НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут телекомунікаційних систем

Кафедра інформаційних технологій в телекомунікаціях

До захисту допущено:
ВО завідувача кафедри
_____ Валерій
ПРАВИЛО
«_____»_____2026 р.

ДИПЛОМНА РОБОТА
на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інформаційно-комунікаційні технології»
спеціальності 172 Телекомунікації та радіотехніка

на тему: Метод оптимізації структури web-сторінки для зменшення

мережевого трафіку

Виконала: здобувачка вищої освіти 4 курсу, групи ТІ-21
Володіна Альона Олексіївна _____

Науковий керівник: доцент кафедри ІТТ НН ІТС, кандидат технічних наук,
доцент Суліма Світлана Валеріївна _____

Рецензент: доцент кафедри ТК НН ІТС, кандидат технічних наук,

доцент Міночкін Д.А. _____

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.
Здобувач вищої освіти _____

Київ – 2026 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут телекомунікаційних систем

Кафедра інформаційних технологій в телекомунікаціях

Рівень вищої освіти – перший (бакалаврський)
Освітньо-професійна програма «Інформаційно-комунікаційні технології» спеціальності 172 Телекомунікації та радіотехніка

ЗАТВЕРДЖУЮ
ВО завідувача кафедри
_____ Валерій ПРАВИЛО
«_____»_____2026 р.

ЗАВДАННЯ
на дипломну роботу студентці
Володіній Альоні Олексіївні

1. Тема дипломної роботи: Метод оптимізації структури web-сторінки для зменшення мережевого трафіку, науковий керівник роботи доцент кафедри ІТТ НН ІТС, кандидат технічних наук, доцент Суліма Світлана Валеріївна - затверджені наказом по університету від 26.05.2026 р. №1912-с.
2. Строк подання студентом дипломної роботи 06.06.2026 р.
3. Об’єкт дослідження: Процес завантаження веб-сторінки та передачі мережевих ресурсів між веб-сервером і браузером клієнта.
4. Вихідні дані до роботи: Статистичні дані щодо розподілу мережевого трафіку вебсторінок за типами ресурсів, технічні характеристики браузерних API (Intersection Observer, Service Worker, Cache Storage), специфікації форматів растрових зображень та алгоритмів їх стиснення, результати вимірювань продуктивності вебресурсів засобами Chrome DevTools та Lighthouse.

5. Перелік завдань, які потрібно розробити:

1. Проаналізувати структуру мережевого трафіку сучасних вебсторінок та визначити основні фактори, що спричиняють надлишкову передачу даних між клієнтом та сервером.
2. Дослідити існуючі методи зниження обсягу мережевого трафіку та обґрунтувати вибір оптимального підходу до оптимізації структури вебсторінки.
3. Розробити бібліотеку OptimizationManager та модуль Service Worker, що реалізують динамічну оптимізацію завантаження ресурсів вебсторінки.
4. Провести порівняльне експериментальне дослідження ефективності розробленого методу та виміряти зміни ключових метрик продуктивності.

6. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо): презентація -18 слайдів, 10 таблиць, 4 рисунків.

7. Дата видачі завдання: 01.09.2025 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Узгодження організаційних питань з науковим керівником	10.09.24-20.09.24	Виконано
2	Узгодження теми роботи та початок роботи над першим розділом	04.02.26-04.03.26	Виконано
3	Робота над першим, другим та	04.03.26-10.05.26	Виконано

	третім розділами		
4	Робота над четвертим розділом. Планування розробки власного рішення	10.05.26-17.05.26	Виконано
5	Розробка власного рішення та його перевірка	15.03.26 - 20.05.26	Виконано
6	Висновки про ефективність рішення	21.05.26 - 31.05.26	Виконано
7	Оформлення дипломної роботи	06.06.26 - 12.06.26	Виконано
8	Підготовка презентації до захисту	01.06.26 - 12.06.26	Виконано
9	Захист дипломної роботи	16.06.26 - 19.06.26	Виконано

Здобувачка вищої освіти _____ Альона ВОЛОДІНА

Науковий керівник роботи _____ Світлана СУЛІМА

РЕФЕРАТ

Дипломна робота «Метод оптимізації структури web-сторінки для зменшення мережевого трафіку» містить 68 сторінок, 10 рисунків, 4 таблиці та 2 додатки, а також використано 20 літературних джерел.

Актуальність теми: Сучасні веб-сайти сильно перевантажені важким мультимедійним контентом, особливо зображеннями високої роздільної здатності. Це створює величезне навантаження на канали зв'язку та мобільні мережі, через що сторінки завантажуються занадто довго. Традиційні методи стиснення на сервері часто важко налаштувати або вимагають переписування коду всього сайту. Тому виникла потреба створити просте та гнучке рішення на стороні клієнта, яке б оптимізувало трафік «на льоту» та пришвидшувало роботу сайтів у реальних умовах (наприклад, коли студенти на смартфонах мають поганий інтернет у навчальних корпусах).

Метою роботи є зменшення мережевого трафіку та пришвидшення відображення сторінок у браузері користувача шляхом створення універсальної клієнтської бібліотеки для динамічного управління структурою та форматами ресурсів.

Об'єктом дослідження є процес завантаження веб-ресурсів та передачі мережевих ресурсів між веб-сервером і браузером клієнта.

Предметом дослідження є метод оптимізації структури web-сторінки для зменшення обсягу мережевого трафіку та часу завантаження веб-сторінки шляхом оптимізації структури ресурсів.

Методи дослідження – аналіз мережевих протоколів, проектування об'єктів та експериментальне моделювання. Під час роботи досліджувалися математичні моделі сучасних кодеків, розроблені алгоритми перехоплення DOM-дерева, а працездатність методу перевірялася шляхом тестування та вимірювань на основі локального сервера Flask.

Отримані результати. Розроблено універсальну JavaScript-бібліотеку OptimizationManager та фоновий модуль Service Worker. Систему

протестовано на двох тестових стендах з принципово різною структурою контенту: фотогалерея тревел-блогу "Wanderlust" (насичена графічним контентом) та інформаційний портал навчального кампусу університету. На фотогалереї динамічна конвертація зображень у формат WebP забезпечила скорочення обсягу переданих даних на 25% — з 2.0 до 1.5 МБ — як при стандартному з'єднанні, так і при емуляції мобільного 3G. Кількість HTTP-запитів скоротилась з 19 до 18. На кампус-порталі, де растрові зображення відсутні, механізм Service Worker кешування покращив метрику LCP на 20% — з 0.5 до 0.4 секунди.

Ключові слова: ОПТИМІЗАЦІЯ ВЕБСТОРІНОК, МЕРЕЖЕВИЙ ТРАФІК, ЛІНИВЕ ЗАВАНТАЖЕННЯ, ФОРМАТ WEBP, SERVICE WORKER, INTERSECTION OBSERVER, КЕШУВАННЯ, ФОТОГАЛЕРЕЯ, НАВЧАЛЬНИЙ КАМПУС.

ABSTRACT

The thesis " Web page structure optimization method for network traffic reduction " contains 68 pages, 10 figures, 4 tables and 2 appendices, and references 20 literary sources.

Relevance of the topic: Modern websites are heavily overloaded with heavy multimedia content, especially high-resolution images. This creates a significant load on communication channels and mobile networks, causing pages to take too long to load. Traditional server-side compression methods are often difficult to configure or require rewriting the entire site's codebase. Therefore, there was a need to create a simple and flexible client-side solution that would optimize traffic on the fly and speed up websites under real-world conditions (for example, when students on smartphones have poor internet reception inside university buildings).

The goal of the work is to reduce network traffic and speed up the display of pages in the user's browser by creating a universal client-side library for dynamic management of resource structure and formats.

The object of the research is the process of loading web resources and transmitting network data between the web server and the client's browser.

The subject of the research is the reduction of network traffic volume and web page load time through optimization of resource structure.The research methods are analytical analysis of network protocols, object design and experimental modeling. During the work, mathematical models of modern codecs were studied, algorithms for intercepting the DOM tree were developed, and the performance of the method was verified through testing and measurements based on a local Flask server.

Results. A universal JavaScript library OptimizationManager and a background Service Worker module were developed. The system was tested on

two test stands with fundamentally different content structures: a travel blog photo gallery "Wanderlust" (rich in graphic content) and an informational campus portal of the university. On the photo gallery, dynamic conversion of images to WebP format reduced the volume of transferred data by 25% — from 2.0 to 1.5 MB — both under standard connection and under mobile 3G emulation. The number of HTTP requests decreased from 19 to 18. On the campus portal, which contains no raster images, the Service Worker caching mechanism improved the LCP metric by 20% — from 0.5 to 0.4 seconds.

Keywords: WEB PAGE OPTIMIZATION, NETWORK TRAFFIC, LAZY LOADING, WEBP FORMAT, SERVICE WORKER, INTERSECTION OBSERVER, CACHING, PHOTO GALLERY, EDUCATIONAL CAMPUS.

ЗМІСТ

СПИСОК ТЕРМІНІВ І СКОРОЧЕНЬ

API	Application Programming Interface (Інтерфейс прикладного програмування)
CSS	Cascading Style Sheets (Каскадні таблиці стилів)
CDN	Content Delivery Network (мережа доставки контенту)
DOM	Document Object Model (Об’єктна модель документа)
DevTools	Developer Tools (інструменти розробника браузера)
DNS	Domain Name System (система доменних імен)
HTML	HyperText Markup Language (Мова гіпертекстової розмітки)
HTTP	HyperText Transfer Protocol (Протокол передавання гіпертексту)
JS	JavaScript (Мова програмування JavaScript)
JPEG	Joint Photographic Experts Group (формат стиснення растрових зображень)
LCP	Largest Contentful Paint (Час рендерингу найбільшого видимого елемента)
PNG	Portable Network Graphics (Портативна мережева графіка)
RTT	Round-Trip Time (Час кругового обходу пакета)
SSL/TLS	Secure Sockets Layer / Transport Layer Security (протоколи захищеної передачі даних)
TCP	Transmission Control Protocol (Протокол керування передаванням даних)
URL	Uniform Resource Locator (Уніфікований локатор доступу)
WebP	Web Picture (Формат стиснення зображень WebP)

ВСТУП

Сьогодні практично кожен популярний вебресурс перевантажений великою кількістю важкого мультимедійного контенту. Якщо раніше сайти складалися переважно з тексту, то зараз значну частину їхньої ваги в мережевому потоці становлять графічні зображення та шрифти. Це призводить до серйозної технічної проблеми: коли на сайт одночасно заходить багато людей, телекомунікаційні мережі та сервери починають працювати на межі своїх можливостей. Черги запитів переповнюються, а швидкість завантаження сторінок сильно падає. Особливо гостро це відчувають користувачі мобільних телефонів, коли намагаються відкрити потрібну сторінку в умовах слабкого сигналу, наприклад, усередині великих корпусів університету чи в дорозі.

Традиційні способи вирішення цієї проблеми зазвичай зводяться до того, що розробники намагаються вручну стискати картинки або змінювати налаштування на стороні сервера. Проте, якщо система вже створена і активно працює, будь-яке втручання в її бекенд вимагає багато часу та несе ризик зламати робочий функціонал. Саме тому виникла ідея перенести логіку управління ресурсами на сторону клієнта, тобто безпосередньо в браузер користувача. Це дозволило б оптимізувати мережевий трафік у реальному часі без необхідності змінювати щось на сервері.

Метою цієї дипломної роботи є суттєве зменшення обсягів мережевого трафіку та прискорення рендерингу сторінок у браузері користувача завдяки створенню універсального клієнтського методу, який автоматично керує структурою сторінки та форматами її файлів. Для реалізації цієї мети було поставлено та вирішено низку практичних інженерних завдань. Спочатку треба детально розібралися зі структурою сучасного трафіку та визначити основні фактори перевантаження каналів зв'язку. Після цього було досліджено сучасні кодеки стиснення та обґрунтовано перехід на

ефективніший формат WebP. На основі цих даних було створено алгоритм, який перехоплює елементи сторінки під час її побудови, а також спроєктовано систему локального кешування, що працює в окремому фоновому потоці.

Об'єктом дослідження є сам процес завантаження вебсторінки та передачі її ресурсів по мережі від сервера до користувача. Предметом дослідження стали алгоритми та програмні інструменти для зменшення обсягу мережевого трафіку та часу завантаження вебсторінки шляхом оптимізації структури ресурсів. У роботі виконувався аналітичний аналіз мережевих протоколів та експериментальне моделювання з фіксацією результатів через інструменти розробника Browser DevTools.

Наукова новизна розробленого підходу полягає в тому, що механізми відкладеного завантаження та перехоплення запитів працюють комплексно всередині браузера. Це дозволяє адаптувати вагу контенту під якість зв'язку конкретного користувача прямо під час скролінгу, взагалі не чіпаючи серверний код.

Практичне значення роботи полягає в написанні готової JavaScript-бібліотеки OptimizationManager, яку можна підключити до сайту, незалежно від технологій бекенду. Створене програмне рішення дозволяє оптимізувати відображення медіафайлів на сторінці та знизити навантаження на серверну інфраструктуру телекомунікаційних мереж. Результати досліджень та запропоновані алгоритми можна впроваджувати для покращення швидкодії інформаційних порталів, освітніх платформ, блогів чи корпоративних сайтів.

РОЗДІЛ 1
АНАЛІЗ РОЛІ МЕРЕЖЕВОГО ТРАФІКУ В СУЧАСНИХ ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМАХ

Сучасний інтернет – це вже давно не просто текст та посилання. Щодня мільйони людей завантажують у соціальні мережі, на портали та персональні сайти терабайти фотографій у високій роздільній здатності. При цьому мережева інфраструктура відчуває все більше і більше навантаження. Тому класичні підходи до передачі даних уже не справляються з тим, що від них вимагають.

Проблема спостерігається одразу на декількох рівнях. Великі обсяги даних переповнюють буфери мережевого обладнання, що тягне за собою затримки і втрати пакетів. Браузер намагаючись завантажити десятки дрібних ресурсів, змушений відкривати окреме TCP-з’єднання для кожного з них, що призводить до суттєвого сповільнення появи контенту на екрані. Для мобільних мереж 4G та 5G ситуація ще більш складніша, бо нестабільний радіоканал і обмежена пропускна здатність роблять передачу важких файлів особливо чутливою.

Саме тому оптимізація структури вебсторінки є не просто питанням зручності чи естетики, це є технічною задачею, яка напряму впливає на те, скільки мегабайт реально передається через мережу і як швидко користувач побачить те, що шукає.

1.1 Класифікація мережевих ресурсів вебсторінки та їх вплив на пропускну здатність

Сучасна вебсторінка – ієрархічна система, де кожен компонент навантажує мережу. Щоб зрозуміти, де саме виникають зайві витрати трафіку, треба розібратись у тому, з чого складається ця система.

HTML-документ є структурним каркасом сторінки. Сам по собі він займає зовсім небагато місця, але відіграє ключову роль. Він визначає порядок, у якому браузер буде завантажувати всі інші ресурси. Погано структурований HTML може змусити браузер зробити зайві мережеві запити або затримати відображення сторінки.

CSS-файли містять стилі, без яких сторінка просто не відобразиться коректно. Тобто браузер розглядає їх як критичні ресурси і блокує рендеринг до повного завантаження. Якщо таблиці стилів містять зайві або дубльовані правила, це збільшує як обсяг переданих файлів, так і час очікування.

JavaScript – тип текстових даних, які є найбільш ресурсомісткими. Великі скрипти не лише збільшують трафік, а й значно навантажують процесор пристрою користувача, що особливо відчутно на бюджетних смартфонах. Якщо скрипт важкий, то він буде сповільнювати роботу сторінки незалежно від швидкості з’єднання.

Растрова графіка є лідером за споживанням трафіку на переважній більшості сайтів. Зображення у застарілих форматах JPEG та PNG, особливо в оригінальній роздільній здатності, можуть займати від кількох мегабайт до десятків мегабайт на одній сторінці. Саме графіка і стала першочерговим об’єктом оптимізації, і саме вона стала центральним елементом розробленого методу. [1, 2]

1.2 Суть та концепція методу динамічної оптимізації структури

Ідея методу проста: замість того щоб передавати всі ресурси сторінки одразу при її відкритті, система динамічно керує тим, що і коли завантажується. Це означає повну відмову від статичної моделі, при якій браузер хоче використовувати усі зображення підряд з серверу, навіть ті, які розташовані внизу сторінки і користувач їх ніколи не побачить.

Метод спирається на три взаємопов’язані механізми. Перший – мінімізація: на початковому етапі завантажуються лише ті ресурси, що потрібні для відображення поточного стану сторінки, а решта блокується. Другий – адаптивне завантаження: мультимедійний контент підтягується

поступово, у міру того як користувач прокручує сторінку вниз. Третій – кешування: при першому відвідуванні необхідні файли зберігаються локально на пристрої, і при повторному відкритті браузер уже не звертається до мережі.

Кожен з цих механізмів має сенс і окремо, але справжній ефект відчувається тільки тоді, коли всі три працюють разом. Тобто саме комплексне застосування дозволяє мінімізувати кількість байтів, що приходять через інтерфейс, зберігаючи при цьому якість відображеного контенту незмінною.

1.3 Методологія дослідження за допомогою інструментів розробника (Browser DevTools)

Будь-яке теоретичне твердження в інженерії потребує практичного підтвердження. Для цього дослідження основним інструментом вимірювань обрано Chrome/Edge Developer Tools – вбудований набір утиліт браузера, що відкривається клавішею F12. Цей інструмент дозволяє спостерігати за роботою вебсторінки на рівні мережевих протоколів і отримувати конкретні числові показники.

Панель Network відображає в реальному часі кожен HTTP-запит, який генерує сторінка. По кожному запиту видно тип ресурсу, його розмір, час виконання та заголовки відповіді сервера. Саме тут фіксується обсяг переданих даних.

Інструмент Waterfall відображає часову послідовність завантаження ресурсів у вигляді діаграми. На ній видно, де виникають затримки, тобто скільки часу пішло на DNS-розв'язання, скільки на встановлення TCP-з'єднання, скільки на отримання першого байта від сервера (TTFB) і скільки власне на передачу контенту.

Функція Throttling дозволяє штучно обмежити швидкість з’єднання і додати затримку передачі пакетів. Це незамінний інструмент для перевірки того, як сторінка поводиться в умовах мобільного інтернету зі слабким покриттям, саме там, де оптимізація трафіку є найбільш критичною.

Lighthouse – автоматизована система аудиту, інтегрована в DevTools. Вона перевіряє сторінку за методикою Google і формує інтегральний показник продуктивності від 0 до 100, що дозволяє об’єктивно порівнювати стан сторінки до та після застосування методу. [3, 4]

1.4 Вибір та обґрунтування метрик оцінки ефективності

Для того, щоб оцінка ефективності методу була об’єктивною, а не суб’єктивною, необхідно чітко визначити показники за якими будуть проводитись вимірювання. У цій роботі обрано три метрики, де кожна з яких відповідає за окремий аспект мережевої продуктивності.

LCP (Largest Contentful Paint) – час до моменту, коли на екрані повністю з’явився найбільш видимий елемент сторінки. Зазвичай це головне зображення чи великий заголовок. Ця метрика характеризує те, як швидко користувач бачить основний контент, і тісно пов’язана з обсягом даних, що передаються на початку завантаження. [3, 4]

Total Transferred Data – загальний обсяг байтів, що були передані через мережевий інтерфейс за час роботи зі сторінкою. Це є кількісним виміром того, скільки трафіку витратив користувач. Саме цей показник є центральним критерієм оцінки ефективності методу.

Request Count – сумарна кількість звернень браузера до сервера. Зменшення цього показника знижує навантаження на серверне обладнання і скорочує час, витрачений на встановлення з’єднань. У поєднанні з попередніми двома метриками він дає повну картинку мережевої поведінки сторінки.

1.5 Аналіз технологій стиснення мультимедійного контенту та обґрунтування переходу на сучасні формати

Головним джерелом надлишкового трафіку на сучасних вебресурсах є растрова графіка. Формати JPEG та PNG, що з’явились ще у 1990-х роках, розроблялись в абсолютно інших умовах, коли про мобільні мережі та щільне

завантаження конкретних платформ ніхто не думав. Їхня архітектура просто не розрахована на сьогоднішні реалії. [2]

Серед основних недоліків цих форматів варто виділити надлишкову вагу файлів і відсутність єдиного механізму одночасної підтримки прозорості та ефективного стиснення. JPEG добре справляється з фотографіями, але не підтримує прозорість. PNG вміє зберігати прозорість, але при цьому видає значно більші за розміром файли. У результаті розробники змушені використовувати обидва формати в залежності від контенту, що збільшує складність і загальну вагу сторінки.

Google розробила формат WebP, який принципово змінює підхід до стиснення. Якщо JPEG обробляє кожен блок зображення розміром 8*8 пікселів незалежно, то WebP аналізує сусідні блоки і прогнозує вміст поточного. У мережу передається не саме зображення, а лише математична різниця між прогнозом та реальними даними. Чим точніший прогноз, тим менше даних треба передати. [6]

За результатами статистичних досліджень, перехід на WebP дозволяє зменшити обсяг графічного трафіку на 25-34% порівняно з JPEG при однаковому значенні індексу структурної подібності (SSIM). Порівняно з PNG економія досягає 26%, при цьому WebP повноцінно підтримує прозорість. Таким чином, динамічна конвертація зображень у формат WebP у процесі роботи вебсторінки є технічно обґрунтованим і вимірюваним способом знизити загальний обсяг мережевого трафіку. [7, 6, 13]

1.6 Математична модель оцінки часу завантаження ресурсів вебсистеми

Для формалізації задачі дослідження та математичного обґрунтування ефективності методу, загальний час завантаження неоптимізованої вебсторінки (T_{total}) можна представити у вигляді функції від параметрів мережі до структури контенту: [11, 14]

$$T_{total} = \sum_{i=1}^{N}(RTT_i + T_{dns} + T_{tcp} + T_{ssl}) + \sum_{j=1}^{M}\frac{S_j}{B}$$

Де:

- N – загальна кількість HTTP-запитів до сервера (Request Count);
- RTT_i – час кругового обходу пакета в мережі для і-го запиту (Round-Trip Time);
- $T_{dns}, T_{tcp}, T_{ssl}$ – часові затримки на вирішення DNS-імені, встановлення TCP-сесії та SSL/TLS-рукостискання; [11, 14]
- M – кількість завантажувальних медіаресурсів на сторінці;
- S_j – розмір j-го зображення в байтах;
- B – поточна пропускна здатність каналу зв’язку користувача (Bandwidth).

Аналіз виведеної моделі показує, що оптимізація може здійснюватися трьома математичними шляхами:

- Зменшення чисельника другого доданку S_j: за допомогою класу OptimizationManager, стискаємо графіки у формат WebP;
- Зниження кількості одночасних запитів (N) на старті завантаження: завдяки впровадженню технологій адаптивного завантаження, ми відсікаємо запити до зображень, що знаходяться поза зоною видимості користувача (Intersection Observer API);
- Обнулення часу повторних запитів: за допомогою Service Worker, шляхом перехоплення запитів, що дозволяє працювати без затримки мережі для закешованих файлів.

Висновки

1. У першому розділі проведено комплексний аналіз проблематики мережевого трафіку в контексті сучасних вебплатформ. Встановлено, що зростання частки мультимедійного контенту є основним чинником перевантаження каналів зв’язку, що проявляється у переповненні буферів мережевого обладнання, збільшенні Round-Trip Time та погіршенні якості обслуговування мобільних користувачів.
2. Визначено три ключові метрики для оцінки ефективності методу: LCP, Total Transferred Data та Request Count. Обґрунтовано доцільність переходу на формат WebP як основний інструмент зменшення графічного трафіку.
3. Математична модель підтвердила, що запропонований метод впливає одночасно на всі складові рівняння часу завантаження і забезпечує вимірюваний результат по кожній з визначених метрик.

РОЗДІЛ 2

РОЗРОБКА ТА АРХІТЕКТУРА МЕТОДУ ОПТИМІЗАЦІЇ СТРУКТУРИ МЕРЕЖЕВИХ РЕСУРСІВ

2.1 Загальна архітектура та взаємодія компонентів системи

Для практичної реалізації описаного в першому розділі методу розроблено JavaScript-бібліотеку OptimizationManager. Вона складається рівно з двох файлів: OptimizationManager.js – основний клас з усією логікою керування ресурсами, та sw.js – модуль Service Worker, що відповідає за фонове кешування та перехоплення мережевих запитів. Саме така двофайлова структура є мінімально необхідною для повноцінної роботи всіх трьох механізмів оптимізації.

Головна перевага цього рішення – повна незалежність від конкретного технологічного стеку. Бібліотеку можна підключити до будь-якого сучасного вебпроекту без переписування існуючого коду. У межах цієї дипломної роботи для демонстрації та верифікації методу використано локальний вебсервер на базі Python та фреймворку Flask. Аналогічні результати можна відтворити використовуючи будь-який проєкт, написаний на будь-якій мові програмування, бо сама бібліотека залишається незмінною.

Щоб бібліотека запрацювала у проєкті треба виконати декілька кроків для підключення. Перше, що треба зробити, це додати у головний HTML-документ (зазвичай це index.html) тег <script src=”OptimizationManager.js”></ script> та після завантаження сторінки створити екземпляр класу і викликати метод init(). Наступним кроком є те, що файл sw.js має бути розміщений у корені домену, бо це є технічною вимогою браузерного API, яка надає воркеру право перехоплювати запити від усього сайту, а не лише від окремої сторінки. Серверний маршрут при цьому повинен коректно повертати файл sw.js із заголовком Content-Type: application/javascript. Без цього браузер просто відмовиться реєструвати воркер з міркувань безпеки.

Загальну структуру методу та взаємозв’язок між компонентами системи наведено на рисунку 2.1 Діаграма відображає три ключові групи механізмів оптимізації – мінімізацію ресурсів, відкладене завантаження та кешування. Вони об’єднані під керування єдиного модуля OptimizationManager. Результатом їх комплексної роботи є оптимізована версія сторінки, параметри якої вимірюються за чотирма метриками продуктивності.

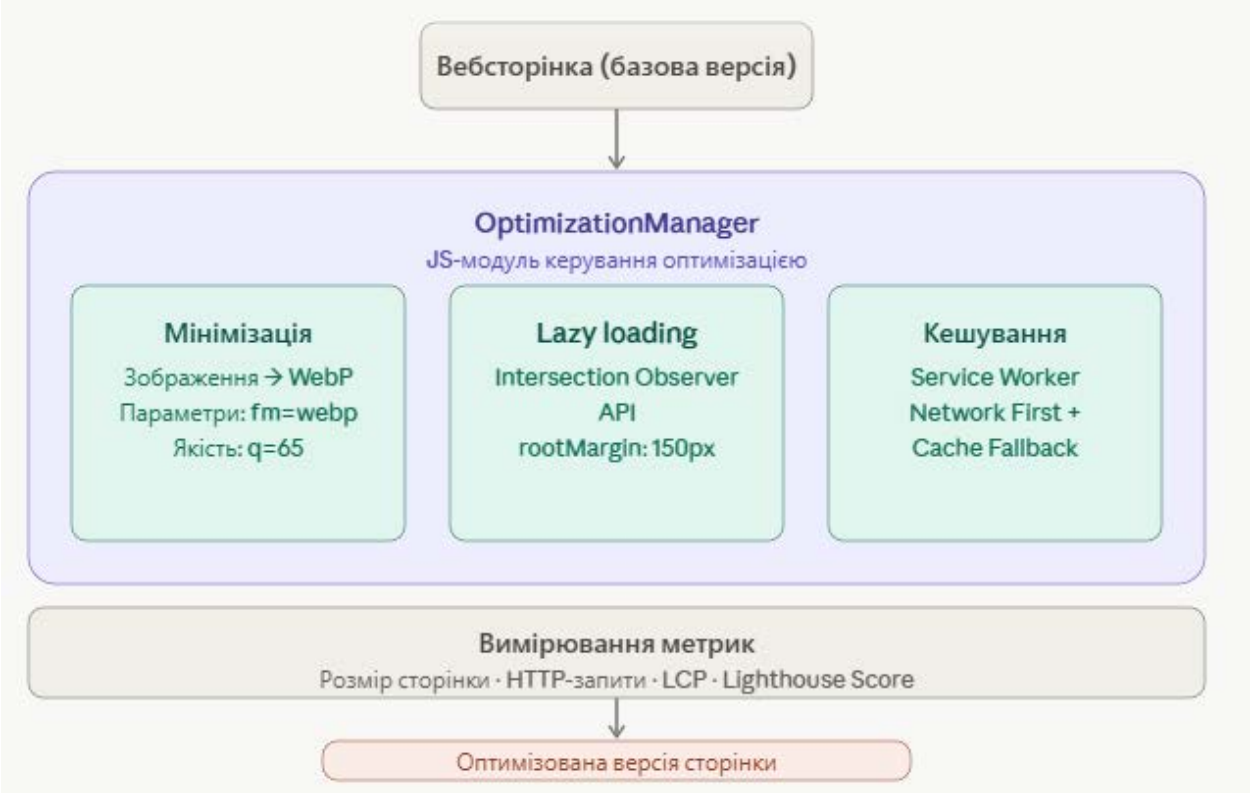


Рисунок 2.1 – Структура методу оптимізації вебсторінки

Комплекс оптимізації розділено на три взаємопов’язані рівні. Перший – контролер трансформації інтерфейсу (клас OptimizationManager) – ядро бібліотеки, яке виконується в основному потоці браузера. Воно відповідає за сканування DOM-дерева, відстеження прокручування та блокування важких ресурсів до моменту їх реальної необхідності. Другий рівень – фоновий мережевий проксі (Service Worker, файл sw.js) – ізольований скрипт у власному потоці на рівні браузерного API. Його завданням є перехоплення всіх мережевих запитів і керування локальним сховищем Cache Storage повністю прозоро для користувача. Третій рівень – серверний

конфігураційний маршрут. Тобто мінімальна серверна частина потрібна виключно для дотримання правил безпеки браузера при реєстрації Service Worker з кореневого домену.

Загальний сценарій роботи системи відбувається таким чином. Браузер завантажує HTML-документ і починає будувати DOM. Оскільки OptimizationManager.js підключається у заголовку сторінки, він перехоплює контроль над зображеннями ще до того, як браузер встигає надісати перший запит на їх завантаження. Паралельно виконується реєстрація Service Worker. Завдяки правильно налаштованому серверному маршруту воркер ініціалізується у корені проєкту, що надає йому повноваження на перехоплення абсолютно всіх мережевих запитів домену. Далі, коли користувач прокручує сторінку, бібліотека модифікує URL-запити в реальному часі, кожен з яких проходить через Service Worker, що вирішує – завантажити файл з мережі у WebP або миттєво повернути його з локального кешу.

2.2 Алгоритм автоматичної трансформації елементів DOM-дерева

Однією з ключових проектних рішень стала відмова від ручного маркування елементів розробником. Це означає, що не потрібно вручну додавати жодних спеціальних атрибутів до тегів у HTML. Бібліотека сама знаходить усі зображення на сторінці та бере їх під контроль. Такий підхід дозволяє впроваджувати бібліотеку в готові проєкти практично без зміни у їхньому коді.

Алгоритм реалізований у методі _transformStandardImages() і працює у кілька послідовних кроків. Система виконує вибірку всіх тегів <img>, які ще не перебувають під контролем бібліотеки, за допомогою CSS-селектора img:not([data-om-src]). Для кожного знайденого елемента зчитується оригінальне посилання з атрибута src і зберігається у службовому атрибуті data-om-src. Після цього стандартний атрибут src повністю видаляється з тегу.

Видалення src – це критична операція з точки зору економії трафіку. Бо браузер починає завантаження зображення лише тоді, коли виявляє валідне значення в атрибуті src. Якщо цей атрибут відсутній, то жоден мережевий запит не надсилається. Сторінка при цьому продовжує нормально функціонувати, але її початкова мережева вага зводиться до мінімуму.

2.3 Реалізація асинхронного відкладеного завантаження та динамічної модифікації ресурсів

Після того, як усі ресурси перехоплені та заблоковані, система переходить до їх інтелектуального підвантаження. Традиційні реалізації Lazy Loading часто будувались на постійному обчисленні координат елементів через подію scroll. Це навантажувало головний потік браузера і само по собі могло викликати мікрозатримки. У розробленій бібліотеці цю проблему вирішено через Intersection Observer API – вбудований браузерний механізм, що відстежує появу елементів у зоні видимості на рівні рушія, без залучення JavaScript-потоку. [9, 19]

Логіка відкладеного завантаження реалізована у методі _initLazyLoading(). Якщо режим lazy вимкнено, то усі зображення завантажуються одразу. Цей режим використовується для базових вимірювань без оптимізації. При увімкненому режимі створюється об’єкт IntersectionObserver з параметром rootMargin:’150px’. Це означає те, що зона спостереження розширюється на 150 пікселів нижче видимої межі екрана, і зображення починають завантажуватись трохи раніше, ніж з’являються на екрані, що усуває ефект порожніх місць при прокручуванні.

Як тільки спостерігач перетин елемента з розширеною зоною видимості, запускається метод getImageUrl(). Він аналізує поточний стан конфігурації. Якщо активовано режим максимальної оптимізації (this.state.minify==true), до URL автоматично дописується параметри fm=webp та q=65 – інструкція серверу конвертувати зображення у WebP зі стисненням 65%. При вимкненому режимі передається fm=jpg з якістю 95% - базовий стан для

порівняльних вимірювань. Готова адреса записується у стандартний src, браузер завантажує вже оптимізований файл, а спостерігач припиняє стежити за цим елементом. Для користувача весь цей процес залишається непомітним. [8, 17]

2.4 Проектування підсистеми низькорівневого мережевого перехоплення

Останнім та найважливішим рівнем архітектури є підсистема фонового кешування, яка реалізована у sw.js. Service Worker функціонує як проміжний вузол між браузером і зовнішньою мережею. Він отримує повний контроль над кожним байтом, що передається між клієнтом та сервером. [15, 16]

В основі логіки роботи Service Worker лежить подійне програмування, прив’язане до його унікального життєвого циклу. На етапі встановлення (install) воркера створюється первинне статичне сховище, куди примусово записуються базові файли сайту – шрифти, логотипи та глобальні файли стилів. Наступним є етап активації (activate). Він очищує застарілі версії кешу, гарантуючи, що користувач завжди працюватиме з актуальним кодом системи. Після успішного розгортання Service Worker переходить у стан постійного очікування події fetch, яка спрацьовує щоразу, коли вебсторінка намагається зробити будь-який мережевий запит.

Для обробки динамічного контенту та медіафайлів у системі було спроектовано та реалізовано адаптивну стратегію кешування під назвою Network-First with Cache Fallback (Спочатку мережа, з поверненням до кешу). Коли активована подія fetch, Service Worker перехоплює запит на завантаження оптимізованої системи WebP-картинки і в першу чергу намагається виконати реальне звернення до сервера через інтернет. Тобто якщо зв’язок стабільний, то файл успішно завантажується, і воркери паралельно роблять його копію у динамічне сховище Cache Storage, після чого віддають сторінці. [15, 18]

Але головна перевага цієї архітектури розкривається в умовах поганого або повністю відсутнього мобільного зв’язку. Якщо запит до мережі завершується помилкою через таймаут або обрив з’єднання, то Service Worker миттєво перехоплює контроль, блокує появу стандартної помилки браузера про відсутність інтернету та автоматично шукає копію цього зображення у локальному кеші. Коли вдається знайти потрібний файл, він миттєво повертає його сторінці, забезпечуючи повну працездатність вебсторінки. У результаті такого проектування затримка повторного доступу до ресурсів знижується до нуля (RTT -> 0), що забезпечує максимальну швидкість взаємодії із системою та суттєво економить загальний мережевий трафік.

2.5 Механізм управління конфігурацією та перемикання режимів

Для проведення коректних порівняльних вимірювань у бібліотеці реалізовано гнучкий механізм управління режимами роботи. Поточна конфігурація системи описується трьома булевими прапорцями: minify відповідає за конвертацію зображень у WebP і стиснення, lazy – за відкладене завантаження через Intersection Observer, cache – за активацію Service Worker. Стан цих прапорців зберігається у localStorage браузера, що забезпечує збереження налаштувань між сесіями.

Зміна будь-якого параметра виконується через метод toggle(), який викликається з консолі розробника. Наприклад, команда optimizer.toggle(‘minify’) вмикає або вимикає конвертацію у WebP. Метод інвертує значення відповідного прапорця, зберігає оновлений стан і перезавантажує сторінку для застосування змін. Такий підхід дозволяє за лічені секунди перемикатись між повністю оптимізованим і базовим режимами, що і є необхідною умовою для відтворюваних результатів вимірювань.

Висновки

1. У другому розділі описано повну архітектуру розробленого методу і детально розглянуто реалізацію кожного його компонента. Показано, що бібліотека складається з двох файлів - OptimizationManager.js та sw.js. І для її підключення достатньо лише додати один тег <script> у головний HTML-документ та розмістити sw.js у корені домену.
2. Клас OptimizationManager автоматично перехоплює всі зображення на сторінці, блокуючи їх завантаження до моменту появи у зоні видимості користувача.
3. Intersection Observer API вирішує задачу Lazy Loading без додаткового навантаження на головний потік браузера.
4. Метод getImageUrl() динамічно модифікує URL-адреси ресурсів, додаючи параметри конвертації у WebP за стисненням 65%, що безпосередньо зменшує значення S_j у математичній моделі.
5. Service Worker реалізує стратегію Network-First with Cache Fallback, що забезпечує мінімальний час відповіді при повторних запитах та стабільну роботу навіть при відсутності з’єднання.
6. Сукупна дія всіх компонентів спрямована на одночасне покращення трьох цільових метрик: LCP, Total Transferred Data та Request Count.

РОЗДІЛ 3

ПОРІВНЯЛЬНИЙ АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ОПТИМІЗАЦІЇ ВЕБРЕСУРСІВ

Перш ніж оцінювати результати розробленого методу, необхідно зрозуміти, які підходи до вирішення тієї ж задачі вже існують. Без такого аналізу неможливо обґрунтувати доцільність нової розробки і показати, чим вона відрізняється від того, що вже є на ринку. У цьому розділі буде розглянуто чотири категорії існуючих рішень, кожне з яких вирішує проблему мережевого трафіку по-своєму і з різним ступенем повноти.

3.1 Нативний атрибут браузера loading=”lazy”

Найпростішим і найдоступнішим інструментом відкладеного завантаження є стандартний HTML-атрибут loading=”lazy”, який підтримується всіма сучасними браузерами починаючи з 2019-2020 років. Щоб його використати, достатньо додати лише один атрибут до тегу зображення <img src=”photo.jpg” loading=”lazy”>. Браузер сам побачить та визначить, чи знаходиться зображення у зоні видимості. І лише потім завантажить його лише тоді, коли воно наближається до екрана. [9, 10]

Перевага цього методу є очевидною. Він не вимагає жодних зовнішніх бібліотек і додаткового JavaScript. Браузер реалізує логіку на рівні свого рушія, що є максимально ефективним з точки зору продуктивності. Але на цьому можливості атрибута закінчується. Він вирішує лише одну задачу зі скорочення трафіку – відкладає завантаження зображень. Конвертація у сучасний формат, стиснення, кешування – все це залишається поза його межами. Також варто зазначити, що атрибут потрібно вручну прописувати до кожного тегу, що у великих проєктах з динамічно генерованим контентом перетворюється на реальну проблему.

3.2 Бібліотека lazysizes

Бібліотека lazysizes є одним із найпопулярніших JavaScript-рішень для реалізації відкладеного завантаження. На відміну від нативного атрибута вона пропонує розширені можливості, такі як підтримку зображень у тегах <picture>, адаптивних зображень через srcset, а також інтеграцію з різними плагінами через систему розширень. Бібліотека побудована на Intersection Observer API, і вважається, що вона доволі мало важить. Її розмір складає близько 3 КБ у стисненому вигляді. [9, 19]

Але lazysizes вирішує виключно задачу відкладеного завантаження. Конвертація зображень у формат WebP, управління його якістю, кешування на рівні Service Worker – це все знаходиться поза межами її відповідальності. Для того, щоб зібрати повноцінне рішення на основі lazysizes, розробнику доведеться додатково підключати окремі інструменти для кожної задачі і самостійно забезпечувати їх спільну роботу. Ще один суттєвий недолік – бібліотека також вимагає ручного маркування. Тобто кожне зображення потрібно переписати, замінивши стандартний атрибут src на data-src. [10, 20]

3.3 Мережі доставки контенту (CDN)

CDN (Content Delivery Network) – це розподілена мережева інфраструктура, яка зберігає копії статичних файлів сайту на серверах у різних географічних точках світу. Коли користувач відкриває сторінку, файли завантажуються не з основного сервера, а з найближчого вузла мережі, що суттєво скорочує фізичну відстань передачі даних і знижує затримку. Найвідоміші представники цього класу рішень є Cloudflare, AWS CloudFront та Fastly. [1]

Сучасні CDN-провайдери нерідко пропонують вбудовані функції оптимізації зображень. Це може бути як і автоматична конвертація у WebP при детектуванні підтримки браузером, так і зміну розміру на льоту з базовим кешуванням на рівні мережевих вузлів. Це робить CDN потужним

інструментом для великих комерційних проєктів. Але у такого підходу є серйозні обмеження. CDN – платна зовнішня інфраструктура, яка потребує DNS, контролю над доменом і, як правило абонентської плати. Для звичайних, навчальних чи некомерційних проєктів це часто є неприйнятним. Крім того, CDN не вирішує задачу відкладеного завантаження на стороні клієнта і не управляє поведінкою браузера при прокручуванні сторінки.

3.4 Інструменти статичної конвертації зображень (Squoosh, imagemin)

Окрему категорію складають інструменти для пакетної обробки зображень на етапі складання проєкту. Squoosh від Google – це як вебзастосунок, так і CLI-утиліта, яка дозволяє конвертувати зображення у WebP, AVIF та інші сучасні формати з гнучким налаштуванням рівня стиснення. Imagemin – Node.js-бібліотека для автоматизованої оптимізації графіки у процесі збірки фронтенду, яка інтегрується з Webpack, Vite та аналогічними інструментами. [6, 7, 13]

Головна перевага цих рішень – висока якість стиснення, оскільки обробка виконується заздалегідь і не обмежена часом відповіді сервера. Але разом із тим ці інструменти вирішують виключно задачу підготовки файлів до публікації. Вони не мають відношення до того, як і коли ці файли передаються браузеру. Відкладене завантаження, динамічне керування запитами, кешування – все це залишається поза їх зоною відповідальності. Для кожної з цих задач знову потрібно підключати окреме рішення.

3.5 Порівняльний аналіз та обґрунтування переваг розробленого методу

Аналіз розглянутих рішень показує спільну закономірність, що кожне з них вирішує лише одну певну проблему зі скорочення трафіку. Атрибут loading=”lazy” і бібліотека lazysizes займаються виключно відкладанням завантаженням. CDN – географічним наближенням контенту і частково конвертацією форматів, але потребує зовнішньої платної інфраструктури.

Squoosh та Imagemin обробляють зображення на етапі збірки, але не впливають на поведінку браузера під час роботи сторінки. [9, 13, 19, 20]

Щоб отримати повноцінний ефект від оптимізації, розробнику доводиться самостійно збирати систему з декількох незалежних інструментів, забезпечуючи їх сумісність і підтримувати цей набір в актуальному стані. Жодне з перелічених рішень не пропонує комплексного підходу в рамках однієї бібліотеки.

Саме цю прогалину заповнює розроблений метод. OptimizationManager об’єднує всі три механізми оптимізації – відкладене завантаження, динамічну конвертацію у WebP та Service Worker кешування, та створює єдину систему кешування, яка підключається двома файлами та не потребує ні ручного маркування HTML, ні зовнішньої серверної інфраструктури, ні зміни процесу збірки проєкту.

Таблиця 3.1 – Порівняння існуючих рішень з розробленим методом

Критерій	loading=”lazy”	lazysizes	CDN	Squoosh, imagemin	Optimization Manager
Lazy Loading	✓	✓	-	-	✓
Конвертація у WebP	-	-	✓ (частково)	✓	✓
Кешування (SW)	-	-	-	-	✓

Не потребує ручного маркування	-	-	✓	-	✓
Не потребує серверної інфраструктури	✓	✓	-	✓ (збірка)	✓
Комплексне рішення	-	-	-	-	✓

З таблиці 3.1 видно, що OptimizationManager є одним єдиним рішенням, яке одночасно реалізує всі три ключові механізми оптимізації трафіку, при цьому не потребуючи ні ручного маркування елементів, ні зовнішньої інфраструктури. Це робить його універсальним інструментом, придатним для впровадження у будь-який вебпроєкт незалежно від його архітектури та технологічного стеку.

Висновки

1. У третьому розділі проведено порівняльний аналіз чотирьох категорій існуючих рішень для оптимізації мережевого трафіку вебсторінок: нативного атрибуту loading=”lazy”, бібліотеки lazysizes, мереж доставки CDN та інструментів статичної конвертації зображень Squoosh та imagemin.
2. Встановлено, що кожне з розглянутих рішень охоплює лише одну складову задачі оптимізації і не пропонує комплексного підходу. Для отримання повноцінного ефекту розробнику необхідно самостійно інтегрувати кілька несумісних між собою інструментів.
3. Розроблений метод на базі бібліотеки OptimizationManager усуває цей недолік, об’єднуючи відкладене завантаження, динамічну конвертацію

у WebP та Service Worker кешування в єдину автономну систему без залежностей від зовнішньої інфраструктури.

РОЗДІЛ 4

ЕКСПЕРИМЕНТАЛЬНА ВЕРИФІКАЦІЯ ТА АНАЛІЗ ЕФЕКТИВНОСТІ МЕТОДУ

4.1 Опис тестового середовища та методологія проведення вимірювань

Для практичної перевірки ефективності розробленого методу організовано експеримент на двох тестових вебресурсах з принципово різною структурою контенту. Такий підхід дозволяє оцінити, наскільки результати оптимізації залежать від типу сайту, і зробити обґрунтовані висновки про межі застосування методу.

Перший тестовий стенд – фотогалерея тревел-блогу “Wanderlust”. Це односторінковий вебресурс із достатньою кількістю повнорозмірних фотографій у форматі JPEG, завантажених із зовнішнього репозиторію зображень. Саме зображення складають понад 90% загального обсягу переданих даних, що робить цей стенд ідеальним для оцінки ефективності конвертації у WebP та відкладеного завантаження.

Другий тестовий стенд – інформаційний портал навчального кампусу університету (Campus Services). Це повноцінний застосунок на базі Flask (Python) і PostgreSQL із системою автентифікації, трьома ролями користувачів та CRUD-операціями. На відміну від галереї, цей портал практично не містить зображень – його трафік складається переважно з HTML-документів, CSS-стилів, файли JavaScript та шрифти. Саме цей контраст між двома стендами дозволяє нам визначити, за яких умов метод дає максимальний ефект, а за яких – мінімальний.

Бекенд-частина фотогалереї також функціонувала на локальному сервері Flask. Обидві програми були запущені на localhost з портами 63342 та 5000 відповідно. Для вимірювань використовувався браузер Chrome з відкритими інструментами розробника (DevTools, клавіша F12).

Методологія вимірювань побудована за принципом контрольованого експерименту з двома чітко визначеними станами системи. Перший стан – базовий, коли всі три механізми оптимізації вимкнено (minify: false, lazy: false, cache: false). У цьому стані бібліотека підключена до сторінки, але не вносить жодних змін, тобто зображення завантажуються в оригінальному форматі JPEG без будь-якої обробки. Другий стан – повністю оптимізований, коли всі три механізми активовано (minify: true, lazy: true, cache: true). Перемикання між станами здійснювалося за допомогою методу toggle() у консолі DevTools з обов'язковим очищенням localStorage перед кожним вимірюванням.

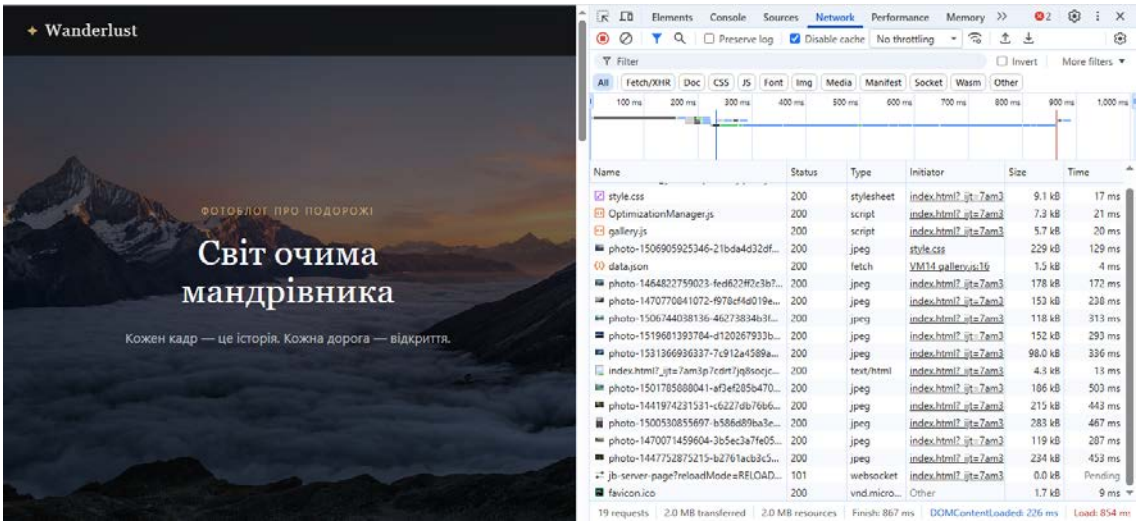
Перед кожним вимірюванням виконувалося три обов'язкові підготовчі дії: повне очищення кешу браузера (Ctrl+Shift+R), увімкнення опції «Disable cache» на вкладці «Network» та перевірка поточного стану конфігурації бібліотеки через консоль. Це гарантувало, що кожне вимірювання було незалежним і не спотвореним даними попередніх сеансів.

Для аналізу результатів використовувалися два інструменти DevTools. Панель «Network» фіксувала фактичний обсяг переданих даних, кількість HTTP-запитів та формат кожного ресурсу. Інструмент Lighthouse у режимі робочого столу виконав автоматизований аудит продуктивності та виміряв метрику LCP (Largest Contentful Paint).

4.2 Результати вимірювань для фотогалереї тревел-блогу “Wanderlust”

Фотогалерея є найбільш ілюстративним тестовим стендом для демонстрації ефективності методу, оскільки її трафік майже повністю складається з зображень JPEG. Саме на цьому типі контенту механізм перетворення WebP та відкладеного завантаження проявляється найчіткіше. [2]

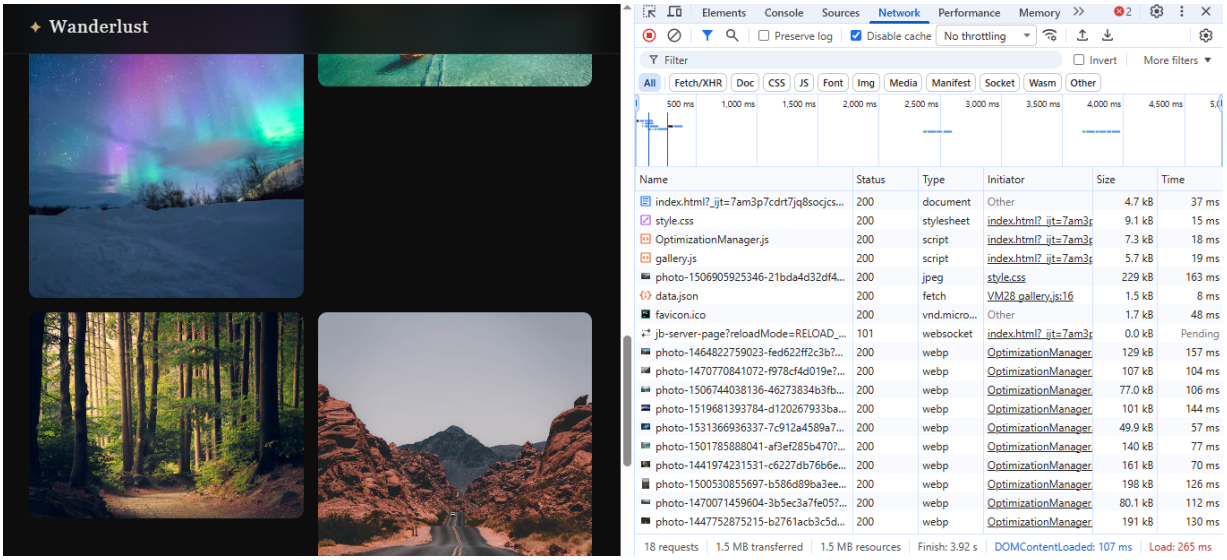
На рисунку 4.1 показано стан панелі «Network» в базовому режимі (усі механізми вимкнено). Браузер завантажив 19 ресурсів загальним обсягом 2,0



МБ. У стовпці «Type» показано, що всі зображення передаються у форматі jpeg – саме так, як вони зберігаються на сервері, без будь-якої оптимізації.

Рисунок 4.1 — Панель Network DevTools у базовому режимі (all false): 19 запитів, 2.0 МБ, формат jpeg

На рисунку 4.2 показано той самий стан після ввімкнення повного пакета оптимізації (всі механізми ввімкнено). Загальний обсяг переданих даних зменшився до 1,5 МБ, кількість запитів зменшилась до 18, а в стовпці "Type" тепер відображається webp замість jpeg. Це підтверджує, що метод getImageUrl() правильно додає параметри fm=webp до кожного запиту, і



сервер повертає зображення в новому форматі.

Рисунок 4.2 — Панель Network DevTools після оптимізації (all true): 18 запитів, 1.5 МБ, формат webp

Для отримання більш реалістичної картини були проведені додаткові вимірювання в умовах емуляції мобільного з'єднання 3G за допомогою функції Throttling у DevTools. На рисунку 4.3 показано базовий стан за умов 3G: сторінка здійснила 19 запитів, передала 2,0 МБ, а загальний час завантаження склав 48,32 секунди — типова ситуація для мобільного користувача з нестабільним покриттям.

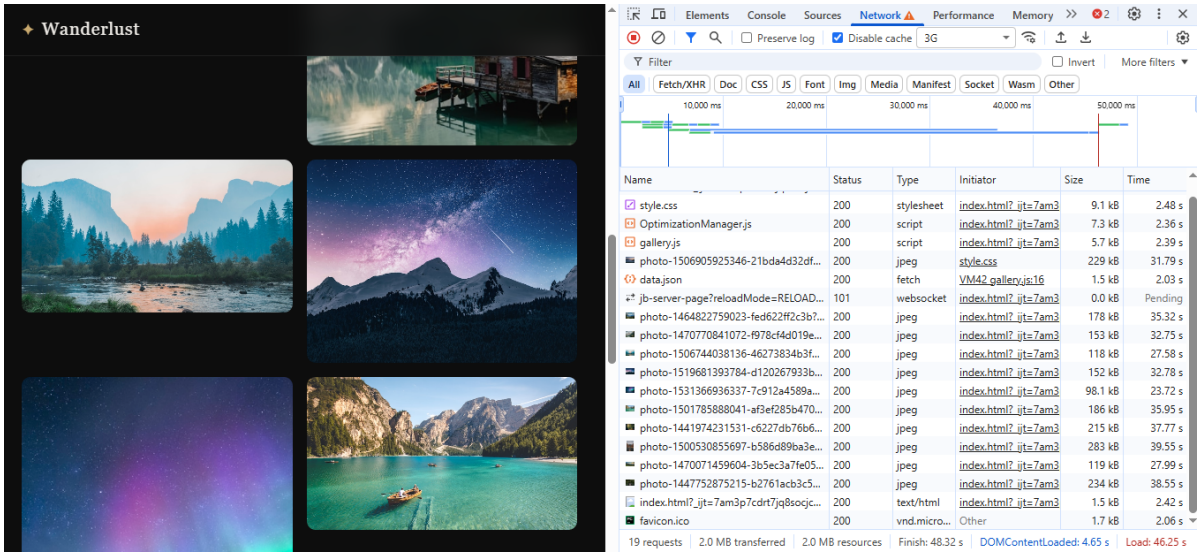


Рисунок 4.3 — Панель Network при емуляції 3G у базовому режимі: 2.0 МБ, Finish 48.32 с

На рисунку 4.4 показано результат при увімкненій оптимізації на 3G. Обсяг переданих даних залишився стабільним — 1.5 МБ, що відповідає 25% економії трафіку порівняно з базовим станом. Час Finish склав 47.43 секунди. Невелика різниця у часі завантаження між базовим і оптимізованим режимом на 3G пояснюється особливістю роботи Lazy Loading на повільному з'єднанні: зображення завантажуються поступово в міру прокручування, тому загальний Finish залишається схожим, але початковий екран з'являється швидше.

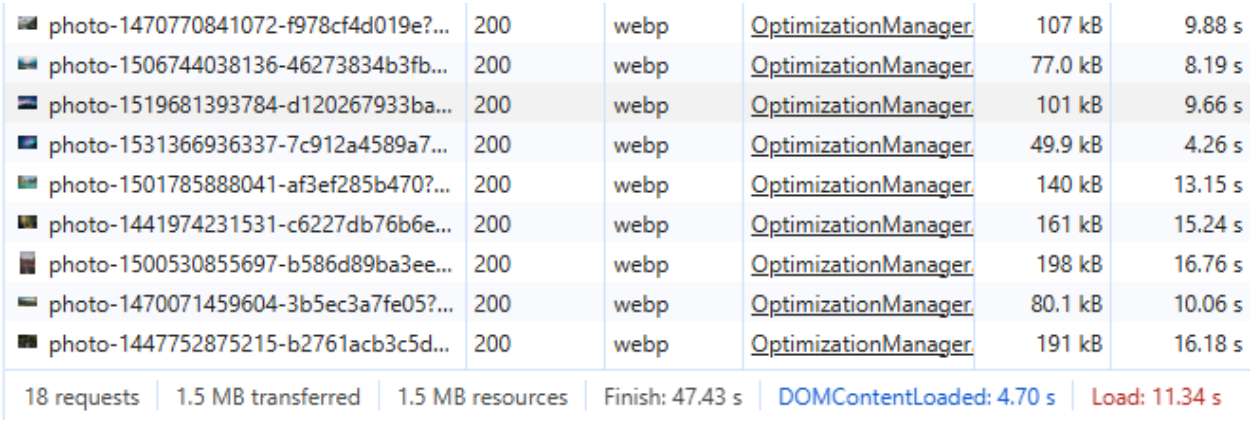


Рисунок 4.4 — Панель Network при емуляції 3G після оптимізації: 1.5 МБ, Finish 47.43 с

Таблиця 4.1 — Зведені результати вимірювань для фотогалереї Wanderlust

Метрика	До оптимізації	Після оптимізації	Різниця (%)
Total Transferred Data (без throttling)	2.0 МБ	1.5 МБ	−25%
Request Count (без throttling)	19	18	−5.3%
Finish (без throttling)	867 мс	3.92 с	— *
Total Transferred Data (3G)	2.0 МБ	1.5 МБ	−25%
Finish (3G)	48.32 с	47.43 с	−1.8%
Формат зображень	JPEG	WebP	—

* Збільшення Finish без throttling після оптимізації з 867 мс до 3.92 с пояснюється тим, що Lazy Loading розтягує процес завантаження зображень у часі — браузер підвантажує їх поступово у міру прокручування, а не одним

пакетом. Для кінцевого користувача це є перевагою, оскільки перший екран з'являється миттєво.

4.3 Результати вимірювань для кампус-порталу університету

Кампусний портал є принципово іншим тестовим стендом за структурою. Його головна сторінка містить статистичні блоки, форми та текстові елементи, але практично не має растрових зображень – лише шрифти, CSS та JavaScript. Це дозволяє оцінити поведінку методу в умовах, коли основний фактор оптимізації (зображення) відсутній.

Перед початком вимірювань було перевірено правильність реєстрації Service Worker. На рисунку 4.5 показано вкладку Application DevTools зі статусом worker «активовано та запущено» – це підтверджує, що файл sw.js був успішно зареєстрований з кореневого маршруту /sw.js сервера Flask та отримав повноваження перехоплювати всі мережеві запити домену.

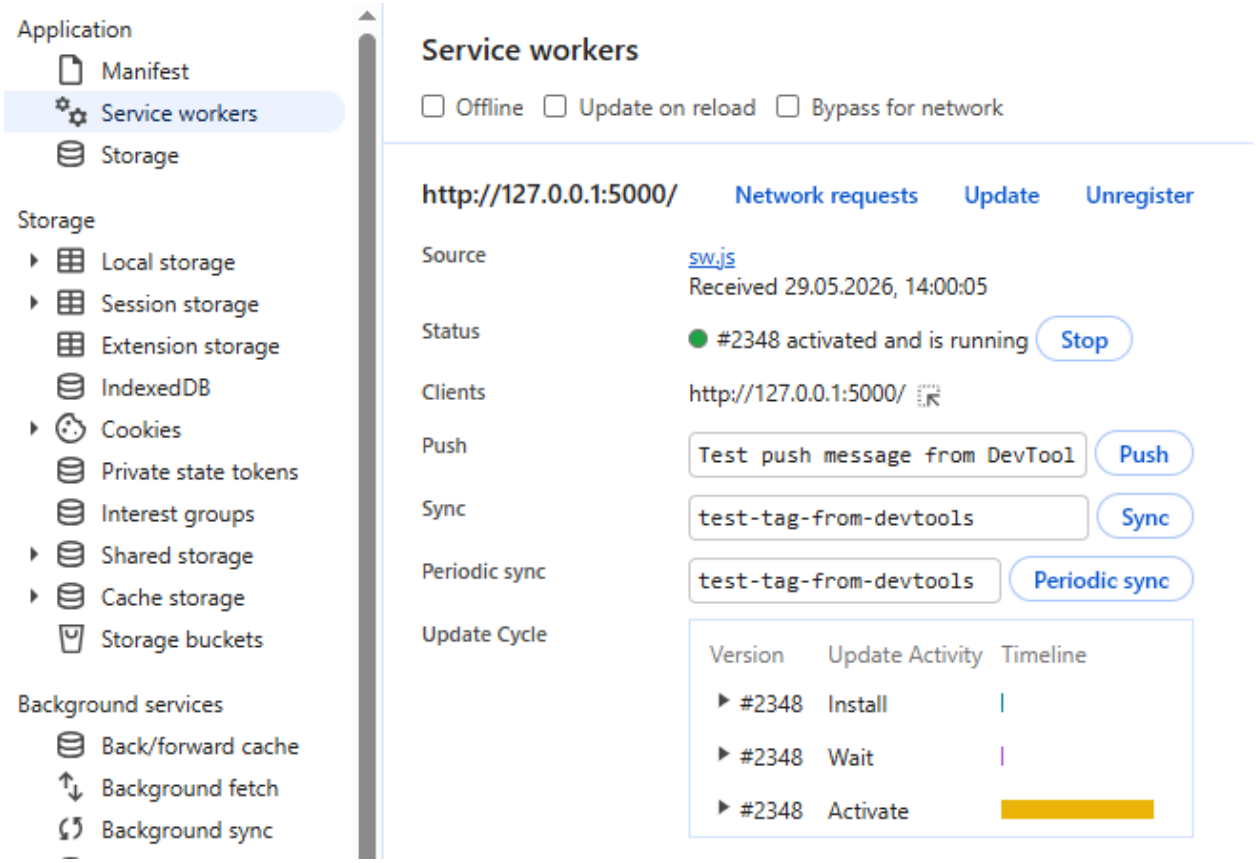


Рисунок 4.5 — Service Worker зареєстровано та активовано

Рисунок 4.6 демонструє панель Network кампус-порталу у базовому режимі. Сторінка виконала 8 запитів і передала 124 КБ. Серед ресурсів — HTML-документ, два CSS-файли, два шрифти з Google Fonts та JavaScript-файл OptimizationManager.js. Жодного зображення у форматі JPEG або PNG немає.

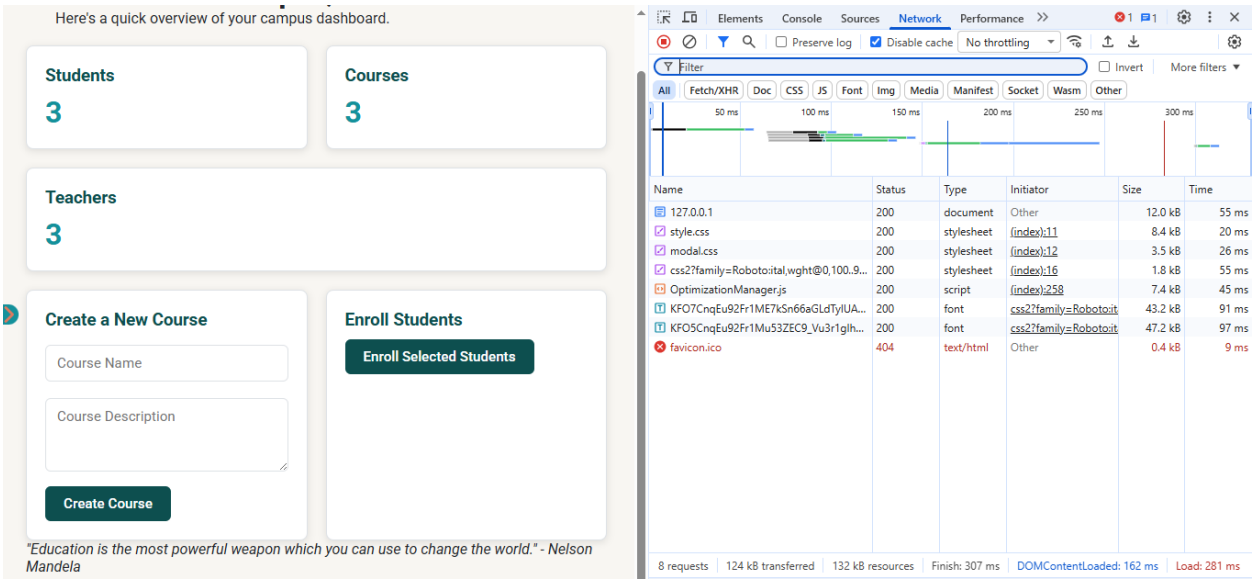


Рисунок 4.6 — Панель Network кампус-порталу у базовому режимі: 8 запитів, 124 КБ

Після ввімкнення повного пакету оптимізації (рис. 4.7) показники трафіку залишилися незмінними — 8 запитів, 124 КБ. Це цілком очікуваний і логічний результат: бібліотека OptimizationManager не знайшла жодних тегів <img> із посиланням на зовнішній ресурс, тому конвертація в WebP та відкладене завантаження просто не мали до чого застосовуватись. Відсутність змін у трафіку є коректною поведінкою системи, а не її недоліком.

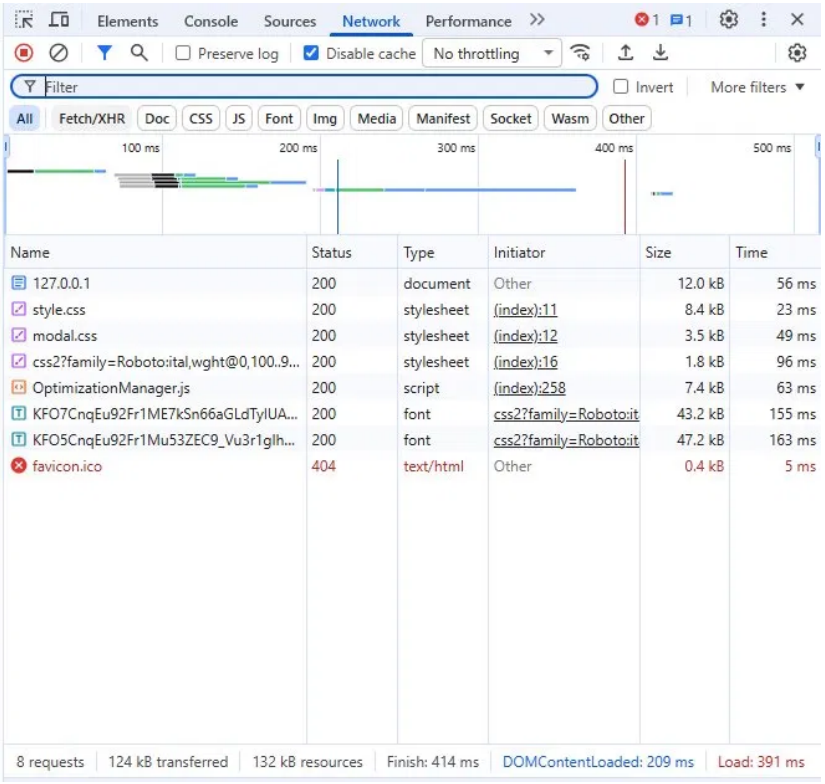


Рисунок 4.7 — Панель Network кампус-порталу після оптимізації: 8 запитів, 124 КБ (без змін)

Водночас метрика LCP показала покращення завдяки роботі Service Worker. На рисунку 4.8 зафіксовано базовий LCP = 0.5 с. Після увімкнення оптимізації (рисунок 4.9) LCP скоротився до 0.4 с — покращення на 20%. Це пояснюється тим, що при повторному завантаженні сторінки Service Worker повертає CSS-файли та шрифти Google Fonts безпосередньо з локального кешу Cache Storage без звернення до мережі, що скорочує час до появи

основного контенту на екрані. [3, 4]

Рисунок 4.8 — Метрики Lighthouse кампус-порталу у базовому режимі: LCP = 0.5 с

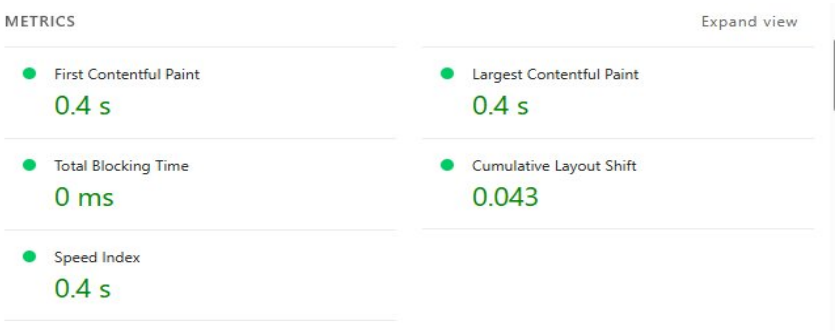


Рисунок 4.9 — Метрики Lighthouse кампус-порталу після оптимізації: LCP = 0.4 с

Таблиця 4.2 — Зведені результати вимірювань для кампус-порталу університету

Метрика	До оптимізації	Після оптимізації	Різниця (%)
Total Transferred Data	124 КБ	124 КБ	0%
Request Count	8	8	0%
LCP (Lighthouse Desktop)	0.5 с	0.4 с	−20%
Finish	307 мс	414 мс	— **

** Незначне збільшення Finish після оптимізації на кампус-порталі (з 307 до 414 мс) пов'язане з початковою реєстрацією Service Worker при першому завантаженні, яка виконується паралельно з рендерингом сторінки і додає невелику затримку. При повторних відвідуваннях ця затримка відсутня.

4.4 Зведений порівняльний аналіз результатів

Зведені результати обох тестових стендів наведено в таблиці 4.3. Порівняння дозволяє виявити закономірність, яка є ключовим практичним висновком дослідження: ефективність методу прямо пропорційна частці зображень у загальному обсязі трафіку сторінки.

Таблиця 4.3 — Зведені результати вимірювань по обох тестових стендах

Тестовий стенд	Метрика	До	Після	Ефект
Фотогалерея	Transferred (без throttling)	2.0 МБ	1.5 МБ	−25%
	Transferred (3G)	2.0 МБ	1.5 МБ	−25%
	Request Count	19	18	−5.3%
	Формат зображень	JPEG	WebP	—
Кампус-портал	Transferred Data	124 КБ	124 КБ	0%
	Request Count	8	8	0%

На фотогалереї, де зображення займають понад 90% трафіку, конвертація у WebP забезпечила стабільну економію 25% як при звичайному з'єднанні, так і при 3G. Кількість початкових запитів скоротилась з 19 до 18 завдяки Lazy Loading, який не завантажував зображення поза зоною видимості. На кампус-порталі, де зображень немає взагалі, механізми конвертації та відкладеного завантаження не дали ефекту по трафіку — і це абсолютно коректна поведінка системи. Єдиним механізмом що дав вимірюваний результат на порталі є Service Worker кешування — LCP покращився на 20% завдяки миттєвому поверненню CSS та шрифтів з локального сховища.

4.5 Перевірка відповідності результатів математичній моделі

Отримані експериментальні дані необхідно порівняти з теоретичною моделлю, сформованою у підрозділі 1.6.

$$T_{total} = \sum_{i=1}^{N} (RTT_i + T_{dns} + T_{tcp} + T_{ssl}) + \sum_{j=1}^{M} \frac{S_j}{B}$$

Перший механізм — зменшення Sj (розміру файлів) — підтверджено на фотогалереї: конвертація у WebP скоротила сумарний обсяг графічного контенту з 2.0 до 1.5 МБ, що відповідає теоретичному прогнозу зменшення на 25–34%. Це значення стабільно відтворюється як при звичайному з'єднанні, так і при 3G, що підтверджує незалежність ефекту від умов мережі.

Другий механізм — зменшення N (кількості початкових запитів) — підтверджено: Request Count скоротився з 19 до 18. Lazy Loading відсікає запити до зображень які знаходяться за межами початкового екрана і завантажує їх лише тоді, коли користувач доскролює до них. Ефект по кількості запитів на даному стенді невеликий, оскільки усі зображення з'являються на сторінці досить швидко при прокручуванні.

Третій механізм — RTT → 0 для повторних запитів — підтверджено на кампус-порталі: Service Worker повертав CSS і шрифти Google Fonts з кешу без звернення до мережі, що призвело до скорочення LCP на 20%. Цей ефект не відображається у показнику Total Transferred Data, але безпосередньо впливає на час відображення контенту.

Таким чином, усі три складові математичної моделі отримали практичне підтвердження в ході експерименту. Ступінь ефекту по кожній складовій визначається структурою конкретного вебресурсу: на графічно насичених сайтах домінує перший і другий механізм, на текстових порталах — третій.

Висновки

1. У четвертому розділі проведено повну експериментальну верифікацію розробленого методу на двох тестових стендах з контрастно різним типом контенту.
2. Визначено методологію вимірювань, що забезпечує відтворюваність і коректність результатів: двоетапне тестування з обов'язковим очищенням кешу та конфігурації перед кожним виміром.
3. На фотогалереї тревел-блогу "Wanderlust" конвертація зображень у формат WebP забезпечила стабільне скорочення обсягу переданих даних на 25% — як при звичайному з'єднанні, так і при емуляції мобільного 3G. Кількість HTTP-запитів скоротилась з 19 до 18 завдяки механізму відкладеного завантаження.
4. На кампус-порталі університету, де растрові зображення відсутні, механізми конвертації та Lazy Loading не вплинули на обсяг трафіку — що є коректною і очікуваною поведінкою системи. Service Worker забезпечив покращення LCP на 20% завдяки кешуванню статичних ресурсів. Результати експерименту підтвердили відповідність усіх трьох складових математичній моделі розділу 1.

ЗАГАЛЬНІ ВИСНОВКИ ПО РОБОТІ

1. У дипломній роботі вирішено важливе науково-практичне завдання, яке полягає в оптимізації структури ресурсів вебсторінок безпосередньо на стороні клієнта для зниження навантаження на телекомунікаційні мережі та прискорення взаємодії користувача з інтерфейсом.
2. Проведені теоретичні та експериментальні дослідження дозволили повністю реалізувати поставлену мету та підтвердили працездатність створеного методу.
3. На основі аналізу структури сучасного вебтрафіку було розроблено та програмно реалізовано клієнтську JavaScript-бібліотеку OptimizationManager, яка працює у зв'язці з фоновим проксі-скриптом Service Worker. Запропонований підхід дозволив автоматизувати процес динамічної трансформації DOM-дерева сторінки та перехоплення мережевих потоків безпосередньо в браузері. Це виключає необхідність внесення змін до архітектури чи налаштувань серверної частини сайту, роблячи рішення повністю універсальним.
4. Практична верифікація розробленого методу здійснювалася шляхом тестування системи на двох експериментальних майданчиках із принципово різним типом контенту, що дозволило перевірити всі три складові побудованої математичної моделі. Для графічно насиченого середовища випробування проводилися на базі фотогалереї тревел-блогу "Wanderlust", де перехід на прогресивний формат кодування WebP забезпечив стабільне скорочення обсягу переданих по мережі даних на 25%. Водночас застосування механізму відкладеного завантаження ресурсів дозволило оптимізувати чергу TCP-сесій та зменшити кількість стартових HTTP-запитів до сервера з 19 до 18, що знижує ризик переповнення мережевих буферів обладнання на початку сесії.

5. Для перевірки роботи методу на вебресурсах із переважно текстовим та інструктивним наповненням тестування було реалізовано на базі інформаційного кампус-порталу університету. Оскільки растрові зображення на цій сторінці були відсутні, механізми динамічної зміни форматів графіки та відкладеного завантаження очікувано не вплинули на підсумковий обсяг переданого трафіку. Проте на цьому стенді повноцінно проявив себе третій механізм моделі — фонове низькорівневе проксі-кешування. Завдяки налаштуванню Сервіс Воркера вдалося забезпечити миттєве підвантаження сторонніх ресурсів Google Fonts безпосередньо з локального сховища пристрою без звернення до зовнішньої мережі. Це дозволило скоротити метрику часу рендерингу найбільшого контенту LCP на 20%, що безпосередньо покращило швидкість відображення сторінки для користувача.
6. Таким чином, результати проведених експериментів повністю довели інженерну цінність розробленого клієнт-орієнтованого методу. Його впровадження дозволяє гнучко адаптувати інтенсивність використання мережевих ресурсів під специфіку конкретного сайту: на графічно насичених порталах ефект досягається за рахунок зниження ваги медіафайлів, а на текстових системах — за рахунок локального проксі-кешування сторонніх ресурсів, що суттєво розвантажувати канали телекомунікаційних систем під час пікових навантажень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. HTTP Archive. Page Weight // The 2024 Web Almanac. — 2024. — URL: https://almanac.httparchive.org/en/2024/page-weight (дата звернення: 15.03.2026).
2. HTTP Archive. Media // The 2024 Web Almanac. — 2024. — URL: https://almanac.httparchive.org/en/2024/media (дата звернення: 15.03.2026).
3. Google Chrome Developers. Largest Contentful Paint. Lighthouse documentation. — 2023. — URL: https://developer.chrome.com/docs/lighthouse/performance/lighthouse-largest-contentful-paint?hl=en (дата звернення: 17.05.2026).
4. Google Chrome Developers. Lighthouse performance scoring. — 2023. — URL: https://developer.chrome.com/docs/lighthouse/performance/performance-scoring?hl=en (дата звернення: 17.05.2026).
5. HTTP Archive. Performance // The 2024 Web Almanac. — 2024. — URL: https://almanac.httparchive.org/en/2024/performance (дата звернення: 15.03.2026).
6. Öztürk E., Mesut A. Performance Evaluation of JPEG Standards, WebP and PNG in Terms of Compression Ratio and Time for Lossless Encoding // 2021 6th International Conference on Computer Science and Engineering (UBMK). — IEEE, 2021. — P. 15–20. — DOI: 10.1109/UBMK52708.2021.9558922 (дата звернення: 10.04.2026).
7. Google Developers. WebP Compression Study. — 2024. — URL: https://developers.google.com/speed/webp/docs/webp_study?hl=en (дата звернення: 14.04.2026).
8. W3C Web Platform Working Group. Intersection Observer. W3C Working Draft. — 2023. — URL: https://www.w3.org/TR/intersection-observer (дата звернення: 15.05.2026).

9. Google web.dev. Lazy load images and iframe elements. — 2023. — URL: https://web.dev/learn/performance/lazy-load-images-and-iframe-elements?hl=en (дата звернення: 20.04.2026).
10. Google web.dev. The performance effects of too much lazy loading. — 2022. — URL: https://web.dev/articles/lcp-lazy-loading?hl=en (дата звернення: 15.05.2026).
11. Grigorik I. High Performance Browser Networking. — Sebastopol : O'Reilly Media, 2013. — 408 p. — URL: https://hpbn.co (дата звернення: 15.05.2026).
12. Nah F. F.-H. A study on tolerable waiting time: how long are Web users willing to wait? // Behaviour & Information Technology. — 2004. — Vol. 23, No. 3. — P. 153–163. — DOI: 10.1080/01449290410001669914 (дата звернення: 18.03.2026).
13. Dornauer B., Felderer M. Web Image Formats: Assessment of Their Real-World-Usage and Performance Across Popular Web Browsers // Product-Focused Software Process Improvement. — Springer, 2023. — P. 119–134. — DOI: 10.1007/978-3-031-49266-2_9 (дата звернення: 20.05.2026).
14. Čegan L. Improving the Performance of Loading Web Pages in the Environment With High Latency Connections // 2015 25th International Conference Radioelektronika (RADIOELEKTRONIKA). — IEEE, 2015. — P. 458–461. — DOI: 10.1109/RADIOELEK.2015.7128988 (дата звернення: 20.04.2026).
15. W3C Web Applications Working Group. Service Workers. W3C Candidate Recommendation. — 2022. — URL: https://www.w3.org/TR/service-workers (дата звернення: 15.05.2026).
16. MDN Web Docs. Service Worker API. — Mozilla Contributors, 2026. — URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API (дата звернення: 15.05.2026).
17. MDN Web Docs. Intersection Observer API. — Mozilla Contributors, 2026. — URL: https://developer.mozilla.org/en-

US/docs/Web/API/Intersection_Observer_API (дата звернення: 15.05.2026).

18. MDN Web Docs. Cache API. — Mozilla Contributors, 2025. — URL: https://developer.mozilla.org/en-US/docs/Web/API/Cache (дата звернення: 17.05.2026).
19. Turcotte A., Gokhale S., Tip F. Increasing the Responsiveness of Web Applications by Introducing Lazy Loading // 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023). — IEEE, 2023. — P. 459–470. — DOI: 10.1109/ASE56229.2023.00192 (дата звернення: 20.05.2026).
20. Iddrisu S., Asenso D., Asiedu W. Improving Web Performance with Lazy Loading and Minification Algorithms // Journal of Web Development and Web Designing. — 2025. — Vol. 10, No. 1. — P. 1–11. — URL: https://matjournals.net/engineering/index.php/JoWDWD/article/view/1377 (дата звернення: 28.04.2026).

ДОДАТКИ

ДОДАТОК А

```
/**
 * @class OptimizationManager
 * @classdesc Універсальна бібліотека для автоматичної оптимізації зображень та керування мережевими ресурсами сторінки.
 * @author Володіна Альона, група ТІ-21
 */
class OptimizationManager {

  /**
   * Ініціалізує екземпляр класу, налаштовує конфігурацію та стан системи.
   * @param {Object} [config={}] - Об'єкт користувацьких налаштувань.
   * @param {string} [config.rootMargin='150px'] - Зона навколо viewport для завчасного завантаження зображень.
   */
  constructor(config = {}) {
    // 1. Встановлення базових налаштувань (параметри за замовчуванням + кастомні)
    this.config = {
```

```
    rootMargin: config.rootMargin || '150px',
  };


  // 2. Завантаження збереженого стану експерименту з localStorage
  // Якщо дані відсутні, виставляються значення за замовчуванням (дефолтний порівняльний тест)
  const savedState = this._loadState();
  this.state = {
    minify: savedState.minify ?? true, // true = сучасний WebP (стиснення), false = стандартний JPEG
    lazy: savedState.lazy ?? true, // true = відкладене завантаження увімкнено, false = вимкнено
    cache: savedState.cache ?? true // Керування кешуванням через Service Worker
  };


  // Посилання на екземпляр IntersectionObserver для керування Lazy Loading
  this._observer = null;
```

```
    // 3. Синхронізація роботи Service Worker із поточним станом прапорця cache
    if (this.state.cache) {
      this._registerServiceWorker();
    } else {
      this._unregisterServiceWorker();
    }
  }

  /**
   * Головний метод запуску бібліотеки.
   * Активує трансформацію DOM-структури та запускає механізм відкладеного завантаження.
   * @public
   */
  init() {
    // Етап 1: Перехоплення стандартних зображень та підготовка їх до оптимізації
    this._transformStandardImages();
```

```
    // Етап 2: Запуск логіки Lazy Loading
    this._initLazyLoading();

    // Візуалізація поточного стану бібліотеки в консолі розробника
    console.log('%c[OptimizationManager] Ініціалізовано успішно. Стан:', 'color: #00ff00; font-weight: bold;');
    console.table(this.state);
  }

  /**
   * Знаходить усі стандартні теги <img> на сторінці та ізолює їх від автоматичного завантаження браузером.
   * Переносить посилання з атрибута `src` у тимчасовий сховище-атрибут `data-om-src`.
   * @private
   */
  _transformStandardImages() {
    // Вибираємо тільки ті зображення, які ще не були оброблені нашою бібліотекою
    const images = document.querySelectorAll('img:not([data-om-src])');
```

```
    images.forEach(img => {
      const originalSrc = img.src;

      // Перевіряємо наявність шляху та виключаємо зображення у форматі Base64 Data-URL
      if (originalSrc && !originalSrc.startsWith('data:')) {
        // Зберігаємо оригінальний URL у безпечний дата-атрибут
        img.setAttribute('data-om-src', originalSrc);
        // Видаляємо стандартний src, щоб запобігти передчасному завантаженню важких ресурсів
        img.removeAttribute('src');
      }
    });
  }

  /**
   * Модифікує URL-адресу зображення, додаючи GET-параметри для динамічного стиснення на сервері (CDN).
   * @param {string} baseUrl - Оригінальне посилання на зображення.
   * @returns {string} Модифікований URL з параметрами оптимізації або оригінальний рядок у разі помилки.
```

```
   * @public
   */
  getImageUrl(baseUrl) {
    try {
      const url = new URL(baseUrl);

      // Встановлюємо фіксовану базову ширину для оптимізованої копії
      url.searchParams.set('w', '800');

      // Динамічне налаштування якості: 65% для WebP (оптимально), 95% для оригінального JPEG (максимальна якість)
      url.searchParams.set('q', this.state.minify ? '65' : '95');

      // Формування формату вихідного файлу залежно від стану прапорця minify
      if (this.state.minify) {
        url.searchParams.set('fm', 'webp'); // Запит на конвертацію у WebP
      } else {
        url.searchParams.set('fm', 'jpg');  // Запит на збереження стандартного JPEG
```

```
      }


      return url.toString();
    } catch (e) {
      // Якщо посилання відносне (локальне) або не є валідним URL, повертаємо його без змін
      return baseUrl;
    }
  }


  /**
   * Контролює процес відкладеного завантаження зображень за допомогою Intersection Observer API.
   * @private
   */
  _initLazyLoading() {
    // Якщо попередній спостерігач існує — скидаємо його для уникнення витоку пам'яті
    if (this._observer) this._observer.disconnect();
```

```
const imagesToLoad = document.querySelectorAll('img[data-om-src]');


// ФОЛБЕК: Якщо функцію Lazy Loading вимкнено користувачем, завантажуємо всі зображення миттєво
if (!this.state.lazy) {
  imagesToLoad.forEach(img => {
    const originalUrl = img.getAttribute('data-om-src');
    img.src = this.getImageUrl(originalUrl);
    img.removeAttribute('data-om-src');
  });
  return;
}


// Створення нового екземпляра спостерігача за появою елементів на екрані
this._observer = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    // Перевірка, чи потрапив елемент у зону видимості (+ rootMargin)
    if (entry.isIntersecting) {
      const img = entry.target;
```

```
        const originalUrl = img.getAttribute('data-om-src');


        if (originalUrl) {
          // Генеруємо оптимізований URL, вставляємо в src і запускаємо завантаження
          img.src = this.getImageUrl(originalUrl);
          img.removeAttribute('data-om-src');


          // Припиняємо спостереження за цим елементом, оскільки його вже завантажено
          this._observer.unobserve(img);
        }
      }
    });
  }, {
    rootMargin: this.config.rootMargin, // Завчасний радіус спрацьовування
    threshold: 0.01                     // Спрацьовує, як тільки з'являється хоча б 1% елемента
  });
```

```
    // Реєструємо кожну знайдену картинку в системі спостереження
    imagesToLoad.forEach(img => this._observer.observe(img));
  }

  // ——— ДОПОМІЖНІ СИСТЕМНІ МЕТОДИ ———————————————————————

  /**
   * Реєструє файл Service Worker для забезпечення кешування мережевих запитів.
   * @private
   */
  _registerServiceWorker() {
    if ('serviceWorker' in navigator) {
      navigator.serviceWorker.register('./sw.js')
        .then(() => console.log('[ОМ] Service Worker успішно зареєстровано.'))
        .catch(err => console.error('[ОМ] Помилка реєстрації SW:', err));
    }
  }
```

```
 /**
  * Знаходить та деактивує всі активні Service Workers бібліотеки.
  * @async
  * @private
  */
 async _unregisterServiceWorker() {
   if ('serviceWorker' in navigator) {
     const registrations = await navigator.serviceWorker.getRegistrations();
     for (let registration of registrations) {
       registration.unregister();
       console.log('[OM] Service Worker деактивовано для тесту.');
     }
   }
 }

 /**
  * Зчитує конфігураційне сховище з LocalStorage.
  * @returns {Object} Об'єкт стану або порожній об'єкт у разі помилки/відсутності даних.
```

```
 * @private
 */
_loadState() {
  try {
    return JSON.parse(localStorage.getItem('om_config') || '{}');
  } catch {
    return {};
  }
}

/**
 * Зберігає поточний стан прапорців оптимізації в LocalStorage.
 * @private
 */
_saveState() {
  localStorage.setItem('om_config', JSON.stringify(this.state));
}

/**
```

```
   * Перемикає стан конкретної опції (minify, lazy, cache) на протилежний,
   * зберігає зміни та перезавантажує сторінку для тестування.
   * @param {string} method - Ключ опції у стані класу (наприклад, 'minify').
   * @public
   * @example
   * // Виклик у консолі браузера:
   * window.optimizationManagerInstance.toggle('minify');
   */
  toggle(method) {
    if (method in this.state) {
      this.state[method] = !this.state[method]; // Зміна true -> false або навпаки
      this._saveState();                    // Фіксація стану
      window.location.reload();                 // Перезавантаження для миттєвого застосування змін
    }
  }
}
```

// Експорт класу у глобальну область видимості (window) для доступу через консоль або інші скрипти

window.OptimizationManager = OptimizationManager;

ДОДАТОК Б

```
/**
 * @file sw.js
 * @description Service Worker — Універсальний модуль динамічного кешування.
 * Перехоплює мережеві запити, реалізує стратегію Network First та забезпечує офлайн-доступ.
 * @author Володіна Альона, група ТІ-21
 */

// Унікальний ідентифікатор поточного кешу (версіонування)
const CACHE_NAME = 'om-dynamic-cache-v1';

/**
 * Подія активації Service Worker (`activate`).
 * Використовується для очищення застарілих версій кешу та швидкого перехоплення контролю над сторінками.
 */
self.addEventListener('activate', (event) => {
  event.waitUntil(
```

```
  // 1. Отримуємо назви всіх наявних сховищ кешу в браузері
  caches.keys().then((cacheNames) => {
    return Promise.all(
      cacheNames.map((cache) => {
        // 2. Якщо знайдений кеш застарів (його назва не збігається з поточною) — видаляємо його
        if (cache !== CACHE_NAME) {
          console.log('[SW] Видалення старого кешу:', cache);
          return caches.delete(cache);
        }
      })
    );
  })
);
  // 3. Активуємо SW на всіх відкритих вкладках сайту негайно, не чекаючи перезавантаження
  return self.clients.claim();
});


/**
```

```
 * Подія перехоплення мережевих запитів (`fetch`).
 * Реалізує стратегію "Network First" (Спочатку мережа, у разі невдачі — локальний кеш).
 */
self.addEventListener('fetch', (event) => {
  // ФІЛЬТР: Обробляємо тільки стандартні HTTP/HTTPS запити
  // Ігноруємо внутрішні протоколи (наприклад, розширення chrome-extension:// чи локальні file://)
  if (!event.request.url.startsWith('http')) return;

  event.respondWith(
    // ЕТАП 1: Намагаємося зробити реальний запит в інтернет за свіжими даними
    fetch(event.request)
      .then((response) => {
        // Перевірка валідності відповіді:
        // Статус 200 (успішно) та тип 'basic' (ресурси нашого домену) АБО запити до фото-хостингу Unsplash
        if (response && response.status === 200 && response.type === 'basic' || response.url.includes('unsplash')) {
```

```
        // ВАЖЛИВО: Клонуємо відповідь, оскільки потік даних (stream) можна прочитати лише один раз
        const responseToCache = response.clone();

        // Відкриваємо наше динамічне сховище та записуємо туди свіжу копію ресурсу
        caches.open(CACHE_NAME).then((cache) => {
          cache.put(event.request, responseToCache);
        });
      }
      // Повертаємо оригінальну відповідь у браузер
      return response;
    })
    .catch(() => {
      // ЕТАП 2: ФОЛБЕК (Спрацьовує, якщо fetch завершився помилкою — немає інтернету або впав сервер)
      console.log('[SW] Мережа недоступна. Завантаження з кешу:', event.request.url);

      // Шукаємо та повертаємо копію ресурсу, яка була закешована раніше
      return caches.match(event.request);
```

```
        })

    );

});
```