Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ДОСЛІДЖЕННЯ АЛГОРИТМІВ КОМПРЕСІЇ У ВЕБСЕРЕДОВИЩІ

Текстова частина до дипломної роботи
«F2 Інженерія програмного забезпечення»

Керівник дипломної роботи
старший викладач
Зважій Д. В.

_____
(підпис)

«____» _____ 2026 р.

Виконав студент
Синяк О. Я.

«____» _____ 2026 р.

Київ 2026

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри мультимедійних систем,
доцент, кандидат наук
_____
(підпис)
Жежерун Олександр Петрович
«____» _____ 2026 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на дипломну роботу

студенту Синяку О. Я. факультету інформатики 4 курсу
ТЕМА Дослідження алгоритмів компресії у вебсередовищі

Зміст ТЧ до дипломної роботи:

Індивідуальне завдання
Вступ
1 Теоретичний аналіз алгоритмів компресії у веб-середовищі
2 Проектування та розробка системи тестування
3 Результати експериментального дослідження
4 Аналіз результатів та практичні рекомендації
Висновки
Список літератури
Додатки

Дата видачі «____» _____ 2026 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Дослідження алгоритмів компресії у вебсередовищі

Календарний план виконання роботи:

№ п/п	Назва етапу роботи	Термін виконання	Примітка
1.	Отримання завдання на дипломну роботу.	15.10.2025	
2.	Попередній огляд літератури та технічних джерел за темою роботи.	31.10.2025	
3.	Формулювання мети, об’єкта, предмета та дослідницьких питань.	15.11.2025	
4.	Аналіз алгоритмів Gzip, Brotli та Zstandard і їх застосування у вебсередовищі.	30.11.2025	
5.	Проектування архітектури системи тестування.	15.12.2025	
6.	Розробка серверної частини та модулів компресії.	15.01.2026	
7.	Розробка клієнтської частини та інтерфейсу аналізу результатів.	31.01.2026	
8.	Реалізація браузерного тестування через Playwright.	15.02.2026	
9.	Проведення серверних і браузерних експериментів.	15.03.2026	
10.	Статистична обробка результатів та побудова графіків.	31.03.2026	
11.	Написання, редагування та остаточне оформлення текстової частини дипломної роботи.	15.05.2026	
12.	Захист дипломної роботи.	25–27.05.2026	

Студент _____

Керівник _____

«_____» _____

Зміст

Анотація

Дипломна робота присвячена порівняльному дослідженню алгоритмів компресії Gzip, Brotli та Zstandard у вебсередовищі. Розроблено автоматизовану систему тестування на базі Node.js, Angular та SQLite, що поєднує серверне профілювання компресії з браузерними вимірюваннями Web Vitals через Playwright. Експериментальна частина охоплює корпус із 53 реальних веб-файлів, повний аналіз рівнів компресії на репрезентативних ресурсах та real-app вимірювання для трьох production-збірок: Angular, React/Vite та Vue/Vite.

У роботі встановлено, що Brotli-6 забезпечує найкращий коефіцієнт стиснення для основних типів текстового веб-контенту, тоді як Zstd-1/3/5 формує найефективнішу швидку зону Парето для сценаріїв динамічного стиснення. Максимальні рівні Brotli-11 та Zstd-22 доцільні переважно для попереднього стиснення статичних ресурсів, де час компресії не впливає на користувацький запит. Браузерні експерименти показали, що HTTP-стиснення може суттєво зменшувати LCP у real-app сценаріях, особливо в Chromium та повільніших мережевих умовах. Для інтерпретації цього ефекту LCP розкладено на спостережувані канали TTFB, Download, Blocking та Render-residual без претензії на ізольоване вимірювання чистого часу декомпресії.

Експериментальне середовище обмежене одним Windows 11 host без конкурентного навантаження; узагальнення результатів на Linux-сервери та виробничі сценарії з високим RPS виходить за межі даного дослідження.

Сформульовано практичні рекомендації щодо вибору алгоритму залежно від типу контенту, сценарію генерації ресурсу, браузерної підтримки та CPU-бюджету. Окремо зафіксовано обмеження headless-build WebKit у Playwright на Windows щодо автоматизованого тестування `Content-Encoding:zstd`.

Ключові слова: компресія, Gzip, Brotli, Zstandard, Web Vitals, LCP, HTTP, Content-Encoding, Node.js, Angular, Playwright.

Вступ

Актуальність теми обумовлена зростанням обсягів веб-ресурсів та розвитком алгоритмів HTTP-компресії, зокрема Brotli та Zstandard. У сучасних веб-застосунках вибір алгоритму стиснення впливає не лише на розмір переданих файлів, а й на час відповіді сервера, навантаження на CPU, сумісність із браузерами та метрики користувацького досвіду, зокрема Largest Contentful Paint. Особливої актуальності тема набула після появи браузерної підтримки `Content-Encoding: zstd`, що створює практичну потребу в порівнянні Gzip, Brotli та Zstandard у реальних веб-сценаріях.

Мета роботи - порівняльне дослідження алгоритмів Gzip, Brotli та Zstandard у вебсередовищі та визначення практичних стратегій їх застосування для статичного і динамічного веб-контенту.

Об'єкт дослідження - процеси стиснення, передавання та декомпресії веб-ресурсів із використанням механізму HTTP `Content-Encoding`.

Предмет дослідження - характеристики алгоритмів Gzip, Brotli та Zstandard за коефіцієнтом стиснення, швидкістю компресії і декомпресії, масштабуванням відносно розміру файлу та впливом на метрики вебпродуктивності.

Для досягнення мети поставлено такі завдання:

- проаналізувати архітектурні особливості алгоритмів Gzip, Brotli та Zstandard;
- розробити систему автоматизованого серверного і браузерного тестування алгоритмів компресії;
- порівняти коефіцієнти стиснення алгоритмів для різних типів веб-контенту;
- дослідити компроміс між швидкістю та виграшем у розмірі через Парето-аналіз рівнів компресії;
- оцінити масштабування швидкості компресії та декомпресії залежно від розміру файлу;
- виміряти вплив HTTP-стиснення на LCP у real-app сценаріях та виконати декомпозицію цієї метрики на спостережувані канали;
- сформулювати практичні рекомендації щодо вибору алгоритму для різних сценаріїв використання.

Дослідження побудовано навколо чотирьох дослідницьких питань: RQ1 - наявність статистично значущої різниці у коефіцієнті стиснення між Gzip, Brotli та Zstandard за типами веб-контенту; RQ2 - компроміс “швидкість-стиснення” через Парето-аналіз; RQ3 - масштабування швидкості залежно від розміру файлу; RQ4 - вплив алгоритму стиснення на LCP у real-app сценаріях. Детальна постановка кожного питання та методика його перевірки наведені у Розділі 3.

Методи дослідження включають теоретичний аналіз технічної літератури та специфікацій, експериментальне вимірювання продуктивності, статистичну обробку результатів, Парето-аналіз, log-log регресію швидкості відносно розміру файлу та автоматизоване браузерне тестування через Playwright.

Практичне значення роботи полягає у розробці відтворюваної системи тестування та формуванні рекомендацій щодо використання Brotli, Gzip і Zstandard для статичних ресурсів, динамічних API-відповідей, real-app веб-застосунків та кросбраузерного production-впровадження.

РОЗДІЛ 1. Теоретичний аналіз алгоритмів компресії у веб-середовищі

1.1 Роль компресії в сучасній веб-архітектурі

Сучасний Інтернет характеризується безперервним зростанням обсягів даних, що передаються між серверами та клієнтськими пристроями. За даними Web Almanac 2025 від HTTP Archive, медіанний розмір домашньої веб-сторінки становив 2,86 МБ для настільних комп'ютерів та 2,56 МБ для мобільних пристроїв, що суттєво перевищує показники попереднього десятиліття [1]. При цьому значну частку цього обсягу складають текстові ресурси - HTML-документи, CSS-стилі, JavaScript-скрипти та JSON-дані, які за своєю природою мають високий ступінь надлишковості та добре піддаються стисненню.

Протокол HTTP, що є основою передачі даних у Всесвітній павутині, від самого початку передбачав механізми узгодження стиснення між клієнтом та сервером. Сучасна специфікація HTTP Semantics, описана в RFC 9110 [2], визначає заголовки `Accept-Encoding` та `Content-Encoding`, які дозволяють клієнту повідомити серверу про підтримувані алгоритми стиснення, а серверу - вказати, який алгоритм було застосовано до тіла відповіді. Цей механізм називається узгодженням вмісту (content negotiation) і працює прозоро для кінцевого користувача.

Процес узгодження стиснення відбувається наступним чином: браузер надсилає HTTP-запит із заголовком `Accept-Encoding:gzip,deflate,br,zstd`, що перелічує алгоритми, які він здатний декодувати. Сервер, отримавши цей заголовок, обирає один із підтримуваних алгоритмів, стискає тіло відповіді та повертає його із заголовком `Content-Encoding`, що вказує на застосований метод. Браузер, у свою чергу, автоматично розпаковує отримані дані перед їх обробкою та відображенням [3].

Вплив компресії на продуктивність веб-застосунків є багатоаспектним. По-перше, зменшення розміру переданих даних безпосередньо скорочує час завантаження сторінки, що є критичним фактором для користувацького досвіду. Дослідження Google свідчать, що збільшення часу завантаження мобільної сторінки з 1 до 3 секунд підвищує ймовірність відмови користувача на 32%, а при збільшенні до 5 секунд - на 90% [4]. По-друге, стиснення зменшує навантаження на мережеву інфраструктуру та знижує витрати на трафік, що особливо актуально для високонавантажених сервісів. По-третє, у протоколах HTTP/2 та HTTP/3 менші за розміром відповіді швидше

завершують передачу в межах мультиплексованого з'єднання і звільняють ресурси для інших потоків [5].

За статистикою W3Techs, станом на квітень 2026 року HTTP-компресію використовують понад 91% веб-сайтів. Серед сайтів, що застосовують компресію, Gzip залишається найпоширенішим алгоритмом із часткою близько 48,6%, Brotli використовується приблизно на 41,3%, а Zstandard - на 11,6% таких сайтів [6]. Алгоритм Zstandard, який отримав підтримку у частині браузерів порівняно нещодавно, наразі знаходиться на етапі активного впровадження та поступово розширює свою присутність.

Окрім прямого впливу на швидкість завантаження, компресія значно впливає на ключові метрики веб-продуктивності, визначені ініціативою Google Web Vitals [7]. Зменшення розміру переданих ресурсів може покращувати First Contentful Paint (FCP) - час до першого відображення контенту, та Largest Contentful Paint (LCP) - час до відображення найбільшого елемента на сторінці. Core Web Vitals входять до сигналів якості сторінки в екосистемі Google Search, хоча не замінюють релевантність і якість самого контенту [40]; це надає компресії додаткову практичну цінність.

Таким чином, компресія даних є невід'ємною складовою сучасної веб-архітектури, що впливає на продуктивність, економічність та конкурентоспроможність веб-ресурсів. Вибір оптимального алгоритму стиснення та його налаштувань потребує глибокого розуміння технічних характеристик наявних рішень, що обумовлює актуальність їх порівняльного дослідження.

1.2 Класифікація алгоритмів компресії

Алгоритми компресії даних можна класифікувати за кількома ключовими ознаками, розуміння яких є необхідним для аналізу їх застосовності у веб-середовищі.

Компресія без втрат та компресія з втратами. Фундаментальним є поділ на алгоритми без втрат (lossless compression) та з втратами (lossy compression). Алгоритми без втрат гарантують точне відновлення оригінальних даних після декомпресії, що є обов'язковою вимогою для текстових веб-ресурсів - HTML, CSS, JavaScript, JSON, XML тощо. Навіть мінімальна зміна у вихідному коді може призвести до некоректної роботи веб-застосунку, тому для стиснення цих типів контенту застосовуються виключно алгоритми без втрат [8]. Компресія з втратами використовується для мультимедійного контенту (зображення, аудіо, відео), де допустима певна деградація якості задля суттєвого зменшення розміру. У рамках даного дослідження

розглядаються лише алгоритми без втрат, оскільки саме вони застосовуються на рівні HTTP `Content-Encoding`.

Словникові методи. Словникові алгоритми стиснення базуються на пошуку повторюваних послідовностей у вхідних даних та заміні їх посиланнями на раніше зустрінуті входження. Родоначальниками цього підходу є алгоритми сімейства Лемпеля-Зіва (LZ). Алгоритм LZ77, запропонований Абрагамом Лемпелем та Якобом Зівом у 1977 році [9], використовує так зване ковзне вікно (sliding window), у межах якого здійснюється пошук збігів. Коли алгоритм знаходить підрядок, що повторює раніше оброблену послідовність, він замінює його компактним посиланням, що складається з відстані до збігу (offset) та довжини збігу (length). LZ77 є основою для більшості сучасних алгоритмів веб-компресії - він лежить в основі DEFLATE (Gzip), є частиною Brotli та Zstandard.

Алгоритм LZ78, запропонований тими ж авторами у 1978 році [10], та його модифікація LZW (Lempel-Ziv-Welch) використовують інший підхід - вони динамічно будують словник фраз під час обробки вхідних даних. LZW знайшов застосування у форматах GIF та ранніх реалізаціях утиліти compress, однак у сучасній веб-компресії він практично не використовується через нижчу ефективність порівняно з методами на основі LZ77.

Ентропійне кодування. Після словникової фази стиснення зазвичай застосовується ентропійне кодування, яке присвоює коротші коди символам, що зустрічаються частіше, і довші коди - символам, що зустрічаються рідше. Класичним прикладом є кодування Хаффмана [11], яке будує оптимальний префіксний код на основі частот символів. Воно використовується у Gzip як частина алгоритму DEFLATE.

Більш сучасним підходом є арифметичне кодування та його практичні реалізації, зокрема ANS (Asymmetric Numeral Systems), запропоноване Яреком Дудою у 2009 році [12]. Метод FSE (Finite State Entropy), що є табличною реалізацією ANS, забезпечує ефективність, близьку до ентропійної межі, при значно вищій швидкості порівняно з арифметичним кодуванням. FSE використовується в алгоритмі Zstandard.

Контекстне моделювання. Окремим напрямком є контекстне моделювання, яке враховує залежності між послідовними символами для більш точного передбачення ймовірностей. Алгоритм Brotli використовує контекстне моделювання другого порядку, що дозволяє враховувати два попередні символи при оцінці ймовірності наступного [13]. Це особливо ефективно для природомовного тексту та структурованих даних, де послідовності символів мають виражені статистичні закономірності.

Статичні словники. Крім динамічних методів, деякі алгоритми використовують попередньо побудовані (статичні) словники, що містять типові фрагменти цільового типу даних. Brotli включає вбудований статичний словник розміром 120 КБ, що містить найпоширеніші слова та фрази з веб-контенту на різних мовах, а також типові фрагменти HTML, CSS та JavaScript [13]. Це дозволяє ефективно стискати навіть невеликі файли, де динамічний словник не встигає накопичити достатньо інформації.

1.3 Технічний аналіз алгоритмів

1.3.1 Gzip: DEFLATE, LZ77, кодування Хаффмана

Gzip є одним із найстаріших та найпоширеніших алгоритмів стиснення у веб-середовищі. Формат Gzip, специфікований у RFC 1952 [14], є контейнером для стиснених даних, що використовує алгоритм DEFLATE, описаний у RFC 1951 [15]. DEFLATE, розроблений Філом Кацем для програми PKZIP у 1993 році, поєднує два фундаментальні методи стиснення: алгоритм LZ77 для усунення повторюваних послідовностей та кодування Хаффмана для ентропійного кодування результату.

Фаза LZ77. На першому етапі DEFLATE застосовує модифікований алгоритм LZ77 з ковзним вікном розміром до 32 КБ. Алгоритм послідовно обробляє вхідний потік байтів, для кожної позиції шукаючи найдовший збіг із раніше обробленими даними у межах вікна. Якщо збіг знайдено і його довжина становить не менше 3 байтів, він замінюється парою (відстань, довжина). Якщо збіг не знайдено або він занадто короткий, символ залишається некодованим (literal). Результатом цієї фази є послідовність літералів та посилань на збіги [8].

Ефективність фази LZ77 безпосередньо залежить від рівня компресії, який визначає глибину пошуку збігів. На низьких рівнях (1–3) використовуються швидкі, але менш ретельні стратегії пошуку, такі як хеш-ланцюги мінімальної довжини. На високих рівнях (7–9) виконується більш вичерпний пошук, що дозволяє знаходити довші та більш віддалені збіги, але потребує значно більше процесорного часу.

Фаза кодування Хаффмана. Після фази LZ77 результуючі символи (літерали та коди довжин/відстаней) кодуються за допомогою алгоритму Хаффмана. Кодування Хаффмана, запропоноване Девідом Хаффманом у 1952 році [11], будує бінарне дерево, де кожному символу відповідає унікальний префіксний код. Символи з вищою частотою отримують коротші коди, а з нижчою - довші, що мінімізує середню довжину закодованого повідомлення.

DEFLATE підтримує два режими кодування Хаффмана: зі статичними табли-

цями, визначеними у специфікації, та з динамічними таблицями, що обчислюються для кожного блоку даних окремо. Динамічні таблиці забезпечують кращу компресію, оскільки адаптовані до конкретних даних, але вимагають включення самих таблиць у стиснений потік, що додає накладні витрати.

Структура стисненого файлу. Gzip-файл складається з 10-байтного заголовка, що містить магічне число (1F 8B), метод стиснення (08 для DEFLATE), прапорці та мітку часу, за яким слідують стиснені блоки DEFLATE та 8-байтний завершальний блок із контрольною сумою CRC-32 та розміром оригінальних даних [14]. Кожен блок DEFLATE може бути нестисненим, стисненим зі статичним кодуванням Хаффмана або стисненим з динамічним кодуванням. Біт BFINAL у заголовку блоку вказує, чи є він останнім.

Gzip підтримує рівні компресії від 1 (найшвидший) до 9 (максимальне стиснення), де рівень 6 є типовим за замовчуванням. Різниця між рівнями полягає головним чином у стратегії пошуку збігів на фазі LZ77: на рівні 1 використовується мінімальний пошук, на рівні 9 - максимально ретельний пошук з найдовшими хеш-ланцюгами [8].

1.3.2 Brotli: словниковий метод, контекстне моделювання

Brotli - алгоритм стиснення без втрат, розроблений компанією Google та специфікований у RFC 7932 [13] у 2016 році. Назва походить від швейцарського хлібного виробу Brötli. Алгоритм спроектований спеціально для стиснення веб-контенту та поєднує кілька передових технік. У технічному звіті Google 2015 року Brotli показав на 20–26% кращий коефіцієнт стиснення порівняно із Zopfli; порівняно з традиційним Deflate/Gzip виграш залежав від типу файлів і був нижчим [16].

Покращений словниковий метод. Подібно до DEFLATE, Brotli використовує алгоритм на основі LZ77, однак із суттєво розширеними параметрами. Максимальний розмір ковзного вікна у Brotli становить 16 МБ (порівняно з 32 КБ у DEFLATE), що дозволяє знаходити збіги на значно більших відстанях. Це особливо ефективно для великих файлів із повторюваними структурами, таких як масивні JavaScript-бандли або об'ємні CSS-файли [13]. Максимальна довжина збігу також збільшена - до 16 МБ замість 258 байтів у DEFLATE.

Статичний словник. Ключовою інновацією Brotli є вбудований статичний словник розміром 120 КБ, що містить 13 504 слів та фраз шістьма мовами (англійська, іспанська, китайська, хінді, російська та арабська), а також поширені фрагменти HTML, CSS, JavaScript та інших веб-форматів [13]. Цей словник використовується як додаткове джерело збігів: якщо фрагмент вхідних даних збігається зі словниковим

записом, він замінюється компактним посиланням на словникову позицію та трансформацію (наприклад, зміна регістру, додавання пробілів або пунктуації). Передбачено 121 трансформацію, що значно розширюють ефективний обсяг словника [13].

Статичний словник є особливо корисним для стиснення невеликих файлів (менше 1 КБ), де динамічний словник не встигає побудувати достатню модель даних. Для типового HTML-документу розміром 500 байтів Brotli може забезпечити помітно кращу компресію порівняно з Gzip саме завдяки використанню статичного словника.

Контекстне моделювання. Brotli застосовує контекстне моделювання другого порядку для покращення передбачення ймовірностей літералів. Алгоритм аналізує два попередні байти для визначення контексту, в якому з'являється поточний байт, і використовує окрему модель ймовірностей для кожного контексту [13]. Це дозволяє точніше моделювати статистичні залежності у структурованих даних. Наприклад, після послідовності `<div cl` ймовірність символу `a` (початок `class`) значно вища, ніж у довільному контексті.

Brotli використовує 256 контекстних класів для літералів та 4 контекстних класи для відстаней. Для кожного метаблоку даних обирається оптимальна конфігурація контекстних моделей, що адаптується до характеру оброблюваного контенту [13].

Ентропійне кодування. Для фінальної фази кодування Brotli використовує комбінацію кодування Хаффмана з префіксними кодами та прямого кодування для великих значень відстаней. На відміну від DEFLATE, який обмежений таблицями Хаффмана з максимальною довжиною коду 15 біт, Brotli підтримує коди довжиною до 15 біт, але з більш гнучкою системою формування кодових таблиць.

Brotli підтримує 12 рівнів компресії (від 0 до 11). Рівні 0–1 забезпечують швидке стиснення, порівнянне з Gzip за швидкістю, але з дещо кращим коефіцієнтом. Рівні 5–6 є типовими для динамічного стиснення на серверах. Рівні 10–11 забезпечують максимальне стиснення, але потребують значних обчислювальних ресурсів і зазвичай використовуються лише для статичного контенту, що стискається заздалегідь [16].

1.3.3 Zstandard: FSE, адаптивні словники

Zstandard (Zstd) - алгоритм стиснення без втрат, розроблений Янном Коле (Yann Collet) у компанії Facebook (нині Meta) та специфікований у RFC 8878 [17]. Zstandard проектувався з метою досягнення оптимального балансу між коефіцієнтом стиснення та швидкістю, прагнучи поєднати ефективність стиснення на рівні DEFLATE з швидкістю, що наближається до алгоритмів сімейства LZ4 [18].

Архітектура алгоритму. Zstandard використовує триетапну архітектуру сти-

снення. На першому етапі застосовується модифікований алгоритм LZ77 з ковзним вікном розміром до 128 МБ для пошуку повторюваних послідовностей. На другому етапі результуючі послідовності (sequences), що складаються з трійок (літерали, відстань, довжина збігу), кодуються за допомогою FSE-кодування. На третьому етапі літерали додатково стискаються методом Хаффмана [17].

Finite State Entropy (FSE). Ключовою технічною інновацією Zstandard є використання кодувальника FSE (Finite State Entropy), розробленого також Янном Коле на основі теорії ANS (Asymmetric Numeral Systems) Ярека Дуди [12]. FSE є табличною реалізацією tANS (tabled ANS), що забезпечує компресію, близьку до теоретичної ентропійної межі Шеннона, але при цьому працює зі швидкістю, значно вищою за арифметичне кодування [18].

Принцип роботи FSE полягає у використанні скінченного автомата зі станами, де кожен стан відповідає певній фазі кодування символу. Перехід між станами є детерміністичним та залежить від поточного символу і поточного стану. Кодування та декодування виконуються за допомогою табличних пошуків без розгалужень, що забезпечує передбачуваний час виконання та ефективне використання конвеєра процесора [18]. На практиці FSE досягає пропускної спроможності декодування 400–500 МБ/с на одному ядрі сучасного процесора.

Адаптивні рівні компресії. Zstandard підтримує 22 рівні компресії (від 1 до 22), а також спеціальні «від'ємні» рівні для надшвидкого стиснення (від -1 до -131072). Ця широка шкала забезпечує значну гнучкість: рівень 1 забезпечує стиснення зі швидкістю понад 500 МБ/с із помірним коефіцієнтом, тоді як рівень 22 досягає максимального коефіцієнта стиснення, порівнянного з LZMA, але з набагато швидшою декомпресією [17]. Рівень 3, що є типовим за замовчуванням, забезпечує коефіцієнт стиснення, порівнянний із Gzip рівня 6, при значно вищій швидкості обробки.

Кожен рівень компресії визначає набір параметрів: розмір вікна, мінімальну та максимальну довжину збігу, глибину пошуку в хеш-таблиці та стратегію пошуку збігів (fast, dfast, greedy, lazy, btlazy2, btopt, btultra) [17]. Стратегії від fast до btultra поступово збільшують ретельність пошуку та обчислювальну складність.

Словники та тренування. Zstandard підтримує механізм зовнішніх словників, що дозволяє побудувати спеціалізований словник на основі репрезентативного набору даних. Цей механізм є особливо ефективним для стиснення великої кількості невеликих однотипних повідомлень, наприклад JSON-відповідей API [17]. Словник

тренується на зразках цільових даних за допомогою утиліти `zstd -train` і може бути спільним для кодувальника та декодувальника. Для веб-контенту тренований словник може забезпечити на 30–50% краще стиснення порівняно з режимом без словника для файлів розміром менше 4 КБ [18].

Особливості декомпресії. Декомпресія Zstandard спроектована як асиметрична операція - вона значно швидша за компресію і не залежить від рівня, з яким дані були стиснені. Швидкість декомпресії Zstandard становить приблизно 1200–1600 МБ/с на одному ядрі процесора, що перевищує показники як Gzip ($\sim$350 МБ/с), так і Brotli ($\sim$400 МБ/с) [18]. Ця характеристика є критично важливою для веб-середовища, де декомпресія виконується на клієнтському пристрої, ресурси якого можуть бути обмеженими.

1.4 Підтримка у веб-технологіях

Практична цінність алгоритму стиснення у веб-середовищі безпосередньо залежить від його підтримки на всіх рівнях технологічного стеку: у браузерах, на веб-серверах та проміжних проксі.

Браузерна підтримка. Підтримка алгоритмів стиснення у браузерах реалізована на рівні мережевого стеку і є прозорою для JavaScript-коду та DOM API.

Gzip підтримується усіма веб-браузерами з кінця 1990-х років. Фактично, це найбільш універсально підтримуваний алгоритм стиснення в Інтернеті, доступний у 100% сучасних браузерів [6].

Brotli отримав підтримку у Google Chrome з версії 50 (квітень 2016), Mozilla Firefox з версії 44 (січень 2016), Microsoft Edge з версії 15 (квітень 2017) та Safari з версії 11 (вересень 2017). Станом на квітень 2026 року Brotli підтримується приблизно у 95,65% браузерів за глобальною статистикою використання [19]. Важливо зазначити, що більшість браузерів підтримують Brotli лише через захищені з'єднання HTTPS, що є свідомим рішенням для запобігання проблемам із проміжними проксі, які можуть некоректно обробляти невідомі їм алгоритми стиснення.

Zstandard отримав підтримку у Google Chrome з версії 123 (березень 2024) під ідентифікатором `zstd` у заголовку `Content-Encoding` [20]. Firefox додав підтримку у версії 126 (травень 2024). Натомість у Safari підтримка Zstandard довгий час була відсутня; за даними Can I Use, настільний Safari до версії 18.7 не підтримує `Content-Encoding:zstd`, а новіші версії мають лише часткову підтримку [19]. Попри відносно недавнє впровадження, Zstandard уже підтримується значною часткою браузерів

за глобальною статистикою, насамперед завдяки Chromium- та Firefox-базованим клієнтам [19].

Серверна інтеграція. На серверному рівні стиснення може бути реалізоване двома способами: динамічне стиснення (on-the-fly) при кожному запиті та статичне стиснення (pre-compression) заздалегідь підготовлених файлів.

Nginx, один із найпоширеніших веб-серверів, підтримує Gzip через вбудований модуль `ngx_http_gzip_module`, Brotli - через сторонній модуль `ngx_brotli`, а Zstandard - через модуль `ngx_http_zstd_filter_module` [21]. Apache HTTP Server підтримує Gzip та DEFLATE через модуль `mod_deflate`, а Brotli - через модуль `mod_brotli`, доступний з версії 2.4.26 [22].

У середовищі Node.js, яке використовується у даному дослідженні, модуль `zlib` зі стандартної бібліотеки надає підтримку Gzip та Brotli; у новіших версіях Node.js також з'явилась експериментальна підтримка Zstandard [23]. У реалізації цієї роботи Zstandard підключено через npm-пакет `@mongodb-js/zstd` [36], а альтернативні JavaScript-реалізації, зокрема `fzstd`, можуть застосовуватись у сценаріях без нативних біндингів. Express.js підтримує стиснення через middleware `compression`, який обирає підтримуване кодування на основі `Accept-Encoding`; типовий набір цього middleware охоплює `gzip`, `deflate` та `br`, але не `zstd` [27].

Мережі доставки контенту (CDN). CDN-провайдери відіграють ключову роль у розповсюдженні стисненого контенту. Cloudflare підтримує Gzip і Brotli для текстових ресурсів та надає Zstandard через механізми налаштування compression rules [24]. AWS CloudFront підтримує Gzip та Brotli, дозволяючи налаштовувати стиснення на рівні кеш-поведінки [25]. Для будь-якого CDN ключовою вимогою залишається коректне кешування окремих варіантів відповіді відповідно до `Accept-Encoding`.

Механізм узгодження `Content-Encoding`**.** Взаємодія між клієнтом та сервером щодо стиснення регулюється протоколом HTTP через механізм узгодження вмісту. Клієнт надсилає заголовок:

```
Accept-Encoding:gzip,deflate,br,zstd
```

Сервер обирає алгоритм на основі власних пріоритетів та можливостей і повертає:

```
Content-Encoding:br
Vary:Accept-Encoding
```

Заголовок `Vary: Accept-Encoding` є критично важливим для коректної роботи кешувальних проксі - він вказує, що відповідь залежить від значення `Accept-Encoding` і різні варіанти мають кешуватися окремо [2]. Без цього заголовка проксі-сервер може повернути стиснену Brotli відповідь клієнту, який підтримує лише Gzip, що призведе до помилки.

1.5 Метрики оцінки ефективності

Для об'єктивного порівняння алгоритмів стиснення у веб-середовищі необхідна система метрик, яка охоплює як характеристики самого стиснення, так і його вплив на кінцеву продуктивність веб-застосунку.

Коефіцієнт стиснення (Compression Ratio). Коефіцієнт стиснення є основною метрикою ефективності алгоритму і може бути виражений кількома способами. У даному дослідженні використовується відсоток зменшення розміру:

$$CR = \frac{S_{\text{original}} - S_{\text{compressed}}}{S_{\text{original}}} \times 100\%$$

де S_{original} - розмір оригінального файлу, $S_{\text{compressed}}$ - розмір стисненого файлу. Вищий коефіцієнт стиснення означає більшу ефективність алгоритму. Для типового веб-контенту коефіцієнти стиснення зазвичай знаходяться в діапазоні 60–85% для HTML, 70–90% для CSS та JavaScript, і 30–50% для вже частково оптимізованих файлів [16].

Швидкість стиснення та декомпресії. Швидкість обробки даних вимірюється у мегабайтах за секунду (МБ/с) і розраховується як відношення розміру оригінальних даних до часу виконання операції:

$$V_{\text{compression}} = \frac{S_{\text{original}}}{T_{\text{compression}}} \tag{1.1}$$

$$V_{\text{decompression}} = \frac{S_{\text{original}}}{T_{\text{decompression}}} \tag{1.2}$$

У веб-контексті швидкість декомпресії є критичнішою, ніж швидкість компресії, оскільки декомпресія виконується на кожному клієнтському пристрої при кожному завантаженні ресурсу, тоді як компресія зазвичай виконується одноразово на сервері або під час збірки [8]. Для статичного контенту швидкість компресії є менш важливою, оскільки файли стискаються заздалегідь. Для динамічного контенту (API-відповіді, серверний рендеринг) швидкість компресії стає критичним фактором, що впливає на

час відповіді сервера (Time to First Byte, TTFB).

Використання обчислювальних ресурсів. Стиснення та декомпресія потребують процесорного часу та оперативної пам'яті. Споживання CPU вимірюється як загальний процесорний час (user time + system time), що дозволяє оцінити обчислювальну складність алгоритму. Споживання пам'яті включає розмір купи (heap), резидентної пам'яті (RSS) та зовнішньої пам'яті (external), що відповідає нативним буферам, виділеним за межами JavaScript-купи [23]. Для серверного застосування ці метрики визначають максимальну кількість одночасних операцій стиснення, які сервер здатний обробити без деградації продуктивності.

Метрики веб-продуктивності (Web Vitals). Для оцінки практичного впливу стиснення на користувацький досвід використовуються метрики Web Vitals, визначені Google [7]. До поточного набору Core Web Vitals належать LCP, INP та CLS, тоді як FCP і TTFB є допоміжними метриками завантаження:

- **Largest Contentful Paint (LCP)** - час від початку навігації до відображення найбільшого видимого елемента контенту. Значення до 2,5 секунди вважається добрим. Стиснення безпосередньо впливає на LCP, оскільки зменшує час передачі ресурсів, необхідних для рендерингу.

- **First Contentful Paint (FCP)** - час до першого відображення будь-якого текстового або графічного контенту. Значення до 1,8 секунди вважається добрим. FCP залежить від розміру критичних ресурсів (HTML, CSS), які блокують рендеринг.

- **Cumulative Layout Shift (CLS)** - кумулятивна оцінка візуальної стабільності сторінки. Значення до 0,1 вважається добрим. Хоча CLS менш безпосередньо пов'язаний зі стисненням, швидше завантаження ресурсів може зменшити зсуви макету, спричинені пізнім завантаженням стилів.

- **Interaction to Next Paint (INP)** - час від взаємодії користувача до наступного оновлення екрану. Значення до 200 мілісекунд вважається добрим. INP може залежати від швидкості декомпресії JavaScript-файлів, особливо на слабких пристроях. У межах цієї роботи INP не вимірюється, оскільки використаний real-app протокол фіксує лише фазу завантаження сторінки без подальших користувацьких взаємодій.

- **Time to First Byte (TTFB)** - час від відправки запиту до отримання першого

байта відповіді. Значення до 800 мілісекунд вважається добрим. TTFB безпосередньо залежить від часу стиснення відповіді на сервері при динамічному стисненні.

Мережеві метрики. Для комплексної оцінки впливу стиснення враховуються також мережеві характеристики: час DNS-резолюції, час встановлення TCP-з'єднання, час відправки запиту, час отримання відповіді та загальний час завантаження ресурсу. Ці метрики дозволяють оцінити ефективність стиснення в різних мережевих умовах - від швидких дротових з'єднань до повільних мобільних мереж 3G [26].

Поєднання алгоритмічних метрик (коефіцієнт стиснення, швидкість) з метриками веб-продуктивності (Web Vitals) та мережевими характеристиками дозволяє комплексно оцінити практичну ефективність алгоритмів стиснення у реальних умовах експлуатації, що є ключовим завданням даного дослідження.

РОЗДІЛ 2. Проектування та розробка системи тестування

2.1 Формування тестового набору даних

Об'єктивність порівняльного дослідження алгоритмів стиснення безпосередньо залежить від репрезентативності тестового набору даних. У рамках даної роботи було сформовано набір, що охоплює основні типи текстового та графічного веб-контенту, які передаються від сервера до клієнта у процесі завантаження типового веб-застосунку.

Типи веб-контенту. Тестовий набір включає сім категорій файлів, що відповідають найпоширенішим типам ресурсів у веб-середовищі:

1. **HTML-документи** - структуровані гіпертекстові документи, що містять розмітку сторінки, вкладені текстові блоки та атрибути елементів. HTML характеризується високим ступенем надлишковості через повторювані теги та атрибути, що робить його добре піддатним стисненню.

2. **CSS-стилі** - каскадні таблиці стилів, що описують візуальне оформлення веб-сторінки. CSS містить повторювані селектори, властивості та значення, а також значну кількість пробільних символів для форматування, що забезпечує високу ефективність стиснення.

3. **JavaScript-файли** - програмний код клієнтської частини веб-застосунку. JavaScript-бандли часто досягають значних розмірів (сотні кілобайтів або мегабайти) і містять як мініфікований код, так і рядкові константи, що створює неоднорідну структуру для стиснення.

4. **JSON-дані** - формат обміну даними, широко використовуваний у REST API. JSON характеризується регулярною структурою з повторюваними ключами та обмеженим алфавітом (лапки, двокрапки, фігурні та квадратні дужки), що забезпечує ефективне словникове стиснення.

5. **SVG-зображення** - масштабовані векторні графічні файли на основі XML. SVG містять числові координати, повторювані XML-теги та атрибути, що робить їх подібними за характеристиками стиснення до HTML.

6. **PNG-зображення** - растрові зображення зі стисненням без втрат. PNG-файли вже містять внутрішню DEFLATE-компресію, тому додаткове HTTP-стиснення є значно менш ефективним порівняно з текстовими форматами.

7. **JPEG-зображення** - растрові зображення зі стисненням з втратами. JPEG використовує DCT-перетворення та ентропійне кодування, що залишає мінімальну надлишковість для додаткового стиснення.

Принципи відбору зразків. При формуванні тестового набору було дотримано кількох принципів. По-перше, файли представляють реальний веб-контент, а не синтетично згенеровані дані, що забезпечує практичну релевантність результатів. По-друге, включено як текстові формати (HTML, CSS, JavaScript, JSON, SVG), де очікується висока ефективність стиснення, так і бінарні формати (PNG, JPEG), де стиснення має бути мінімально ефективним. Це дозволяє дослідити поведінку алгоритмів на різних типах даних та визначити межі їх застосовності.

Управління тестовими файлами. Розроблена система передбачає два джерела тестових файлів: вбудовані файли за замовчуванням, що постачаються разом із системою та зберігаються в директорії `test-files`, та файли, завантажені користувачем через інтерфейс. Кожен файл характеризується набором метаданих: унікальне ім'я, тип (розширення), розмір у байтах, шлях на диску, статус активності та ознака походження (вбудований чи завантажений). Файли можуть бути активовані або деактивовані для включення чи виключення з бенчмарків без видалення з системи.

2.2 Архітектура системи тестування

Для проведення порівняльного дослідження алгоритмів стиснення було розроблено веб-застосунок із клієнт-серверною архітектурою, що складається з серверної частини (бекенд) на базі Node.js, клієнтської частини (фронтенд) на базі Angular та бази даних SQLite. Система забезпечує автоматизоване виконання бенчмарків, збір метрик продуктивності, зберігання результатів та їх візуалізацію.

2.2.1 Загальна архітектура

Архітектура системи побудована за трирівневою моделлю: рівень представлення (фронтенд), рівень бізнес-логіки (бекенд) та рівень даних (база даних). Взаємодія між рівнями здійснюється через два канали: REST API для синхронних запитів (отримання результатів, управління файлами, налаштування) та WebSocket для асин-

хронних повідомлень у реальному часі (прогрес виконання бенчмарку, сповіщення про завершення).

Бекенд запускає два HTTP-сервери: основний API-сервер на порті 3001, що обслуговує REST-запити від фронтенду та WebSocket-з'єднання, та допоміжний сервер стиснення на порті 8080, що використовується виключно під час браузерного тестування для роздачі стиснених файлів через HTTP з відповідними заголовками `Content-Encoding`.

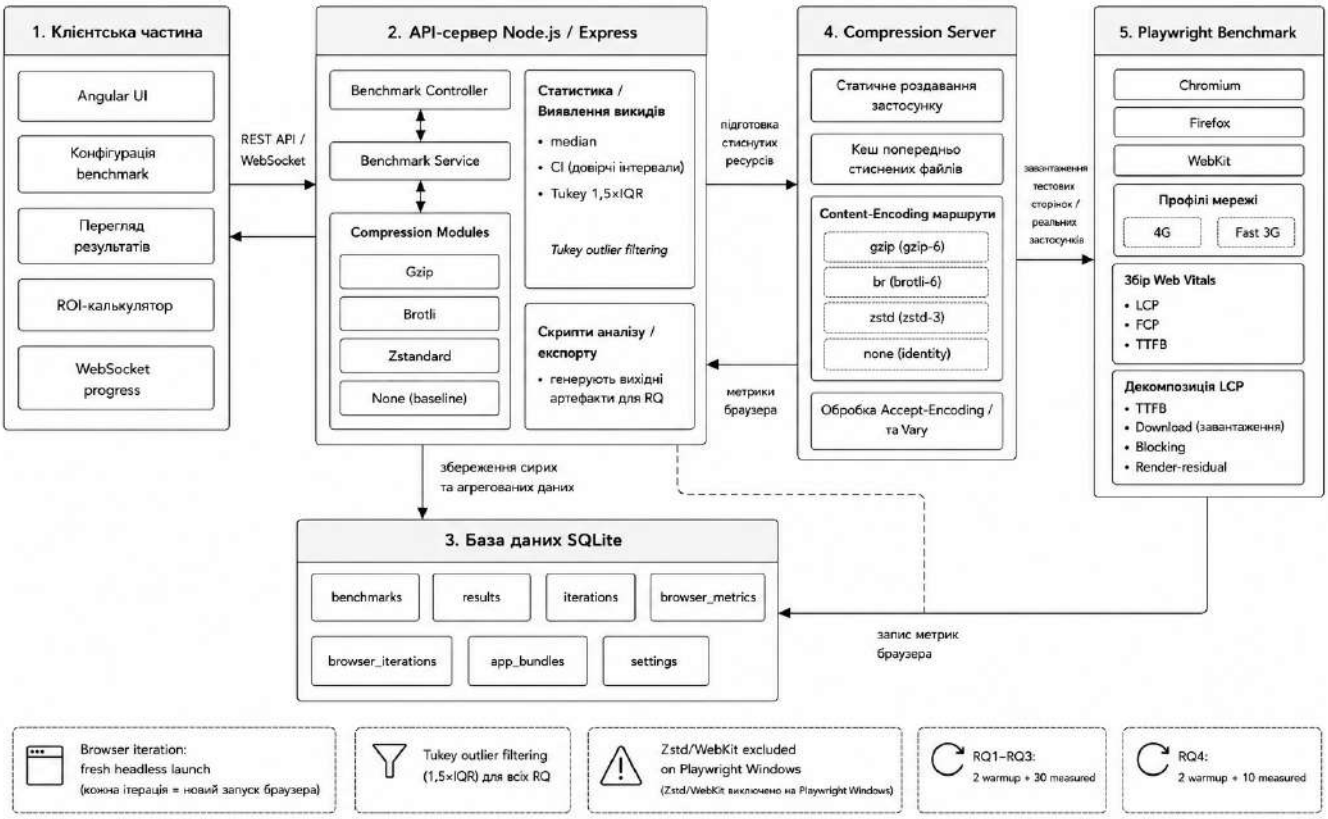

Рисунок 2.1 - Архітектура системи тестування алгоритмів компресії

2.2.2 Серверна частина (бекенд)

Серверна частина реалізована на платформі Node.js з використанням фреймворку Express.js [27] та побудована як шарова архітектура з рівнями маршрутів і контролерів, сервісів та репозиторіїв для абстракції доступу до даних.

Рівень маршрутизації та контролерів. Система визначає основні групи REST-маршрутів, кожна з яких обслуговує окрему функціональну область: управління бенчмарками (`/api/benchmark`, а також сумісний alias `/api/benchmarks`), результати стиснення (`/api/results`), управління файлами (`/api/files`), налаштування (`/api/settings`), експорт даних (`/api/export`), статистика та аналітика (`/api/statistics`) та production-збірки застосунків для real-app тестування

(/api/apps). Контролери відповідають за валідацію вхідних даних, делегування обробки відповідним сервісам та формування HTTP-відповідей. Загалом система надає понад 70 API-ендпоінтів.

Рівень сервісів. Бізнес-логіка інкапсульована в сервісних класах. Ключовим є BenchmarkService, що оркеструє процес виконання бенчмарку: генерує унікальний ідентифікатор, розраховує загальну кількість тестів, виконує серверні вимірювання з warmup-ітераціями, зберігає сирі та агреговані результати, запускає real-app браузерні тести та транслює прогрес через WebSocket. ExportService генерує звіти у форматах CSV та JSON. StatisticsService забезпечує агрегацію даних для аналітичних дашбордів. Додатково в системі реалізовано допоміжний модуль оцінки економії CDN-трафіку, результати застосування якого виходять за межі основних дослідницьких питань і у розділі 3 не наводяться.

Рівень репозиторіїв. Доступ до бази даних абстрагований через репозиторії, що реалізують шаблон Repository Pattern [28]. Основні сутності обслуговуються окремими репозиторіями: BenchmarkRepository, ResultsRepository, FilesRepository, SettingsRepository, BrowserMetricsRepository, IterationsRepository, BrowserIterationsRepository та AppBundlesRepository. Репозиторії інкапсулюють SQL-запити та надають об'єктно-орієнтований інтерфейс для операцій CRUD, агрегації, фільтрації та пошуку. Пакетні операції виконуються в транзакціях для забезпечення атомарності.

WebSocket-сервіс. Для передачі оновлень у реальному часі використовується бібліотека Socket.io [29]. Сервіс реалізує модель підписки на кімнати (rooms): клієнт підписується на конкретний бенчмарк через подію subscribe:benchmark, після чого отримує всі повідомлення про прогрес, логи та результати цього бенчмарку. Сервіс відстежує підключених клієнтів та їх підписки через структуру Map<clientId, {socket, connectedAt, subscriptions}>.

2.2.3 База даних

Для зберігання даних використовується SQLite - вбудована реляційна СУБД, що не потребує окремого серверного процесу [30]. Вибір SQLite обумовлений кількома факторами: простота розгортання (один файл бази даних), відсутність потреби у конфігурації серверної інфраструктури, достатня продуктивність для дослідницького застосунку та вбудована підтримка транзакцій. Взаємодія з SQLite здійснюється через бібліотеку better-sqlite3, що надає синхронний API та забезпечує вищу продуктивність порівняно з асинхронними альтернативами завдяки відсутності

накладних витрат на колбеки [31].

Під час ініціалізації підключення увімкнено підтримку зовнішніх ключів та режим WAL (Write-Ahead Logging), що дозволяє виконувати читання під час запису та зменшує блокування між операціями збереження результатів і запитами інтерфейсу.

Схема бази даних включає вісім основних таблиць:

Таблиця `benchmarks` зберігає метадані кожного запуску бенчмарку: унікальний ідентифікатор, опис, конфігурацію у форматі JSON, статус виконання (`pending`, `running`, `completed`, `failed`, `stopped`), лічильники прогресу, кількість файлів і алгоритмів, часові мітки початку та завершення, тривалість, повідомлення про помилку та зведений результат.

Таблиця `results` містить агрегований результат для кожної комбінації "файл–алгоритм–рівень". У ній зберігаються назва і тип файлу, оригінальний та стиснений розміри, коефіцієнт стиснення, медіанні часи компресії та декомпресії, швидкості обробки у МБ/с, медіанні показники CPU і пам'яті, прапорець верифікації цілісності, кількість виміряних і warmup-ітерацій, кількість викидів та описова статистика часу. Для оптимізації запитів створено індекси за бенчмарком, алгоритмом, файлом та складений індекс (`benchmarkId, algorithm, fileName`).

Таблиця `iterations` зберігає сирі серверні вимірювання для кожного рядка `results`: номер ітерації, час компресії та декомпресії, стиснений розмір, CPU, пам'ять, ознаку викиду та результат перевірки цілісності. Це дозволяє повторно аналізувати дані без втрати первинних вимірювань.

Таблиця `browser_metrics` зберігає агреговані результати браузерного тестування. Для кожної комбінації застосунок/файл, алгоритм, рівень, браузер і мережевий профіль у ній фіксуються LCP, FCP, CLS, INP, TTFB, таймінги навігації, характеристики ресурсу або сукупності ресурсів, шлях до скріншоту, режим тестування (`singleFile` або `realApp`), посилання на app bundle, кількість ітерацій, кількість викидів, статистика Web Vitals та канали декомпозиції LCP: TTFB, Download, Blocking і Render-residual.

Таблиця `browser_iterations` містить сирі браузерні ітерації: значення Web Vitals, Navigation Timing, Resource Timing, сукупний розмір ресурсів для real-app режиму, ознаку викиду, повідомлення про помилку, `responseStart`, `responseEnd`, кількість і тривалість long tasks, Total Blocking Time, `blocking_after_response`, `render_time` та джерело вимірювання блокувань (`longtask` або `raf`).

Таблиця `files` каталогізує тестові файли з їх метаданими, статусом активності

та ознакою походження.

Таблиця `app_bundles` зберігає production-збірки веб-застосунків, завантажені для real-app тестування: назву, шлях до розпакованого bundle, початкову назву ZIP-файлу, кількість файлів, сумарний розмір, наявність `index.html`, entry-файл, статус активності та метадані.

Таблиця `settings` реалізує сховище типу «ключ-значення» для конфігурації застосунку, включаючи алгоритми та рівні за замовчуванням, кількість серверних і браузерних ітерацій, warmup-ітерації, тайм-аути та візуальні налаштування.

Міграції бази даних зберігаються у директорії `src/database/migrations` бекенду і виконуються окремим runner-скриптом `npm run db:migrate`. Скрипт послідовно застосовує SQL-файли міграцій, перевіряє наявні таблиці та заповнює каталог вбудованих тестових файлів із директорії `test-files`.

2.2.4 Клієнтська частина (фронтенд)

Клієнтська частина реалізована на фреймворку Angular версії 20 [32] з використанням Standalone API (без NgModule), що є сучасним підходом до побудови Angular-застосунків. Для стилізації використовується Tailwind CSS [33], для візуалізації даних - Chart.js із бібліотекою-обгорткою ng2-charts [34], для real-time комунікації - `socket.io-client`.

Маршрутизація. Застосунок використовує lazy loading для завантаження функціональних модулів за вимогою, що зменшує розмір початкового бандлу. Визначено основні користувацькі маршрути: головна панель (`/dashboard`), створення бенчмарку (`/benchmarks`), деталі бенчмарку (`/benchmark/:id`), історія результатів (`/results`) із дочірнім маршрутом порівняння, управління файлами (`/files`), аналіз (`/analysis`) із п'ятьма дочірніми вкладками, налаштування, документація та сторінка відомостей про застосунок.

Управління станом. Стан застосунку управляється через паттерн Behavior Subject-based state management з використанням RxJS [35]. Для кожної функціональної області створено окремий сервіс стану: `BenchmarkStateService` зберігає список бенчмарків, активний бенчмарк, прогрес виконання та статистику; `ResultsState Service` - результати з можливістю фільтрації за алгоритмом та файлом; `FilesState Service` - список файлів із розподілом на активні, вбудовані та завантажені; `UIState Service` - стан інтерфейсу (тема, бокова панель). З кожного базового Subject обчислюються похідні Observable, наприклад `runningBenchmarks` автоматично фільтрує лише бенчмарки зі статусом `running`.

HTTP-інтеграція. Усі HTTP-запити проходять через централізований `ApiService`, що інкапсулює Angular `HttpClient` та надає типізовані методи для GET, POST, PUT, PATCH, DELETE, upload та download операцій. Базова URL-адреса (http://localhost:3001/api) та тайм-аут (30 секунд) визначаються у конфігурації середовища. Два HTTP-перехоплювачі (interceptors) забезпечують наскрізну функціональність: `LoggingInterceptor` логує метод, URL, статус та тривалість кожного запиту; `HttpErrorInterceptor` перехоплює помилки та формує уніфіковані повідомлення для кожного HTTP-коду стану.

Спільні компоненти. Для забезпечення консистентності інтерфейсу розроблено бібліотеку спільних компонентів: картки з тінню (`CardComponent`), графіки з обгорткою Chart.js (`ChartComponent`), індикатори завантаження, бейджі статусу бенчмарку, бейджі браузера та мережевого профілю, картки ключових показників, прогрес-бар та спеціалізований компонент візуалізації Web Vitals.

Пайпи для форматування. Реалізовано набір Angular-пайпів для форматування специфічних для домену значень: `BytesPipe` перетворює байти у зручночитаний формат (КБ, МБ, ГБ), `DurationPipe` - мілісекунди у відповідний масштаб (мс, с, хв), `PercentagePipe` форматує відсотки, `SpeedPipe` - швидкість стиснення у МБ/с, `RelativeTimePipe` відображає час у відносному форматі («2 хвилини тому»).

2.3 Реалізація модулів компресії

Ядром системи тестування є модулі компресії, що забезпечують уніфікований інтерфейс для роботи з трьома досліджуваними алгоритмами. Кожен модуль реалізує два основних методи - `compress(data, level)` та `decompress(data)` - та надає інформацію про підтримуваний діапазон рівнів компресії.

2.3.1 Модуль Gzip

Модуль Gzip реалізовано на основі вбудованого модуля `zlib` платформи Node.js [23], що надає біндинги до бібліотеки zlib, написаної мовою C. Використовуються функції `zlib.gzip()` та `zlib.gunzip()`, обгорнуті у проміси за допомогою утиліти `util.promisify()` для забезпечення асинхронного інтерфейсу. Параметр рівня компресії передається через об'єкт конфігурації `{ level: N }`, де N - ціле число від 1 до 9. Рівень 1 забезпечує мінімальне стиснення з максимальною швидкістю, рівень 9 - максимальне стиснення з мінімальною швидкістю, а рівень 6 є значенням за замовчуванням, що забезпечує збалансоване співвідношення.

2.3.2 Модуль Brotli

Модуль Brotli також використовує вбудований модуль `zlib`, де підтримка Brotli доступна з Node.js версії 10.16.0. Функції `zlib.brotliCompress()` та `zlib.brotliDecompress()` обгорнуті у проміси аналогічно до Gzip. Рівень якості задається через параметр `BROTLI_PARAM_QUALITY` у об'єкті `params`: `{ params: { [zlib.constants.BROTLI_PARAM_QUALITY]: N } }`, де N - ціле число від 0 до 11. Рівень 0 забезпечує швидку компресію, порівнянну з Gzip, а рівень 11 - максимальне стиснення, що потребує значних обчислювальних ресурсів.

2.3.3 Модуль Zstandard

У реалізації цієї роботи Zstandard використовується через npm-пакет `@mongodb-js/zstd` [36] - нативний біндинг до бібліотеки libzstd, написаної мовою C. Такий підхід було обрано для сумісності з використаним середовищем Node.js, хоча у новіших версіях Node.js модуль `zlib` уже містить експериментальні Zstd API [23]. Функції `compress(data, level)` та `decompress(data)` пакету `@mongodb-js/zstd` є асинхронними за замовчуванням і не потребують додаткового обгортання у проміси. Рівні компресії варіюються від 1 до 22, при цьому рівень 3 є значенням за замовчуванням і забезпечує коефіцієнт стиснення, порівнянний із Gzip рівня 6, при значно вищій швидкості.

2.3.4 Процес виконання тесту стиснення

Виконання тесту стиснення реалізовано як послідовність warmup-ітерацій та виміряних ітерацій. Warmup-запуски відкидаються, а для виміряних ітерацій зберігаються сирі значення, після чого обчислюються агреговані метрики з використанням медіани та фільтрації викидів за правилом Tukey $1{,}5 \times IQR$, де IQR означає міжквартильний розмах.

Крок 1: Профілювання компресії. Вхідний файл подається на вхід компресора обраного алгоритму з визначеним рівнем. Під час виконання операції стиснення збираються метрики за допомогою класу `Profiler`: час виконання вимірюється через `performance.now()` з модуля `perf_hooks` Node.js, споживання CPU - через `process.cpuUsage()` з подальшим переведенням user та system time у мілісекунди, а споживання пам'яті - через `process.memoryUsage()`, що надає розміри heap, RSS та external memory у байтах.

Крок 2: Профілювання декомпресії. Стиснені дані розпаковуються відповідним декомпресором. Для декомпресії фіксується час виконання, який використовує-

ться для розрахунку швидкості декодування та для перевірки практичної вартості відновлення даних.

Крок 3: Верифікація цілісності. Розпаковані дані порівнюються з оригіналом за допомогою методу `Buffer.equals()`, що виконує побайтове порівняння. Результат перевірки зберігається у прапорці `isValid`. Ця верифікація є критичною для підтвердження коректності реалізації кожного алгоритму.

Крок 4: Обчислення метрик. На основі зібраних даних обчислюються похідні метрики:

- коефіцієнт стиснення:

$$CR = \frac{S_{\text{orig}} - S_{\text{comp}}}{S_{\text{orig}}} \times 100\%;$$

- швидкість компресії:

$$V_c = \frac{S_{\text{orig}}}{1024^2} \div \frac{T_c}{1000} \text{ (МБ/с)};$$

- швидкість декомпресії:

$$V_d = \frac{S_{\text{orig}}}{1024^2} \div \frac{T_d}{1000} \text{ (МБ/с)}.$$

Крок 5: Збереження результату. Агреговані метрики зберігаються у таблиці `results`, а сирі вимірювання кожної ітерації - у таблиці `iterations`. Прогрес бенчмарку оновлюється та транслюється підписаним клієнтам через WebSocket.

2.3.5 Конфігурація рівнів компресії

Система визначає три пресети для швидкого налаштування бенчмарків (табл. 2.1).

Табл. 2.1 - Пресети рівнів компресії

Пресет	Gzip	Brotli	Zstd
Fast	1	0	1
Balanced	6	6	3
Best	9	11	22

Для повного тестування визначено набір репрезентативних рівнів: Gzip - [1, 3, 6, 9], Brotli - [0, 4, 6, 11], Zstd - [1, 3, 10, 22]. Ці значення обрані для покриття всього діапазону від мінімального до максимального стиснення з рівномірним розподілом.

2.4 Браузерне тестування з Playwright

Окрім серверних вимірювань швидкості та ефективності стиснення, система забезпечує тестування практичного впливу компресії на продуктивність завантаження веб-сторінок у реальних браузерах. Для цього використовується фреймворк Playwright [37] - інструмент автоматизації браузерів, що підтримує Chromium, Firefox та WebKit.

У системі реалізовано два режими браузерного тестування. Режим `singleFile` використовується як допоміжний діагностичний механізм для перевірки коректності `Content-Encoding` на окремому ресурсі. Фінальне браузерне дослідження виконується у режимі `realApp`, оскільки метрики Web Vitals, зокрема LCP, формуються не одним файлом, а повною критичною траєкторією завантаження: HTML, CSS, JavaScript, виконанням коду, побудовою DOM, layout та paint. Тому real-app режим є методологічно придатнішим для висновків про користувацький досвід.

2.4.1 Архітектура браузерного тестування

Браузерне тестування архітектурно відокремлене від серверного і складається з трьох компонентів: сервера стиснення, модуля керування браузерами та колекторів метрик.

Сервер стиснення - це окремий Express.js HTTP-сервер на порті 8080, що обслуговує тестові сторінки, production-збірки застосунків та стиснені ресурси. Для діагностичного single-file режиму він надає ендпоінт тестової сторінки (`GET /test-page/:algorithmWithLevel/:fileName`), ендпоінт стисненого файлу (`GET /:algorithmWithLevel/:fileName`) і ендпоінт нестисненого файлу (`GET /raw/:fileName`). Для real-app режиму використовується маршрут `GET /app/:appId/:algorithmWithLevel/*`, який роздає файли завантаженої production-збірки з відповідним `Content-Encoding`. Значення `algorithmWithLevel` може мати вигляд `gzip-6`, `brotli-6`, `zstd-3` або `none` для baseline без стиснення. Для SPA-застосунків сервер підтримує fallback на `index.html` і переписує `base href`, щоб маршрути клієнтського застосунку працювали у вкладеному шляху

`/app/...`. Перед браузерним запуском ресурси попередньо стискаються у кеші, щоб вимірювання TTFB не включало повторну runtime-компресію на сервері.

Модуль керування браузерами (`BrowserBenchmark`) оркеструє запуск тестів у різних комбінаціях браузерів та мережевих умов. Для кожної комбінації виконується запуск браузера у headless-режимі з роздільною здатністю 1920×1080 пікселів, створення ізольованого контексту, застосування мережевого профілю, навігація на тестову URL-адресу, збір метрик та створення скріншоту. У real-app режимі сторінка очікується до стану `networkidle`, щоб у вимірювання потрапило завантаження всієї production-збірки.

2.4.2 Емуляція мережевих умов

Для дослідження впливу стиснення в різних мережевих умовах система емулює три типи з'єднань (табл. 2.2).

Табл. 2.2 - Мережеві профілі для браузерного тестування

Профіль	Пропускна здатність (↓)	Пропускна здатність (↑)	Латентність
Fast 3G	1,6 Мбіт/с	0,75 Мбіт/с	150 мс
4G	4 Мбіт/с	3 Мбіт/с	50 мс
DSL	2 Мбіт/с	1 Мбіт/с	5 мс

Реалізація емуляції залежить від типу браузера. Для Chromium використовується протокол Chrome DevTools Protocol (CDP) з методом `Network.emulateNetworkConditions`, що забезпечує обмеження пропускної здатності та латентності на мережевому рівні. Для Firefox та WebKit, де CDP недоступний, емуляція реалізована через перехоплення маршрутів (`page.route()`): система отримує тіло відповіді, визначає його розмір і додає затримку, що складається з латентності профілю та наближеного часу передавання `size/downloadThroughput`. Такий fallback не моделює потокову передачу та мультиплексування HTTP/2, але дозволяє враховувати різницю у розмірі стиснених і нестиснених ресурсів.

2.4.3 Збір метрик Web Vitals

Збір метрик Web Vitals реалізовано через ін'єкцію бібліотеки web-vitals [38] у контекст браузера. Бібліотека у форматі IIFE (Immediately Invoked Function Expression) завантажується з `node_modules/web-vitals/dist/web-vitals.iife.js` та додається до контексту браузера через метод `context.addInitScript`, що гарантує

виконання скрипту до завантаження сторінки. Це є критичним для коректного збору метрик CLS та FCP, які фіксуються з самого початку навігації.

Ін'єктований скрипт реєструє колектори для LCP, FCP, CLS, INP та TTFB з параметром `reportAllChanges: true`, що забезпечує отримання кожного оновлення значення, а не лише фінального. Додатково в контекст сторінки додається вимірювання блокувань головного потоку: у Chromium використовується native Long Tasks API, а у Firefox та WebKit - rAF-delta polyfill. Зібрані значення зберігаються у глобальних об'єктах `window.__webVitalsData` та `window.__longTasks`. Після завантаження сторінки система очікує стабілізації метрик: 2 секунди у single-file режимі та 5 секунд у real-app режимі.

2.4.4 Збір метрик продуктивності

Окрім Web Vitals, система збирає детальні метрики через стандартні браузерні API продуктивності [39]. Navigation Timing API (`performance.getEntriesByType('navigation')`) надає таймінги етапів навігації: `domContentLoaded` (час до спрацювання події DOMContentLoaded), `loadComplete` (час до повного завантаження сторінки), `domInteractive` (час до інтерактивності DOM).

Resource Timing API (`performance.getEntriesByType('resource')`) надає характеристики завантаження конкретного ресурсу: `encodedBodySize` (стиснений розмір), `decodedBodySize` (розпакований розмір), `transferSize` (загальний розмір передачі з заголовками), `resourceDuration` (загальний час завантаження ресурсу).

Додатково обчислюється детальний розподіл часу за мережевими фазами: DNS-резолюція (`domainLookupEnd - domainLookupStart`), TCP-з'єднання (`connectEnd - connectStart`), відправка запиту (`responseStart - requestStart`), отримання відповіді (`responseEnd - responseStart`). Paint Timing API (`performance.getEntriesByName('first-paint')`) фіксує момент першого відображення пікселів на екрані.

2.4.5 Потік виконання браузерного тесту

Повний потік виконання браузерного тестування інтегрований із загальним процесом бенчмарку:

1. Для real-app бенчмарку користувач обирає production-збірку застосунку, алгоритми, рівні, браузери, мережеві профілі та кількість ітерацій.

2. `BenchmarkService` формує комбінації `none`, `gzip-N`, `brotli-N`, `zstd-N`, попе-

редньо стискає файли bundle у кеші сервера стиснення та запускає браузерні вимірювання.

3. Для кожної комбінації `BrowserBenchmark` виконує warmup-запуски, після чого запускає виміряні ітерації у свіжому headless-браузері з новим контекстом, ін'єктованими колекторами web-vitals та Long Tasks/rAF.

4. Після навігації на URL формату `/app/:appId/:algorithmWithLevel/` збираються Web Vitals, Navigation Timing, Resource Timing, сукупні розміри ресурсів і канали декомпозиції LCP.

5. Для кожної ітерації створюється скріншот для візуальної верифікації та зберігається у директорії `screenshots` з іменем, що включає ідентифікатор бенчмарку, назву браузера, мережевий профіль та ідентифікатор застосунку.

6. Агреговані результати зберігаються у таблиці `browser_metrics`, а сирі значення кожної ітерації - у таблиці `browser_iterations`.

7. Прогрес транслюється клієнтам через WebSocket-подію `playwright:progress`.

2.5 Забезпечення достовірності результатів

Забезпечення достовірності та відтворюваності результатів є критичним аспектом дослідницької системи. У розробленому програмному забезпеченні реалізовано комплекс заходів, спрямованих на мінімізацію випадкових відхилень та гарантування коректності вимірювань.

Верифікація цілісності даних. Кожен тест стиснення включає обов'язкову перевірку цілісності: дані після циклу компресія-декомпресія побайтово порівнюються з оригіналом. Результат перевірки зберігається у прапорці `isValid`, що дозволяє автоматично виявити та виключити з аналізу некоректні результати.

Високоточне вимірювання часу. Для вимірювання часу компресії та декомпресії використовується функція `performance.now()` з модуля `perf_hooks` Node.js, що повертає час у мілісекундах із субмілісекундною точністю (порівняно з мілісекундною роздільною здатністю `Date.now()`). Це є критичним для коротких операцій стиснення невеликих файлів, де час виконання може становити частки мілісекунди.

Профілювання ресурсів. Споживання CPU вимірюється через системний виклик `process.cpuUsage()`, що повертає фактичний процесорний час user та system;

у системі ці значення переводяться у мілісекунди. Споживання пам'яті вимірюється як різниця показників `process.memoryUsage()` до та після операції, що дозволяє оцінити інкрементальне споживання кожного окремого тесту.

Повторення, warmup та очищення викидів. Серверні та браузерні тести виконуються з відокремленням warmup-ітерацій від виміряних ітерацій. Для кожної комбінації зберігаються сирі ітерації, після чого система обчислює описову статистику та позначає викиди за правилом Tukey $1{,}5 \times IQR$. У підсумкових рядках використовуються медіанні значення, що зменшує вплив випадкових сплесків навантаження операційної системи або браузера.

Ізоляція браузерних тестів. Кожна браузерна ітерація виконується у свіжому контексті (`browser.newContext()`), що забезпечує ізоляцію від кешів, cookies та іншого стану попередніх тестів. Бібліотека web-vitals і колектори блокувань ін'єктуються через `addInitScript()` до початку навігації, що дозволяє збирати метрики з ранніх етапів життєвого циклу сторінки. Після завантаження сторінки система очікує стабілізації метрик перед зчитуванням результатів.

Емуляція мережевих умов. Використання трьох різних мережевих профілів (Fast 3G, 4G, DSL) дозволяє оцінити вплив стиснення в умовах, наближених до реальних сценаріїв використання. Профілі включають не лише обмеження пропускної здатності, а й емуляцію мережевої латентності, що впливає на метрику TTFB та загальний час завантаження.

Множинність браузерних рушіїв. Тестування у трьох браузерних рушіях (Chromium, Firefox, WebKit) дозволяє виявити відмінності у реалізації декомпресії та рендерингу між різними платформами. Це є особливо важливим, оскільки алгоритм Zstandard має різний рівень зрілості підтримки у різних браузерах.

Автоматизація та відтворюваність. Весь процес тестування повністю автоматизований - від ініціалізації бенчмарку через REST API до збереження результатів у базу даних. Конфігурація кожного бенчмарку зберігається у форматі JSON, що дозволяє точно відтворити будь-який тестовий запуск. Система підтримує зупинку та повторний запуск бенчмарків, а також видалення некоректних результатів.

Візуальна верифікація. Автоматичне створення скріншотів для кожного браузерного тесту забезпечує можливість візуальної перевірки коректності завантаження сторінки. Скріншот підтверджує, що ресурс було успішно декомпресовано та відрендерено браузером, а не повернуто у вигляді помилки.

РОЗДІЛ 3. Результати експериментального дослідження

3.1 Методика проведення експерименту

Експериментальне дослідження проведено з використанням розробленої системи тестування, описаної у розділі 2. У цьому підрозділі наведено характеристики тестового середовища та параметри проведених експериментів.

Розділ побудовано навколо чотирьох дослідницьких питань, сформульованих у Вступі; нижче наведено методику, корпуси даних та матрицю експериментів для їх перевірки.

Апаратне та програмне середовище. Усі тести виконувались на персональному комп'ютері з операційною системою Windows 11 Home (версія 10.0.26200). Серверна частина працювала на платформі Node.js із використанням вбудованого модуля `zlib` для алгоритмів Gzip та Brotli та пакету `@mongodb-js/zstd` для Zstandard. Браузерне тестування виконувалось через Playwright у трьох рушіях: Chromium (версія 145.0), Firefox та WebKit, кожен у headless-режимі з роздільною здатністю 1920×1080 пікселів.

Тестові корпуси даних. Фінальний експеримент не спирається на один універсальний набір із кількох демонстраційних файлів. Для кожного дослідницького питання сформовано окремий корпус, оскільки RQ1, RQ2, RQ3 та RQ4 перевіряють різні властивості алгоритмів. Зведення використаних корпусів наведено у табл. 3.1.

Таке розділення корпусів усуває проблему змішування різних експериментальних цілей. Для RQ1 важлива кількість файлів у межах кожного типу контенту, щоб статистичний тест мав достатню потужність. Для RQ2 потрібні не десятки однотипних файлів, а повний перебір рівнів компресії на кількох характерних ресурсах. Для RQ3 потрібен широкий діапазон розмірів файлів, а для RQ4 - не окремі ресурси, а повноцінні застосунки, у яких компресія впливає на мережеве завантаження, парсинг, виконання JavaScript та рендеринг одночасно.

Параметри тестування. Кожен алгоритм протестовано на кількох рівнях компресії, організованих у три пресети:

- Fast - мінімальна компресія: Gzip рівень 1, Brotli рівень 0, Zstd рівень 1;
- Balanced - збалансований режим: Gzip рівень 6, Brotli рівень 6, Zstd рівень 3;
- Best - максимальна компресія: Gzip рівень 9, Brotli рівень 11, Zstd рівень 22.

Табл. 3.1 - Корпуси даних для фінальних експериментів

Питання	Корпус	Призначення
RQ1	53 реальні веб-файли: CSS, HTML, JavaScript, JSON, SVG; статистичний тест виконано для 50 текстових файлів у групах з достатньою кількістю спостережень	Порівняння коефіцієнта стиснення між алгоритмами за типами контенту
RQ2	7 репрезентативних файлів: `bootstrap-5.css`, `bootstrap-icons.svg`, `caniuse.html`, `geojson-counties.json`, `lodash.min.js`, `three.min.js`, `wikipedia-web.html`	Побудова кривих “швидкість–стиснення” та виділення Парето-оптимальних рівнів
RQ3	53 файли з корпусу RQ1 для базової log-log регресії; додатково 4 JavaScript-файли (`react.production.min.js`, `lodash.min.js`, `chart.umd.js`, `three.min.js`) для перевірки різних рівнів компресії	Аналіз масштабування швидкості залежно від розміру файлу
RQ4	Три production-збірки веб-застосунків: Angular-застосунок дослідницької системи, React/Vite dashboard та Vue/Vite dashboard	Оцінка впливу стиснення на Web Vitals у реальному сценарії завантаження SPA

Підсумкова матриця фінальних експериментів наведена у табл. 3.2. Вона відокремлює серверні вимірювання, де аналізуються властивості алгоритмів як операцій над байтовими послідовностями, від браузерних вимірювань, де оцінюється вплив стиснення на реальне завантаження сторінки.

Для браузерного тестування використано профілі Fast 3G (пропускна здатність 1,6 Мбіт/с, латентність 150 мс), 4G (4 Мбіт/с, 50 мс) та DSL (2 Мбіт/с, 5 мс). У фінальній матриці RQ4 залишено профілі Fast 3G та 4G, оскільки DSL у попередніх прогонах давав близькі до 4G висновки, але суттєво збільшував тривалість експерименту. Результати кожної ітерації зберігались у базі даних SQLite для подальшого аналізу.

Браузерне тестування RQ4. Фінальний браузерний експеримент виконано у режимі real-app: завантажується повноцінна production-збірка веб-застосунку, а стиснення застосовується до всіх його ресурсів одночасно. Такий режим використано замість ізольованого завантаження одного файлу, оскільки метрики Web Vitals формуються не лише розміром окремого ресурсу, а й взаємодією мережевого завантаження, парсингу, виконання JavaScript, побудови DOM та рендерингу. Тому real-app сценарій краще відображає практичний вплив алгоритму стиснення на користувацький досвід.

Табл. 3.2 - Матриця фінальних експериментів RQ1–RQ4

Питання	Призначення	Матриця	Кількість вимірювань
RQ1	Значущість різниці коефіцієнта стиснення для типів веб-контенту	53 файли × 3 алгоритми на balanced-рівнях + baseline	212 результатів, 6784 фізичні ітерації з warmup
RQ2	Компроміс “швидкість–стиснення” та Парето-оптимальні рівні	7 файлів × 44 конфігурації	308 результатів, 9856 фізичних ітерацій з warmup
RQ3	Масштабування швидкості відносно розміру файлу	53 файли balanced + 4 JS-файли × 9 рівнів	1280 фізичних ітерацій у підтверджувальному запуску
RQ4	Вплив стиснення на LCP у реальних застосунках	3 застосунки × браузери × мережі × алгоритми	1020 виміряних browser-ітерацій + 204 warmup

Протокол повторень та очищення даних. Для серверних експериментів RQ1–RQ3 використано схему $2 + 30$: дві warmup-ітерації відкидаються, після чого аналізуються 30 виміряних повторень для кожної комбінації. Для браузерного RQ4 використано схему $2 + 10$, оскільки кожна ітерація передбачає холодний запуск нового процесу браузера і повне завантаження застосунку. Викиди визначаються правилом Tukey $1{,}5\times IQR$: для кожної серії вимірювань обчислюється міжквартильний розмах $IQR = Q_3 - Q_1$, а значення нижче $Q_1 - 1{,}5IQR$ або вище $Q_3 + 1{,}5IQR$ позначаються як викиди; для браузерних метрик застосовується об’єднання викидів за LCP, FCP та TTFB, щоб одна проблемна ітерація не впливала на різні канали декомпозиції по-різному. Для серверних тестів порядок комбінацій `file–algorithm–level` перемішується перед запуском, щоб теплова деградація або фоновий шум не корелювали з конкретним алгоритмом.

Матриця RQ4. Для фінального real-app експерименту набір застосунків розширено до трьох production builds: Angular-застосунок дослідницької системи (43 файли, 1 595 КБ), React/Vite dashboard (3 файли, 436 КБ) та Vue/Vite dashboard (3 файли, 250 КБ). Основна фаза RQ4 має матрицю 3 застосунки × 3 конфігурації (`none`, `gzip-6`, `brotli-6`) × 3 браузери × 2 мережі × 10 ітерацій, тобто 540 виміряних запусків. Окрема Zstd-фаза має матрицю 3 застосунки × 4 конфігурації (`none`, `gzip-6`, `brotli-6`, `zstd-3`) × 2 браузери × 2 мережі × 10 ітерацій, тобто 480 виміряних запусків. Разом RQ4 містить 1020 виміряних браузерних ітерацій та 204 warmup-запуски.

Декомпозиція LCP. У RQ4 LCP не інтерпретується як ізольований час де-

компресії, оскільки сучасні браузери можуть виконувати streaming-декодування паралельно із завантаженням та парсингом. Натомість LCP розкладається на спостережувані канали: TTFB, Download, Blocking та Render-residual. Канал Blocking обчислюється як main-thread blocking після `responseEnd` і до LCP: у Chromium використано native Long Tasks API, а у Firefox та WebKit - rAF-delta polyfill з нижчою часовою точністю. Рядки, у яких `responseEnd` не є надійним через route interception у Playwright, маркуються окремо і не використовуються для висновків про частки TTFB/Download. Конфігурацію Zstd у WebKit виключено з RQ4, оскільки headless-build WebKit у Playwright на Windows не підтримує `Content-Encoding:zstd`; це не узагальнюється на інші браузерні середовища та майбутні версії Safari.

3.2 RQ1: коефіцієнт стиснення за типами контенту

Перше дослідницьке питання стосується того, чи існує статистично значуща різниця у коефіцієнті стиснення між Gzip, Brotli та Zstandard для основних типів веб-контенту. Для цього використано корпус реальних веб-файлів, згрупованих за типами CSS, HTML, JavaScript, JSON та SVG. Порівняння виконувалось на збалансованих рівнях компресії: Gzip-6, Brotli-6 та Zstd-3.

Коефіцієнт стиснення визначався як відсоток зменшення розміру файлу:

$$CR = \frac{S_{\text{orig}} - S_{\text{comp}}}{S_{\text{orig}}} \times 100\%,$$

де S_{orig} - розмір початкового файлу, а S_{comp} - розмір після стиснення. Оскільки для фіксованої пари “файл–алгоритм–рівень” коефіцієнт стиснення є детермінованим, повторні ітерації використовувались для контролю стабільності виконання, а варіативність у табл. 3.3 відображає розкид між файлами одного типу контенту.

Табл. 3.3 - Коефіцієнт стиснення за типами контенту на збалансованих рівнях

Тип	n	Brotli-6	Gzip-6	Zstd-3	χ^2	p
CSS	10	$88{,}92 \pm 7{,}15$	$87{,}60 \pm 6{,}87$	$85{,}94 \pm 8{,}24$	15,800	$3{,}707 \times 10^{-4}$
HTML	10	$84{,}25 \pm 6{,}30$	$82{,}21 \pm 6{,}81$	$81{,}86 \pm 7{,}48$	15,800	$3{,}707 \times 10^{-4}$
JS	11	$67{,}96 \pm 4{,}21$	$66{,}07 \pm 3{,}91$	$64{,}86 \pm 4{,}04$	22,000	$1{,}670 \times 10^{-5}$
JSON	10	$67{,}95 \pm 14{,}63$	$67{,}35 \pm 15{,}27$	$66{,}16 \pm 15{,}84$	18,200	$1{,}117 \times 10^{-4}$
SVG	9	$49{,}69 \pm 20{,}67$	$47{,}53 \pm 23{,}31$	$46{,}14 \pm 22{,}72$	16,222	$3{,}002 \times 10^{-4}$

Графічне подання цих самих значень наведено на рис. 3.1. Воно показує стабільну перевагу Brotli-6 за розміром для всіх типів текстового веб-контенту, але

також демонструє, що абсолютна різниця між алгоритмами є значно меншою, ніж між самими типами ресурсів.

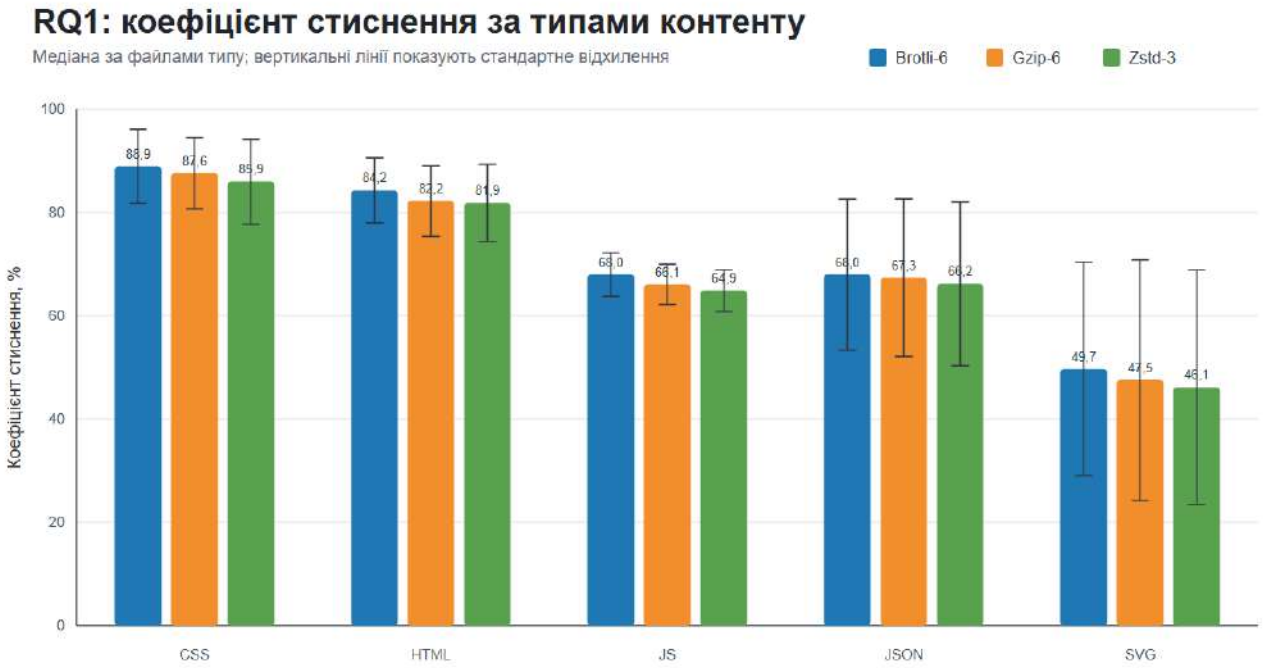

Рисунок 3.1 - Коефіцієнт стиснення за типами контенту на збалансованих рівнях

Для перевірки статистичної значущості використано критерій Фрідмана з блокуванням за файлом. Критерій Фрідмана є непараметричним аналогом дисперсійного аналізу для пов'язаних вибірок і порівнює ранги методів усередині кожного блоку. Такий підхід є доречним, оскільки кожен файл стискався всіма трьома алгоритмами, а розподіли коефіцієнтів стиснення між різними типами файлів не можна вважати нормальними. В усіх п'яти групах отримано $p < 0{,}001$, тому нульова гіпотеза про відсутність різниці між алгоритмами відхиляється.

Найвищий медіанний коефіцієнт стиснення в усіх групах показав Brotli-6. Його перевага над Gzip-6 становить приблизно 1–2 відсоткові пункти для CSS, HTML, JavaScript та SVG, а для JSON різниця є меншою, але зберігає той самий напрям. Це узгоджується з архітектурою Brotli: статичний словник і контекстне моделювання краще використовують повторювані фрагменти веб-контенту, особливо у HTML, CSS та SVG.

Gzip-6 стабільно посідає друге місце за коефіцієнтом стиснення. Його результати близькі до Zstd-3, але в більшості груп він дає дещо менший стиснений розмір. Водночас ця перевага Gzip над Zstd не означає загальної переваги алгоритму, оскільки

RQ1 оцінює лише розмір результату, а не швидкість компресії чи декомпресії. Ці аспекти аналізуються окремо у RQ2 та RQ3.

Zstd-3 показав найнижчий коефіцієнт стиснення серед трьох алгоритмів на збалансованому рівні, але різниця з Gzip-6 є помірною. Це важливий результат для подальшої інтерпретації: Zstandard не перемагає Brotli за чистим коефіцієнтом стиснення, однак може бути практично вигіднішим у сценаріях, де критичною є швидкість обробки. Таким чином, RQ1 підтверджує, що Brotli є найкращим вибором, якщо головним критерієм є мінімальний розмір переданих текстових ресурсів.

3.3 RQ2: компроміс між швидкістю та коефіцієнтом стиснення

Друге дослідницьке питання аналізує не окремо коефіцієнт стиснення або швидкість, а компроміс між ними. Для цього використано корпус із семи репрезентативних веб-ресурсів: CSS, SVG, HTML, JSON та JavaScript-файлів різного розміру. Для кожного файлу виконано повний перебір рівнів Gzip 1–9, Brotli 0–11 та Zstd 1–22, а також baseline без стиснення. Таким чином, для кожного файлу отримано 44 конфігурації, а загалом - 308 точок у просторі “швидкість–стиснення”.

Для кожного файлу побудовано дві Парето-множини. Перша, серверна, використовує координати `compressionSpeed` та коефіцієнт стиснення; вона відповідає питанню, які конфігурації дають найкращий виграш у розмірі за заданої швидкості кодування. Друга, клієнтська, використовує `decompressionSpeed` та коефіцієнт стиснення; вона відповідає питанню, які конфігурації є найкращими з точки зору швидкого декодування у браузері. Конфігурація вважається Парето-оптимальною, якщо не існує іншої конфігурації, що одночасно має не нижчу швидкість і не нижчий коефіцієнт стиснення, причому хоча б за одним із критеріїв є строго кращою.

У табл. 3.4 наведено зведення для ключових конфігурацій. Значення швидкості та коефіцієнта стиснення є медіанами за сімома файлами корпусу, а останні два стовпці показують, для скількох файлів конфігурація потрапила на серверну або клієнтську Парето-множину.

На рис. 3.2 показано розташування всіх конфігурацій у просторі “швидкість компресії–коефіцієнт стиснення”. Більший розмір точки означає, що конфігурація частіше потрапляла на серверну Парето-межу для файлів корпусу RQ2.

Перший висновок із Парето-аналізу полягає в тому, що Gzip не формує ефективної межі в цьому корпусі. Gzip-6 та Gzip-9 не потрапили на жодну з Парето-множин: у режимах, близьких до Gzip за коефіцієнтом стиснення, існують конфігурації Zstd, які

Табл. 3.4 - Ключові конфігурації RQ2 та їх присутність на Парето-множинах

Конфігурація	CR, %	Comp., МБ/с	Decomp., МБ/с	Парето (comp./decomp.)
Gzip-6	74,96	50,9	192,4	0/7 / 0/7
Gzip-9	75,03	33,2	182,7	0/7 / 0/7
Brotli-6	77,43	43,2	187,3	6/7 / 0/7
Brotli-10	79,40	1,9	173,2	7/7 / 2/7
Brotli-11	79,79	0,8	185,8	7/7 / 7/7
Zstd-1	71,74	280,3	506,7	6/7 / 3/7
Zstd-3	74,10	182,0	468,2	6/7 / 0/7
Zstd-5	75,67	100,2	435,6	7/7 / 0/7
Zstd-18	78,66	5,4	492,4	6/7 / 1/7
Zstd-22	78,68	3,4	481,9	3/7 / 6/7

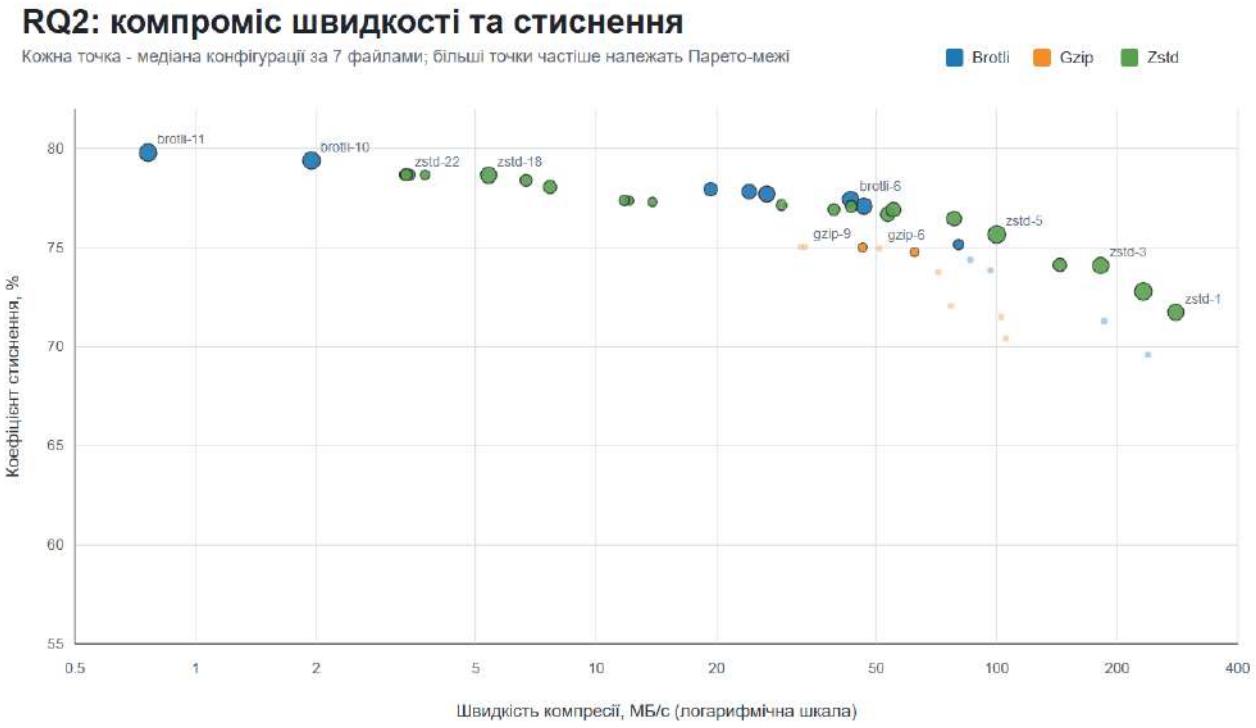

Рисунок 3.2 - Компроміс між швидкістю компресії та коефіцієнтом стиснення для конфігурацій RQ2

стискають швидше і не поступаються за розміром результату. Це не означає, що Gzip втрачає практичну цінність: він залишається універсальним fallback-алгоритмом, але в оптимізаційному просторі “швидкість–стиснення” його роль пояснюється насамперед сумісністю, а не ефективністю.

Друга закономірність стосується Brotli. Рівні Brotli-10 та Brotli-11 потрапили на серверну Парето-множину для всіх семи файлів, тобто забезпечують найкращий розмір у зоні максимального стиснення. Водночас медіанна швидкість Brotli-11 становить лише 0,8 МБ/с, тому цей рівень не є придатним для динамічного стиснення на

запит. Його сильний сценарій - попереднє стиснення статичних ресурсів під час build/deploy, де час компресії сплачується один раз, а виграш у розмірі використовується багаторазово.

Третя закономірність полягає в тому, що Zstd займає швидку частину Парето-межі. Конфігурації Zstd-1, Zstd-3 та Zstd-5 мають медіанну швидкість компресії від 100,2 до 280,3 МБ/с і потрапляють на серверну Парето-множину для більшості або всіх файлів. Особливо показовим є Zstd-5: він має вищий медіанний коефіцієнт стиснення, ніж Gzip-6, і приблизно вдвічі вищу швидкість компресії. Zstd-3 дає дещо менший розмір виграшу, але залишається привабливим для runtime-сценаріїв завдяки дуже високій швидкості.

Клієнтська Парето-множина має іншу структуру. Brotli-11 залишається Парето-оптимальним для всіх файлів, оскільки його висока вартість компресії не впливає на декодування у клієнта. Водночас Zstd-22 потрапляє на клієнтську межу для 6 із 7 файлів: він дає коефіцієнт стиснення, близький до Brotli-11, але має медіанну швидкість декомпресії 481,9 МБ/с проти 185,8 МБ/с у Brotli-11. Це показує, що для статичного контенту Zstd-22 може бути конкурентом Brotli-11 у середовищах, де клієнти гарантовано підтримують `Content-Encoding:zstd`.

Отже, RQ2 показує, що універсального “найкращого” рівня не існує. Для статичного контенту, який можна стиснути заздалегідь, найсильнішими кандидатами є Brotli-11 та Zstd-22. Для динамічного стиснення доцільніше використовувати швидкі рівні Zstd, насамперед Zstd-3 або Zstd-5. Gzip варто залишати як сумісний fallback, але не як оптимальний вибір за критерієм “ціна–виграш”.

3.4 RQ3: масштабування швидкості від розміру файлу

Третє дослідницьке питання перевіряє, як швидкість компресії та декомпресії змінюється зі збільшенням розміру файлу. Для цього використано log-log регресію, оскільки розміри файлів у корпусі охоплюють кілька порядків: від невеликих HTML/CSS/JSON/SVG-файлів до великих JavaScript-бандлів. Для кожної конфігурації апроксимувалась модель:

$$\log_{10}(V) = a + b \cdot \log_{10}(S),$$

де V - швидкість обробки у МБ/с, S - розмір файлу в байтах, a - вільний член, а b - нахил регресійної прямої. Якщо $b > 0$, то ефективна швидкість у МБ/с зростає зі

збільшенням файлу, тобто фіксовані накладні витрати на ініціалізацію компресора, підготовку словника та виклики нативних бібліотек амортизуються на більших обсягах даних. Якщо $b \approx 0$, швидкість майже не залежить від розміру; якщо $b < 0$, великі файли обробляються повільніше у перерахунку на МБ/с.

Основну регресію виконано на 53 реальних веб-файлах для збалансованих рівнів Gzip-6, Brotli-6 та Zstd-3. Результати наведено у табл. 3.5.

Табл. 3.5 - Log-log регресія швидкості на збалансованих рівнях, $n = 53$

Конфігурація	b_c	95% CI	R^2_c	b_d	95% CI	R^2_d
Gzip-6	0,499	[0,436; 0,562]	0,829	0,669	[0,616; 0,722]	0,924
Brotli-6	0,626	[0,575; 0,677]	0,920	0,662	[0,610; 0,714]	0,926
Zstd-3	0,566	[0,504; 0,628]	0,865	0,662	[0,596; 0,728]	0,885

b_c - нахил для швидкості компресії; b_d - нахил для швидкості декомпресії.

Усі три алгоритми на збалансованих рівнях мають додатний нахил як для компресії, так і для декомпресії. Це означає, що на малих файлах суттєву частку часу займають фіксовані накладні витрати, а на більших файлах алгоритми виходять на ефективніший потоковий режим. Високі значення R^2 (0,829–0,926) показують, що розмір файлу добре пояснює зміну швидкості у цьому корпусі.

Найбільший нахил компресії має Brotli-6 ($b = 0{,}626$), тобто його швидкість найсильніше виграє від збільшення розміру файлу. Це узгоджується з тим, що на малих файлах Brotli має відчутні накладні витрати на контекстне моделювання та роботу зі словником, а на більших файлах ці витрати розподіляються на більший обсяг даних. Zstd-3 має нахил 0,566 і водночас, як показано у RQ2, залишається найшвидшим у абсолютному вимірі серед runtime-конфігурацій. Gzip-6 має нижчий нахил компресії (0,499), але також демонструє стабільну амортизацію накладних витрат.

Для декомпресії нахили трьох алгоритмів дуже близькі: 0,662–0,669. Це означає, що у збалансованих режимах декодування всіх трьох форматів однаково виграє від збільшення розміру файлу. Практичний висновок полягає в тому, що різниця між алгоритмами на клієнті визначається не лише формою масштабування, а й абсолютною швидкістю декомпресії та підтримкою конкретного браузера.

Щоб перевірити, чи зберігається така поведінка на різних рівнях компресії, виконано додаткову регресію на чотирьох JavaScript-файлах для дев'яти конфігурацій: швидкі, збалансовані та максимальні рівні кожного алгоритму. Результати для швидкості компресії наведено у табл. 3.6.

Табл. 3.6 - Log-log регресія швидкості компресії на різних рівнях, $n = 4$

Конфігурація	b	95% CI	R^2
Gzip-1	0,463	[0,344; 0,582]	0,987
Gzip-6	0,310	[0,170; 0,451]	0,961
Gzip-9	0,147	[0,047; 0,247]	0,916
Brotli-0	0,593	[0,229; 0,957]	0,931
Brotli-6	0,521	[0,297; 0,746]	0,965
Brotli-11	-0,027	[-0,153; 0,099]	0,191
Zstd-1	0,511	[0,246; 0,776]	0,950
Zstd-3	0,347	[0,022; 0,673]	0,852
Zstd-22	-0,109	[-0,240; 0,023]	0,775

Візуальне порівняння нахилів наведено на рис. 3.3. Додатні нахили для швидких і збалансованих рівнів показують амортизацію фіксованих витрат, тоді як Brotli-11 та Zstd-22 наближаються до нульового або від'ємного масштабування.

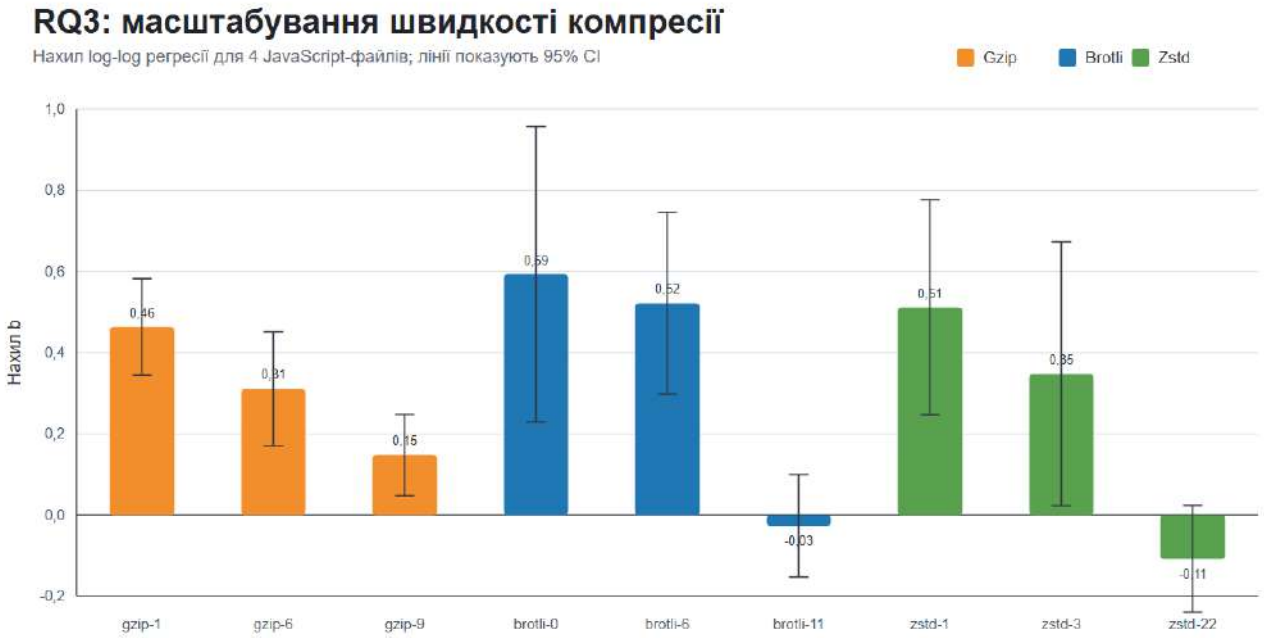

Рисунок 3.3 - Нахили log-log регресії швидкості компресії для різних рівнів RQ3

Підтверджувальний запуск показує, що масштабування залежить не лише від алгоритму, а й від рівня компресії. Для Gzip нахил поступово зменшується від 0,463 на рівні 1 до 0,147 на рівні 9. Це означає, що вищі рівні Gzip гірше виграють від збільшення розміру файлу: додатковий пошук збігів збільшує обчислювальні витрати швидше, ніж зростає корисний обсяг обробки.

Для Brotli різниця між рівнями ще виразніша. Brotli-0 та Brotli-6 мають додатні нахили 0,593 і 0,521, тобто поводяться як runtime-рівні з амортизацією накладних

витрат. Натомість Brotli-11 має нахил $-0{,}027$ з довірчим інтервалом, що включає нуль, і дуже низьке $R^2 = 0{,}191$. Іншими словами, для максимального рівня швидкість компресії майже не масштабується як проста функція розміру: її визначає складність пошуку оптимального представлення, а не лише кількість байтів.

Zstd демонструє подібний поділ. Zstd-1 та Zstd-3 мають додатні нахили 0,511 і 0,347, що підтверджує їх придатність для швидкого стиснення на льоту. Zstd-22 має від'ємний нахил $-0{,}109$, тобто на великих JavaScript-файлах максимальний рівень втрачає ефективність у МБ/с. Це не робить Zstd-22 поганим рівнем загалом; навпаки, у RQ2 він потрапляє на клієнтську Парето-множину для 6 із 7 файлів. Але його місце - статичне попереднє стиснення, а не runtime-компресія.

Отже, RQ3 підтверджує поділ рівнів на дві практичні групи. Швидкі та збалансовані рівні (Gzip-1/6, Brotli-0/6, Zstd-1/3) масштабуються передбачувано і можуть використовуватись для динамічного контенту. Максимальні рівні Brotli-11 та Zstd-22 поводяться інакше: вони можуть давати найкращий розмір, але їхня швидкість компресії не масштабується достатньо добре для стиснення на запит. Цей результат узгоджується з висновком RQ2: вибір рівня повинен залежати від сценарію використання, а не лише від бажання отримати найменший файл.

3.5 RQ4: декомпозиція LCP у real-app сценаріях

Четверте дослідницьке питання переходить від властивостей окремого алгоритму до практичного ефекту у браузері: як стиснення впливає на Largest Contentful Paint реального веб-застосунку і через які спостережувані канали формується цей вплив. На відміну від single-file тестів, у RQ4 завантажувались повноцінні production-збірки трьох SPA: Angular-застосунок дослідницької системи, React/Vite dashboard та Vue/Vite dashboard. Стиснення застосовувалось до всіх ресурсів застосунку одночасно.

Для кожної ітерації LCP розкладався на чотири канали:

$$LCP = TTFB + Download + Blocking + Render.$$

Тут *TTFB* відповідає часу до першого байта, *Download* - часу від першого до останнього байта відповіді, *Blocking* - сумарному main-thread blocking після `responseEnd` і до LCP, а *Render* - залишку, що включає парсинг, побудову DOM, layout, paint та іншу роботу, яку неможливо ізолювати публічними browser API. Така декомпозиція

не претендує на вимірювання “чистої декомпресії”: сучасні браузери можуть виконувати streaming-декодування паралельно із завантаженням та парсингом. Тому канал Blocking слід трактувати як спостережувану post-network роботу головного потоку, а не як ізольований CPU-час декомпресора.

У Chromium для каналу Blocking використано native Long Tasks API. У Firefox та WebKit застосовано rAF-delta polyfill, що має нижчу часову точність і не дає атрибуції джерела блокування. Крім того, рядки Firefox, де `responseEnd` згортається до нуля через route interception у Playwright, позначено як ненадійні для висновків про частки TTFB та Download. Такі рядки використовуються лише для порівняння загального LCP.

Табл. 3.7 - Сукупний виграш LCP від стиснення порівняно з `none`

Алгоритм	Рядків	Median saved, мс	p25, мс	p75, мс
Brotli-6	18	635	8	1020
Gzip-6	18	629	7	993
Zstd-3	6	882	670	1390

До таблиці включено лише рядки з надійним Navigation Timing.

Зведений результат у табл. 3.7 показує, що стиснення у real-app сценарії дає практично значущий виграш LCP. Для надійних рядків медіанна економія становить 629 мс для Gzip-6, 635 мс для Brotli-6 та 882 мс для Zstd-3. Водночас кількість рядків для Zstd менша, оскільки Zstd-фаза виконувалась лише для Chromium та Firefox, а WebKit/Zstd було виключено через обмеження Playwright WebKit на Windows.

Найчіткіше ефект видно у Chromium, де мережева емуляція виконується через CDP, а канал Blocking вимірюється Long Tasks API. Табл. 3.8 порівнює LCP для трьох застосунків у профілях 4G та Fast 3G.

Табл. 3.8 - LCP у Chromium для real-app сценаріїв, мс

Застосунок	Мережа	None	Gzip-6	Brotli-6	Zstd-3	Найкращий виграш
Angular	4G	1910	920	902	958	1008
Angular	Fast 3G	4624	2160	2096	2212	2528
React	4G	1052	424	418	430	634
React	Fast 3G	2518	938	908	982	1610
Vue	4G	686	348	348	366	338
Vue	Fast 3G	1616	760	742	804	874

На рис. 3.4 ці результати подано як порівняння чотирьох конфігурацій для кожної пари “застосунок–мережа”. В усіх випадках перехід від `none` до стисненого варіанта дає більший ефект, ніж вибір між Gzip-6, Brotli-6 та Zstd-3.

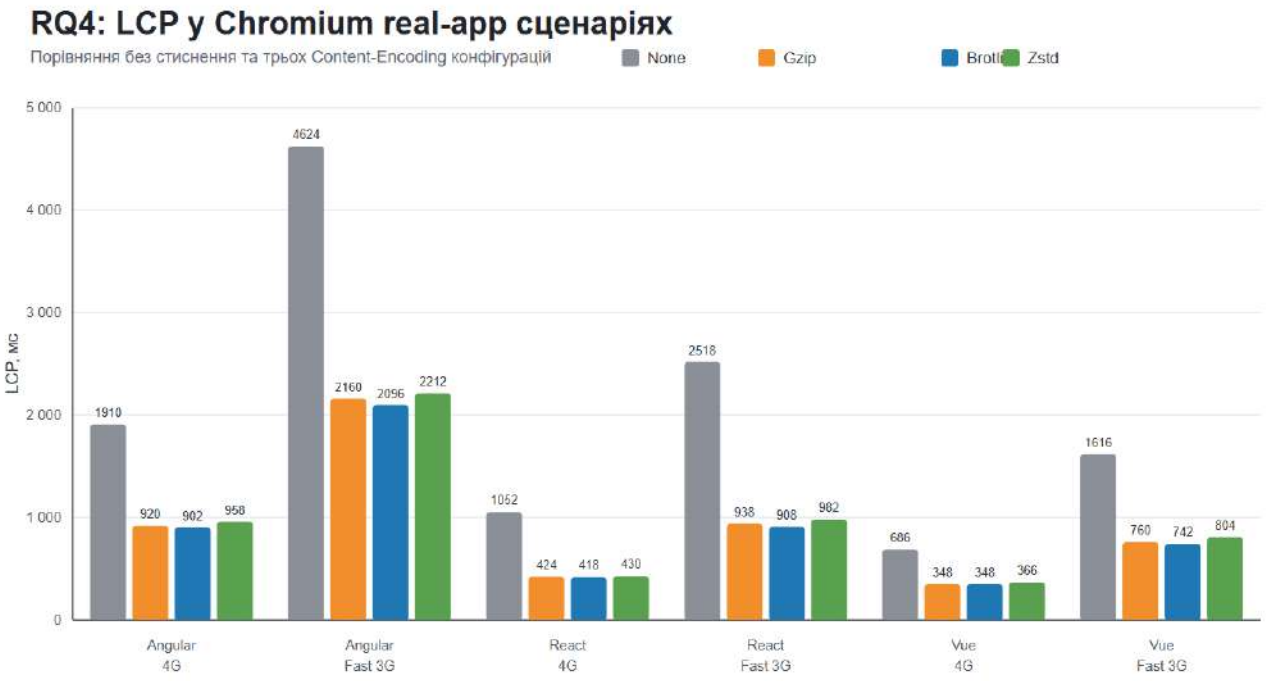

Рисунок 3.4 - LCP у Chromium для real-app сценаріїв RQ4

У всіх Chromium-сценаріях стиснення суттєво зменшує LCP. Для Angular-застосунку на Fast 3G перехід від нестиснених ресурсів до Brotli-6 зменшує LCP з 4624 мс до 2096 мс, тобто економить 2528 мс. Для React-збірки у тому самому мережевому профілі економія становить 1610 мс, а для Vue - 874 мс. Отже, ефект залежить не лише від алгоритму, а й від розміру та структури конкретної production-збірки: більший Angular bundle дає більший абсолютний виграш, тоді як менша Vue-збірка має нижчий baseline і відповідно меншу економію.

Brotli-6 найчастіше дає мінімальний LCP у Chromium, але різниця з Gzip-6 зазвичай невелика порівняно з різницею між будь-яким стисненням і `none`. Наприклад, для React/Fast 3G Brotli-6 випереджає Gzip-6 на 30 мс, але обидва алгоритми економлять понад 1,5 с відносно нестисненого варіанта. Zstd-3 стабільно покращує LCP порівняно з `none`, однак у цих real-app збірках не перевершує Brotli-6 за загальним LCP у Chromium.

Для пояснення цього результату важлива не лише величина LCP, а й його структура. У табл. 3.9 наведено приклади декомпозиції для конфігурації Brotli-6 у Chromium.

Структуру цих значень для Brotli-6 додатково візуалізовано на рис. 3.5. Оскільки у Chromium для цих рядків Blocking дорівнював нулю, на рисунку залишено три

Табл. 3.9 - Декомпозиція LCP для Brotli-6 у Chromium, мс

Застосунок	Мережа	LCP	TTFB	Download	Blocking	Render	Download, %
Angular	4G	902	3	73	0	826	8,1
Angular	Fast 3G	2096	3	174	0	1919	8,3
React	4G	418	3	76	0	338	18,3
React	Fast 3G	908	4	172	0	732	18,9
Vue	4G	348	4	71	0	273	20,4
Vue	Fast 3G	742	4	174	0	564	23,5

видимі складові: TTFB, Download та Render-residual.

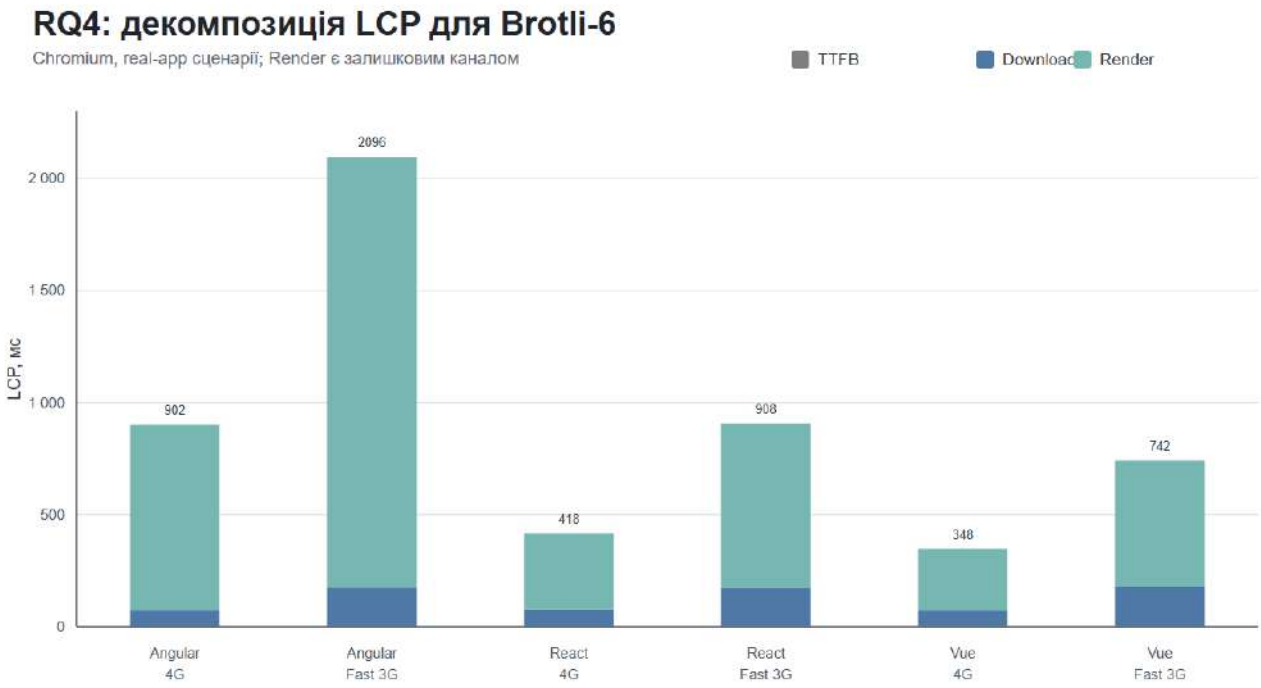

Рисунок 3.5 - Декомпозиція LCP для Brotli-6 у Chromium

Декомпозиція показує, що після стиснення видимий канал Download не є єдиним домінуючим чинником LCP. У Chromium він становить приблизно 8% LCP для Angular, 18–19% для React і 20–24% для Vue. Менший стиснений розмір не завжди автоматично дає найнижчий LCP. Канал Render-residual у Chromium є залишковою величиною ($LCP - TTFB - Download - Blocking$) і включає завантаження та виконання JavaScript, побудову DOM, layout, paint, а також роботу, яку метод не виокремлює явно. Тому переважання Render-residual слід інтерпретувати не як доведене обмеження rendering pipeline, а як свідчення того, що після стиснення мережевий етап перестає бути єдиним домінуючим компонентом LCP.

Водночас порівняння з `none` показує, що стиснення впливає не лише на

Download як ізольований канал. У Chromium для Angular/Fast 3G нестиснений варіант має LCP 4624 мс, тоді як Brotli-6 - 2096 мс. Різниця більша за саму різницю медіанного `responseEnd`, що вказує на каскадний ефект: швидше отримання ресурсів раніше запускає парсинг, виконання JavaScript і рендеринг, тому виграш проявляється у всій критичній траєкторії LCP.

Кросбраузерні результати потребують обережнішої інтерпретації. У WebKit рядки мають надійний Navigation Timing, але виграш від стиснення для цих real-app збірок був мінімальним: наприклад, для Angular/4G LCP становив 1119 мс без стиснення, 1117 мс з Gzip-6 та 1118 мс з Brotli-6. Для React/4G різниця між `none`, Gzip-6 та Brotli-6 також не перевищувала кількох мілісекунд. Це означає, що у WebKit у даному сценарії LCP визначався не розміром переданих ресурсів, а пізнішими етапами рендерингу.

Для Firefox загальний LCP можна порівнювати, однак рядки не використовуються для висновків про частки TTFB та Download, оскільки route interception у Playwright згортає `responseEnd` до нульових значень. Це є обмеженням вимірювального середовища, а не властивістю алгоритмів компресії. Окремо слід зазначити, що конфігурацію Zstd у WebKit виключено з RQ4: headless-build WebKit у Playwright на Windows не підтримує `Content-Encoding:zstd`. Цей результат не слід узагальнювати на всі майбутні WebKit/Safari-середовища, однак поточна браузерна підтримка Zstandard у Safari залишається обмеженою.

Отже, RQ4 підтверджує практичну цінність HTTP-стиснення для реальних SPA, особливо у Chromium та повільнішому мережевому профілі Fast 3G. Основний висновок полягає не в тому, що один алгоритм завжди перемагає, а в тому, що стиснення змінює всю критичну траєкторію LCP: скорочує мережевий етап, раніше запускає рендеринг і зменшує загальний час до найбільшого видимого елемента. Після цього вибір між Gzip-6, Brotli-6 та Zstd-3 має враховувати підтримку браузерів, вартість компресії на сервері та структуру конкретного застосунку.

3.6 Кросбраузерні обмеження експерименту

Оскільки RQ4 виконувався у реальних браузерних рушіях, результати потрібно інтерпретувати з урахуванням відмінностей між Chromium, Firefox та WebKit. Частина цих відмінностей пов'язана з самими браузерами, а частина - з можливостями автоматизації Playwright на Windows. Тому цей підрозділ відокремлює підтверджені експериментальні висновки від обмежень інструментального середовища.

Chromium як основний еталон для декомпозиції. Найповніші та найнадійніші дані отримано у Chromium. Для цього рушія Playwright використовує Chrome DevTools Protocol, що дозволяє задавати мережеві умови на рівні браузера, а канал Blocking вимірюється через native Long Tasks API. Саме тому висновки про частки TTFB, Download, Blocking та Render у підрозділі 3.5 базуються передусім на Chromium-рядках. Це не означає, що Chromium є єдиним релевантним браузером, але для задачі декомпозиції він дає найменше інструментальних спотворень.

Firefox і обмеження route interception. Для Firefox мережева емуляція у Playwright виконувалась через перехоплення маршрутів, а не через CDP. У цьому режимі Navigation Timing для локально перехоплених відповідей іноді згортає `responseStart` та `responseEnd` до нульових або близьких до нуля значень. Через це рядки Firefox не використовувались для висновків про частки TTFB та Download. Загальний LCP у Firefox залишається корисним як end-to-end метрика, але його декомпозиція на мережеві канали є ненадійною.

WebKit і підтримка Zstandard у Playwright на Windows. У фінальному експерименті WebKit/Zstd було виключено, оскільки headless-build WebKit, який постачається з Playwright на Windows, не декодує відповіді з `Content-Encoding: zstd`. Це є обмеженням конкретного автоматизованого середовища тестування, а не універсальним твердженням про всі WebKit-середовища. Водночас станом на актуальні таблиці сумісності підтримка `zstd` у Safari залишається відсутньою або частковою залежно від версії, тому для production-сценаріїв Safari/Zstd потрібне окреме тестування на macOS або у хмарних сервісах на кшталт BrowserStack чи SauceLabs.

Long Tasks API і rAF-delta polyfill. Для Chromium канал Blocking вимірювався через Long Tasks API, який фіксує задачі головного потоку довші за 50 мс. Firefox та WebKit у цьому експерименті використовували rAF-delta polyfill: блокування оцінюється через збільшені інтервали між послідовними `requestAnimationFrame` callback-ами. Такий підхід має нижчу часову точність і не може вказати, чи спричинене блокування декомпресією, виконанням JavaScript, layout або paint. Тому порівняння абсолютних значень Blocking між браузерами слід вважати орієнтовним.

Відмова від старої single-file інтерпретації. Попередні exploratory-запуски з ізольованим великим файлом показували різкі аномалії у WebKit для окремих конфігурацій Brotli та Zstd. У фінальному варіанті ці результати не використовуються як основа висновків, оскільки вони були отримані до стабілізації протоколу повторень,

warmup-ітерацій та real-app сценарію. Фінальний аналіз спирається на production-збірки застосунків, 10 виміряних повторень після warmup і очищення викидів за правилом Tukey.

Таким чином, кросбраузерний висновок роботи є обережним: Chromium дає найнадійнішу картину впливу стиснення на LCP та його канали; Firefox придатний для порівняння загального LCP, але не для точного розкладання мережевих фаз; WebKit у Playwright на Windows не дозволяє автоматизовано перевірити Zstd. Для production-впровадження це означає необхідність стандартного `Accept-Encoding` negotiation, заголовка `Vary: Accept-Encoding`, fallback до Brotli/Gzip та окремої перевірки Safari на реальному macOS/iOS-середовищі.

РОЗДІЛ 4. Аналіз результатів та практичні рекомендації

4.1 Валідовані висновки за дослідницькими питаннями

Результати розділу 3 показують, що вибір алгоритму стиснення не можна звести до одного універсального правила. Алгоритми мають різні сильні сторони залежно від типу ресурсу, моменту виконання компресії, браузерної підтримки та того, яка метрика є основною: розмір файлу, CPU-вартість, швидкість декомпресії або LCP. Тому практична інтерпретація результатів повинна спиратися на сценарій використання, а не лише на середній коефіцієнт стиснення.

У табл. 4.1 зведено основні висновки за чотирма дослідницькими питаннями та їх практичне значення.

Табл. 4.1 - Валідовані висновки RQ1–RQ4 та їх практичне значення

Питання	Підтверджений висновок	Практичне значення
RQ1	Brotli-6 має найвищий коефіцієнт стиснення для всіх досліджених текстових типів; критерій Фрідмана дав $p < 0{,}001$ у кожній групі.	Якщо головна мета - мінімізувати розмір переданого текстового ресурсу, Brotli є найсильнішим кандидатом.
RQ2	Парето-межа розділяється між Brotli та Zstd: Brotli-10/11 дає найменший розмір, Zstd-1/3/5 - найкращу швидку зону. Gzip не формує ефективної межі у корпусі RQ2.	Gzip варто залишати як fallback сумісності, а не як оптимальний алгоритм за критерієм “швидкість–стиснення”.
RQ3	Швидкі та збалансовані рівні масштабуються передбачувано зі збільшенням файлу; максимальні рівні Brotli-11 і Zstd-22 не є runtime-рівнями.	Для динамічного контенту потрібні Zstd-3/5 або помірні Brotli/Gzip; максимальні рівні доречні для попереднього стиснення.
RQ4	У real-app сценаріях стиснення суттєво зменшує LCP, але після скорочення мережевого етапу його вплив перестає бути домінуючим.	Вибір алгоритму потрібно поєднувати з оптимізацією бандлів, code splitting та перевіркою браузерів.

Ключова інженерна інтерпретація така: Brotli найкраще відповідає задачі мінімізації байтів, Zstd - задачі швидкої обробки, а Gzip - задачі максимальної сумісності. У статичному сценарії можна дозволити дорогі рівні компресії, бо їх вартість сплачується один раз під час build/deploy. У динамічному сценарії час компресії входить у TTFB, тому максимальні рівні стають невиправданими навіть тоді, коли дають найменший файл.

4.2 Фреймворк вибору алгоритму

Практичні рекомендації подано у три рівні деталізації: фреймворк прийняття рішень (§ 4.2) для швидкого вибору алгоритму, детальні рекомендації за типовими сценаріями використання (§ 4.3) і стратегія production-впровадження з урахуванням браузерної підтримки (§ 4.5). § 4.4 окремо оцінює очікуваний економічний та продуктивний ефект.

Практичне рішення щодо HTTP-стиснення доцільно приймати у кілька кроків: спочатку визначити тип ресурсу, потім сценарій його генерації, після цього - браузерну підтримку та CPU-бюджет. У табл. 4.2 наведено компактний фреймворк вибору.

Табл. 4.2 - Фреймворк вибору стратегії стиснення

Умова	Рекомендована дія	Обґрунтування
Ресурс уже стиснений або бінарний: PNG, JPEG, WebP, AVIF, WOFF2, audio/video	Не застосовувати HTTP `Content-Encoding`.	Додаткове стиснення майже не зменшує розмір і витрачає CPU.
Текстовий статичний ресурс, доступне попереднє стиснення	Створювати `.br` на Brotli-11 та `.gz` на Gzip-6; за потреби додавати `.zst` на Zstd-22 для сучасних клієнтів.	Brotli-11 і Zstd-22 перебувають у зоні максимального стиснення; Gzip-6 забезпечує fallback.
Текстовий статичний ресурс без build-time стиснення	Використовувати Brotli-6 або Gzip-6, залежно від CPU-бюджету та підтримки сервера.	Помірні рівні дають стабільний виграш без неприйнятної затримки.
Динамічний JSON або SSR HTML, клієнт підтримує `zstd`	Використовувати Zstd-3; Zstd-5 розглядати, якщо важливіший розмір і є CPU-бюджет.	У RQ2 Zstd-3 має медіанну швидкість компресії 182,0 МБ/с, а Zstd-5 - 100,2 МБ/с при кращому стисненні за Gzip-6.
Динамічний текстовий ресурс без підтримки `zstd`	Використовувати Brotli-4/6 або Gzip-6.	Це збалансований fallback для клієнтів і проксі без Zstd.
Ресурс менший за 1 КБ	Стискати лише за наявності явного виграшу або у попередньо стисненому статичному режимі.	Заголовки й ініціалізація компресора можуть нівелювати економію.
JavaScript/CSS-бандл понад 1 МБ	Попередньо стискати і розбивати на менші чанки.	RQ4 показує, що після зменшення Download значну роль відіграє виконання JS та рендеринг.

Цей фреймворк навмисно розділяє “чим стискати” і “коли стискати”. Для статичних ресурсів найкраща стратегія - підготувати кілька варіантів файлу заздалегідь і дозволити серверу або CDN обрати потрібний варіант за `Accept-Encoding`. Для динамічних ресурсів головним обмеженням стає час компресії, тому Zstd-3 є сильним кандидатом саме як runtime-рівень, тоді як Brotli-11 та Zstd-22 потрібно залишити

для build/deploy етапу.

4.3 Рекомендації за сценаріями використання

Для **JavaScript- та CSS-бандлів** рекомендовано попереднє стиснення Brotli-11 з Gzip-6 fallback. Якщо цільова аудиторія має високу частку сучасних Chromium/Firefox-клієнтів і інфраструктура підтримує кешування окремих варіантів, можна додати Zstd-22 як експериментальний або поступово розгорнутий варіант. Для великих бандлів стиснення не замінює code splitting: у RQ4 видно, що після зменшення мережевого часу значна частина LCP залишається у Render-residual, тобто у парсингу, виконанні JavaScript, layout та paint.

Для **API-відповідей у JSON** найдоцільнішим базовим вибором є Zstd-3, якщо клієнт явно підтримує `zstd`. Цей сценарій добре відповідає профілю Zstd: висока швидкість компресії з помірною втратою коефіцієнта стиснення відносно Brotli. Для малих однотипних JSON-відповідей перспективним напрямком є треновані словники Zstd, але їх потрібно впроваджувати лише там, де клієнт і сервер можуть узгоджено використовувати однаковий словник.

Для **SSR HTML і персоналізованих сторінок** рішення залежить від TTFB-бюджету. Якщо сервер має Zstd і клієнт його підтримує, Zstd-3 є природним runtime-рівнем. Якщо підтримка Zstd відсутня або аудиторія неоднорідна, варто використовувати Brotli-4/6 або Gzip-6. Для сторінок, які кешуються на CDN, доцільно перетворювати їх на статичні варіанти й застосовувати попереднє стиснення.

Для **SVG, HTML, CSS та інших текстових статичних ресурсів** стиснення майже завжди корисне, якщо розмір ресурсу перевищує мінімальний поріг. RQ1 підтвердив, що Brotli-6 стабільно дає найкращий коефіцієнт стиснення для CSS, HTML, JS, JSON та SVG, тому для статичних текстових файлів Brotli має бути першим кандидатом за наявності підтримки.

Для **зображень, відео, аудіо та WOFF2-шрифтів** HTTP-стиснення потрібно вимикати. Такі формати вже містять власне стиснення або спеціалізоване ентропійне кодування, тому повторна компресія рідко дає виграш, а іноді збільшує розмір через службові дані формату.

Для **CDN та edge-кешування** необхідно зберігати окремі варіанти відповідей для Brotli, Gzip, Zstd та identity-версії, якщо вона потрібна. Кожна така відповідь має містити заголовок `Vary: Accept-Encoding`, інакше проміжний кеш може повернути клієнту варіант, який той не здатен декодувати.

4.4 Оцінка економічного та продуктивного ефекту

Економічний ефект від стиснення формується не лише різницею між алгоритмами, а передусім самим фактом увімкнення HTTP-компресії для текстових ресурсів. Для трафіку базова оцінка має вигляд:

$$Traffic_{saved} = (S_{identity} - S_{encoded}) \times N,$$

де $S_{identity}$ - розмір ресурсу без стиснення, $S_{encoded}$ - розмір стисненого варіанта, а N - кількість завантажень. Після цього грошова економія визначається тарифом CDN або провайдера трафіку.

На практиці різниця між “без стиснення” і “будь-яке коректне стиснення” зазвичай більша, ніж різниця між Gzip і Brotli. Це видно і в RQ4: для надійних рядків медіанний виграш LCP відносно `none` становив 629 мс для Gzip-6, 635 мс для Brotli-6 та 882 мс для Zstd-3. Отже, перший рівень ROI - не вибір ідеального алгоритму, а гарантія, що текстові ресурси взагалі стискаються.

Другий рівень ROI пов’язаний із CPU-вартістю. У RQ2 Zstd-3 стискав корпус із медіанною швидкістю 182,0 МБ/с, тоді як Gzip-6 - 50,9 МБ/с, а Brotli-6 - 43,2 МБ/с. Це означає, що для динамічних відповідей Zstd може зменшити серверну вартість стиснення без переходу на нестиснений трафік. Для високонавантажених API така різниця перетворюється на більшу пропускну здатність або меншу кількість CPU-ядер для тієї самої кількості запитів.

Третій рівень ROI проявляється у Web Vitals. У Chromium на Fast 3G найкращий стиснений варіант зменшив LCP на 2528 мс для Angular-застосунку, на 1610 мс для React-збірки та на 874 мс для Vue-збірки. Це не означає, що алгоритм стиснення самостійно вирішує проблему продуктивності, але він скорочує мережеву частину критичної траєкторії й раніше запускає парсинг, виконання JavaScript і рендеринг. У сценаріях, де LCP перебуває поблизу порогу 2,5 с, такий виграш може змінити якісну оцінку сторінки у Core Web Vitals.

4.5 Кросбраузерна стратегія впровадження

Виробниче впровадження має спиратися на стандартне узгодження `Content-Encoding`. Сервер не повинен припускати підтримку алгоритму за User-Agent; надійнішим джерелом є заголовок `Accept-Encoding`. Рекомендований порядок для сучасного стеку такий: використовувати Zstd для динамічного текстового контенту,

якщо клієнт явно підтримує `zstd`; використовувати Brotli для статичних попередньо стиснених ресурсів, якщо клієнт підтримує `br`; використовувати Gzip як універсальний fallback.

Кожна відповідь, що залежить від `Accept-Encoding`, повинна містити `Vary: Accept-Encoding`. Для CDN це не формальність, а умова коректного кешування: Brotli-, Zstd-, Gzip- та identity-варіанти мають бути різними об'єктами кешу. Для безпечного розгортання Zstd варто починати з менш ризикових типів ресурсів, наприклад JSON-відповідей і невеликих текстових файлів, а потім розширювати покриття під контролем метрик помилок, LCP та TTFB.

Кросбраузерні результати RQ4 потрібно трактувати обережно. Chromium у цьому дослідженні є найнадійнішим джерелом для декомпозиції LCP, оскільки мережеві умови задавались через CDP, а Blocking вимірювався Long Tasks API. Firefox придатний для порівняння загального LCP, але не для точного аналізу TTFB та Download у цьому стенді через route interception. WebKit у Playwright на Windows не декодував `Content-Encoding: zstd`, тому Safari/Zstd потрібно перевіряти окремо на macOS або iOS, зокрема через реальний Safari чи хмарні сервіси кросбраузерного тестування. Це обмеження не є доказом непрацездатності Zstd у всіх майбутніх Safari-середовищах, але узгоджується з поточним обмеженим станом підтримки.

4.6 Обмеження дослідження та подальша робота

Поточне дослідження має кілька меж застосовності. Серверні тести виконувались на одному комп'ютері з Windows 11, тому абсолютні швидкості можуть відрізнятися на Linux-серверах, у контейнерах або на машинах з іншим CPU. Тести не моделювали конкурентне навантаження, тоді як у виробничому середовищі багато одночасних компресій можуть конкурувати за CPU, кеші процесора та пам'ять.

Браузерні тести виконувались у headless-режимі. Chromium мав найточнішу мережеву емуляцію через CDP, тоді як Firefox і WebKit використовували route interception, що обмежує точність Navigation Timing. Крім того, у RQ4 не тестувались реальні мобільні пристрої, Safari на macOS/iOS, нестабільні мобільні мережі з втратами пакетів, HTTP/2 або HTTP/3 мультиплексування, TLS handshake, браузерний кеш та повторні відвідування.

Декомпозиція LCP у RQ4 є спостережуваною, а не фізичною декомпозицією роботи браузера. Канал Blocking не дорівнює чистому часу декомпресії: браузер може виконувати streaming-декодування паралельно із завантаженням, парсингом та

іншою роботою. Тому ці результати коректно використовувати для оцінки критичної траєкторії LCP, але не для ізольованого мікробенчмарку декомпресора у браузері.

Подальша робота має рухатися у кількох напрямах. По-перше, варто повторити серверні бенчмарки на Linux і під конкурентним навантаженням, щоб оцінити масштабованість Zstd, Brotli та Gzip при реальному RPS. По-друге, потрібно окремо перевірити Safari на macOS/iOS для Zstd та великих статичних ресурсів. По-третє, перспективним є тестування тренованих словників Zstd для малих JSON API та Compression Dictionary Transport для інкрементальних оновлень ресурсів [41]. По-четверте, доцільно дослідити HTTP/2/HTTP/3, кешування, мобільні пристрої та апаратне прискорення на кшталт Intel QAT [42].

Висновки

У дипломній роботі проведено комплексне дослідження алгоритмів компресії Gzip, Brotli та Zstandard у контексті їх застосування для оптимізації передачі веб-контенту. Основні результати роботи:

1. Проаналізовано теоретичні основи трьох алгоритмів: Gzip (DEFLATE = LZ77 + кодування Хаффмана), Brotli (LZ77 + контекстне моделювання + вбудований статичний словник 120 КБ), Zstandard (LZ77 + FSE + адаптивні словники). Визначено ключові архітектурні відмінності, що обумовлюють різницю у продуктивності та ефективності стиснення.

2. Розроблено автоматизовану систему тестування на базі Node.js, Angular 20 та SQLite, що поєднує серверне профілювання компресії з браузерним тестуванням через Playwright у трьох рушіях - Chromium, Firefox та WebKit - із емуляцією мережевих профілів (4G, Fast 3G, DSL).

3. Встановлено кількісні характеристики алгоритмів: Brotli-6 забезпечує найкращий коефіцієнт стиснення для всіх досліджених текстових типів контенту; Zstd-1/3/5 формує швидку частину Парето-межі, зокрема Zstd-3 має медіанну швидкість компресії 182,0 МБ/с у корпусі RQ2; максимальні рівні Brotli-11 та Zstd-22 доцільні для попереднього стиснення статичних ресурсів.

4. Браузерне тестування трьох production-збірок підтвердило практичний вплив стиснення на LCP: у надійних рядках медіанний виграш становив 629 мс для Gzip-6, 635 мс для Brotli-6 та 882 мс для Zstd-3. Для інтерпретації цього ефекту LCP розкладено на спостережувані канали TTFB, Download, Blocking та Render-residual; результати показали, що після скорочення мережевого етапу вузьким місцем часто стає rendering pipeline конкретного застосунку. Також визначено обмеження вимірювального середовища: Firefox придатний для порівняння загального LCP, але не для точної декомпозиції мережевих фаз, а Playwright WebKit на Windows не дозволив автоматизовано перевірити Zstd.

5. Сформульовано практичні рекомендації щодо вибору алгоритму: Zstd-3 або Zstd-5 для динамічного текстового контенту за наявності підтримки клієнта, Brotli-11 з Gzip-6 fallback для статичних ресурсів, опційний Zstd-22 для сучасних

клієнтів, обов'язковий заголовок `Vary: Accept-Encoding` та окреме тестування Safari/macOS перед production-впровадженням Zstd.

Список використаної літератури

1. HTTP Archive. Page Weight. In The 2025 Web Almanac (Chapter 14). Published January 15, 2026. https://almanac.httparchive.org/en/2025/page-weight

2. Fielding R., Nottingham M., Reschke J. (Eds.). HTTP Semantics. RFC 9110. IETF, June 2022. https://www.rfc-editor.org/rfc/rfc9110

3. Grigorik I. High Performance Browser Networking: What Every Web Developer Should Know About Networking and Web Performance. O'Reilly Media, 2013. 398 p. ISBN 978-1-449-34476-4.

4. Google/SOASTA Research. Mobile page speed benchmarks, 2017. As cited in Think with Google (2018). Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed. https://www.thinkwithgoogle.com/data/mobile-bounce-rate-statistics/

5. Thomson M., Benfield C. (Eds.). HTTP/2. RFC 9113. IETF, June 2022. https://www.rfc-editor.org/rfc/rfc9113; Bishop M. (Ed.). HTTP/3. RFC 9114. IETF, June 2022. https://www.rfc-editor.org/rfc/rfc9114

6. W3Techs. Usage Statistics of Compression for Websites, April 2026. https://w3techs.com/technologies/details/ce-compression [Accessed May 12, 2026].

7. Google. Web Vitals: Essential metrics for a healthy site. web.dev, 2020. Updated 2024. https://web.dev/articles/vitals

8. Salomon D., Motta G., Bryant D. Handbook of Data Compression. 5th ed. Springer, 2010. 1361 p. ISBN 978-1-84882-902-2.

9. Ziv J., Lempel A. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory. 1977. Vol. 23, No. 3. P. 337–343. DOI: 10.1109/TIT.1977.1055714.

10. Ziv J., Lempel A. Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory. 1978. Vol. 24, No. 5. P. 530–536. DOI: 10.1109/TIT.1978.1055934.

11. Huffman D. A. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the IRE. 1952. Vol. 40, No. 9. P. 1098–1101. DOI: 10.1109/JRPROC.1952.273898.

12. Duda J. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. arXiv:1311.2540, 2013. https://arxiv.org/abs/1311.2540 [препринт, не рецензовано].

13. Alakuijala J., Szabadka Z. Brotli Compressed Data Format. RFC 7932. IETF, July 2016. https://www.rfc-editor.org/rfc/rfc7932

14. Deutsch P. GZIP file format specification version 4.3. RFC 1952. IETF, May 1996. https://www.rfc-editor.org/rfc/rfc1952

15. Deutsch P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. IETF, May 1996. https://www.rfc-editor.org/rfc/rfc1951

16. Alakuijala J., Kliuchnikov E., Szabadka Z., Vandevenne L. Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms. Google, Inc., Technical Report, September 2015. http://www.gstatic.com/b/brotlidocs/brotli-2015-09-22.pdf [не рецензовано].

17. Collet Y., Kucherawy M. (Ed.). Zstandard Compression and the 'application/zstd' Media Type. RFC 8878. IETF, February 2021. https://www.rfc-editor.org/rfc/rfc8878

18. Meta (Facebook). Zstandard - Fast real-time compression algorithm. Reference implementation and documentation. https://github.com/facebook/zstd

19. Caniuse. Brotli Accept-Encoding/Content-Encoding. Chrome 50+ (квітень 2016), Firefox 44+ (січень 2016), Edge 15+ (квітень 2017), Safari 11+ (вересень 2017). Глобальна підтримка: 95,65% станом на квітень 2026. https://caniuse.com/brotli; Zstandard content-encoding. https://caniuse.com/zstd

20. Chrome for Developers. Chrome 123 beta: zstd Content-Encoding. February 2024. https://developer.chrome.com/blog/chrome-123-beta

21. Nginx Documentation. Compression modules: gzip, Brotli and Zstandard. https://docs.nginx.com/nginx/admin-guide/web-server/compression/; https://nginx.org/en

/docs/http/ngx_http_gzip_module.html; https://docs.nginx.com/nginx/admin-guide/dynamic-modules/brotli/; https://github.com/tokers/zstd-nginx-module

22. Apache Software Foundation. mod_brotli. Apache HTTP Server Version 2.4. Доступний з версії 2.4.26 (липень 2017). https://httpd.apache.org/docs/2.4/mod/mod_brotli.html

23. Node.js Documentation. Zlib. https://nodejs.org/api/zlib.html

24. Cloudflare. Announcing support for Zstandard compression. Cloudflare Blog, 2024. https://blog.cloudflare.com/new-standards/

25. Amazon Web Services. Serving compressed files. Amazon CloudFront Developer Guide. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/ServingCompressedFiles.html

26. Souders S. High Performance Web Sites: Essential Knowledge for Front-End Engineers. O'Reilly Media, 2007. 168 p. ISBN 978-0-596-52930-7.

27. Express.js. Compression middleware. https://expressjs.com/en/resources/middleware/compression.html; npm: https://www.npmjs.com/package/compression

28. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 560 p. ISBN 978-0-321-12742-6.

29. Socket.IO. Socket.IO Documentation (v4). https://socket.io/docs/v4/

30. Hipp R. D. SQLite Documentation. https://www.sqlite.org/docs.html

31. better-sqlite3. The fastest and simplest library for SQLite3 in Node.js. https://github.com/WiseLibs/better-sqlite3

32. Angular. Angular Developer Documentation. https://angular.dev

33. Tailwind CSS. Utility-First CSS Framework. https://tailwindcss.com/

34. ng2-charts. Beautiful charts for Angular based on Chart.js. https://github.com/valor-software/ng2-charts

35. RxJS. Reactive Extensions Library for JavaScript. https://rxjs.dev/

36. MongoDB. @mongodb-js/zstd - Zstandard compression library for Node.js. https://github.com/mongodb-js/zstd

37. Microsoft. Playwright Documentation. https://playwright.dev

38. Google. web-vitals - Essential metrics for a healthy site. https://github.com/GoogleChrome/web-vitals

39. W3C. Performance Timeline Level 2. W3C Candidate Recommendation Draft, May 21, 2025. https://www.w3.org/TR/performance-timeline/

40. Google. Evaluating page experience for a better web. Google Search Central Blog, May 28, 2020. https://developers.google.com/search/blog/2020/05/evaluating-page-experience

41. Meenan P., Weiss Y. (Eds.). Compression Dictionary Transport. RFC 9842. IETF, September 2025. https://www.rfc-editor.org/rfc/rfc9842

42. Intel. Intel QuickAssist Technology. https://www.intel.com/content/www/us/en/developer/topic-technology/open/quick-assist-technology/overview.html

Глосарій

ANS - метод ентропійного кодування (Asymmetric Numeral Systems).

Brotli - алгоритм стиснення без втрат (RFC 7932).

CDN - мережа доставки контенту (Content Delivery Network).

CLS - метрика візуальної стабільності (Cumulative Layout Shift, Web Vitals).

Content-Encoding - HTTP-заголовок, що визначає алгоритм стиснення відповіді.

DEFLATE - алгоритм стиснення на основі LZ77 та кодування Хаффмана (RFC 1951).

FCP - First Contentful Paint - час до першого відображення контенту.

FSE - Finite State Entropy - метод ентропійного кодування.

Gzip - формат стиснення на основі DEFLATE (RFC 1952).

LCP - Largest Contentful Paint - час до відображення найбільшого елемента.

LZ77 - словниковий алгоритм стиснення (Зів, Лемпель, 1977).

Playwright - фреймворк автоматизації браузерів (Microsoft).

TTFB - Time to First Byte - час до отримання першого байта відповіді.

Web Vitals - ключові метрики вебпродуктивності (Google).

Zstandard - алгоритм стиснення від Meta (RFC 8878).

Додатки

Додаток А
(довідковий)

ER-модель бази даних системи тестування

На рисунку А.1 наведено логічну ER-модель бази даних системи тестування з ключовими таблицями та зв’язками між ними.

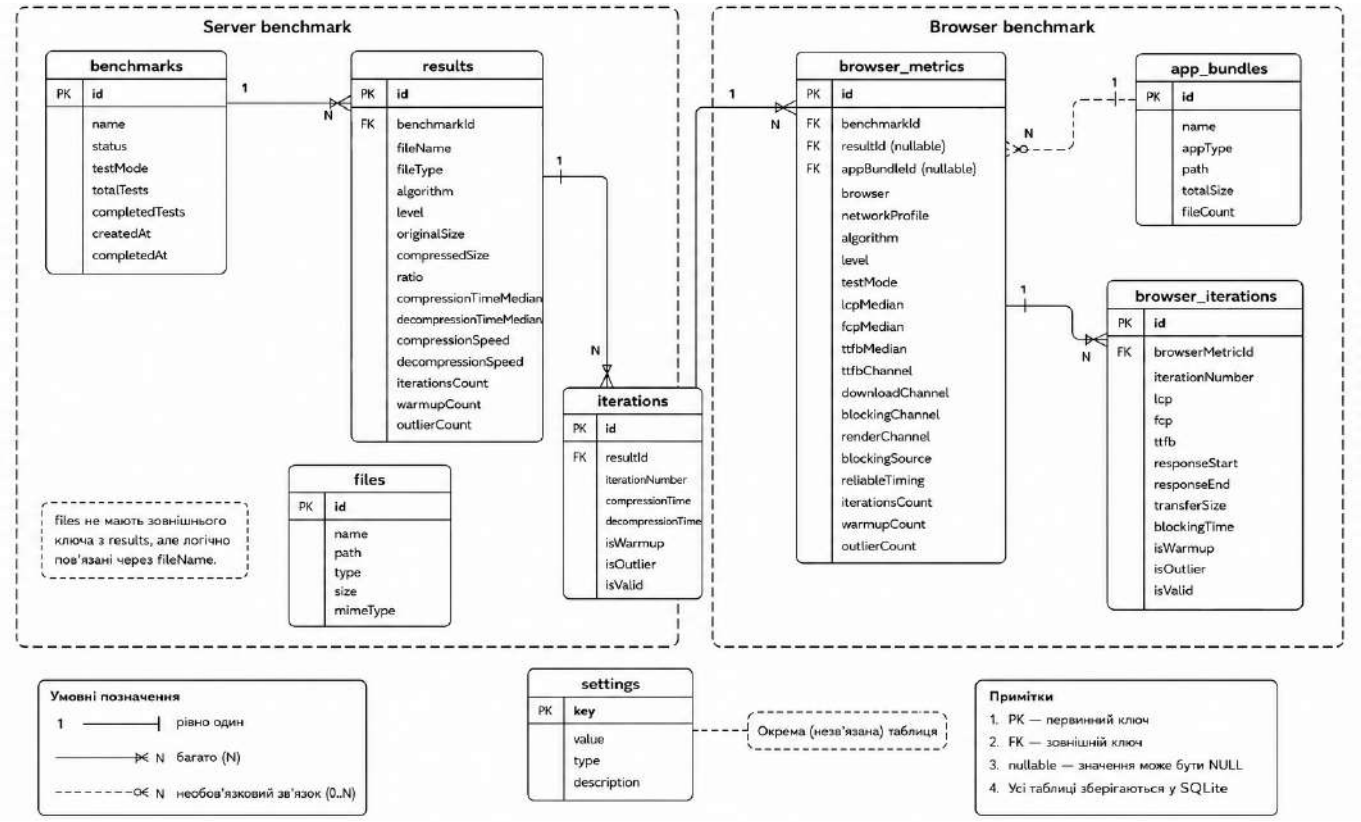

Рисунок А.1 - ER-модель бази даних системи тестування алгоритмів компресії

Додаток Б
(довідниковий)

Склад тестового корпусу веб-ресурсів

У таблиці Б.1 наведено текстові файли локального корпусу, використані у серверних експериментах RQ1–RQ3. Позначення RQ3-base означає базову log-log регресію на повному корпусі, а RQ3-level - додатковий контрольований запуск на чотирьох JavaScript-файлах з різними рівнями компресії. Окремо у таблиці Б.2 наведено production-збірки, використані у браузерному real-app тестуванні RQ4.

Табл. Б.1 - Склад серверного корпусу веб-ресурсів

Тип	Файл	Розмір, КБ	Використання
CSS	`animate.min.css`	70,1	RQ1, RQ3-base
CSS	`bootstrap-5.css`	274,5	RQ1, RQ2, RQ3-base
CSS	`bootstrap-5.min.css`	227,3	RQ1, RQ3-base
CSS	`bulma.css`	745,3	RQ1, RQ3-base
CSS	`bulma.min.css`	661,4	RQ1, RQ3-base
CSS	`foundation.min.css`	133,1	RQ1, RQ3-base
CSS	`normalize.css`	6,0	RQ1, RQ3-base
CSS	`pure.css`	25,6	RQ1, RQ3-base
CSS	`spectre.min.css`	45,6	RQ1, RQ3-base
CSS	`tailwind-base.css`	2865,3	RQ1, RQ3-base
HTML	`caniuse.html`	9,5	RQ1, RQ2, RQ3-base
HTML	`github-explore.html`	322,2	RQ1, RQ3-base
HTML	`hackernews.html`	34,0	RQ1, RQ3-base
HTML	`mdn-home.html`	117,2	RQ1, RQ3-base
HTML	`mdn-js-guide.html`	193,4	RQ1, RQ3-base
HTML	`reddit-programming.html`	123,5	RQ1, RQ3-base
HTML	`stackoverflow-tags.html`	168,1	RQ1, RQ3-base
HTML	`web-dev-vitals.html`	104,9	RQ1, RQ3-base
HTML	`wikipedia-compression.html`	388,3	RQ1, RQ3-base
HTML	`wikipedia-web.html`	484,4	RQ1, RQ2, RQ3-base
JavaScript	`axios.min.js`	52,8	RQ1, RQ3-base
JavaScript	`chart.umd.js`	200,8	RQ1, RQ3-base, RQ3-level
JavaScript	`d3.v7.min.js`	273,2	RQ1, RQ3-base
JavaScript	`jquery.min.js`	85,5	RQ1, RQ3-base
JavaScript	`lodash.min.js`	71,3	RQ1, RQ2, RQ3-base, RQ3-level
JavaScript	`moment.min.js`	57,5	RQ1, RQ3-base
JavaScript	`react.production.min.js`	10,5	RQ1, RQ3-base, RQ3-level
JavaScript	`react-dom.production.min.js`	128,7	RQ1, RQ3-base
JavaScript	`rxjs.umd.min.js`	86,0	RQ1, RQ3-base

Продовження таблиці Б.1

Тип	Файл	Розмір, КБ	Використання
JavaScript	three.min.js	654,2	RQ1, RQ2, RQ3-base, RQ3-level
JavaScript	vue.global.prod.js	154,2	RQ1, RQ3-base
JSON	angular-package.json	3,1	RQ1, RQ3-base
JSON	geojson-counties.json	673,6	RQ1, RQ2, RQ3-base
JSON	geojson-states.json	111,9	RQ1, RQ3-base
JSON	github-emojis.json	167,6	RQ1, RQ3-base
JSON	github-licenses.json	2,0	RQ1, RQ3-base
JSON	npm-react-full.json	6478,1	RQ1, RQ3-base
JSON	openapi-petstore.json	16,7	RQ1, RQ3-base
JSON	react-package.json	1,6	RQ1, RQ3-base
JSON	tsconfig-schema.json	422,4	RQ1, RQ3-base
JSON	typescript-package.json	3,9	RQ1, RQ3-base
JSON	webpack-package.json	8,8	RQ1, RQ3-base
SVG	bootstrap-icons.svg	1072,4	RQ1, RQ2, RQ3-base
SVG	feather-sprite.svg	58,9	RQ1, RQ3-base
SVG	heroicon-globe.svg	0,7	RQ1, RQ3-base
SVG	icon-github.svg	0,8	RQ1, RQ3-base
SVG	icon-react.svg	2,9	RQ1, RQ3-base
SVG	icon-typescript.svg	1,3	RQ1, RQ3-base
SVG	logo-google.svg	0,4	RQ1, RQ3-base
SVG	logo-x-twitter.svg	0,2	RQ1, RQ3-base
SVG	lucide-sprite.svg	364,5	RQ1, RQ3-base

Табл. Б.2 - Production-збірки для real-app тестування RQ4

Застосунок	Фреймворк	Файлів	Розмір, КБ	Використання
Дослідницька система	Angular	43	1595	RQ4
Dashboard	React/Vite	3	436	RQ4
Dashboard	Vue/Vite	3	250	RQ4