Keywords: *automatic text classification, bibliographic information systems, machine learning, BERT, support vector machines, deep learning, natural language processing, text vectorisation.*

Одержано редакцією 15.11.2025 р.
Прийнято до публікації 17.12.2025 р.

УДК 004.415.2

DOI 10.31651/2076-5886-2025-1-85-95

PACS 07.05.Tp, 89.20.Ff

ТКАЧЕНКО Олексій Олексійович
студент спеціальності «Інформаційні системи та технології» Черкаського національного університету імені Богдана Хмельницького

ДІДКОВСЬКИЙ Руслан Михайлович,
доктор технічних наук, доцент, доцент кафедри прикладної математики та інформатики Черкаського національного університету імені Богдана Хмельницького
e-mail: didkovskyirm@vu.cdu.edu.ua
ORCID 0000-0002-5166-7564

ХОВАЙБА Дарина Євгенівна
викладач кафедри прикладної математики та інформатики Черкаського національного університету імені Богдана Хмельницького
e-mail:
tovstopiat.daryna1618@vu.cdu.edu.ua
ORCID 0009-0001-4202-0451

ДОСЛІДЖЕННЯ МІКРОФРОНТЕНД АРХІТЕКТУРИ ДЛЯ ПОБУДОВИ МАСШТАБОВАНИХ ВЕБ-СИСТЕМ

У роботі досліджено підходи до проектування та реалізації масштабованих веб-систем на основі мікрофронтенд архітектури з використанням технологій Module Federation та інструментарію Nx. Виконано порівняльний аналіз методів інтеграції мікрофронтендів: інтеграції на етапі збірки, ізоляції через iframe та інтеграції на етапі виконання. Обґрунтовано вибір патерну Shell-Remote як основи для проектування розподіленої системи управління задачами. Розроблено методику організації монорепозиторію з чітким поділом на шари (Applications, Feature Libraries, Shared UI, Data Access), що забезпечує повторне використання коду та уникнення дублювання. Проведено експериментальне дослідження ефективності запропонованого підходу: встановлено, що використання бандлера Rspack скорочує час збірки більш ніж у 10 разів порівняно із Webpack, а застосування стратегії ледачого завантаження (Lazy Loading) зменшує обсяг початкового завантаження сторінки на 81%. Отримані результати підтверджують доцільність застосування досліджуваного архітектурного підходу для побудови корпоративних (Enterprise-рівня) веб-систем.

Ключові слова*: мікрофронтенд архітектура, Module Federation, монорепозиторій, Nx, Rspack, Shell-Remote, масштабованість, продуктивність збірки.*

Вступ

Сучасний етап розвитку веб-технологій характеризується кардинальними змінами у парадигмі розробки програмного забезпечення. Веб-застосунки перетворилися зі статичних інформаційних сторінок на складні корпоративні системи, функціональність яких не поступається настільним програмним продуктам. В умовах цифрової трансформації бізнесу ключовими вимогами до таких систем стають не лише надійність та продуктивність, але й гнучкість архітектури та здатність до швидкого масштабування.

Проблема архітектурного проектування великих веб-систем набуває особливої актуальності, коли над продуктом одночасно працюють десятки команд розробників. Традиційні монолітні підходи у таких умовах стають бар'єром для подальшого розвитку: зростає надмірна зв'язаність компонентів, ускладнюється процес розгортання, уповільнюються процеси інтеграції.

Концепція мікрофронтендів (Micro Frontends) виникла як відповідь на обмеження монолітних клієнтських застосунків та базується на адаптації принципів мікросервісної архітектури для клієнтської частини веб-систем [1, 2]. Вона пропонує розбиття інтерфейсу на незалежні функціональні блоки, що розробляються, тестуються та розгортаються автономними командами. Проте впровадження розподілених систем у фронтенді породжує нові інженерні виклики, пов'язані з оркестрацією модулів, управлінням спільним станом та оптимізацією продуктивності.

Метою роботи є обґрунтування методики проектування та реалізації масштабованих веб-систем на основі мікрофронтенд архітектури з використанням технології динамічної федерації модулів (Module Federation) та порівняння її ефективності з традиційними підходами.

Для досягнення поставленої мети вирішено такі завдання:

1. Проаналізувати існуючі підходи до інтеграції мікрофронтендів та визначити їх переваги і недоліки.
2. Розробити архітектурний шаблон розподіленої системи на основі патерну Shell-Remote.
3. Провести порівняльний аналіз метрик продуктивності збірки та завантаження.

Виклад основного матеріалу

1. Аналіз підходів до реалізації мікрофронтенд архітектури

Еволюція архітектур веб-застосунків відбувалась у декілька ключових етапів. На ранніх стадіях домінувала модель генерації сторінок на стороні сервера (Server-Side Rendering). Поява AJAX та JavaScript-фреймворків (Angular, React, Vue) зумовила перехід до односторінкових застосунків (Single Page Applications, SPA), де браузер завантажує єдиний HTML-файл та JavaScript-пакет, що відповідає за відтворення інтерфейсу та маршрутизацію.

Хоча SPA суттєво покращили інтерактивність, вони породили нову проблему – «Frontend Monolith». Зі зростанням функціональності кодова база клієнтської частини стає надмірно великою. Основними недоліками монолітних SPA у контексті великих команд є: надмірна зв'язаність компонентів, конфлікти злиття коду (merge conflicts), технологічна стагнація та зростання часу збірки.

Концепція мікрофронтендів, описана Кемом Джексоном та Мартіном Фаулером [1], пропонує декомпозицію інтерфейсу на незалежні застосунки. Існують три принципово відмінні стратегії їх інтеграції.

Інтеграція на етапі збірки (Build-time Integration) базується на публікації кожного мікрофронтенду як окремого npm-пакету. Батьківський застосунок підключає ці пакети

як залежності. Перевагою є звичний процес розробки, а недоліком – тісна зв'язаність: для оновлення будь-якого модуля необхідно перезбирати та розгортати весь застосунок, що нівелює ключову перевагу мікрофронтендів – незалежність розгортання.

Інтеграція через iframe є найстарішим методом ізоляції. Серед критичних недоліків – складність адаптивного дизайну, дублювання бібліотек (кожен iframe завантажує власні копії React тощо) та труднощі міжвіконної комунікації через postMessage API [3].

Інтеграція на етапі виконання (Run-time Integration) є найбільш прогресивним підходом. Кожен мікрофронтенд розгортається за власною URL-адресою, а батьківський застосунок завантажує необхідний код динамічно у браузері. Сучасна реалізація цього підходу базується на технології Module Federation, представленій у Webpack 5 [4].

Таблиця 1 узагальнює порівняльний аналіз методів.

Таблиця 1

Порівняльний аналіз методів інтеграції мікрофронтендів

Характеристика	Build-time (NPM)	Iframe	Run-time (Module Federation)
Незалежність розгортання	Низька	Висока	Висока
Ізоляція CSS/JS	Низька	Абсолютна	Середня (керується розробником)
Продуктивність (розмір коду)	Оптимізована	Низька (дублювання)	Висока (спільні бібліотеки)
Складність налаштування	Низька	Низька	Висока
Користувацький досвід (UX)	Безшовний	Фрагментований	Безшовний

На підставі аналізу для подальшого дослідження обрано підхід Run-time Integration як такий, що поєднує незалежність розгортання з якісним користувацьким досвідом.

2. Проблеми масштабування та інструментарій

Впровадження мікрофронтенд архітектури, попри вирішення проблем моноліту, породжує новий клас інженерних викликів.

Проблема «Dependency Hell». У розподіленій системі кожен мікрофронтенд може мати власний набір залежностей. Якщо різні команди використовують різні версії React, завантаження обох модулів на одній сторінці призведе до завантаження двох версій фреймворку, що спричинить збільшення часу завантаження, конфлікти у глобальному просторі імен та непередбачувану поведінку.

Забезпечення консистентності інтерфейсу. Коли над продуктом працюють різні команди, існує ризик розбіжності у дизайні. Вирішенням є виділена бібліотека спільних компонентів (Shared UI Library).

Оркестрація та обмін даними. Мікрофронтенди мають бути автономними, але не ізольованими: їм необхідно обмінюватися такими даними, як токен автентифікації або налаштування інтерфейсу. Традиційні методи (LocalStorage, Custom Events) часто є недостатньо типізованими. Для вирішення необхідна архітектура, яка б дозволяла передавати контекст між незалежними додатками зі збереженням суворої типізації [6, 7].

Вирішення перелічених проблем вимагає спеціалізованих інструментів. Для

управління кодовою базою у вигляді монорепозиторію обрано інструмент **Nx** [6]. На відміну від попередніх рішень (Lerna), Nx надає: кешування результатів попередніх збірок (Computation Caching), команду `nx affected` для запуску тестування лише змінених модулів та автоматичну побудову графу залежностей.

Як інструмент збірки обрано *Rspack* – бандлер нового покоління, написаний мовою Rust. Rspack майже повністю сумісний з конфігурацією Webpack та підтримує Module Federation [9]. Завдяки архітектурі на основі Rust, Rspack забезпечує у 5–10 разів вищу швидкість збірки порівняно з Webpack, що безпосередньо впливає на Developer Experience та швидкість доставки змін.

3. Методологія проектування системи

3.1. Архітектурний патерн Shell-Remote

Для побудови системи обрано патерн Shell-Remote, реалізований за допомогою Module Federation. Він передбачає чіткий поділ відповідальності між головним застосунком-контейнером (Shell) та незалежними функціональними модулями (Remotes).

Shell виступає як точка входу та оркестратор системи. Він не містить складної бізнес-логіки, а відповідає за: глобальний роутинг, відображення спільних елементів інтерфейсу (Header, Sidebar), а також ініціалізацію сесії та завантаження глобального контексту користувача.

Систему декомпозовано на два автономні мікрофронтенди відповідно до предметних областей:

– Auth Remote – відповідає за ідентифікацію, реєстрацію та відновлення доступу;
– Tasks Remote – містить основну бізнес-логіку управління задачами (Kanban-дошка).

Такий поділ уможливлює застосування стратегії Lazy Loading: код модуля Tasks не завантажується у браузер доти, доки користувач не пройде авторизацію (рис. 1).

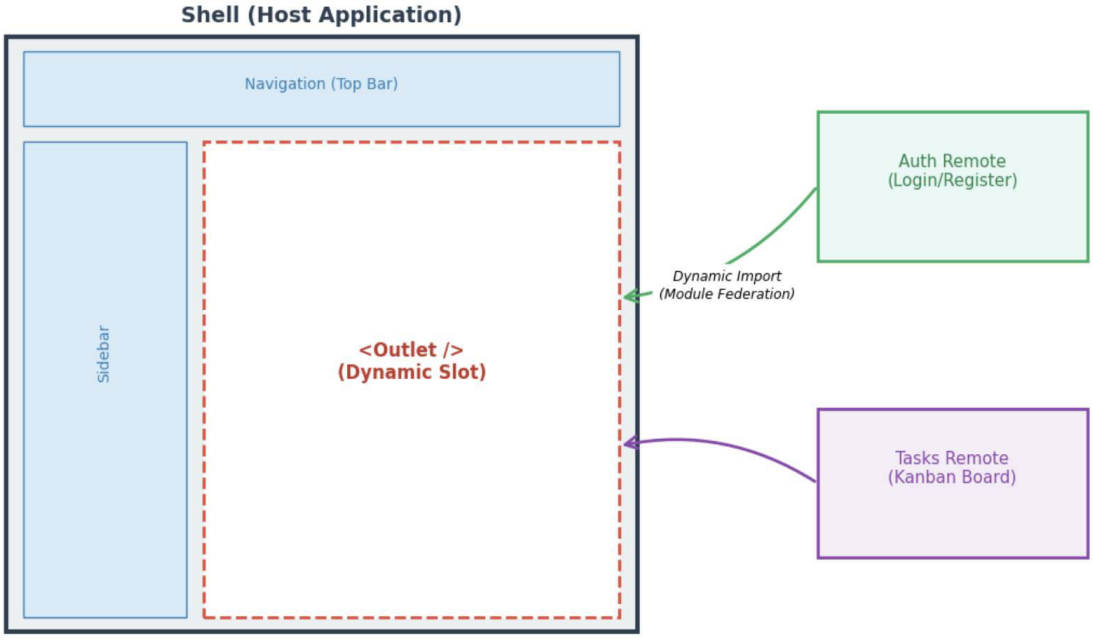

Рис. 1. Архітектура взаємодії Shell та Remote модулів

Це критично зменшує розмір початкового пакету даних (Initial Bundle Size).

3.2. Організація монорепозиторію Nx

Структуру проекту спроектовано за методологією Enterprise Monorepo Pattern, яка передбачає розподіл коду на дві категорії: Apps (застосунки) та Libs (бібліотеки).

Впроваджено суворий поділ бібліотек за типами (рис. 2):

– Feature Libraries (`libs/feature-*`) – містять «розумні» компоненти (Smart Components), підключені до сховища даних та реалізують конкретні бізнес-сценарії. Мікрофронтенди (Apps) є тонкими обгортками, що імпортують ці бібліотеки;
– Shared UI Libraries (`libs/shared-ui`) – набір «презентаційних» компонентів (кнопки, модальні вікна, поля вводу) без залежностей від бізнес-логіки. Спільне використання гарантує візуальну цілісність системи;
– Data Access Libraries (`libs/shared-data-access`) – шар доступу до даних, що містить API-клієнти та типи TypeScript (Zod-схеми). Винесення цього коду в окрему бібліотеку дозволяє різним мікрофронтендам використовувати одні й ті самі методи API без дублювання;
– Utility Libraries (`libs/util-*`) – чисті функції (Pure Functions) для форматування дат, валідації рядків тощо.

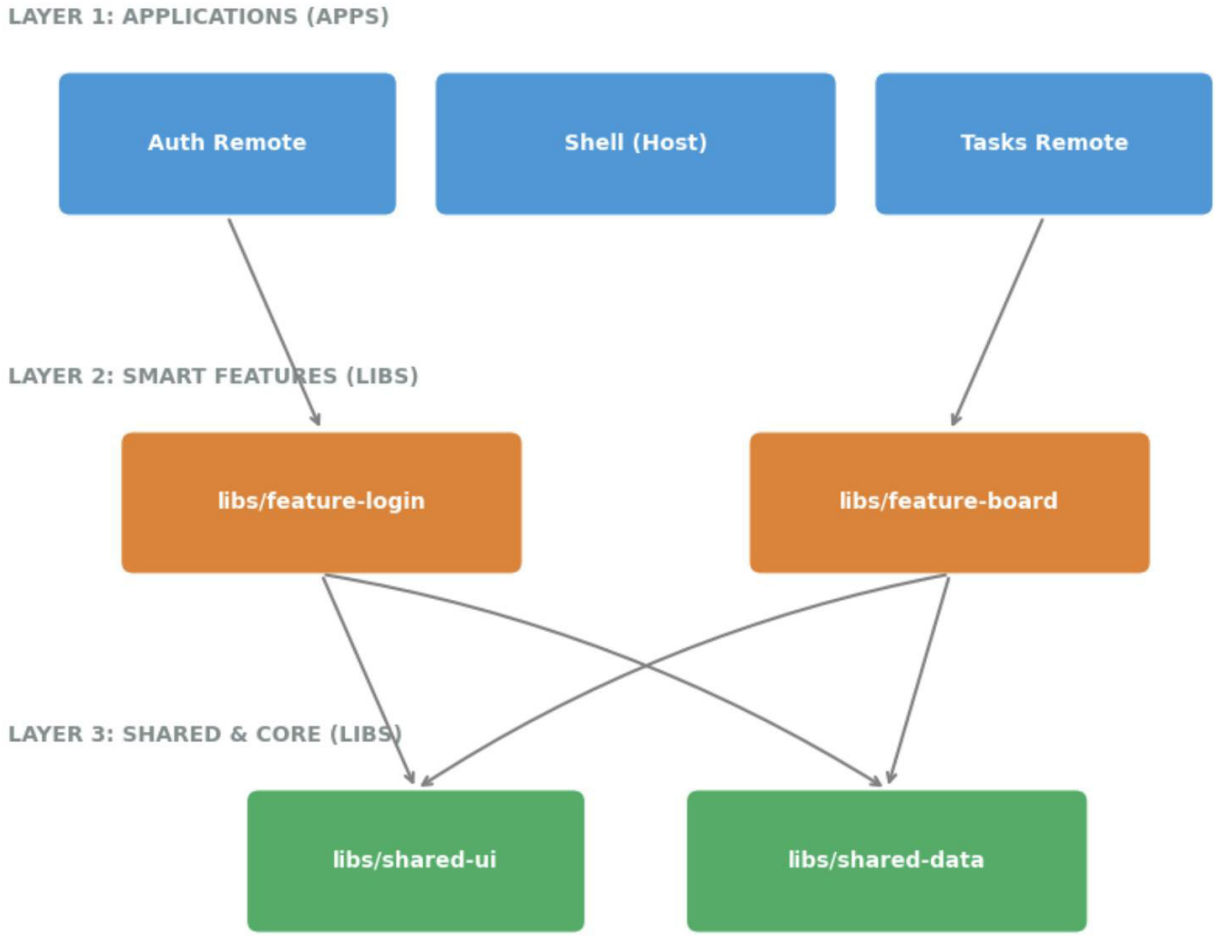

Рис. 2. Логічна структура бібліотек у монорепозиторії

Така організація реалізує принцип «Write Once, Use Everywhere». Інструмент Nx автоматично аналізує імпорти між бібліотеками та при зміні у shared-ui перезбирає лише ті додатки, що її використовують.

3.3. Керування станом та типізація

Однією з найскладніших проблем мікрофронтендів є обмін даними між незалежними додатками. Замість громіздких бібліотек управління станом (Redux, MobX), складних у налаштуванні для Module Federation, обрано нативний React Context API.

Реалізовано патерн «Provider in Shell, Consumer in Remote»:

- AuthContext створюється у спільній бібліотеці shared-data-access;
- AuthProvider огортає весь застосунок на рівні Shell;
- Будь-який мікрофронтенд (Auth або Tasks) отримує доступ до даних користувача через хук useAuth().

Критичним технічним нюансом є налаштування параметра `singleton: true` для бібліотеки React у конфігурації Rspack. Якщо кожен застосунок завантажить власну версію React, контекст не функціонуватиме (Provider і Consumer «не побачать» один одного).

Для забезпечення надійності даних застосовано бібліотеку Zod [7]. Вона дозволяє описувати схеми даних, що використовуються як для валідації форм, так і для генерації TypeScript-інтерфейсів. Це вирішує проблему розсинхронізації: якщо сервер змінить формат відповіді, Zod-схема одразу сигналізує про помилку, запобігаючи «тихому» збою інтерфейсу.

3.4. Конфігурація збірки та Bootstrap Pattern

Для коректної ініціалізації Module Federation точку входу застосунку розподілено на два файли: `index.ts` та `bootstrap.tsx`. Файл `index.ts` містить лише динамічний імпорт (`import('./bootstrap')`). Це надає браузеру час завантажити спільні бібліотеки (Shared Scope) перед тим, як почне виконуватися основний код React. Без цього кроку застосунок завершував би роботу з помилкою відсутності залежностей.

Інтеграція віддалених модулів реалізована через функцію `React.lazy`. Це забезпечує завантаження JavaScript-пакету мікрофронтенду лише у момент переходу користувача на відповідний маршрут (рис. 3). Для покращення UX використовується компонент `<Suspense>` – поки віддалений модуль завантажується мережею, користувач бачить глобальний індикатор завантаження (скелетон інтерфейсу).

Для ізоляції помилок мережі реалізовано патерн Error Boundary. Кожен мікрофронтенд огорнуто у компонент-обгортку: якщо завантаження модуля Tasks зазнає невдачі, Shell не «падає» повністю, а відображає повідомлення про тимчасову недоступність сервісу.

4. Експериментальне дослідження ефективності

4.1. Методика та умови проведення

Метою експерименту є кількісна оцінка ефективності запропонованої архітектури порівняно з традиційним монолітним підходом (SPA на Webpack). Дослідження проводилося на обладнанні з процесором Intel Core i5 та 16 GB оперативної пам'яті під управлінням macOS. Для вимірювань застосовувалися: вбудовані засоби профілювання часу збірки Rspack/Webpack, Google Lighthouse для аналізу метрик Core Web Vitals, Webpack Bundle Analyzer для аналізу структури вихідних файлів.

4.2. Продуктивність збірки

Порівнювалися дві конфігурації:

1. Базова: Webpack 5 + Babel (стандартний стек).
2. Запропонована: Rspack + SWC (реалізований стек).

Результати наведено у таблиці 2.

Застосування Rspack дозволило скоротити час збірки більш ніж у 10 разів (рис. 4). Для практичного значення: при команді з 10 розробників, які виконують по 20 збірок на день, економія становить десятки людино-годин щомісяця.

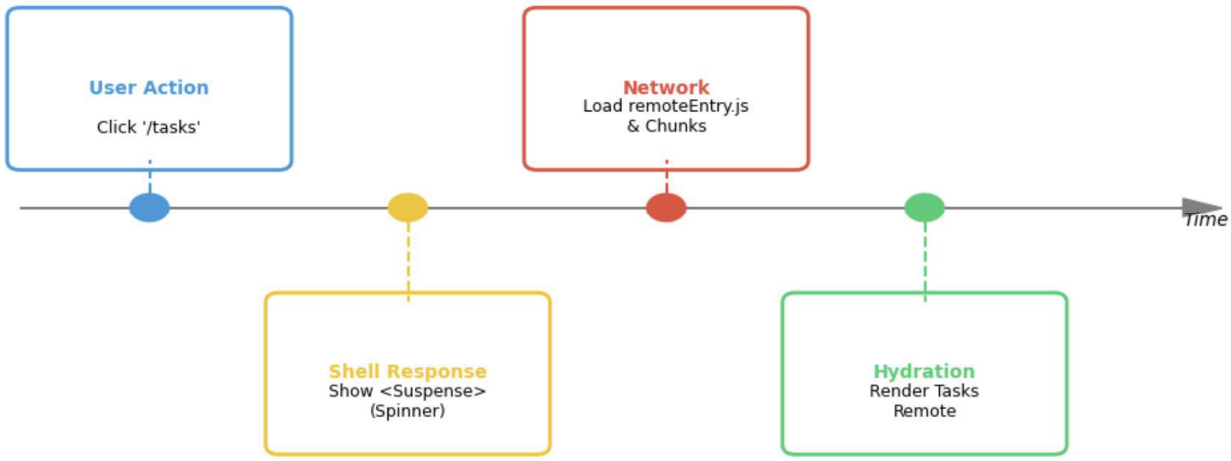

Рис. 3. Послідовність завантаження віддаленого модуля (Lazy Loading Flow)

4.3. Метрики завантаження (Core Web Vitals)

Для кінцевого користувача ключовою є швидкість відкриття сторінки. Аудит Shell-застосунку засобами Google Lighthouse показав наступні результати:

- LCP (Largest Contentful Paint): 0.8 сек (зона «Good»). Завдяки тому, що Shell завантажує лише мінімальний JavaScript-код (навігація + макет), основний контент з'являється миттєво;
- CLS (Cumulative Layout Shift): 0.002. Використання фіксованих розмірів для блоків-скелетонів запобігає «стрибанню» верстки при довантаженні мікрофронтендів;
- FID (First Input Delay): 12 мс. Розбиття коду на чанки (Code Splitting) розвантажує головний потік браузера.

Таблиця 2

Результати вимірювання швидкості збірки

Тип операції	Webpack (сек)	Rspack (сек)	Приріст продуктивності
Холодний старт (Dev Server)	45.2	3.8	~11.8×
Production Build (повна збірка)	62.5	5.1	~12.2×
HMR (зміна UI-компонента)	1.5	0.15	~10×

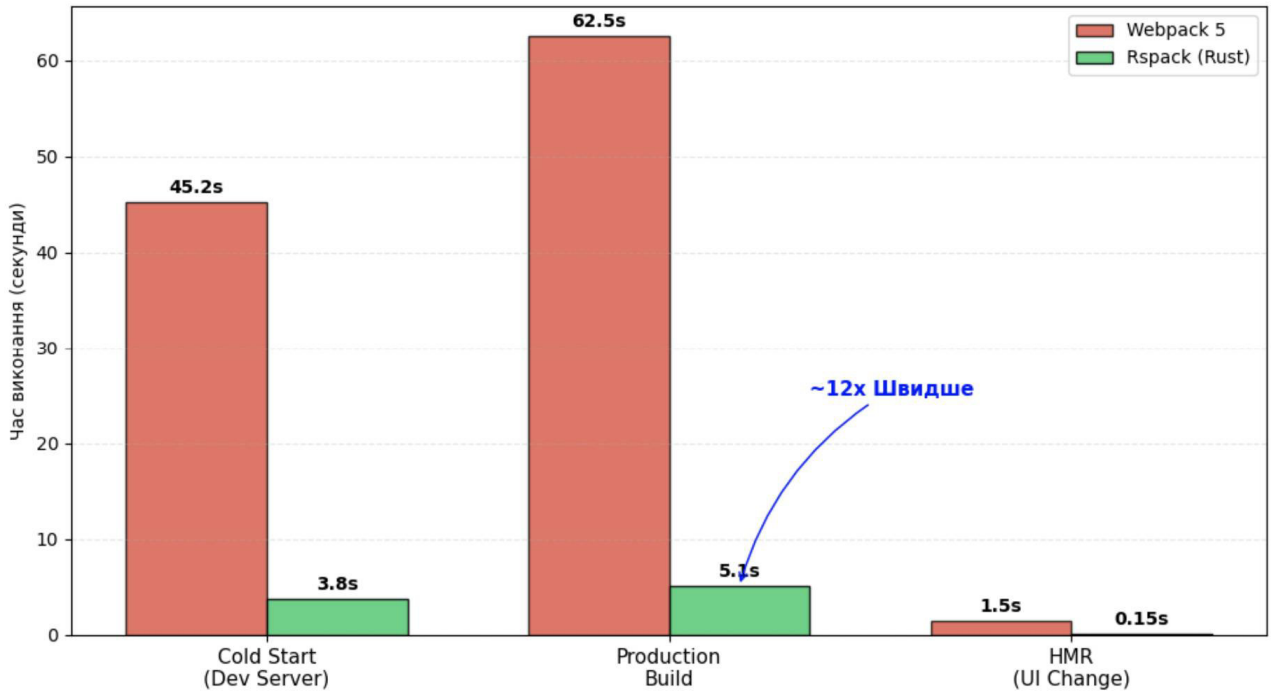

Рис. 4. Порівняльна діаграма часу збірки (Build Time)

Всі три показники відповідають рекомендованим порогам Google (LCP < 2.5 с, CLS < 0.1, FID < 100 мс).

4.4. Ефективність ледачого завантаження

Проаналізовано обсяг даних при першому вході на сторінку (Initial Load):

- монолітний підхід: 2400 KB – завантажується весь код застосунку, включно зі сторінками, які користувач може ніколи не відвідати.
- мікрофронтенд підхід: 450 KB (Shell + Shared Libs). Модуль Tasks (800 KB) завантажується лише за вимогою.

Результат: зменшення початкового розміру пакету на 81% (рис. 5).

4.5. Дедублікація залежностей та CI/CD

За допомогою Bundle Analyzer перевірено роботу механізму shared у ModuleFederationPlugin. При переході на модуль Auth браузер не завантажує React повторно, а використовує вже закешовану версію з пам'яті. Завантажується лише унікальний код модуля (~45 KB), що підтверджує коректність налаштування singleton: true.

Для оцінки масштабованості процесів CI/CD розглянуто сценарій:

- внесено зміни лише у бібліотеку libs/feature-login;
- стандартний підхід: запуск тестів для Shell, Auth, Tasks, API – 4 хвилини;
- Nx Affected: запуск тестів лише для модуля Auth – 45 секунд.

Це демонструє нелінійне зростання часу CI/CD: при збільшенні кількості модулів час перевірки змін залишається константним і залежить лише від локальних зв'язків зміненого компонента.

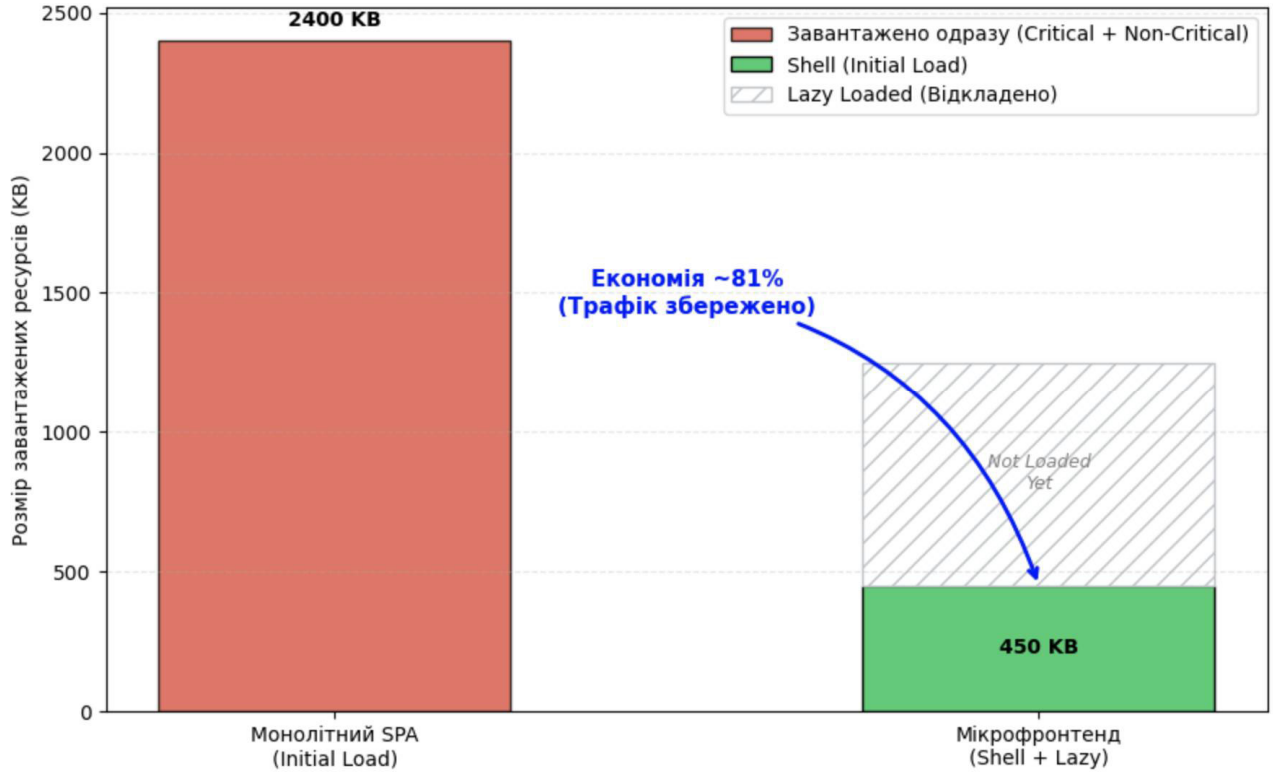

Рис. 5. Вплив Lazy Loading на розмір початкового завантаження

5. Аналіз результатів та наукова новизна

Отримані результати підтверджують ефективність запропонованої архітектури за всіма досліджуваними показниками. Наукова новизна роботи полягає у наступному.

По-перше, розроблено комплексну методику проектування розподілених веб-систем, яка поєднує патерн Shell-Remote з організацією монорепозиторію за рівнями відповідальності. На відміну від розрізнених підходів, описаних у [2, 3], запропонована методика включає не лише архітектурне рішення, але й конкретні патерни організації коду та управління залежностями.

По-друге, вирішено проблему «Shared State» у федеративному середовищі без застосування сторонніх бібліотек стану. Патерн «Provider in Shell, Consumer in Remote» у поєднанні з параметром `singleton: true` у конфігурації Rspack/Webpack забезпечує коректну роботу React Context між незалежно розгорнутими модулями – технічний аспект, що часто залишається поза увагою у відповідній літературі.

По-третє, отримано верифіковані кількісні метрики переходу від монолітної архітектури до мікрофронтендів у середовищі Rspack, яких бракує у наявних публікаціях. Зокрема, встановлено, що приріст продуктивності збірки перевищує декларовані у документації Rspack показники [9] та сягає 12.2× для повної виробничої збірки.

Слід зазначити й обмеження дослідженого підходу. Висока складність початкового налаштування (конфігурація Rspack, налаштування Module Federation, організація монорепозиторію) потребує значних компетенцій від команди. Для невеликих проектів з одним-двома командами розробників додаткові витрати на налаштування інфраструктури можуть не виправдати себе. Крім того, стратегія Lazy Loading потребує ретельного планування Error Boundaries, оскільки недоступність окремого Remote-модуля не повинна порушувати роботу всього застосунку.

Висновки

У роботі досліджено методи побудови масштабованих веб-систем на основі мікрофронтенд архітектури. Проведено порівняльний аналіз підходів до інтеграції мікрофронтендів, за результатами якого обґрунтовано перевагу Run-time Integration на базі Module Federation над альтернативними методами (Build-time, iframe).

Розроблено та реалізовано систему управління задачами на основі патерну Shell-Remote у середовищі монорепозиторію Nx з бандлером Rspack. Запропоновано структуру монорепозиторію з чітким поділом на рівні відповідальності, що забезпечує повторне використання спільних компонентів та типів даних.

Вирішено проблему передачі спільного стану між незалежними мікрофронтендами шляхом застосування нативного React Context API у поєднанні з налаштуванням Singleton для ядра фреймворку.

Проведено експериментальне дослідження, яке підтвердило ефективність запропонованого підходу:

- перехід на Rspack забезпечив приріст швидкості збірки у 12 разів порівняно з Webpack;
- застосування Lazy Loading зменшило розмір початкового завантаження сторінки на 81%;
- досягнуто відповідність метрикам Core Web Vitals (LCP < 1 секунди);
- використання nx affected скоротило час виконання тестів у CI/CD у 5–6 разів для локальних змін.

Отримані результати свідчать про доцільність застосування дослідженого підходу для побудови корпоративних (Enterprise-рівня) веб-систем з великими командами розробників. Перспективним напрямком подальших досліджень є вивчення стратегій тестування у федеративному середовищі та розробка методик міграції існуючих монолітних застосунків на мікрофронтенд архітектуру.

Список використаної літератури

1. Jackson C. Micro Frontends / C. Jackson // Martin Fowler Blog. – 2019. – [Електронний ресурс]. – Режим доступу: https://martinfowler.com/articles/micro-frontends.html
2. Mezzalira L. Building Micro-Frontends: Scaling Teams and Projects / L. Mezzalira. – O'Reilly Media, 2021. – 300 с.
3. Geers M. Micro Frontends in Action / M. Geers. – Manning Publications, 2020. – 256 с.
4. Webpack Module Federation. Official Documentation. – [Електронний ресурс]. – Режим доступу: https://webpack.js.org/concepts/module-federation/
5. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. – O'Reilly Media, 2021. – 600 с.
6. Nx Documentation. Monorepo for modern development. – [Електронний ресурс]. – Режим доступу: https://nx.dev/
7. Zod Documentation. TypeScript-first schema declaration and validation. – [Електронний ресурс]. – Режим доступу: https://zod.dev/
8. Nrwl. Nx: Smart Monorepos, Fast CI. – [Електронний ресурс]. – Режим доступу: https://nx.dev/concepts/mental-model
9. Rspack Team. Rspack: The Rust-based web bundler. – [Електронний ресурс]. – Режим доступу: https://www.rspack.dev/

References

1. Jackson C. (2019) Micro Frontends. Martin Fowler Blog. Available at: https://martinfowler.com/articles/micro-frontends.html
2. Mezzalira L. (2021) Building Micro-Frontends: Scaling Teams and Projects. O'Reilly Media, 300 p.
3. Geers M. (2020) Micro Frontends in Action. Manning Publications, 256 p.
4. Webpack Module Federation. Official Documentation. Available at: https://webpack.js.org/concepts/module-federation/
5. Newman S. (2021) Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 600 p.

6. Nx Documentation. Monorepo for modern development. Available at: https://nx.dev/
7. Zod Documentation. TypeScript-first schema declaration and validation. Available at: https://zod.dev/
8. Nrwl. Nx: Smart Monorepos, Fast CI. Available at: https://nx.dev/concepts/mental-model
9. Rspack Team. Rspack: The Rust-based web bundler. Available at: https://www.rspack.dev/

TKACHENKO Oleksii,
Student, Department of Applied Mathematics and Informatics, The Bohdan Khmelnytsky National University of Cherkasy, Ukraine
DIDKOWSKY Ruslan,
Doctor of Technical Sciences, Associate Professor, Department of Informatics and Applied Mathematics, The Bohdan Khmelnytsky National University of Cherkasy, Ukraine

RESEARCH INTO MICROFRONTEND ARCHITECTURE FOR BUILDING SCALABLE WEB SYSTEMS

Summary. Introduction. *Modern web applications have evolved from simple informational pages into complex enterprise systems. With the growth in codebase and the number of development teams, traditional monolithic frontend architectures exhaust their potential, creating barriers to further product development. The concept of Micro Frontends emerged as an adaptation of microservice architecture principles to the client side of web applications.*

Purpose. *The aim of this article is to substantiate a methodology for designing and implementing scalable web systems based on microfrontend architecture using Module Federation technology, and to compare its effectiveness with traditional approaches.*

Results. *A comparative analysis of microfrontend integration methods was conducted: Build-time Integration, iframe-based isolation, and Run-time Integration. The Shell-Remote architectural pattern was selected and implemented using Module Federation technology within the Nx monorepo environment with the Rspack bundler. A monorepo structure was developed with a clear separation into responsibility layers (Applications, Feature Libraries, Shared UI, Data Access). A solution to the shared state problem was implemented using native React Context API combined with the Singleton configuration for the React framework instance. An experimental study was conducted comparing build performance and loading metrics between the proposed architecture and traditional Webpack-based monolith.*

Conclusion. *The experimental study confirmed the high effectiveness of the proposed architecture: the transition to Rspack provided a 12-fold build speed increase compared to Webpack; applying the Lazy Loading strategy reduced the initial page bundle size by 81%; Core Web Vitals metrics were achieved (LCP < 1 second); the use of Nx tooling reduced CI/CD testing time by 5–6 times for local changes. The obtained results confirm the practical value of the investigated approach for building enterprise-level web systems.*

Keywords: *microfrontend architecture, Module Federation, monorepo, Nx, Rspack, Shell-Remote, scalability, build performance.*

Одержано редакцією 10.11.2025 р.
Прийнято до публікації 17.12.2025 р.

УДК 004.415.2:004.738.5

DOI 10.31651/2076-5886-2025-1-95-106

PACS 07.05.Tp, 89.20.Ff

ПЕДЧЕНКО Максим Анатолійович
студент спеціальності «Прикладна математика» Черкаського національного університету імені Богдана Хмельницького
e-mail: pedchenko.maksym@vu.cdu.edu.ua

СЕРДЮК Олександр Анатолійович
кандидат економічних наук, доцент, доцент кафедри прикладної математики та інформатики Черкаського національного