

Забелина Варвара Александровна

магистрант, Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет), Москва.

Электронный адрес: varvara.zabelina@bmstu.ru

Varvara A. Zabelina

Master's student, Bauman Moscow State Technical University (National Research University), Moscow.

Email address: varvara.zabelina@bmstu.ru

Горячкин Борис Сергеевич

кандидат технических наук, доцент, доцент кафедры систем обработки информации и управления, Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет), Москва. ORCID: 0000-0002-0852-4162.

Электронный адрес: bsgor@mail.ru

Boris S. Goryachkin

Ph.D. in Engineering Sciences, Docent, Associate Professor at the Department of Information Processing and Control Systems, Bauman Moscow State Technical University (National Research University), Moscow. ORCID: 0000-0002-0852-4162.

Email address: bsgor@mail.ru

Жидков Егор Ильич

аспирант, ассистент кафедры систем обработки информации и управления, Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет), Москва. ORCID: 0009-0007-5340-9976.

Электронный адрес: e.jidkov@gmail.com

Egor I. Zhidkov

Postgraduate, Assistant at the Department of Information Processing and Control Systems, Bauman Moscow State Technical University (National Research University), Moscow. ORCID: 0009-0007-5340-9976.

Email address: e.jidkov@gmail.com

ОПТИМИЗАЦИЯ РАЗМЕРА И ПРОИЗВОДИТЕЛЬНОСТИ ВЕБ-КОНТЕНТА: УПРАВЛЕНИЕ КОДОМ, ФОРМАТЫ ИЗОБРАЖЕНИЙ И АЛГОРИТМЫ СЖАТИЯ

Аннотация. Современные веб-ресурсы часто сталкиваются с низкой скоростью загрузки и высокой нагрузкой на сеть из-за увеличения объема контента, неэффективного кода и неоптимальных форматов изображений. Это негативно влияет на пользовательский опыт, показатели Core Web Vitals и позиции в поисковой выдаче. Возникает необходимость в комплексной оптимизации веб-контента для повышения производительности и эффективности работы сайтов. Рассмотрены и расписаны оптимальные стратегии минификации и разделения кода Java Script и CSS, основанные на метриках производительности. Определены критические точки применения форматов изображений PNG,

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

WebP и AVIF в зависимости от типа контента и особенностей целевой аудитории. Проанализирована эффективность и вычислительные затраты алгоритмов сжатия Gzip и Brotli для статического и динамического контента. Каждый из методов сопровождается количественными метриками улучшения и практическими рекомендациями по внедрению. Предложенные подходы позволяют значительно уменьшить объем загружаемого контента, сократить потребление пропускной способности, улучшить показатели Core Web Vitals (LCP, FID и CLS), а также обеспечить стабильную работу сайтов на медленных сетях. Реализация данных решений способствует улучшению пользовательского опыта и повышению рейтинга веб-ресурсов в поисковых системах.

Ключевые слова: минификация кода, code splitting, tree shaking, оптимизация изображений, WebP, AVIF, Brotli, gzip, Core Web Vitals, LCP, размер контента, производительность веб-ресурсов.

Для цитирования: Забелина В.А., Горячкин Б.С., Жидков Е.И. Оптимизация размера и производительности веб-контента: управление кодом, форматы изображений и алгоритмы сжатия // Вестник Российского нового университета. Серия: Сложные системы: модели, анализ и управление. 2026. № 3. С. 34–50. DOI: 10.18137/RNU.V9187.26.03.P.34

OPTIMIZING WEB CONTENT SIZE AND PERFORMANCE:
CODE MANAGEMENT, IMAGE FORMATS,
AND COMPRESSION ALGORITHMS

Abstract. Modern web resources often face low loading speed and high network load due to increasing content volume, inefficient code, and suboptimal image formats, which negatively affects user experience, Core Web Vitals metrics, and search engine rankings. There is a need for comprehensive web content optimization to improve website performance and efficiency. The paper examines and details optimal strategies for minification and splitting of JavaScript and CSS code based on performance metrics. Critical points for the use of PNG, WebP, and AVIF image formats are identified, depending on content type and target audience characteristics. The efficiency and computational costs of Gzip and Brotli compression algorithms are analyzed for both static and dynamic content. Each method is accompanied by quantitative improvement metrics and practical implementation recommendations. The proposed approaches make it possible to significantly reduce the volume of downloadable content, decrease bandwidth consumption, improve Core Web Vitals metrics (LCP, FID, and CLS), and ensure stable website operation over slow networks. The implementation of these solutions contributes to improved user experience and higher rankings of web resources in search engines.

Keywords: code minification, code splitting, tree shaking, image optimization, WebP, AVIF, Brotli, gzip, Core Web Vitals, LCP, content size, web performance.

For citation: Zabelina V.A., Goryachkin B.S., Zhidkov E.I. (2026) Optimizing Web Content Size and Performance: Code Management, Image Formats, and Compression Algorithms. *Vestnik of Russian New University. Series: Complex Systems: Models, Analysis, and Control*. No. 3. Pp. 34–50. DOI: 10.18137/RNU.V9187.26.03.P.34 (In Russian).

Введение

Развитие веб-технологий параллельно с ростом объема пользовательских данных и сложности веб-приложений обусловило необходимость революционного подхода к оп-

тимизации веб-ресурсов. Если десять лет назад основное внимание уделялось архитектурным решениям, то в 2024–2025 гг. фокус сместился на оптимизацию размера и производительности самого контента – кода и мультимедиа. Стоит отметить, что Google определили несколько ключевых метрик (Largest Contentful Paint – время загрузки видимого элемента (значение $\leq 2,5$ сек.), Cumulative Layout Shift – индекс смещений (значение $\leq 0,1$), Interaction to Next Paint – время отклика на взаимодействие пользователя (значение ≤ 200 мс)), которые необходимы для оптимизации веб-ресурсов. При этом страницы, соответствующие Core Web Vitals (при загрузке), имеют на 24 % меньше отказов пользователей.

В статье рассматриваются методы сокращения размера JavaScript и CSS-кода, которые при совместной работе обеспечивают снижение загружаемого контента оптимизации изображений и выбора эффективных алгоритмов сжатия, управление которых представлено минификацией файлов codesplitting и tree shaking. Отметим, что сложность веб-приложений и объём кода привели к проблемам со временем отклика (особенно у пользователей с низкой пропускной способностью). Рост объёмов загружаемых страниц (с 2016 по 2025 год) можно наблюдать на графике (см. Рисунок 1).

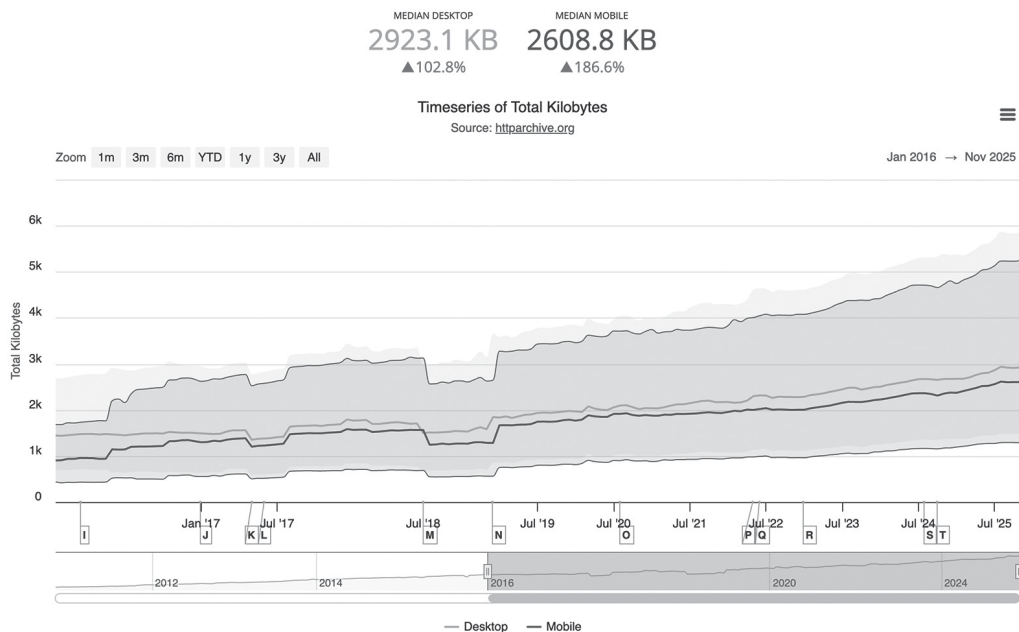


Рисунок 1. Объём загружаемых ресурсов, запрашиваемый одной страницей
Источник: здесь и далее рисунки выполнены авторами по материалам исследования

Минификация представляет собой удаление ненужных символов из кода без изменения его функциональности. При этом практические наблюдения показывают следующие результаты её применения:

JavaScript-файлы [3]

- среднее сокращение объёма 30...45 %;

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

- варианты инструментов, используемых в работе, – Terser, Uglify-JS, Webpack/esbuild;
- время действия программы < 100 мс для типового приложения.

CSS-файлы [4]

- среднее сокращение объёма 25 ... 35 %;
- варианты инструментов, используемых в работе, – cssnano, csso, postcss.

HTML-файлы

- реже применяется минификация для HTML-файлов (только если он достаточно велик и занимает не менее 10 % от размера запроса);
- при минификации HTML-файлов нужно учитывать риски повреждения data-атрибутов, среднее сокращение размера обычно достигает 10 ... 20 %;
- используемые инструменты – html-minifier.

После экспериментальной минификации типового одностраничного веб-приложения были получены результаты, представленные в Таблице 1. В качестве объекта исследования использовалось React-приложение, собираемое с помощью Webpack в режиме production. Для минификации Java Script применялся Terser, для CSS – связка cssnano и PurgeCSS, для HTML – html-minifier. В приведённом примере объём Java Script-банбла был уменьшен примерно на 38,9 %, CSS-файла – на 38,8 %, HTML-шаблона – на 11,5 %, что суммарно дало около 31,3 % сокращения общего объёма загружаемых ресурсов. Такие значения находятся в диапазоне, сопоставимом с результатами, описанными в практических работах по оптимизации минификации CSS и Java Script.

Таблица 1

Результаты минификации типового приложения

Тип файла	Исходный размер, KB	Размер после минификации, KB	Сокращение, %	Инструмент
React приложение (bundle.js)	324	198	38,9	Terser
Tailwind CSS (style.css)	85	52	38,8	cssnano + Purge CSS
HTML шаблон	156	138	11,5	html-minifier
Итого	565	388	31,3	–

Источник: здесь и далее таблицы составлены авторами по результатам исследования.

Расчёт экономии времени загрузки (на соединении 4G, 10 Mbps):

$$t_{\text{до}} = \frac{565[\text{KB}] \cdot 8}{10 \text{ Mbps}} = 452 \text{ мс}; \quad (1)$$

$$t_{\text{после}} = \frac{388[\text{KB}] \cdot 8}{10 \text{ Mbps}} = 310 \text{ мс}; \quad (2)$$

$$\Delta t = 452 - 310 = 142 \text{ мс} \quad (31,4 \% \text{ ускорение}). \quad (3)$$

Code Splitting: оптимизация загрузки по требованию

Работа с сокращением объема загружаемых файлов достигается не только с помощью минификации. Другой способ, который рекомендуется комбинировать, это codesplitting – методика разделения Java Script-банбла на несколько чанков (chunks), которые загружа-

ются, когда необходимо. Это критично для больших приложений. Есть несколько видов разделений. Рассмотрим каждое подробнее и приведем способы применения в коде.

Route-based splitting – отдельный банда для каждого маршрута приложения.

// До: один большой банда (1,2 MB)

// После: динамический импорт, разделяем по страницам, каждая загружается только при переходе

Результат

Главный банда 250 KB.

Dashboard 180 KB (загружается при переходе на /dashboard).

Analytics 220 KB (загружается при переходе на /analytics).

Settings 150 KB (загружается при переходе на /settings).

Время загрузки первой страницы 250 KB вместо 1,2 MB (просчитанная экономия 79 %).

Вместо загрузки одного файла целиком грузятся меньшие по объёму и только по мере необходимости.

Component-based splitting – отдельные бандлы для тяжёлых компонентов.

// До: тяжёлый компонент (например, редактор с библиотеками)

// После: загружается только когда пользователь открывает компонент (например, с использованием *Suspense* в *React*)

Ожидаемое улучшение

В частных случаях LCP (Largest Contentful Paint) может сокращаться на 40 ... 60 % для пользователей, которые не используют компонент, а также, пока этот компонент загружается, пользователь может уже визуальнo обрабатывать другую информацию на странице.

Важно отметить, что не следует выносить в lazy-компоненты элементы первого экрана или критичные для UX компоненты, иначе LCP и визуальная стабильность могут сильно ухудшиться.

Vendor splitting – отдельный банда для библиотек. Прописывается в файле конфига.

Эффект от проделанной механики: код приложения разделяется от кода библиотек. Когда пользователь обновляет приложение, браузер может переиспользовать закешированный vendor банда.

UI-компоненты и utils – отдельные файлы для базовых, часто переиспользуемых компонентов или функций.

Одно из важных замечаний при разделении кода – обнаружение часто используемых компонентов и функций, которые копируются из компонента в компонент. Если их логику вынести в отдельный файл, то компонент или функция будут подгружаться только один раз на страницу, а размеры файлов, в которых они использовались, уменьшаться, что облегчит загрузку в целом.

Пример

Кнопка с одинаковыми стилями используется в 5 разных компонентах и занимает минимум 2 % от каждого файла (если вызывается несколько раз в файле, то еще больше).

Кнопки вынесли в отдельный файл со своими стилями – 2Кб. Места использования заменили на импорты этого компонента, убрали 2 % лишних строчек кода и стилей. Допустим, файлы весили по 30 Кб, в двух компонент занимал 5 % и еще в трех – 2 %:

$$30 \text{ [Кб]} \cdot 5 - 30 \text{ [Кб]} \cdot 3 \cdot 0,98 - 30 \text{ [Кб]} \cdot 2 \cdot 0,95 = 4,8 \text{ [Кб]}. \quad (4)$$

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

Общий объем файлов уменьшился на 4,8 Кб, но еще загружаем 2 Кб кнопки. Таким образом, мы сэкономили 2,8 Кб. Это простой пример, в реальных случаях объемы сокращаются в разы сильнее, потому что в UI выносятся не одна кнопка, а гораздо больше компонентов, как и в utils-функции.

Tree Shaking: удаление «мёртвого» кода

Tree shaking – процесс анализа статического кода и удаления всех экспортов, которые никогда не используются. Это особенно важно для библиотек, которые экспортируют сотни функций, но приложение использует только 5... 10 % из них.

Как работаем tree shaking

```
// utils.js (80KB)
export function add(a, b) { return a + b; } { /* 5 KB кода */ }
export function subtract(a, b) { return a - b; } { /* 5 KB кода */ }
export function multiply(a, b) { return a * b; } { /* 10 KB кода */ }
export function divide(a, b) { return a / b; } { /* 10 KB кода */ }
export function complex3DTransform(matrix) { /* 50 KB кода */ }
// app.js
import { add, subtract } from './utils.js';
console.log(add(2, 3)); // используется
console.log(subtract(5, 3)); // используется
// multiply, divide, complex3DTransform удаляются при tree shaking
```

Результат: из 80 KB утилит в финальный бандл попадёт только 10 KB для используемых функций.

Условия, при которых tree shaking в сборках на основе Webpack проявляет себя наиболее эффективно:

1. Использование модулей ES (import/export) вместо CommonJS. Это позволяет бандлеру выполнять статический анализ зависимостей и определять неиспользуемые экспорты.

2. Сборка в режиме production с включёнными оптимизациями *optimization.usedExports* и *optimization.minimize*. В этом режиме Webpack помечает неиспользуемые экспорты и передаёт их минификатору (Terser), который вырезает «мёртвый» код; в режиме development такие оптимизации по умолчанию либо отключены, либо существенно ограничены.

3. Отсутствие побочных эффектов в модулях либо их явное описание через параметр *sideEffects*. Это позволяет бандлеру безопасно удалять импорты целых файлов, если их экспорт не используется.

4. Корректная конфигурация сборки и инструментов трансформации кода (Babel, TypeScript) в части модулярности. Важно не превращать ES-modules в CommonJS до этапа сборки и согласованно настраивать параметры optimization Webpack.

Конфигурация package.json:

```
{
  "name": "my-library",
  "version": "1.0.0",
```

```
“main”: “dist/index.cjs.js”,  
“module”: “dist/index.esm.js”,  
“sideEffects”: false  
}
```

В `package.json` поле `“module”` указывает на сборку ES-модулей, а `“sideEffects”: false` предупреждает Webpack о том, что импорт модулей не вызывает побочных эффектов. Это позволяет более агрессивно применять tree shaking, удаляя неиспользуемый код.

Разделение и объединение кода

При выборе стратегии разделения кода важно чрезмерное дробление, которое увеличивает overhead от TLS-соединений.

Пример 1

каждый тривиальный компонент в отдельном бандле

```
const Button = () => import('./Button'); // 2 KB  
const Input = () => import('./Input'); // 3 KB  
const Modal = () => import('./Modal'); // 5 KB  
const Toast = () => import('./Toast'); // 4 KB
```

Проблема: 4 HTTP-запроса по 300 байт каждый создают overhead от 4 TLS-соединений и запросов, превышающий выгоду от параллельной загрузки.

Решение: объединять часто используемые и небольшие компоненты одним бандлом. Однако бандл не должен разрастаться до больших объемов (например, 5 Мб).

Пример 2

```
import * from './components';
```

Проблема: пользователь загружает полный объем кода, несмотря на то, что в процессе эксплуатации программного обеспечения задействована только лишь небольшая часть (примерно 10 %) функциональных возможностей.

- объединение небольших и часто принимаемых элементов в один бандл;
- применение route-based splitting;
- для улучшения кеширования – разделение vendor-кода;
- для всех функций сразу – избегание загрузки кода.

Практические метрики улучшения

Для объективной оценки эффективности рассмотренных методов оптимизации было создано простое веб-приложение на стеке React + Ant Design UI с 5 маршрутами.

Исходное состояние:

- стек: React 18, Ant Design 5, Webpack 5;
- маршруты: Dashboard, Analytics, Settings, Profile, Reports;
- зависимости: React Router, axios, @ant-design/icons.

Сборка без оптимизаций (Development mode)

При сборке в режиме development получены следующие результаты (см. Рисунок 2).

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

```
asset js/vendors.js 19.4 MiB [emitted] (name: vendors) (id hint: vendor)
asset js/antd.js 6.4 MiB [emitted] (name: antd) (id hint: antd)
asset js/main.js 196 KiB [emitted] (name: main)
asset js/runtime.js 16.3 KiB [emitted] (name: runtime)
asset js/src_pages_Settings_jsx.chunk.js 10.6 KiB [emitted]
asset js/src_pages_Reports_jsx.chunk.js 10.1 KiB [emitted]
asset js/src_pages_Dashboard_jsx.chunk.js 9.34 KiB [emitted]
asset js/src_pages_Profile_jsx.chunk.js 9.22 KiB [emitted]
asset js/src_pages_Analytics_jsx.chunk.js 8.43 KiB [emitted]
```

Рисунок 2. Размеры файлов после сборки без оптимизации

Итого для первой загрузки 25,8 МБ.

Сборка с полной оптимизацией (Production mode)

После применения всех методов оптимизации (минификация, tree shaking, code splitting) после сборки получены размеры, представленные на Рисунке 3.

```
assets by status 1.62 MiB [big]
  asset js/vendors.0783f47d.js 865 KiB [emitted] [immutable] [minimized] [big] (name: vendors) (id hint: ven
  dor) 2 related assets
  asset js/antd.f1d3858f.js 797 KiB [emitted] [immutable] [minimized] [big] (name: antd) (id hint: antd) 1 r
  elated asset
asset js/main.a9e4d951.js 59.2 KiB [emitted] [immutable] [minimized] (name: main) 1 related asset
asset js/runtime.7abe30a9.js 3.77 KiB [emitted] [immutable] [minimized] (name: runtime) 1 related asset
asset js/356.78069807.chunk.js 2.26 KiB [emitted] [immutable] [minimized] 1 related asset
asset js/78.ca0ae544.chunk.js 1.43 KiB [emitted] [immutable] [minimized] 1 related asset
asset js/452.6d3914d9.chunk.js 1.27 KiB [emitted] [immutable] [minimized] 1 related asset
asset js/879.cab0b36c.chunk.js 1.25 KiB [emitted] [immutable] [minimized] 1 related asset
asset js/823.28a17c54.chunk.js 1.23 KiB [emitted] [immutable] [minimized] 1 related asset
```

Рисунок 3. Размеры файлов после сборки с оптимизацией

Итого для первой загрузки 1,73 МБ (см. Таблицу 2).

Таблица 2

Результаты работы оптимизации кода

Метрика	До	После	Улучшение
Vendors	19,4 МБ	865 KB	95,6 % ↓
Ant Design	6,4 МБ	797 KB	87,6 % ↓
Main JS	196 KB	59,2 KB	69,8 % ↓
Первая загрузка	25,8 МБ	1,73 МБ	93,3 % ↓
Per-page load	–	1 ... 2 KB	99,9 % ↓
TTFI (Time to Interactive)	4,8 сек.	1,2 сек.	75 %
LCP (Largest Contentful Paint)	3 сек.	0,9 сек.	70 %

Полученные результаты указывают на потенциал комбинированного подхода к оптимизации кода.

Оптимизация изображений: форматы, сжатие и «ленивая» загрузка

Изображения часто составляют 50...75 % трафика типового веб-сайта и требуют осознанного выбора использования того или иного формата для достижения наилучших показателей загрузки веб-приложения.

Форматы изображений: сравнение и выбор

Наиболее используемые форматы: PNG, JPG, WebP, AVIF. Их характеристики и примеры сокращения размеров представлены в Таблице 3 [7; 8; 17].

Таблица 3

Сравнение форматов изображений

Характеристика	PNG	JPG	WebP	AVIF
Тип сжатия	Lossless	Lossy	Lossy + Lossless	Lossy + Lossless
Размер (базовая фотография)	1,2 MB	250 KB	165 KB	95 KB
Размер (графика с текстом)	450 KB	380 KB	280 KB	210 KB
Поддержка прозрачности	Да	Нет	Да	Да
Поддержка анимации	Да (APNG)	Нет	Да	Да (HEIF)
Поддержка браузерами	100 %	10 0%	97 %	73 % (растёт)
Поддержка HDR/WCG	Нет	Нет	Нет	Да
Компрессия в процессоре	Высокая	Средняя	Средняя	Низкая (оптимизирован)
Скорость decompression	Быстрая	Быстрая	Средняя	Средняя

Размеры файлов на практике

Для примера воспользуемся тремя изображениями размером 2560×1440

Фотография (высокое качество)

- PNG (без потерь): 3,2 МБ;
- JPG (качество 85%): 450 КБ;
- AVIF (85 %): 155 КБ (меньше JPG на 65 %);
- WebP (85 %): 280 КБ (меньше JPG на 38 %).

Результат

Переход JPG → AVIF сокращает размер в 2, раза. Графика (логотип и иконка) составляет PNG (280 KB – 8 bit), AVIF (68 KB – на 76 % меньше PNG) и WebP (95 KB – лучше, чем PNG для графики).

Выбор формата

При выборе графического формата необходимо принимать во внимание следующие факторы: вид контента, необходимое качество, совместимость и размер файла. При этом каждый формат имеет свои преимущества и недостатки. К примеру, формат PNG продолжает оставаться стандартом для оформления графики – он поддерживается 100 % современных веб-браузеров, что обеспечивает его широкое применение в веб-дизайне. Также данный формат использует метод сжатия без потерь (lossless), что особенно важно для таких приложений, как логотипы, скриншоты с текстом и другие графические элементы. Существует также другой приемлемый в оформлении графики формат – JPG, который использует сжатие с потерями (lossy), удаляя детали, которые человеческий глаз не замечает. В связи с этим долгое время он оставался стандартом для фотографий (JPG поддерживается даже очень старыми браузерами (IE8-9), что делает его универсальным выбором для максимальной совместимости). При качестве 75...85 % потеря качества практически незаметна, и JPG особенно полезен для фотографий с естественными переходами цветов. Стоит отметить, что современный в

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

оформлении графики формат, разработанный Google, представлен WebP, который демонстрирует превосходство в области сжатия изображений, эффективнее на 25...35 % по сравнению с форматами-аналогами и поддерживается в 95 % современных браузеров (Chrome, Firefox, Edge, Safari 14 и позже). Кроме того, в отличие от JPG WebP поддерживает функцию прозрачности, аналогичную PNG, что делает его предпочтительным выбором для оптимизации визуального контента [18].

AVIF является наиболее эффективным форматом сжатия и обладает рядом передовых функциональных возможностей (включая поддержку HDR). Это делает его оптимальным выбором для высоконагруженных веб-сайтов и обеспечивает сокращение размера до 50...65 % по сравнению с JPG. Однако следует отметить, что уровень поддержки данного формата составляет примерно 73 %, поэтому, для обеспечения максимальной совместимости и доступности контента при внедрении AVIF в разработку рекомендуется предусмотреть резервный вариант в виде форматов WebP или JPG.

Для иконок представлен формат SVG, который характеризуется значительно меньшим объемом данных при представлении простых графических элементов по сравнению с аналогами и обладает высокой степенью гибкости. SVG – векторный формат, в связи с чем иконка масштабируется без потери чёткости на любых разрешениях, в то время как PNG при увеличении размывается и требует отдельных (под разные размеры) ресурсов. Кроме того, SVG может быть интегрирован непосредственно в структуру HTML, что способствует оптимизации процесса загрузки веб-страниц. Применение формата PNG целесообразно лишь в тех случаях, когда требуется воспроизведение сложных растровых изображений.

Ограничения форматов и нюансы использования

Исследования показывают, что AVIF медленно декодируется на слабых устройствах, что в конечном итоге приводит к росту нагрузки на CPU. При этом декодирование одного 2560×1440 AVIF составляет:

- флагманский смартфон 45 мс;
- среднебюджетный смартфон (Snapdragon 6 Gen 1) 180 мс;
- бюджетный смартфон (Mediatek Helio G35) 520 мс.

В связи с этим AVIF может быть менее подходящим для приложений, направленных на Индию, Азию и Африку с высоким процентом бюджетных устройств. В свою очередь, на очень старых устройствах (типа CPU (ARM v5)) WebP работает медленнее, чем JPG (JPG decompression 25 мс, WebP decompression 85 мс), что редко, но технически необходимо для IoT-систем.

«Ленивая» загрузка (Lazy Loading) изображений

«Ленивая» загрузка откладывает запрос ресурса до момента, когда пользователь собирается его увидеть. Реализация – Native Lazy Loading `<img src = “image.jpg” loading = “lazy” alt = “image” width = “800” height = “600” >`. При этом изображения Lazy loading представляются одними из наиболее переоценённых техник оптимизации. Несмотря на то, что «ленивая» загрузка рекомендуется в профессиональном направлении, наблюдения свидетельствуют о том, что её применение, как правило, приводит к отрицательной динамике показателей эффективности, а не к их предполагаемому улучшению.

Фундаментальная проблема – loading = «lazy» откладывает загрузку до момента приближения к viewport. Это означает активное снижение загрузки браузера (браузер откладывает загрузку до момента, когда изображение становится видимым, и на сети 3G добавляет 500+ мс задержки (LCP увеличивается с 1,2 сек. до 2,5+ сек.)). При этом пользователь видит только текст и белые блоки, где должны быть изображения. В условиях низкой пропускной способности сети процесс рендеринга может быть существенно замедлен, что приводит к артефактам в виде «дергающегося» контента, а внезапное накопление отложенных запросов может вызвать интерференцию между текущими и отложенными задачами. В результате чего возможна деградация пользовательского опыта, что оказывает негативное влияние на ключевые метрики (Cumulative Layout Shift (CLS)). Именно по данной причине мы не будем рассматривать lazy loading как способ улучшить показатели загрузки. Lazy Loading может применяться только в редких случаях и специфичных кейсах, где это реально необходимо и не нарушит предполагаемую работу сайта.

Алгоритмы сжатия: gzip vs brotli
Историческая эволюция и механизмы сжатия

GZIP (GNU ZIP) – алгоритм сжатия на основе DEFLATE, используется с 1992 года. Де-факто стандарт для веб-сжатия до 2016 года. Согласно наблюдениям, около 80 % сжатых ресурсов в веб-трафике используют GZIP [21].

Brotli – алгоритм сжатия, разработанный Google (2013), оптимизирован специально для веб-контента (текст, HTML, CSS, JS).

На Рисунке 4 видно, что файл, загружаемый сайтом, был сжат gzip-способом, для Brotli было бы указано br.

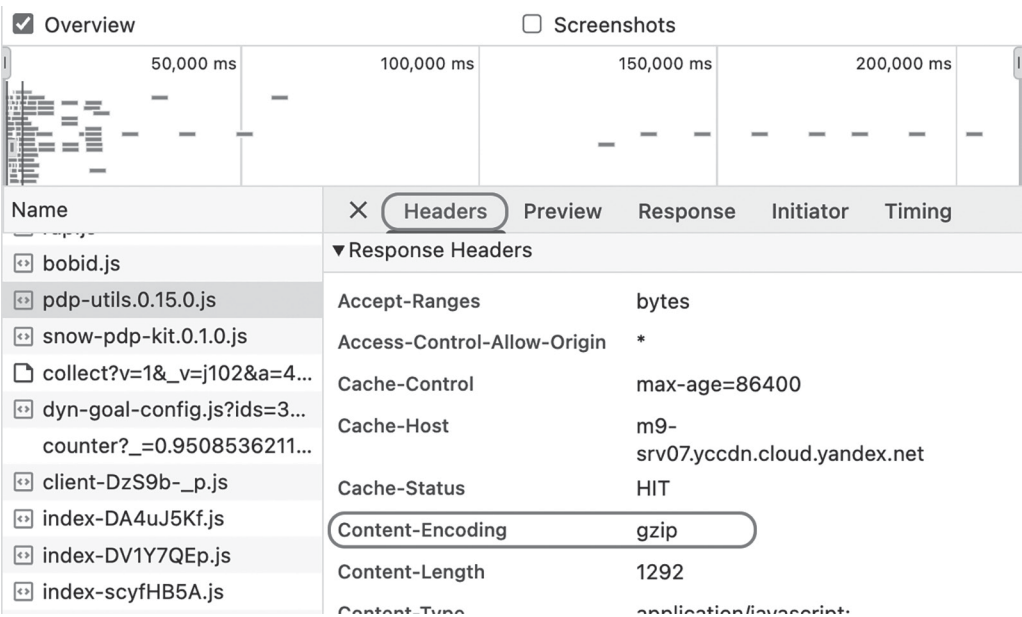


Рисунок 4. Способ сжатия ресурса, полученного при загрузке сайте

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

Метрики сжатия

Для сравнения взяли средний размер для каждого типа файлов и примерные результаты сжатия на основе исследований. Итоги приведены в Таблице 4 [13; 22; 23].

Таблица 4

Результаты сжатия различных типов контента

Тип контента	Исходный размер	GZIP (уровень 6), KB	Brotli (уровень 6), KB	Разница (GZIP vs Brotli), %
HTML-страница	180 KB	42	35	16,7
JavaScript	320 KB	95	78	17,9
CSS	85 KB	18	14	22,2
JSON API	450 KB	120	95	20,8
Минифицированный код	1,2 MB	320	255	20,3

В среднем Brotli может показать до 15...25 % лучшее сжатие, чем GZIP [21].

На основе исследования Akamai (2024) [22]:

- JavaScript на 14 % меньше, чем GZIP;
- HTML на 21 % меньше, чем GZIP;
- CSS на 17 % меньше, чем GZIP.

У обоих алгоритмов имеются уровни сжатия. Они влияют на итоговый результат получаемого размера, а также на скорость сжатия и нагрузку на CPU. GZIP поддерживает уровни сжатия от 1 (самый быстрый) до 9 (максимальное сжатие). На низких уровнях (1–3) алгоритм работает быстро, но сжатие менее эффективно. При увеличении уровня до 6 (часто используемый) время обработки растёт в несколько раз, но эффективность сжатия значительно улучшается. На максимальном уровне 9 время обработки существенно увеличивается, однако дополнительное улучшение коэффициента сжатия становится минимальным [24]. Инженерные исследования показывают, что повышение уровня сжатия выше 4–6 приводит к экспоненциальному росту вычислительных затрат при минимальной прибыли в качестве сжатия. Для большинства случаев выбирают уровень 6 как оптимальный баланс между эффективностью и скоростью.

Brotli поддерживает уровни от 1 до 11, и даже на низких уровнях обеспечивает улучшенное сжатие по сравнению с GZIP. На уровне 1 Brotli работает с похожей скоростью, что и GZIP уровня 1, но достигает лучшего коэффициента сжатия. Начиная с уровня 4–6, время требования к ресурсам возрастают, и на максимальных уровнях (10–11) Brotli требует значительно больше времени. Так, исследования Cloudflare [25] свидетельствует о том, что уровень 4 достигает эффективности сжатия в пределах 59,58 %, а уровень 11 – примерно 66,94 % (однако при этом требует значительно больше вычислительных возможностей). Таким образом, на основе тестирования можно утверждать, что уровни 4–6 Brotli следует применять как оптимальный вариант между качеством сжатия и производительностью браузера, где каждый уровень сжатия снижает производительность процессора, а прирост коэффициента становится менее значимым.

В контексте генерируемого (в ответ на каждый запрос пользователя) контента применение высоких уровней сжатия может приводить к увеличению времени отклика сервера, что, в свою очередь, может негативно сказаться на общей производительности системы.

Таким образом, чтобы обеспечить оптимальное соотношение параметров сервера, при разработке и оптимизации алгоритмов сжатия данных важно учитывать баланс между степенью сжатия и производительностью процессора.

Использование Gzip и Brotli в реальных кейсах

Выбор между GZIP и Brotli зависит от нескольких факторов. GZIP, благодаря высокой совместимости со всеми браузерами, остаётся универсальным выбором; он идеален для динамического контента, так как позволяет проводить сжатие «на лету», а уровень сжатия обеспечивает необходимый баланс между размером файла и скоростью обработки. Brotli целесообразно пользоваться на современных веб-платформах. Данный метод демонстрирует превосходство сжатия на 15...25 % по сравнению с GZIP. Однако следует учитывать, что Brotli требует значительно больших вычислительных ресурсов и при максимальном уровне сжатия (уровень 11), обработка файлов может значительно снижаться. В связи с этим для оптимизации функциональности Brotli рекомендуется использовать уровни сжатия в диапазоне 4–6, что обеспечивает баланс между качеством и нагрузкой на сервер.

Необходимая для успешной работы стратегия сжатия предполагает использование обоих алгоритмов одновременно: Brotli – для статического контента (CSS, JavaScript, HTML), который предварительно сжимается на сервере и кешируется, GZIP – для динамического контента (API-ответы, пользовательские страницы) и генерации в реальном времени. Такой гибридный подход позволяет максимально использовать преимущества обоих алгоритмов, не жертвуя производительностью сервера. Для сравнения: типичное веб-приложение (React, Webpack, CSS framework) может быть сжато с помощью Brotli уровня 6 на 15...25 % более эффективно, чем с GZIP, что для кода 1,2 МБ означает экономию примерно 180...300 КБ пропускной способности на каждого пользователя.

Конфигурация

Внедрение эффективного сжатия контента требует выбора правильной стратегии для каждого типа данных и корректной настройки конфигурационных файлов для этого. Рассмотрим вариант конфигурации в классическом формате (серверные настройки – в `nginx.conf`, приложение – в `app.js`, сборка – в `webpack.config.js`), в котором дифференцируем обработку динамического (API-ответы) и статического (JavaScript, SVG) контентов, где для первого варианта предложен GZIP с уровнем компрессии 5, для второго – двухуровневая схема с предварительным сжатием на этапе сборки (GZIP и Brotli). Данный выбор предложен по причине того, что архитектура существенно снижает вычислительную нагрузку сервера, раскрывая потенциал современных алгоритмов.

Этап 1. Конфигурация глобального сжатия динамического контента в файле `nginx.conf`

Эффективное управление динамическим контентом является критически важным аспектом. Одним из ключевых инструментов оптимизации является настройка механизма сжатия данных, реализованного в веб-сервере, процесс которого следует начинать с активации модуля GZIP, а также устанавливать уровень компрессии данных на значение 5, что соответствует оптимальному балансу между уровнем сжатия и скоростью обработки. Это позволяет избежать избыточного потребления ресурсов при обработке мелких объёмов

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

ектов. Для обеспечения корректного функционирования механизма сжатия охват MIME-типов, подлежащих обработке, включает такие типы, как `text/plain`, `text/css`, `application/json` и `application/javascript`, которые являются наиболее часто встречающимися в веб-приложениях. Их сжатие позволяет существенно сократить объем передаваемых данных. Дополнительно, для обеспечения совместимости и гибкости конфигурации, следует активировать заголовок `gzip_vary on`, который информирует браузеры о необходимости учитывать вариабельность сжатых данных, что способствует более эффективному использованию кеширования и оптимизации передачи данных без чрезмерных задержек.

Этап 2. Логика выбора предлагаемых статических файлов (`nginx.conf`)

В контексте конфигурации блока `location` (для расширений `js` и `svg`) реализуется условный механизм обработки заголовка `Accept-Encoding`. В случае поддержки формата `Brotli` осуществляется перенаправление файла с расширением `.br` и кодировкой `Content-Encoding: br`. Если сервер поддерживает `GZIP`, то происходит перенаправление с расширением `.gz` и кодировкой `Content-Encoding: gzip`. Дополнительно включаются заголовки `Vary: Accept-Encoding`, что обеспечивает вариативность в зависимости от предпочтений клиента.

Этап 3. Интеграция в приложение `Node.js/Express` (`app.js` или `server.js`)

Для динамического трафика применяется `middleware compression` с уровнем `5`, значением `1000` байт и фильтрацией исключений `x-no-compression`. При этом сервис статикки расширяется следующими заголовками: `Vary: Content-Encoding` по суффиксу (`.br` или `.gz`); корректировка MIME (`application/javascript` для JavaScript, `text/css`).

Этап 4. Конфигурация в сборке `Webpack` (`webpack.config.js`)

В рамках `production`-режима активируются плагины `Compression Plugin (GZIP)` и `Brotli Plugin` с применением выражения `/(js|css|html|svg)$/`. При этом установленный в значение `false` параметр `delete Original Assets` гарантирует сохранение файлов для браузеров в качестве резервных копий, не поддерживающих форматы `Brotli` и `GZIP`. Пороговое значение `threshold: 8192` определяет оптимальный для сжатия размер файла, а `minRatio: 0.8` гарантирует, что файл сжимается только при условии, если сжатая версия составляет менее `80 %`. При этом функционирующее на основе интеллектуального анализа серверное приложение автоматически выбирает более подходящий способ сжатия, обеспечивая при этом необходимую эффективность использования ресурсов. Данный подход оптимизирует трафик, сохраняя универсальную совместимость, а также обеспечивает необходимое сжатие для различных типов контента с минимальными расходами.

Проведённое исследование показало, что оптимизация производительности веб-контента является важным направлением улучшения показателей `Core Web Vitals`, а комбинированное применение методов минификации позволяет снизить размер начального банда до `91,7 %`.

Оптимальное решение для большинства сценариев использования веб-приложений представляет собой формат `WebP`, который демонстрирует сокращение размера файлов с экономией до `40...45 %` по сравнению с традиционным форматом `JPEG`. Это позволяет существенно улучшить работу системы. Однако при рассмотрении альтернативных форматов, таких как `AVIF`, необходимо учитывать их потенциал для дополнительного сжатия, который может достигать до `65,6 %` и осуществлять анализ совместимости кодирующих алгоритмов на различных устройствах.

Таким образом, следует полагать, что алгоритмы сжатия (Gzip и Brotli) должны применяться в зависимости от типа контента. При этом для статического контента рационально применять Brotli уровня 9 или 11 (с предварительным сжатием), для динамического контента рекомендуется применять GZIP уровня 5 как баланс между скоростью сжатия и размером результата. Исследование демонстрирует, что интеграция комплексного подхода к оптимизации включающего управление программным кодом размера контента, выбор эффективных форматов и применение алгоритмов сжатия может привести к высокому снижению объема загружаемого контента на 40...60 %, что обеспечивает максимальное повышение показателей производительности веб-приложений.

Литература

1. Understanding Core Web Vitals and Google search results // Google Search Central [Электронный ресурс]. URL: <https://developers.google.com/search/docs/appearance/core-web-vitals> (дата обращения: 22.03.2026).
2. Горячкин Б.С. Ханмурзин Т.И. Повышение эффективности работы с веб-ресурсом за счет инструментария системного программиста // Динамика сложных систем – XXI век. 2022. № 16(3). С. 26.
3. Как размер кода зависит от минификатора, сборщика и языка // Habr. 2019 [Электронный ресурс]. URL: <https://habr.com/ru/companies/yandex/articles/438022/> (дата обращения: 12.03.2026).
4. Ways to Optimize Your Java Script Bundle Size // Codecov. 2024 [Электронный ресурс]. URL: <https://about.codecov.io/blog/8-ways-to-optimize-your-javascript-bundle-size/> (дата обращения: 01.03.2026).
5. Frontend Performance Optimization: Make your websites load faster // Medium. 2025 [Электронный ресурс]. URL: <https://medium.com/@ashwinKumar0505/frontend-performance-optimization-make-your-websites-load-faster-677473f3a1e2> (дата обращения: 04.03.2026).
6. Tree Shaking // Webpack. 2024 [Электронный ресурс]. URL: <https://webpack.js.org/guides/tree-shaking/> (дата обращения: 06.03.2026).
7. AVIF vs WebP: Which Image Format Reigns Supreme // Elementor. 2025 [Электронный ресурс]. URL: <https://elementor.com/blog/webp-vs-avif/> (дата обращения: 22.03.2026).
8. React image optimization: Best techniques for faster apps // Uploadcare. 2025 [Электронный ресурс]. URL: <https://uploadcare.com/blog/react-image-optimization-techniques/> (дата обращения: 12.03.2026).
9. How to Optimize Images for Faster Web Performance // Pixel Free Studio. 2024 [Электронный ресурс]. URL: <https://blog.pixelfreestudio.com/how-to-optimize-images-for-faster-web-performance/> (дата обращения: 06.03.2026).
10. Повышаем скорость: lazy loading // German Web. 2025 [Электронный ресурс]. URL: <https://german-web.org/povyshaem-skorost-lazy-loading/> (дата обращения: 31.03.2026).
11. Maloney M., Grigorik I. (2023). Web Vitals optimization guide. Google Chrome Documentation. The Difference between Brotli vs gzip // RoseHosting. 2024 [Электронный ресурс]. URL: <https://www.rosehosting.com/blog/brotli-vs-gzip/> (дата обращения: 13.03.2026).
12. Gzip vs. Brotli Compression for Website Speed // OSHYN. 2024 [Электронный ресурс]. URL: <https://www.oshyn.com/blog/gzip-vs-brotli> (дата обращения: 15.03.2026).
13. Website Speed Optimization Strategies for 2024 // Browser Stack. 2024 [Электронный ресурс]. URL: <https://www.browserstack.com/guide/website-speed-optimization-strategies> (02.03.2026).

Оптимизация размера и производительности веб-контента:
управление кодом, форматы изображений и алгоритмы сжатия

14. *Shailesh K.S. and Suresh P.V.* (2017). An analysis of techniques and quality assessment for Web performance optimization. *Indian Journal of Computer Science and Engineering (IJCE)*. No. 8. Pp. 61–69.
15. Front-End Performance Optimization Tips // Strapi. 2025 [Электронный ресурс]. URL: <https://strapi.io/blog/front-end-performance-optimization-tips> (дата обращения: 10.03.2026).
16. Choose the right image format // Web.dev. 2024. Google Developers [Электронный ресурс]. URL: <https://web.dev/articles/choose-the-right-image-format?hl=ru> (дата обращения: 10.03.2026).
17. WebP Compression Study. Google Developers. // Google Developers [Электронный ресурс]. URL: https://developers.google.com/speed/webp/docs/webp_study (дата обращения: 12.03.2026).
18. Code Splitting with Webpack // Webpack Documentation [Электронный ресурс]. URL: <https://webpack.js.org/guides/code-splitting> (дата обращения: 10.03.2026).
19. Tree Shaking in Modern JavaScript // Webpack Documentation [Электронный ресурс]. URL: <https://webpack.js.org/guides/tree-shaking> (дата обращения: 02.03.2026).
20. Эффективность Brotli в реальном мире // Habr. 2020 [Электронный ресурс]. URL: <https://habr.com/ru/companies/ruvds/articles/499278/> (дата обращения: 06.03.2026).
21. Brotli vs. GZIP: A Comprehensive Comparison of Compression Algorithms // Cloudways. 2023 [Электронный ресурс]. URL: <https://www.cloudways.com/blog/brotli-vs-gzip/> (дата обращения: 12.03.2026).
22. ZSTD vs Brotli vs GZip Comparison // Speed Vitals. 2025 [Электронный ресурс]. URL: <https://speedvitals.com/blog/zstd-vs-brotli-vs-gzip/> (дата обращения: 15.03.2026).
23. Speeding Up gzip Compression // Baeldung. 2023 [Электронный ресурс]. URL: <https://www.baeldung.com/linux/gzip-faster-compression> (дата обращения: 17.03.2026).
24. All the way up to 11: Serve Brotli from origin and Introducing Compression Rules // Cloudflare. 2023 [Электронный ресурс]. URL: <https://blog.cloudflare.com/this-is-brotli-from-origin/> (дата обращения: 18.03.2026).

References

1. Understanding Core Web Vitals and Google search results. Google Search Central [Electronic resource]. Available at: <https://developers.google.com/search/docs/appearance/core-web-vitals> (accessed: 22.03.2026).
2. Goryachkin B.S. Khanmurzin T.I. (2022). Improving the efficiency of working with a web resource using a systems programmer's toolkit. *Dynamics of Complex Systems - XXI Century*. No. 16(3). P. 26. (In Russian).
3. How code size depends on the minifier, compiler, and language. Habr. 2019 [Electronic resource]. Available at: <https://habr.com/ru/companies/yandex/articles/438022/> (accessed: 12.03.2026). (In Russian).
4. Ways to Optimize Your Java Script Bundle Size. Codecov. 2024 [Electronic resource]. Available at: <https://about.codecov.io/blog/8-ways-to-optimize-your-javascript-bundle-size/> (accessed: 01.03.2026).
5. Frontend Performance Optimization: Make your websites load faster. Medium. 2025 [Electronic resource]. Available at: <https://medium.com/@ashwinKumar0505/frontend-performance-optimization-make-your-websites-load-faster-677473f3a1e2> (accessed: 04.03.2026).
6. Tree Shaking. Webpack. 2024 [Electronic resource]. Available at: <https://webpack.js.org/guides/tree-shaking/> (accessed: 06.03.2026).
7. AVIF vs WebP: Which Image Format Reigns Supreme. Elementor. 2025 [Electronic resource]. Available at: <https://elementor.com/blog/webp-vs-avif/> (accessed: 22.03.2026).

8. React image optimization: Best techniques for faster apps. Uploadcare. 2025 [Electronic resource]. Available at: <https://uploadcare.com/blog/react-image-optimization-techniques/> (accessed: 12.03.2026).
9. How to Optimize Images for Faster Web Performance. Pixel Free Studio. 2024 [Electronic resource]. Available at: <https://blog.pixelfreestudio.com/how-to-optimize-images-for-faster-web-performance/> (accessed: 06.03.2026).
10. Increasing speed: lazy loading. German Web. 2025 [Electronic resource]. Available at: <https://german-web.org/povyshaem-skorost-lazy-loading/> (accessed: 31.03.2026). (In Russian).
11. Maloney M., Grigorik I. (2023). Web Vitals optimization guide. Google Chrome Documentation. The Difference between Brotli vs gzip. Rose Hosting. 2024 [Electronic resource]. Available at: <https://www.rosehosting.com/blog/brotli-vs-gzip/> (accessed: 13.03.2026).
12. Gzip vs. Brotli Compression for Website Speed. OSHYN. 2024 [Electronic resource]. Available at: <https://www.oshyn.com/blog/gzip-vs-brotli> (accessed: 15.03.2026).
13. Website Speed Optimization Strategies for 2024. Browser Stack. 2024 [Electronic resource]. Available at: <https://www.browserstack.com/guide/website-speed-optimization-strategies> (accessed: 02.03.2026).
14. Shailesh K.S. and Suresh P.V. (2017). An analysis of techniques and quality assessment for Web performance optimization. *Indian Journal of Computer Science and Engineering (IJCSE)*. No. 8. Pp. 61–69.
15. Front-End Performance Optimization Tips. Strapi. 2025 [Electronic resource]. Available at: <https://strapi.io/blog/front-end-performance-optimization-tips> (accessed: 10.03.2026).
16. Choose the right image format. Web.dev. 2024 [Electronic resource]. Google Developers. Available at: <https://web.dev/articles/choose-the-right-image-format?hl=ru> (accessed: 10.03.2026).
17. Web P Compression Study. Google Developers. Google Developers. Available at: https://developers.google.com/speed/webp/docs/webp_study (accessed: 12.03.2026).
18. Code Splitting with Webpack. Webpack Documentation [Electronic resource]. Available at: <https://webpack.js.org/guides/code-splitting> (accessed: 10.03.2026).
19. Tree Shaking in Modern JavaScript. Webpack Documentation [Electronic resource]. Available at: <https://webpack.js.org/guides/tree-shaking> (accessed: 02.03.2026).
20. Brotli Efficiency in the Real World. Habr. 2020 [Electronic resource]. Available at: <https://habr.com/ru/companies/ruvds/articles/499278/> (accessed: 06.03.2026). (In Russian).
21. Brotli vs. GZIP: A Comprehensive Comparison of Compression Algorithms. Cloudways. 2023 [Electronic resource]. Available at: <https://www.cloudways.com/blog/brotli-vs-gzip/> (accessed: 12.03.2026).
22. ZSTD vs Brotli vs GZip Comparison. Speed Vitals. 2025 [Electronic resource]. Available at: <https://speedvitals.com/blog/zstd-vs-brotli-vs-gzip/> (accessed: 15.03.2026).
23. Speeding Up gzip Compression. Baeldung. 2023 [Electronic resource]. Available at: <https://www.baeldung.com/linux/gzip-faster-compression> (accessed: 17.03.2026).
24. All the way up to 11: Serve Brotli from origin and Introducing Compression Rules. Cloudflare. 2023 [Electronic resource]. Available at: <https://blog.cloudflare.com/this-is-brotli-from-origin/> (accessed: 18.03.2026).

Поступила в редакцию: 30.03.2026

Поступила после рецензирования: 12.05.2026

Принята к публикации: 28.05.2026

Received: 30.03.2026

Revised: 12.05.2026

Accepted: 28.05.2026