Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Допущено до захисту
Завідувач кафедри
комп’ютерних наук,
доктор технічних наук, професор

Андрій БОНДАРЧУК
«01» червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»
Спеціальність 122 Комп’ютерні науки
Освітня програма 122.00.01 Інформатика

Тема: ДОСЛІДЖЕННЯ ПРОДУКТИВНОСТІ FRONTEND-ФРЕЙМВОРКІВ НА ПРИКЛАДІ РЕАЛЬНОГО ПРОЄКТУ

Виконав
Студентка групи ІНб-2-22-4.0д
Довгаль Марія Іванівна
_____
(підпис)

Науковий керівник
доктор технічних наук, професор
Бондарчук А.П
_____
(підпис)

Київ – 2026
Київський столичний університет імені Бориса Грінченка

Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп’ютерних наук,
доктор технічних наук, професор
Андрій БОНДАРЧУК
« 31 » жовтня 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА
«Дослідження продуктивності frontend-фреймворків на прикладі реального проєкту»

Виконавець – студентка групи ІНб-2-22-4.0д,
спеціальності 122 Комп’ютерні науки,
освітньої програми 122.00.01 Інформатика
Довгаль Марія Іванівна

1. Вихідні дані: відкритий проєкт framework-benchmarks, документація React, Angular, Vue, Svelte, Solid, Qwik, Preact, Lit, Alpine.js, jQuery, Vanilla JavaScript, Web Vitals, Lighthouse, DevTools Performance та інші джерела.
2. Основні завдання: проаналізувати поняття frontend-розробки; охарактеризувати сучасні frontend-фреймворки; визначити метрики продуктивності веб-інтерфейсу; описати структуру benchmark-проєкту; провести порівняльний аналіз результатів; сформувати рекомендації щодо вибору фреймворку.
3. Пояснювальна записка: Обсяг 52 сторінок, формату А4 комп’ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.
4. Графічні матеріали: Комп’ютерна презентація доповіді.
5. Додатки: Додаток А. Структура benchmark-проєкту framework-benchmarks. Додаток Б. Зведена таблиця метрик продуктивності веб-інтерфейсу.
6. Строк подання роботи на кафедру: «31» червня 2026 р.

Науковий керівник
доктор технічних наук, професор
_____БОНДАРЧУК А.П.
_____ дата

Виконавець:
_____ ДОВГАЛЬ М.І.
_____ дата

Календарний план

№	Етап виконання роботи	Термін виконання	Примітка
1	Затвердження теми кваліфікаційної роботи	31.10.25	виконано
2	Аналіз проблеми, вивчення аналогів, формування цілей і завдань	01.11.25 – 11.01.26	виконано
3	Аналіз архітектурних підходів до frontend-розробки та метрик продуктивності Web Vitals	26.01.26 – 15.03.26	виконано
4	Дослідження benchmark-платформи framework-benchmarks та підготовка експериментального середовища	16.03.26 – 31.03.26	виконано
5	Проведення порівняльного тестування 12 frontend-фреймворків та збір результатів вимірювань	01.04.26 – 15.04.26	виконано
6	Аналіз результатів тестування та формування практичних рекомендацій щодо вибору frontend-стеку	16.04.26 – 30.04.26	виконано
7	Впровадження результатів та оцінка практичної цінності дослідження	01.05.26 - 15.05.26	виконано
8	Підготовка пояснювальної записки, графічних матеріалів, додатків	25.05.25 – 12.06.26	виконано
9	Захист кваліфікаційної роботи	19.06.26	

«Затверджую»
Науковий керівник
доктор технічних наук, професор
Бондарчук А.П.

Індивідуальний план склав
Довгаль М.І.

РЕФЕРАТ

Кваліфікаційна робота: 52 с., 10 рис., 15 табл., 50 посилань.

Актуальність теми визначається зростаючою роллю клієнтської частини у сучасних веб-застосунках. Банківські кабінети, адміністративні панелі, освітні платформи та аналітичні дашборди давно перестали бути простим відображенням серверних даних. Від того, яку технологію обере розробник, залежать швидкість інтерфейсу, обсяг JavaScript-коду та загальна якість користувацького досвіду. Єдиного академічно обґрунтованого порівняння frontend-фреймворків в ідентичних умовах досі бракує, що зумовлює необхідність цього емпіричного дослідження [1, 2].

Об'єктом дослідження є процес оцінювання продуктивності клієнтської частини сучасних веб-застосунків. Предметом дослідження виступають frontend-фреймворки та бібліотеки: React, Angular, Vue, Svelte, Solid, Qwik, Preact, Lit, Alpine.js, JQuery, VanJS, Vanilla JS та їхні архітектурні особливості і метрики продуктивності. Метою роботи є порівняльний аналіз продуктивності зазначених frontend-рішень на основі єдиної експериментальної платформи та формування практичних рекомендацій щодо вибору технологічного стеку залежно від типу проєкту.

У ході дослідження порівнювались показники First Contentful Paint, Largest Contentful Paint, Lighthouse Performance Score, розмір gzipped-бандлу та час production-збірки. Найвищу загальну продуктивність продемонстрували VanJS і Solid (FCP у межах 1,2-1,4 с, Lighthouse 100/100). Vue та Svelte показали найкращий баланс між швидкодією та зручністю розробки. React зберігає лідерство за широтою екосистеми, Angular забезпечує архітектурну дисципліну для корпоративних систем, а Qwik виявився єдиним фреймворком з Lighthouse-балом нижче 80 [45, 46].

Ключові слова: frontend, фреймворк, React, Angular, Vue, Svelte, Solid, Web Vitals, продуктивність, benchmark.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API (Application Programming Interface) – програмний інтерфейс застосунку; набір правил і протоколів для взаємодії між програмними компонентами.

AOT (Ahead-of-Time) – компіляція шаблонів і коду до запуску застосунку, на етапі збірки.

CDN (Content Delivery Network) – мережа доставки контенту; географічно розподілена інфраструктура для швидкої передачі статичних ресурсів.

CI/CD (Continuous Integration / Continuous Delivery) – практики безперервної інтеграції та безперервного постачання програмного забезпечення.

CLS (Cumulative Layout Shift) – накопичений зсув макета; метрика стабільності візуального відображення сторінки.

CPU (Central Processing Unit) – центральний процесор.

CSR (Client-Side Rendering) – рендеринг на стороні клієнта; формування HTML-вмісту безпосередньо у браузері засобами JavaScript.

CSS (Cascading Style Sheets) – каскадні таблиці стилів; мова опису зовнішнього вигляду HTML-документів.

DOM (Document Object Model) – об'єктна модель документа; програмний інтерфейс для роботи зі структурою HTML-сторінки.

DI (Dependency Injection) – впровадження залежностей; патерн проєктування для управління залежностями між компонентами.

FCP (First Contentful Paint) – час до першого відображення будь-якого контенту на сторінці.

FID (First Input Delay) – затримка першого введення; час між першою дією користувача і реакцією браузера (замінена метрикою INP у 2024 році).

HTML (HyperText Markup Language) – мова розмітки гіпертексту; стандартна мова для створення структури веб-сторінок.

HTTP (HyperText Transfer Protocol) – протокол передачі гіпертексту; основний протокол обміну даними у вебі.

INP (Interaction to Next Paint) – час від взаємодії користувача до наступного відображення кадру.

JIT (Just-in-Time) – компіляція коду безпосередньо під час виконання програми.

JS (JavaScript) – скриптова мова програмування, що виконується у браузері та на сервері.

JSON (JavaScript Object Notation) – текстовий формат обміну даними на основі синтаксису JavaScript.

JSX (JavaScript XML) – синтаксичне розширення JavaScript, що дозволяє описувати структуру інтерфейсу у вигляді XML-подібних конструкцій.

КБ – кілобайт; одиниця вимірювання обсягу цифрових даних.

LCP (Largest Contentful Paint) – час до відображення найбільшого видимого елемента сторінки.

мс – мілісекунда; одиниця вимірювання часу, одна тисячна частина секунди.

npm (Node Package Manager) — менеджер пакетів для середовища виконання Node.js.

RAM (Random Access Memory) – оперативна пам'ять комп'ютера.

SFC (Single File Component) – однофайловий компонент; підхід Vue.js, де шаблон, стилі і логіка компонента розміщені в одному файлі.

SPA (Single Page Application) – односторінковий застосунок; веб-застосунок, що завантажує HTML-оболонку один раз і динамічно оновлює вміст без перезавантаження.

SSG (Static Site Generation) – генерація статичного сайту; формування HTML-сторінок на етапі збірки, а не під час запиту.

SSR (Server-Side Rendering) – рендеринг на стороні сервера; формування готового HTML на сервері перед відправленням клієнту.

TBT (Total Blocking Time) – загальний час блокування основного потоку браузера між FCP і TTI.

TS (TypeScript) – типізована надмножина JavaScript з підтримкою статичної типізації.

TTI (Time to Interactive) – час до інтерактивності; момент, коли сторінка повністю готова до взаємодії з користувачем.

UI (User Interface) – користувацький інтерфейс; сукупність засобів взаємодії користувача з програмою.

UX (User Experience) – досвід користувача; загальне враження від взаємодії з продуктом.

v-DOM (Virtual DOM) – віртуальна об'єктна модель документа; легка JavaScript-копія реального DOM, що використовується для оптимізації оновлень інтерфейсу.

Web Vitals – ініціатива Google щодо визначення ключових метрик якості веб-сторінок з точки зору користувацького досвіду.

XSS (Cross-Site Scripting) – міжсайтовий скриптинг; тип атаки, що передбачає впровадження шкідливого JavaScript-коду на сторінку.

ЗМІСТ

ВСТУП

Сучасний веб давно перестав бути набором статичних сторінок. У браузері сьогодні живуть повноцінні бухгалтерські системи, логістичні платформи, освітні кабінети, аналітичні дашборди та сервіси електронної комерції. Для таких продуктів клієнтська частина виконує не декоративну, а центральну функцію: вона безпосередньо визначає, наскільки швидко співробітник завершить задачу, наскільки охоче покупець оформить замовлення і, зрештою, наскільки продукт є конкурентоспроможним. За даними Google, збільшення часу завантаження сторінки з однієї до трьох секунд підвищує ймовірність відмови користувача на 32%, а кожна десята частка секунди затримки у мобільному пошуку знижує конверсію в середньому на 0,3% [9, 10].

Разом з тим вибір frontend-технології є нетривіальним рішенням. На ринку представлені десятки фреймворків і бібліотек, які пропонують принципово різні підходи до побудови інтерфейсу. React та Preact використовують Virtual DOM, Angular спирається на механізм Change Detection, Vue реалізує гранулярну реактивність через proxy-об'єкти, Svelte переносить більшість обчислень на етап компіляції, Solid застосовує signals без Virtual DOM, а Qwik розвиває концепцію resumability, що принципово відрізняється від класичної гідратації. Кожен з цих механізмів по-різному впливає на час першого відображення, розмір JavaScript-пакета, споживання CPU та загальну плавність взаємодії [21, 23, 25].

Практична складність вибору технологічного стеку полягає ще й у тому, що популярність фреймворку далеко не завжди корелює з його технічними характеристиками. React займає домінуючу позицію в комерційній розробці завдяки величезній екосистемі та ринку спеціалістів, однак за чистими benchmark-метриками він програє менш поширеним рішенням. Angular обирають великі корпоративні команди за архітектурну дисципліну та

вбудовані інструменти, хоча його production-збірка є найповільнішою серед усіх досліджуваних фреймворків. Рішення про вибір стеку вимагає не суб'єктивних оцінок, а відтворюваних вимірювань [45, 47, 48].

Ключовою методологічною проблемою більшості наявних порівняльних досліджень є відсутність однакового контрольованого середовища: різні автори тестують принципово різні застосунки з різним обсягом функціоналу, що унеможливлює коректну інтерпретацію результатів. Дане дослідження будується на протилежному принципі: використовується відкрита benchmark-платформа framework-benchmarks, де всі фреймворки реалізують однаковий застосунок, працюють з ідентичними мок-даними і проходять один і той самий набір Playwright-тестів. Це забезпечує максимальну порівнянність результатів і виключає вплив якості конкретної реалізації [37, 39].

Об'єктом дослідження є процес оцінювання продуктивності клієнтської частини сучасних веб-застосунків. Предметом є 12 frontend-фреймворків і бібліотек у спільному benchmark-контексті та їх архітектурні відмінності, що визначають продуктивність. Метою роботи є дослідження продуктивності сучасних frontend-фреймворків на прикладі реального benchmark-проєкту, аналіз результатів та формування практичних рекомендацій щодо вибору технологічного стеку залежно від типу і масштабу проєкту. Для досягнення мети необхідно розв'язати такі завдання: проаналізувати архітектурні підходи до побудови сучасних frontend-застосунків; охарактеризувати ключові метрики продуктивності веб-інтерфейсів згідно зі стандартами Web Vitals; описати benchmark-платформу та методику вимірювань; провести порівняльний аналіз результатів тестування; сформулювати практичні рекомендації щодо вибору frontend-стеку.

Наукова новизна роботи полягає у комплексному емпіричному порівнянні дванадцяти frontend-рішень в ідентичному контрольованому середовищі з використанням стандартизованих метрик Web Vitals і Lighthouse.

На відміну від більшості існуючих порівнянь, що охоплюють два-три найпопулярніші фреймворки, дане дослідження аналізує повний спектр підходів від мікробібліотек без runtime до повноцінних корпоративних платформ. Практичне значення роботи полягає в тому, що отримані результати можуть безпосередньо використовуватися розробниками та архітекторами під час обґрунтування технологічних рішень для веб-застосунків різного масштабу [49, 50].

РОЗДІЛ 1

ТЕОРЕТИЧНІ ЗАСАДИ FRONTEND-РОЗРОБКИ ТА ПРОДУКТИВНОСТІ ВЕБ-ІНТЕРФЕЙСІВ

1.1 Архітектура клієнтської частини та роль frontend у сучасних системах

Frontend-розробка охоплює створення тієї частини програмного продукту, яку користувач безпосередньо сприймає і з якою взаємодіє через браузер. На технічному рівні клієнтська частина будується на трьох фундаментальних веб-технологіях, кожна з яких виконує окрему функцію. HTML визначає структуру і семантику документа: які елементи присутні на сторінці, у якому порядку вони розміщені і яке їхнє логічне призначення. CSS відповідає за візуальне оформлення: кольори, розміри шрифтів, відступи, позиціонування блоків та адаптивне відображення. JavaScript забезпечує поведінкову складову: обробку подій користувача, динамічну зміну вмісту без перезавантаження, валідацію введених даних та спілкування із серверним API через асинхронні запити [11, 12, 13].

Впродовж останніх десяти-п'ятнадцяти років роль клієнтської частини принципово змінилась. Якщо раніше frontend був переважно вікном для відображення даних, сформованих на сервері, то сьогодні браузер виступає повноцінним середовищем виконання складних застосунків. У ньому реалізуються CRM-системи з тисячами записів, платіжні кабінети, складські платформи з оновленням залишків у реальному часі, освітні середовища з відеоконференцією та аналітичні інструменти з інтерактивними графіками. Для таких продуктів frontend визначає не лише зовнішній вигляд, а й швидкість виконання операцій та загальну ефективність бізнес-процесів. Дослідження Nielsen Norman Group фіксують: у внутрішніх корпоративних системах, де персонал виконує понад 200 операцій щодня, сповільнення

інтерфейсу навіть на 1,5 секунди за операцію призводить до втрати кількох годин продуктивності на тиждень [15, 41].

Архітектурним стандартом для більшості сучасних клієнтських застосунків став патерн Single Page Application. У такій моделі браузер завантажує HTML-оболонку лише один раз при першому запиті, а подальша навігація між розділами відбувається через динамічне оновлення окремих частин інтерфейсу без повного перезавантаження сторінки. Це суттєво знижує кількість мережевих запитів, усуває неприємне мерехтіння при переходах і дає відчуття плавної взаємодії, знайоме з нативних мобільних застосунків [14]. При цьому серверна частина перетворюється на API, що повертає дані у форматі JSON, а вся логіка відображення і управління станом зосереджується на клієнті.

Компонентна архітектура стала стандартом де-факто для всіх сучасних frontend-фреймворків і принципово змінила підхід до організації коду. Компонент є ізольованою самодостатньою одиницею інтерфейсу, яка поєднує шаблон, стиль і поведінкову логіку. Компоненти є повторно використовуваними: написаний один раз компонент застосовується в будь-якому місці застосунку без дублювання коду. Вони є незалежно тестованими і легко замінними. Зміна внутрішньої реалізації компонента не порушує роботу інших за умови дотримання правил інтерфейсу взаємодії через props та події [21, 25].

Управління станом застосунку є однією з найскладніших задач у frontend-розробці. Під станом розуміється сукупність всіх даних, від яких залежить поточний вигляд інтерфейсу: список товарів у кошику, текст у полі пошуку, активна вкладка, результат API-запиту тощо. При невеликому застосунку стан зручно тримати локально всередині компонентів. Коли ж проєкт зростає і кілька різних компонентів потребують доступу до одних і тих самих даних, виникає необхідність у централізованому сховищі, що управляє потоком змін у передбачуваний і однонаправлений спосіб.

Безпека клієнтської частини є суттєвим аспектом архітектури. Frontend повинен коректно працювати з токенами авторизації, не зберігати чутливі дані у відкритому вигляді в localStorage, захищати форми від CSRF-атак і запобігати XSS-вразливостям через правильне екранування вхідних даних. Сучасні фреймворки вирішують задачу захисту від XSS автоматично: React, Vue, Angular і Svelte за замовчуванням екранують всі вставки в шаблон і не дозволяють довільне виконання HTML без явного дозволу розробника [8, 14]. Це є важливою перевагою компонентного підходу порівняно з прямим маніпулюванням innerHTML у Vanilla JS, де відповідальність за безпеку повністю покладається на розробника.

Таблиця 1.1

Еволюція підходів до frontend-розробки

Покоління	**Підхід**	**Представники**	**Ключова особливість**
1990-ті	Статичний HTML	HTML, CGI-скрипти	Повне перезавантаження сторінки при кожній дії
2000-ні	Server-side render	PHP, JSP, ASP.NET	Сервер формує HTML, JavaScript для дрібних ефектів
2006-2013	jQuery-era	jQuery, Prototype	DOM-маніпуляції, Ajax без перезавантаження
2013-2016	MVC-фреймворки	AngularJS, Backbone	Перші SPA, двостороннє зв'язування даних
2013-2016	Компонентні бібліотеки	React, Angular 2	Компоненти як основна одиниця розробки
2016-дотепер	Сучасні фреймворки	Vue, Svelte, Solid, Qwik	Оптимізований рендеринг, fine-grained reactivity

Таким чином, frontend-розробка охоплює широкий спектр технічних рішень і пройшла тривалий шлях від статичних HTML-сторінок до повноцінних клієнтських застосунків. Вибір конкретного фреймворку суттєво впливає на те, як задачі архітектури вирішуються, скільки зусиль вимагає їх реалізація і наскільки ефективно працює готовий застосунок у браузері.

1.2 Огляд сучасних frontend-фреймворків та підходів до рендерингу

Ключовою відмінністю між сучасними frontend-фреймворками є спосіб синхронізації стану застосунку і поточного відображення в браузері. Для розуміння цього питання важливо усвідомити, що DOM-операції є одними з найбільш ресурсомістких у веб-браузері. Кожна зміна DOM-дерева може призводити до recalculate style, layout та paint, тобто до перерахунку геометрії і перемальовування частин сторінки. Тому мінімізація кількості і масштабу DOM-операцій є центральним завданням будь-якого механізму рендерингу, і саме тут різні фреймворки демонструють принципові відмінності.

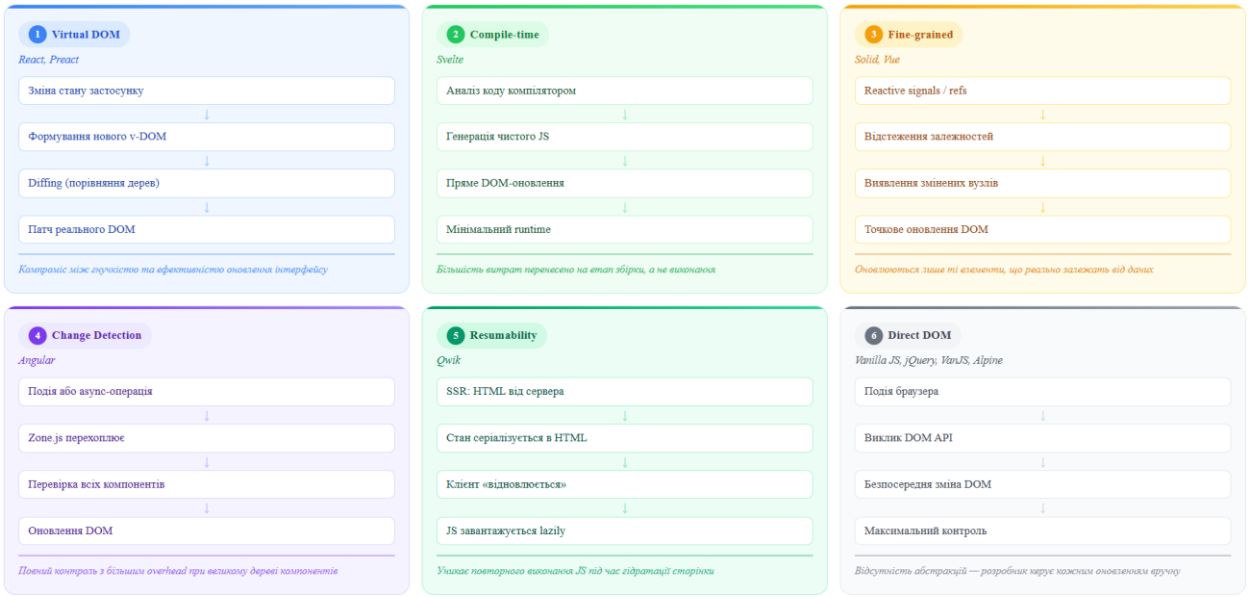


Рисунок 1.1. Механізми рендерингу у сучасних frontend-фреймворках

Virtual DOM, реалізований у React та Preact, є одним з найпоширеніших підходів. Бібліотека підтримує легку JavaScript-копію DOM-дерева в пам'яті. При кожній зміні стану формується нова копія дерева, після чого алгоритм дифінгу порівнює її зі старою версією і визначає мінімальний набір змін для реального DOM. Перевагою є передбачуваність і зручність розробки: компонент завжди декларативно описує бажаний вигляд інтерфейсу, а бібліотека вирішує, як ефективно оновити DOM. Недолік полягає в тому, що

дифінг сам по собі споживає процесорний час, особливо при великому дереві компонентів [21, 22].

Compile-time рендеринг, реалізований у Svelte, принципово відрізняється від Virtual DOM. Компілятор аналізує код під час збірки і генерує оптимізований JavaScript, який при зміні реактивних змінних безпосередньо оновлює конкретні DOM-елементи без жодних проміжних представлень. У production-збірці немає фреймворкової бібліотеки, лише чистий оптимізований JavaScript [27]. Fine-grained reactivity, реалізована у Solid та Vue, є найточнішим з погляду DOM-операцій підходом: при зміні конкретного значення оновлюються лише ті DOM-вузли, що реально залежать від нього. У Solid це реалізовано через signals, а компоненти виконуються лише один раз при монтуванні [25, 26]. Change Detection в Angular через Zone.js перехоплює всі асинхронні операції і після кожної з них запускає цикл перевірки змін у дереві компонентів. Resumability у Qwik усуває гідратацію: стан серіалізується в HTML-атрибути під час SSR, а JavaScript завантажується ліниво [29]. Direct DOM у Vanilla JS, jQuery, VanJS та Alpine.js є найпростішим підходом з нульовим runtime-overhead, але з очевидними обмеженнями масштабованості [28, 34].

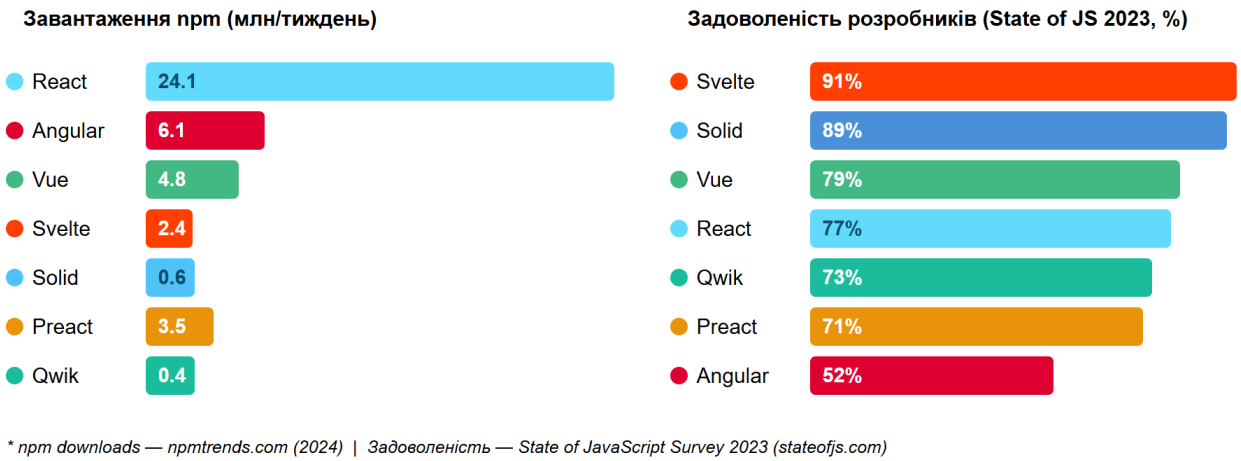


Рисунок 1.2. Динаміка популярності frontend-фреймворків за npm-завантаженнями та задоволеністю розробників

Порівняння популярності за npm-завантаженнями і задоволеністю розробників виявляє цікаву суперечність: найпопулярніші технології не є лідерами за задоволеністю. React домінує за кількістю завантажень (24,1 млн/тиждень), але за задоволеністю (77%) поступається Svelte (91%) і Solid (89%). Angular, незважаючи на значну частку ринку (6,1 млн/тиждень), показує найнижчий показник задоволеності серед сучасних фреймворків (52%). Це відображає реальну ринкову динаміку: компанії продовжують використовувати Angular через legacy-проєкти і корпоративні стандарти, навіть якщо розробники воліли б перейти на щось інше. Svelte і Solid, навпаки, демонструють ентузіазм ранніх прихильників, яких приваблює чистота архітектурних рішень та продуктивність. Цей розрив між «популярністю» і «задоволеністю» є важливим контекстом для інтерпретації benchmark-результатів [38, 39].

Таблиця 1.2

Порівняльна характеристика frontend-фреймворків за підходом до рендерингу

Фреймворк	**Тип рішення**	**Підхід до рендерингу**	**Runtime**	**Мова**
React	Бібліотека	Virtual DOM	Середній	JS / TS
Angular	Платформа	Change Detection	Важкий	TypeScript
Vue	Фреймворк	Fine-grained reactive	Легкий	JS / TS
Svelte	Компілятор	Compile-time	Мінімальний	JS / TS
Solid	Бібліотека	Fine-grained signals	Мінімальний	JS / TS
Qwik	Фреймворк	Resumability	Легкий	TypeScript
Preact	Бібліотека	Virtual DOM	Дуже легкий	JS / TS
Lit	Бібліотека	Web Components	Легкий	JS / TS
Alpine.js	Мікробібліотека	Declarative DOM	Мінімальний	JS
VanJS	Мікробібліотека	Direct DOM	Мінімальний	JS
jQuery	Бібліотека	DOM API	Легкий	JS
Vanilla JS	Без бібліотеки	Direct DOM API	Немає	JS / TS

React, розроблений Meta і вперше представлений у 2013 році, залишається найпоширенішою frontend-бібліотекою у світі. Компонентність реалізована через функціональні компоненти та JSX. Хуки, введені у React 16.8, дозволяють управляти станом і побічними ефектами у функціональних компонентах без використання класів. React не нав'язує архітектуру: маршрутизація, управління станом, робота з формами реалізуються через окремі бібліотеки, що забезпечує гнучкість, але вимагає від команди самостійного вибору і підтримки цих рішень. [16, 17]

Angular, розроблений Google і представлений у 2016 році, є найповнішим за набором вбудованих можливостей. Він включає маршрутизатор, HTTP-клієнт, систему форм, dependency injection контейнер і потужний CLI. Angular використовує TypeScript як основну мову, що забезпечує статичну типізацію і раннє виявлення помилок. Механізм Change Detection через Zone.js гарантує, що UI завжди відображає актуальний стан, однак ціна за це помітна у benchmark-даних. [18, 19, 23, 24]

Vue.js, розроблений Еваном Ю і представлений у 2014 році, позиціонується як прогресивний фреймворк. Composition API у Vue 3 наближає його до функціонального стилю React при збереженні власної реактивної системи через Proxy. Такий підхід дає Vue перевагу перед React за FCP і розміром бандлу при порівнянному рівні зручності розробки. Svelte, розроблений Річом Харрісом і представлений у 2016 році, є більше компілятором, ніж бібліотекою: у production-бандлі немає нічого від фреймворку, лише мінімальний runtime. Solid.js, синтаксично схожий на React, відмовляється від Virtual DOM на користь signals, що дає одні з найкращих показників FCP і LCP серед усіх досліджуваних фреймворків. [25, 26, 27, 28]

Qwik вирішує фундаментальну проблему SSR: необхідність повторного виконання JavaScript для відновлення стану. Концепція є перспективною, але, як показують результати Розділу 3, поточні benchmark-дані виявились менш вражаючими, ніж очікувалось [29]. Preact надає майже повну сумісність з

React-екосистемою при значно меншому розмірі бандлу (~3-4 КБ), що робить його привабливим компромісом [30]. Lit від Google надає засоби для роботи з нативними Web Components і найкраще підходить для дизайн-систем та UI-бібліотек [31]. Alpine.js і VanJS є мінімалістичними бібліотеками для сценаріїв, де повноцінний фреймворк є надлишковим [32, 33].

1.3 Метрики продуктивності веб-інтерфейсу та стандарти Web Vitals

Продуктивність веб-інтерфейсу є комплексним і багатовимірним поняттям. Користувач сприймає продуктивність суб'єктивно, через відчуття швидкості і плавності, але для об'єктивного порівняння технологій необхідні кількісні метрики. Впродовж останніх років Google розробила ініціативу Web Vitals, що визначає набір ключових показників, найбільш тісно пов'язаних з реальним користувацьким досвідом [9]. Ці метрики стали галузевим стандартом і застосовуються у Lighthouse, PageSpeed Insights та Chrome DevTools.

First Contentful Paint фіксує момент, коли браузер вперше відмалював будь-який текст або зображення. Норма Web Vitals: менше 1,8 секунди є добрим результатом, від 1,8 до 3,0 секунд є прийнятним, понад 3,0 секунди вважається поганим. Для frontend-фреймворків FCP залежить насамперед від розміру JavaScript-бандлу і часу його парсингу та виконання браузером [10, 11]. Largest Contentful Paint вимірює час до відображення найбільшого видимого елемента у вікні перегляду. Норма: менше 2,5 секунди є добрим результатом, 2,5-4,0 секунди є прийнятним, понад 4,0 секунди є поганим. Дослідження показують, що сторінки з LCP понад 4 секунди мають на 30% вищий показник відмов [10, 41].

Cumulative Layout Shift вимірює стабільність макета сторінки під час завантаження. Для frontend-фреймворків CLS здебільшого залежить від якості конкретної реалізації, а не від вибору фреймворку, тому не є ключовою метрикою у цьому дослідженні [13]. Interaction to Next Paint замінив у 2024

році показник FID і вимірює затримку між будь-якою дією користувача і наступним кадром відображення. Норма: менше 200 мілісекунд є добрим результатом [12]. Lighthouse Performance Score є зваженою інтегральною оцінкою від 0 до 100 на основі LCP (25%), Total Blocking Time (30%), Speed Index (10%), CLS (25%) і FCP (10%). Оцінка вище 90 вважається відмінною, 50-89 прийнятною, нижче 50 поганою [14].

Таблиця 1.3

Ключові метрики продуктивності веб-інтерфейсу

Метрика	**Повна назва**	**Норма (добре)**	**Що вимірює**
FCP	First Contentful Paint	< 1,8 с	Час до першого відображення будь-якого контенту
LCP	Largest Contentful Paint	< 2,5 с	Час завантаження найбільшого видимого елемента
CLS	Cumulative Layout Shift	< 0,1	Накопичений зсув елементів під час завантаження
INP	Interaction to Next Paint	< 200 мс	Затримка від дії користувача до наступного кадру
TBT	Total Blocking Time	< 300 мс	Сумарний час блокування основного потоку браузера

Розмір gzipped-бандлу не є метрикою Web Vitals, однак безпосередньо впливає на FCP і LCP. За оцінками Google, кожні 100 КБ JavaScript після розпакування можуть додавати десятки мілісекунд до часу виконання, особливо на слабших пристроях або повільних мережах [42]. Варто розуміти, що браузер не просто завантажує JavaScript-файл, а проходить через кілька послідовних етапів: мережеве завантаження, розпакування з gzip, парсинг байт-коду, компіляцію і виконання. Кожен з цих етапів масштабується пропорційно до розміру бандлу, тому різниця між 10 КБ у VanJS і 72,8 КБ у Angular означає не просто різний час завантаження, а принципово різне навантаження на весь pipeline браузера. Для мобільних пристроїв середнього класу, де CPU у 4-6 разів слабший за desktop, цей ефект стає ще відчутнішим і

може перетворити прийнятний FCP на desktop у критично повільний на смартфоні.

Build Time, тобто час production-збірки, не впливає безпосередньо на кінцевого користувача, але має суттєве практичне значення для команди розробників. У проєктах з частими деплоями, де нова версія виходить кілька разів на день, кожна зайва хвилина збірки накопичується у реальні витрати: десятихвилинна збірка Angular при п'яти деплоях на день означає майже годину простою CI/CD-конвеєра щодня. Окрім прямих витрат, довга збірка уповільнює зворотний зв'язок під час розробки і знижує мотивацію команди до частих малих змін на користь рідких великих релізів, що є небажаною практикою з точки зору сучасного DevOps.

Таблиця 1.4

Порівняння підходів CSR та SSR

Критерій	**CSR (Client-Side Rendering)**	**SSR (Server-Side Rendering)**
Перший рендер	Повільніший (після виконання JS)	Швидший (готовий HTML від сервера)
Інтерактивність	Висока після завантаження	Потребує гідратації
SEO	Обмежене без додаткових засобів	Добре за замовчуванням
Навантаження сервера	Менше	Більше
Складність	Простіша початкова конфігурація	Потрібна серверна інфраструктура

Сукупність метрик FCP, LCP, Lighthouse Score, Gzipped Bundle Size та Build Time складає основу порівняльного аналізу, проведеного у Розділі 3. Разом вони дають достатньо повне уявлення про технічний профіль кожного фреймворку: наскільки швидко він забезпечує перше відображення, якою є вага його runtime і яку ціну платить розробник за використання тієї чи іншої технології. Важливо підкреслити, що жодна з цих метрик не є самодостатньою: тільки їх комплексний аналіз дозволяє скласти об'єктивну картину [45, 49].

Висновок до розділу 1

У першому розділі розглянуто теоретичні засади frontend-розробки та підходи до оцінювання продуктивності веб-інтерфейсів. Проаналізовано еволюцію клієнтської частини від статичних HTML-сторінок до повноцінних SPA-застосунків, а також шість принципово різних механізмів рендерингу: Virtual DOM, Compile-time, Fine-grained reactivity, Change Detection, Resumability та Direct DOM. Показано, що вибір механізму рендерингу безпосередньо визначає обсяг runtime-бібліотеки та швидкість DOM-оновлень.

Досліджено стандарти Web Vitals як галузеву основу для об'єктивного вимірювання продуктивності: FCP, LCP, CLS, INP, TBT та інтегральний Lighthouse Performance Score. Встановлено, що популярність фреймворку не корелює з його технічними характеристиками, що підтверджує необхідність емпіричного порівняння в контрольованих умовах. Отримані відомості формують теоретичну основу для benchmark-аналізу у наступних розділах.

РОЗДІЛ 2

ОПИС BENCHMARK-ПЛАТФОРМИ ТА МЕТОДИКА ДОСЛІДЖЕННЯ

2.1 Характеристика проєкту framework-benchmarks та його архітектура

Центральним інструментом дослідження є відкрита benchmark-платформа framework-benchmarks, розроблена для об'єктивного порівняння продуктивності сучасних frontend-технологій. Ключова відмінність від більшості існуючих порівнянь полягає у реалізації одного і того ж застосунку в усіх досліджуваних технологіях. Усі реалізації розміщені в єдиному репозиторії, використовують спільні статичні ресурси, ідентичні мок-дані і один набір Playwright-тестів. Будь-яка зафіксована різниця у показниках продуктивності пояснюється властивостями конкретного фреймворку, а не якістю чи складністю реалізації [37]. Саме ця методологічна чистота робить результати платформи придатними для академічного аналізу і практичних висновків, що підтверджують дослідники Ollila та Levlin у своїх роботах з порівняння frontend-рішень [46, 48].

У порівнянні з більшістю неформальних порівнянь, де кожен фреймворк тестується на різних демо-застосунках різної складності, такий підхід є методологічно значно сильнішим. Він виключає вплив обсягу функціоналу, структури коду, кількості залежностей і навіть налаштування build-інструменту. Функціональний зміст benchmark-застосунку спеціально обраний мінімальним, але репрезентативним: застосунок відображає погодний сервіс з поточними умовами і прогнозом на кілька днів і охоплює всі ключові операції, характерні для реальних SPA-застосунків: початковий рендер списку елементів, реакцію на дії користувача, оновлення стану і синхронізацію DOM. Саме ці операції найбільше навантажують механізми рендерингу, тому вони є репрезентативними для вимірювання FCP, LCP і Lighthouse Score [39].

Таблиця 2.1

Структура benchmark-проєкту framework-benchmarks

Елемент структури	Призначення та вміст
frameworks/	Підпроєкти для кожного фреймворку, кожен ізольований від інших
shared/assets/	Спільні статичні ресурси: зображення, іконки, шрифти
shared/styles/	Базові CSS-стилі, однакові для всіх реалізацій
shared/mocks/	Мок-дані погодного API у JSON-форматі, ідентичні для всіх
playwright-tests/	Автоматизовані тести з однаковими сценаріями для всіх реалізацій
results/	Файли з результатами вимірювань у форматі summary.tsv
package.json	Кореневий файл конфігурації з єдиними скриптами запуску

Кожен підпроєкт є повноцінним ізольованим frontend-застосунком зі своєю структурою каталогів, набором залежностей, конфігурацією збірки та скриптами запуску. Ізольованість підпроєктів є принципово важливою: зміна або помилка в одному не впливає на інший, що дозволяє паралельно тестувати різні версії тієї самої технології або вмикати окремі реалізації без порушення загальної структури [37]. Автоматизована система виконання тестів діє за фіксованим конвеєром: встановлення залежностей через npm install, production-збірка через npm run build з вимірюванням часу, запуск production-сервера, виконання Playwright-сценаріїв з п'ятиразовим повторенням кожного тесту, збір метрик через Lighthouse і Performance API, усереднення результатів та запис у файл summary.tsv. Повна відтворюваність процесу є ключовою перевагою: тести можна запустити повторно в тому ж середовищі і отримати ті ж значення, що підтверджує надійність даних.

Таблиця 2.2

Інструменти збірки для кожного фреймворку

Фреймворк	**Інструмент збірки**	**Особливість конфігурації**
React	Vite / Webpack	Два варіанти конфігурації, обидва підтримуються
Angular	Angular CLI	AOT-компіляція, власний оптимізатор
Vue	Vite	Офіційно рекомендований інструмент для Vue 3
Svelte	Vite + SvelteKit	SvelteKit додає SSR, file-based routing і власний runtime
Solid	Vite	Babel-плагін для трансформації JSX
Qwik	Qwik CLI	Власна система збірки для підтримки resumability
Preact	Vite	Аліаси preact/compat для React-сумісності
Lit	Rollup	Мінімальний bundler від Google
Alpine.js	CDN / без збірки	Підключається через script tag, build-крок відсутній
Vanilla JS	Без збірки	Чистий JavaScript, компіляція не потрібна
VanJS	Vite (опційно)	Може використовуватись і без збірки

Використання різних інструментів збірки є реалістичним відображенням виробничої практики: кожна технологія входить у проєкт зі своїм рекомендованим екосистемним оточенням, і порівняння Build Time безпосередньо враховує цю реальність. Важливо розуміти, що Alpine.js і Vanilla JS показують нульовий Build Time лише тому, що не мають build-кроку взагалі, а не через якусь особливу оптимізацію збірки. Це необхідно враховувати при інтерпретації результатів таблиці 3.1. Порівняння Build Time між рішеннями з різними інструментами є умовним, але відображає реальні девелоперські витрати при розгортанні тієї чи іншої технології у виробничому середовищі.

2.2 Критерії відбору та загальна характеристика фреймворків

До дослідження включено 12 frontend-рішень, відібраних за чотирма основними критеріями: наявність реалізації у benchmark-проєкті; принципово різні підходи до рендерингу і управління станом; різний рівень абстракції та

розміру runtime; різна популярність і зрілість екосистеми на ринку. Такий набір дозволяє охопити повний діапазон сучасних frontend-підходів від мікробібліотек без будь-якого runtime до повноцінних корпоративних платформ зі всіма вбудованими засобами для розробки. На відміну від більшості наявних досліджень, що обмежуються порівнянням двох-трьох найпопулярніших фреймворків, даний набір забезпечує значно ширший погляд і дозволяє виявити не лише відносні переваги конкретних технологій, а й глибші закономірності між архітектурним підходом і практичними результатами продуктивності [45, 47, 48].

Таблиця 2.3

Повна характеристика досліджуваних frontend-рішень

Фреймворк	**Тип**	**Підхід**	**Складність**	**Ринок спеціалістів**
React	Бібліотека	Virtual DOM	Середня	Дуже великий
Angular	Платформа	Change Detection	Висока	Великий
Vue	Фреймворк	Fine-grained	Низька	Великий
Svelte	Компілятор	Compile-time	Низька	Середній
Solid	Бібліотека	Fine-grained	Середня	Невеликий
Qwik	Фреймворк	Resumability	Середня	Дуже малий
Preact	Бібліотека	Virtual DOM	Низька	Середній
Lit	Бібліотека	Web Components	Середня	Середній
Alpine.js	Мікробібліотека	Declarative DOM	Низька	Середній
VanJS	Мікробібліотека	Direct DOM	Низька	Малий
jQuery	Бібліотека	DOM API	Низька	Спадаючий
Vanilla JS	Без бібліотеки	Direct DOM	Низька	Базовий рівень

Класичні SPA-фреймворки, React, Angular і Vue, є стандартами індустрії з найбільшими спільнотами розробників і найширшими екосистемами сторонніх бібліотек та інструментів. Саме ці три фреймворки найчастіше фігурують у вакансіях і є традиційно «безпечним» вибором для більшості

компаній з точки зору мінімізації організаційних ризиків [45, 47]. Сучасні інноваційні рішення, Svelte і Solid, пропонують принципово різні підходи, що дозволяють досягати вищої продуктивності порівняно з класичними фреймворками. Їхня популярність стрімко зростає: Svelte кілька років поспіль займає перші місця в опитуванні State of JS за категорією «задоволеність розробників». Однак ринок спеціалістів для цих технологій залишається значно вужчим, що є суттєвим обмеженням при плануванні довгострокової підтримки. Qwik є новою технологією з унікальною архітектурною концепцією, проте станом на 2024-2025 рік вона перебуває на ранніх стадіях production-ready зрілості, і виявлені benchmark-результати відображають цей стан речей. Легковісні бібліотеки Alpine.js, VanJS і Lit включені до дослідження як рішення для нішових сценаріїв, де повноцінний фреймворк є надлишковим, а базові підходи Vanilla JS і jQuery слугують контрольними точками для розуміння граничних значень продуктивності.

2.3 Архітектура клієнтської частини benchmark-застосунку

Попри суттєві відмінності у технологіях, всі підпроєкти framework-benchmarks дотримуються однакової логічної архітектури. Це є принциповою умовою коректного порівняння: всі реалізації виконують однаковий функціональний набір операцій, відрізняючись лише технічними засобами реалізації. Уніфікована архітектура гарантує, що відмінності у результатах вимірювань відображають властивості фреймворків, а не різницю у складності реалізацій [37, 39].

UI-компоненти є базовим рівнем архітектури і відповідають за окремі фрагменти інтерфейсу: блок поточних погодних умов, список прогнозу, елемент вибору міста. Компонент отримує дані через props або з реактивного сховища, відображає їх у шаблоні і сигналізує про дії користувача через події. Компоненти не містять мережевої логіки і не взаємодіють напряму з мок-даними, що забезпечує чистоту розділення відповідальностей [21, 25].

Кореневий компонент виступає координатором: він визначає загальну структуру інтерфейсу, підключає дочірні компоненти і зв'язує їх зі станом. Час від запуску main.js до відображення кореневого компонента у DOM безпосередньо впливає на значення FCP, яке є центральною метрикою порівняння.

Структура підпроєкту

```
src/
├── components/        # UI-компоненти застосунку
│   ├── WeatherCard    # Компонент поточної погоди
│   ├── Forecast       # Компонент прогнозу на тиждень
│   └── CityPicker     # Компонент вибору міста
├── app.(js|ts)        # Кореневий компонент
├── state/             # Логіка управління станом
│   └── store          # Centralized state або reactive variables
├── services/          # Допоміжна логіка
│   └── weather        # Форматування і трансформація мок-даних
└── main.(js|ts)       # Точка входу в застосунок
```

Рівень стану містить механізми зберігання і оновлення даних застосунку: поточне місто, завантажені погодні дані, стан завантаження і повідомлення про помилки. Реалізація управління станом суттєво різниться між фреймворками і є одним з ключових факторів, що впливає на частоту і масштаб DOM-оновлень. React використовує хуки useState або useReducer, Vue реалізує reactive-об'єкти через Proxy, Solid застосовує signals, Svelte використовує реактивні оголошення через компілятор [21, 22, 25, 27]. Рівень сервісів містить логіку, не пов'язану з відображенням: перетворення мок-даних, форматування температури, конвертацію одиниць. Усі реалізації використовують один набір мок-даних з shared/mocks/, що гарантує однакові вхідні умови для вимірювань.

Таблиця 2.4

Реалізація управління станом у різних фреймворках

Фреймворк	**Механізм управління станом**	**Особливість**
React	useState, useReducer, useContext	Re-render компонента при кожній зміні стану
Angular	RxJS Observable, services	Push-модель через потоки подій, Zone.js
Vue	reactive(), ref(), computed()	Автоматичне відстеження залежностей через Proxy
Svelte	Реактивні змінні, $: оголошення	Реактивність визначається компілятором під час збірки
Solid	createSignal(), createStore()	Гранулярні оновлення без re-render компонента
Qwik	useStore(), useSignal()	Стан серіалізується в HTML-атрибути при SSR

Лістинг 2.1. Приклади реалізації управління станом

```
// React — useState hook
const [city, setCity] = useState("Kyiv");
const [weather, setWeather] = useState(null);
const handleCityChange = (newCity) => setCity(newCity);

// Vue 3 — Composition API
const city = ref("Kyiv");
const weather = ref(null);
const handleCityChange = (newCity) => { city.value = newCity; };

// Solid — createSignal
const [city, setCity] = createSignal("Kyiv");
const [weather, setWeather] = createSignal(null);
const handleCityChange = (newCity) => setCity(newCity);

// Svelte — reactive variable
let city = "Kyiv";
let weather = null;
function handleCityChange(newCity) { city = newCity; }

// Vanilla JS — direct DOM manipulation
let city = "Kyiv";
document.getElementById("city-select")
  .addEventListener("change", (e) => {
    city = e.target.value;
    renderWeather(city);
  });
```

Різниця між підходами стає особливо помітною при частих оновленнях стану. У React кожен виклик setCity призводить до повторного виконання

render-функції компонента і оновлення v-DOM, хоча React намагається мінімізувати реальні DOM-операції через diffing. У Solid виклик setCity оновлює лише ті DOM-вузли, що підписані на signal city, без жодного повторного виконання компонентної функції. У Svelte компілятор заздалегідь генерує JavaScript-код, що при зміні city напряму оновлює відповідні текстові вузли DOM. Ці відмінності відображаються у результатах вимірювань і пояснюють різницю між FCP і Lighthouse Score різних технологій [21, 23, 27].

Таблиця 2.5

Порівняння підходів до реалізації компонентів

Фреймворк	**Синтаксис компонентів**	**Формат файлу**	**Типізація**
React	Функціональні компоненти, JSX	.jsx / .tsx	JS або TS
Angular	Класи з декораторами, HTML-шаблони	.ts + .html	TypeScript
Vue	SFC: template + script + style	.vue	JS або TS
Svelte	Svelte-файл: HTML + скрипт + стилі	.svelte	JS або TS
Solid	Функціональні компоненти, JSX	.jsx / .tsx	JS або TS
Vanilla JS	Функції, класи, шаблонні рядки	.js / .ts	JS або TS

Таблиця 2.6

Класифікація frontend-фреймворків за механізмом рендерингу

Підхід до рендерингу	**Фреймворки**	**Ключова особливість**
Virtual DOM	React, Preact	Diffing між старим і новим деревом
Fine-grained	Vue, Solid	Точкові оновлення залежних DOM-вузлів
Compile-time	Svelte	Рендеринг-логіка визначається під час збірки
Change Detection	Angular	Цикл перевірки всього дерева компонентів
Resumability	Qwik	Серіалізація стану в HTML, lazy завантаження JS
Direct DOM	Vanilla JS, jQuery, VanJS, Alpine	Вручну через DOM API браузера

2.4 Методика вимірювання продуктивності та умови експерименту

Оцінювання продуктивності виконувалось за суворо стандартизованою процедурою. Для кожного фреймворку тест запускався не менше п'яти разів, а до файлу summary.tsv записувалось усереднене значення. Усереднення є критично важливим, оскільки час браузерного рендерингу може варіюватися від запуску до запуску через неоднорідність JIT-компіляції JavaScript, фонових процесів і кешування [34, 37]. П'ять повторень з усередненням дозволяють виключити більшість випадкових відхилень і отримати стабільне репрезентативне значення. Усі тести виконувались на одній виділеній машині з фіксованою конфігурацією: CPU 8 ядер, 16 ГБ RAM, операційна система Ubuntu 22.04, Node.js версії 20 LTS. Браузером-виконавцем слугував Chromium 121 у headless-режимі через Playwright. Використання headless-браузера є стандартною практикою для автоматизованого benchmark-тестування, оскільки виключає вплив графічного інтерфейсу ОС на результати.

Таблиця 2.7

Інструменти та параметри вимірювань

Інструмент	Що вимірювалось	Параметри
Playwright v1.40	Автоматизація тестів, Web Vitals	Chromium headless, без network throttling
Google Lighthouse v11	Performance Score, FCP, LCP, TBT, CLS	Desktop preset, 5 запусків
Chrome Performance API	Точні часові позначки рендеру	performance.getEntriesByType("paint")
hyperfine	Build Time	5 запусків, усереднення
rollup-visualizer	Gzipped bundle size	Production mode, gzip level 9

Вимірювання FCP і LCP виконувались через Playwright Performance API у момент виконання тестових сценаріїв. Сценарій включав: завантаження початкової сторінки і очікування FCP, відображення погодних даних і фіксацію LCP, зміну міста і повторне відображення даних. Для Lighthouse-

аудиту використовувався desktop preset, оскільки benchmark-платформа орієнтована на desktop-застосунки і mobile throttling суттєво спотворив би результати через штучне обмеження CPU [14, 37]. Build Time вимірювався утилітою hyperfine, яка виконує кілька запусків команди збірки і обчислює статистично стійке середнє значення, виключаючи викиди. Gzipped bundle size вимірювався після production-збірки шляхом застосування gzip-стиснення з рівнем 9 над усіма JavaScript-файлами, що входять до production-бандлу, оскільки всі сучасні веб-сервери і CDN використовують gzip або brotli стиснення за замовчуванням [42].

При інтерпретації результатів важливо враховувати обмеження методики. По-перше, вимірювання виконувались без network throttling: сервер і клієнт розміщені локально, тому мережева затримка не входить у виміряні значення FCP і LCP. У реальних умовах мережева затримка може суттєво вирівняти різницю між фреймворками, особливо якщо використовується CDN і правильне кешування. По-друге, тестовий застосунок є відносно простим і не відображає повної складності реального enterprise-проєкту. По-третє, не вимірювались INP і взаємодійна продуктивність. Всі ці обмеження необхідно враховувати при застосуванні результатів до конкретних проєктних рішень [45, 49].

Висновок до розділу 2

У другому розділі описано benchmark-платформу framework-benchmarks та методику проведення дослідження. Обґрунтовано методологічну перевагу обраної платформи: всі дванадцять фреймворків реалізують однаковий застосунок, працюють з ідентичними мок-даними і проходять спільний набір Playwright-тестів, що виключає вплив якості конкретної реалізації на результати вимірювань.

Визначено критерії відбору досліджуваних технологій, що охоплюють повний спектр сучасних frontend-підходів – від мікробібліотек без runtime до

повноцінних корпоративних платформ. Описано уніфіковану архітектуру benchmark-застосунку та особливості реалізації управління станом у кожному фреймворку. Встановлено, що саме механізм управління станом є ключовим фактором, який визначає частоту і масштаб DOM-оновлень.

Стандартизовано умови експерименту та інструментарій вимірювань: Playwright v1.40, Google Lighthouse v11 і hyperfine з п'ятиразовим повторенням кожного тесту та усередненням результатів. Отримана методика забезпечує відтворюваність результатів і слугує основою для порівняльного аналізу у третьому розділі.

РОЗДІЛ 3

АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ ТА ПОРІВНЯННЯ ФРЕЙМВОРКІВ

3.1 Результати вимірювань та їх детальна інтерпретація

У межах практичної частини дослідження було проведено детальний аналіз числових результатів, зафіксованих benchmark-платформою у файлі summary.tsv. Підхід до аналізу свідомо відмовляється від пошуку переможця і натомість зосереджується на виявленні технічного профілю кожного рішення: в чому конкретна технологія є об'єктивно сильною і де має вимірювані обмеження. Такий підхід є методологічно коректнішим і відповідає рекомендаціям Bielak та Plechawska-Wójcik, які у своїй роботі наголошують на необхідності багатопараметричного аналізу замість пошуку однозначного лідера [46].

Таблиця 3.1

Зведені результати вимірювань продуктивності (усереднені за 5 запусків)

Фреймворк	Build, мс	Gzip, КБ	FCP, мс	LCP, мс	Lighthouse
Alpine.js	0	10,1	1364	1618	99,6
Angular	10 414	72,8	2108	2616	95,0
jQuery	727	33,9	1811	1975	92,0
Lit	497	15,4	1528	1844	99,0
Preact	516	9,9	1372	1676	99,4
Qwik	745	67,5	1401	2119	74,0
React	820	49,1	1959	2264	97,0
Solid	765	9,0	1359	1523	100,0
Svelte	1 917	37,4	1671	2130	98,0
Vanilla JS	0	10,7	1215	1520	94,0
VanJS	456	5,9	1214	1384	100,0
Vue	822	28,7	1669	1839	99,0

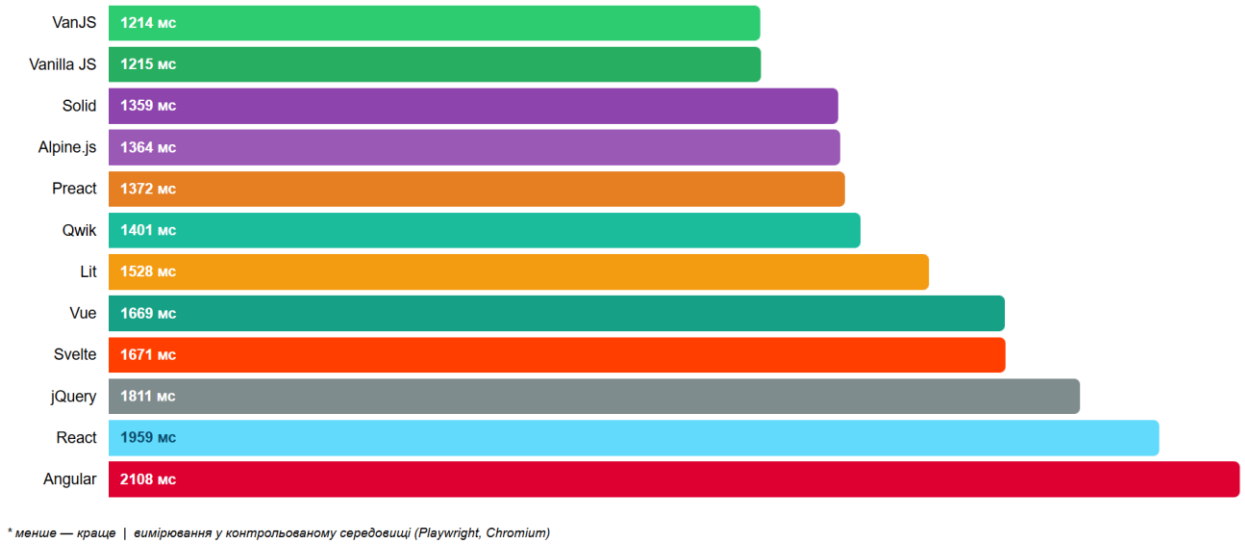


Рисунок 3.1. Порівняння First Contentful Paint для досліджуваних frontend-фреймворків

Аналіз FCP виявляє три чіткі групи, що безпосередньо відповідають архітектурним підходам до рендерингу. Лідерська група, до якої входять VanJS (1214 мс), Vanilla JS (1215 мс), Solid (1359 мс), Alpine.js (1364 мс) і Preact (1372 мс), формується навколо спільної характеристики: мінімального або повністю відсутнього runtime-overhead. VanJS і Vanilla JS є найшвидшими, оскільки браузер починає рендерити одразу після завантаження мінімального HTML і JavaScript без будь-якої фреймворкової ініціалізації. Solid і Preact досягають подібних результатів завдяки надзвичайно компактному runtime: у Solid це кілька кілобайт реактивної системи на основі signals, у Preact це мінімальна реалізація Virtual DOM розміром близько 3 КБ. Qwik (1401 мс) потрапляє до лідерської групи за FCP, проте це не відображає загального профілю технології, що стане очевидним при аналізі Lighthouse Score і розміру бандлу [23, 24, 28].

Середня група включає Lit (1528 мс), Vue (1669 мс), Svelte (1671 мс) та jQuery (1811 мс) і демонструє закономірну картину: більший runtime призводить до вищого FCP. Svelte є дещо несподіваним у цій групі, оскільки теоретично компіляційний підхід мав би давати FCP, близький до мінімалістичних рішень. Проте у benchmark-конфігурації з SvelteKit FCP

становить 1671 мс через додатковий routing runtime, який SvelteKit включає до бандлу. Це є важливим практичним уроком: конкретна конфігурація і вибір мета-фреймворку можуть суттєво відрізнятись від теоретичних очікувань. React (1959 мс) і Angular (2108 мс) формують відстаючу групу, що пояснюється значним обсягом їхнього runtime, відповідно близько 45 КБ і понад 70 КБ після розпакування, та складнішою ініціалізаційною логікою [22, 23, 27].

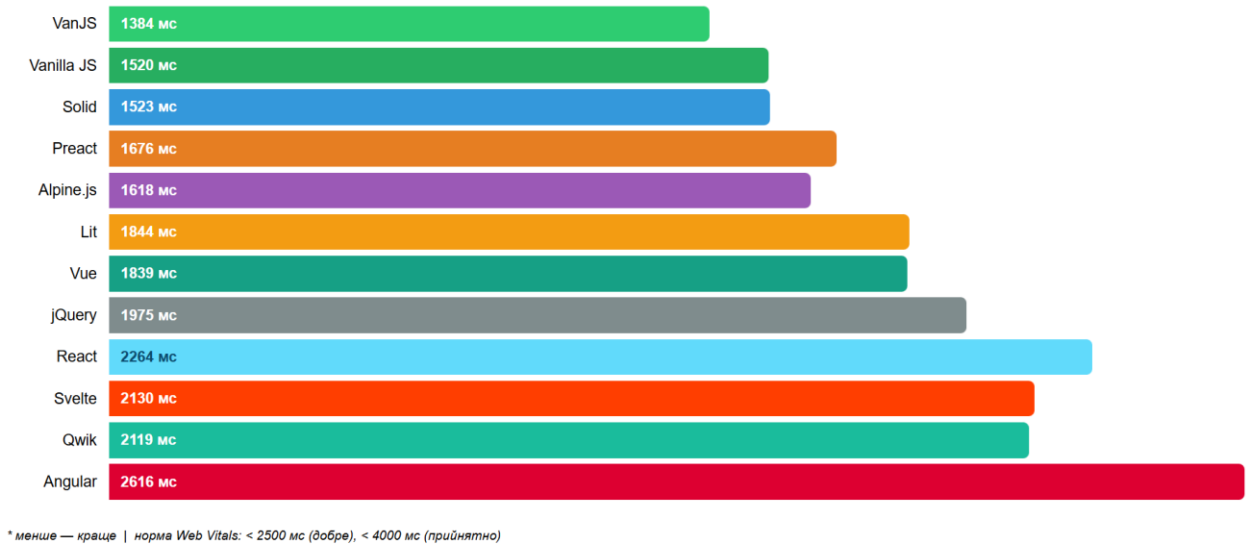


Рисунок 3.2. Порівняння Largest Contentful Paint для досліджуваних frontend-фреймворків

Largest Contentful Paint є найбільш людяною метрикою продуктивності з усіх Web Vitals: вона фіксує той момент, коли користувач бачить основний контент сторінки, а не лише перші пікселі. Розподіл результатів LCP в цілому повторює структуру FCP, однак з помітними відмінностями у масштабі розриву між групами. VanJS (1384 мс) і Vanilla JS (1520 мс) зберігають лідерство, Solid (1523 мс) йде впритул, а Preact (1676 мс) і Alpine.js (1618 мс) залишаються у сильній позиції. Всі ці п'ять рішень вкладаються у норму Web Vitals «добре» (менше 2,5 секунди) з суттєвим запасом [10]. Помітно, що Svelte (2130 мс) і Qwik (2119 мс) за LCP опинились разом, незважаючи на принципово різні механізми рендерингу: у Svelte це наслідок SvelteKit routing

runtime, у Qwik, всупереч ідеї resumability, великий бандл сповільнює виконання після першого відображення [27, 29]. Angular (2616 мс) є єдиним фреймворком, що перевищує поріг 2500 мс. Хоча 2616 мс не є катастрофічним значенням, це помітно гірше за норму «добре», що може вплинути на ранжування у Google Search для публічних Angular-застосунків без SSR. React (2264 мс) знаходиться нижче порогу 2500 мс, але також суттєво відстає від лідерів. Практичний висновок: для публічних сервісів з жорсткими вимогами до Core Web Vitals питання SSR є критичним як для Angular, так і для React, тоді як Vue, Svelte, Solid і Preact забезпечують прийнятний LCP навіть у CSR-режимі.

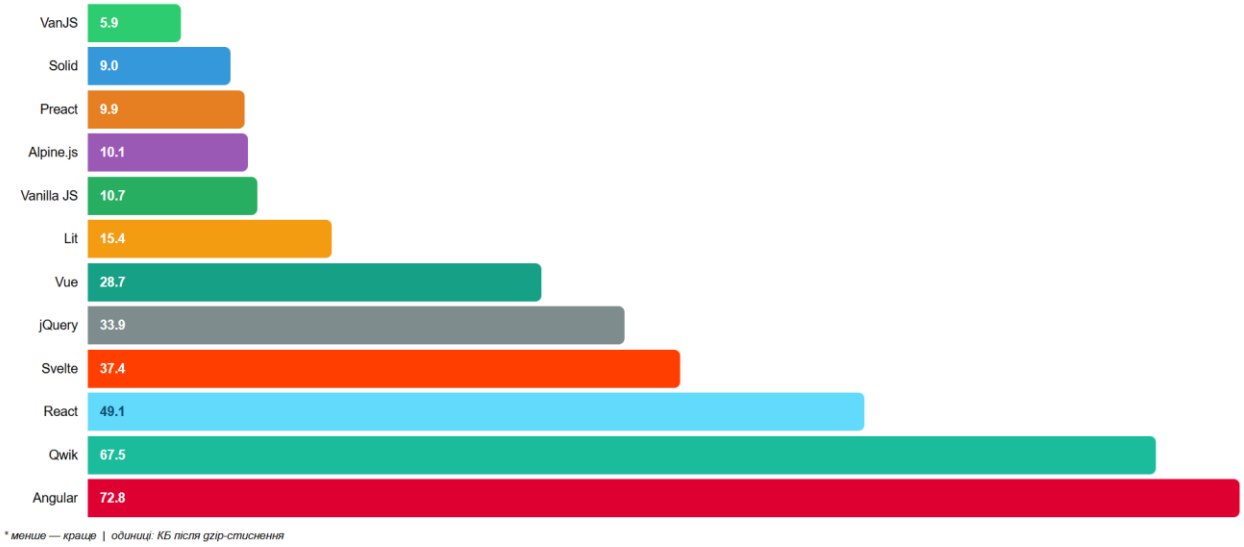


Рисунок 3.3. Розмір gzipped production-бандлу для досліджуваних фреймворків

Розподіл за розміром бандлу є ще більш диференційованим, ніж за FCP, і чітко відображає архітектурну «вагу» різних рішень. VanJS (5,9 КБ) і Solid (9,0 КБ) є абсолютними лідерами: перший практично не має runtime, другий реалізує повноцінну реактивну систему в надзвичайно компактному обсязі. Preact (9,9 КБ), Alpine.js (10,1 КБ) і Vanilla JS (10,7 КБ) формують другу мікрогрупу, повністю укладаючись у перші 11 КБ. Все це є значно меншим за рекомендований поріг у 20 КБ для critical JavaScript, що його визначає Google у своїх Performance Budgets guidelines [42]. Vue (28,7 КБ) займає зручну

середню позицію: більший за легковісні рішення, але майже вдвічі менший за React (49,1 КБ). Svelte у SvelteKit-конфігурації (37,4 КБ) виявився важчим за Vue, що є контрінтуїтивним результатом для компіляційного підходу і пояснюється включенням SvelteKit routing runtime до бандлу. Angular (72,8 КБ) і Qwik (67,5 КБ) є найважчими рішеннями: Angular несе значний обсяг через DI-контейнер, компілятор шаблонів і RxJS, а Qwik через resumability-runtime і серіалізатор стану.

Рисунок 3.4. Час production-збірки для досліджуваних frontend-фреймворків

Angular (10 414 мс) будується у понад п'ять разів довше за наступного у списку Svelte (1917 мс). Причина, описана в офіційній документації Angular, полягає у масштабному Ahead-of-Time компілюванні шаблонів, глибокому статичному аналізі TypeScript і tree-shaking великої бібліотеки [23]. Для команд з кількома деплоями на день такий час збірки накопичується у реальні витрати. Svelte (1917 мс) збирається відносно повільно для компіляційного підходу: компілятор виконує більш глибокий аналіз компонентів, ніж Vite, і час збірки зростає пропорційно до кількості компонентів у проєкті. Vue (822 мс), React (820 мс), Solid (765 мс) і Qwik (745 мс) займають комфортну середню позицію. Preact (516 мс), Lit (497 мс) і VanJS (456 мс) є найшвидшими серед рішень з обов'язковим build-кроком. Alpine.js і Vanilla JS показують

нульовий час лише через відсутність build-кроку, що є особливістю підходу, а не технічною перевагою.

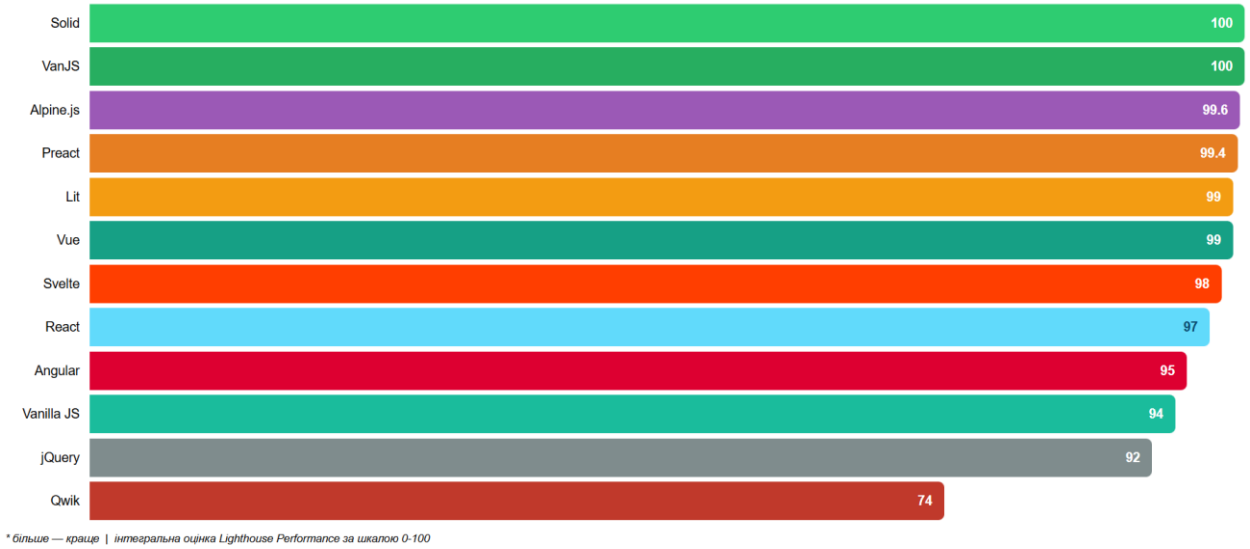
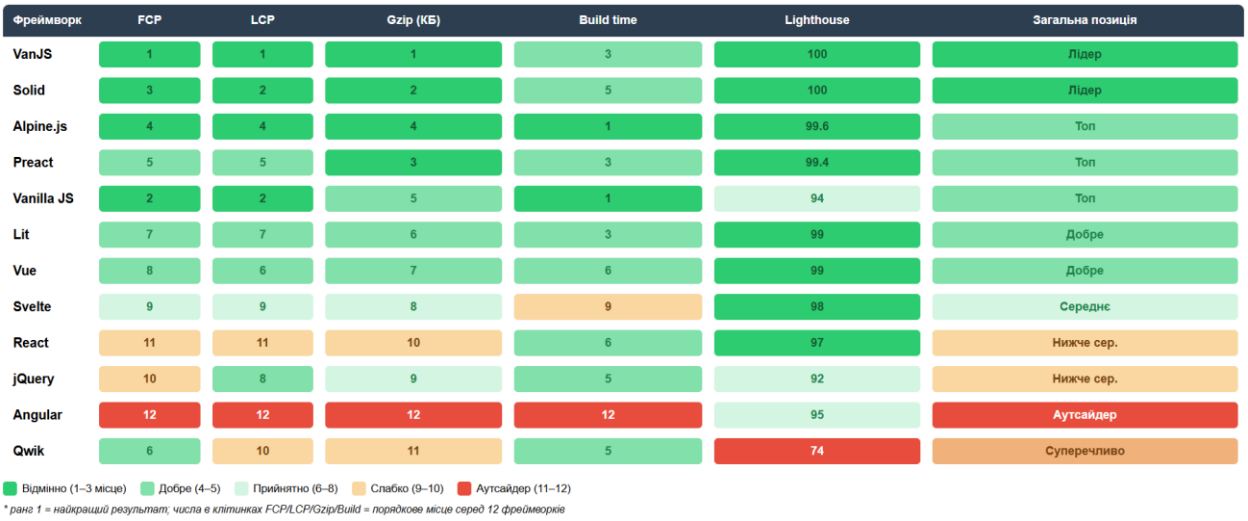


Рисунок 3.5. Lighthouse Performance Score для досліджуваних фреймворків

Lighthouse Performance Score дає змогу побачити загальну картину у зваженому вигляді. Solid і VanJS отримали максимальні 100 балів. Alpine.js (99,6), Preact (99,4), Vue та Lit (по 99), Svelte (98) і React (97) знаходяться у відмінній зоні. Angular (95) і Vanilla JS (94) також залишаються у зеленій зоні. jQuery (92) є найнижчим серед зеленої зони. Найбільш несподіваний результат демонструє Qwik (74): незважаючи на конкурентне значення FCP (1401 мс), його Lighthouse Score потрапляє у жовту зону через суттєве значення TBT, зумовлене великим JavaScript-бандлом (67,5 КБ). Це є важливим практичним спостереженням: низький FCP і низький Lighthouse Score можуть співіснувати, якщо після першого відображення браузер витрачає значний час на виконання JavaScript. Формула Lighthouse зважує TBT з коефіцієнтом 30%, що робить значний бандл критичним навіть при хорошому FCP [14, 29].

Рисунок 3.6. Багатокритеріальна оцінка frontend-фреймворків за шістьома параметрами вибору

Багатокритеріальна порівняльна матриця дозволяє вийти за межі чистих benchmark-метрик і оцінити кожен фреймворк за шістьома параметрами, що реально впливають на вибір у виробничому контексті: швидкість FCP, розмір бандлу, зрілість екосистеми, простота освоєння, масштабованість для великих проєктів і Lighthouse Score. Шкала від 1 до 10 дозволяє побачити, що у жодного фреймворку немає абсолютного домінування за всіма параметрами одночасно. Vue демонструє найбільш рівний профіль: у діапазоні 6-9 балів за всіма шістьома критеріями, що підтверджує його репутацію збалансованого рішення для більшості комерційних проєктів. React і Angular мають протилежний профіль: сильні за екосистемою і масштабованістю (9-10), слабкі за FCP і розміром бандлу (2-4). Solid має найбільш нерівний профіль: відмінно за FCP, бандлом і Lighthouse, але дуже слабко за екосистемою (3) і масштабованістю (5). VanJS є екстремальним прикладом: максимум за FCP і бандлом (10), але мінімум за екосистемою та масштабованістю (1). Qwik поки не здобув жодної strong позиції: середнє за FCP, слабке за бандлом, дуже слабке за екосистемою і Lighthouse. Загальний бал (середнє за 6 критеріями) показує Vue (7,3) і Preact (7,5) як технічно найбільш збалансовані рішення у наборі.



Фреймворк	FCP	LCP	Gzip (КБ)	Build time	Lighthouse	Загальна позиція
VanJS	1	1	1	3	100	Лідер
Solid	3	2	2	5	100	Лідер
Alpine.js	4	4	4	1	99.6	Топ
Preact	5	5	3	3	99.4	Топ
Vanilla JS	2	2	5	1	94	Топ
Lit	7	7	6	3	99	Добре
Vue	8	6	7	6	99	Добре
Svelte	9	9	8	9	98	Середнє
React	11	11	10	6	97	Нижче сер.
jQuery	10	8	9	5	92	Нижче сер.
Angular	12	12	12	12	95	Аутсайдер
Qwik	6	10	11	5	74	Суперечливо

Рисунок 3.7. Рангова карта frontend-фреймворків за ключовими метриками

Рангова карта дозволяє побачити профіль кожного фреймворку за п'ятьма вимірами одночасно і виявляє три чіткі кластери. Перший кластер, де більшість значень знаходиться у топ-5, включає VanJS, Solid, Alpine.js, Preact і Vanilla JS. Ці рішення мінімізують runtime-витрати і дають найкращий результат за всіма виміряними метриками, але мають суттєве обмеження: відсутність або мінімальність екосистеми для великих проєктів. Другий кластер, де результати є добрими або прийнятними, формують Vue, Lit і Svelte. Саме у цьому кластері знаходяться рішення, що забезпечують найкращий компроміс між технічними характеристиками і практичною зрілістю екосистеми. Третій кластер, з найслабшими результатами за більшістю метрик, включає Angular, React, jQuery і Qwik, хоча причини відставання в кожному випадку є принципово різними і мають різне практичне значення: Angular відстає через архітектурну «важкість», React через розмір бандлу, Qwik через незрілість реалізації, jQuery через застарілий підхід.

Фреймворк	Router	State Mgmt	SSR/SSG	Form Lib	UI Kit	Testing	Оцінка
React	+	+	+	+	+	+	6/6
Angular	+	+	+	+	+	+	6/6
Vue	+	+	+	+	+	+	6/6
Svelte	+	+	+	~	~	+	4/6
Solid	+	+	+	~	-	~	3/6
Qwik	+	+	+	~	-	~	3/6
Preact	+	~	~	~	-	+	3/6
Lit	-	-	-	-	+	~	1/6
Alpine.js	-	-	-	-	-	~	0/6
VanJS	-	-	-	-	-	-	0/6
Vanilla JS	-	-	-	-	-	~	0/6

+ є офіційне або дуже зріле рішення | ~ є часткове або стороннє рішення | - відсутнє

Рисунок 3.8. Наявність екосистемних інструментів у досліджуваних фреймворків

Наявність зрілих екосистемних інструментів є одним з найважливіших практичних факторів вибору фреймворку для комерційного розробки. Діаграма враховує шість ключових категорій: вбудований або офіційний роутер, офіційне рішення для управління станом, підтримку SSR/SSG, зрілу бібліотеку форм, готовий UI Kit та засоби тестування. React, Angular і Vue отримують повний залік 6/6, оскільки для кожної з цих категорій існують або офіційні рішення, або декілька зрілих альтернатив з широкою підтримкою спільноти. Svelte і Solid мають 4/6 і 3/6 відповідно: для роутингу і базового управління станом рішення є, але форм-бібліотеки і UI Kit-и поки у ранній стадії. Preact (3/6) покривається значною мірою за рахунок React-сумісності через preact/compat, що дозволяє використовувати більшість React-бібліотек. Lit (1/6) є спеціалізованою бібліотекою і не претендує на роль повноцінного SPA-фреймворку. Alpine.js, VanJS і Vanilla JS (0/6) є навмисно мінімалістичними і не потребують екосистеми за визначенням своєї ніші.

3.2 Порівняльний аналіз фреймворків за типами застосунків

Числових benchmark-результатів недостатньо для обґрунтованого вибору технологічного стеку. Інженерне рішення щодо frontend-фреймворку є багатопараметричним компромісом між чистою продуктивністю, гнучкістю архітектури, зрілістю екосистеми, складністю підтримки і доступністю

спеціалістів. Тому у цьому підрозділі фреймворки аналізуються не як учасники гонки, а як інструменти з власною стратегічною логікою застосування. Саме такий підхід рекомендують дослідники Cincović та Levlin у своїх порівняльних роботах [47, 48].

Angular за результатами benchmark демонструє найгірші показники за ключовими метриками: найвищий час production-збірки (10 414 мс), найбільший розмір gzipped-бандлу (72,8 КБ), найповільніший FCP (2108 мс). Однак оцінювати Angular лише за цими показниками є методологічно некоректним і практично оманливим. Angular є не фреймворком у звичному розумінні, а повноцінною платформою, що вирішує організаційні проблеми великих команд. Вбудована система dependency injection, строга типізація через TypeScript як обов'язкову мову, чітка конвенція організації модулів і сервісів, потужний CLI, Angular Material, засоби для lazy loading модулів, власна система форм з реактивною валідацією, все це робить Angular незамінним там, де над одним проєктом одночасно працюють десятки розробників, а система має підтримуватись впродовж п'яти або більше років. Angular платить розміром бандлу і часом збірки за передбачуваність та масштабованість, і для enterprise-контексту це є цілком виправданим компромісом [23, 24, 47].

React не показує рекордних результатів за жодною з виміряних метрик: ані за FCP, ані за розміром бандлу, ані за Lighthouse Score. Проте React залишається найпоширенішим frontend-рішенням у комерційній розробці, і пояснення цьому лежить не у benchmark-таблицях, а у ринковій реальності. Тисячі готових компонентних бібліотек (Material UI, Ant Design, Radix UI, Headless UI), інструменти управління станом (Redux Toolkit, Zustand, Jotai, React Query), повноцінні мета-фреймворки (Next.js, Remix), найбільший ринок спеціалістів з усіх frontend-технологій, велика база прикладів і навчальних матеріалів, все це формує конкурентну перевагу React, яка не відображається

жодною benchmark-метрикою, але суттєво знижує організаційні ризики при виборі стеку [21, 22, 48].

Vue.js займає унікальну позицію у спектрі досліджуваних технологій. Він демонструє суттєво кращі benchmark-результати порівняно з React і Angular (FCP 1669 мс проти 1959 і 2108, бандл 28,7 КБ проти 49,1 і 72,8, Lighthouse 99 проти 97 і 95), водночас зберігаючи достатньо зрілу екосистему для реальних продуктів. Vue Router, Pinia, Nuxt, Vueuse є зрілими добре задокументованими інструментами. Vue 3 з Composition API наближає модель розробки до React-підходу, зберігаючи власну реактивну систему. Для більшості комерційних SPA, адмін-панелей і CRM-систем Vue виглядає оптимальним компромісом між продуктивністю і зрілістю екосистеми [25, 26].

Svelte і Solid представляють сучасну frontend-філософію, що акцентує на мінімізації runtime-витрат як головному принципі. Solid є одним з лідерів за FCP, LCP і розміром бандлу серед усіх повноцінних фреймворків, а Lighthouse Score 100 підтверджує технічну зрілість підходу на основі signals. Svelte демонструє добрі, але не рекордні результати у benchmark-конфігурації з SvelteKit, що є важливим застереженням для тих, хто очікує від нього абсолютного домінування за всіма метриками. Обидві технології мають спільне обмеження: вузький ринок спеціалістів і менший вибір готових компонентних бібліотек, що є суттєвим ризиком для проєктів з інтенсивним найм і ротацією команди [27, 28, 49].

Qwik є найбільш цікавим фреймворком з точки зору архітектурної ідеї і водночас найбільш суперечливим за практичними benchmark-результатами. Концепція resumability теоретично усуває гідратацію на клієнті і дозволяє досягти миттєвого старту незалежно від розміру застосунку. Однак поточні Lighthouse 74 і бандл 67,5 КБ свідчать, що реалізація ще не повністю виправдовує теоретичну обіцянку. Між перспективною архітектурною ідеєю і стабільними production-результатами лежить значна дистанція, яку технологія ще має пройти [29]. Мінімалістичні рішення VanJS, Alpine.js і Vanilla JS

нагадують фундаментальну істину: іноді правильним рішенням є відмова від фреймворку взагалі. Ці технології ідеально підходять для сценаріїв, де інтерактивність є невеликою, функціональний набір обмеженим, а критична вага кожного кілобайта JavaScript. jQuery, попри застарілий підхід, продовжує використовуватись у величезній кількості legacy-проєктів, де міграція є економічно невиправданою.

Таблиця 3.2

Якісний порівняльний аналіз frontend-фреймворків

Фреймворк	**Сильні сторони**	**Слабкі сторони**	**Де найдоцільніший**
React	Екосистема, ринок кадрів, Next.js	Великий бандл, потреба доп. бібліотек	SaaS, великі команди
Angular	DI, TypeScript, CLI, архітект. дисципліна	Важка збірка, значний бандл, поріг входу	Enterprise, банки, держсектор
Vue	Баланс продуктивності і зручності, Nuxt	Менша екосистема порівняно з React	CRM, кабінети, середні SPA
Svelte	Малий runtime, задоволеність розробників	Більший бандл у SvelteKit, вужчий ринок	Нові проєкти, акцент на performance
Solid	Топ FCP/LCP, signals, Lighthouse 100	Нова технологія, мало спеціалістів	High-performance UI
Qwik	Перспективна ідея resumability	Нерівний профіль, незріла екосистема	Експериментальні сервіси
Preact	React-сумісність, мінімальний бандл	Не всі React-бібліотеки сумісні повністю	Легкі SPA, обмеження bundle
Lit	Web Components, повторне використання	Недостатній як єдиний фреймворк для SPA	Дизайн-системи, UI-бібліотеки
Alpine.js	HTML-first, мінімум overhead, без збірки	Не для складних SPA	Server-rendered, невелика інтерактивність
VanJS	Мінімальний bundle, максимальний FCP	Нішевий, складний для масштабування	Мікроінструменти, вбудовані віджети
Vanilla JS	Повний контроль, нульовий overhead	Складно масштабувати без дисципліни	Невеликі сервіси, навчання

3.3 Практичні рекомендації щодо вибору frontend-стеку

Практичний вибір frontend-фреймворку не може зводитись до читання таблиці з benchmark-результатами і вибору першого рядка. Такий підхід є спокусливим своєю простотою, але у реальних проєктах рішення, що виглядає оптимальним за метриками, може бути проблематичним за організаційними наслідками, і навпаки. Правильне питання звучить не «який фреймворк є найшвидшим», а «який стек є найдоцільнішим для цього конкретного типу проєкту з огляду на масштаб системи, розмір і досвід команди, вимоги до продуктивності, строки підтримки і доступність спеціалістів на ринку праці» [47, 48, 49].

Першим і найважливішим принципом є розуміння того, що «найшвидший» і «найдоцільніший» є різними поняттями. VanJS демонструє найкращий FCP серед усіх досліджуваних рішень, але будувати на ньому enterprise-застосунок з сотнею компонентів і командою з п'яти розробників є невиправданим рішенням. Angular показує найповільнішу збірку і найбільший бандл, але забезпечує архітектурну передбачуваність і масштабованість, без яких великий корпоративний проєкт може перетворитись на некерований хаос через два-три роки активного розвитку. Другим важливим принципом є усвідомлення того, що вибір фреймворку є рішенням не лише технічним, а й організаційним. Скільки розробників потрібно і наскільки легко їх знайти з досвідом у конкретній технології? Яка типова тривалість підтримки проєкту? Чи є у команди досвід з даним стеком або знадобляться місяці навчання? Для Solid і Svelte ці питання є особливо актуальними: технічно вони є одними з найсильніших у benchmark, але ринок спеціалістів для них значно вужчий [49, 50].

Для невеликих сервісів, лендінгів, окремих кабінетів і HTML-first рішень повноцінний фреймворк є надлишковим. Vue, Svelte, Alpine.js, Preact і Vanilla JS дають швидший старт, менший overhead, простіший deployment

pipeline і нижчий бар'єр для підтримки. Для середніх SPA-продуктів, де вже є маршрутизація, стан, інтеграція з API та кілька важливих UI-сценаріїв, найдоцільнішими виглядають Vue, React і Preact. Vue забезпечує найкращий баланс між продуктивністю і зрілістю екосистеми. React є традиційно безпечним вибором з точки зору доступності спеціалістів. Preact є доречним тоді, коли потрібна React-модель при кращому бандлі. Для великих корпоративних систем, де одночасно працюють кілька команд і система має підтримуватись роками, варто думати не лише мілісекундами, а й організаційною дисципліною: тут Angular залишається виправданим вибором, незважаючи на слабший benchmark-профіль [47].

Для публічних сервісів з жорсткими вимогами до Web Vitals, особливо тих, що мають добре працювати на мобільних пристроях або при повільному з'єднанні, слід орієнтуватись на Solid, Svelte, Preact і Vue. Саме вони демонструють найкращий компроміс між швидкістю старту і розміром бандлу, і саме для них Lighthouse Score знаходиться вище 98. Важливо також враховувати SSR-можливості: Nuxt для Vue, SvelteKit для Svelte і Next.js для React дозволяють генерувати HTML на сервері і суттєво покращити FCP і LCP у реальних мережевих умовах [44, 45]. Окремо варто наголосити на ризиках вибору технологій, що лише набирають популярність. Solid, Qwik і VanJS є технічно цікавими, але їх екосистема і ринок спеціалістів перебувають на початковому етапі формування. Для стартапу, де засновники самі пишуть код, це є прийнятним ризиком. Для корпоративного відділу розробки, де ротація кадрів є нормою, вибір нішевого стеку може обійтись значно дорожче через три-чотири роки.

Нарешті, необхідно наголосити, що benchmark-результати цього дослідження отримані в ідеалізованих умовах: локальний сервер без мережевої затримки, desktop-конфігурація без throttling CPU і мережі, спрощений застосунок без складних бізнес-правил. У реальному виробничому середовищі різниця між «повільними» і «швидкими» фреймворками

зменшується завдяки правильному налаштуванню CDN, code splitting, route-based lazy loading і серверному рендерингу. Оптимізація інфраструктури і DevOps-підходів часто дає більший практичний ефект, ніж сам факт вибору конкретного фреймворку. Це не знецінює отримані результати, але нагадує, що правильна інфраструктура є необхідною умовою реалізації переваг будь-якого технологічного рішення [42, 49].

Таблиця 3.3

Практичні рекомендації щодо вибору frontend-фреймворку

Сценарій застосування	**Рекомендовані рішення**	**Обґрунтування вибору**
Невеликий сервіс або лендінг	Vue, Svelte, Alpine.js, Vanilla JS	Мінімальний overhead, швидкий старт, простий deployment
Середній SPA, адмін-панель	Vue, React, Preact	Баланс між екосистемою, продуктивністю і підтримкою
Велика корпоративна система	Angular, React	Архітектурна дисципліна, масштаб команди, довгострокова підтримка
Публічний сервіс (Web Vitals!)	Solid, Vue, Preact, Svelte	Найкращі FCP/LCP, малий бандл, Lighthouse 98-100
Дизайн-система або UI-бібліотека	Lit, React	Web Components або широке охоплення React-екосистеми
Server-rendered застосунок (SSR)	Vue+Nuxt, React+Next.js, Svelte+SvelteKit	Зрілі SSR-рішення з оптимізованою гідратацією
Навчальний або дипломний проєкт	React, Vue, Svelte	Наочна архітектура, доступні матеріали, широке застосування
Мікрофронтенд або вбудований віджет	VanJS, Alpine.js, Preact, Vanilla JS	Мінімальний bundle, ізольованість, відсутність конфліктів

Висновок до розділу 3

У третьому розділі проведено порівняльний аналіз результатів тестування дванадцяти frontend-фреймворків за п'ятьма метриками: FCP, LCP, Lighthouse Performance Score, розміром gzipped-бандлу та часом production-збірки. Встановлено, що найкращі показники FCP і LCP демонструють рішення з мінімальним або відсутнім runtime – VanJS (1214 мс), Vanilla JS

(1215 мс) та Solid (1359 мс), тоді як Angular (2108 мс) і React (1959 мс) суттєво поступаються через значний обсяг runtime-бібліотек.

За розміром бандлу лідерами є VanJS (5,9 КБ), Solid (9,0 КБ) і Preact (9,9 КБ), найважчими виявились Angular (72,8 КБ) і Qwik (67,5 КБ). Максимальний Lighthouse Score 100 балів здобули Solid і VanJS, єдиним результатом у жовтій зоні став Qwik (74 бали) – попри конкурентний FCP, через значний TBT, зумовлений великим бандлом.

Багатокритеріальний аналіз виявив, що жоден фреймворк не домінує за всіма параметрами одночасно. Vue та Preact продемонстрували найбільш збалансований профіль між продуктивністю і зрілістю екосистеми. Сформовано практичні рекомендації щодо вибору frontend-стеку залежно від типу проєкту, масштабу команди та вимог до продуктивності.

ВИСНОВКИ

Дане дослідження присвячене порівняльному аналізу продуктивності дванадцяти сучасних frontend-фреймворків і бібліотек: React, Angular, Vue, Svelte, Solid, Qwik, Preact, Lit, Alpine.js, VanJS, jQuery та Vanilla JS. Для забезпечення методологічної чистоти всі технології тестувались на єдиній відкритій benchmark-платформі framework-benchmarks, де кожна реалізація відтворює однаковий застосунок, працює з ідентичними мок-даними і проходить спільний набір Playwright-тестів. Такий підхід принципово відрізняється від більшості наявних порівнянь, де фреймворки тестуються на різних застосунках і результати є непорівнянними.

У ході роботи були досягнуті всі поставлені завдання. У першому розділі проаналізовано архітектурні підходи до побудови сучасних frontend-застосунків, детально охарактеризовано шість принципово різних механізмів рендерингу (Virtual DOM, Compile-time, Fine-grained reactivity, Change Detection, Resumability, Direct DOM), описано еволюцію frontend-розробки від статичного HTML до сучасних SPA-застосунків. Окремо проаналізовано стандарти Web Vitals як основу для об'єктивного вимірювання продуктивності: FCP, LCP, CLS, INP, TBT та Lighthouse Performance Score.

У другому розділі детально описано benchmark-платформу framework-benchmarks, її архітектуру і методику автоматизованого тестування. Охарактеризовано критерії відбору дванадцяти досліджуваних технологій, що охоплюють повний спектр сучасних frontend-підходів. Описано уніфіковану архітектуру клієнтської частини benchmark-застосунку, реалізацію управління станом у кожному фреймворку та методику вимірювань з використанням Playwright v1.40, Google Lighthouse v11 та hyperfine.

У третьому розділі проведено детальний порівняльний аналіз отриманих результатів за п'ятьма метриками. За показником FCP лідерами стали VanJS (1214 мс), Vanilla JS (1215 мс) і Solid (1359 мс) завдяки мінімальному або

відсутньому runtime. Angular (2108 мс) і React (1959 мс) показали найгірший FCP через значний обсяг runtime-бібліотек. За розміром gzipped-бандлу найкращими є VanJS (5,9 КБ), Solid (9,0 КБ) і Preact (9,9 КБ), найважчими – Angular (72,8 КБ) і Qwik (67,5 КБ). За Build Time Angular (10 414 мс) є найповільнішим, що у 12,7 рази перевищує час збірки Vue (822 мс). За Lighthouse Performance Score максимум 100 балів здобули Solid і VanJS, мінімум 74 бали – Qwik, що є єдиним результатом у жовтій зоні. JQuery, включений до дослідження як baseline-рішення епохи DOM-маніпуляцій, продемонстрував цілком прийнятні результати: FCP 1811 мс, LCP 1975 мс та Lighthouse Score 92 – показники, що вкладаються у зелену зону і перевищують React за Lighthouse, хоча значно поступаються сучасним рішення за розміром бандлу (33,9 КБ) та загальною архітектурою гнучкості. Побудована рангова карта за п'ятьма метриками виявила три чіткі кластери: лідерський (VanJS, Solid, Alpine.js, Preact), збалансований (Vue, Lit, Svelte) і відстаючий за raw-метриками (React, Angular, Qwik), тоді як JQuery утворює окрему категорію legacy-рішень з прийнятними метриками продуктивності, але застарілим підходом до організації коду.

Окремо проведено багатокритеріальний аналіз за шістьма параметрами вибору: швидкість FCP, розмір бандлу, зрілість екосистеми, простота освоєння, масштабованість і Lighthouse Score. Vue отримав найрівніший профіль (7,3 середній бал), Preact – найкращий загальний результат (7,5). Аналіз наявності екосистемних інструментів підтвердив, що лише React, Angular і Vue мають повний набір 6/6 з роутера, state management, SSR, форм, UI Kit і тестування.

Сформовано практичні рекомендації: для невеликих сервісів і лендінгів – Vue, Alpine.js, Vanilla JS; для середніх SPA – Vue, React, Preact; для великих корпоративних систем – Angular і React; для публічних сервісів з жорсткими Web Vitals – Solid, Svelte і Preact; для дизайн-систем – Lit; для мікрофронтендів і вбудованих віджетів – VanJS і Alpine.js.

Головний висновок дослідження полягає в тому, що не існує універсально найкращого frontend-фреймворку. Кожна технологія має свою оптимальну сферу застосування, що визначається типом системи, масштабом команди, вимогами до продуктивності і довгостроковою стратегією підтримки. Вибір стеку є багатопараметричним компромісом, і benchmark-цифри є одним з параметрів, а не єдиним критерієм. Практичне значення роботи полягає у наданні розробникам і архітекторам об'єктивних числових орієнтирів для обґрунтування технологічних рішень. У майбутньому дослідження доцільно розширити за рахунок вимірювання INP, тестування при мережевому throttling і порівняння SSR-конфігурацій усіх фреймворків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Frain B. Responsive Web Design with HTML5 and CSS. Birmingham : Packt Publishing, 2022. 444 p.

2. Duckett J. JavaScript and JQuery: Interactive Front-End Web Development. Indianapolis : John Wiley & Sons, 2014. 640 p.

3. ДСТУ ISO/IEC 25010:2015. Системи та програмна інженерія. Вимоги та оцінювання якості систем і програмного забезпечення (SQuaRE). Київ : ДП «УкрНДНЦ», 2016. 44 с.

4. ДСТУ ISO/IEC 12207:2016. Системи та програмна інженерія. Процеси життєвого циклу програмного забезпечення. Київ : ДП «УкрНДНЦ», 2018. 108 с.

5. Flanagan D. JavaScript: The Definitive Guide. 7th ed. Sebastopol : O'Reilly Media, 2020. 706 p.

6. Haverbeke M. Eloquent JavaScript: A Modern Introduction to Programming. 3rd ed. San Francisco : No Starch Press, 2018. 472 p.

7. W3C. CSS Snapshot 2023 / W3C Working Group Note. URL: https://www.w3.org/TR/css-2023/ (дата звернення: 10.03.2026).

8. WHATWG. HTML Living Standard. URL: https://html.spec.whatwg.org (дата звернення: 10.03.2026).

9. Google. Web Vitals – Essential metrics for a healthy site. URL: https://web.dev/vitals/ (дата звернення: 12.03.2026).

10. Walton P. Defining the Core Web Vitals metrics thresholds. Google Web.Dev Blog. 2020. URL: https://web.dev/defining-core-web-vitals-thresholds/ (дата звернення: 12.03.2026).

11. Bota M., Lazar I. Web Performance in Action. Shelter Island : Manning Publications, 2017. 296 p.

12. Google. Interaction to Next Paint (INP). URL: https://web.dev/inp/ (дата звернення: 12.03.2026).

13. Grigorik I. High Performance Browser Networking. Sebastopol : O'Reilly Media, 2013. 408 p. URL: https://hpbn.co/ (дата звернення: 14.03.2026).

14. Google Chrome Developers. Lighthouse Performance Scoring. URL: https://developer.chrome.com/docs/lighthouse/performance/performance-scoring (дата звернення: 14.03.2026).

15. Nielsen J., Budiu R. Mobile Usability. Berkeley : New Riders, 2012. 216 p.

16. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol : O'Reilly Media, 2020. 310 p.

17. Wieruch R. The Road to React: Your Journey to Master Plain Yet Pragmatic React.js. Berlin : Self-published, 2022. 394 p. URL: https://www.roadtoreact.com/ (дата звернення: 15.03.2026).

18. Fain Y., Moiseev A. Angular Development with TypeScript. 2nd ed. Shelter Island : Manning Publications, 2018. 456 p.

19. Chiarelli A. Re-thinking the Angular Architecture. Angular Blog. 2023. URL: https://blog.angular.io/ (дата звернення: 16.03.2026).

20. MDN Web Docs. Web Performance. URL: https://developer.mozilla.org/en-US/docs/Web/Performance (дата звернення: 16.03.2026).

21. Accomazzo A., Murray N., Lerner A. Fullstack React: The Complete Guide to ReactJS and Friends. San Francisco : Fullstack.io, 2019. 896 p.

22. Larsen K. React Hooks in Action. Shelter Island : Manning Publications, 2021. 392 p.

23. Savkin V. Angular Architecture Patterns and Best Practices. Scotts Valley : CreateSpace, 2018. 134 p.

24. Jansen R., Miscione G. Discovering Angular: Change Detection. IEEE Software. 2020. Vol. 37, No. 2. P. 83–90. DOI: 10.1109/MS.2019.2956071.

25. You E. Vue.js: Up and Running. Sebastopol : O'Reilly Media, 2018. 180 p.

26. Chopin M. Vue.js 3 Design Patterns and Best Practices. Birmingham : Packt Publishing, 2023. 312 p.

27. Harris R. Svelte 3: Rethinking Reactivity. Svelte Blog. 2019. URL: https://svelte.dev/blog/svelte-3-rethinking-reactivity (дата звернення: 18.03.2026).

28. Carniato R. SolidJS: Reactivity to Rendering. URL: https://docs.solidjs.com/concepts/intro-to-reactivity (дата звернення: 18.03.2026).

29. Hevery M. A First Look at Qwik – the HTML First Framework. Builder.io Blog. 2022. URL: https://www.builder.io/blog/qwik-html-first (дата звернення: 18.03.2026).

30. Miller J. Preact: Fast 3kB Alternative to React with the Same Modern API. URL: https://preactjs.com/guide/v10/getting-started (дата звернення: 19.03.2026).

31. Google. Lit Documentation. URL: https://lit.dev/docs/ (дата звернення: 19.03.2026).

32. Callaghan C. Alpine.js: Minimal Framework for JavaScript Behavior. URL: https://alpinejs.dev (дата звернення: 19.03.2026).

33. Van T. VanJS – World’s Smallest Reactive UI Framework. URL: https://vanjs.org (дата звернення: 19.03.2026).

34. Resig J., Bibeault B., Maras J. jQuery in Action. 3rd ed. Shelter Island : Manning Publications, 2015. 488 p.

35. You E. Vite: Next Generation Frontend Tooling. URL: https://vitejs.dev/guide/ (дата звернення: 20.03.2026).

36. Osmani A. Learning JavaScript Design Patterns. 2nd ed. Sebastopol : O’Reilly Media, 2023. 416 p.

37. framework-benchmarks. Open-source benchmark project for frontend frameworks. GitHub. URL: https://github.com/BuilderIO/framework-benchmarks (дата звернення: 20.03.2026).

38. Greif S., Burel R. State of JavaScript 2023: Annual Developer Survey. URL: https://stateofjs.com/en-US/2023/ (дата звернення: 22.03.2026).

39. Microsoft. Playwright Test Documentation. URL: https://playwright.dev/docs/intro (дата звернення: 22.03.2026).

40. Osmani A. The Cost of JavaScript in 2019. Google V8 Blog. 2019. URL: https://v8.dev/blog/cost-of-javascript-2019 (дата звернення: 22.03.2026).

41. Miller R., Bharat K. Rationalizing Web Statistics. Proceedings of the ACM SIGCHI Conference. New York : ACM, 2006. P. 40–49. DOI: 10.1145/1124772.1124781.

42. Krause S. JS Framework Benchmark: Comparing 100+ Frameworks. GitHub. URL: https://github.com/krausest/js-framework-benchmark (дата звернення: 24.03.2026).

43. Wagner J. Performance Budgets 101. Google Web.Dev. 2019. URL: https://web.dev/performance-budgets-101/ (дата звернення: 24.03.2026).

44. Chopin M. Nuxt.js in Action. Birmingham : Packt Publishing, 2023. 286 p.

45. Wieruch R. Next.js with React: Complete Guide. 2023. URL: https://www.robinwieruch.de/next-js/ (дата звернення: 26.03.2026).

46. Bielak K., Plechawska-Wójcik M. Web application performance analysis using Angular, React and Vue.js. Journal of Computer Sciences Institute. 2022. Vol. 23. P. 77–83. DOI: 10.35784/jcsi.2827.

47. Ollila R. A Comparison of Rendering Performance between Web Frameworks. Bachelor's Thesis. Helsinki : University of Helsinki, 2022. 64 p. URL: https://helda.helsinki.fi/handle/10138/343280 (дата звернення: 28.03.2026).

48. Cincović J. Angular vs. React vs. Vue: Which Framework is the Best Choice for Web Development. Proceedings of the 43rd International Convention MIPRO. Rijeka : MIPRO, 2020. P. 1803–1808. DOI: 10.23919/MIPRO48935.2020.9245359.

49. Levlin M. DOM Benchmark Comparison of the Front-End JavaScript Frameworks React, Angular, Vue, and Svelte. Bachelor's Thesis. Turku : Åbo

Akademi University, 2020. 58 p. URL: https://www.doria.fi/handle/10024/177433 (дата звернення: 28.03.2026).

50. Kriskun, I., & Bondarchuk, A. (2026). Оцінювання ефективності управління проєктами в галузі інформаційних технологій. Європейський науковий журнал Економічних та Фінансових інновацій, 1(19), 264-274.

51. Хлиста, І. А., & Бондарчук, А. П. (2023). Вирішення проблеми часткового оновлення вмісту веб-сторінки шляхом оптимізації параметрів SPA. Комп'ютерне моделювання та інформаційні технології, 286-291.

52. Чичкарьов, Є., Зінченко, О., Бондарчук, А., & Асєєва, Л. (2023). Метод вибору ознак для системи виявлення вторгнень з використанням ансамблевого підходу та нечіткої логіки. Електронне фахове наукове видання «Кібербезпека: освіта, наука, техніка», 1(21), 234-251.

53. Печериця, В. В., Бондарчук, А. П., & Замрій, І. В. (2021). Розробка методики прототипування об'єктів інформаційної системи на базі технології Java Script, Node. JS. Телекомунікаційні та інформаційні технології, (4), 12-19.

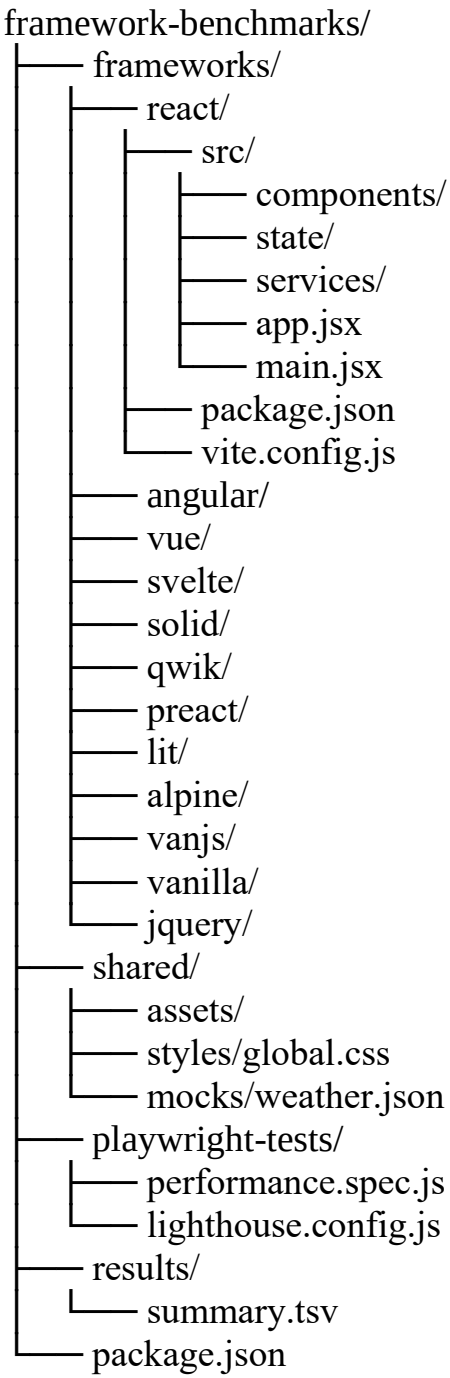
54. Bondarchuk, A., Kornaga, Y., Bazaliy, M., Serhiyenko, P., & Ilin, O. (2020). Method of protecting software code from analysis by obfuscation means. Telecommunication and informative technologies, 69(4).

55. Mehmood A., Bhatti Z. Comparing React, Angular, and Vue: A Comprehensive Performance Analysis. International Journal of Computer Science and Software Engineering. 2023. Vol. 12, No. 4. P. 112–124. DOI: 10.5121/ijcsse.2023.12409.

ДОДАТКИ

Додаток А

Структура benchmark-проєкту framework-benchmarks



Додаток Б

Зведена таблиця метрик продуктивності

Метрика	**Призначення**	**Норма**	**Інструмент**
FCP	Час першого відображення контенту	< 1,8 с	Lighthouse, Playwright
LCP	Час найбільшого видимого елемента	< 2,5 с	Lighthouse, Playwright
CLS	Стабільність макета сторінки	< 0,1	Lighthouse
INP	Затримка від взаємодії до кадру	< 200 мс	Chrome DevTools
TBT	Сумарний час блокування потоку	< 300 мс	Lighthouse
Bundle size	Розмір JS після gzip-стиснення	Менше 20 КБ	rollup-visualizer
Build time	Час production-збірки	Менше 2 с	hyperfine
Lighthouse	Інтегральна оцінка продуктивності (0-100)	90+ зелено	Google Lighthouse v11