International Science Journal of Engineering & Agriculture
2026; 5(4): 69-80
https://isg-journal.com/isjea
doi: 10.46299/j.isjea.20260504.06
ISSN: 2720-6319

INTERNATIONAL
SCIENCE GROUP

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

Ігор Андрущак
Луцький національний технічний університет, Луцьк, Україна
ORCID: 0000-0002-8751-4420

Ілля Пуршак
Луцький національний технічний університет, Луцьк, Україна
ORCID: 0009-0007-0407-2625

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 рендер при підвантаженні даних. Встановлено, що агрегований бал Lighthouse Performance сто із ста збігається 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 покликане 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 сто 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 «теплому» 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 мережевих 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 мережевих затримок між сервісами; підготувати базу даних у «теплому» 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 документаційні 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-а, 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.

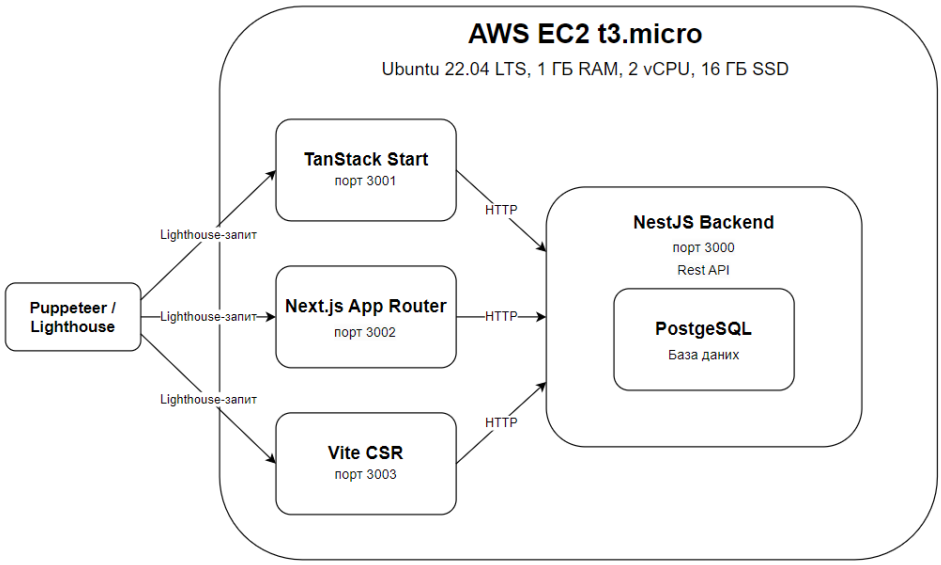

Рис. 1. Структура тестового стенду

Шість зібраних метрик не є довільним набором – кожна фіксує конкретну точку в конвеєрі 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 сто 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):

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

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):

$$\Delta = \frac{M_A - M_B}{M_B} \times 100\%, \quad (2)$$

де M_A, M_B – середні значення однойменної метрики фреймворків A та B відповідно.

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

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

Фреймвок	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 «теплому» 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.

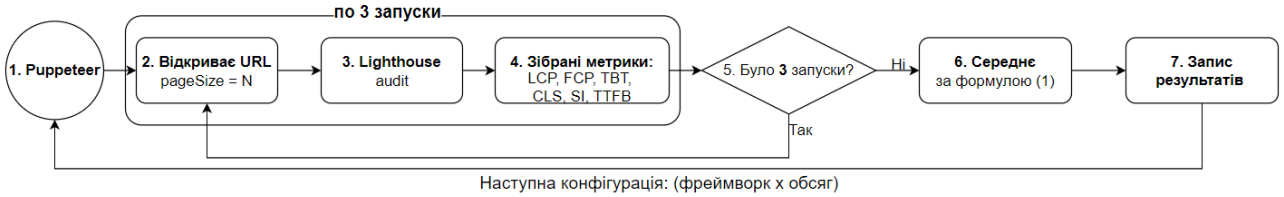

Рис. 2. Конвеєр автоматизованого вимірювання метрик

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.

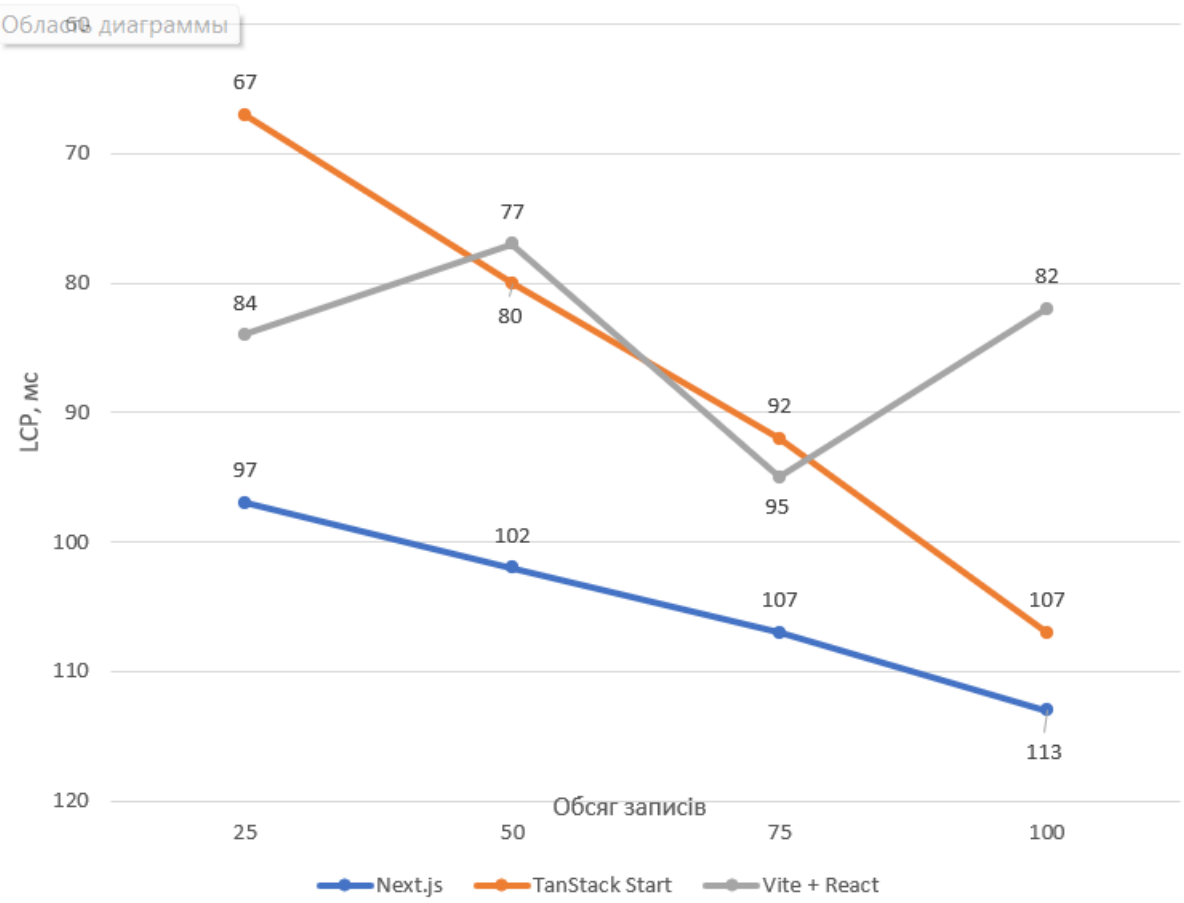

Рис. 3. LCP залежно від обсягу даних

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-і; 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 сто із ста збіглася 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 звужується 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 проєкту 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 записах; 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-стримінгом.

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

Illia Purshak
Lutsk National Technical University, Lutsk, Ukraine
ORCID: 0009-0007-0407-2625

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.