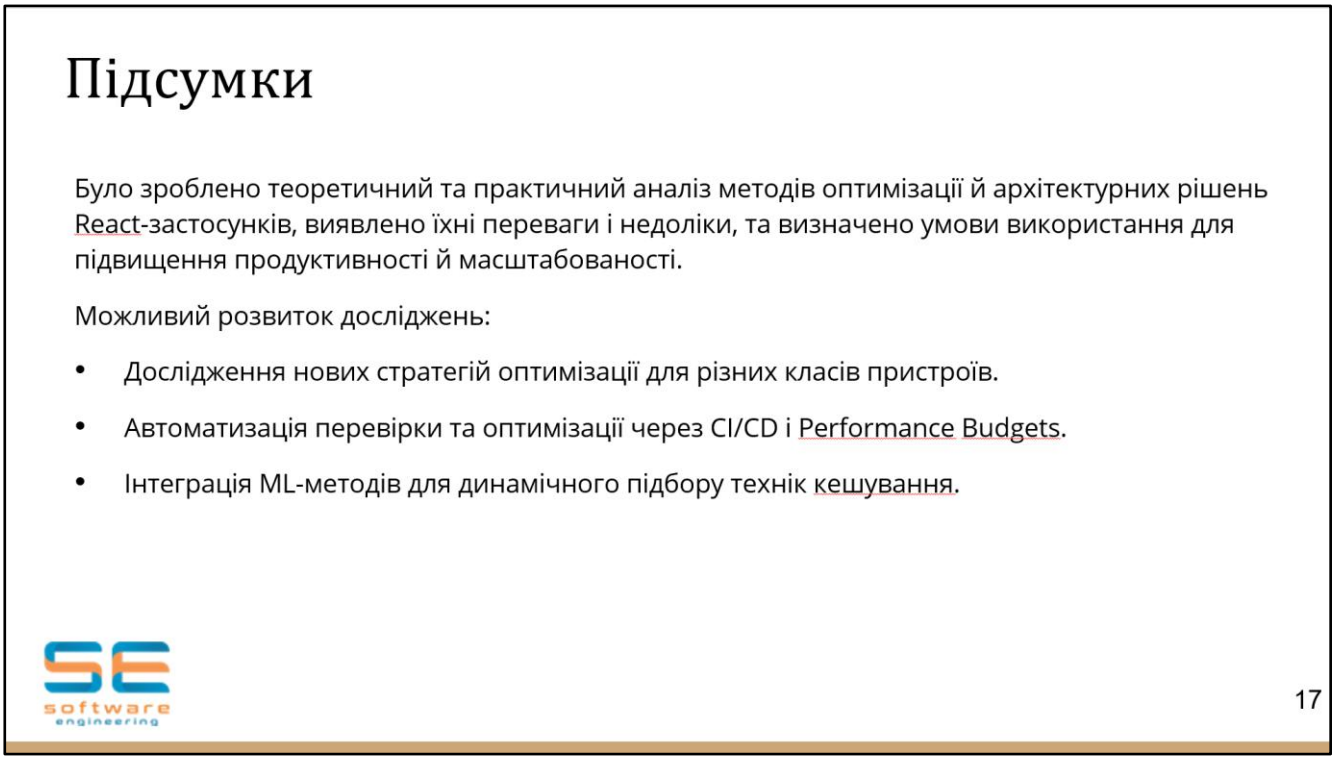


Рисунок А. 17 – 17 слайд (рисунок виконаний самостійно)

ДОДАТОК В

Апробація результатів роботи на конференції «1 Міжнародна науково-практична конференція «Modern Information Technologies and Artificial Intelligent Systems MIT@AIS-2025»

Improving Web Application Performance Using Optimization Strategies in Next.js

Anatolii Filipenko[a] and Nataliia Golian[a]

[a] *Kharkiv National University of Radio Electronics, Nauky Avenue, 14, Kharkiv, 61166, Ukraine*

Abstract

Next.js has emerged as a leading framework for building high-performance React applications by offering hybrid rendering modes and built-in performance optimizations. This paper investigates the impact of key optimization strategies – memoization (React.memo, useMemo, useCallback), list virtualization (react-window), lazy loading (Next.js dynamic imports and Image component), and network request caching (TanStack Query) – on web application performance. Two versions of a sample Next.js 13+ application were developed: one without optimizations and one integrating the techniques which were mentioned before. Performance was measured using Lighthouse metrics for desktop and mobile contexts. The optimized version consistently outperformed the baseline, achieving up to 100/100 scores, halving Largest Contentful Paint times and reducing main-thread blocking. These results demonstrate that a combined approach to server-side and client-side optimizations yields significant gains in responsiveness and user experience.

Keywords

Memoization, Lazy Loading, Virtualization, TanStack Query, Next.js, SSR, CSR, Web Application Optimization, Test automation, Software systems development technologies, Performance

1. Introduction

These days, as web technologies keep evolving, apps are becoming more complex and handle larger amounts of data than ever before. This brings new challenges for developers, because they need to make sure apps run fast, respond instantly, and make smart use of both client-side and server-side resources. Users expect smooth and quick experiences, so it's crucial to minimize delays and keep everything running efficiently. Luckily, there are proven ways to boost web app performance. Techniques like memoization, lazy loading, list virtualization, and frameworks that support Server-Side Rendering (SSR) and Static Site Generation (SSG), with Next.js being a prime example, can make a big difference. These methods help cut down loading times, reduce network load, and improve how the app performs overall.

Next.js has become a very popular choice for many developers because it offers hybrid rendering out of the box, built-in tools to optimize performance, and a flexible architecture. Combined with the React ecosystem, it provides everything needed to build fast, scalable, and SEO-friendly web apps that meet modern standards.

The aim of this work is to analyze and experimentally study the impact of these optimization methods on the performance of web applications based on the React library and the Next.js framework, and to determine their effectiveness and feasibility for practical projects.

2. Description of the main optimization methods

Memorization remains a significant optimization technique in React, avoiding redundant calculations and repeated component rendering. React.memo, useMemo, and useCallback preserve the

MIT@AIS'2025s: 1st International Scientific and Practical Conference "Modern Information Technologies and Artificial Intelligence Systems", May 19–22, 2025, Kharkiv-Yaremche, Ukraine
EMAIL: anatolii.filipenko@nure.ua (A. Filipenko); nataliia.golian@nure.ua (N. Golian)
ORCID: 0009-0001-4723-4854 (A. Filipenko); 0000-0002-1390-3116 (N. Golian)

Рисунок В.1 – Перша сторінка статті (рисунок виконаний самостійно)

outcomes of costly logic or maintain stable function references, reducing CPU cycles and memory allocation during demanding computations or rapid state updates. Under conditions such as continuous list scrolling careful deployment of these mechanisms lowers frame render duration and sustains fluid visual output.

Applying memoization everywhere comes at a cost: tracking dependencies and storing results demands extra memory and processing. Overuse can bloat an application's memory footprint and slow garbage collection. Focusing memoization on the heaviest work – functions or calculations that run most often, yields the greatest benefit. Tools such as useCallback keep function references stable so effect hooks trigger only when truly necessary, avoiding unintended reruns and timing bugs. Similarly, useMemo holds derived data steady, preventing flicker in components that aggregate or transform growing datasets. Thoughtful selection of memoization points ensures smooth performance without needless overhead.

List virtualization mitigates rendering strain when vast datasets populate the user interface. Traditional approaches draw each entry, burdening the browser and eroding responsiveness. Virtualization renders only rows inside or near the viewport and replaces off-screen elements with lightweight placeholders. The browser consequently performs fewer style calculations, which translates into steadier frame cadence during rapid scrolling. Interaction patterns such as pull-to-refresh, infinite scroll, and scroll-linked animations benefit directly from the lower, predictable DOM weight.

Tools such as react-window and react-virtualized deliver components like FixedSizeList [1] that automatically determine the number of active rows suited to the visible area. Consequently, catalogues containing tens of thousands of records appear rapidly while only a small fraction resides in the DOM, conserving memory and ensuring smooth, uninterrupted interaction. Fine controls governing scroll physics, row height, and render strategy contribute further resilience during intensive usage.

TanStack Query streamlines retrieval and management of server state within React applications [2]. Automatic caching stores previously fetched objects, preventing surplus requests and easing backend strain while heightening responsiveness. Cache-time holds onto less-used data longer before clearing it out, so heading back to a section soon after leaving feels fast since no extra network request is needed. Optimistic updates apply changes right away – whether submitting a form or reordering a list, and then sync with the actual server response, undoing everything only if the update is rejected.

Lazy loading postpones component or asset delivery until required [3], shrinking the initial footprint, accelerating first paint, and improving the time-to-interactive metric. In Next.js, the dynamic() helper divides components from the primary bundle, integrating more deeply than React.lazy and Suspense[4]. The built-in Image element defers image requests until each visual nears the viewport, coordinating resources far more effectively than conventional React pipelines.

Client-Side Rendering (CSR), Server-Side Rendering (SSR), and Static Site Generation (SSG) represent central rendering paradigms in Next.js. CSR assembles content within the browser after a minimal HTML shell reaches the client, supporting rich interactivity yet extending initial load and complicating search-engine indexing. SSR outputs complete HTML on the server for every call, accelerating first paint and elevating search visibility – a critical advantage for performance-sensitive, highly discoverable platforms. SSG compiles pages during build into static files, enabling exceptionally swift delivery and favorable SEO where content changes infrequently. Route-level configuration in Next.js enables coexistence of these strategies: an analytics dashboard may employ CSR for intensive client-side state transitions, a product landing page may prefer SSR for instant shareable previews, while documentation and blog sections can rely on SSG.

Next.js version 13 refines this architecture through Server Components and Client Components. Server Components generate markup wholly on the server without dispatching surplus JavaScript, trimming bundle weight and memory use. Client Components take care of all browser-side interactions, leaving Server Components to produce static markup on the server. This clear split keeps load times low by sending only essential code for interactive parts when needed. Critical secrets like API keys and database credentials stay locked in Server Components, never exposed to the browser and boosting security.

Combining memoization, virtualization, TanStack Query, lazy loading, Server Components and Client Components forms a layered optimization strategy. Memorization reduces computation overhead; virtualization constrains DOM size, TanStack Query limits network chatter, lazy loading cuts initial payloads, rendering paradigms place HTML generation where it performs best, Server

Рисунок В.2 – Друга сторінка статті (рисунок виконаний самостійно)

Components remove unnecessary JavaScript and Client Components, on the contrary, provide interactivity. Each layer addresses distinct performance pressures – CPU time, memory footprint, network latency, and first-paint speed – yet all cooperate to create a cohesive, responsive interface capable of handling modern data-rich requirements. Proper instrumentation through tools such as React Profiler, WebPageTest and Lighthouse confirms improvements across metrics including First Contentful Paint (FCP), Time to Interactive, Total Blocking Time, and Cumulative Layout Shift.

3. Performance measurement tool

Lighthouse, Google's open-source auditing platform, provides a straightforward way to evaluate how well a web application meets modern quality and performance benchmarks[5]. It carries out automated tests in five core areas: performance, accessibility, adherence to best practices, search-engine optimization, and progressive-web-app capabilities, and then compiles the findings into a single report that highlights strengths and flags issues. Developers and site owners can use this overview to align their projects with current industry standards.

What sets Lighthouse apart is the user-centered design of its metrics. FCP and LCP register the first appearance of meaningful content. Speed Index estimates how quickly the page seems visually complete. Total Blocking Time (TBT) records stretches when the main thread is too busy to respond. CLS quantifies unexpected layout movements. Collectively, these figures illustrate how loading speed, responsiveness, and visual stability shape the visitor's experience. Lighthouse can also emulate page loads on mobile and desktop devices, replaying them to uncover bottlenecks that could otherwise go unnoticed and offering clear, data-driven guidance for performance improvements.

Other tools also help in measuring and tuning web performance. React Profiler, built into the React DevTools, tracks component render times and pinpoints slow spots in UI updates. WebPageTest runs full-page tests from various locations and browsers, delivering waterfall charts and detailed resource timing breakdowns. Lighthouse stands out by combining multiple audit categories into one report, offering both lab and simulated field data, and providing actionable, prioritized fixes right alongside scores – making it a one-stop shop for quickly spotting and resolving a broad range of issues.

4. Analysis of the results of implementing optimization methods

Two versions of the same web application were developed for comparison – one left unoptimized and another refined with modern performance techniques. Both relied on Next.js 13 App Router and a mixed rendering strategy that combines server-side rendering for the initial paint with client-side rendering for interactivity. The main page presents a card-based user list featuring pagination, sorting, and instant name search, while additional routes display detailed user profiles and a modal overlay of recent purchases that includes product images. JSON data is delivered through API routes and fetched on the client with React Query or standard fetch calls.

In the unoptimized build, every sort, search, or page change triggered full re-renders because caching and memoization were absent, increasing CPU usage and slowing the browser once the lists grew large. The optimized build mitigated that overhead through several measures: server components supplied the initial markup, client components managed user interactions, and React hooks such as useMemo and useCallback eliminated redundant calculations. Heavy purchase-modal code loaded only on demand via dynamic import, and images were delivered progressively through the Next.js component's built-in lazy loading.

Lighthouse benchmarks highlighted the impact of these changes. On desktop, the optimized edition achieved a perfect score of 100 with a LCP of 0.7 s, compared with 88/100 and an LCP of 1.1 s for the baseline. On mobile, the optimized build registered 88/100 and an LCP of 2.8 s, whereas the unoptimized version reached only 56/100 with an LCP of 5.1 s. The data confirm that disciplined use of memoization, virtualization, lazy loading, and strategic data fetching sharply reduces render times and maintains responsiveness even when processing large datasets. These findings supply concrete evidence that targeted optimization markedly improves speed and stability in data-intensive single-page and hybrid SSR/SSG applications.

Рисунок В.3 – Третя сторінка статті (рисунок виконаний самостійно)

5. Conclusions

The research confirmed that introducing a modern optimization toolkit into a React / Next.js project can markedly sharpen both speed and stability. Memorization techniques (React.memo, useMemo, useCallback), list virtualization via react-window, on-demand loading with dynamic() and the component, and intelligent request caching through TanStack Query worked together to cut the load time of core interface elements by nearly half. Lighthouse scores, which had fluctuated between 56 and 88, rose consistently to the 88-100 band across desktop and mobile tests – a gain felt most clearly on mobile handsets, where limited hardware makes every saved millisecond count.

Beyond headline numbers, the tuned build drew less CPU time and memory, making interactions noticeably smoother when rendering large datasets or updating the UI in rapid bursts. By stripping out unnecessary DOM work and trimming network traffic, visualization and lazy loading allowed the application to stay responsive even under heavy, sustained load.

Taken as a whole, the evidence suggests that a balanced approach – splitting work sensibly between server and client components, pairing SSR/SSG with client-side hydration, and layering in targeted performance patterns – serves today's single-page and hybrid apps exceptionally well. Developers looking to ship fast, resilient interfaces can lean on these results: thoughtful optimization not only raises performance scores but also eases CPU pressure, scales gracefully, and simplifies long-term maintenance.

Crucially, user testing backed up the metrics. Page changes felt instant, long lists scrolled fluidly even when they contained hundreds of items, and modals, buttons, or search filters reacted at once to every tap or click. In short, trimming technical overhead translated directly into a faster, more dependable, and more enjoyable experience-evidence that front-end performance work pays real dividends for everyday users.

6. References

[1] Performance pattern: List Virtualization / Patterns.dev. URL: https://www.patterns.dev/vanilla/virtual-lists/ (access date: 06.12.2024).
[2] React Query Overview / TanStack Query Documentation. URL: https://tanstack.com/query/latest/docs/framework/react/overview (access date: 15.11.2024).
[3] React.lazy: APIS lazy / React Official Documentation. URL: https://react.dev/reference/react/lazy (access date: 02.11.2024).
[4] Next.js: The React Framework for the Web / Next.js Official Website. URL: https://nextjs.org/ (access date: 24.11.2024).
[5] Web Performance Tools and Insights / web.dev. URL: https://web.dev/ (access date: 03.12.2024).

Рисунок В.4 – Четверта сторінка статті (рисунок виконаний самостійно)

ДОДАТОК Г

Експертний висновок результатів перевірки кваліфікаційної роботи на відповідність оформлення вимогам ДСТУ 3008: 2015

Експертний висновок результатів перевірки кваліфікаційної роботи

студент (посада) | програмної інженерії (кафедра) | ІПЗм-23-1□ (група)

Анатолій ФІЛІПЕНКО
(прізвище, ім'я, по батькові)

Зауваження

Пункт ДСТУ 3008-2015	Зміст пункту	Сторінка кваліфікаційної роботи
1	2	3
	7.1 Загальні положення	
7.1.25	Не дозволено розміщувати назву розділу, підрозділу, а також пункту й підпункту на останньому рядку сторінки.	31
	7.3 Нумерація сторінок звіту	
	7.5 Рисунки	
	7.6 Таблиці	
	7.7 Переліки	
	7.8 Примітки	
	7.9 Виноски	
	7.10 Формули та рівняння	
	7.11 Посилання	
	7.13 Список авторів	
	7.14 Скорочення та умовні познаки	
	7.15 Додатки	

Експерт _____ (підпис) Вадим НЕЧВОЛОД (прізвище, ініціали)

14.06.2025

Рисунок Г.1 – Експертний висновок результатів перевірки кваліфікаційної роботи (рисунок виконаний самостійно)