TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of System Architecture, Chair of Computer Networks

Streaming-Based Progressive Enhancement of Websites for Slow and Error-Prone Networks

Lucas Jacob Vogel

Dissertation

to achieve the academic degree

Doktoringenieur (Dr.-Ing.)

First referee
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Second referee
Prof. Dr. Piotr Gaj
Advisor
Prof. Dr. Raimund Dachselt

Submitted on: February 15, 2023

Acknowledgements

Thanks to my supervisor Dr. Thomas Springer for his guidance and feedback and to Prof. Alexander Schill for the opportunity to work on this topic. Also, thanks to my family and friends, who supported me on this journey. Additionally, I want to thank 3m5. for enabling me to contact one of their customers, Solarwatt, for providing their source code without conditions to be evaluated as part of this work. Also, thanks to Prof. Gai for his report. Special thanks go to Nicole Schneider for her valuable feedback and insight into the technologies used to speed up web pages in real-world commercial projects.

Plagiatism check

This work was checked for plagiarism and self-plagiarism using Turnitin, provided by Scribbr (report ID: oid:3471:137844881).

Contents

1 Introduction

1.1 Motivation

In this modern era, omnipresent access to information is often taken for granted. Through services like the World Wide Web, seemingly all knowledge of modern humanity is obtainable by anyone with internet access. As a result of the ever-increasing expansion of web services, users have become accustomed to short loading times, as it has an ever-increasing importance in all areas of life. From online shopping to research, from media consumption to work: users rely on fast, readily-available web applications. This is also reflected by user behavior. According to Nah et al., users expect web pages to load in about 2 seconds [Nah04]. Furthermore, 53% of users leave a web page if it is not loaded after 3 seconds [Kir16]. In contrast, the average loading time of mobile web pages is 22 seconds (study by Google Research and Webpagetest.org, a sample of 900.000 mobile pages, loaded via a 3G connection [Laj19; Goo17]). As shown in reports of the HTTP Archive, improvements in network speed are immediately consumed by larger web pages [HTT21a; 22s; HTT21b]. These limitations have major economic consequences. According to Google, advertising revenue decreases by 20% when results take 0.9 seconds to load instead of 0.4 seconds [Lin06]. Amazon reports a 1% loss in sales if the results page takes 100ms longer to load [KL07; Opt08]. Accordingly, it is in the interest of users as well as providers to load web pages faster. However, there are multiple challenges that lead to slower loading times. Mainly, both sides cannot influence the network speed of the connection between server and client. Still, what data is sent and in which order is up to the page owner.

The technique described in this work proposes a new solution in which all code that prevents the rendering of the page is streamed in chunks instead of being loaded as files. Doing so continuously adds content to the page, making it possible to interact with the page while it still loads. This is beneficial for both users and providers: the page loads faster, even at slower network speeds, which results in fewer users leaving a page and, thus, higher user satisfaction as well as revenue for the website providers. Furthermore, this technique can be integrated into existing solutions without rewriting the entire code-base.

1.2 Problem Discussion

Currently, a trend is visible with HTTP. In the first generation of the web built with HTTP/1.0, every request required a TCP Handshake. This caused multiple issues negatively impacting loading speed, as detailed in the performance analysis of the W3C, which concluded that ”HTTP/1.0 interacts badly with TCP” [97]. Therefore, developers used, among other things, resource bundling as a way to optimize loading speed. With bundling, multiple resources are combined into one file. As a result, only one handshake is needed, and the impact caused by TCP slow start is reduced due to a larger file size [97; 22aq]. However, the issue of requiring a new TCP connection for every request was addressed with HTTP/2. There, multiplexing is introduced, which allows for transferring different files, and file types over one stream [19a; 22al]. Furthermore, it allowed for stream prioritization, giving the opportunity to load more important elements first, and less important elements later [19a]. This represents a contradiction to HTTP/1.0, as bundling does not represent a performance benefit anymore. As Erwin Hofman, a web performance expert, explained: ”Bundling is an anti-pattern in HTTP/2 [...] In HTTP/2, this behaviour will end up impacting the download-time of other resources as well, because of the way HTTP/2 works.” [22aq]. He added, ”Even better, try to keep framework, library and add-on resources codesplitted” [22aq]. Splitting is, therefore, a better approach for optimizing resources when using HTTP/2, as it allows prioritizing the streaming of code sections necessary to display the initial page and delay everything else. This trend towards using streams is further reinforced with HTTP/3, as it is built on top of QUIC to provide reliable streams directly on top of UDP [22am]. There are also other factors that influenced the move toward implementing a more significant portion of the network stack. However, a clear trend is visible: streams have become a central aspect of how web pages are transferred and will be in the future as well. While this change will lead to new improvements in how browsers and servers communicate, modern front-ends are still made on a file-by-file basis, for example React.js, Angular or Vue [22ai; 22g; 22av]. When a user requests a web page, all required files are transmitted via streams and combined back to files on the client side. While the system works, it does not utilize the full potential of the streams, as in its current state, progress is made only when the whole file is transferred [19b]. For example, if a user experiences a network timeout due to a slow-loading web page, the resulting page is not shown due to incomplete files, even if all necessary code is loaded.

The techniques presented in this work build on top of both trends: users expect to access ever-faster web pages with ever-increasing complexity while the web is heading towards a streaming-based delivery. By delivering the whole web page via a stream, loading time improvements can be made. However, this approach leads to further challenges on how to split the main resource types (HTML, JavaScript and CSS) into streams. Therefore, new techniques for all three elements are presented as well.

1.3 Problem Areas

The main challenges arise from the concept of stream-based delivery itself. Streaming an entire web page is a high-level goal, which cannot directly benefit from (and be built on top of) existing frameworks, as they do not exist. Therefore, the main problem areas are diverse:

1. Assessing the remaining improvement potential of modern web pages
2. Building a backward-compatible streaming approach for delivering HTML
3. Split resources necessary for displaying the page
   a) Splitting CSS: Rendering CSS and sending it as-needed
   b) Splitting or delaying JavaScript without breaking functionality
   c) Splitting the final HTML with in-lined code
4. Create a course of action for including said techniques into a build workflow, which separates actions required to take by the developer and fully automatic optimizations

Especially the integration of the developed techniques is necessary to ensure that the developed methods are applicable in real-world scenarios.

1.4 Goal of the Thesis

The main goal of this thesis is to research the performance impact of modified stream-based loading behavior of web pages. More specifically, this thesis provides a set of tools that allow web pages to be streamed instead of loading as a set of render-blocking files. For this change, multiple sub-steps are required. First, a large-scale analysis is necessary to evaluate the improvement potential and state of the art of current web pages. This itself brings significant insight into the current state of the web. Next, based on the analysis results, a concept will be created that targets the highest performance possible, also based on current trends of the HTTP protocol. Furthermore, the goal is to create a backward-compatible approach, which is also easy to integrate by developers with low manual effort. Lastly, the evaluation will include code coverage measurements as well as loading speed and user satisfaction tests, with the goal of testing and determining the scale of possible improvements.

1.5 Research Questions

The following research questions emerge based on the problem description:

RQ1: Are streamed web pages with a reordered loading schedule faster than the traditionally loaded counterparts?
This is the most important question, as the technique needs to present significant loading speed improvements to be viable.
RQ2: Are current (streaming-) protocols sufficient for delivering such web pages, or are new protocols needed?
Streaming web pages does not fit directly into the file-based HTTP approach. However, modern Streaming options like WebSockets and SSE exist, which both need to be evaluated over their capabilities for delivering a web page's content.
RQ3: To which extent can the new method be used for existing web pages?
For any modification of this scale, backward compatibility or at least compatibility without changing client-side software must be possible. Otherwise, adaptation would not be feasible.
RQ4: How much of this process can be automated in order to reduce development effort?
When developing a framework that integrates deeply into multiple aspects of web development, the effort for developers to use said tool has to be minimal.
RQ5: To which extent is the new loading behavior accepted by users?
The developed technique can only be successfully used in real-world applications if most users accept a modified loading behavior.

1.6 Conflict of Interest

During the work on this thesis, the author worked a part-time job for the software development company "3m5. Media GmbH". During this time, both this thesis and work stayed completely separate, with one exception. For the final evaluation test, "3m5." provided the code base of "solarenergie.de" with the explicit permission of the customer, "Solarwatt." For this, no extra compensation in any form was provided. It was also not bound to any condition. The code base was only provided "as-is." This allowed testing the software developed as part of this work on a real-world web page.

1.7 Structure of the Thesis

First, the fundamentals of web technology relevant to loading performance are described in chapter 2. These include explaining how web pages are built and how they interact with each other when loading a website. Furthermore, it contains detailed descriptions and discussions about the most important performance markers used to evaluate web page performance. This is followed by the related work of all significant concept components in chapter 3. These include research papers and popular open-source or industry technologies, focusing on optimizing HTML, CSS, and JavaScript for a faster loading time and streaming web pages. Next, a large-scale analysis is presented in chapter 4. This is required, as no existing large-scale data set includes freely accessible and sufficient code usage information until render and the render-blocking property of modern web pages. Only with this information can a viable concept be built, as the problem scale is currently unknown. The concept in chapter 5 is then based on said results. This chapter in itself is split into three separate parts: First, a concept for optimizing and splitting CSS is described in section 5.3, developing the `Essential` framework. Then, the same optimizing and splitting are conceptualized in section 5.4 for JavaScript, presenting `Waiter` and `AUTRATAC`, which allow for more flexible code splitting. Lastly, a combined concept of `Essential`, `Waiter`, and `AUTRATAC` is created in section 5.5, which also includes a splitting technique for HTML and conceptualizes the backward-compatible streaming method. Afterward, the concept is evaluated in chapter 6. There, the code efficiency until render is tested, as well as a case study of a real-world code base. The user satisfaction of the modified loading behavior is also tested in this chapter. Finally, the conclusion is described in chapter 7, summarizing all findings.

2 Fundamentals

```html
<!DOCTYPE html>
<html>
  <head>
  <title>page title</title>
     <!-- head content -->
  </head>
  <body>
    <!-- body content -->
  </body>
</html>
```

Listing 1: Small example of the structure of a valid HTML file

In order to discuss existing approaches that speed up the loading times of websites, multiple concepts have to be explained first. These include the basics of web pages, web data transfer, and different modification areas of the delivered code. The described terms are also necessary for the following concept.

2.1 Render Pipeline and Render-Blocking

When a web page is loaded, the browser first requests the main HTML file. Multiple dependencies can be declared inside this file, like external CSS, JavaScript, or font files. Especially CSS and JavaScript will be loaded in a render-blocking way by default, which means stopping the main HTML file's parsing process and loading and parsing the requested render-blocking resource. Processing all render-critical files is called the critical rendering path, and optimizing this process is one of the goals of modern web development [22i].

2.2 HTML, CSS, and JavaScript

The basis of all modern web pages consists of HTML, CSS, and JavaScript. All three components interact with each other, allowing developers to build modern web pages and web applications. Therefore, all three have to be considered separately.

2.2.1 HTML

The Hypertext Markup Language (HTML for short) is a text-based markup language that defines the base of all web pages today [w3o19]. Listing 1 shows what a minimal web page structure looks like. A valid HTML page starts with a Document Type Declaration (`DOCTYPE`), followed by the main `<html>` tag. Inside, two standard tags can be seen, the `<head>` and `<body>`-tags. Both serve different functions. Mainly, the `head` is used for meta information and linking necessary resources, like setting the page's title or linking CSS files. Valid "tags" and "text" contained in the `body` are (by default) visible if the file is opened in a browser. Specific tags are only valid in the context of other tags. For example, using the `<title>My title</title>`-element in the `head` will set the page's title. When the same element is used in the body, it might be ignored [dev22]. However, modern browsers might differ. It is possible that certain elements will still work, even though they are present in the context of the wrong parent tag. For example, setting the title in the `body` will still set the page title (tested in Chrome 97 and Firefox 96). As a result, more `head`-elements (like linking CSS-files) are now "body-ok" by the specification [wha22]. Therefore style files can also be linked in the `body`, possibly shifting the declaration and loading "up front" to a later point. It has to be noted that CSS files can be added in the `head` at a later point by using

```html
<!DOCTYPE html>
<html>
  <head>
  <script>
  //in-file script block
  function clickFunction(){
    console.log("function 'a' called")
  }
  </script>
  </head>
  <body>
   <!-- inline-JavaScript call:-->
    <button onclick="clickFunction()">Click</button>

   <!-- external-JavaScript file:-->
    <script type="text/javascript" src="./externalFile.js"></script>
  </body>
</html>
```

Listing 2: Different ways to incorporate JavaScript into a HTML-file

JavaScript. However, using this new flexibility is a necessary building block for the CSS processor and the HTML streaming described in this work.

2.2.2 JavaScript

JavaScript, or *JS* for short, is one of the main building blocks of the modern web and is used by nearly 98% of all web pages [w3t22]. It allows web pages to incorporate dynamic elements and enables additional functionality outside the provided HTML5 elements. JavaScript can be embedded inline, in-page script blocks, or as external files, as visible in Listing 2.

Parsing inline code and in-file JavaScript blocks is strictly render-blocking. This is also true for external JavaScript files per default, but can be changed by additional attributes like `async` and `defer`, as can be seen in figure 2.1 [Sop22]. The issue with *async* and *defer* scripts is the possibility of the code missing listeners to the completion of the DOM or not being available if other JavaScript functions depend on the availability of the included code. Therefore it is necessary to modify the included code in a way that makes it robust for being loaded asynchronously. This presents a challenge to developers, as they also need to improve the time until all relevant elements are shown on a page, which might depend on the execution of certain JavaScript functions. As optimizing JavaScript and making it robust for asynchronous loading is a significant area of optimization, the *async* and *defer* loading methods will be a central point in the techniques developed in this work.

2.2.3 CSS

CSS is a language that describes styling information for markup-language-based documents like HTML. It is therefore considered one of the core technologies of the modern web [22w]. With CSS, the content and layout of a page are separated. Other than JavaScript, CSS can technically be parsed and applied at every stage of the rendering process, as there are no external factors like ”listeners” which influence the behavior of CSS without using JavaScript. The only factor is the order in which the CSS files are loaded if the styling of elements is overridden. However, there is no native or easy universal way of delaying the loading and parsing of external CSS without JavaScript (excluding media attributes, which are not universal).

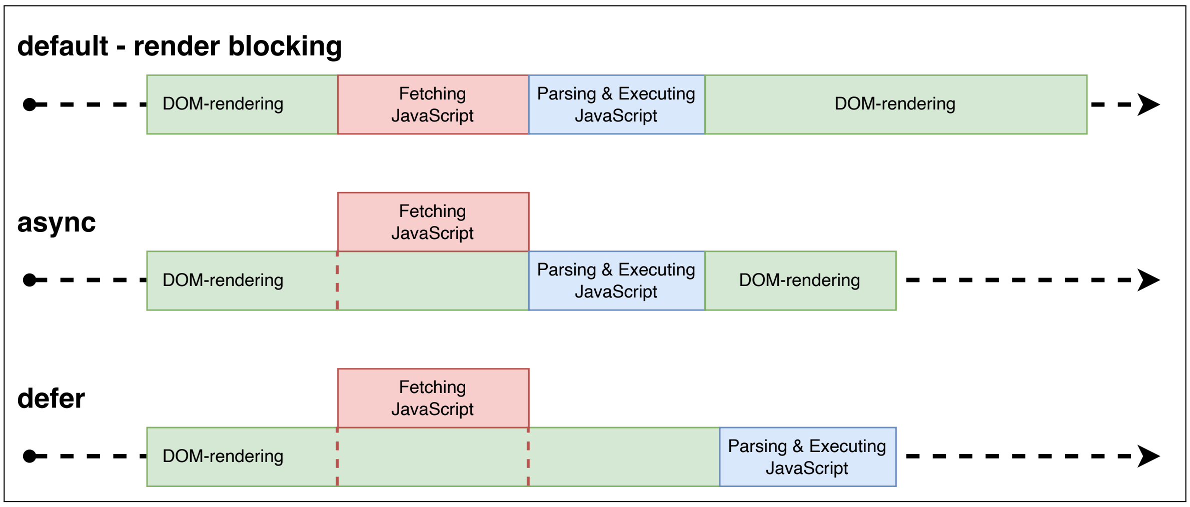


Figure 2.1: Different loading methods of JavaScript are compared and displayed on the time axis. It is shown that async and defer can improve loading speeds.

One factor is the movement created by applying CSS after the first render, called Cumulative Layout Shift (CLS). While loading a page, layout shift is generally seen as unfavorable behavior, as it might degrade the user experience [22l]. The straightforward solution is to load all necessary CSS before rendering. CSS can be inserted in a page in different ways, similar to JavaScript, as shown in Listing 2. All shown methods are render-blocking. Optimizing the amount of CSS needed universally is, therefore, one of the major research areas in this work.

2.3 HTTP

HTTP is a stateless, generic protocol used for data transmission in networks. It is mainly used to transfer web pages, making it one of the foundations of the modern Internet [Fie99]. HTTP is under constant development, and the current version is HTTP/3. One of the main problems with HTTP/1 is that a new TCP handshake must be performed for each resource. This has been solved with HTTP/2 [BPT15b]. Since then, transmitting different requests over a single TCP connection has been possible. This is done via streams, which are an essential part of HTTP/2. The multiplexing capability also allows for asynchronous data transfer. Figure 2.2 shows how this can shorten the request times of resources. However, HTTP/2 still has multiple shortcomings. Mainly, as HTTP/2 is based on TCP, underlying TCP blocking problems cannot be resolved by the protocol itself. Therefore, the multiplexing functionality is impacted at slow and lossy connections if TCP packets are lost or delayed [sec21]. Therefore, HTTP/3 is based on QUIC, which is only based on UDP and replaces TCP and TLS. As a result, a more significant portion of the protocol stack can be controlled without needing updated infrastructure and kernels, which transport and process TCP protocol packages. With this optimization, it is possible that the native multiplexing functionality of HTTP/3 only halts individual streams if package loss occurs, which will improve loading times for slow network conditions. However, this does not fix the fundamental problem with render-blocking and file-based resources. When content is loaded from other servers, a handshake must still be performed for each server. Thus, the problem regarding reducing HTTP requests with the associated handshakes is not sufficiently solved. Nevertheless, HTTP forms the basis of the transmission of web pages in browsers and is, therefore, unavoidable.

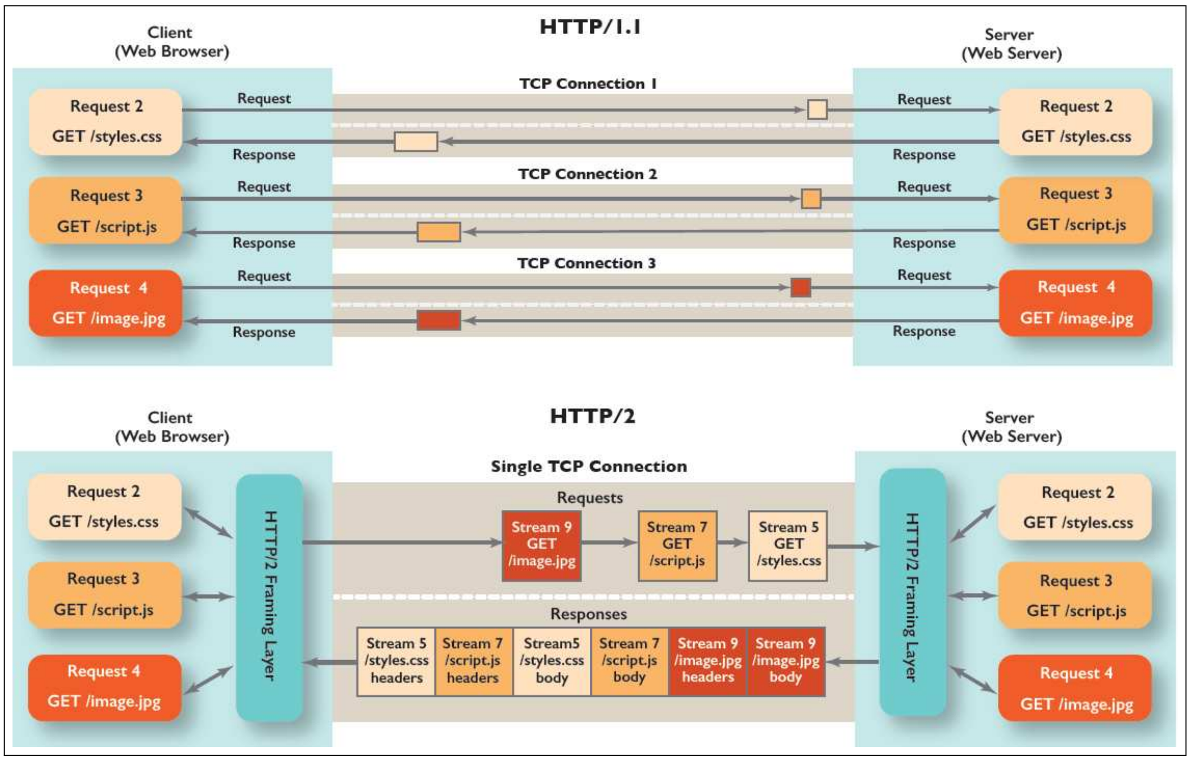


Figure 2.2: Comparison of data transmission via TCP with HTTP/1 and HTTP/2. Image Source: [Pol19]

2.4 Browser Web APIs

APIs of a browser provide various interfaces to the host system, the DOM, and functions that can be used in connection with a server. This browser-side interface is standardized by the W3C [W3C19]. Access to these interfaces is provided by the individual browser to the JavaScript of a web page. For example, the device's position, battery status, or WebSocket API can be accessed. Browser APIs provide extra functionality that can be used on a client by exposing certain System APIs to a web page.

2.5 Existing Streaming Techniques

Various methods exist to stream data from a server to a browser, with varying browser support and flexibility. The most important techniques are described next, focusing on browser support. Methods with better browser support will be preferred, as it improves backward compatibility and enables real-world deployment.

2.5.1 Server Push

Server-Push works by providing the MIME-type "multipart/x-mixed-replace." This is then interpreted by the browser as a file that is updated repeatedly every time the server sends a new version. It can be used to stream a web page, as techniques exist to persist data already sent to the client, for example, by using local storage. However, the most significant disadvantage is the lack of browser support, as especially Chrome only supports server push partially [22b].

2.5.2 Server-Sent Events

In contrast to Server Push, Server-Sent Events (SSE) are supported in a wide range of browsers [22an]. Furthermore, the required functionality for SSE (the EventSource interface) can be re-added to outdated browsers via Polyfill [Mod22]. According to the data provided by *CanIUse*, the main browser which requires a Polyfill is Internet Explorer. However, this browser is now deprecated, leaving only Opera Mini without direct support. The Opera Mini browser provides its own server-side optimizations, which is why the lack of support can be overlooked due to existing alternatives. However, SSEs have a limit of a maximum of 6 simultaneous connections. They are limited by browsers (Firefox and Chrome). Even though this is a known bug, it will not be fixed [22c; 22d]. Generally, this would not be an issue. However, loading more pages or resources at once is still possible, leaving an edge case. If more pages are loaded at the same time, starting at the seventh page, users would only see a white page.

2.5.3 WebSockets

Based on HTTP, WebSockets can be used by means of an upgrade header [KL00]. After the upgrade, these enable bidirectional, full duplex communication between client and server with the lowest possible overhead [KL00]. No other interface allows this, and is provided as a Browser API.

Protocol Structure

The protocol consists of two parts: Handshake and data transfer [KL00]. The handshake is initiated by the client, which transfers the following data (source: [MF11]):

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xxxxxxxxxxxxxxxxxxxxxxxx
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

And the answer from the server (source: [MF11]):

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: xxxxxxxxxxxxxxxxxxxxxxxxx
Sec-WebSocket-Protocol: chat
```

After that, the WebSocket connection can be used. The prerequisites for this are a server that supports WebSockets and a web browser on the client side that provides a WebSocket API. On the server side, this can be realized by frameworks. For example, for Node, there is the package `websocket` [McK19], which provides the client-side and server-side code for this. WebSockets do not communicate classically over HTTP but use their own protocol. For this purpose, the prefix `ws://` is available for unencrypted connections, and `wss://` for encrypted connections, so the same port as for `HTTP` (port 80) can be used [MF11; KL00]. The client-side API standard is described by the W3C and is implemented in the majority of current browsers [API15]. According to CanIUse.com, more than 97% of browsers support WebSockets, including all major browsers such as Chrome, Firefox, Safari, Internet Explorer, Edge, or Opera. Especially on mobile devices (except for Opera Mini) [DS13]. Establishing a connection to a compatible server is supported by default with just a single line of JavaScript. Other code is necessary to intercept callbacks, such as receiving messages or terminating the connection:

```
//Open connection
var socket = new WebSocket('ws://exampleWebsocketServer.com');
// Callback for receiving messages
socket.onmessage = function (event) {
    console.log(event.data);
};
// Callback if errors occur
socket.onerror = function (error) {
    console.error(error);
};
//Callback if the connection is closed
socket.onclose = function (closeEvent) {
    console.log('connection closed: ' +closeEvent.reason);
};
```

The advantage of this is that this code can be executed correctly on all compatible browsers without modification and without any additional framework (such as a Polyfill for older browsers). Furthermore, this WebSocket code directly allows detection and reaction to connection errors. WebSockets represent a solution to potentially minimize HTTP requests since bidirectional communication can be performed.

Comparing approaches, streaming with WebSockets appears to be superior due to a lack of limitations when compared to SSE and Server Push. However, this needs to be evaluated to ensure real-world implementations match these expectations.

Compatibility of Functions

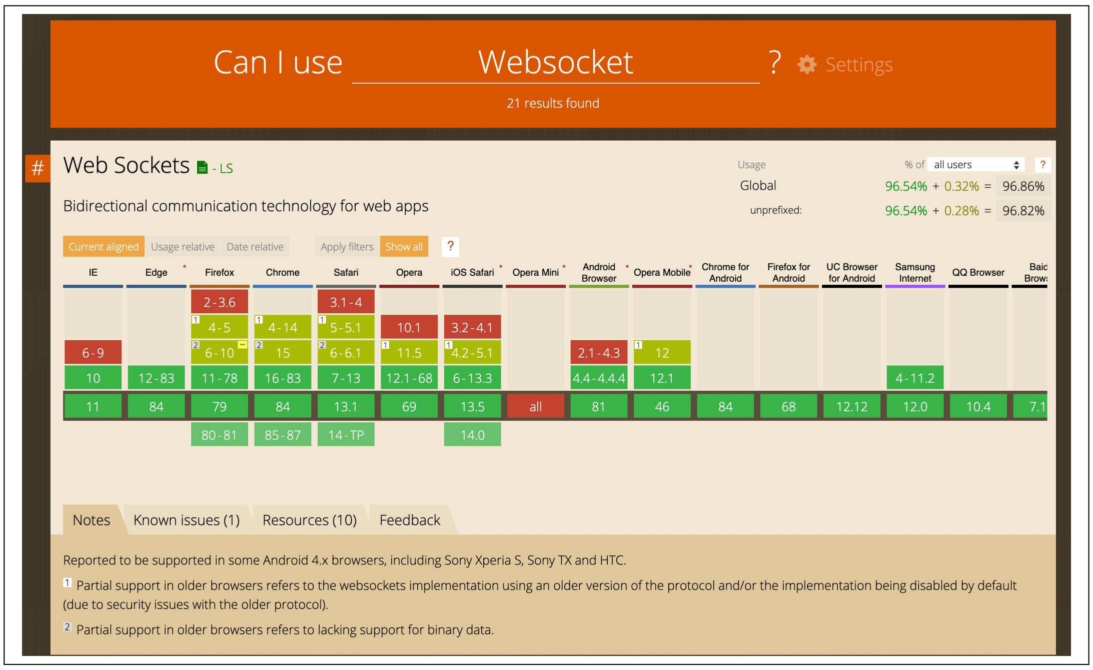
The introduction of browser APIs has created cross-browser interfaces standardized by the W3C [W3C20]. However, the implementation is not the same in all browsers. Resources such as `caniuse.com` provide compatibility tables for all default functions to check theoretical executability. These show, for a particular function or API, which browsers support them and since which version. Furthermore, information is provided on what percentage of browsers support the feature. Figure 2.3 shows this table as an example for WebSockets. It is visible that all current web browsers (except Opera Mini) support this feature. This web page is used to check the compatibility of the produced work.

2.6 Server Techniques

The following optimization techniques are executed mainly on the server side. This method's advantage is that no additional software usually needs to be installed on the users' devices. The existing browser is sufficient.

2.6.1 Server-side Optimizations and SSR

If the optimization is done on the server that provides the original code, the web page can structurally be delivered differently. One example is Nuxt JS, a popular framework for VUE



Can I use Websocket ? Settings

21 results found

# Web Sockets - LS

Bidirectional communication technology for web apps

Usage	% of all users	?
Global	96.54% + 0.32% =	96.86%
unprefixed:	96.54% + 0.28% =	96.82%

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baic Brows
		2-3.6		3.1-4											
		[1] 4-5	[1] 4-14	[1] 5-5.1	10.1	3.2-4.1									
6-9		[2] 6-10	[2] 15	[2] 6-6.1	[1] 11.5	[1] 4.2-5.1		2.1-4.3	[1] 12						
10	12-83	11-78	16-83	7-13	12.1-68	6-13.3		4.4-4.4.4	12.1				4-11.2		
11	84	79	84	13.1	69	13.5	all	81	46	84	68	12.12	12.0	10.4	7.1
		80-81	85-87	14-TP		14.0									

Notes | Known issues (1) | Resources (10) | Feedback

Reported to be supported in some Android 4.x browsers, including Sony Xperia S, Sony TX and HTC.

[1] Partial support in older browsers refers to the websockets implementation using an older version of the protocol and/or the implementation being disabled by default (due to security issues with the older protocol).

[2] Partial support in older browsers refers to lacking support for binary data.

Figure 2.3: Exemplary compatibility table of WebSockets
Source: caniuse.com/websockets [DS13]

[JS20]. With Nuxt JS, the web page is rendered beforehand on the server side. This is done by embedding attributes and JavaScript code directly in the HTML, which prevents having to load and render them on the client side. The only disadvantage for the user is that this technique optimizes only the specific website or web application. Until now, no universal server-side rendering can be used without a framework. However, server-side optimization is optimal for compatibility reasons since all the necessary resources of a web page can be accessed there (see Google Pagespeed) [Pag20].

Server-side rendering (SSR for short) is commonly associated with loading and rendering a website into the final state after executing all code on a web page. Sending a web page's final, rendered state can significantly reduce loading times, depending on the amount of JavaScript necessary to display the entire page. As a result, the HTML sent to the client includes a different DOM than the original. Therefore, the original JavaScript might break if it is not adapted to this new change. This form of rendering is sufficient for static pages without JavaScript but not dynamic web applications.

2.6.2 Middleware/Proxy

A middleware or proxy for optimizing web pages can be implemented in three locations. These differ in the possible type of optimizations that can be implemented in each step. Therefore, they are described separately below.

External Proxy

It is possible to create a proxy that optimizes web pages using additional hardware. Examples of this are Amazon's Silk browser or Puffin-OS [Shi11; OS20]. However, both have the disadvantage of either being able to optimize only unencrypted HTTP requests or breaking end-to-end encryption. Optimizing HTTPS requests is not feasible in this way since the headers are also encrypted [Res00]. Puffin-OS, in particular, uses its own apps, which access the corresponding proxies and come with the required certificates pre-installed. The encryption ends at the Puffin-OS proxy. This represents a security risk since all website data, including passwords, is transferred to the server [Müh19]. Accordingly, this is not an acceptable solution to optimize a website for data security reasons.

Client-side Proxy

To a certain extent, it is possible to perform client-side optimization of the website. A proxy, in this case, sits in the browser, most often as an Add-On, which can intercept network requests locally. One example of this is the popular uBlock browser extension, which makes it possible to remove advertisements and trackers from websites. In doing so, requests to the respective servers are blocked, which can improve loading times [Hil20]. Furthermore, it was shown that such client-side optimization could work without violating the encryption of a request [Vog18]. To achieve this, an Add-on removes various parts of the web page on the browser side before rendering (such as external resources) and loads them after displaying the page. The disadvantage is that for mobile scenarios, the Add-on can currently only be used on Android and requires the software to be installed. This is also not an optimal solution.

2.7 Performance Marker

Measuring web loading performance depends on the tested subject. For example, a web page can be marked as ”loaded” when all render-blocking resources are loaded, when all resources, including images, are loaded, or when the web page displays the first content. In this case, mainly user-centric performance markers are chosen.

2.7.1 First Contentful Paint

A website’s performance can commonly be measured with the help of the `Performance`-API, provided by Browsers. One of the markers is the First Contentful Paint (FCP). It marks the point in time when the first content appears on the screen. This timestamp is important as it is user-centric and labels the first time when a user can form a decision if the requested page suits the goal of visiting it [21]. It is, therefore, also an essential element in PageSpeed Insights, a popular tool developed by Google to measure the loading times of web pages [22af]. Various experiments in this work will reference the FCP, as it strongly indicates whether a page loads faster or slower and if the developed techniques improve the status quo.

2.7.2 Largest Contentful Paint

In contrast to the FCP, the Largest Contentful Paint (LCP) describes the largest appearance of content on the page. As described by web.dev, the LCP marks the time when the main content of a page has likely been loaded [22v]. While the FCP describes the beginning of rendering a page, the LCP can give better insights into the actual appearance. However, no metric is perfect, and the LCP does not guarantee that the content is actually usable. For example, if a large image is placed as a background on the web page and loads slower than the text, the LCP might not give valuable insight as it might detect the background but not the content.

2.7.3 DOM Interactive

The ”DOM Interactive” Performance marker might be one of the most controversial measurements provided by the performance API [22ag]. According to Ilya Grigorik on web.dev, it "[...] marks the point when the browser has finished parsing all of the HTML and DOM construction is complete." This, in theory, would allow scripts and users to interact with the page. [Gri22]. However, as noted by Steve Sounders, web performance pioneer and author of "High Performance Web Sites," this might not be a fully conclusive measurement [15]. This is only strengthened by the fact that asynchronous scripts exist. Therefore, the DOM might be interactive, but the necessary code to react to user input still needs to load. As a result, when using this performance marker to check for page interactivity, individual scripts' loading behavior must also be considered.

2.8 Initial Page Load

Client-side caching can significantly improve the loading speeds of web pages and even provide offline functionality via Service Workers, as they prevent repeated fetching of resources. However, these are ineffective if a client requests a web page or resource for the first time. Techniques that reduce the size of resources or improve their effectiveness are also helpful while caching, as they reduce their occupied caching space and, in turn, allow more resources to be cached. Therefore, this work aims to primarily focus on the resources themselves and not caching, as the overall benefit to page loading speeds is higher. This also implies that the developed techniques of this work target the initial page load, which starts with an empty cache.

2.8.1 DOM Content Loaded

The "DomContentLoaded"-Event marks the moment when the DOM is fully created, and no style-sheet file prevents JavaScript from running [Gri22]. In contrast, the FCP denotes the start of rendering user-visible content. This marker was chosen because it demonstrates advances in the execution of render-blocking resources, such as unaltered CSS referenced in an HTML document's `HEAD`. Depending on the browser's implementation, faster CSS execution will generally be noticeable in the "DomContentLoaded"-Event end marker. It will be considered in the evaluation as a result.

2.9 Existing Optimizations

Various optimizations already exist for all primary components of a web page. The following ideas are broad methods that either describe an entire area of optimization techniques or one that tries to achieve a multitude of optimizations together, which is why they are excluded from the categories described in the related work. Some examples of this are compression algorithms like *gzip* [18], modern image formats like *webp* [22f], or font subsetting [22ap]. However, optimizing the main render-blocking resources like CSS and JavaScript significantly impacts the loading time performance. Both will be examined in detail next.

2.9.1 CSS Rendering

In contrast to SSR, the rendering aspect can also be limited to CSS. In this case, the page is rendered similarly to SSR, but the only aspect changed is the included CSS. For this to happen, popular frameworks like "Critical" resort to using a remote-controlled browser on a server [add22]. All included CSS is extracted using the browser and visiting the targeted page. Then, the CSS is converted into an Abstract Syntax Tree (AST) representation to find all CSS selectors. Next, a library like "css-mediaquery" can be used to find every element matching a selector [22k]. Finally, all selectors with one or more matches are classified as "critical" for rendering, and the remaining CSS as "uncritical." By reducing the render-blocking CSS to the "critical" parts and delaying the rest, loading speed improvements can be made. In contrast to SSR, rendering CSS is highly unlikely to interfere with the JavaScript of the page due to the delivered HTML being the same. The only exception consists of code dependent on attributes set by the CSS, and only if it is marked as ”non-render-critical.” However, this case is highly improbable.

2.9.2 JavaScript Code Splitting and Dead Code Elimination

Dead code elimination is an area of (web) code optimization in which an algorithm tries to determine whether or not functions are used. The unused code is then removed, transferring less code to the client. Therefore, the dead code elimination is done preemptively. While this can be done reliably with code that relies on *import* and *export* statements via tree shaking, all other types of JavaScript code present a greater challenge [22at]. One reason is the multiple entry points and how JavaScript functions can be called. Especially the absence of knowledge about every possible state makes this problem difficult. Sophisticated dead code elimination tools like Muzeel try to emulate all possible user input [Kup+21]. However, in practice, not even this type of emulation is 100% correct and can break the functionality of a web page.

Therefore, another type of optimization emerges: code splitting. In this case, the code is separated into smaller chunks or files, which are then loaded delayed (via async or defer) or on-demand. The challenge is hereby to specify which part of the code can be delayed. This requires the developer’s input but can result in even better performance than dead code elimination if done correctly. However, it results in a higher initial effort.

2.9.3 Google Pagespeed

The 2015 published PageSpeed framework, developed by Google, is a plugin for `Nginx` and `Apache` servers. It promises to automatically modify and improve the initial loading times by enforcing as many best practices as possible [Pag20]. They include structural improvements like resource bundling or code optimizations like removing comments. In contrast to other frameworks like Amplified Mobile Pages, the PageSpeed platform does not require specific code to be written, which makes it easier to implement even in large code bases. However, no data exists on the performance and actual structural improvements which PageSpeed produces. This technology could be helpful in the development of the final concept. However, its capability has to be measured first.

3 Related Work

The related work for the following chapters, including the concept, is discussed in this chapter. The overview of all techniques and the structure of the following chapter is shown in Figure 3.1.

3.1 Research Method

Part of the challenge of researching related work for this and the following chapters was the trend of current articles related to streaming, primarily focusing on the streaming of video and audio. In this case, the search engines Google Scholar and Scopus were used to find scientific publications related to the streaming approach. However, both only returned a limited number of relevant articles when excluding the keyword ”video.” These are all evaluated in the following chapters. However, the search for related work was first focused on the sub-categories, described in section 4.2, section 3.2, section 3.3 and section 3.4. Next, the found articles are further used for a forward search using Google Scholar and a backward search utilizing the references of found articles. Due to the scarce source of (text-) streaming-related literature, no form of publication was excluded. Next, the same procedure was done using Google and GitHub, searching for related technical implementations. These resulted in a more extensive set of results. These results are then further researched via Google by adding the keyword ”alternative” to every relevant technique. This was then repeated until no more relevant techniques were found. After publishing the paper [VS22b], which targeted the text streaming itself, the published article was searched on Scopus, and the ”Related documents” feature was used to search for additional works. However, this also did not return any more relevant publications. To the best of our knowledge, all relevant techniques are discussed in this thesis.

3.2 Related Work for CSS Optimization

The following techniques are the foundation for optimizing CSS, discussed in more detail in section 5.3. Improving CSS can be done in various ways. For streaming, the following concepts are applicable: Reducing the amount of CSS transferred to the client, removing code duplicates (due to the cascading property of CSS), and detecting the location where CSS is used. Especially location detection is crucial, as only the necessary amount of CSS should be transferred for every section of HTML.

3.2.1 Frameworks

This first category of optimizations are based on some form of framework, or provide their own.

CSS Rendering in React

The popular React framework, developed by Meta (formerly known as Facebook), uses components to subdivide the logic of an application [Rea21]. These components often have their own CSS, which is loaded with the logic. This can create an overhead as the styling information is not loaded in one piece but as separate elements. The reason for this is the client-side logic handling. To transfer this concept to the server, another framework named `next.js` is frequently used [Ver21]. With `next.js`, components can be rendered on the server, including CSS. However, this technique mainly addresses the given overhead of React. The initial website will still load as a block and closely resembles a traditional one. Therefore, problems regarding the transmission of a website at low network speed remain.

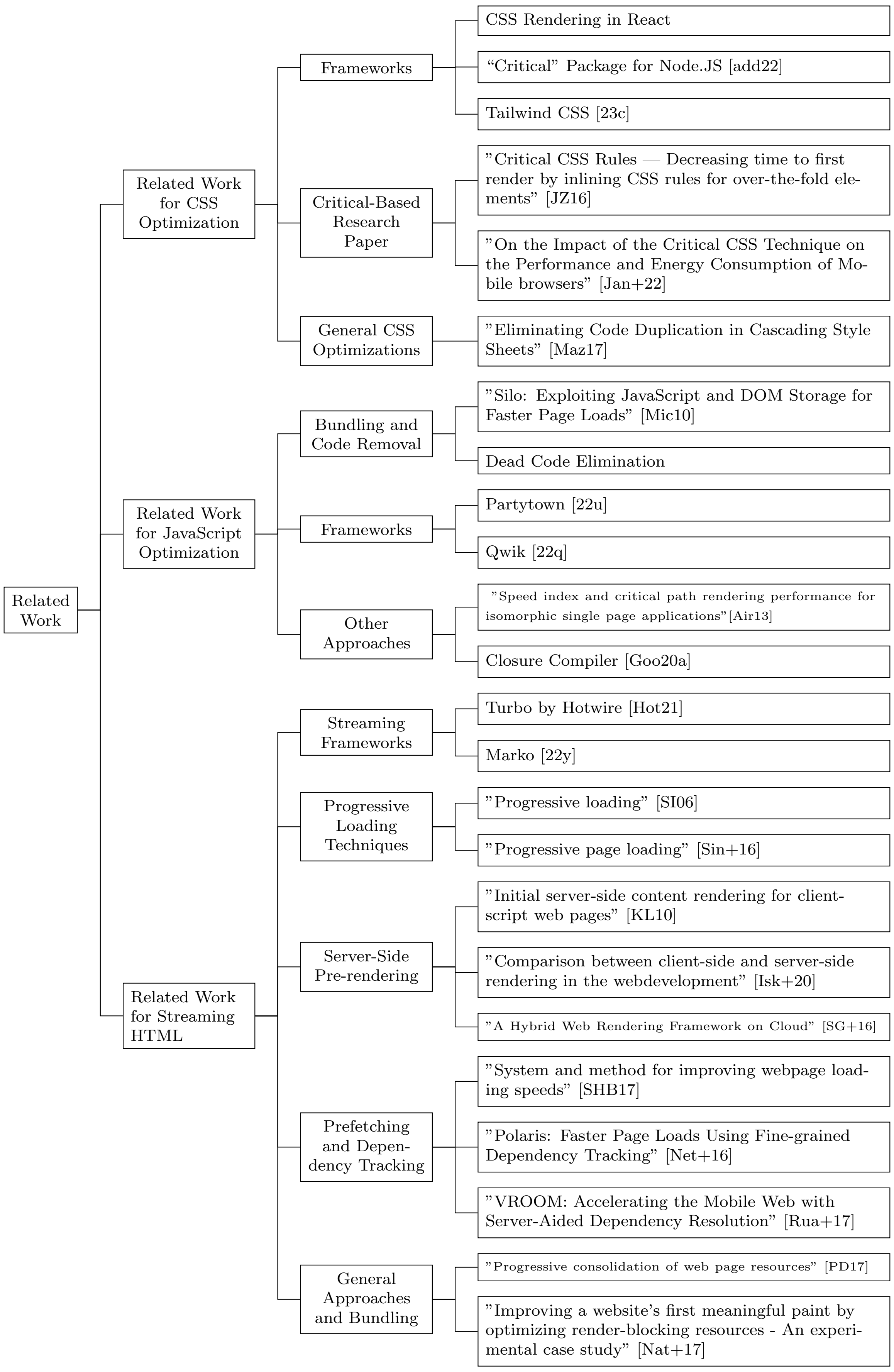

Related Work
Related Work for CSS Optimization
Frameworks
CSS Rendering in React
“Critical” Package for Node.JS [add22]
Tailwind CSS [23c]
Critical-Based Research Paper
”Critical CSS Rules — Decreasing time to first render by inlining CSS rules for over-the-fold elements” [JZ16]
”On the Impact of the Critical CSS Technique on the Performance and Energy Consumption of Mobile browsers” [Jan+22]
General CSS Optimizations
”Eliminating Code Duplication in Cascading Style Sheets” [Maz17]
Related Work for JavaScript Optimization
Bundling and Code Removal
”Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads” [Mic10]
Dead Code Elimination
Frameworks
Partytown [22u]
Qwik [22q]
Other Approaches
”Speed index and critical path rendering performance for isomorphic single page applications”[Air13]
Closure Compiler [Goo20a]
Related Work for Streaming HTML
Streaming Frameworks
Turbo by Hotwire [Hot21]
Marko [22y]
Progressive Loading Techniques
”Progressive loading” [SI06]
”Progressive page loading” [Sin+16]
Server-Side Pre-rendering
”Initial server-side content rendering for client-script web pages” [KL10]
”Comparison between client-side and server-side rendering in the webdevelopment” [Isk+20]
”A Hybrid Web Rendering Framework on Cloud” [SG+16]
Prefetching and Dependency Tracking
”System and method for improving webpage loading speeds” [SHB17]
”Polaris: Faster Page Loads Using Fine-grained Dependency Tracking” [Net+16]
”VROOM: Accelerating the Mobile Web with Server-Aided Dependency Resolution” [Rua+17]
General Approaches and Bundling
”Progressive consolidation of web page resources” [PD17]
”Improving a website’s first meaningful paint by optimizing render-blocking resources - An experimental case study” [Nat+17]

Figure 3.1: Overview of the related work chapter, classifying the approaches into multiple categories.

"Critical" Package for Node.JS

Multiple frameworks exist today which can render CSS on a server. These include `Penthouse` [23b], `Critical CSS` [23a], and `Critical` [add22], the last one being the most promising. The `Critical` framework will automatically render CSS for a page into "critical," and "uncritical" files, the "critical" ones being visible "Above-the-Fold" [Osm21]. However, all other CSS is delayed by only including the part "Above-the-Fold." This also includes CSS necessary to render the part "Below-the-Fold." This framework is, therefore, a valuable starting point. However, for streaming a page, such a framework has to be modified to render the full page and provide location-aware CSS, as the goal is to stream the entire page. Location-aware in this concept describes the method of CSS code being loaded directly before the HTML elements are parsed, which are styled by the respective section of CSS. In other words, the optimization script must be aware of the position of elements targeted by the CSS. Both aspects are not included in the current version of all mentioned Frameworks.

3.2.2 Tailwind CSS

The tailwind CSS library provides a different approach to writing and managing CSS [23c]. Instead of full-fledged CSS classes, it provides a significant set of classes mainly serving only a single function, for example, only centering text. Therefore, the goal is to ideally not require developers to write any CSS code themselves, instead relying on predefined tailwind classes. This allows tailwind to detect which classes are used in a code-base and generate CSS, which only contains the required code. This approach has the same goal as "Critical." However, tailwind currently requires a full rewrite of all CSS classes to the tailwind presets, as integrating existing CSS code at scale is not intended. Therefore, tailwind is not suitable as it requires significant developer effort.

3.2.3 Critical-Based Research Paper

The following subsection explores research papers describing and evaluating techniques similar or identical to "Critical."

"Critical CSS Rules — Decreasing time to first render by inlining CSS rules for over-the-fold elements"

This paper [JZ16] describes a technique of rendering CSS "Above-the-Fold," using PhantomJS to find all matching CSS declarations of an element. If the element is inside a predefined viewport (Above-the-Fold), it is marked as critical and will be part of the CSS included in the modified web page [VS22b]. This inlining works by collecting all critical CSS declarations and inserting them in a `style`-element in the page header. All remaining CSS is then asynchronously loaded with JavaScript once the page is finished loading [JZ16; VS22b]. This concept is implemented with a now-popular framework called "Critical," described next. In itself, the CSS rendering is insufficient for streaming, as it mainly optimizes files. However, such a technique can be modified to use it as part of a streaming-ready CSS renderer.

"On the Impact of the Critical CSS Technique on the Performance and Energy Consumption of Mobile browsers"

Janssen et al. described in their experiment that using Critical CSS can speed up the time until FCP and decrease loading times [Jan+22]. The evaluated websites were hosted on a Raspberry Pi, that also served as the test device. They processed 40 websites, randomly chosen using the Tranco list [Poc+18] and compared them to their unmodified original

Name	Usable?	Comment
CSS Rendering in React	❍	Not applicable
“Critical” Package for Node.JS [add22]	◗	Base idea useful, not streaming-ready
”Critical CSS Rules — Decreasing time to first render by inlining CSS rules for over-the-fold elements” [JZ16]	◗	Similar to Critical, not streaming-ready
”On the Impact of the Critical CSS Technique on the Performance and Energy Consumption of Mobile browsers” [Jan+22]	◗	Similar to Critical, not streaming-ready
”Eliminating Code Duplication in Cascading Style Sheets” [Maz17]	◗	Base idea useful, is only a prototype

● = complete solution, ◗ = partial solution, concepts applicable but not sufficient, and ❍ = no solution

Table 3.1: Overview over related work for CSS optimizations, comparing their usefulness for the concept of this work. Sources to the individual techniques are linked in the respective sections.

versions. However, this choice could affect the results' validity because the data set might be too small for reliable results. A list of the used web pages is also missing, and the version of the Tranco list was not provided, which is why the gathered data has limited conclusiveness.

3.2.4 General CSS Optimizations

This last category includes one method defining a more broadly applicable CSS modification. This idea can be applied to a broad spectrum of web pages and therefore defines its own category.

”Eliminating Code Duplication in Cascading Style Sheets”

Davood Mazinanian described in his thesis that code duplication is a significant problem of CSS [Maz17]. A solution was described for solving said issue. The thesis did not directly focus on stream-ability or render-blocking CSS. Its application reduces the amount of CSS data that needs to be transmitted, which also affects the loading speed of streamed web pages. When CSS code is modified, code duplication must be avoided as much as possible.

3.2.5 Summary of Related Work for CSS Optimizations

No existing technique can provide a universal solution for optimizing CSS as described for all related methods. This is also described in more detail in Table 3.1. The main problems are a lack of testing parameters or a small sample size. Current popular approaches are not using their full potential, as shown by the Critical and Critical-related techniques (section 3.2.1). For example, it is unclear if code duplicates are removed, as they can significantly improve loaded CSS's efficiency. Therefore, no ideal solution for optimizing CSS is currently available.

3.3 Related Work for JavaScript Optimization

The following techniques discuss future-oriented methods for optimizing JavaScript on web pages. In this section, current approaches for optimizing JavaScript are discussed. More specifically, this focuses on all techniques that reduce or eliminate render-blocking JavaScript code. Furthermore, their limitations are highlighted as well. In general, there are multiple ways how to improve web page loading performance. However, for streaming web pages, unoptimized code will delay the time until a user can interact with the page. Their applicability for streaming web pages will be discussed.

3.3.1 Bundling and Code Removal

This first sub-category includes methods that try to decrease overhead or try to delete unused JavaScript. Using different approaches, both aim to load the least amount of code with the least amount of requests.

”Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads”

This paper describes the problem that by opening a multitude of HTTP connections, the web page will load slower due to increased overhead [Mic10]. The authors also stated that caching is not an option because 40-60% of users will visit any given website with an empty cache. The proposed solution involves merging all resources into one file. By sending only a single file, the amount of HTTP requests is drastically reduced [Mic10]. A currently available framework for this kind of merging is `webpack`[web21]. However, this also introduces a different kind of overhead, as all resources must load first to display a web page. For example, deferred loading of scripts is not possible in this way. As a worst-case scenario, all data is transmitted, but no function is actually called. Therefore, it describes the opposite of the streaming approach, as all code is loaded before render.

Dead Code Elimination

One approach to optimize JavaScript is to detect and delete all code that is generally never used. This procedure is also described in more detail in subsection 2.9.2. Many different approaches exist which try to accurately detect unused JavaScript code [Obb+18; Kup+21; QL20; Cha+20; GS20; Cha+21]. However, none of the mentioned approaches manage to detect unused code perfectly. This is due to the high complexity of JavaScript. For example, functions can be called from the HTML directly via user interaction with inline calls. However, some HTML code might not be visible to the user, and the code will, therefore, never be executed. In this case, detecting whether or not the code will be used is difficult. One of the most sophisticated approaches, Muzeel, tries to counteract this problem by emulating all possible user interactions on a given page [Kup+21]. Still, no 100% accurate detection is being made, which is why this approach cannot be used, at least not in the current state. Only a 100% accurate detection algorithm is acceptable for any real-world web application, as only a single missing character in the wrong location can break the entire code.

3.3.2 Frameworks

This sub-category includes methods that are based on a framework. These techniques are mainly evaluated by their ease of use and optimization potential.

Partytown

The Partytown framework by builder.io is a solution for delaying JavaScript and executing it in a ”web worker” [22u]. The main problem of render-blocking JavaScript, especially third-party code, is the inability to modify the code itself while blocking the main thread of a browser. This blocking behavior results in longer loading times and unresponsive web pages. Partytown solves this issue by moving the code into a worker, which has a separate worker thread. The main issue of worker threads is the inability to make synchronous DOM requests. This is solved by the framework by exploiting synchronous XMLHttpRequests, which are already deprecated. Still, as long as they exist, Partytown can move code to another thread. When worker code tries to access the DOM, the synchronous request is made, which is, in turn, captured by a service worker. This service worker then redirects the request to the main thread, where the result is then returned via the initial request back to the worker. This presents a novel idea that can be used to delay code with minimal effort from the developer. However, even though the implementation effort is low, the possible improvements might also be low (especially when considering it for streaming), as it only coarsely splits code. Especially in large code bases, where the code is bundled, a more fine-grained option would enhance the performance improvement of a streaming approach.

Qwik

In contrast to Partytown, the Qwik framework allows for the maximum fine granularity [22q]. Both are developed by builder.io, but targeting different ends of the JavaScript optimization spectrum. In contrast to Partytown, the Qwik framework provides a platform to develop highly code-efficient web pages from the ground up. The only way to integrate Qwik is by rewriting the components of a web page. However, it results in a website that can delay nearly all of the JavaScript included in a page. Instead, the necessary sections of code are loaded on demand. This means that only a minimal part of the Qwik framework is needed for the initial page load. While this is a highly sophisticated approach, it might overshoot the need for more fine-grained optimization. Due to the lack of integration, it might also not be an ideal approach for splitting up JavaScript for web pages. Ideally, for a stream-based web page, a solution is created that combines the best of both worlds: acceptable developer effort for integration and manual control over the level of granularity.

3.3.3 Other Approaches

In this category, other methods for optimizing JavaScript are discussed. These include articles used as a leading example for all techniques using the same approach.

"Speed index and critical path rendering performance for isomorphic single page applications"

This paper discusses the idea of isomorphic code [Hak16]. The term isomorphic JavaScript was first described by Airbnb engineers in 2013 [Air13]. It describes a way to implement JavaScript that runs simultaneously on the client and server. With this setup, client-side HTML can be pre-rendered on the server, improving website loading times. The mentioned paper compared this approach to traditional loading behavior. The results showed that the performance of isomorphic web apps is better than traditional methods [Hak16]. However, this does not change how a website is delivered to the client. A web page with large dependencies will still load slowly on a slow network, which is why this approach is not viable for stream-based web page delivery.

Closure Compiler

Starting in 2013, the still maintained *Closure Compiler* developed by Google is a JavaScript transpiler that increases the performance of written code [Goo20a]. According to Google, it "compiles from JavaScript to better JavaScript" [Goo20a]. It works by applying various optimizations to the code, ranging from renaming variables to dead code elimination. However, the most effective modifications, like dead code removal, are only available in the "advanced" compilation level [Goo20b]. This requires the code to be written in a certain way, as the compiler has to make assumptions when processing the given code. If they are not met, the produced code might not run. Furthermore, code splitting is not performed. If some form of splitting technique is used, the closure compiler has to run separately for every chunk of code [22e]. This makes it a non-viable option for streaming pages due to a lack of optimization depth.

Name	Usable?	Comment
"Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads" [Mic10]	❍	Bundling is the opposite approach of the concept pursued in this work
Dead Code Elimination	❍	The detection is not accurate enough, can break entire page
Partytown [22u]	◗	Is easy to implement, but can lead to unwanted results
Qwik [22q]	◗	Produces excellent results, but developer effort is too high to achieve high coverage (requires rewrite)
"Speed index and critical path rendering performance for isomorphic single page applications" [Hak16]	❍	Does not solve the core problem
Closure Compiler [Goo20a]	❍	Limited results, would require adhering to code assumptions

● = complete solution, ◗ = partial solution, concepts applicable but not sufficient, and ❍ = no solution

Table 3.2: Overview over related work for JavaScript optimizations, comparing their usefulness for the concept of this work.

3.3.4 Summary of Related Work for JavaScript Optimization

As shown in Table 3.2, no existing method exists to reliably and universally optimize and split the JavaScript of a given web page. Neither bundling approaches (like Silo), compilers like the Closure Compiler, or Frameworks like Partytown or Qwik could achieve this goal without some form of caveat. Furthermore, the well-researched optimization strategy of dead code elimination proved insufficient due to a lack of detection accuracy. Therefore, no existing method provides an acceptable and universal solution.

3.4 Related Work for Streaming HTML

The most important related works concerning streaming and content delivery will be reviewed next. The techniques' ability to collectively optimize resources and stream content is thereby discussed. Streaming web pages requires a method to split and transfer the entire initial page. Furthermore, the technique needs to be backward-compatible in order for it to be adapted in real-world scenarios. Both will be discussed next.

3.4.1 Commercial Apps and Software

This first category consists of apps for mobile phones and other kinds of software which are distributed commercially.

3.4.2 Opera Mini

This browser, published by Opera as a separate app, aims to improve page loading speeds by compressing the requested data [Ope21]. It uses a proxy combined with Operas `Presto`-engine to compress data up to 90% and, with that, improve loading speeds [Bov15]. However, this approach does not fix the issue of the actual loading behavior. It only transfers data in a significantly more efficient form. With that, at certain compression modes, the feature set of websites is reduced as the website renders on a server. Furthermore, a very large website will still load slowly, as the compression only reduces relatively to the original website size. As far as publicly known, no streaming approach was chosen.

Puffin OS

This operating system, developed by CloudMosa, offloads almost all computation from the device to a server by rendering websites on a server [Clo20]. The device uses similar techniques to Browsers like Opera Mini, but on a system-wide level [Ope21]. All installable apps are, therefore, web apps. However, as far as publicly known, personal information like passwords or form data has to be sent to a puffin server [Kap19; Müh20]. This proposes security risks, as (for example) user sessions tied to a real person's personal account are stored on a central server, which can be a significant security risk. Furthermore, the user cannot control whether the submitted data is stored and handled safely. As this can be insecure, optimization using pure server-side rendering with session handling is not an option. Furthermore, no streaming approach is mentioned.

3.4.3 Streaming Frameworks

This next section discusses open-source frameworks which allow for streaming text-based HTML content.

Turbo by Hotwire

Turbo and especially *Turbo Streams* is part of a series of frameworks provided by Hotwire [Hot21]. The core idea is to send page changes via streams from the server to the client. This significantly decreases the amount of JavaScript needed on the client, as the server can render the page modification. For this to happen, the page is split into `<turbo-stream>`-sections, which are self-contained. These sections can be updated with HTML sent mainly via WebSockets or SSE. One example is updating the inbox of a mail web client if new mail arrives. While this is directly in line with the concept presented in this work, it only applies to updates, not loading the initial page itself. The initial page is still loaded as a block. While it is a promising idea, it does not reach far enough.

3.4.4 Marko

Starting in 2014, the Marko web framework uses SSR to render web pages and allow for streaming components [22y]. The core idea is to allow long-running tasks, like fetching APIs, to occur on the server while the client renders the rest of the page. When the long-running task is finished, it can then be streamed to the client. In theory, this solves a multitude of challenges for streaming web pages. However, it also comes with significant caveats. Most importantly, it is not intended to stream the entire page, as the focus lies on "filling the gaps" by delivering results in a delayed fashion[22y]. Secondly, similarly to section 3.4.3, the whole code-base must be rewritten, which is unfavorable. Next, the streaming approach used (chunked transfer encoding) is deprecated in HTTP/2 [BPT15a]. This can lead to problems with CDNs and proxies, which the developers acknowledge [22au]. Lastly, HTML is not directly split, as loading elements is done on a component-based level, which requires additional client-side JavaScript [22as]. Therefore, it is not a viable solution for streaming HTML.

3.4.5 Progressive Loading Techniques

The following techniques all describe methods of progressively adding content to a web page or similar form of digital information access. The techniques that do not explicitly mention web pages are included, as they lay a foundation of what is possible when transferring the method to web pages.

"Progressive loading"

The patent describes a system that only loads the currently visible elements of a web page [Sch07]. Elements outside of the currently visible area are not loaded. This does not consider that even those elements can be separated into smaller portions. No information is given on how this system will split the original web page into those segments and how it is ensured that those elements load correctly. The provided images only show an asynchronous loading of images on an already loaded website. The patent does not include streaming and is expired.

"Progressive page loading"

This active patent from Microsoft claims the method of loading pages of a (digital) document progressively [Sin+16]. It describes that digital documents and sets of documents can contain large amounts of pages. However, not all pages might be needed. To solve this, a system is designed that can provide individual pages or parts of pages to a user. As other pages are requested, the software on the client device loads them progressively [Sin+16]. This system is not designed for web pages. However, a similar approach can be made with parts of a DOM tree. As described by the patent with a focus on documents, websites can also contain large amounts of data. This technique can be adapted to progressively load or stream parts of websites that are needed by a user.

3.4.6 Server-Side Pre-rendering

The next set of approaches use a form of server-side pre-processing in order to improve loading speed.

"Initial server-side content rendering for client-script web pages"

This Microsoft patent describes how a website's text that is constructed and loaded by JavaScript can be rendered on a server [KL10]. The system described uses a form of client-side emulation to execute the JavaScript beforehand and to extract the loaded content [KL10]. This patent does not describe ways to render other elements like CSS. The patent does not include streaming and is expired.

"Comparison between client-side and server-side rendering in the webdevelopment"

This paper compares the pre-rendering of websites on the server as well as JavaScript-based loading on the client [Isk+20]. It argues, based on speed measurements, that server-side rendering is faster and improves the SEO of the page, as it is easier for web crawlers to read the content of the website [Isk+20]. However, no indication is made that the server-side rendering will also include CSS rendering. Therefore this paper is missing crucial information on the scope of optimization and details on the used test setup. No streaming approach is discussed.

"A Hybrid Web Rendering Framework on Cloud"

The authors of this paper state that browsers like Opera Mini or UC Mini that use "cloud-assisted rendering" have shortcomings regarding rendering quality or animations [SG+16]. The proposed system translates the web page in different layers and maintains CSS attributes like animation timing and z-order as an XML format [SG+16]. The client then renders the layers. It can do so independently, receiving the data as rendering instructions. The web page can be re-rendered for changes in resolution or zooming on the client. This approach can reduce the used bandwidth by up to 47%, and the authors filed a patent claim for this method. However, client-side JavaScript execution is not discussed in this paper. If

JavaScript needs to access the DOM, new drawing instructions are needed, as the created XML-Version does not preserve or transmit the original DOM. Even though the loading time is faster, not the full feature set can be provided on the client. Furthermore, this concept does not address streaming itself.

3.4.7 Prefetching and Dependency Tracking

The following methods aim to improve loading time by optimizing the loading behavior of dependencies. This includes loading them before they are used (prefetching) and also dependency tracking for optimizing the loading order.

"System and method for improving webpage loading speeds"

This patent uses several techniques to optimize the loading speed of a website via a proxy [SHB17]. These techniques include (amongst other things) heuristic prefetching and multiple simultaneous connections to one server, to load the resources. The goal is to optimize web pages universally. However, the end-to-end encryption has to be broken on HTTPS requests, as the header is also encrypted. In this form, no universal optimization is acceptable.

"Polaris: Faster Page Loads Using Fine-grained Dependency Tracking"

The authors of this paper describe the issue that websites have to load dependencies, which themselves might require further resources [Net+16]. This dependency graph is render-blocking, and the browser has to request all dependencies first. However, the current method of loading these dependencies is insufficient. In this paper, two methods were presented. First is a fine-grained analyzer that can search the included JavaScript for further dependencies. Secondly, a client-side scheduler called Polaris that can decrease the median loading time by 34% by using the knowledge gained beforehand. This is a universal way of improving the fetching of resources. However, no optimization is made to the resources. A website with large dependencies will still load slowly, as it is only faster in reference to the original page. No streaming was mentioned.

"VROOM: Accelerating the Mobile Web with Server-Aided Dependency Resolution"

The current loading behavior of websites requires clients to fetch all dependencies of a web page. However, the authors of "VROOM" [Rua+17] state that there is idle time in between these requests, as the client only loads them at the time the parser encounters them or when the JavaScript sends the request [Rua+17]. The proposed solution sends a list with the initial HTML file and the URLs of all future dependencies. The client is, therefore, capable of fetching all dependencies, even though the parser did not encounter them yet [Rua+17]. The problem with this system lies in network conditions. The speedup only helps if the fetching of resources is faster than the parser of the browser. With decreased network speeds, the saved time decreases as well. Additionally, the initially transmitted data includes an overhead, as the resource list has to be fetched with it. The concept does not mention streaming.

3.4.8 General Approaches and Bundling

This last category of optimization methods includes other forms of optimizations that do not belong to one of the before-mentioned categories or span multiple categories simultaneously as they evaluate multiple techniques.

"Progressive consolidation of web page resources"

This patent by Akamai describes an improvement in website loading speeds by consolidating common resources like JavaScript or CSS [PD17]. Furthermore, the patent also claims that those dependencies can be used without fully loading them onto the client device. Therefore, the website can load faster. However, the patent describes no improvement to the resources or the HTML-Page containing them. It might also be possible to slow down loading speeds, as one of the claimed points refer to sub-pages being loaded into i-frames, which might induce an overhead at low network speeds. The concept does not include streaming.

"Improving a website's first meaningful paint by optimizing render-blocking resources - An experimental case study"

This master thesis project aims to reduce the loading time of web pages by eliminating unused CSS and delaying JavaScript [RN17]. As this is an experimental study, the authors did not automate the process and instead used various existing frameworks and tools to achieve this goal manually. Their experiment found that the render time can be reduced by using both techniques. However, critical data is missing. In the timing evaluation, no details are given on what website was used to test the technique. Furthermore, the technique does not focus on streaming. As this might be a suitable base technique for automation, more details are needed.

3.4.9 Summary of Related Work for Streaming HTML

As shown by the significant number of commercial products, frameworks, and research papers, the topic of streaming HTML and optimized content shows great potential. This is also described in more detail in Table 3.3. However, none of the discussed techniques sufficiently solves the issue, as in most cases, the resources are not modified and split. For example, Opera Mini only compresses the files, while VROOM works by dependency tracking. Methods that do modify the actual JavaScript require a rewrite of the code-base, as shown by Marko and Turbo. Both yield the most promising optimization potential. However, the streaming and delivery do not go far enough in both cases. The two frameworks do not allow streaming the initial page without loading a significant amount of JavaScript first. Therefore, no technique exists to fully stream a web page.

3.4.10 Summary

The individual sections of the related work show that no concept exists that fulfills the requirement of streaming web pages completely. The individual sections will be discussed in more detail in their concept chapters respectively. However, in general, no acceptable automatic optimization for JavaScript could be found, with Muzeel being the closest (but not yet sufficient and acceptable) candidate. For CSS, methods like `Critical` can universally optimize CSS. However, it lacks the required location of use detection, which is essential for streaming web pages. Furthermore, no HTML splitting or streaming method exists which can stream the entire initial page, with *Turbo* and *Marko* being close candidates. Both could not be used, as they do not provide the initial streaming capabilities and require a complete page rewrite.

Lastly, it is unclear to which proportion each of the three render-blocking resources (HTML, JavaScript, and CSS) affect the loading performance to which extent. Not only is there a lack of existing solutions for streaming web pages. The amount of optimization potential for JavaScript and CSS is unclear as well. For this, an analysis is required, which is described next.

Name	Usable?	Comment
Turbo by Hotwire [Hot21]	◗	Valuable concept, not far enough, does not stream initial page
Marko [22y]	◗	Valuable concept, not far enough, does not stream initial page
”Progressive loading” [SI06]	◗	Valuable concept, not directly applicable
”Progressive page loading” [Sin+16]	◗	Valuable concept, not directly applicable
”Initial server-side content rendering for client-script web pages” [KL10]	❍	Missing information, too vague
”Comparison between client-side and server-side rendering in the webdevelopment” [Isk+20]	❍	Too vague
”A Hybrid Web Rendering Framework on Cloud” [SG+16]	❍	Highly complicated process, limited use
”System and method for improving webpage loading speeds” [SHB17]	❍	Security risk due to broken encryption
”Polaris: Faster Page Loads Using Fine-grained Dependency Tracking” [Net+16]	❍	No splitting and not streaming-ready
”VROOM: Accelerating the Mobile Web with Server-Aided Dependency Resolution” [Rua+17]	❍	Limited usability at slow network speeds
”Progressive consolidation of web page resources” [PD17]	❍	Bundling is an anti-pattern since HTTP/2 [22ar]
”Improving a website’s first meaningful paint by optimizing render-blocking resources - An experimental case study” [Nat+17]	◗	Good proof of work, only manual prototype

● = complete solution, ◗ = partial solution, concepts applicable but not sufficient, and ❍ = no solution

Table 3.3: Overview over related work for splitting and streaming HTML, comparing their usefulness for the concept of this work.

4 Analysis

The following section describes a large-scale analysis of the current state of the art due to a lack of existing data on how much HTML, JavaScript, and CSS affect the render-blocking loading behavior of web pages. As described beforehand, various optimizations exist to improve loading times. However, it is unclear how much each component influences the time until a web page is loaded and usable. First, the question will be answered why this analysis is necessary. Next, related analysis results are discussed, followed by details about the targeted data which needs to be extracted. Next, the test software structure is explained, followed by the evaluation of the gathered data. Finally, the conclusion summarizes the findings and highlights areas with significant improvement potential.

4.1 Necessity of Analysis

Before creating a concept on improving the loading speed of web pages, the optimization potential has to be measured first. In this analysis, the main focus lies within the render-blocking properties of JavaScript and CSS, which, together with HTML, build the foundation of modern web pages. If no optimizations like delaying or splitting code are used, web pages will load slower. However, due to the diversity of the modern web and especially web applications, the question changes from "How fast are web pages?" to "How fast can web pages load if the code is optimized?". In other words, data is missing on how efficiently the delivered code is used by web pages, especially when focusing on render-blocking resources. In order to develop the best possible version of stream-based web page loading, the current state and remaining potential will be determined first. Furthermore, the impact of frameworks will be measured as well. The following analysis results are published in [VS22a].

4.2 Related Analysis Approaches, Methods and Limitations

This section discusses related analysis approaches, as well as existing techniques. For both, the pros and cons are highlighted, focusing on the limitations related to the before-mentioned goal of analysis: analyzing web pages structurally, and determining code efficiency (how much code is used until render), render-blocking properties, the impact of popular frameworks, and differences between desktop and mobile.

"Structural Profiling of Web Sites in the Wild"

The authors analyzed the DOM structure of the internet's 500 most popular web pages [CH20]. A filtered list from moz.com was used. Some of the analyzed attributes were the node depth, the name of the tags, if an element is visible or not based on the current viewport, the location of an element, and the CSS selectors that target a given element. In order to analyze said pages, the browser extension "TamperMonkey" was used to inject additional JavaScript into the pages, which in turn analyzed the page inside a browser context. Their results showed that, on average, half of the DOM elements are not visible to the user and, therefore, possibly unnecessary. However, the paper lacks context for other results like the number of nodes or maximum DOM depth. It needs to be clarified what the measured data means, especially for the page load. Information like setting the gathered data in relation to the page load is missing. The authors acknowledged this by stating that the paper serves as a future reference.

Wappalyzer

The core software and website of Wappalyzer provide insights into what tools and frameworks were used when building a given web page [Wap21]. It analyzes the code base of a website by searching markers that hint at the usage of a specific tool. With it, an analysis can be broadened by including the results of Wappalyzer, as measurements can thereby be connected with the frameworks used. Wappalyzer will therefore be an essential part of the following analysis.

HTTP Archive

One of the most significant online resources about web performance and statistics is the HTTP Archive [HTT21a]. At the time of writing, the sample size for the monthly reports exceeds 10 million pages on desktop and 15 million pages on mobile, making it one of the most powerful web structure analysis platforms. By utilizing a hosted instance of Web Page Test [Mee21], in-depth analysis data is fetched. Additionally, Wappalyzer is used to enrich the data with the frameworks used. While this data exists, there is a lack of focus on render-blocking resources, and any form of evaluation in this area is missing. Even if this data existed, the raw data is only accessible via Google Big Query, which is a paid service. Therefore, the data provided by the HTTP Archive is not sufficient.

Summary

In general, no available framework or data set sufficiently provides insights into what optimization potential remains. Even the most sophisticated attempts are either not freely available or analyze the given topic insufficiently. Therefore, a large-scale analysis was needed to focus on the mentioned aspects: extract the remaining optimization potential and evaluate the impact of popular frameworks.

4.3 Overview Over All Structural Aspects Which Directly Impact Render Time

Multiple aspects affect loading speed, for example, server response times. However, this is the result of individual provider and developer choices. Generally, these aspects are subject to change based on external factors, e.g., distance to the server, time of day, or server load. As a result, measuring the absolute time it takes to load a web page only partially returns meaningful data. A better, more cause-oriented approach is to look at the page structurally and the choices that will affect loading times. By doing so, more meaningful measurements will be made, independent of external factors. According to web.dev, two types of additional render-blocking resources directly affect loading times: unmodified loading of external JavaScript and external CSS files [20]. The main resource, the HTML file, is, therefore, also at least indirectly render-blocking. If fewer data is needed to be transferred to display a page, the faster the loading will be. Therefore, both JavaScript and CSS and their render-blocking property will be the main focus of the upcoming measurements. The main aspects that will give clues into the remaining potential are the following:

1. Portion of code that is render-blocking
2. Total size percentage of the render-blocking resource
3. Percentage of code that is used until the rendering of the page (efficiency)

In detail, aspect 1 includes checks for how many external files are loaded, how they are loaded, and at which location they are linked in the HTML file. The location aspect gives a clue if advanced splitting techniques are already used. Aspect 2 affects loading time as the size of the render-blocking file directly impacts download time. Lastly, aspect 3 determines if the code is actually necessary for rendering. For example, a large, render-blocking JavaScript file that is used 100% until the page is displayed will impact the time it takes to download, but it cannot be avoided. As a result, no generalized statement can be made. Therefore, all three aspects will be used as a base for the next measurements.

4.4 Type of Measurements

The following measurements are equivalent to what was published in paper [VS22a]. The seven aspects are selected for their importance towards page loading times, described in more detail next.

1) Number of external and internal JavaScript-blocks
Multiple ways exist to include JavaScript in a web page. The first option is to in-line the code via special attributes, for example, with `onclick=""`. The second option is to insert the code via a code block, using the `<script>`-tag. The third option consists of externally linking a JavaScript file. Only the external file can be loaded asynchronously by using the `async` or `defer`-attributes. One exception are JavaScript modules, which are deferred by default. This measurement does not only give insight into the ratio between external and internal code blocks but also into the number of external files. A large number of (render-blocking) external files can lead to slower loading times, as the available bandwidth of the client will be shared to download the files. In turn, a small number of external files hints towards the usage of optimizations like bundling, where all JavaScript files are combined into one large file to reduce round trips for the requests. A large number of internal blocks hint towards optimizing this step even further by eliminating the request completely. However, it also can result in slower loading times due to their render-blocking property if they are executed on the main thread.

2) Efficiency of JavaScript
While the quantity of render-blocking code on a given web page indicates slow loading times, it does not represent the full story. For example, it is possible that a web application loads a large quantity of JavaScript but uses every single loaded function. Or, as another example, a web page loads an average amount of JavaScript but does not use a single piece of code. To find out how much JavaScript code is used until the page render, the efficiency will be tested. Ideally, every single character of render-blocking JavaScript will be executed before the page is rendered. Therefore, the efficiency is calculated by the used characters divided by all characters. If the percentage is small, it can be assumed that there is still a significant amount of optimization potential left.

3) Efficiency of CSS
Similarly to measuring the efficiency of JavaScript, unused and render-blocking CSS will also slow down the page. However, there is a lack of methods on how to load CSS asynchronously, except for the limited functionality provided by the `media`-attributes. Frameworks like `Critical`, therefore, resort to using JavaScript in order to enable this behavior. For CSS, the optimal case would be 100% of the CSS code being used until the page render, which is why the efficiency will be calculated in the same way: dividing the used characters by all (render-blocking) characters. The lack of asynchronous loading options combined with the benefit of using CSS frameworks leads to the assumption that the measured efficiency might be lower compared to the efficiency of JavaScript. CSS frameworks allow for convenient usage of pre-styled elements but add an additional overhead compared to individually created style-sheet files. It is unlikely that all functionality of a CSS framework is used on a single page. Furthermore, to the best of our knowledge, there is no simple and popular tool to detect CSS that does not affect the page's visual appearance.

4) CSS locations
As with JavaScript, the CSS of a page can also be inserted in multiple ways. The first and most direct version uses the `style=""`-attribute active on various HTML tags. Secondly, CSS can be inserted via the `<style>`-tag. Lastly, external CSS files can be linked in the `<head>`

		Test	Number
Tests on Desktop and Mobile	JavaScript	Number of external and internal blocks	1
		Efficiency & Render-blocking and non-render-blocking property	2
	CSS	Efficiency	3
		Link locations	4
		Render-blocking and non-render-blocking property	5
		Matches per CSS rule	6
	HTML	Resource distribution & Element positions	7

Table 4.1: Overview over the measurements as they are described in subsection 4.4.1, showing the groups to which every test belongs

of an HTML page via the `<link>`-element. On the one hand, this measurement will give insights into how much external or internal CSS code is used. On the other hand, a higher number of external files might indicate that the page uses multiple frameworks, as there is no technical necessity to split the code. For example, using *swiper.js* as a complimentary slider framework for images on a web page requires additional CSS [1]. This is also provided by the framework itself, even as a CDN-based link. Therefore, using a large number of external files might indicate less CSS optimization effort.

5) Render-blocking and non-render-blocking CSS
In contrast to JavaScript, only limited methods are available to delay CSS due to the lack of options like the `media`-attribute. As the most popular option is using JavaScript (like `Critical` is using [add22]), it represents an additional effort for the developer. As with JavaScript, all internal CSS is render-blocking. This measurement will check how much of the total CSS is render-blocking and test if a widespread option exists to delay CSS code. This is crucial for the concept of this work, as non-render-blocking CSS improves the time until FCP.

6) Matches of CSS selectors
CSS is comprised of rules which use different selectors to affect parts of the page. These selectors can be broad, for example, selecting every element of a given tag name, or specific, like selecting one single element via the provided ID. A rule affecting a larger number of elements or just one would not make a difference for the calculated efficiency in measurement 4. A large number of rules with just one selector would increase efficiency more than a single selector with a significantly higher number of uses. Therefore, the matches have to be checked as well to ensure accuracy.

7) Resource distribution and element positions
Streaming HTML content requires distributing the resources so they can be loaded on demand. Therefore, it has to be checked how (mainly external) files are loaded. Additionally, it is crucial to check the size of the `<head>`-element, as it might need to be loaded first due to meta-information like encoding. If the results show that it exceeds the size of the `<body>`, then the concept has to be adapted to consider mixing the data or filtering and sending the most important parts affecting the visible part of the page first. For example, the `manifest.json` of PWAs does not affect the functionality of a given page [2], but the meta-information about the encoding used on a page can. Therefore, the distribution of external resources is monitored.

[1]swiperjs.com/get-started
[2]web.dev/learn/pwa/

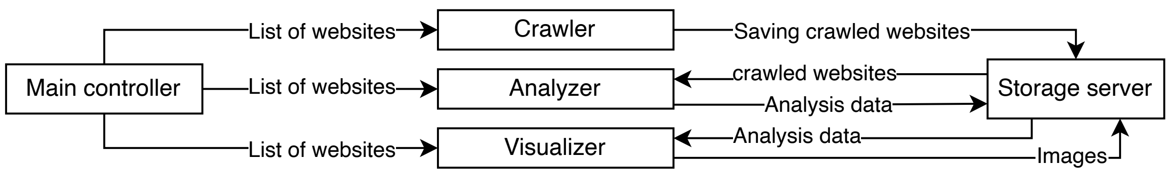


Figure 4.1: Structure of the analysis software. Source: [VS22a]

Additional Test Parameter: Desktop and Mobile
Devices with varying types and screen sizes can result in different code behavior. Responsive design can lead to CSS not being used due to invisible elements, such as a menu that is only visible on mobile clients. Therefore, all measurements are tested for desktop and mobile platforms.

4.4.1 Measurement Summary

Table 4.1 shows the individual measurements visually grouped by their common category. It is visible that the tests mainly focus on JavaScript and CSS as those are types of external resources that can block the rendering of the web page.

4.5 Crawling Limitations

Due to availability, it is inevitable that some pages cannot be downloaded. There are multiple reasons for this behavior. For example, a page might go offline, the connection is slow due to accessing a server from another part of the world, or the page is unavailable from the region accessing the page. Furthermore, some pages might not be able to be included in some of the measurements mentioned in section 4.4. Due to this uncertainty, the number of pages will be named for every test, with n_d for desktop pages and n_m for mobile.

4.6 Test Setup

In order to measure all parameters, the top 10.000 web pages of the research-oriented Tranco-List were crawled and evaluated (downloaded April 29, 2021) [Poc+18]. The structure of the analysis software itself is shown in Figure 4.1. The main controller first sends sections of the full list of web pages to a given number of crawlers. They then download the whole page and store it on a server for processing. Multiple crawlers are used in parallel to speed up this process. After every page is downloaded, multiple parallel analyzers are used to process the list of downloadable pages. Finally, one instance combines all analyzed data into one file. Figure 4.1 further shows a visualization component. This part of the framework is used to produce graphs in order to find patterns and correlations within the gathered data.

4.7 Technical Implementation

The crawlers were implemented using the `puppeteer`-framework for accessing and downloading a given website [22ah]. `Puppeteer` allows for remote-controlling headless Firefox or Chromium browsers. In this case, the default Chromium browser was chosen, as this choice does not affect the page structure itself. For the analysis itself, a feature was used that allows for accessing the page's final DOM after JavaScript execution. First, the remote-controlled browser waits until the *networkidle0*-marker. According to the puppeteer docs, this is the point in time when no network connections are active for more than 500ms [22ah]. Then, JavaScript is injected in order to extract the targeted data. Puppeteer also allows

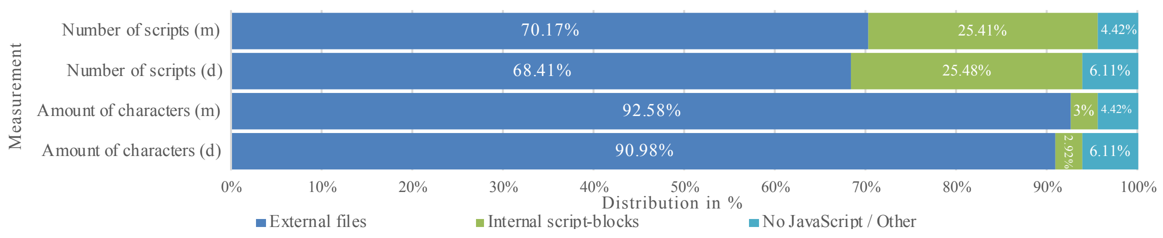


Figure 4.2: The average percentage of JavaScript locations in number of files/blocks and characters on desktop (d) and mobile (m). Source: [VS22a]

for accessing the execution coverage of CSS and JavaScript. This coverage represents the code that is executed until the measurement ends, which is therefore used to gather data for measurements 2 and 3. The page's HTML is analyzed using the `htmlparser2`-library and the used libraries with `Wappalyzer`.

4.8 Results

The measured data will be discussed in order of appearance in section 4.4. When the performance differences of the most popular frameworks are discussed, `React`, `jQuery`, `Angular`, and `Vue` are mentioned in this order. This is based on a Statista survey for the most popular web frameworks by extracting the four most popular tools that are front-end only [Sta21]. The following data is also published in paper [VS22a].

1) Number of external and internal JavaScript-blocks:
This measurement includes data from n_d=8417 and n_m=8468 pages. On average, external files account for 68.41% of all included JavaScript blocks and 70.17% on mobile devices. The distribution of external and internal sections are shown in Figure 4.2. There, it is visible that 25.48% are internal script blocks on desktop and 25.41% on mobile. More crucially, on desktop, external files make up 90.98% of characters compared to all JavaScript characters on an average page (92.58% on mobile). This shows that external files might account for fewer blocks but are responsible for the vast majority of code.

A large amount of external JavaScript does not directly indicate if a web page will load slowly, as it could also be delayed and, therefore, would not impact the render time. However, the data showed that 73.73% of the external JavaScript blocks are render-blocking on desktop (76.25% on mobile, n_d=8417, n_m=8468) [VS22a]. In general, when including external and internal JavaScript, 91.78% of JavaScript is render-blocking (on desktop, 93.36% on mobile, when comparing the number of characters). Therefore, nearly three-quarters of all loaded external JavaScript is slowing down the time until FCP, which indicates major optimization potential.

Filtering the results by the technology used, some differences can be detected. 74.74% of all pages with at least one detected framework use more than 95% render-blocking JavaScript (on desktop, 87.37% on mobile). The least amount of render-blocking code were pages built with the "Twitter Flight" framework, with only 73.42% render-blocking JavaScript on desktop and only 51.61% on mobile. In contrast, pages that use "AlloyUI" (now deprecated) or "BEM" used 100% render-blocking JavaScript on desktop and mobile platforms [Edu22; Vse22; VS22a]. This shows that optimizing render-blocking JavaScript is possible but is still not widely used.

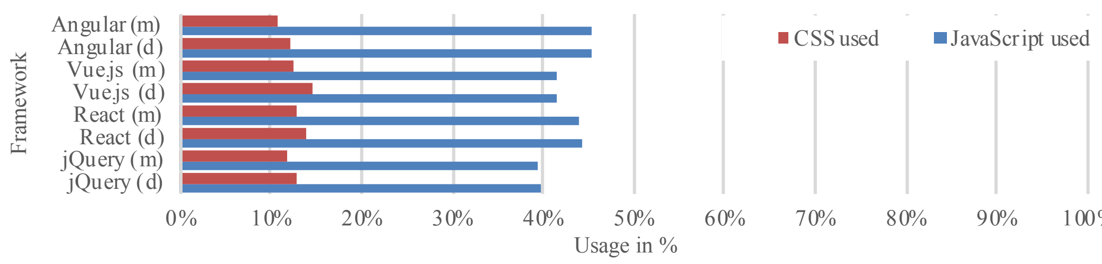


Figure 4.3: The average percentage of CSS and JavaScript used until `networkidle0` on desktop (d) and mobile (m). Source: [VS22a]

Framework	Number of websites detected	Percentage of external files (in characters)	Percentage of render-blocking scripts (in characters)
React	n_d: 1413 n_m: 1457	d: 95.51% m: 97.68%	d: 94.56% m: 96.65%
jQuery	n_d: 5140 n_m: 5222	d: 97.16% m: 98.24%	d: 96.46% m: 97.39%
Angular	n_d: 378 n_m: 373	d: 94.24% m: 98.65%	d: 91.46% m: 95.53%
Vue	n_d: 418 n_m: 432	d: 97.47% m: 98.36%	d: 96.59% m: 97.29%

Table 4.2: Comparing values in percent of the most popular JavaScript-frameworks. d = desktop, m = mobile. Source: [VS22a]

The differences are less prominent when focusing on the four most used JavaScript frameworks. The individual results are shown in Table 4.2. Most notably, jQuery is the most popular front-end framework. The usage of render-blocking scripts of React, Angular, and Vue only differ by 5.13% on desktop and 1.86% on mobile, with all frameworks being more than 90%. This could also be due to the fact that some of the frameworks can be used in combination. For example, Angular can be used with jQuery. All four frameworks also use over 90% externally linked JavaScript files, on both desktop and mobile. Therefore, it is shown that even the most popular frameworks are not designed to optimize render-blocking JavaScript, as they allow their co-existence.

2) Efficiency of JavaScript:
The efficiency of JavaScript in this context is calculated by the number of characters used until render divided by the total detected JavaScript characters. In a perfect scenario, the page uses 100% of the render-blocking code until the page is displayed.

When considering both render-blocking and non-render-blocking code, on average, 40.81% of all code is used until render on desktop and 40.61 % on mobile [VS22a]. Examining only render-blocking code reveals that there is no significant difference, with 40.94% of code being executed on desktop and 40.65% on mobile (with n_d=7741 and n_m=7741). The code was measured until `networkidle0`.

Differences are minimal but present when considering only the four most popular JavaScript frameworks. As shown in Figure 4.3, both Rect.js and Angular improve code usage by ≈3-5% compared to the other two frameworks. The lowest usage was measured on pages using jQuery, which have less than average code efficiency. This could be explained by jQuery having an extensive feature set, but its most popular feature of element selection and DOM modification make up only a small percentage of the total framework. Therefore, not all loaded functions are used, which decreases efficiency.

3) Efficiency of CSS: As speculated in the measurement mentioned above 3, the efficiency of CSS is lower compared to JavaScript. The efficiency is also calculated by dividing all used CSS characters by the total number of characters. When measuring until `networkidle0`, on desktop, 15.86% of all CSS is used and 14.84% on mobile (n_d=7739, n_m=7739). This includes both render-blocking and non-render-blocking CSS. When exam-

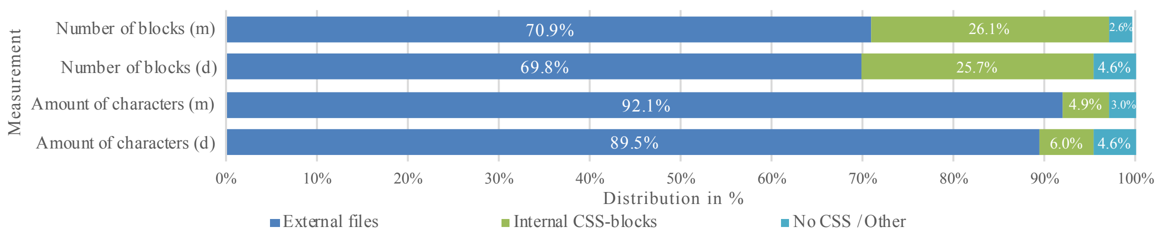


Figure 4.4: The average percentage of CSS locations in the number of files and blocks and characters on desktop (d) and mobile (m). Source: [VS22a]

ining only the render-blocking CSS, no significant difference can be found, with 15.87% on desktop and 15.01% on mobile.

As front-end frameworks like React or Angular also manage the CSS of the developed components, their CSS code efficiency is also evaluated. Shown in Figure 4.3, the average CSS code usage is lower than the average of all measured pages. However, this also includes code from jQuery, which itself does not manage CSS directly. This might be due to it being used on pages that do not optimize the page in general, as the majority of the functionality provided by jQuery can also be re-created by modern JavaScript [3].

4) CSS locations:
The location of CSS blocks, shown in Figure 4.4, is similar compared to JavaScript shown in Figure 4.2. This also implies that a smaller number of external CSS files ($\approx$70%) is responsible for $\approx$90% of all CSS measured by the number of characters. An average page on desktop links 8.50 external CSS files, and an average mobile page links 8.74. The increase in average external files on mobile might be due to optimization techniques similar to `Critical`, as internal files are always render-blocking [add22]. Therefore, the ratio of render-blocking and non-render-blocking CSS has to be examined as well for testing this hypothesis.

5) Render-blocking and non-render-blocking CSS:
As also speculated in measurement 3, there is no flexible and native way to delay CSS without using JavaScript. As a result, the average desktop page uses 89.47% render-blocking CSS and 92.09% on mobile. Therefore, the additional files on mobile detailed in the previous measurement are highly unlikely to be part of an asynchronous loading optimization. On desktop, only 5.96% of pages (and 4.92% on mobile) use methods to load CSS asynchronously. They include (but are not limited to) using media-queries. The remainder of the measured pages do not link CSS externally or were unmeasurable (2.99% on mobile and 4.57% on desktop). When considering external CSS only, on average, 63.65% of all characters on desktop could be detected as render-blocking (with 65.40% on mobile). However, this might be due to large fluctuations, as the median is 86.2% on desktop and 88.62% on mobile.

As described before, three out of the four most popular front-end frameworks manage the CSS of the application as well. Their results are displayed in Figure 4.5. Compared to the overall average, no significant difference can be found in the use of render-blocking and non-render-blocking CSS. It has to be mentioned, that jQuery has similar results as well. This is also because it is likely used in combination with other frameworks. React, Angular, or Vue have all methods of linking external CSS via the framework itself. As a result, the usage of render-blocking CSS is a conscious choice of the framework's creators or the developer. Either way, due to the disconnection from the actual compiling of the website's HTML, which is created by the frameworks, developers might be unaware of this issue. Another explanation could be that the CSS was not delayed due to backward compatibility or for clients who disabled JavaScript to ensure all clients see the page correctly.

[3] youmightnotneedjquery.com

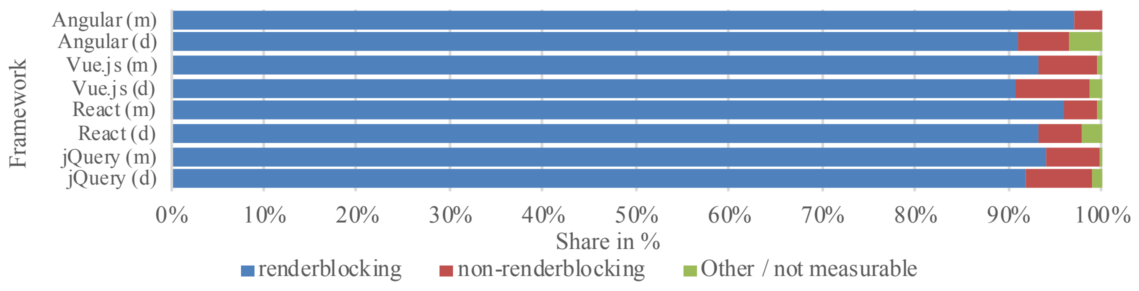


Figure 4.5: Average percentage of render-blocking and non-render-blocking CSS per most popular framework on desktop (d) and mobile (m). Source: [VS22a]

6) Matches of CSS selectors:
Measuring the efficiency of CSS alone does not tell the whole picture, as CSS can be re-used or written in a way to affect more than one element. Therefore testing the number of matches for every CSS rule is important. However, it has to be mentioned that, on average, 24.64% of the CSS selectors on desktop and 25.04% on mobile are pseudo-selectors. These include rules like `:focus` or `:hover`, which are only active if a user interacts with the element. In contrast, there are also elements like `:root`, which are commonly used to specify global CSS variables that work without user input. The following statements all refer to elements that do not require user input.

Averaging out all HTML documents, the CSS selectors have 3.07 matches on desktop and 2.98 matches on mobile (n_d=8417, n_m=8468). Therefore, every given CSS rule contains code that affects three HTML elements on average until `networkidle0`. However, this data contains large extremes. One of them is unused CSS, with 74.79% of rules on desktop and 77.26% on mobile having zero matches. Calculating the corresponding amount of CSS, it is responsible for 73.57% of all CSS code on desktop and 75.87% on mobile [VS22a]. Therefore, approximately three-quarters of all transmitted CSS code style selectors that require user input, media-queries for responsive design or are entirely unused. Furthermore, it was found that pages that load more external CSS have fewer average matches per selector.

7) Resource distribution and element positions:
According to the HTML specification, it is possible to link external JavaScript files in the head or body[4]. Scripts linked in the `head` of an HTML document made up an average of 60.55% on desktop and 61.58% on mobile. 31.95% of scripts are linked on the `body` of an HTML-document, and 32.73% on mobile (n_d=8417, n_m=8468). The remaining links are linked outside both elements.

In contrast, linking CSS is only *body-ok* since a newer version of the HTML specification [wha22]. Due to this change, 81.75% pages on desktop and 82.51% on mobile link external CSS in the `head` of an HTML document. As expected, on desktop, only 5.06% of links to external CSS is in the body, and only 0.41% on mobile. The significantly smaller percentage on mobile could be due to a conscious change to ensure backward compatibility made by the developers. The remaining web pages utilize locations outside the `head` and `body` to link external CSS or do not use CSS at all. Comparing the sizes of the `head` and `body` is important for a streaming-based delivery. The `head` of a document might contain crucial meta information which is used for displaying the page, like links to external CSS or the used encoding. As the data shows, the size of the `body` is responsible for 73.28% of the total document size on desktop (72.75% on mobile), while the head spans 24.91% of the document on desktop (and 15.8% on mobile). Therefore, the body contains the majority of transferred data. 1.80% of the data on desktop and 0.09% on mobile are outside both elements.

[4]`html.spec.whatwg.org`

	Desktop	Mobile
% of used JavaScript until idle	p: 39.05% r: 40.81%	p: 39.93% r: 40.61%
% of used CSS until idle	p: 10.84% r: 15.86%	p: 11.10% r: 14.81%
% of renderblocking JavaScript (iaoc)	p: 90.04% r: 91.78%	p: 98.07% r: 93.36%
% of non-renderblocking JavaScript (iaoc)	p: 1.96% r: 2.11%	p: 1.93% r: 2.22%
% of unused CSS classes	p: 79.25% r: 74.79%	p: 84.21% r: 77.26%
% of CSS from selectors with 0 matches	p: 77.54% r: 73.57%	p: 82.39% r: 75.87%

Table 4.3: Average values of the page optimization framework Google PageSpeed for desktop and mobile in comparison to the average of all analyzed pages (reference) with p = PageSpeed, r = reference average, JS = JavaScript, idle = `networkidle0`, iaoc = in amount of characters. Source: [VS22a]

Comparing the average amount of HTML, CSS, and JavaScript to each other, 13.29% of the code consists of HTML, 70.07% JavaScript, and 16.64% CSS (11.54% HTML, 75.51% JavaScript and 16.95% on mobile). The largest part of the transmitted code is, therefore, JavaScript, followed by CSS. HTML represents the smallest fraction of data.

Similarly to CSS, external JavaScript files are mainly linked at the start of a document, with a smaller fraction also being inserted at the end. This trend is the same for both desktop and mobile. Render-blocking JavaScript files that are linked at the end of the document ensure that all DOM elements are available, as the code is executed after all relevant HTML is already parsed by the browser. JavaScript that is linked and executed at the beginning or before the targeted elements are parsed needs an additional listener, which waits until the elements are available. However, this is only true if the code accesses or modifies the DOM. Therefore, code that is independent of the DOM can also be loaded at the beginning. If this is the case, then it also could be loaded asynchronously.

4.8.1 Desktop vs. Mobile

As shown in the previous results, the data set for the crawled mobile pages was larger than the number of crawled desktop pages. One possible reason is the crawler detection, as the puppeteer user agent combined with the default (desktop) window size might indicate a bot, which is correct. We argue that this difference is insignificant due to the number of crawled pages, as only averages were used to evaluate the measurements. However, some differences in the number of pages might be inevitable regardless of setup due to factors like timeouts.

Still, on average, the mobile pages performed worse than the same page on desktop in nearly every scenario. This could be because it is common practice for developers to use desktop computers to develop web pages, resulting in the desktop version of browsers being the default development environment. Even though the mobile-first approach is now the standard as it is used for indexing and ranking pages by Google [22aa], it might not be how web pages are built. Instead of showing more data for desktop pages, the data suggests that pages are still developed desktop first, removing elements for mobile later. However, the code for the now-hidden elements is still sent to the client. This would explain the overall worse results. For example, the JavaScript code is still part of the transmitted code-base, but their functions are not called as a mobile platform is detected.

4.8.2 Frameworks

The previous measurements showed that the difference between the four most popular frameworks is small. However, differences still exist. For example, the "Twitter Flight" framework showed that developing pages with significantly less render-blocking code is possible.

On average, 2.12 libraries and frameworks were detected by Wappalyzer on a given web page, with identical results on both desktop and mobile.

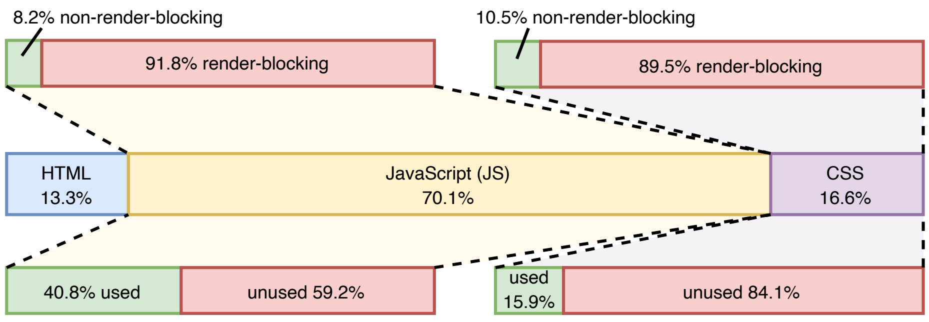


Figure 4.6: Summary of the measured proportions (on desktop) of code (HTML, JavaScript, and CSS) in the middle, with the percentages of render-blocking and non-render-blocking code above, and the code usage until render below. [VS22a]

Google PageSpeed was described in section 2.9.3 and will be evaluated separately. While it is not one of the most popular frameworks (n_d=46, n_m=49 of all crawled pages), it promises a near-universal optimization of web pages by applying best practices to a code base. When compared to the global average, pages that use PageSpeed fare worse in almost all categories, as shown in 4.3. We hypothesize that this is due to developers over-estimating the capabilities of the PageSpeed framework due to false expectations. With PageSpeed claiming to improve the page, developers might be less careful with the amount and quality of included resources, resulting in worse results. Therefore, PageSpeed does not achieve the goal of better web pages, at least when compared to the global average on a structural level.

4.9 Optimization Potential

JavaScript and CSS are responsible for 86.7% of all transmitted code. The majority of both resources are loaded in a render-blocking way (73.73% JavaScript and 89.47% CSS). However, only a fraction of both resources is used until render (40.8% of JavaScript and 15.9% of CSS). As more render-blocking means more data must be loaded and parsed before it is shown to the user, less render-blocking code will directly lead to loading improvements. It was also shown that popular frameworks do not improve loading speed, as displayed in Figure 4.3. It indicates that especially full-fledged front-end frameworks like React, Angular, or Vue do not automatically contain optimization techniques and, as a result, do not deliver faster pages by default. In general, JavaScript has to be modified in a form so that it does not block the render of the page. CSS has to be split so that only render-critical elements load synchronously. Both will significantly increase loading times due to the amount of unused code until render.

4.10 Summary and Effects on the Concept

In order to gather data necessary for developing the concept of this work, a large-scale analysis was created by crawling and evaluating the top 10.000 web pages of the web, based on the research-oriented Tranco list [Poc+18]. The evaluation was structured into seven categories which consist of render-critical aspects of the page, which influence loading times. These include, among others, the number of render-critical JavaScript and CSS, the code efficiency of both until the page is rendered, and their distribution. The individual aspects were also evaluated by testing the differences on desktop and mobile and extracting the data of the four most popular front-end frameworks, respectively. Furthermore, the PageSpeed framework was evaluated separately. The results showed major optimization potential for both JavaScript and CSS. The key points are also visualized in Figure 4.6. Both are responsible for the majority of code, with a combined 86.7% of all data transmitted. For both, the majority is transmitted in a render-blocking way, with 91.78% of JavaScript and 89.47% of CSS being loaded render-blocking on desktop. However, only 40.8% of the loaded JavaScript and 15.9% of CSS is used until the page render. This presents major optimization potential since delivering unused JavaScript and CSS will slow down the time until FCP. Therefore, when creating the concept, methods have to be developed in order to address both the CSS and JavaScript inefficiency and render-blocking properties by presenting better solutions.

5 Concept

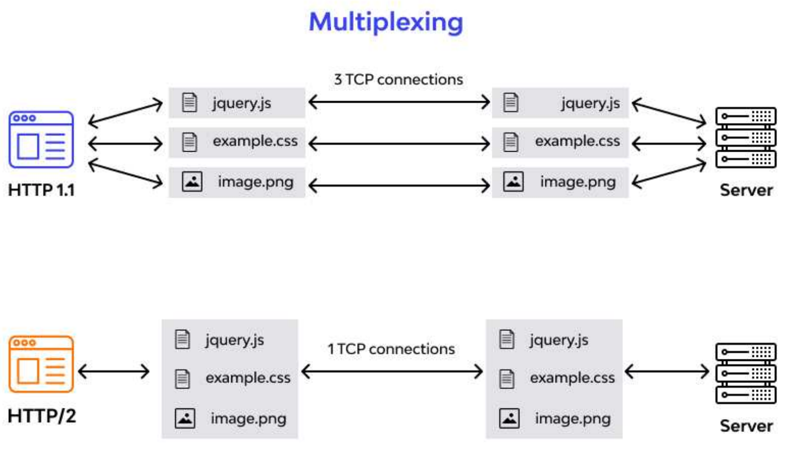


Figure 5.1: Multiplexing comparison of HTTP/1.1 and HTTP/2. Image source: [22t]

In order to develop a concept for streaming web page content, the recent trends and changes of the web itself, as well as demands for backward compatibility, have to be considered. Mainly, the underlying HTTP protocol is changing how web pages are delivered. For every change of the protocol, different optimizations are used in order to counteract limitations. This will be detailed in the following sections. However, the base concept of splitting, which first emerged in HTTP/2, will be taken as a starting point to create the concept goal described next.

5.1 Concept Idea

As analyzed in section 4.9, there is a significant amount of optimization potential left. More specifically, most JavaScript and CSS are loaded in a render-blocking way, with both being used less than 50% until render. CSS, in this case, is only used by $\approx 16\%$. This implies that there are significant pieces of code that can be delayed until after render, as they are (at least initially) unused. By doing so, the time until FCP could be improved, as less data would be required to display a given web page. However, the most important question remains: Why are websites still inefficient to this extent? The answer lies in the way web pages are built. The current popular approach is to consolidate code in the form of large external files, libraries, or bundles [22aq; 22aj]. As Google Chrome software engineer Houssein Djirdeh stated on web.dev: *"npm makes adding code to your project a breeze. But are you really using all those extra bytes? [...] Registries like npm have transformed the JavaScript world for the better by allowing anyone to easily download and use over half a million public packages. But we often include libraries we're not fully utilizing."* [22aj]. In essence, it is easy for developers to add and combine code but hard to detect if code is unused [22aj]. Furthermore, popular frameworks like React, Angular, or Vue.js all use a bundler called webpack [Rea21; 22g; 22av; web21]. With webpack, even more resources can be combined, allowing for fewer requests made to the server. Analyzing the size impact of bundled resources and installed packages with webpack requires additional frameworks, as described by Djirdeh [22aj]. These additional steps present another barrier for developers to create efficient code used until render. The necessary, modern solution to this problem is called splitting, a reaction to changes in the underlying HTTP protocol [22ar]. To further explain splitting and the ever-increasing issues with bundling, the major changes of HTTP versions need to be discussed first.
Differences between HTTP/1 and HTTP/2: As shown in Figure 5.1, multiplexing

is one of the most significant improvements of HTTP/2. With this change, a single TCP connection can be shared to load different resource types. With HTTP/1, a new connection was necessary for each resource, which slowed down the loading times of a given web page. Therefore, one possible countermeasure is to *bundle* resources together to decrease the number of connections. This is also what webpack does [web21]. However, as described by Erwin Hofman, bundling resources for web pages delivered via HTTP/2 is an *"anti-pattern."* He stated: *"In HTTP/2, this behavior will end up impacting the download-time of other resources as well, because of the way HTTP/2 works. So [...] you should be stepping away from obsessively bundling your resources."* [22ar]. However, as stated before, popular frameworks like React, Angular or Vue.js currently use bundler like webpack in order to deliver the generated code. This is a significant factor for loading unused and, therefore, inefficient code, as described by Vijay Dharap, who showed that the webpack bundle is often larger than necessary: *"[...] one must pay attention to what kind of libraries are getting bundled via webpack. This can have a large impact on application bundle sizes and thereby on your first page load and parse time"* [Dha18]. Due to the widespread usage of webpack resulting from the popularity of the before-mentioned frameworks like React, bundling remains being a relevant topic of modern front-end development. In contrast, Erwin Hofman suggested doing the opposite: splitting resources instead of bundling them [22ar]. This has multiple advantages. The most crucial being new ways of delaying parts of code, which are only necessary after rendering the page itself. When all resources are bundled into one single file, then these options don't exist: either load everything as render-blocking code or delay everything.

Comparing bundling and splitting: To summarize, the concept of bundling and splitting is now compared directly. As shown in Figure 5.1, resource consolidation approaches were used to mitigate the negative performance aspect of needing new TCP connections for every request with HTTP/1. This technique is called bundling and reduces the number of handshakes by decreasing the number of linked external files. However, all sections of code which are part of one bundle cannot be separated before all data is transferred due to the nature of the bundling process itself.
In contrast, the opposite approach is true for splitting. Here, one resource file is divided into multiple separate elements called chunks. Splitting can be done in newer HTTP versions as an optimization step, as individual resources can be transferred based on their priority. Furthermore, since HTTP/2, multiple files can be transferred over one TCP connection via multiplexing, eliminating the overhead which is present in HTTP/1. Delaying code that is not render-critical will therefore improve loading times. However, this is the opposite of what bundling intends to do. As a result, bundling has been a deprecated concept since HTTP/2.

Resulting structure considering the future of the HTTP protocol: Even with new splitting approaches, the underlying stream-based foundation described beforehand is not fully utilized. When considering the current state of HTTP/2 and future improvements like 0-RTT of HTTP/3, choices on how front-ends are built need to be re-evaluated. Code-splitting techniques can help decrease the issues caused by loading large resources. However, the next generation of web front-ends has to go one step further. The solution is provided by HTTP itself: sending all render-blocking code of a given web page via streams. More precisely, streaming the split chunks as packages. Splitting the code into a large number of files would also result in a large number of requests, which can be eliminated using one stream, sending data without requiring requests from the client at the correct time. This fine-grained splitting approach has the potential to fully utilize the streaming features supported by the new versions of HTTP. If necessary, this approach provides advantages over the all-or-nothing approach of bundles, and even the splitting technique, by viewing the transfer of data and the page rendering as a process over time. With streaming, page elements and resources can be added continuously, not only when the page is initially displayed to the user.

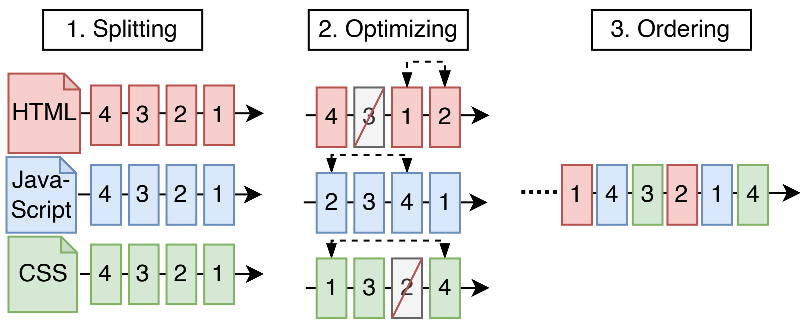


Figure 5.2: Stages of streaming web pages, starting with splitting individual files into chunks, optimizing and filtering them (for example, including removing comments), and ordering them based on the given priority.

Ideally, the individual elements would start with a code usage of 100% at the beginning, then decrease until all code is transferred. For this, three steps are necessary:

1. **Splitting** the individual resources,
2. **Optimizing** each chunk by determining their priority, and
3. **Ordering** all chunks based on the priority given, which is also the order they will be delivered and rendered later.

This order of elements can then be transferred. However, re-combining them back to files and sending them to the client would defeat the improvements made by splitting code. Instead, another solution is proposed: streaming all resources to the client. By doing so, it would also allow for splitting the HTML of the page as well. Streaming in this context describes the continuous addition of split resource chunks in an order determined by the server. To achieve the splitting step 1, techniques need to be found for CSS, JavaScript, and HTML, as they are the only render-blocking elements of web pages [20]. This is also visualized in Figure 5.2, step 1. Other media, for example, images, are therefore not directly modified in this concept. However, by the nature of HTTP/2 and the next generation HTTP/3, they are implicitly streamed as well, just with a larger chunk size, determining the image in its form as the smallest package. Next, CSS and JavaScript resources need to be optimized, as shown in chapter 4 because they are used only partially on average. Optimization also includes removing code parts, such as comments, as shown in Figure 5.2. This is the goal of step 2. The required steps for CSS, JavaScript, and HTML are individually described in section 5.3, section 5.4, and section 5.5, respectively. Next, the correct order of all elements is needed based on their priority. This is the goal of step 3. As CSS and JavaScript can be in-lined (and also delayed), this is mainly done as part of the HTML splitting described in section 5.5.1. Lastly, as described before, the chunks are transferred via streams to optimally decrease the first render time.

5.2 Summary and Next Steps

As shown in Figure 5.3, four relevant areas need to be addressed in order to be able to stream web pages. These are HTML, CSS, JavaScript, and the streaming itself, visualized as the four sections in the second shell in Figure 5.3. The outer ring displays the techniques which allow these four sections to be implemented sufficiently for streaming. However, as marked with grey sections, implementation gaps exist when considering the related work in chapter 3. These grey gaps will be discussed in more detail in their respective sections

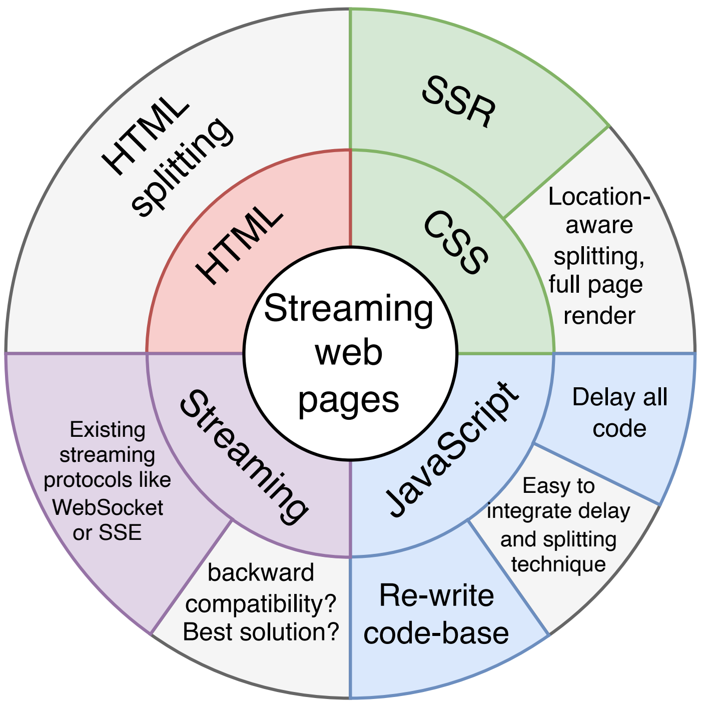

Streaming web pages
HTML
CSS
Streaming
JavaScript
HTML splitting
SSR
Location-aware splitting, full page render
Delay all code
Easy to integrate delay and splitting technique
Re-write code-base
backward compatibility? Best solution?
Existing streaming protocols like WebSocket or SSE

Figure 5.3: The layers of required software tools to stream web pages. The missing concepts are marked with grey, showing the required techniques necessary for a complete solution

(section 5.3 for CSS, section 5.4 for JavaScript, and section 5.5 for HTML and Streaming). In general, as the complete approach relies on render-critical resources like JavaScript and CSS, their concepts are described first section 5.3 and section 5.4. Both describe solutions and improvements which are required in order to create a developer-oriented and full-featured solution. Afterward, the streaming concept will be described in section 5.5, incorporating the previous JavaScript and CSS solutions. This also includes the technique necessary for splitting HTML. With this setup, all gaps in Figure 5.3 will be addressed, which allows for streaming web pages.

5.3 Concept for CSS Streaming and Rendering

As shown in chapter 4, CSS is the most inefficient render-blocking resource. Furthermore, as the streaming-based approach requires some form of code-splitting, both aspects will be addressed in this section.

Summary of Related CSS Optimization Techniques

The related work of the CSS concept can be found in section 3.2. All current methods have drawbacks, as demonstrated by the techniques and papers described. This primarily refers to articles that only examined a small number of web pages or lacked details on the used parameters, such as the version of the list of tested pages. Furthermore, currently used methods show significant remaining potential. Additionally, existing approaches do not focus on a streaming-compatible system. Supporting quick user interaction after FCP requires expanding the render by processing the entire page, as opposed to the current "Above-the-Fold" method. Furthermore, a location-aware renderer is necessary, which needs to support a more fine-grained splitting of CSS.

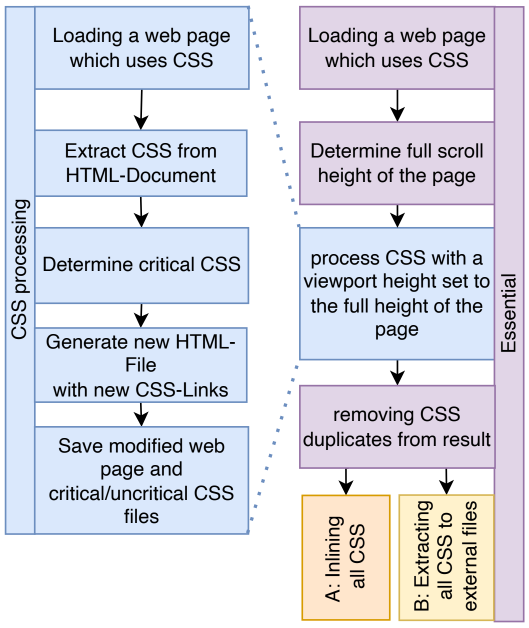

CSS processing
Loading a web page which uses CSS
Extract CSS from HTML-Document
Determine critical CSS
Generate new HTML-File with new CSS-Links
Save modified web page and critical/uncritical CSS files
Loading a web page which uses CSS
Determine full scroll height of the page
process CSS with a viewport height set to the full height of the page
removing CSS duplicates from result
A: Inlining all CSS
B: Extracting all CSS to external files
Essential

Figure 5.4: Workflow of `Essential`(right), including a detailed view of the CSS processing (left). Source: [VS23b]

5.3.1 Concept

As shown in 3.2, no existing technique fulfills all necessities presented by a streaming-based approach. More precisely, no full-page CSS renderer or location-aware CSS processing technique exists. Both are required if the source code should be able to split for a stream-based approach.

The solution will be presented in this section, which is called `Essential`. The goal of `Essential` is to extract the minimal amount of render-blocking CSS needed to display the web page correctly and group it at the correct position of the HTML document so that the necessary CSS is loaded directly before their matching elements are shown. Furthermore, as described in 3.2, duplicated code will be removed, which should increase code efficiency significantly. Placing the CSS in the correct positions and simultaneously optimizing it will improve the time until FCP. This optimization is shown and described in more detail at Figure 5.5. Furthermore, based on the related work, it is unclear if large amounts of in-line CSS will decrease the render performance due to browser-specific implementations. This is true for the streaming approach, as the CSS is not loaded as external files. Therefore, the concept will include two versions, both shown in Figure 5.4. The orange path shows the first version, ”A,” which in-lines all CSS. Version ”B” consists of the opposite and places all CSS into external files. Both versions will be evaluated, focusing on their performance. Only version ”A” can be efficiently used for the stream-based concept. However, if the in-lined CSS shows to be significantly slower than external files, the main concept has to be adapted.

Server-Side CSS Processing

In order to specify the full concept, it is necessary to discuss how CSS processing generally works. In order to extract the render-”critical” CSS, a given web page is opened with a remote-controlled browser. Then, all CSS is extracted. With the aid of JavaScript, all CSS selectors of the extracted code are then matched on the page. If a match is found, the tested CSS can be marked as ”used for render” or, in other words, ”critical .”Otherwise, the tested code is unnecessary for displaying the page (”uncritical).

```html
<!doctype html>
<html>
    <head>
    <style>
        p {
            color: blue;
        }
        h1 {
            color: red;
        }
        p { /* duplicated CSS */
            color: blue;
        }
    </style>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

```html
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <style>
            h1 {
                color: red;
            }
        </style>
        <h1>Headline</h1>
        <style>
            p {
                color: blue;
            }
        </style>
        <p>Text</p>
    </body>
</html>
```

Figure 5.5: Two versions of the same web page, showing the traditional use of CSS on the left and the CSS placed by a location-aware optimizer on the right. The pages both result in an identical web page visually. However, the CSS on the page on the right can be loaded in chunks via streams more efficiently. The reason is that not all CSS needs to be loaded before showing the first section of HTML. Furthermore, all duplicated CSS is also removed.

`Essential` also uses this method to extract critical and uncritical CSS. This is shown in Figure 5.4.

Next, a copy of the original file is used and processed. All in-line CSS and links to external CSS files are deleted in this step. Furthermore, CSS code duplicates are removed, as described in 3.2. This copy of the original HTML file will be used in the next steps.

Preparing CSS for Streaming

However, until now, the CSS is not ready for streaming, as this only extracted the full amount of render-critical and delay-able code. To fix this, the following steps are taken:

1. First, for every match of the selector, the targeted element is modified. A uniquely-named attribute is added consisting of a list of numbers, which reflect a generated ID of the parsed CSS rule.

2. After all CSS is matched, the CSS is iterated over again. This time, the code location of every critical CSS rule is marked at the first occurrence of the matching ID in the document, inside a `style`-tag. To illustrate this behavior, this location-aware placing of CSS is shown in Figure 5.5. There, it can be seen that the CSS can be placed directly before it is first used. For example, the `p`-tag is only styled after the headline.

3. For additionally preparing the CSS for streaming, the splitting, in general, will remove comments and CSS duplicates, as shown in Figure 5.6. There, it is also visible how the "critical" and "uncritical" files are ordered in such a way that the most important CSS will be sent first. For the shown example, the order is, therefore, purposefully placed to allow the switch of CSS code to be made.

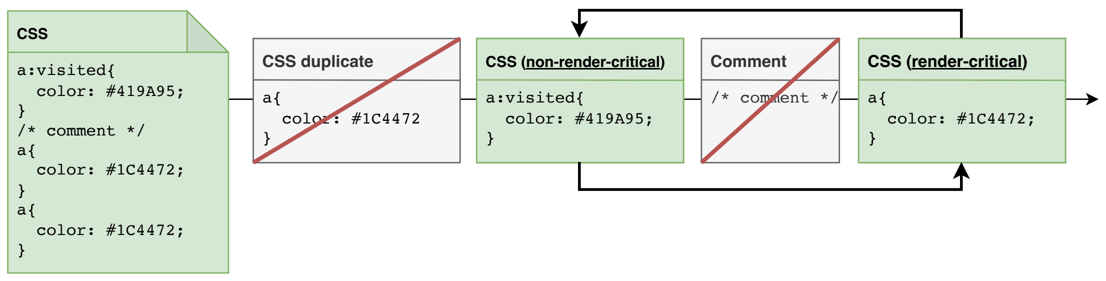


Figure 5.6: Simplified CSS splitting approach via `Essential`, showing how the original CSS is processed, ordered, and split. The render-critical CSS is prioritized, with the non-render-critical CSS being delayed. Duplicate CSS is also removed. The location-aware splitting feature is shown in more detail in Figure 5.5, as this example is only implying the targeted HTML of the link element (”a”).

5.3.2 Preparation Results of Essential

The main result is a list of CSS chunks or files, including information about their respective priorities and targeted HTML locations. Furthermore, an HTML document is generated, consisting of all required styling information in the form of in-lined CSS. There is also an ”uncritical” CSS file, which can be streamed later or loaded in a delayed way via JavaScript. This describes version ”A” in Figure 5.4. For some of the tests in the following evaluation, a ”delayed” link to the ”uncritical” CSS is inserted into the document in order to compare the results to `Critical`. However, this step is not necessary for streaming.

For version ”B,” the original HTML is filtered by removing all in-line CSS and links to CSS. Next, a render-blocking link to the ”critical” CSS is inserted, followed by a ”delayed” link to the ”uncritical” CSS.

5.3.3 Algorithm Summary

To summarize the complete CSS preparation, the steps of the algorithm are described again in detailed steps. These steps are also visualized as a full example in section 8.1.

1. First, the original page is opened via a remote-controlled browser to get the full height.
2. Then, the page is processed into critical and uncritical categories:
   a) All CSS that is loaded (externally or internally) on the page is extracted.
   b) Next, all selectors of the extracted CSS are gathered, and the CSS classes are parsed.
   c) All selectors are then queried in the HTML document. If a match is found inside the full height of the document (gathered in step 1), the CSS is marked as ”critical.” Otherwise, it is marked ”uncritical.”
   d) Implicitly, both categories do not contain any CSS comments as they are missing selectors. The parsing step removes code comments inside classes, as they are simply ignored 2b.
3. After categorizing into ”critical” and ”uncritical,” all CSS duplicates are removed. This includes duplicate classes with identical code in their respective category. The categorization into ”critical” and ”uncritical,” as well as duplication removal, is also an optimization method that allows for the next step to be quicker due to reduced input.

4. Next, the CSS is prepared for streaming.
   a) First, the "critical" CSS selectors are iterated over. Every critical selector is given a unique ID, and their match location in the HTML document is marked with this ID by matching them to all DOM-Elements.
   b) After all matches are marked, the first marker location of every selector-ID is used as the primary HTML-DOM target for the CSS class. This means that the respective critical CSS class has to be streamed directly before the HTML of the target is loaded in order to ensure that the page is displayed correctly.
5. At this point, the CSS processing is finished. However, the resulting CSS can be processed in various ways to be more useful. Mainly, for streaming the CSS, the critical and DOM-location-marked CSS can be placed inside the HTML file (with all original CSS removed). This step is called in-lining and is also shown in Figure 5.4 version "A ."Therefore, when splitting the document as pure HTML later, the CSS in-lined via `<style>`-tags can also be processed. Alternatively, the CSS can be placed in "critical" and "uncritical" files, as described in Figure 5.4 version "B."

5.3.4 Implementation

`Essential` is implemented by using `Critical` as a foundation. For this, the CSS processing section shown in Figure 5.4 is replaced by `Critical`. As `Critical` only considers CSS "Above-the-Fold" by default, the measured height of the page is used as input to process the full page. The full scrollable height of the original page is measured by opening the page via puppeteer. One additional advantage of using `Critical` is the built-in CSS generator. If `Critical` can process a page without errors, the resulting "critical" and "uncritical" CSS files will always consist of syntactically correct and error-free CSS code. Even `Critical` cannot process every page due to CSS code errors. But this behavior ensures that if `Critical` can render a page, `Essential` will also work. This is taken advantage of in the next step. In order to find the locations of every match, the "critical" CSS is parsed by the `css` package [22j]. The before-mentioned copy of the page is then loaded via puppeteer. Next, the resulting selectors are then query-matched via JavaScript onto the page. It is also important that the CSS rules are kept in the correct order due to CSS being "cascading." Furthermore, elements like `@media` have to be considered as well in order for the page to keep the responsive properties.

5.3.5 Evaluation

As described in subsection 5.3.1, there are three software versions that render CSS:
`Essential` with all in-lined CSS, `Essential` with all external CSS, and `Critical`. The main factors that will be evaluated are visual similarity to the original, code efficiency, and loading speed. All three have a direct or indirect impact on loading performance.

Test Setup

The following tests use the 1000 most popular web pages according to the Tranco-list (ID: 5Y67N)[Poc+18]. Downloading the pages was done using an automated script, which remote-controlled Chrome version 103 to download the pages, including all external resources, like images or CSS. The result is a set of 869 pages that could be downloaded.
The remaining 131 web pages could not be loaded or were unavailable in Germany, where the test location was.

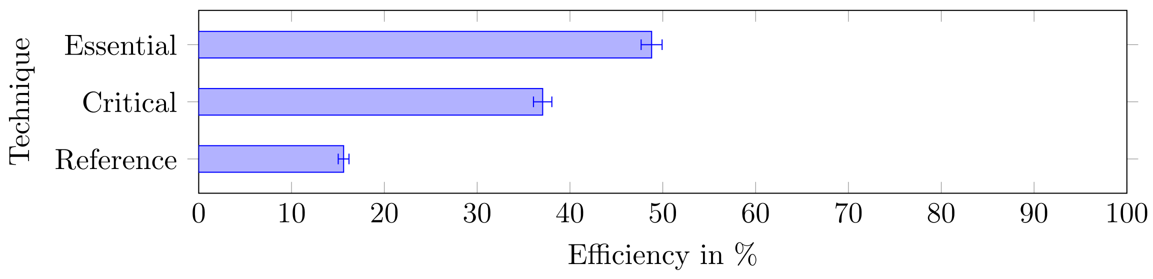


Figure 5.7: Measured average CSS code efficiency (CSS used until render). The graph includes standard error bars. Source: [VS23b]

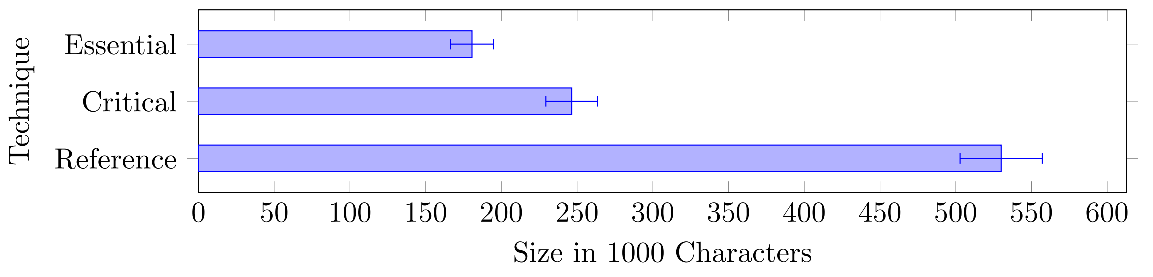


Figure 5.8: Measured average code size of all transferred (render-critical) CSS in characters. The graph includes standard error bars. Source: [VS23b]

Code Efficiency

The efficiency of CSS code is measured by measuring the number of CSS characters used until page render and dividing it by the total number of loaded (render-blocking) CSS. This was done by using the code-coverage feature of puppeteer.

Figure 5.7 shows, that `Critical` and the reference page is outperformed by `Essential` considering the code efficiency. The reference page utilizes 15.5% of the render-blocking CSS, `Critical` uses 37.1%. However, `Essential` further improves code usage to 48.8%. One explanation is the removal of code duplicates shown in Figure 5.5, which decreases the total code size and, in turn, increases efficiency.

Code Size Change

Assuming that the CSS file was ASCII-encoded, and one Byte equals one character. The original CSS would, on average, have a size of 529.9 KB. Using `Critical` decreases the size to 246.4KB, and `Essential` further reduces it to 180.1 KB. In general, this represents a decrease of 65.9%, comparing `Essential` with the CSS of the original pages.

Visual Similarity

The visual similarity of the web pages is arguably the most crucial aspect of this evaluation. To test the similarity, a pixel-by-pixel approach was chosen. For this, screenshots were created of all page variations, processed with the different frameworks. Then, the size of the screenshots is equalized, as this is necessary for the next step. The remaining space of smaller screenshots is filled with a contrasting color. Next, the popular pixelmatch-library[1] was used, which calculates a factor of similarity between two given images. The test was done with a 0.0 threshold, highlighting even minor changes in the provided images.

[1] npmjs.com/package/pixelmatch

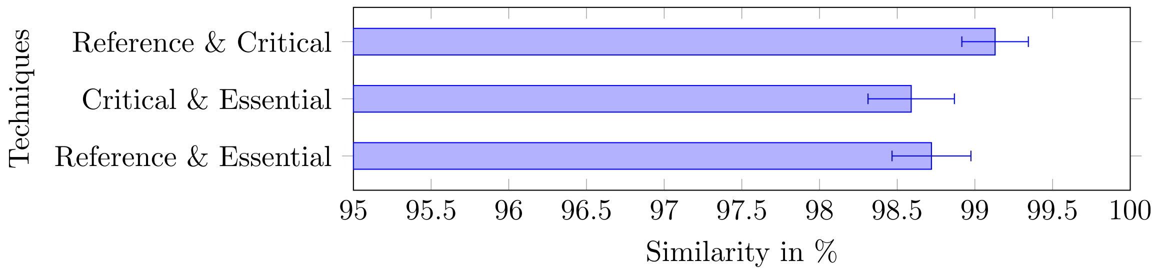


Figure 5.9: Visual similarity between fully processed versions, based on a pixel-by-pixel comparison. Note: The x-axis starts at 95% to highlight differences. The graph includes standard error bars. Source: [VS23b]

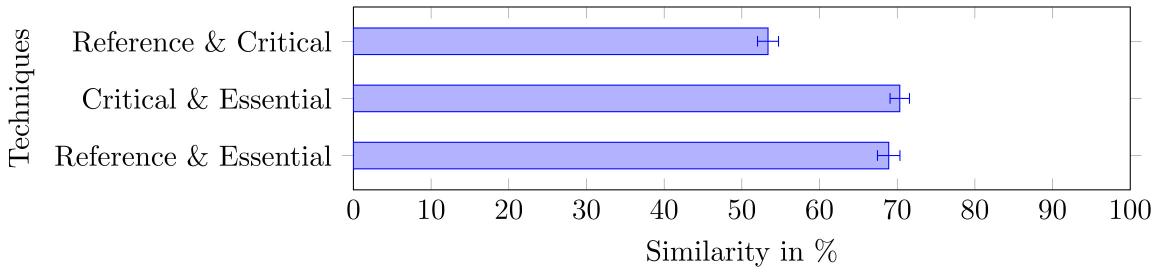


Figure 5.10: Visual similarity between versions, based on a pixel-by-pixel comparison. Both `Critical` and `Essential` were loaded without the "uncritical" CSS. The graph includes standard error bars. Source: [VS23b]

Figure 5.9 shows the visual similarity of the three versions. All processed pages generally had a high similarity score of more than 98.5%. However, this test included the "uncritical" CSS, as it is present in a real-world scenario.

When removing the "uncritical" CSS before processing, differences can be seen more concisely, as shown in Figure 5.10. Most notably, the similarity of `Essential` compared to the original is significantly higher than the result of `Critical`. This is to be expected, as `Critical` only considers the part "Above-the-Fold." However, it is unlikely that a user would notice these differences, especially when using `Essential`. This can be seen in Figure 5.11, where both versions are compared. For this image, the shown page was chosen as it highlights the difference in the modified area.

Conversion Times

Conversion times matter when using the software in a production environment. Mainly, it determines if the software can be used in real-time or if it can be used as part of a CDN deployment or build process. In order to test the speed, the conversion times of both `Critical`- and `Essential` are measured. The results are shown in Figure 5.12. 650 of the 869 downloadable pages could successfully be converged by `Critical`. One main limiting factor of the remaining pages was the inclusion of semantically incorrect CSS, which is normally ignored by the browser. The graph shows that `Essential` took 7.62 seconds on average to convert the pages. The test was conducted on a 2020 MacBook Pro with a 2.3 GHz Intel i7. It has to be higher than `Critical`, as `Critical` is part of `Essential`. On average, `Critical` took 4.397 seconds to process a page. Furthermore, Figure 5.12 also compares with an unmodified version of `Critical`. Therefore, real-time usage might not be possible. Usage as part of a built- or delivery- chain would be favorable.

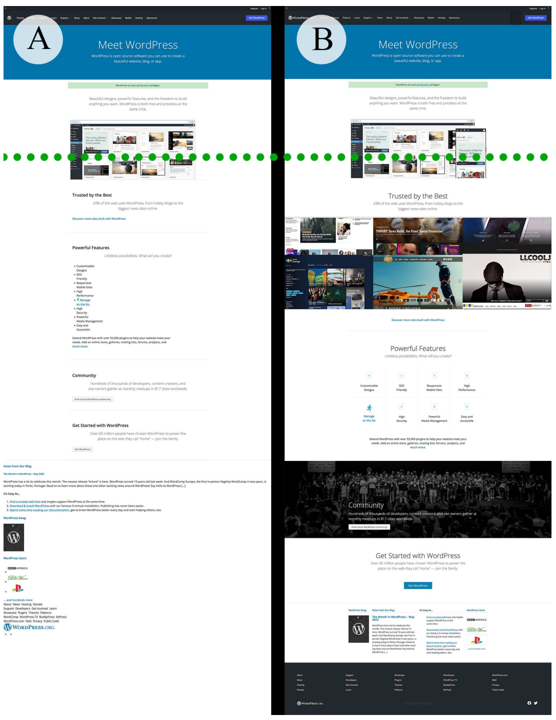


Figure 5.11: Screenshots of wordpress.org modified using `Critical` (A) as well as `Essential` (B), without including the CSS marked "uncritical." The dotted line symbolizes the "Above-The-Fold" mark when considering a 1920x1080 monitor. Source: [VS23b]

Loading Times

The times until page load was measured by comparing the time until FCP and the DomContentLoaded event. Like previous tests, puppeteer was used with Chromium 105.0 and Firefox Nightly 105.0a1. The network speed was limited by the "Network Link Conditioner"[2]-tool provided by Apple. The DSL preset with a download speed of 10 Mbps was used. Different configurations were chosen in order to test their individual impact:

- reference: the original web page
- an unmodified `Critical` version
- a modified `Critical` version used as part of `Essential`
- `Essential` with only internal (render-blocking) CSS (see Figure 5.4 version "A")
- `Essential` with only external CSS (see Figure 5.4 version "B")

Figure 5.13 shows that any form of CSS processing will significantly decrease the loading time compared to the original page. Most notably, loading all CSS as external files negatively affects the time until FCP. Both `Critical` and `Essential` had similar loading times, even though `Essential` targeted CSS of a significantly larger portion of the page. The results show that in-lining CSS not only matches the performance of comparable methods but is also favorable when a decrease in time until FCP is targeted.

The FCP represents one of the first performance markers, while "DomContentLoad" represents one of the last. Therefore, differences can be seen more prominently, as shown in

[2]developer.apple.com/download/more/?q=Additional%20Tools

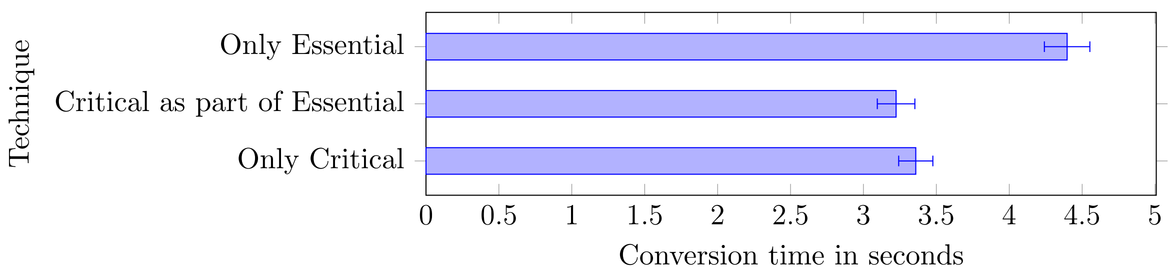


Figure 5.12: Conversion times of different versions. "Only `Critical`" describes a website's conversion with `Critical`in a non-modified state. `Essential` is separated into the `Essential`-exclusive and `Critical` parts. The graph includes standard error bars. Source: [VS23b]

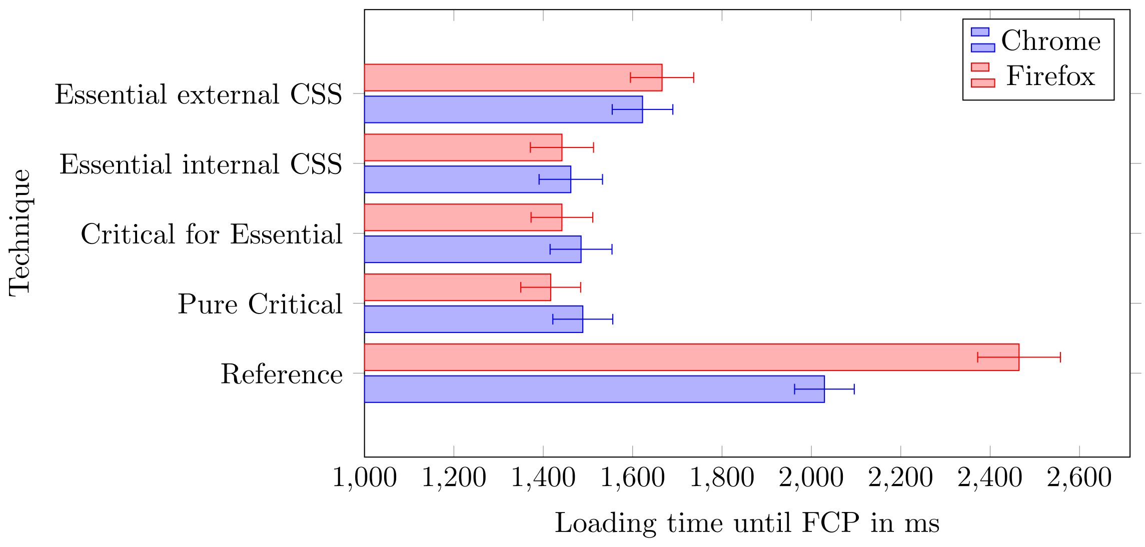


Figure 5.13: Average time until different versions take until the FCP at 10 Mbps. The x-axis begins at 1000 ms to highlight differences. The graph includes standard error bars. Source: [VS23b]

Figure 5.14. Especially when viewing the results of Firefox, it shows that, again, any form of CSS processing will positively affect the loading time. In this case, `Essential` undercuts the loading time of `Critical` when used with external CSS. One reason for this is the increase in code efficiency (section 5.3.5) and the decrease in the amount of CSS (Figure 5.8). However, Chrome showed faster loading times when used with version "A" (all in-file CSS). As the files and test setup matches in both cases, it can be explained by the implementation differences of both browsers. In any case, `Essential` showed major performance increases.

5.3.6 Limitations

In this test setup, the number of pages was limited by the availability of the web page. Furthermore, only the initial page was tested, as opposed to all available pages of a given domain. This is due to a variety of limiting factors, like login-restricted areas. Furthermore, the efficiency of `Essential` is still less than 50%. One explanation could be that modern responsive design is based on `media`-queries, which are only active at a specific screen size. Therefore, it might be sufficient for this area of application.

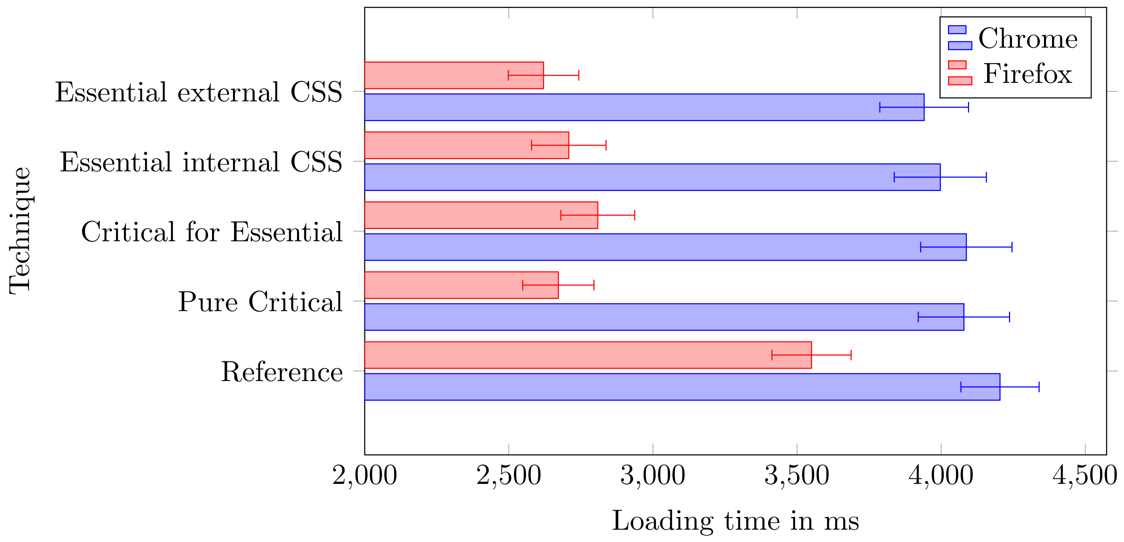


Figure 5.14: Average time until different versions load the "DomContentLoaded"-event end at 10 Mbps. The x-axis begins at 2000 ms to highlight differences. The graph includes standard error bars. Source: [VS23b]

5.3.7 Conclusion

This section presented a solution called `Essential`, which allows for processing CSS in a way that allows streaming the web page's source code. The developed software extended the functionality of the popular `Critical` -framework by processing the whole page, removing code duplicates, and inserting the resulting CSS into the correct locations inside the HTML code. The software was then tested by downloading and rendering all available web pages of the top 1000 list, according to Tranco. The results showed that pages rendered with `Essential` have a high visual similarity compared to the original, produce on average 65.9% less CSS, and match or even surpass the performance of `Critical`. Therefore, a sufficient CSS-processing solution for streaming web pages was created. These results were also published as a paper: [VS23b].

5.4 Concept for Javascript Splitting or Delaying

Next, the necessary solution for optimizing JavaScript will be discussed. As described in Figure 5.1, the importance of bundling code is steadily declining with the decreasing distribution of HTTP/1. Instead, optimizing code tries to decrease the amount of code loaded, whether the goal is a stream-based web page or not.

Summary of Related Work

As shown by the described techniques in section 3.3, three options exist to stream JavaScript: eliminate all unused code and wait for all JavaScript to be transferred, split JavaScript into smaller chunks, or delay all code. As shown with Muzeel, not even the most sophisticated approaches allow for an acceptable result [Kup+21]. Furthermore, the render-blocking properties of unmodified external JavaScript files would hinder the fast rendering of a page. In contrast, delaying all JavaScript would result in fast render times but would also increase the time until the page is interactive, depending on how the page and JavaScript code are written. It also shows that no fully automatic usage-based JavaScript optimization is currently possible. The best option is to split the JavaScript into sections and load it on demand. With a framework like Qwik, fine granularity like this can be achieved [22q]. Furthermore, the resulting Code chunks could be streamed as well. However, this requires rewriting all code, which might not be economically feasible. Instead, a flexible but easily integrate-able approach is needed, which balances both ease of use and the degree of granularity.

5.4.1 Concept

In order to create an approach that combines the positive aspects of all mentioned working approaches would represent a middle-ground between *Partytown* and *Qwik*. This missing approach would require more work done by the developer compared to *Partytown*, but less than *Qwik*, as the code-base does not have to be rewritten. Therefore, the goal is to provide a straightforward approach to delay pieces of code like *Partytown* but allow developers to vary the granularity of possible splits. The developed solution is called `Waiter`. By delaying pieces of code, the render-time will be improved, as loading code via *async* or *defer* is not render-blocking, as shown in 5.15. The schematic illustration also shows what `Waiter` allows code to do: split code by waiting for functions to be available. Therefore, code becomes independent of availability. In contrast to *Partytown*, the loss of user input before the entire code is loaded can therefore be prevented by loading input-capturing JavaScript code first. Therefore, no delayed hydration is necessary, and the state can be held implicitly via `Waiter` [22ao]. The improvement in render times, therefore, originates from the developer's decisions. However, this does not solve the ease of use. To address this issue, another framework was created: `AUTRATAC`. It is a helper framework that allows for automatic conversion code by inserting `Waiter` into all asynchronous function calls. Both frameworks are described in more detail next.

5.4.2 Waiter

The main goal of `Waiter` is to accept function calls or resource requests, wait until they are available, and finally deliver the results. This requires asynchronous code to work. There are three main objectives that `Waiter` has to achieve:

1. `Waiter` has to be usable in any form of asynchronous code
2. the core JavaScript required has to be as small as possible, and
3. `Waiter` has to be easy to use and understand by developers.

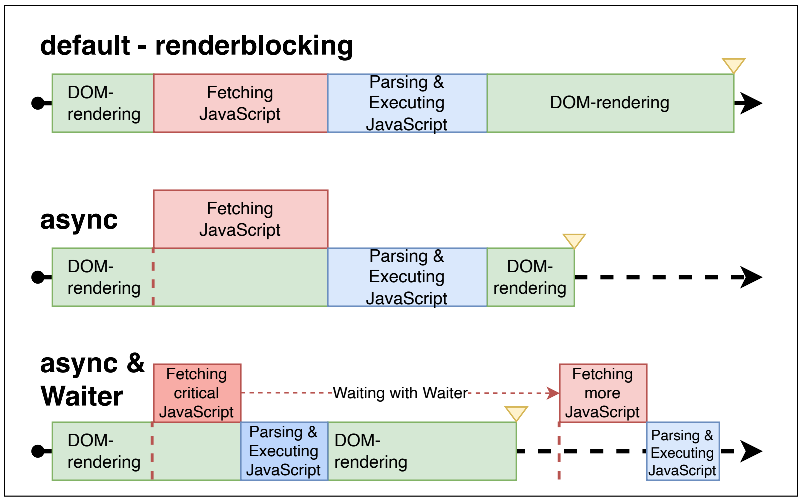


Figure 5.15: Comparison of the loading times of the default loading behavior, async and async with `Waiter`. The yellow triangle marks the first possible render time. Source: [VS23c]

The third objective is being achieved by using `AUTRATAC` to automatically insert `Waiter` into asynchronous functions in a code-base. However, it is also possible for the developer to insert calls manually. For it to be simple and easy, as required by objective 1 and 3, the call syntax has to mimic the original calls as closely as possible. Therefore, the possible syntax is discussed first. To achieve objective 2, a discussion of the options of use will follow.

Call Syntax

In order to specify the call syntax of `Waiter`, multiple challenges need to be addressed first. Mainly, calling functions that are not previously loaded results in an error being thrown. This also happens when this function call is never executed. When this occurs, all JavaScript execution stops. To prevent this, two options are available. The first method requires the use of `eval()`, as it allows the evaluation and execution of JavaScript code from a string. Even though it works, it also has significant drawbacks: `eval()` is considered to be unsafe [22o], and the written code does not highlight errors, as IDEs will interpret the call as a string. Both issues are solved by using arrow functions, which are the second option.

```
1 function a(){ //is not called
2   eval('b()'); //will not throw an error before execution
3   (
4     ()=>b() //will not throw an error before execution
5   )()
6 }
```

Listing 3: Comparison of eval and arrow functions. Function `b()` is undefined in this case. Source: [VS23c]

Both `eval()` and arrow functions are compared in Listing 3. In this example, the function named "b" does not exist, and the function called "a" is never called. If a call to function "b" were made without both techniques, it would result in an error. Instead, both versions can exist in valid code and, in this example, would only throw an error when called. Line 4 shows the use of an arrow-function, which in this case is wrapped in an IIFE to enforce immediate execution of code when called. Otherwise, the arrow function would only declare

```
1 //Waiter is loaded beforehand
2 async function a(){
3     await __w(()=>b())
4 }
```

Listing 4: Example call syntax calling a function with Waiter. Source: [VS23c]

code. In the final version of `Waiter`, this expression is not needed.

One of the drawbacks of `eval()` is also visible in Line 2, where code-highlighting is not working. Arrow functions, conversely, can even be made type-safe [22n]. As a result, the arrow functions are chosen. Listing 4 shows the final syntax of how `Waiter` is called. The parent function is marked asynchronous via the `async`-keyword. Waiter is then called with `__w`. The double underscores only serve to reduce double declarations, as it will be inserted and provided globally. The `await` statement shown in line 3 is not strictly necessary, depending on the goal of the code.

Detecting Resource Availability

Context determines the availability of resources in JavaScript. Therefore, availability has to be checked repeatedly. As `Waiter` targets the modification of the DOM by the addition of asynchronous, external JavaScript files, a *MutationObserver* provides the best solution. It can be configured so that it is triggered if the DOM changes, which also listens to changes in the code. This feature is supported by at least 98.5% of all tracked users by *caniuse.com* [22ac]. It is also possible to check in a time interval. However, the observer only triggers when changes occur, so it is preferred due to fewer checks needed. Both can also be used in combination.

Implementation of `Waiter`

The goal of objective 2 was for `Waiter` to be as small as possible. Listing 5 shows the first version. In its uncompressed state, it only consists of 23 lines of code. When compressed, the size is reduced to 244 Byte. with less than a quarter kilobyte, objective 2 is achieved. When `Waiter` is called, a Promise is returned. As both the call syntax and the framework itself both use arrow functions, the original context is preserved. Therefore, calling objects with `this` still works. The framework then listens via the observer for changes made in the code. When a change is detected, the availability of the targeted resource is checked. If it is available, the answer is returned, and the observer is disconnected. This test function is also called once in line 21 to ensure the fastest possible return of data if the targeted resource is already available when `Waiter` is called.

In summary, all three goals of `Waiter` can be solved. With a size of less than 250 Byte and with a type-safe call syntax, it has a minimal footprint and can be used in any asynchronous code. However, until this point, it requires manual insertion by the developer. To address this issue, `AUTRATAC` is described next.

5.4.3 AUTRATAC

Even if manual insertion of `Waiter` allows the developer to have a high level of precision and control, wrapping all calls in asynchronous functions in a large code-base might challenge goal 3. To solve this issue, an automatic converter is necessary, which is called `AUTRATAC`. In this case, all calls inside asynchronous functions are automatically wrapped with a call to

```
const __w = async (call) => {
    return new Promise(async (resolve) => {
        let isTesting = false;  //Prevents double availability tests
        const tf = async () => {
            if (!isTesting) {
                isTesting = true;
                try {
                    resolve(await call());
                    observer.disconnect();
                    isTesting = false;
                } catch (e) {
                    isTesting = false;
                }
            }
        }
        const observer = new MutationObserver(tf);
        observer.observe(
            document.documentElement,
            { childList: true, subtree: true }
        );
        tf();
    });
}
```

Listing 5: First uncompressed version of the Waiter framework

`Waiter`. `AUTRATAC` has, therefore, three goals as well, similar to `Waiter`:

1. `AUTRATAC` has to convert asynchronous JavaScript code correctly,
2. it has to be easily integrate-able
3. calls to `Waiter` need to be inserted.

Before those goals are discussed, two important questions must be addressed first: If `AUTRATAC` can transpile code, why is not all code converted with calls to `Waiter`? Is it possible to convert all synchronous code to asynchronous code first?

One of the main downsides of both questions is the resulting overhead. Using jQuery version 3.6.0 as an example, only adding the `await`-keyword and a wrapper for `Waiter` results in a 34.7% file size increase. This amount of overhead might counteract improvements made by splitting the code.

Next, promise chaining reduces the performance of the code. While this might not be perceivable under average circumstances, adding `Waiter`and, in turn, a promise to every function call can chain promises significantly. Shown in Figure 5.16, different results can be measured on different browsers, depending on the individual implementation. The difference between `async` and `Promises` was minor in most cases. However, the difference compared to synchronous code is indisputable. Therefore, the amount of asynchronous code should be kept at a minimal level until browser manufacturers fix the performance difference.

Finally, some challenges arise from automatically converting synchronous code into the asynchronous equivalent. These include the inability to create asynchronous constructors natively. Moreover, JavaScript allows functions to be instantiated as classes. The so-called function-based classes represent a significant challenge to determine in which form the code is used. Both forms would require different ways of conversion. Without a dynamic code analysis, errors like the one shown in Listing 6 are nearly impossible to convert efficiently.

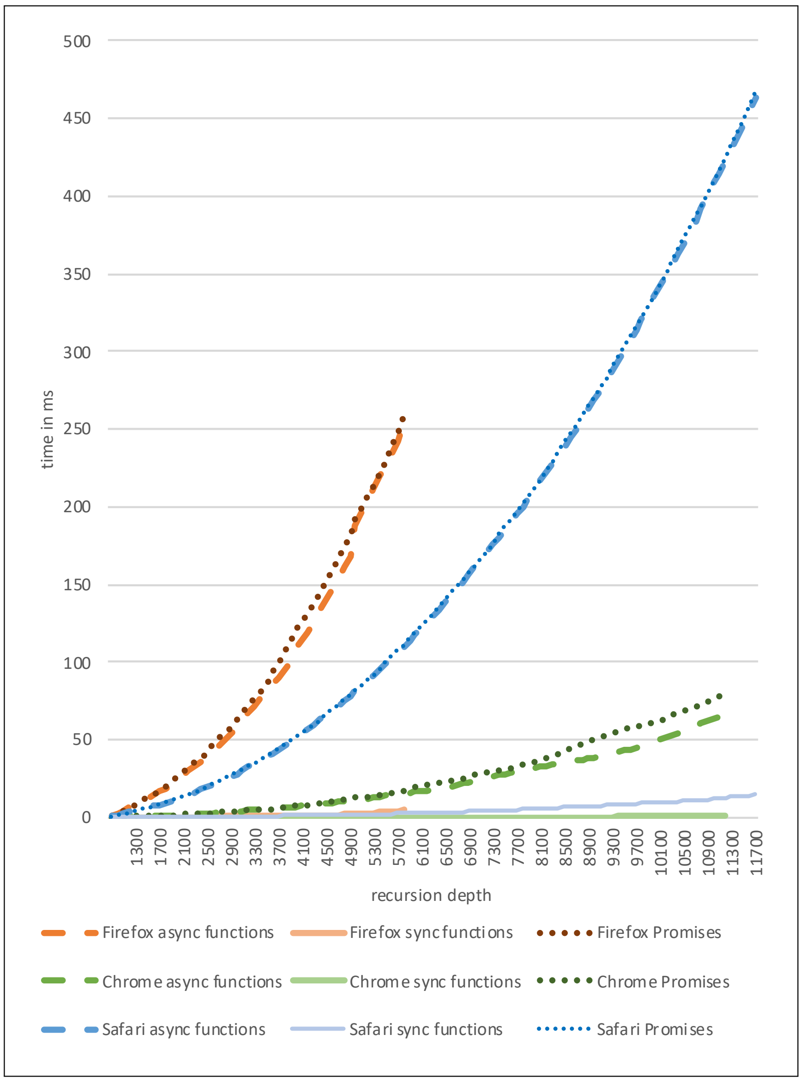


Figure 5.16: The average time calling recursive functions (synchronous, async, and with Promises) using different call depths called 100 times each on Chrome, Firefox, and Safari. The depth of the recursion is shown on the x-axis. The test was performed on a 2020 MacBook Pro with a 2.3 GHz Quad-Core Intel i7 and 32GB 3733 MHz RAM. Source: [VS23c]

```
function A(){}
const a = new A(); //works

async function B(){}
const b = new B();  //will throw TypeError "B is not a constructor"
```

Listing 6: Function-based class example with and without using an asynchronous function. Source: [VS23c]

As a result, only converting already asynchronous code represents the only currently viable option. This ensures maximum performance and minimal conversion issues.

Implementing `AUTRATAC`

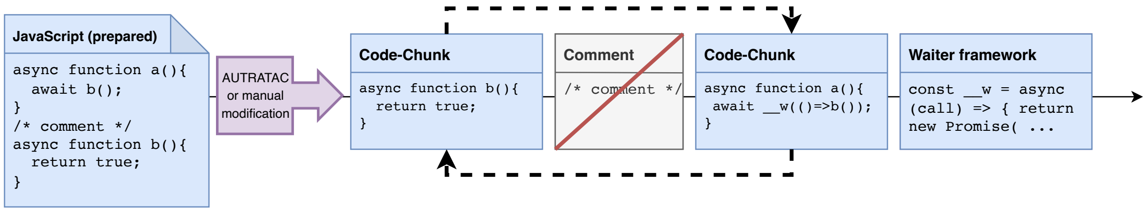
In order to allow integration with minimal effort, the *Babel* framework was chosen. Babel is a widely used and popular framework for transpiling JavaScript into another form of JavaScript [bab22]. For example, it is used to convert the "jsx"-files used by *React* into syntactically correct JavaScript [22a]. Therefore, `AUTRATAC` is created via a *Babel* plugin, as it ensures compatibility. Using Babel plugins requires minimal effort, as a large variety of integration options are available [bab22].

5.4.4 Preparing for Streaming

A full visual example of the complete JavaScript processing steps can be found in section 8.2. In order to stream the prepared JavaScript, the code splitting is decided by the developer, with the help of `Waiter` and `AUTRATAC`. Figure 5.17 shows this process in action. The original JavaScript code is prepared by the developer by making all targeted functions asynchronous. In the shown example, it only requires adding `async` and `await`-keywords. Then, `AUTRATAC` is used to convert the files. This example shows that the developer separated both functions into individual chunks. A JavaScript chunk is based on one input file or one `<script>`-tag. In this case, it can contain one or more functions, depending on the developer's decision. In the future, other splitting techniques that already incorporate `Waiter`might be available. However, no such technique currently exists and far exceeds the scope of this thesis. Still, both developed frameworks make it easy to set up this step manually. Figure 5.17 also shows the removal of comments. This does not have to be implemented, as `AUTRATAC`is based on Babel, which provides the comment-removal functionality already (by setting *comments: false* in the *.babelrc*-file or using *–no-comments* when using the CLI). Lastly, the figure shows that both code chunks are interchangeable. As described in section 3.3.1 about dead code removal, detecting which code needs to be prioritized automatically is not yet fully researched or commercially available. `Waiter` can eliminate this issue by making the loading order arbitrary. Therefore, the stream-able JavaScript chunks start with the `Waiter`-framework, followed by all `AUTRATAC`-converted code-chunks in the order that the developer decided.

5.4.5 Evaluating `Waiter` and `AUTRATAC`

Both frameworks will be tested separately, as they target different elements: `AUTRATAC` is used by the developers, while `Waiter` affects how the code works on the client. The test for `AUTRATAC` contains conversion time measurements for different code lengths. `Waiter` is tested by measuring the average time until FCP and the execution time of the JavaScript code itself.



```
async function a(){
  await b();
}
/* comment */
async function b(){
  return true;
}
```

```
async function b(){
  return true;
}
```

```
/* comment */
```

```
async function a(){
  await __w(()=>b());
}
```

```
const __w = async
(call) => { return
new Promise( ...
```

Figure 5.17: Simplified JavaScript processing step by using `AUTRATAC` and `Waiter`, by modifying and splitting the code. The dotted lines represent interchangeability due to the use of `Waiter`.

`AUTRATAC`: Conversion Speed

One of the main challenges for the conversion speed test is the lack of available data on what and how many functions are used in an average web page. More specifically, the function-to-call ratio is unknown. However, even if said data were available, it would not represent a realistic basis, as only a smaller amount of asynchronous code is expected to be translated. Instead, extremes are tested in order to extract relevant data. As a result, the conversion speed test consists of a file containing JavaScript code with an ever-increasing number of functions of varying types. Three tests were performed. The first test consisted of a number x of functions that do not contain calls. Secondly, x functions with exactly one call were converted. Lastly, a single function with x number of calls is used.

Test setup

In order to measure the three versions, a node script generates the individual code blocks for all versions with increasing numbers x. Then, the code blocks are converted via the `AUTRATAC`. The duration of this conversion was measured. For every x, the test ran 100 times. The results are an average of 100 runs. The test ranged from $x = 1$ to $x = 500$ and was performed on a 2020 MacBook Pro with a 2.3 GHz Quad-Core Intel i7 and 32GB 3733 MHz RAM.

Results

Figure 5.18 shows the results of all three measurements. For all versions, linear growth is perceivable. When the code complexity increased, the time for conversion also increased, as shown by the blue line. As the combination of functions and calls create an Abstract Syntax Tree (AST) of a larger size, a time increase is to be expected. Depending on the size and complexity of the code, it might be possible to convert the code on-the-fly, as `AUTRATAC` can translate code on a file-by-file basis, which can be done in parallel. To give a frame of reference, jQuery in version 3.6.0 consists of 607 functions and 1826 calls, which is an approximate ratio of 1:3.

`AUTRATAC` Code Correctness

As `AUTRATAC` consists of a plugin for Babel, it depends on what Babel considers as a *call* or a *function*. If Babel detects both correctly, then this part of `AUTRATAC` will also work as intended. The only additional logic `AUTRATAC` adds is a check which uses a Babel-provided function to test if a *call* is already `Waiter` -wrapped. Ensuring that this procedure will always produce syntactically correct code is therefore dependent on the correctness of Babel.

Loading Speed

Due to the unavailability of web page code-bases and the required effort, converting existing web pages is not feasible. This is mainly because `Waiter` requires manual decisions made

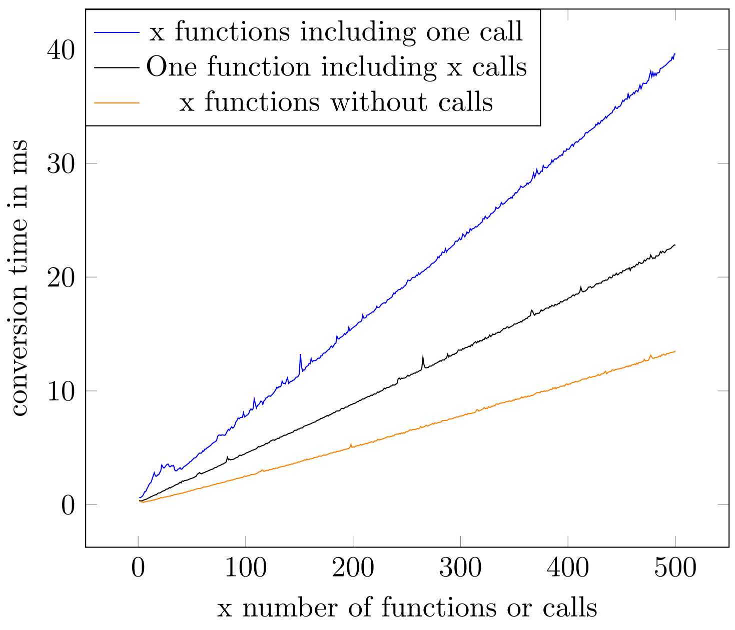


Figure 5.18: Average conversion times with `AUTRATAC` of 100 tests each for an increasing number of functions to convert

by the developer. Instead, an artificial web page was created. The web page is created as follows. From a 2022 Statista survey, it was determined that React is the most popular front-end framework (when extracting frontend-exclusive frameworks) [22ab]. Using a framework also tests if `Waiter` and `AUTRATAC` can be combined with existing code-bases and if it is easy to integrate. Therefore, the "Create React App"[3] tool was used to generate a web page with react version 18.2.0. From this, the pre-existing demo page, which is generated by the tool, was used. Next, the amount of external, render-blocking JavaScript was calculated using data from the previous analysis study [VS22a]. It was determined that, on average, $\approx$ 25 render-blocking external files are loaded per web page and that they contain 94756 JavaScript characters each. The web page was therefore modified by adding 25 render-blocking external files. Each file consists of a callable function. The rest of the file contains randomly generated characters. It was ensured that all functions are called in sequence in the main *App*-component, with the time until full execution measured. This shows if an additional delay is introduced by adding `Waiter` in this way. All modified JavaScript files are marked as "deferred" to delay the download. Furthermore, the execution was started at the *DOMContentLoaded* event, ensuring the maximum possible delay was tested.

The main test consists of two machines communicating via a local network. The host device provides both web servers serving one version each, built with the `express` node framework [22p]. The "client" is a 2020 MacBook Pro with a 2.3 GHz Quad-Core Intel i7 and 32GB 3733 MHz RAM, which uses Apple's "Network Link Conditioner"[4] to load the web pages at 2, 4, 6, 8 and 10 Mbps. This process was automated using *puppeteer* with Chromium version 105.0 [22ah]. All timestamps are gathered using Chrome's Performance-API [22ag]. Every version was loaded 100 times, and mean values were calculated.

FCP

The times until the First Contentful Paint are shown in Figure 5.19. It can be observed that

[3]create- react-app.dev

[4]developer.apple.com/download/more/?q=Additional%20Tools

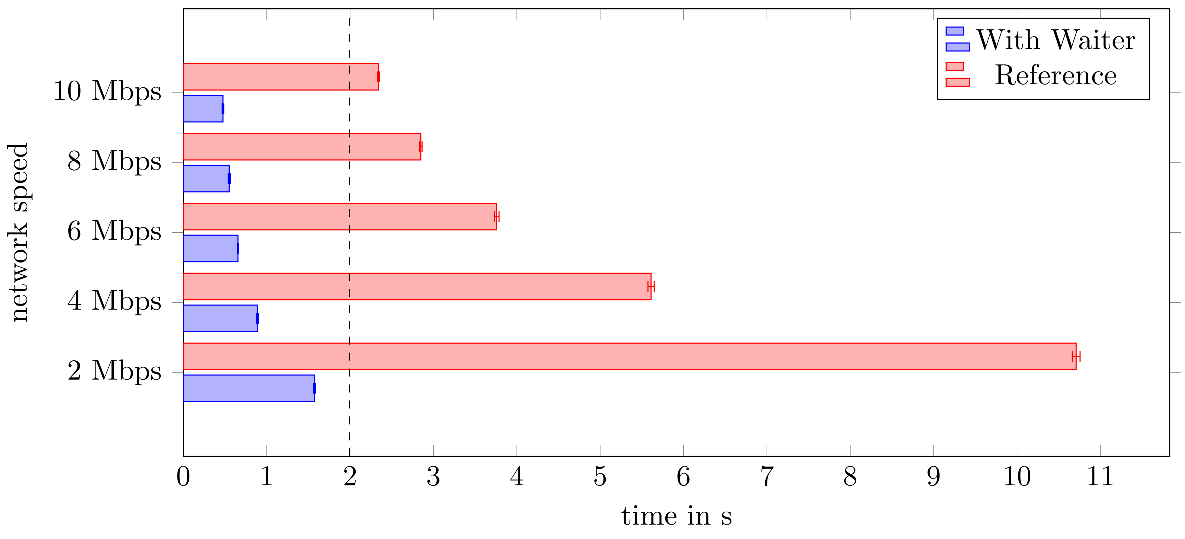


Figure 5.19: Comparison of the average loading time until the First Contentful Paint of a website optimized with and without `Waiter` when loaded at different network speeds. The graph includes standard error bars. Source: [VS23c]

using `Waiter` improves loading times in every measured case. Especially at 2 Mbps, a time difference of 9.14 seconds is visible. Using `Waiter` allows the tested web page to be loaded in less than 2 seconds for all tested network speeds, fulfilling the estimation of acceptable waiting times for users by Nah et al. [Nah04].

JavaScript Execution Time

All of the generated external JavaScript files include one callable function, which are all called successively in the main "App" component. The time for the execution of all calls is measured via timestamps. The code modified by `Waiter` further includes "await"-statements to ensure the execution order. The measured time frame begins directly before the client requests the page. The end timestamp is captured directly after the last call to the last external file is returned. Therefore, all overhead is captured as well. This ensures to capture most meaningful data, as the total overhead for including `Waiter` is critical for real-world applicability.

Figure 5.20 displays the results of the measurement. Comparing the reference with the version, which includes `Waiter`, it shows that execution time is nearly identical. The time spans differ only by a maximum of 31ms, which is in the margin of error. Adding `Waiter`does not introduce significant overhead, at least not at the tested number of calls.

Data Transferred

The total size of the code converted by `AUTRATAC` was also compared. Every function call produces a file size increase of 17 characters. The 25 external files increased, therefore, by a total of 425 characters. `Waiter` itself is 244 Bytes large, with the same number of characters (244).

Challenges and Limits

Even though `AUTRATAC` converted all code successfully in all tests made as part of this concept, there is no guarantee of correctness. As described in section 5.4.5, the dependence on Babel increases the complexity of such proof even further. With the described possibility of manual use, it is therefore outside the scope of this work.

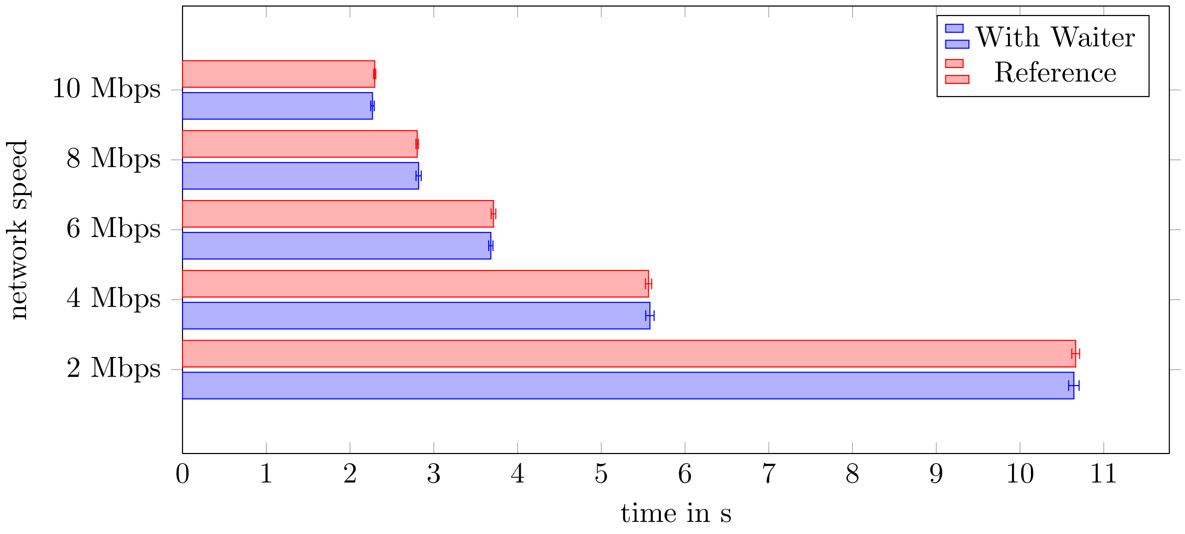


Figure 5.20: Comparison of the average loading time until all linked external JavaScript functions are called and executed, showing a web page optimized with and without `Waiter` loaded at different network speeds. The graph includes standard error bars. Source: [VS23c]

Usability for Streaming

`Waiter` provides the functionality of delaying and executing code even if it is currently unavailable. This allows for highly dynamic splitting approaches: in theory, if all code is asynchronous, functions could be loaded randomly and one by one without breaking the web page's functionality. Therefore, it is ideal for streaming, as the order is unimportant. In any case, as long as calling functions or resources outside the individual code block are made by `Waiter`, the code will work. Optimizing the location of the files, however, is a decision that has to be made by developers, as it depends on individual preference.

Conclusion

This section describes and evaluates a solution for delaying JavaScript code. With the first developed solution, called `Waiter`, code can be called even though it is not loaded yet. The framework will then handle function and resource availability. This requires asynchronous code to work, as it is based on *Promises*. Then, `Waiter` can be wrapped around function calls. In order to assist developers in this transition, a second framework was created. Called `AUTRATAC`, it automatically adds the `Waiter`-wrapper to all calls inside asynchronous functions. Ideally, a developer just has to add the "async" keyword in front of targetted functions. This reduces overhead, as described in subsection 5.4.3. The results show that adding `Waiter` increases the time until FCP significantly, in one measured case, by more than 9 seconds at 2 Mbps. Simultaneously, the total execution time stays the same. Therefore, an easy JavaScript splitting solution was presented, ready to be used for streaming. These findings were also published in paper [VS23c].

5.5 Concept for Streaming Html-Based Web Pages

The stream-based loading of web pages will be based on the concept explained in section 5.3 and section 5.4. Both show examples of how CSS and JavaScript can be effectively split, solving the gaps shown in Figure 5.3. What is left is developing a solution for splitting HTML and testing the streaming itself.

Summary of Related Work

In summary, no existing technique focuses directly on streaming the initial page, as shown in section 3.4. Existing methods for streaming, like Turbo or Marko, do not extend the concept far enough. More specifically, no technique exists that streams the full initial page or that can split HTML universally. Still, a large variety of different approaches are shown that aim to improve the loading speed. It shows that this topic is still relevant and not fully solved. Furthermore, most related techniques still focus on optimizing a file-based delivery. Therefore, a method needs to be conceptualized which can incorporate frameworks like `Essential` and `Waiter`while being able to split the produced content and especially its HTML. Furthermore, this technique needs to be able to stream all initial content while ideally being backward-compatible.

5.5.1 Concept

To create a concept which surpasses the features of the related work, it is necessary to include the location-aware CSS rendering framework ”`Essential`,” described in section 5.3, and the ”`Waiter`” and ”`AUTRATAC`” frameworks detailed in section 5.4. The following goals need to be met:

1. the approach has to be backward-compatible in order to be adapted realistically
2. the manual steps required by the developer need to be considered, as no automatic approach for JavaScript is currently feasible, as described in section 5.4
3. All processed resources must be splittable as much as realistically possible.

In order to meet all three goals, the required process is shown in Figure 5.21. Generally, two steps are visible: the *preparation* stage and the *delivery* step. First, the preparation process starts with the manual modification of JavaScript by the developer. The `Waiter` and `AUTRATAC` frameworks can and should be used, providing maximum splitting capability while simultaneously reducing modification effort. Next, the prepared code-base is transferred to the server, where automatic scripts process the page further. As described in section 5.3, CSS can be rendered and inserted automatically using `Essential`, which is also location-aware and streaming-ready. Lastly, the HTML and in-line CSS mixture will be split by the method described in the following section 5.5.1, resulting in a mixture of split JavaScript, CSS, and HTML. These chunks are then moved to a web server, ready for transfer. With this, the preparation is finished. The delivery, as shown in Figure 5.21 is later described in section 5.5.1.

Splitting HTML Into Chunks

The following steps of splitting HTML are also visualized as a full example in section 8.3. If a client-side cache is used, splitting HTML could technically be done everywhere if it is recombined before a page's HTML in the browser is set. In other words, the HTML displayed on the client could also be stored as a string. When new data arrives via stream, it can be appended to the string, and the complete HTML can be updated. This works, but better

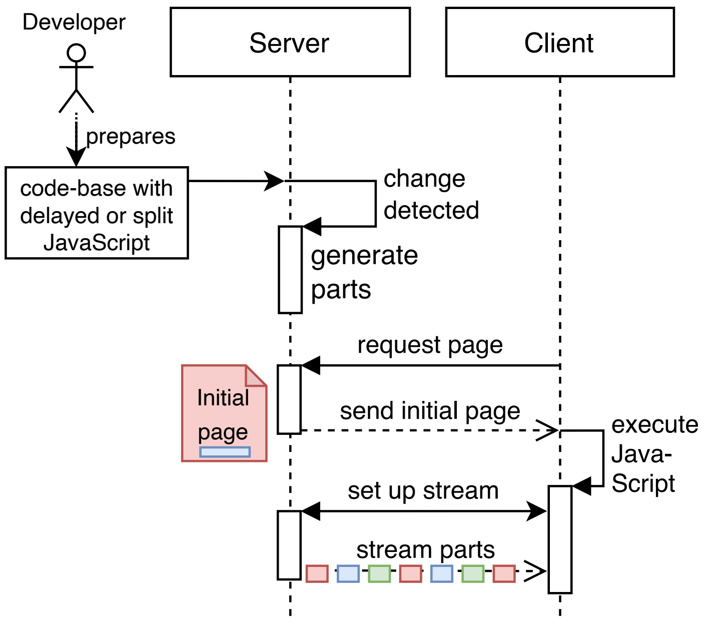


Figure 5.21: Simplified diagram of streaming web pages, highlighting the steps of a client loading a page after the chunks are generated. Modified from: [VS23a]

solutions exist, as this caching and full-updating approach has some major downsides. First, the split can be inside a tag, which will be misinterpreted by the browser, which can lead to unwanted behavior. This is especially true if custom HTML tags are registered and used. Next, updating the whole page costs performance, as it requires a complete re-render. This can lead to unresponsive pages or "flickering," as resources like images might be re-loaded as well. Lastly, this leads to the deletion of all user input, which to a large part, would be counterproductive to the performance increase made by splitting the page.

However, two issues can be solved simultaneously by splitting the HTML into start and end tags, text, important tag attributes, and non-splittable HTML chunks. First, the HTML code can be added to the page without using a cache. Secondly, no full-page updates are necessary, as broken tags are now non-existent. Lastly, text can be split as desired, as it can always be appended to the existing parent element. This splitting process is shown in Figure 5.22. There, the individual chunks with their content type are displayed. This splitting works, as browsers fail gracefully, by inserting a closing tag if it is not provided [22m; 16; 22r]. Therefore, those main options will be defined as one "package" of HTML (or in-line CSS and JavaScript), respectively. However, there are sections of "HTML"-code that cannot be split, as inserting into these chunks can lead to errors. This mainly includes the `<noscript>`-tag, which did not allow for adding content, at least in Firefox, where it was tested. Therefore, edge cases exist where a small chunk of HTML has to be transferred entirely and cannot be split further. Another edge-case are scripts. In-line scripts via `<script>` can also be added like any other element. However, then the script would not be executed. Therefore, it has to be marked differently, so it can be added by creating an `<script>`-element at the page, ensuring execution. This approach was also published in paper [VS22b].

HTML Package Types

Figure 5.22 Shows, in a simplified form, how HTML is classified into different types. However, these do not represent all available types, as described beforehand. There are other types as well, like comments, which are not necessary for streaming.

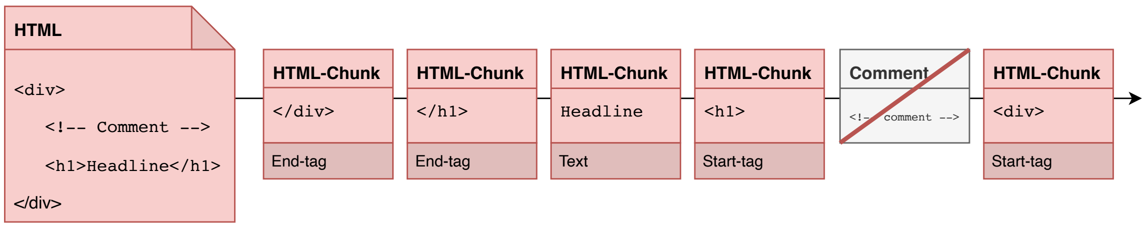


Figure 5.22: Simplified process of splitting and processing HTML. It shows how HTML is separated into multiple types of chunks: start- and end-tags, as well as text. Comments get removed. To ensure simplicity, HTML code chunk reordering is not shown in this specific example. To optimize the loading speed, it would move entire sections, like sending `body`-content before `head`-content.

In summary, the following package types are therefore required:

1. **Start-tag**: The beginning of a DOM-element
2. **End-tag**: The end of a DOM-element
3. **Text**: Non-executable content of an element
4. **Script**: JavaScript-code
5. **HTML**: Non-splittable HTML, like "noscript"
6. **Attributes**: Important attributes of the HTML or BODY element. As the minimal initial HTML file provides the base HTML structure, these attributes need to be set in order to ensure functionality and prevent layout shift (for example, if the body has a specific class)

Delivery of the Pages

After the before-mentioned preparation stage is finished, the page is ready to be sent via streams, as shown in Figure 5.21. This begins with a client requesting the page from the server. Then, an initial, minimal HTML file containing a minimal amount of JavaScript is transferred to ensure backward compatibility. This process is shown in more detail in Figure 5.23. After the client parses the initial page, the containing JavaScript establishes a streaming connection back to the server as soon as possible. The resulting connection is then used to stream actual data to the client. When focusing on the best possible content delivery performance, the order of the data which is sent to the client is the following: First, all attributes of the HTML- and BODY-tag is transferred. This is necessary to ensure the final HTML-DOM is identical to the original and can be styled correctly without layout shift. Next, the content of the BODY is sent. Lastly, the HEAD elements follow. This process is also detailed in the full example in section 8.4.

The initial connection process introduces an overhead. However, by doing so, the client-side browser implementation does not need to be modified. This is necessary, as global browser usage is highly diverse [22h]. It would be ideal to have full browser support from the start, as then the browser could directly try to load a web page via stream, eliminating said overhead. However, judging by the lack of implementation of currently available optimization techniques as described in chapter 4, such an expectation would be unrealistic. Still, if such an approach exists in the future, the concept does not have to be adapted, instead providing a future-proof solution.

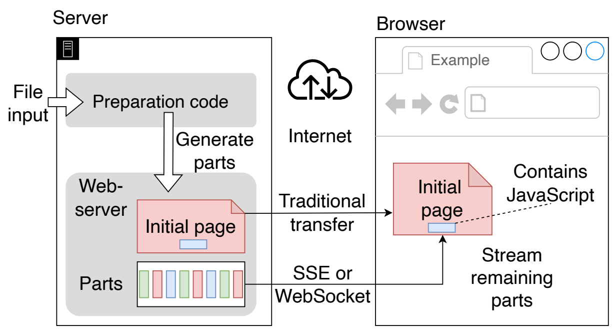


Figure 5.23: Concept diagram showing the streaming of web pages, including the transfer of the minimal initial page and a separate interface for streaming chunks. Source: [VS23a]

Current Streaming Options

The main method of streaming has to be addressed as well. Currently, multiple existing techniques allow for continuously sending data. All relevant approaches have pros and cons, which is why they are described next in more detail. Some approaches, like *chunked transfer encoding*, will not be compared due to irrelevance or deprecation [BPT15a]. This is followed by a comparison, selecting one or more candidates who fit the requirements.

HTTP-Server-Push
When web data is sent with the MIME-type "multipart/x-mixed-replace," the browser recognizes it as a document that updates when new data arrives. However, two drawbacks exist. First, a cache is needed to keep track of all data sent to the client. Secondly, not all browsers support this feature fully, with Chrome being one of them [22b]. Server push allows for one-way data transfer.
WebSocket
WebSocket is described in RFC 6455 preliminary as a protocol for continuous two-way communication between a client and a server [22ak]. However, this is not strictly necessary for this concept of stream-based delivery. Commonly browsers provide a built-in WebSocket API, which allows for easy connections [API15]. Connecting to a server in this way requires upgrading the protocol. This introduces an overhead, as described in paper [VS22b]. However, the broad support of all major browsers is a major positive aspect [22aw].
Server-Sent Events:
Server-Sent Events (SSE) do not need to upgrade to another protocol, compared to WebSocket. A Polyfill also exists, allowing to retrofit SSE capabilities to outdated browsers [Mod22]. This increases browser support significantly. However, SSE has one significant drawback: The number of simultaneously usable connections is limited by both Chrome and Firefox browsers to a maximum of six. Even though it was marked as a bug, both browser manufacturers decided not to fix the issue [22c; 22d]. As a result, if all connections are used up, no new connection can be made. It is unlikely that such a scenario will occur if SSE is only used for the initial page load and closed when finished, but it is not impossible.

Comparing all three options, only SSE and WebSockets are viable options. Both have pros and cons, but in general, both fulfill the requirements of being able to stream HTML, CSS, and JavaScript. Due to the non-exclusivity, both can be combined as a kind of fallback system. However, WebSockets should be used if possible due to the missing connection limitation. Still, both will be evaluated to test their capabilities.

5.5.2 Conclusion

The described concept showed that all goals of a stream-based web page delivery framework could be addressed fully. The final concept includes the before-developed frameworks `Essential` for CSS and `Waiter` as well as `AUTRATAC` for JavaScript. Combined with the presented HTML splitting and streaming approach, a method is presented that addresses all lacking areas shown in Figure 5.3. This concept was also published as a demo paper: [VS23a]. The approach will be implemented and evaluated next.

6 Evaluation

	HTML	JavaScript	CSS	Streaming
Essential (5.3.5)			✓	
Waiter/AUTRATAC (5.4.5)		✓		
Automated Test (6.3)	✓		✓	✓
User Satisfaction Test (6.5)				✓
Case Study Test (6.4)	✓	✓	✓	✓

Table 6.1: Overview over tests of the individual components. The check-mark represents that the individual components (HTML, JavaScript, CSS and Streaming) were tested in the section described on the left-most column.

The following section includes the tests for the complete system. At first, an overview is given of the number and types of tests conducted in general as part of this work.

6.1 Test Overview

As shown in Table 6.1, multiple technical tests are performed, spanning one or more of the main four components of this work. The table shows that every component will be tested at least twice in this chapter. This is because some tests are more suitable for one or more aspects if not all components are tested together. In this case, the individual tests for `Essential`, `Waiter`, and `AUTRATAC` are done in their respective concept chapter, as they showed that the component itself works individually. Now, in this chapter, three additional tests will be performed. The first test checks the CSS processing and HTML splitting capability on five popular web pages. The case study test is a full implementation of all components together by modifying "solarenergie.de" to be streamed. And lastly, the user satisfaction test checks if and how much users prefer the streamed or non-streamed version.

6.2 Test Metrics

First, the main categories of test metrics will be explained, as well as their importance for the overall evaluation. Generally, these consist of three types: code coverage and efficiency, loading time tests, and user satisfaction. All three will be discussed in more detail next.

6.2.1 Code Coverage and Efficiency

This first metric calculates how much code is used by what kind of code. More specifically, this allows testing how much render-blocking code is used until the page is displayed. This includes both JavaScript and CSS code. This metric is important for the overall test, as a large amount of render-blocking code that is used by 100% might not be ideal, but still valid due to all code being necessary for displaying a page. This metric will be used to check how the usage until render changes when comparing the original page to the one modified by the technique developed as part of this thesis.

6.2.2 Loading Time Measurements

Testing and measuring a web page's loading time is arguably the most relevant for the overall concept. As described in section 1.1, web pages that load slower directly result in revenue loss and user dissatisfaction. Therefore, measurements of the time it takes for content to appear on the screen present a high level of importance. A set of different performance markers will be used, like FCP or DOM Interactive, which are explained in more detail in section 2.7. In essence, the goal is to capture the first time content is displayed on the page, how long it takes for the whole page to be loaded, and how much overhead is produced for the stream connection setup.

6.2.3 User Satisfaction

Testing how users react to this new way of loading web pages is important to ensure the overall success of the developed technique. Therefore, an anonymous online questionnaire will be used to determine the overall preference of users. This is because the streamed version continuously adds content to the page, which is visible due to the non-render-blocking property of the initial code. As a result, this aspect needs to be tested as well.

6.3 Automatic Performance Test

This first section of the evaluation focuses on the parts that can be automatically converted. The main reason is the lack of access to the original code-base, which is necessary to convert JavaScript correctly. However, as shown by implementations of techniques like *Qwik* and `Waiter`, the maximum possible performance can be emulated by removing all render-blocking JavaScript. The isolated performance test of `Waiter` and `AUTRATAC` in subsection 5.4.5 showed that such a presumption is realistic. On this basis, the following tested pages will be prepared with both the described `Essential` framework in section 5.3 and the HTML splitting and streaming technique described in section 5.5. This ensures that the outcome reflects realistic results.

6.3.1 Test Pages and Preparation

For this automatic test, *file input* described in the concept diagram Figure 5.23 is gathered by using the "save page" feature, which is built into Google Chrome. It allows downloading a page, including all external files. The tool also re-writes all links to external resources to a local folder, which is also saved next to the initial HTML file. The pages were selected based on the research-oriented "Tranco" web page list from May 2022 (ID: 5Y67N) [Poc+18]. The first five pages that could be downloaded successfully (including JavaScript and CSS) and did not include syntax errors were selected. More specifically, the page must be parseable by "`Critical`," as it is the basis for `Essential`. This selection results in the following pages: https://netflix.com, https://microsoft.com, https://amazonaws.com, https://wordpress.org, and https://cloudflare.com. All web pages were part of the top 30 on the list.

6.3.2 Preparation Steps

First, all render-blocking JavaScript is removed to emulate the maximum achievable JavaScript optimization performance. This emulates the full use of `Waiter`, which allows for this modification, as described in section 5.4. Next, the code is transferred to the server, where automated scripts process the code further. In this case, `Essential` was used to render all included CSS as location-aware. At this stage, all CSS is already split and in-lined into minimal `style`-blocks that contain only the CSS necessary for the next section of HTML. This HTML file, which contains all render-blocking CSS, is split. As described in section 5.5.1, this is done by start- and end-tag, pure text content, or a non-splittable HTML chunk. The parts themselves are ordered in a specific way: first, the attributes of the HTML, HEAD, as well as BODY-tag are transmitted. If styling was done via classes and not tag types, this ensures that styling will be done correctly. Next, the content elements of the body (which also contain the in-lined CSS) are transmitted to ensure the fastest time to FCP. Lastly, the HEAD elements are sent. This order was chosen to measure the maximum content delivery performance. However, other sequences are also possible. The described order of elements is then saved as a JSON file for later use by the web server.

6.3.3 Loading Times

Similarly to the test in section 5.4.5, the loading speed test was performed via two machines linked by a local network. One machine provided the web pages via a set of *express* servers and a reverse proxy provided by *caddy* [22p; Ser22]. Overall, three versions were tested: one streamed via WebSocket, one via SSE, and the reference pages. The server device consisted of a 2021 MacBook Pro with 32GB RAM and an M1 Pro chip. The client was a 2020 MacBook Pro with a 2.3 GHz Intel i7, which used Apple's *Network Link Conditioner* to slow down connection speed to either 2, 4, 6, 8, or 10 Mbps. No additional package loss or additional delay was set. The test itself consisted of a puppeteer-controlled Chromium, which loaded the pages individually. The browser was reset and re-opened before every page load to ensure the cache was empty. The built-in performance API of Chrome was used to measure the loading speed. Additional performance markers were gathered to ensure that comparable data could be gathered for the stream-based pages. These include the time until the first package arrives at the client, as well as the time until the last package of the BODY.

6.3.4 Initial Page Size

As described in the concept in section 5.5, the initial transferred HTML file has to be as small as possible to ensure fast loading times. In this case, the initial file in its uncompressed state had a Size of 902 Byte for WebSocket and 862 Byte for SSE. The size difference is due to different APIs being used to establish the stream.

6.3.5 Results of First and Last Package

The time it takes to transfer the first package of data to the client details the overall overhead introduced by WebSocket or SSE, created by the backward-compatible approach chosen.

When comparing the time until the last package of the body, a user-centered transmission time is created, which reflects the time it takes to stream or download all user-visible content. The additional markers were set for both tested streaming approaches, and the DomComplete event was used for the reference. For all gathered data, the time until the responseStart was subtracted as a starting point to reduce fluctuations of network conditions [22ag].

The results for the first package are visible in Figure 6.1. As expected, there is a noticeable overhead both for SSE and WebSockets. Both took up to 0.1 seconds longer. As described before, optimizations are used to mitigate this issue. The results in Figure 6.2 show that this can be done successfully. At most, the data transfer streamed with SSE was 18.98 s faster than the reference. Therefore, initial streaming overhead can be neglected based on the total transfer results. Both WebSocket and SSE transferred the BODY data significantly faster than the reference.

6.3.6 Data Reduction

One of the main advantages of faster delivery is the optimizations made to the transferred code. In total, a reduction by 23.7% from 5.06MB per average reference page to 3.86MB for both average pages delivered by SSE and WebSocket streaming was measured via the Chrome DevTools. All individual size reductions are shown in Table 6.2.

The table shows that in one instance, the total file size could be reduced by 45%. However, cloudflare.com increased by 20% after the CSS was in-lined. Comparing the results with the data in Figure 6.2, the total improvement of the loading time is, therefore, larger than the

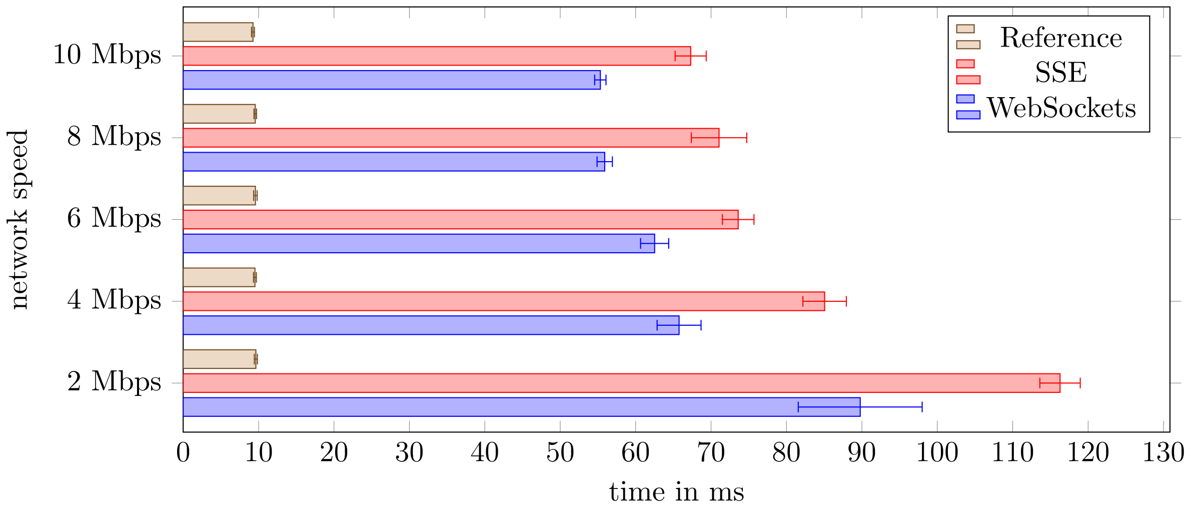


Figure 6.1: Time in ms until the first data of the document arrives, highlighting the connection overhead of the stream-based approaches. The graph includes standard error bars.

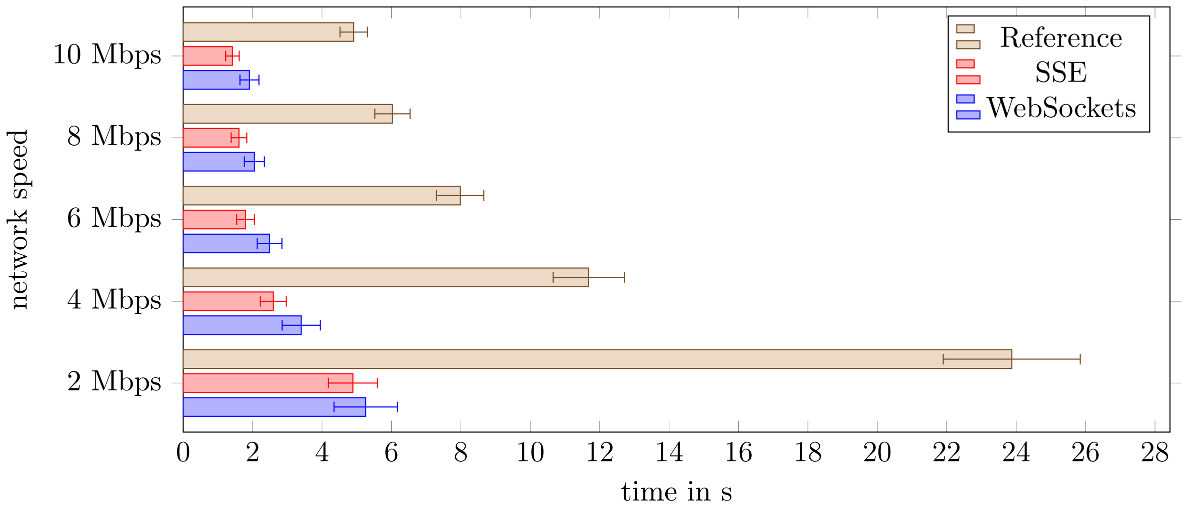


Figure 6.2: Time in s until the last data of the document arrives, highlighting the difference in transfer time until the final "body" data of the stream-optimized pages as well as the original. The graph includes standard error bars.

difference	**page**	**original**	**streamed**
-45%	amazonaws.com	10.2 MB	5.6 MB
-43%	netflix.com	3.7 MB	2.1 MB
-37%	microsoft.com	3 MB	1.9 MB
0%	wordpress.org	1.8 MB	1.8 MB
+20%	cloudflare.com	6.6 MB	7.9 MB

Table 6.2: Differences in the amount of transferred data compared between the streamed web page and the original, measured via Chrome DevTools

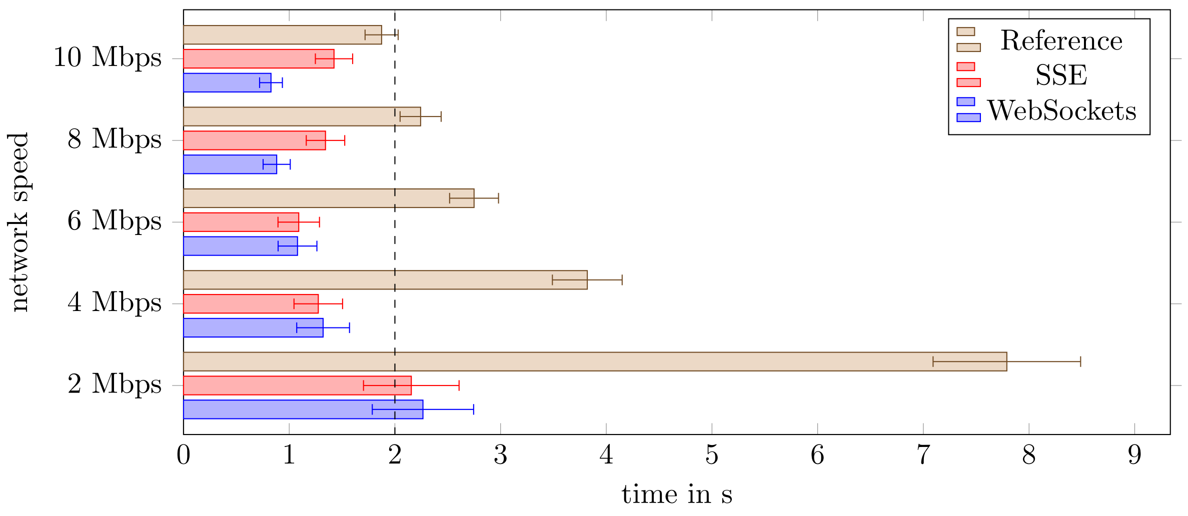


Figure 6.3: Time in seconds until the First Contentful Paint between the different versions. The 2-second mark highlights the acceptable access time, referencing Nah et al. [Nah04]. The graph includes standard error bars.

relative size reduction. This could be due to a lack of external files and a reduction of total requests.

First Contentful Paint

The expressiveness of the first and last packages is limited if the time it takes to show the first section of the page is lower than the original. Therefore, the First Contentful Paint (FCP) will also be measured. This marker is suitable to check for the 2-second mark, determined as the tolerable waiting time by Nah et al. [Nah04]. Figure 6.3 shows this results. Overall, the stream-based approaches are faster for all measured network speeds. The 2-second mark could also be reached in almost all cases. It also shows that the time it takes until the FCP increases exponentially with slower network speeds. However, this growth was significantly slower for stream-based approaches, as hypothesized in subsection 7.2.1. Furthermore, continuously adding data via streams is well-received by users in general, as shown in paper [VS22b].

However, considering SSE at network speeds higher than 6 Mbps, the time until FCP increases again require deeper investigation. WebSockets do not show the same behavior, instead further improving loading times. For this, the render behavior of the Chromium version used was examined in detail. It was thereby determined that at those speeds, the DOM modifications are done at a frequency that prevented re-rendering. Figure 6.4, which is a screenshot made from the Chrome DevTools, shows this behavior in more detail. The area marked with *"A"* describes the missing frames being rendered. The colored markers to the right of marker *"B"* are the individual tasks that result in the DOM modification. Therefore, the time shown in Figure 6.3 is not only the FCP but also the last modification made to render the full page. Therefore the total loading times via SSE are possibly faster than the time until FCP of the reference and, therefore, is not an issue.

6.3.7 Comparing WebSocket and SSE

As determined in section 5.5.1, both WebSocket and SSE are theoretically sufficient for streaming web pages. Both achieve similar results of total transfer time and initial overhead, as shown in Figure 6.1 and Figure 6.2. Due to this similarity, additional factors can be considered when comparing both. Mainly, the current maximum connection limit of SSE

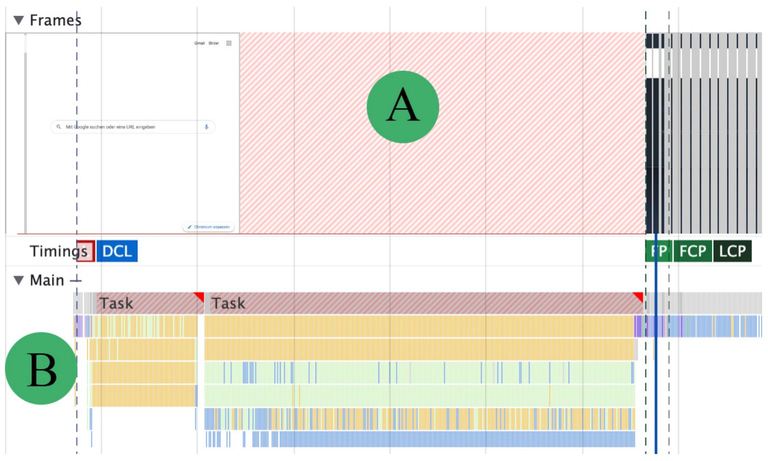
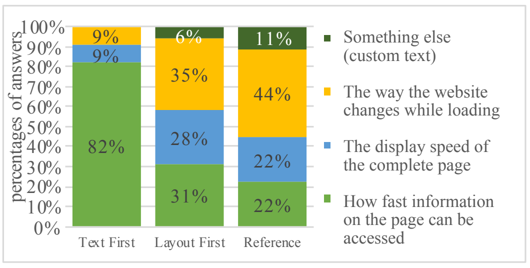
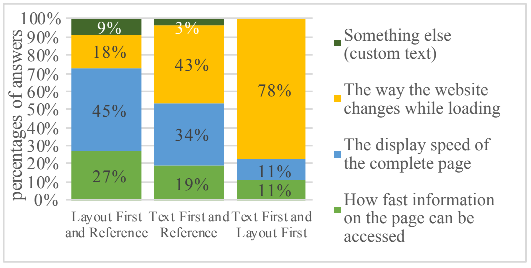


Figure 6.4: Missing render while rapid DOM modification when loading "amazonaws.com" at 10 Mbps. Marker "A" shows the lack of rendered frames. The yellow blocks to the right of marker "B" depict DOM inserts and animation frame requests.



(a) Reasons why users preferred a specific technique. Source: [VS22b]



(b) Reasons why users disliked techniques. Source: [VS22b]

Figure 6.5: Reasons why users prefer or dislike techniques

makes using WebSockets a more preferable approach. Additionally, SSE experienced render issues to rapid DOM-modification described in section 6.3.6, which were not measurable for WebSocket. In general, the usage of WebSocket is favorable.

6.3.8 Conversion Time

The time it took to automatically convert all code into a streaming-ready form was done by adding the time it took for each step. This consisted of converting CSS into a streamable form and splitting the HTML. All conversions of the five pages were done on a 2020 MacBook Pro with a 2.3 GHz Intel i7. On average, pages took 17.5 s to render the CSS with `Essential`. The HTML splitting took 40.8 ms. Longer conversion times for `Essential` are expected, as shown in section 5.3.5 due to the reliance on a headless Chromium browser.

6.3.9 Validity of Results and Limitations

This test only included testing the automated parts of the streaming process due to a lack of available code-bases and reliance on developer choice. Also, non-render-blocking resources like images are traditionally transferred in this test. The correctness of the DOM after the transmission was tested manually to ensure that an identical result is achieved.

6.4 Case Study Test

As shown in Table 6.1, the case study test is the last and most comprehensive, spanning all major components. This required gathering a code-base of a live, real-world web page. In this case, the web page was "solarenergie.de," provided via the company "Solarwatt." For this test, only the code-base was provided, without compensation, and was not bound to any form of condition. The code was provided "as-is" from the software service provider "3m5", which develops said web page. This code-base was chosen as it was built with a different set of tools than previous tests. For example, `Waiter` and `AUTRATAC` were evaluated on a "React" code-base to show it working with modern front-end frameworks. However, "solarenergie.de" is built using Neos, a CMS [22ad]. This is different, as the HTML is rendered before delivering a page, and it lacks JavaScript-defined components, which are present in frameworks like React [22ai]. Both of them are valid and represent modern ways of developing web pages. For this test, the initial web page will be described as the reference. Then, a copy of this page was modified so that all manual JavaScript improvements were made according to the concept. Afterward, the following steps were identical to the ones of the test in section 6.3 (automated test). This version is called "streaming" in the following section. Also, the same test software was used. However, as described in subsection 6.3.7, WebSocket is preferable to SSE for streaming web pages. Therefore, in this case, all stream-based tests were conducted using WebSocket only.

6.4.1 Type of Measurements

Two groups of measurements were conducted: Code efficiency measurements and loading speed tests. The first group of tests used the CSS and JavaScript code coverage provided by puppeteer, a browser automation tool. These measurements, which provide code usage information, were gathered over the initial rendering of the page without simulating input. Special care was taken to check that the coverage ranges do not overlap when counting the amount of code used. Furthermore, all HTML of the page was analyzed to find all instances of render-blocking links to JavaScript and CSS. For JavaScript, this included all in-file code snippets, as well as "script"-elements with the "src"-attribute and both "async"- and "defer"-attribute not set. For CSS, this included "style"-elements and "link"-elements with a valid "href"-attribute set to a CSS file, with the "media"-attribute not set or set to "all." Therefore, the individual code ranges could also be mapped to check if it belongs to render-blocking code.

The second suite of loading speed tests consists of a mixture of data points. First, the performance API is used to gather timestamps and measurements for loading time and markers. These include "RequestStart" and "ResponseStart," performance-API markers, which will be used in multiple of the following tests as a baseline [22ag]. This allows for reducing the impact of hosting-related performance impacts even further, as the time will only be counted by the first Byte arriving at the client. Also, own performance markers were gathered for the streamed version, as data from the performance API only measures the initial minimal HTML page. To gather comparable data, timestamps of the first and last sent packages were measured. Furthermore, it was determined by opening the web page in an unmodified Chromium browser on a standard full-HD monitor which element is the last transmitted package for the elements "Above-the-Fold." A timestamp is also gathered after the client receives this package.

6.4.2 Test Setup

The test setup was identical to the automated test described in section 6.3. The web pages were again hosted on one device, while another was used to fetch both versions. The server device consisted of a 2021 MacBook Pro with 32GB RAM and an M1 Pro chip. The client was a 2020 MacBook Pro with a 2.3 GHz Intel i7, which used Apple's *Network Link Conditioner* to slow down connection speed to either 2, 4, 6, 8, or 10 Mbps. Every page was loaded 100 times at each network speed and the averages were used. However, two modifications needed to be made. Both the reference and streamed versions of the web page were downloaded by the built-in Chrome download tool first and then hosted via an "express"-server as well as "caddy" as a proxy for HTTP/2 [22p; Ser22]. This is due to the limitation that the Neos environment, which runs via ddev, is not intended to run twice on one machine, as needed in this setup. These additional steps were taken to reduce hosting interference and ensure the results are as valid as in the automated tests. However, this is only done to maximize consistency and comparability of the test results and is explicitly not required in real-world scenarios. Secondly, the CSS code coverage tests needed to be done on a separate version of the page. The streamed version includes only non-render-blocking CSS. While puppeteer can handle this form of anonymous code for JavaScript, the CSS coverage tool returns empty results. Therefore, the results of the fully transmitted and streamed version of the page were saved as a static HTML file and then used for CSS coverage tests. Due to the code being identical, the CSS code coverage could therefore be measured accurately only in this way.

6.4.3 Extend of Tests and JavaScript Functionality of the Page

The test was performed on the start page of the website. However, due to the structure of the code project and the CMS itself, all JavaScript code is loaded for all possible sub-pages all the time as well. This is because the code is bundled in the original version, which does not allow for separation based on the individual page's needs. As a result, the initialization function is universal. However, internally, separate functionalities like the menu, search, image sliders, or video controls are all coded as modules in separate files. These were taken as a basis for splitting, which only requires modifying the initialization function with `Waiter` and `AUTRATAC`, as no cross-references between functions of different files could be found. Furthermore, all JavaScript code is streamed as well in this test. While not strictly required, delaying code can be done by in-lining the JavaScript code into the main HTML file. As a result, the JavaScript is delayed and non-render-blocking, but remains in its initial order. Therefore, all code keeps its full functionality. It was also checked manually that the required functions were called correctly and that no errors were logged in the console of the client's browser.

6.4.4 Results of the Code Efficiency Tests

The usage of code until render will be discussed for JavaScript and CSS separately, but both results are shown in Figure 6.6. In this case, efficiency means percentage of code used until render. In total, the CSS of the reference page is used by 24.9%, which is significantly higher than the average of 15.9% described in the analysis chapter 4. The CSS of the streamed version had an efficiency of 66.9%, roughly 2.7 times that of the reference. However, the efficiency number of the reference stays the same when considering only the render-blocking CSS, as all CSS of the reference page is loaded in this way. In contrast, all CSS of the streamed version is non-render-blocking, so there is no data for this measurement shown in Figure 6.6.

Similarly to CSS, the reference page uses 55.7% of all JavaScript, again higher than the average of 40.8% measurement in the analysis. Both cases show that the web page's code base was optimized above average before testing. However, the overall code usage is lower for the streamed version in this case. After investigating this issue further, it seems to be a result of code, which is, in turn, loaded by other JavaScript code. Frameworks like Google Tag Manager load additional trackers to a page, which is usually not recorded, as puppeteer

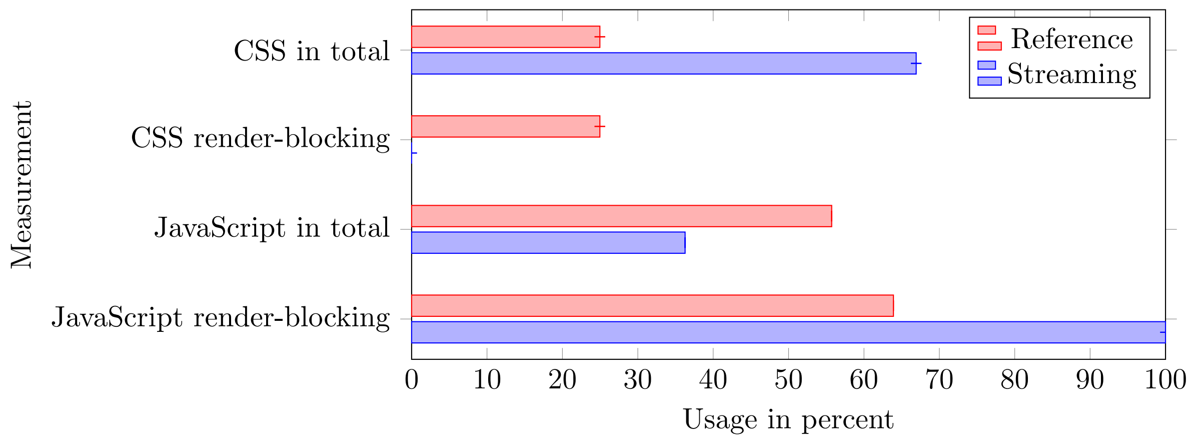


Figure 6.6: Percentage of code usage used by both versions in total until render. The graph includes standard error bars.

stops capturing the code coverage directly after the page is rendered. For the streamed version, the page needed to be captured longer, as puppeteer recognized only the render of the initial minimal HTML page. As a result, the additional code segments are also recorded, decreasing code efficiency. However, the results show major differences when discussing only the render-blocking JavaScript code. The JavaScript efficiency of the original (reference) page increases from 55.7% to 63.9%. For streaming, only the initial, minimal HTML page is render-blocking, which results in a 100% usage of the transmitted render-blocking code.

6.4.5 First Contentful Paint

In subsection 2.7.1 the importance of the FCP was described. Generally, it is a user-centric measurement, showing the point in time when the first content is visible to the user. This also shows progress, which is welcomed by users [Nah04; VS22b]. As shown in Figure 6.7, the time until the FCP is significantly faster in all cases and is also below the 2-second mark, determined as the tolerable waiting time until information retrieval according to Nah et al. [Nah04]. The 2-second mark is therefore also highlighted by a dotted line in Figure 6.7. Special care was taken to ensure that the measurement of the streamed version does not trigger for the initial page but only for transferred content. This is why the "First Paint (FP)" property of the performance API is not compared, as it would show skewed results. However, the data for the FCP highlights the true potential of the streaming method. As visible in Figure 6.7, the time until FCP does not change significantly independent of the network speed, as the first package will arrive at the same time independent of the length or complexity of the web page. In this case, at 2 Mbps, the FCP was reached at 2.81 seconds for the reference and only 0.47 seconds for the streamed version. This marks a decrease of 83.3%. Comparing this data to the automated test in Figure 6.3 shows that major loading time improvements can be made with an optimized page and non-render-blocking JavaScript.

6.4.6 Data Above-the-Fold

The FCP only marks the point for initial data reaching the client. However, the time it takes to transfer all data necessary for displaying the initial viewport can give a better understanding of loading time speeds.

This measurement is created for the streaming variant by calculating the time between the "responseStart" and the completed transfer of the last package, which is required for displaying the initial part Above-the-Fold in a Chromium browser opened on a full-HD monitor. For the reference page, the FCP is used as a best-case scenario. The timing

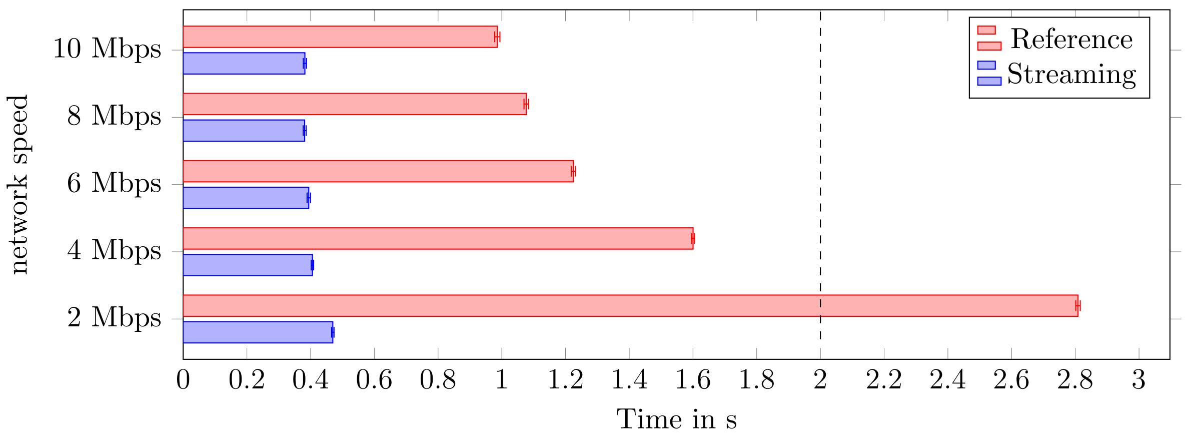


Figure 6.7: Time until the "FirstContentfulPaint"-performance marker. The dotted line represents the time for user-accepted loading time according to Nah et al. [Nah04]. The graph includes standard error bars.

difference is still similar, as shown in Figure 6.8. In all cases, the time to show the data "Above-the-Fold" is significantly faster than the reference and similar to the time until FCP. However, a more drastic increase in loading time is shown at 2 Mbps for the streaming variant. Still, there is significant headroom left until the 2-second mark is reached.

6.4.7 First Package and Overhead

The measurements of the first package show the overhead of both versions. For reference, this measurement included measuring the difference between "RequestStart" and "ResponseStart," while for streaming, the difference between "RequestStart" and the finished transfer of the first streamed package was used. As shown in Figure 6.9, the time until the first package is significantly faster for the reference pages. However, this is to be expected. For the streamed version, the "ResponseStart" only measures the first Byte of the initial minimal HTML file. This data shows major similarities to the same test shown in Figure 6.1. For 2 Mbps, the reference took 30.9 milliseconds until the first Byte, with 77.4 milliseconds for the streamed version.

6.4.8 Last Package

To compare the total transfer time, the time until the last package was gathered. For the reference page, the difference between the "ResponseStart" and "DomContentLoaded"-events were used. For the streamed web page, the time difference between the "ResponseStart" and the last streamed package was calculated. Furthermore, this included also all CSS and JavaScript, as the streamed version had all said code in-lined. Therefore, such a comparison shows the time a user has to expect until all render-critical components of the page are loaded and functional. Figure 6.10 shows this data. In all cases, the streamed version was significantly faster to transfer all content. Furthermore, the total transfer time for all network speeds falls below the 2-second mark. At 2 Mbps, the reference took 4.53 seconds to load all code, while the streamed version took only 1.34 seconds. However, it has to be noted that this measurement depends on the size and complexity of the web page. Comparing the data to Figure 6.2 shows that the proportions between the streamed and non-streamed versions stay the same, but the total loading time will differ significantly depending on the page.

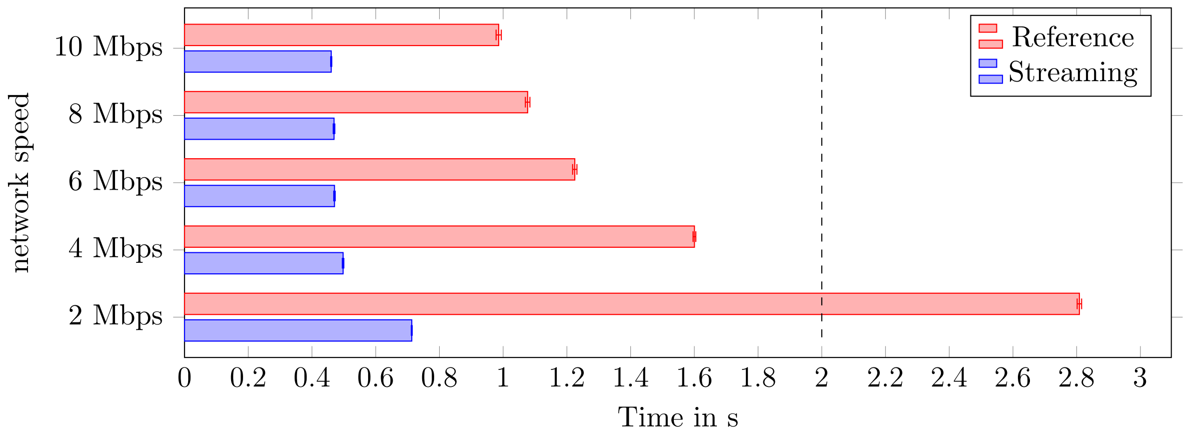


Figure 6.8: Time until all data "Above-the-Fold" is shown. For reference, the FCP marker is used as a "best case." For the streamed version, the time for the last package is measured, representing the last code necessary to display all HTML of the viewport of a Chrome opened on a full-HD monitor. Both measurements are calculated starting from the "ResponseStart"-marker. This reduces network interference, as it only measures from the first received Byte. Again, the 2-second mark of tolerable waiting time is shown with a dotted line. The graph includes standard error bars.

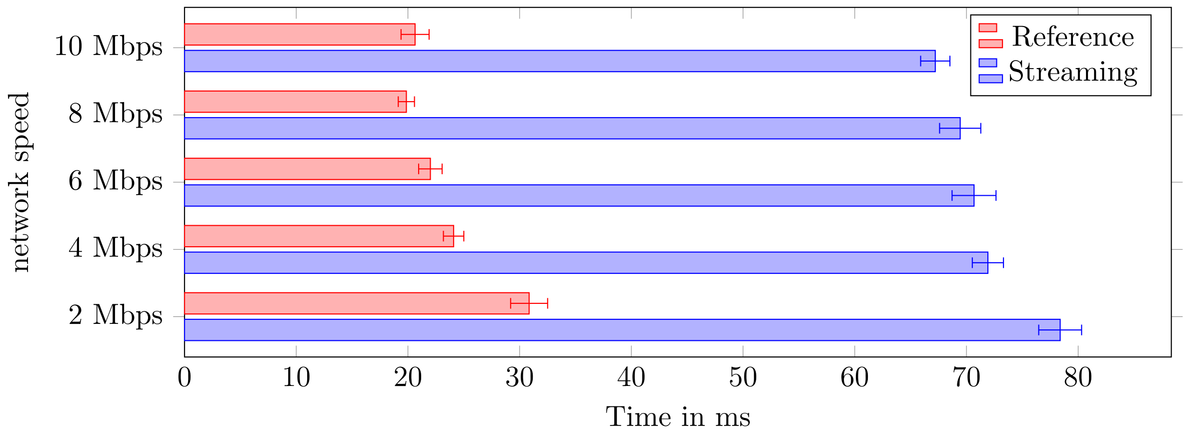


Figure 6.9: Time until the first transmitted package. Both measurements are calculated starting from the "RequestStart"-marker. This reduces network interference, as it only measures from the first received Byte. In this case, the "RequestStart" was used as the "ResponseStart" equals the first package for the reference. The streaming variant uses the first streamed package. The graph includes standard error bars.

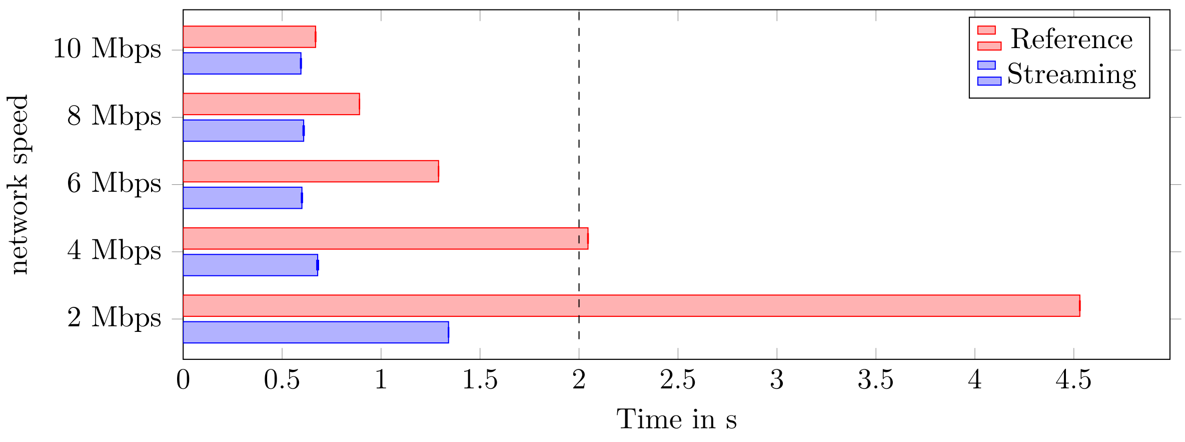


Figure 6.10: Time until the last transmitted package of render-blocking content. Both measurements are calculated starting from the "ResponseStart"-marker. This reduces network interference, as it only measures from the first received Byte. The streaming variant uses the last streamed package, and the "DomContentLoaded" marker is used for the reference. The graph includes standard error bars. The dotted line represents the time for user-accepted loading time according to Nah et al.

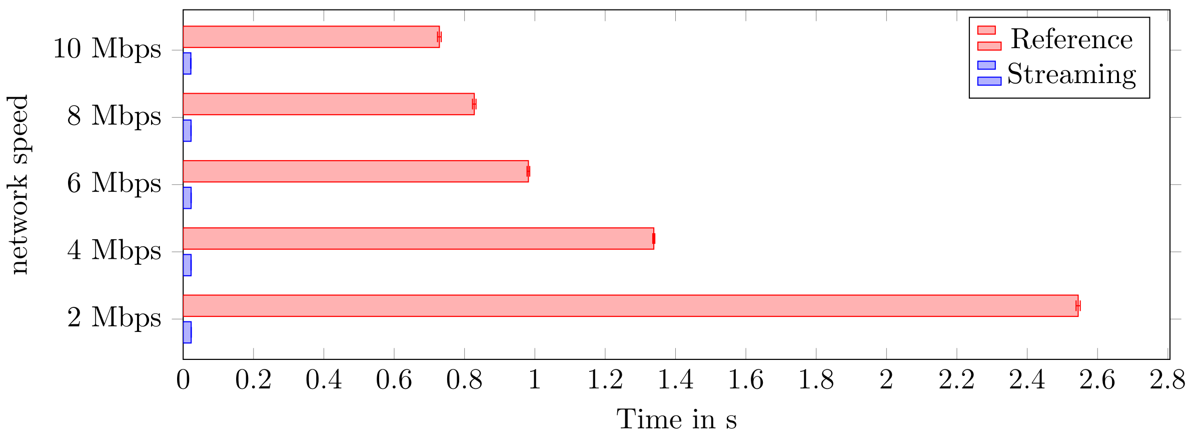


Figure 6.11: Time until the "DomInteractive"-performance marker. The graph includes standard error bars.

6.4.9 Interactive DOM

As discussed in subsection 2.7.3, measurements of the DOM interactivity need to be set into context when using the "domInteractive" performance marker. However, in this case, it is only used to measure the time it takes until the DOM has the potential to become interactive. Figure 6.11 shows the times it takes for both versions to reach this marker. Therefore, the focus will rely mainly on the streamed version and its relation to the reference page. It is visible that the streamed version only takes a fraction of a second to become interactive. However, as shown in Figure 6.10, it takes significantly longer to load all page data.

However, due to the HTML of the streamed web page being added continuously, this data proves an important feature of streamed web pages: as soon as the HTML is added to the page, it will be interactive. For example, if the first package contains a link to another web page, it is clickable as soon as it is displayed. In contrast to the reference page, the streamed version can therefore be interacted with even if the main DOM is not fully loaded.

6.5 User Satisfaction Tests

The following test will evaluate user satisfaction of a stream-based modified loading behavior. Due to DOM-Content being loaded continuously, a different visual experience will be noticeable by users at slow network speeds, loading the page primarily top-to-bottom. A web-based questionnaire was created for this test, with texts in German and English. This allowed for testing a large, unsupervised group of people. The test was adapted from the paper done by Moshagen and Thielsch, which also features a web-based study comparing page properties [MT13]. The results are also published in a paper [VS22b].

In this case, users were shown a video of different loading behaviors of different web pages. The selection of web pages is also explained in more detail in the paper [VS22b], which results in selecting one correctly render-able page from multiple web categories. They are `amazon.com` from "E-commerce And Shopping," `netflix.com` from "TV Movies and Streaming," `qq.com` from "News and Media" as well as `medium.com` from "Social Networks And Online Communities" [VS22b]. The capture was made at 64 KB/s network speed, limited by the Network Link Conditioner [22ae]. The video ensures that all participants see the exact same version and timing of the page. One shown version was the reference, and one showed the streamed page via WebSocket (called content first). The last version was a test that prioritized written content in the parsing order called text-first. This version was also streamed via WebSocket. After watching the video, the participants were asked to mark their favorite loading type on a Likert scale with five steps, ranging from "very bad" to "very good." Next, the reasons for their selection were also collected. The users could select one of the pre-defined answers or insert their own text, allowing maximum flexibility.

6.5.1 Results of the User Evaluation

The web-based survey was accessible from November 10th to November 23rd, 2020 [VS22b]. Overall, 138 individual answers were given. The user group is expected to consist mainly of computer science faculty members, as the survey invitation was published via a newsletter. Therefore, the participants might possess domain knowledge. However, no information on the displayed versions was given.

In general, 85.29% of the participants preferred the streaming version as described in section 5.5 (layout first). 8.09% preferred the streamed version, which prioritizes text content (text first), and 6.62% preferred the original loading behavior.

Mainly, users liked the layout first version, as "they liked the way the web page changes visually while loading" (selected by 35% of users) [VS22b]. The results are shown in more detail in Figure 6.5. The speed of information access was also important, with 31% of participants selecting this answer and the speed of displaying the information (selected by 28% of users). The most important reason for disliking the page is also the way the web page changes while loading, as shown in Figure 6.5 [VS22b]. So, in general, this is the most critical factor. Reducing visual change could potentially increase user satisfaction even further.

6.6 Summary of the Evaluation Results

In this chapter, the previously designed concept was tested. This evaluation consists of three tests. The first described test consisted of an automatic performance test, converting and evaluating five popular web pages according to the Tranco list. This test allowed for gathering data regarding the possible best-case performance of real web pages while simultaneously comparing WebSockets and SSE as streaming methods and gathering data about conversion times. The tests showed that significant performance increases could be made with a streamed web page version, surpassing the average file size decrease of 23.7% made by processing CSS and splitting the HTML. It also showed that WebSockets are the preferred streaming method due to faster connection times and fewer limitations than SSE, like the maximum possible simultaneous connections. Furthermore, the average processing time of `Essential` was determined to be 17.5 seconds, and the average HTML splitting time was 40.8 milliseconds. However, this test lacked JavaScript optimization, and no access to the original code-bases are available.

Next, a case study was performed using the code-base of "solarenergie.de." This test was similar to the automatic tests. However, in this case, the JavaScript of the page was adapted as well. Comparing these results showed that the CSS efficiency could be improved similarly to what the `Essential` code efficiency tests described in subsection 5.3.5 predicted. Furthermore, the efficiency of the JavaScript code, which is transferred via the initial minimal HTML file from the streamed version, is used by 100% until render. Next, the data for First Contentful Paint showed that even at 2 Mbps, the FCP was reached by the reference in 2.81 seconds, while the streamed version only took 0.47 seconds. This improvement is also visible for the transfer of all data necessary "Above-the-Fold." In this case, the streamed version needed 0.7 seconds instead of 2.81 seconds for the FCP of the reference. These improvements were made, even when the data showed that the time to first Byte is 60.6% slower for the streamed version at 2 Mbps due to the overhead of the initial connection. Still, all render-blocking data was shown to be transferred significantly faster with the streamed version, only taking 1.3 seconds instead of 4.5 seconds of the reference. It was also shown that during this process, the page is interactive and can be interacted with by the user while it is still loading.

Lastly, the user acceptance of the streaming behavior was tested. For this, different web pages from different web categories were downloaded and converted. Then, the original and streamable pages were recorded at 64 KB/s and shown side-by-side. The following test consisted of an anonymous online questionnaire, where users were shown the video and asked about their most and least favorite loading version. The test was filled out completely by 138 individuals, which preferred the streamed version by 85.29%. For both user groups who liked or disliked the streamed version, the most important aspect of their choice was the change of the page while it was loading. This showed that the total layout shift is not well received but welcomed at slow network speed as it indicates progress.

7 Conclusion

In this chapter, the research questions will be answered, and the work will be summarized. In it, the contributions of the dissertation are highlighted, followed by a discussion and future work.

7.1 Answers of the Research Questions

At the beginning of this thesis, four research questions were brought forward, which will now be answered in detail.

7.1.1 RQ1: Are streamed web pages with a reordered loading schedule faster than the traditionally loaded counterparts?

In short, the case study test in section 6.4 showed that streamed web pages load significantly faster, especially at slow network speeds. The streamed version reduced the time until FCP by up to 83.3% at 2 Mbps and was able to transfer the whole page (until the completed transfer of the last render-blocking package) by up to 70.4% less time at 2 Mbps. Furthermore, the results described in subsection 6.4.9 also showed that the streamed page is interactive while it is loading, allowing for even faster access to, e.g., links. This data is backed up by the automated test described in section 6.3, showing similar results for FCP and time to last package. However, in both tests, an initial stream connection overhead is visible due to the backward-compatible approach taken, as described in Figure 5.23. However, this overhead is negligible compared to the overall loading times. Therefore, streamed web pages with a reordered loading schedule present a significantly faster loading time, both compared to the time until the FCP and the transfer of all render-blocking elements.

7.1.2 RQ2: Are current (streaming-) protocols sufficient for delivering such web pages or are new protocols needed?

Current viable options for streaming data were discussed in section 5.5.1 and tested in subsection 6.3.7. The discussion concluded that both WebSocket and SSE are viable methods of transfer. However, when testing both options, it was concluded that it would be preferred due to faster connection times, fewer restrictions regarding the maximum number of simultaneous open connections, and fewer render-related issues. Furthermore, WebSockets are supported by the vast majority of browsers [22aw]. As a result, WebSocket can be declared fully sufficient for streaming web pages. Therefore, this research question can be answered with ”yes.”

7.1.3 RQ3: To which extent can the new method be used for existing web pages?

As shown as part of the related work in chapter 3, various options exist for optimizing web pages, like Marko or Qwik, which require re-writing the entire code-base to conform to the restrictions of the given framework. In contrast, all techniques developed in this thesis are framework-independent. `Essential`, the CSS optimization framework, is in itself universal, as it is based on `Critical`, which does not require any form of preliminary adaptation of the web page itself. Secondly, `Waiter` and `AUTRATAC` are both specifically developed to provide a flexible and easy tool for delaying JavaScript, independent of the framework. As proof, it was integrated into a React code-base described in subsection 5.4.5 and into a Neos code-base as part of the final evaluation (section 6.4). Lastly, the HTML splitting can be done universally as well, processing different types of HTML as part of the automatic tests (shown in section 6.3) and Neos-based HTML in section 6.4. The streaming itself is fully independent of the initial web page, only requiring a set of sendable chunks. Therefore, it was shown that, yes, the technique developed in this thesis can be integrated into various existing technologies and is not bound to one specific style of creating a web page.

7.1.4 RQ4: How much of this process can be automated in order to reduce development effort?

In order to ease development efforts, the goal was to automate this technique as much as possible. This goal was achieved, as the only exception is the manual preparation of JavaScript code. This was discussed in detail in section 5.4, showing that no automatic JavaScript optimization is currently feasible due to limited accuracy. Therefore, the `AUTRATAC` framework was created, which is able to help developers integrate `Waiter` easier. `AUTRATAC` consists of a Babel plugin that can convert asynchronous JavaScript code into code that already integrates all calls to `Waiter`. The developer needs to simply mark functions as ”async” and ensure all necessary calls are awaited. Special care was also taken to make the call syntax of `Waiter` as easy as possible. All other frameworks, like `Essential` or the HTML code splitting, can be done fully automatically. Therefore, the answer to this research question is: All steps except the preparation of JavaScript code can be automated. However, this assumes that there are no syntactical errors in the HTML and CSS code of the page.

7.1.5 RQ5: To which extent is the new loading behavior accepted by users?

Modifying the loading behavior of web pages represents the fundamental idea of this work. However, it also changes how users perceive and consume web content, which radically differs from the traditional all-or-nothing method from a render-blocking file-based page structure. Therefore, a user study was conducted in section 6.5, which tested different loading types. The first loading method is a streamed version that prioritizes text over everything else, including HTML structure. This version is called ”text first.” The second version is the exact loading behavior produced by the techniques of this work, called ”layout first.” The third version was a control version (”reference”) with the traditional render-blocking and file-based loading behavior. The results showed that only 6.62% of the 138 participants preferred the ”reference” version. The ”text first” loading method was preferred by 8.09%. However, the technique developed in this work was by far the most-liked loading behavior, with 85.29% of participants selecting it as their preferred loading type. As a result, any form of streamed web page represents an improvement over the status quo. However, the significant preference for the ”layout first” version developed as part of this work shows that such modification is also ready for real-world applications.

7.2 Discussion

Streaming web pages is a fundamental change in how web pages load. Therefore, some implications will be discussed next.

7.2.1 Reasons for Streaming a Web Page

A full streaming approach has one unmatched advantage over other approaches: the initial loading time of web pages does not depend on the total size of the web page. While traditional page loading is based on the size and loading type of render-blocking resources, splitting the page and converging the times until FCP to each other. This means that if a user only needs the menu or login button commonly placed at the top of a web page, the page can be left without loading it fully. Both allow for a more user-centered navigation approach while simultaneously providing flexibility for larger and more complex web pages.

7.2.2 Advertisements

A study created by OnAudience estimated that in 2017, ad-blockers were responsible for a $42 billion in global revenue loss, which is an estimated increase of $14 billion from $28 billion in 2016 [17]. Therefore, advertisements are a huge economic factor on the internet. It is, therefore, in the best interest of providers to block the render of a web page until the advertisement is shown. While streaming web pages allow for faster navigation without said render-blocking property, it also makes it harder to block, as individual requests are missing, which could be blocked by said extensions. Therefore, the impact of streaming web pages is unclear but also outside the scope of this work.

7.2.3 Caching and CDNs

Streaming web pages will break traditional file-based caching approaches. However, they exist as a result of how web pages are built. This is taken as an advantage, as cache-busting via IDs is a common way of forcing an update to a web page. Similarly, if a 100% streamed approach is targeted, new approaches might be possible to tell the server beforehand which code sections already exist by using a hash or an ID. This could prevent streaming the individual code elements but also enable the re-use of individual code blocks across multiple pages. Secondly, fewer caches might be needed by optimizing the stream if the loading speed is the sole goal. As described by Tenni Theurer, former Senior Director at Yahoo, 40-60% of users visit their web page with an empty cache [07]. This highlights the importance of focusing on the "empty cache experience," as Theurer called it [07]. In this case, the main page might not require caching render-blocking resources. The other improvement made by caches is the decrease in server load, which is also one of the goals of CDNs. This is explained in more detail in subsection 7.2.4. Additionally, by adapting the way CDNs load, it is also possible to serve the stream directly via CDNs. This has two advantages. First, the initially sent file is as small as possible, making it harder to overload the server. Secondly, the stream can easily be pointed to another server, allowing new load distribution methods. One example would be to include a list of servers ordered by priority, allowing the client to autonomously use a backup if the first server is unreachable. As the streams do not need two-way communication, the transfer of data more closely represents the provision of classically streamed media, which is also already offered by companies like Akamai [22z]. It is, therefore, plausible that such adaptations can be made. Lastly, companies offering streams like WebSockets via CDNs already exist [22ax]. These show that the concept is directly usable with CDNs in mind, as the required technology already exists.

7.2.4 Streaming Other Media

According to the HTTP Archive page weight report, resources like images and video represent a significantly larger portion compared to HTML, CSS, and JavaScript [22s]. As these are non-render-blocking, they might also be loaded without using a stream, allowing them to be served by a traditional CDN. This is the reason why they are not focused in this work. However, streaming other elements like images and video is already possible. Especially web videos are already streamed, with existing standards and APIs [22x]. For images, CDN providers like Cloudflare already showed that by taking advantage of HTTP/2 multiplexing, images could also be progressively streamed [Gal20]. Combining both or using additional streams for video and images is, therefore, already possible.

7.2.5 Implications of Streaming Web Pages

The backward-compatible approach in this work only represents the first step towards fully streamed pages by providing the necessary groundwork. More precisely, it keeps in mind that in the future, browsers might include the capability of streaming web pages directly, eliminating the need for the initial HTML file. This eliminates the current overhead and speeds up web page loading even further. For this, a standard has to be created to ensure browser and server compatibility. The underlying protocol, HTTP/2 and HTTP/3, already provide sufficient options for streaming content. Eliminating the last step of re-assembling entire files before rendering would therefore be a sufficient change. This could be done by inlining all render-blocking code and adding a tag like `<render/>` to the page, which can tell the browser to display the previously transmitted section. With this single change, it is possible to eliminate a large portion of the splitting scripts described in this concept, as the choice can be made by the developer. However, it would also allow for using it incorrectly by still linking render-blocking external files, possibly diminishing improvements made by slitting code. Finding the best option to standardize and integrate such a stream-based solution into existing web technology is, therefore, part of future work and outside the scope of this thesis.

7.3 Summary

At the beginning of this thesis, a large-scale analysis was performed, spanning all downloadable pages of the top 10.000 web pages according to the Tranco-list [Poc+18]. This analysis aimed to gather data about the render-blocking properties of web page resources, including HTML, JavaScript, and CSS. It further gathered data about code coverage, giving insight into how much of the render-blocking code is actually used. Therefore, the structural optimization potential could be determined. Less render-blocking code will, in turn, lead to faster loading times due to requiring less data to display the page. The analysis showed that there is significant optimization potential left. On average, modern web pages are built with a combined 86.7% of JavaScript and CSS, the rest being HTML. Both JavaScript and CSS are loaded mostly render-blocking, with 91.8% of JavaScript and 89.47% of CSS loaded in this way. Furthermore, only 40.8% of JavaScript and 15.9% of CSS is used until render. This shows that, on average, web pages have significant room for improvement. The concept, which is then developed based on the results of this analysis, aims to load web pages in a new way by streaming all render-blocking content. The related work showed that multiple sub-techniques are required first, which were conceptualized next. First, an optimization and splitting tool for CSS is proposed, called `Essential`. This is followed by an optimization framework concept for JavaScript, consisting of `Waiter` and `AUTRATAC`. Lastly, a backward-compatible approach was developed, which allows for splitting HTML and streaming all content to a client. The evaluation showed that the streamed web page loads significantly faster when comparing FCP, content "Above-the-Fold," and total transfer time of all render-blocking resources of the document. For example, the case study test determined that the streamed page could reduce the time until FCP by 83.3% at 2 Mbps and the time until the last render-blocking data is transferred by up to 70.4% at 2 Mbps. Furthermore, existing streaming methods were also compared, determining that WebSockets meets the requirements to stream web page content sufficiently. Lastly, an anonymous online user questionnaire showed that 85% of users preferred this new style of loading pages.

7.4 Future Work

This thesis showed that the concept of streaming web pages is viable and that it also can be created in a backward-compatible way. However, even in this concept, it has to be acknowledged that this can be improved further in the future. As part of the concept, an initial file is transferred every time a new stream-based page is requested. This introduces an overhead. However, it is nearly negligible. Still, this is required, as modern browsers do not support such stream-based delivery. In the future, when this form of stream-based pages becomes more popular, the base method can be integrated into browsers directly without removing its current backward-compatible functionality. One possible method would be for browsers to request the main page and the streamed page via a standardized port directly. By removing the overhead completely, even faster pages are possible. Furthermore, this work focused on streaming only render-critical content by highlighting the fact that all other content does not affect loading times. However, as shown by Cloudflare, other media, like pictures, can be progressively loaded as well [Gal20]. Therefore, future work could also focus on non-render-blocking elements to provide a fully streamed experience beyond the initial page load.

8 Appendix

8.1 CSS Processing Example

This example shows how CSS is processed via `Essential` and prepared for streaming.

```
<!doctype html>
<html>
    <head>
        <style>
        p {
            color: blue;
        }
        h1 {
            color: red;
        }
        /* comment */
        p {
            color: blue;
        }
        a {
            color: green;
        }
        </style>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

Figure 8.1: Original HTML Code

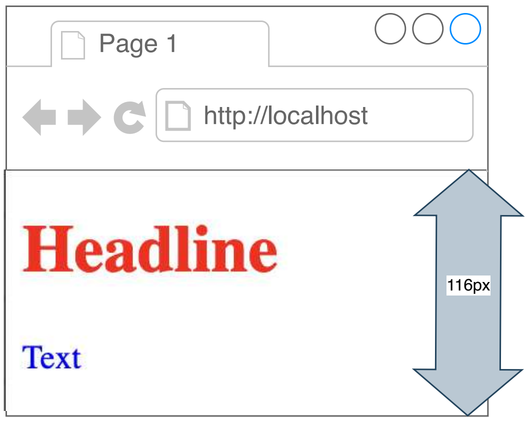


Figure 8.2: Step 1: Get the full page height by opening the page in a browser

```
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

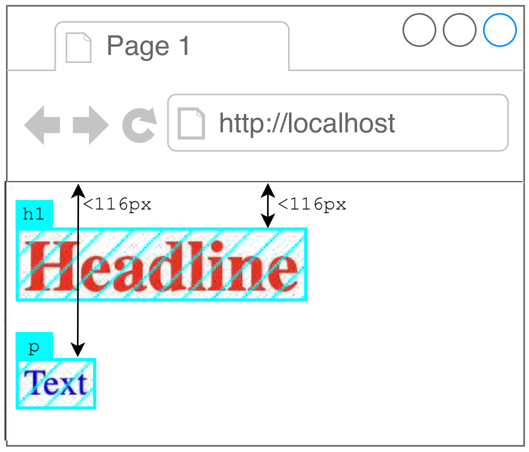
```
p {
    color: blue;
}
h1 {
    color: red;
}
/* comment */
p {
    color: blue;
}
a {
    color: green;
}
```

Figure 8.3: Step 2: Extracting CSS

```
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

```
p {
    color: blue;
}
h1 {
    color: red;
}
/* comment */
p {
    color: blue;
}
a {
    color: green;
}
```

Figure 8.4: Step 3: Extracting all selectors: "p","h1", "p" and "a"



```html
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

```css
p {
    color: blue;
}
h1 {
    color: red;
}
/* comment */
p {
    color: blue;
}
a {
   color: green;
}
```

Figure 8.5: Step 4: Match selectors with the HTML and test if they are visible on the page (inside the top 116px) (matches are highlighted)

```html
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

Critical CSS:

```css
p {
    color: blue;
}
h1 {
    color: red;
}
p {
    color: blue;
}
```

Uncritical CSS:

```css
a {
    color: green;
}
```

Figure 8.6: Step 5: Sort into ”critical” (top) and ”uncritical” (bottom). Note: the comment got removed, as it is not a valid selector. This example is also partially published at: [VS23a]

```html
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

```css
p {
    color: blue;
}
h1 {
    color: red;
}
```

Uncritical CSS:

```css
a {
    color: green;
}
```

Figure 8.7: Step 6: Remove code duplicates (second "p"-class)

```html
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <h1>Headline</h1>
        <p>Text</p>
    </body>
</html>
```

```css
ID: 0 p {
    color: blue;
}
ID: 1 h1 {
    color: red;
}
```

Uncritical CSS:

```css
a {
    color: green;
}
```

Figure 8.8: Step 7: Giving every critical selector a unique ID

```html
<!doctype html>
<html>
    <head>
    </head>
    <body>
        <h1 Match: 1 >Headline</h1>
        <p Match: 0 >Text</p>
    </body>
</html>
```

```css
ID: 0 p {
    color: blue;
}
ID: 1 h1 {
    color: red;
}
```

Uncritical CSS:

```css
a {
    color: green;
}
```

Figure 8.9: Step 8: Matching of critical CSS classes and HTML object locations

```html
<!doctype html>
<html>
  <head>
  </head>
  <body>
    <style>h1 {color: red;}</style>
    <h1>Headline</h1>
    <style>p {color: blue;}</style>
    <p>Text</p>
    <style>a {color: green;}</style>
    </body>
</html>
```

Figure 8.10: Optional step 9: Exemplary inlining of CSS into HTML so that the HTML can be split and streamed with all CSS included at the correct location. All CSS is loaded before it is used (highlighted). The uncritical CSS is loaded at the end.

8.2 JavaScript Preparation Example

This example shows how JavaScript is prepared for streaming by the developer using `Waiter` and `AUTRATAC`. The following setup is given:

```html
[...]
<button onclick="toggleMenu(this)">
Open Menu
</button>
<div id="menu" class="hidden">
  <a href="index.html">Homepage</a>
  <a href="shop.html">Shop</a>
</div>
[...]
```

Figure 8.11: HTML code used

```css
.hidden{
  display:none;
}
```

Figure 8.12: CSS code used

```javascript
// toggles the Menu, gets called
function toggleMenu(button){
  const menu = document.getElementById("menu");
  toggleVisibility(menu) //important
  toggleMenuText(button) //not important
}
//toggles the visibility of an element
function toggleVisibility(element){
 element.classList.toggle("hidden");
}
//toggles the menu button text
function toggleMenuText(button){
  button.innerText = button.innerText=="Close Menu"?"Open Menu":"Close Menu";
}
```

Figure 8.13: Original JavaScript code used in this example

```javascript
// toggles the Menu, gets called
async function toggleMenu(button){
  const menu = await document.getElementById("menu");
  await toggleVisibility(menu) //important
  await toggleMenuText(button) //not important
}
//toggles the visibility of an element
async function toggleVisibility(element){
 await element.classList.toggle("hidden");
}
//toggles the button text
function toggleMenuText(button){ ← not marked async as it does not contain other function calls
  button.innerText = button.innerText=="Close Menu"?"Open Menu":"Close Menu";
}
```

Figure 8.14: Step 1: Manually marking all relevant functions as asynchronous by adding "async" and "await" (highlighted)

```
async function toggleMenu(button) {
  const menu = await  __w (async () => document.getElementById("menu"));
  await  __w (async () => toggleVisibility(menu));
  await  __w (async () => toggleMenuText(button));
}

async function toggleVisibility(element) {
  await  __w (async () => element.classList.toggle("hidden"));
}

function toggleMenuText(button) {
  button.innerText = button.innerText=="Close Menu"?"Open Menu":"Close Menu";
}
```

Figure 8.15: Step 2: Manually use `AUTRATAC` to insert `Waiter`-calls (highlighted). This also removes comments due to the "`--no-comments`"-option of Babel being used.

`main.js`

```
async function toggleMenu(button) {
  const menu = await __w(async () => document.getElementById("menu"));
  await __w(async () => toggleVisibility(menu));
  await __w(async () => toggleMenuText(button));
}
```

`menu_visibility.js`

```
async function toggleVisibility(element) {
  await __w(async () => element.classList.toggle("hidden"));
}
```

`menu_text.js`

```
function toggleMenuText(button) {
  button.innerText = button.innerText=="Close Menu"?"Open Menu":"Close Menu";
}
```

`index.html`

```
[...]
<script src="main.js"></script> ← critical, captures user input
<button onclick="toggleMenu(this)">Open Menu</button>
<div id="menu" class="hidden">
  <a href="index.html">Homepage</a>
  <a href="shop.html">Shop</a>
</div>
<script src="menu_visibility.js" defer ></script> ← less critical
[...]
<script src="menu_text.js" defer ></script> ← not critical
[...]
```

Figure 8.16: Step 4, option 1: Splitting code into separate chunks (files) and linking them in the HTML. The less critical code is deferred (highlighted). Note: the JavaScript itself will not be streamed. Therefore, it can also be used without streaming the page and improve render times due to less render-blocking code.

```
[...]
<script>
async function toggleMenu(button) {
  const menu = await __w(async () => document.getElementById("menu"));
  await __w(async () => toggleVisibility(menu));
  await __w(async () => toggleMenuText(button));
}
</script>
<button onclick="toggleMenu(this)">Open Menu</button>
<div id="menu" class="hidden">
  <a href="index.html">Homepage</a>
  <a href="shop.html">Shop</a>
</div>
<script>
async function toggleVisibility(element) {
  await __w(async () => element.classList.toggle("hidden"));
}
</script>
[...]
<script>
function toggleMenuText(button) {
  button.innerText = button.innerText=="Close Menu"?"Open Menu":"Close Menu";
}
</script> ← loaded at the end, before the end of the body
</body>
[...]
```

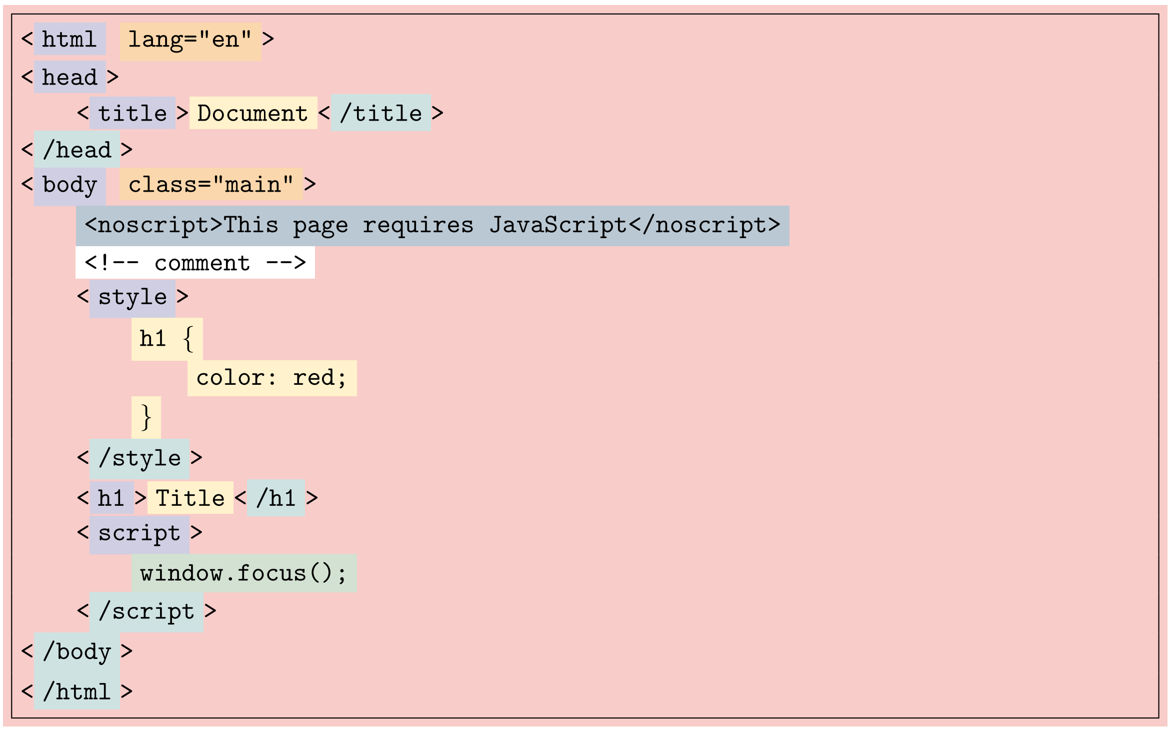
Figure 8.17: Step 4, option 2: In-lining all JavaScript code into the main HTML. The code would be render-blocking in this state, but by splitting the HTML with the in-lined JavaScript, the streamed variant would load the scripts in order, making them non-render-blocking. Therefore, the less important JavaScript is loaded directly at the end of the body. The script part before the menu is important, as it captures the user's input (clicking the "Open Menu"-button). This procedure of in-lining all JavaScript is, therefore, used in the final case study (section 6.4).

8.3 HTML Splitting Example

This example shows how different aspects of an HTML-based page are split by the technique developed in this work.

```html
<html lang="en">
<head>
    <title>Document</title>
</head>
<body class="main">
    <noscript>This page requires JavaScript</noscript>
    <!-- comment -->
    <style>
        h1 {
            color: red;
        }
    </style>
    <h1>Title</h1>
    <script>
        window.focus();
    </script>
</body>
</html>
```

Figure 8.18: Original HTML of the splitting example, containing inline style and inline scripts, as well as a non-splittable element (”noscript”, see section 5.5.1).



```
< html  lang="en" >
< head >
    < title > Document < /title >
< /head >
< body  class="main" >
    <noscript>This page requires JavaScript</noscript>
    <!-- comment -->
    < style >
        h1 {
            color: red;
        }
    < /style >
    < h1 > Title < /h1 >
    < script >
        window.focus();
    < /script >
< /body >
< /html >
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.19: Step 1: Parse and categorize every element of the page required for streaming. Note that the comment was detected but does not represent a relevant element. The individual package types are explained in section 5.5.1.

HTML-tag attributes

```
lang="en"
```

HEAD-elements:

```
<title> Document </title>
```

BODY-tag attributes

```
class="main"
```

BODY-elements:

```
<noscript>This page requires JavaScript</noscript> <style> h1 { color: red; }
↪ </style> <h1> Title </h1> window.focus();
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.20: Step 2: Remove comments, HTML, HEAD, and BODY tags, as they are already available in the initial minimal HTML file. Furthermore, SCRIPT-tags are removed, as scripts require element creation on the client (explained in section 5.5.1). The individual elements are also sorted into groups.

(1) HTML-tag attributes

```
lang="en"
```

(2) BODY-tag attributes

```
class="main"
```

(3) BODY-elements:

```
<noscript>This page requires JavaScript</noscript> <style> h1 { color: red; }
↪ </style> <h1> Title </h1> window.focus();
```

(4) HEAD-elements:

```
<title> Document </title>
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.21: Step 3: Re-order all element groups so that they prioritize loading performance. In this case, both HTML- and BODY attributes need to be sent first to guarantee the correct layout and functionality (as an identical DOM needs to be generated). Then, the content of the BODY is sent before the HEAD to prioritize content delivery speed. As shown in both the CSS splitting and JavaScript preparation examples (section 8.1 and section 8.2), the HEAD does not need to contain any code relevant to rendering the page (as it can be in-lined) and will therefore be delayed.

8.4 Streaming HTML Example

The example described in section 8.1 showed that all CSS can (and should) be in-lined for stream-based page delivery. The same was shown in section 8.1 for JavaScript. In section 8.3, the example used both in-lined CSS and JavaScript in the HTML splitting example. Therefore, all render-critical resources can be processed and split. In this example, the output of the example shown in section 8.3 is used as input to show how every chunk will be sent. Here, the example will mainly show how the HTML is modified by the individual chunks.

Initial, minimal HTML

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <script> ← initial setup script, includes code to insert received chunks
        [...]
    </script>
</body>
</html>
```

(1) HTML-tag attributes

```
lang="en"
```

(2) BODY-tag attributes

```
class="main"
```

(3) BODY-elements:

```
<noscript>This page requires JavaScript</noscript> <style> h1 { color: red; }
↪ </style> <h1> Title </h1> window.focus();
```

(4) HEAD-elements:

```
<title> Document </title>
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.22: The initial, minimal HTML page (top), with the individual groups to send at the bottom.

HTML-DOM on the client

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body class="main">
    <script> [...] </script>
</body>
</html>
```

(1) HTML-tag attributes

```
lang="en"
```

(2) BODY-tag attributes

```
class="main"
```

(3) BODY-elements:

```
<noscript>This page requires JavaScript</noscript> <style> h1 { color: red; }
↪  </style> <h1> Title </h1> window.focus();
```

(4) HEAD-elements:

```
<title> Document </title>
```

The colors represent: Start-tag , End-tag , Text , Unsplittable , JavaScript and Attribute .

Figure 8.23: Step 1: After the initial minimal HTML file is transferred to the client and the stream connection is made, the attributes for both the HTML- and the BODY tag are transferred and set via `setAttribute()`.

```html
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body class="main">
    <script> [...] </script>
    <noscript>This page requires JavaScript</noscript>▼
</body>
</html>
```

(3) BODY-elements:

```
<noscript>This page requires JavaScript</noscript> <style> h1 { color: red; }
↪ </style> <h1> Title </h1> window.focus();
```

(4) HEAD-elements:

```
<title> Document </title>
```

The colors represent: Start-tag , End-tag , Text , Unsplittable , JavaScript and Attribute .

Figure 8.24: Step 2: The first HTML-chunk of unsplittable HTML is transferred (highlighted). The red triangle shows the insertion cursor, where the code is inserted into the page.

```html
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body class="main">
    <script> [...] </script>
    <noscript>This page requires JavaScript</noscript>
    <style>▼</style> ← automatically generated by the browser
</body>
</html>
```

(3) BODY-elements:

```
<style> h1 { color: red; } </style> <h1> Title </h1> window.focus();
```

(4) HEAD-elements:

```
<title> Document </title>
```

The colors represent: Start-tag , End-tag , Text , Unsplittable , JavaScript and Attribute .

Figure 8.25: Step 3: Inserting the first regular HTML start tag. Note that the end tag ("</style>") is auto-generated by the browser, and the cursor is placed inside.

```html
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body class="main">
    <script> [...] </script>
    <noscript>This page requires JavaScript</noscript>
    <style>h1 { color: red; }▼</style>
</body>
</html>
```

(3) BODY-elements:

```
h1 { color: red; }  </style>  <h1>  Title  </h1>  window.focus();
```

(4) HEAD-elements:

```
<title>  Document  </title>
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.26: Step 4: Inserting the content of the first element.

```html
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body class="main">
    <script> [...]  </script>
    <noscript>This page requires JavaScript</noscript>
    <style>h1 { color: red; }</style>▼
</body>
</html>
```

(3) BODY-elements:

```
</style>  <h1>  Title  </h1>  window.focus();
```

(4) HEAD-elements:

```
<title>  Document  </title>
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.27: Step 5: Closing the element. This means that the insertion cursor is moved to the right after the end tag.

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body class="main">
    <script> [...] </script>
    <noscript>This page requires JavaScript</noscript>
    <style>h1 { color: red; }</style>
    <h1>Title</h1>▼
</body>
</html>
```

(3) BODY-elements:

```
<h1>  Title  </h1>  window.focus();
```

(4) HEAD-elements:

```
<title>  Document  </title>
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.28: Step 6: Inserting the headline in the same way.

```
<!DOCTYPE html>
<html lang="en">
<head>

</head>
<body class="main">
    <script> [...] </script>
    <noscript>This page requires JavaScript</noscript>
    <style>h1 { color: red; }</style>
    <h1>Title</h1>
    <script type="text/javascript">window.focus();</script>▼
</body>
</html>
```

(3) BODY-elements:

```
window.focus();
```

(4) HEAD-elements:

```
<title>  Document  </title>
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.29: Step 7: Inserting a script by creating an element via *document.createElement("script")*, setting the type and content, and appending it to the BODY. JavaScript appended in the HEAD can be moved to the body beforehand or left as-is when used with `Waiter`.

```html
<!DOCTYPE html>
<html lang="en">
<head>
<title>Document</title>▼
</head>
<body class="main">
    <script> [...] </script>
    <noscript>This page requires JavaScript</noscript>
    <style>h1 { color: red; }</style>
    <h1>Title</h1>
    <script type="text/javascript">window.focus();</script>
</body>
</html>
```

(4) HEAD-elements:

```
<title> Document </title>
```

The colors represent: Start-tag, End-tag, Text, Unsplittable, JavaScript and Attribute.

Figure 8.30: Step 8: After all content of the BODY is transferred, the HEAD elements follow. For this, the insertion cursor is moved to the HEAD. There, the <title>-element is inserted in the same way as described before.

```html
<!DOCTYPE html>
<html lang="en">
<head>
<title>Document</title>
</head>
<body class="main">
    <script> [...] </script>
    <noscript>This page requires JavaScript</noscript>
    <style>h1 { color: red; }</style>
    <h1>Title</h1>
    <script type="text/javascript">window.focus();</script>
</body>
</html>
```

Figure 8.31: The final HTML after all code is inserted.

List of Figures

List of Tables

Bibliography

[07] *Yahoo! User Interface Blog*. [Online; accessed 17. Nov. 2022]. Jan. 2007. URL: https://web.archive.org/web/20070713072219/http://yuiblog.com/blog/2007/01/04/performance-research-part-2.

[15] *domInteractive: is it? really? | High Performance Web Sites*. [Online; accessed 18. Dec. 2022]. Aug. 2015. URL: https://www.stevesouders.com/blog/2015/08/07/dominteractive-is-it-really.

[16] *H74: Ensuring that opening and closing tags are used according to specification | Techniques for WCAG 2.0*. [Online; accessed 17. Nov. 2022]. Oct. 2016. URL: https://www.w3.org/TR/WCAG20-TECHS/H74.html.

[17] *Ad blocking in the Internet*. [Online; accessed 17. Nov. 2022]. Jan. 2017. URL: https://www.onaudience.com/files/adblock_report.pdf.

[18] *The gzip home page*. [Online; accessed 19. Oct. 2022]. Aug. 2018. URL: https://www.gzip.org.

[19a] *Introduction to HTTP/2*. [Online; accessed 22. Nov. 2022]. Sept. 2019. URL: https://web.dev/performance-http2/#request-and-response-multiplexing.

[19b] *Introduction to HTTP/2*. [Online; accessed 9. Nov. 2022]. Sept. 2019. URL: https://web.dev/performance-http2.

[20] *Eliminate render-blocking resources*. [Online; accessed 18. Nov. 2022]. Oct. 2020. URL: https://web.dev/render-blocking-resources.

[21] *First Contentful Paint (FCP)*. [Online; accessed 25. Feb. 2022]. Jan. 2021. URL: https://web.dev/fcp.

[22a] *@babel/plugin-transform-react-jsx-compat · Babel*. [Online; accessed 10. Nov. 2022]. Nov. 2022. URL: https://babeljs.io/docs/en/babel-plugin-transform-react-jsx-compat.

[22b] *249132 - Remove support for multipart/x-mixed-replace main resources - chromium*. [Online; accessed 10. Sep. 2022]. Sept. 2022. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=249132.

[22c] *275955 - Limit of 6 concurrent EventSource connections is too low - chromium*. [Online; accessed 10. Sep. 2022]. Sept. 2022. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=275955.

[22d] *906896 - Increase number of permitted EventSource connections*. [Online; accessed 10. Sep. 2022]. Sept. 2022. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=906896.

[22e] *Advanced Compilation | Closure Compiler | Google Developers*. [Online; accessed 9. Nov. 2022]. Mar. 2022. URL: https://developers.google.com/closure/compiler/docs/api-tutorial3.

[22f] *An image format for the Web - WebP - Google Developers*. [Online; accessed 19. Oct. 2022]. Aug. 2022. URL: https://developers.google.com/speed/webp.

[22g] *Angular*. [Online; accessed 9. Nov. 2022]. Mar. 2022. URL: https://angular.io.

[22h] *Browser Market Share Worldwide | Statcounter Global Stats*. [Online; accessed 17. Nov. 2022]. Nov. 2022. URL: https://gs.statcounter.com/browser-market-share.

[22i] *Critical rendering path - Web Performance | MDN*. [Online; accessed 25. Feb. 2022]. Feb. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/Performance/Critical_rendering_path.

[22j] *css.* [Online; accessed 6. Nov. 2022]. Nov. 2022. URL: `https://www.npmjs.com/package/css`.

[22k] *css-mediaquery.* [Online; accessed 22. Dec. 2022]. Dec. 2022. URL: `https://www.npmjs.com/package/css-mediaquery`.

[22l] *Cumulative Layout Shift (CLS).* [Online; accessed 25. Feb. 2022]. Feb. 2022. URL: `https://web.dev/cls`.

[22m] *Debugging HTML - Learn web development | MDN.* [Online; accessed 17. Nov. 2022]. Sept. 2022. URL: `https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/Debugging_HTML#Different_browsers_interpret_invalid_HTML_differently`.

[22n] *Documentation - More on Functions.* [Online; accessed 10. Nov. 2022]. Nov. 2022. URL: `https://www.typescriptlang.org/docs/handbook/2/functions.html`.

[22o] *eval() - JavaScript | MDN.* [Online; accessed 3. Oct. 2022]. Oct. 2022. URL: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`.

[22p] *Express - Node.js web application framework.* [Online; accessed 15. Nov. 2022]. Sept. 2022. URL: `http://expressjs.com`.

[22q] *Framework reimagined for the edge! - Qwik.* [Online; accessed 26. Oct. 2022]. Oct. 2022. URL: `https://qwik.builder.io`.

[22r] *HTML Standard.* [Online; accessed 17. Nov. 2022]. Nov. 2022. URL: `https://html.spec.whatwg.org/multipage/syntax.html#an-introduction-to-error-handling-and-strange-cases-in-the-parser`.

[22s] *HTTP Archive: Page Weight.* [Online; accessed 30. Nov. 2022]. Nov. 2022. URL: `https://httparchive.org/reports/page-weight`.

[22t] *HTTP/1 vs HTTP/2 What is the xn–Difference-8i3g.* [Online; accessed 4. Nov. 2022]. Nov. 2022. URL: `https://www.wallarm.com/what/what-is-http-2-and-how-is-it-different-from-http-1`.

[22u] *Introduction - Partytown.* [Online; accessed 26. Oct. 2022]. Oct. 2022. URL: `https://partytown.builder.io`.

[22v] *Largest Contentful Paint (LCP).* [Online; accessed 18. Oct. 2022]. Aug. 2022. URL: `https://web.dev/lcp`.

[22w] *Learn to style HTML using CSS - Learn web development | MDN.* [Online; accessed 25. Feb. 2022]. Feb. 2022. URL: `https://developer.mozilla.org/en-US/docs/Learn/CSS`.

[22x] *Live streaming web audio and video - Developer guides | MDN.* [Online; accessed 30. Nov. 2022]. Oct. 2022. URL: `https://developer.mozilla.org/en-US/docs/Web/Guide/Audio_and_video_delivery/Live_streaming_web_audio_and_video`.

[22y] *Marko.* [Online; accessed 9. Nov. 2022]. Sept. 2022. URL: `https://markojs.com`.

[22z] *Media Delivery Solutions - Content Delivery Network Solutions | Akamai.* [Online; accessed 30. Nov. 2022]. Nov. 2022. URL: `https://www.akamai.com/solutions/content-delivery-network/media-delivery`.

[22aa] *Mobile-first Indexing Best Practices | Google Search Central | Google Developers.* [Online; accessed 31. Oct. 2022]. Sept. 2022. URL: `https://developers.google.com/search/mobile-sites/mobile-first-indexing`.

[22ab] *Most used web frameworks among developers 2022 | Statista.* [Online; accessed 5. Sep. 2022]. Sept. 2022. URL: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web.

[22ac] *Mutation Observer | Can I use...* [Online; accessed 10. Nov. 2022]. Nov. 2022. URL: https://caniuse.com/mutationobserver.

[22ad] *Neos CMS - das Open Source Content Application Framework.* [Online; accessed 18. Dec. 2022]. Dec. 2022. URL: https://www.neos.io/de.

[22ae] *Network Link Conditioner.* [Online; accessed 18. Dec. 2022]. Dec. 2022. URL: https://developer.apple.com/download/more/?q=Additional%20Tools.

[22af] *PageSpeed Insights.* [Online; accessed 25. Feb. 2022]. Feb. 2022. URL: https://pagespeed.web.dev.

[22ag] *Performance - Web APIs | MDN.* [Online; accessed 15. Nov. 2022]. Nov. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance.

[22ah] *puppeteer.* [Online; accessed 6. Nov. 2022]. Nov. 2022. URL: https://www.npmjs.com/package/puppeteer.

[22ai] *React – A JavaScript library for building user interfaces.* [Online; accessed 9. Nov. 2022]. Nov. 2022. URL: https://reactjs.org.

[22aj] *Remove unused code.* [Online; accessed 7. Dec. 2022]. Dec. 2022. URL: https://web.dev/remove-unused-code.

[22ak] *RFC 6455 - The WebSocket Protocol.* [Online; accessed 17. Nov. 2022]. Nov. 2022. URL: https://datatracker.ietf.org/doc/html/rfc6455.

[22al] *RFC9113.* [Online; accessed 25. Oct. 2022]. Aug. 2022. URL: https://httpwg.org/specs/rfc9113.html#StreamsLayer.

[22am] *RFC9114.* [Online; accessed 25. Oct. 2022]. Aug. 2022. URL: https://httpwg.org/specs/rfc9114.html#stream-mapping.

[22an] *Server-sent events - Can I use... Support tables for HTML5, CSS3, etc.* [Online; accessed 18. Oct. 2022]. Oct. 2022. URL: https://caniuse.com/eventsource.

[22ao] *Server-Side Rendering (SSR) | Vue.js.* [Online; accessed 10. Nov. 2022]. Nov. 2022. URL: https://vuejs.org/guide/scaling-up/ssr.html#client-hydration.

[22ap] *Subsetting - Fonts Knowledge - Google Fonts.* [Online; accessed 19. Oct. 2022]. Oct. 2022. URL: https://fonts.google.com/knowledge/glossary/subsetting.

[22aq] *The 2 main performance debts of HTTP/1.* [Online; accessed 22. Nov. 2022]. Oct. 2022. URL: https://www.erwinhofman.com/blog/two-main-performance-debts-of-http1/#bundling-javascript.

[22ar] *The 2 main performance debts of HTTP/1.* [Online; accessed 4. Nov. 2022]. Nov. 2022. URL: https://www.erwinhofman.com/blog/two-main-performance-debts-of-http1.

[22as] *The Ultimate Guide to Optimizing JavaScript for Quick Page Loads.* [Online; accessed 9. Nov. 2022]. Nov. 2022. URL: https://www.builder.io/blog/the-ultimate-guide-to-optimizing-javascript-for-quick-page-loads.

[22at] *Tree shaking - MDN Web Docs Glossary: Definitions of Web-related terms | MDN.* [Online; accessed 22. Dec. 2022]. Sept. 2022. URL: https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking.

[22au] *Troubleshooting HTTP Streams | Marko.* [Online; accessed 9. Nov. 2022]. Sept. 2022. URL: https://markojs.com/docs/troubleshooting-streaming.

[22av] *Vue.js - The Progressive JavaScript Framework | Vue.js.* [Online; accessed 9. Nov. 2022]. Nov. 2022. URL: `https://vuejs.org`.

[22aw] *WebSocket API | Can I use... Support tables for HTML5, CSS3, etc.* [Online; accessed 17. Nov. 2022]. Nov. 2022. URL: `https://caniuse.com/mdn-api_websocket`.

[22ax] *Websocket CDN | Content Delivery Network to Improve Communication.* [Online; accessed 30. Nov. 2022]. Sept. 2022. URL: `https://www.belugacdn.com/websocket-cdn`.

[23a] *Critical CSS.* [Online; accessed 21. Jan. 2023]. Jan. 2023. URL: `https://www.npmjs.com/package/critical-css`.

[23b] *penthouse.* [Online; accessed 21. Jan. 2023]. Jan. 2023. URL: `https://www.npmjs.com/package/penthouse`.

[23c] *Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.* [Online; accessed 6. Jan. 2023]. Jan. 2023. URL: `https://tailwindcss.com`.

[97] *Analysis of HTTP Performance Problems.* [Online; accessed 22. Nov. 2022]. Feb. 1997. URL: `https://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html`.

[add22] addyosmani. *critical.* [Online; accessed 20. Oct. 2022]. Oct. 2022. URL: `https://github.com/addyosmani/critical`.

[Air13] AirbnbEng. *Isomorphic JavaScript: The Future of Web Apps.* 2013. URL: `https://medium.com/airbnb-engineering/isomorphic-javascript-the-future-of-web-apps-10882b7a2ebc` (visited on 01/28/2021).

[API15] W3C: The WebSocket API. *Web sockets.* 2015. URL: `https://www.w3.org/TR/websockets/` (visited on 10/24/2015).

[bab22] babeljs.io. *Babel - The compiler for next generation JavaScript.* [Online; accessed 10. Jan. 2022]. Jan. 2022. URL: `https://babeljs.io`.

[Bov15] Andreas Bovens. *Opera Browsers, Modes and Engines.* 2015. URL: `https://dev.opera.com/articles/browsers-modes-engines/` (visited on 02/08/2021).

[BPT15a] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2).* [Online; accessed 9. Nov. 2022]. May 2015. DOI: `10.17487/RFC7540`.

[BPT15b] Mike Belshe, R Peon, and ME Thomson. “RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2)”. In: *Hypertext transfer protocol version* 2 (2015), p. 46.

[CH20] Xavier Chamberland-Thibeault and Sylvain Hallé. “Structural Profiling of Web Sites in the Wild”. In: *International Conference on Web Engineering.* Springer. 2020, pp. 27–34.

[Cha+20] Moumena Chaqfeh et al. “JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup”. In: *Proceedings of The Web Conference 2020.* 2020, pp. 763–773.

[Cha+21] Moumena Chaqfeh et al. “To Block or Not to Block: Accelerating Mobile Web Pages On-The-Fly Through JavaScript Classification”. In: *arXiv preprint arXiv: 2106.13764* (2021).

[Clo20] CloudMosa. *Puffin OS - Next-generation technology to bridge the digital divide.* 2020. URL: `https://www.puffin.com/os/` (visited on 01/28/2021).

[dev22] developer.mozilla.org. *title - HTML: HyperText Markup Language MDN.* [Online; accessed 31. Jan. 2022]. Jan. 2022. URL: `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/title`.

[Dha18] Vijay Dharap. “Webpack Bundle Analysis — A necessary step for all React, Angular, Vue application developers!” In: *Medium* (June 2018). URL: `https://medium.com/hackernoon/webpack-bundle-analysis-a-necessary-step-for-all-react-angular-vue-application-developers-fe6564fa62ca`.

[DS13] Alexis Deveria and L Schoors. *Can I use web sockets.* 2013. URL: `http://caniuse.com/websockets` (visited on 06/04/2013).

[Edu22] Nate Cavanaugh Eduardo Lundgren. *AlloyUI.* [Online; accessed 9. Nov. 2022]. Jan. 2022. URL: `https://alloyui.com`.

[Fie99] Roy Fielding. “Hypertext transfer protocol-HTTP/1.1. IETF RFC 2616”. In: *http://www. ietf. org/rfc/rfc2616. txt* (1999).

[Gal20] Andrew Galloni. “HTTP/2 progressive image streaming”. In: *Cloudflare Blog* (Aug. 2020). URL: `https://blog.cloudflare.com/parallel-streaming-of-progressive-images`.

[Goo17] Google. *Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed.* [Online; accessed 21. Jan. 2023]. Feb. 2017. URL: `https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf`.

[Goo20a] Google. *Closure Compiler - Google Developers.* [Online; accessed 15. Jan. 2022]. Aug. 2020. URL: `https://developers.google.com/closure/compiler`.

[Goo20b] Google. *Closure Compiler Compilation Levels | Google Developers.* [Online; accessed 15. Jan. 2022]. Aug. 2020. URL: `https://developers.google.com/closure/compiler/docs/compilation_levels`.

[Gri22] Ilya Grigorik. *Measuring the Critical Rendering Path.* [Online; accessed 8. Aug. 2022]. Aug. 2022. URL: `https://web.dev/critical-rendering-path-measure-crp`.

[GS20] Utkarsh Goel and Moritz Steiner. “System to Identify and Elide Superfluous JavaScript Code for Faster Webpage Loads”. In: *arXiv preprint arXiv:2003.07396* (2020).

[Hak16] Malek Hakim. “Speed index and critical path rendering performance for isomorphic single page applications”. In: *The 16th Winona Computer Science Undergraduate Research Symposium.* 2016, p. 41.

[Hil20] Raymond Hill. *uBlock Origin.* 2020. URL: `https://github.com/gorhill/uBlock/` (visited on 2020).

[Hot21] Hotwired. *Turbo.* 2021. URL: `https://github.com/hotwired/turbo` (visited on 01/28/2021).

[HTT21a] HTTP Archive. *HTTP Archive.* [Online; accessed 5. Oct. 2021]. July 2021. URL: `https://httparchive.org/reports/state-of-the-web`.

[HTT21b] HTTP Archive. *HTTP Archive.* [Online; accessed 5. Oct. 2021]. July 2021. URL: `https://httparchive.org/reports/loading-speed`.

[Isk+20] Taufan Fadhilah Iskandar et al. “Comparison between client-side and server-side rendering in the web development”. In: *IOP Conference Series: Materials Science and Engineering.* Vol. 801. 1. IOP Publishing. 2020, p. 012136.

[Jan+22] Kalle Janssen et al. “On the Impact of the Critical CSS Technique on the Performance and Energy Consumption of Mobile Browsers”. In: *Proceedings of the International Conference on Evaluation and Assessment on Software Engineering (EASE).* 2022.

[JS20] Nuxt JS. *Nuxt JS - The Intuitive VUE Framework.* 2020. URL: `https://nuxtjs.org/` (visited on 09/10/2020).

[JZ16] Gorjan Jovanovski and Vadim Zaytsev. "Critical CSS Rules—Decreasing time to first render by inlining CSS rules for over-the-fold elements". In: *Postproceedings of 2016 Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE).* 2016, pp. 353–356.

[Kap19] Sahil Kapoor. "Puffin OS Is The Mobile Operating System That Can Crush Android Go". In: *iGyaan Network* (July 2019). URL: `https://www.igyaan.in/191408/puffin-os-features-crowdfunding`.

[Kir16] David Kirkpatrick. *Google: 53 Percent of mobile users abandon sites that take over 3 seconds to load.* 2016. URL: `https://www.marketingdive.com/news/google-53-of-mobile-users-abandon-sites-that-take-over-3-seconds-to-load/426070/` (visited on 09/12/2016).

[KL00] R Khare and S Lawrence. *RFC 2817: Upgrading to TLS within HTTP/1.1, 2000.* 2000.

[KL07] Ron Kohavi and Roger Longbotham. "Online experiments: Lessons learned". In: *Computer* 40.9 (2007), pp. 103–105.

[KL10] Nikhil Kothari and Bertrand Le Roy. *Initial server-side content rendering for client-script web pages.* US Patent 7,814,410. Oct. 2010.

[Kup+21] Tofunmi Kupoluyi et al. "Muzeel: A Dynamic JavaScript Analyzer for Dead Code Elimination in Today's Web". In: *arXiv preprint arXiv:2106.08948* (2021).

[Laj19] Peep Laja. *11 Low-Hanging Fruits for Increasing Website Speed (and Conversions).* 2019. URL: `https://conversionxl.com/blog/11-low-hanging-fruits-for-increasing-website-speed-and-conversions/` (visited on 08/07/2019).

[Lin06] Greg Linden. *Marissa Mayer at Web 2.0.* 2006. URL: `http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html` (visited on 11/06/2006).

[Maz17] Davood Mazinanian. "Eliminating code duplication in cascading style sheets". PhD thesis. Concordia University, 2017.

[McK19] Brian McKelvey. *WebSocket Client and Server Implementation for Node.* 2019. URL: `https://www.npmjs.com/package/websocket` (visited on 12/06/2019).

[Mee21] Patrick Meenan. *WebPageTest - Website Performance and Optimization Test.* [Online; accessed 5. Oct. 2021]. Oct. 2021. URL: `https://webpagetest.org`.

[MF11] Alexey Melnikov and Ian Fette. *The WebSocket Protocol.* RFC 6455. Dec. 2011. DOI: `10.17487/RFC6455`. URL: `https://www.rfc-editor.org/info/rfc6455`.

[Mic10] James Mickens. "Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads." In: *WebApps.* 2010.

[Mod22] Modernizr. *Modernizr.* [Online; accessed 10. Sep. 2022]. Sept. 2022. URL: `https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills`.

[MT13] Morten Moshagen and Meinald Thielsch. "A short version of the visual aesthetics of websites inventory". In: *Behaviour & Information Technology* 32.12 (2013), pp. 1305–1311.

[Müh19] Adrian Mühlroth. *Ist dieses neue Betriebssystem besser als Android?* 2019. URL: `https://www.techbook.de/mobile/puffin-os` (visited on 2019).

[Müh20] Adrian Mühlroth. "Ist dieses neue Betriebssystem besser als Android?" In: *TECHBOOK* (Jan. 2020). URL: `https://www.techbook.de/news/puffin-os`.

[Nah04] Fiona Fui-Hoon Nah. “A study on tolerable waiting time: how long are web users willing to wait?” In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163.

[Nat+17] Harini Natarajan et al. “Improving a website’s first meaningful paint by optimizing render blocking resources-An experimental case study”. In: (2017).

[Net+16] Ravi Netravali et al. “Polaris: Faster page loads using fine-grained dependency tracking”. In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16).* 2016.

[Obb+18] Niels Groot Obbink et al. “An extensible approach for taming the challenges of JavaScript dead code elimination”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE. 2018, pp. 291–401.

[Ope21] Opera. *Opera Mini.* 2021. URL: `https://www.opera.com/de/mobile/mini` (visited on 01/28/2021).

[Opt08] Vebsite Optimization. *The Psychology of Web Performance.* 2008. URL: `http://www.websiteoptimization.com/speed/tweak/psychology-web-performance/` (visited on 05/30/2008).

[OS20] Puffin OS. *Puffin OS - Next-generation technology to bridge the digital divide.* 2020. URL: `https://www.puffin.com/os/` (visited on 07/06/2020).

[Osm21] Addy Osmani. *critical.* 2021. URL: `https://www.npmjs.com/package/critical` (visited on 01/29/2021).

[Pag20] Google PageSpeed. *PageSpeed Documentation.* 2020. URL: `modpagespeed.com/doc/` (visited on 09/20/2020).

[PD17] Guy Podjarny and Christopher R Dumoulin. *Progressive consolidation of web page resources.* US Patent 9,785,621. Oct. 2017.

[Poc+18] Victor Le Pochat et al. “Tranco: A research-oriented top sites ranking hardened against manipulation”. In: *arXiv preprint arXiv:1806.01156* (2018).

[Pol19] Barry Pollard. *HTTP/2 in Action.* Manning, 2019. URL: `https://freecontent.manning.com/mental-model-graphic-how-is-http-1-1-different-from-http-2/` (visited on 08/06/2020).

[QL20] Gao Qiong and Wenmin Li. “An Optimization Method of Javascript Redundant Code Elimination based On Hybrid Analysis Technique”. In: *2020 17th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP).* IEEE. 2020, pp. 300–305.

[Rea21] React. *React - A JavaScript library for building user interfaces.* 2021. URL: `https://reactjs.org` (visited on 01/28/2021).

[Res00] Eric Rescorla. “HTTP Over TLS (RFC 2818)”. In: *Internet Engineering Task Force* (2000).

[RN17] Rashmi Rashmi and Harini Natarajan. *Master Thesis.* 2017.

[Rua+17] Vaspol Ruamviboonsuk et al. “Vroom: Accelerating the mobile web with server-aided dependency resolution”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* 2017, pp. 390–403.

[Sch07] Scott Schiller. *Progressive loading.* US Patent App. 11/364,992. Aug. 2007.

[sec21] section.io. *Comparison between the HTTP/3 and HTTP/2 Protocols.* [Online; accessed 22. Nov. 2021]. 2021. URL: `https://www.section.io/engineering-education/http3-vs-http2`.

[Ser22] Caddy Web Server. *Caddy 2 - The Ultimate Server with Automatic HTTPS.* [Online; accessed 19. Nov. 2022]. Nov. 2022. URL: `https://caddyserver.com`.

[SG+16] Naveen Kumar SG et al. "A Hybrid Web Rendering Framework on Cloud". In: *2016 IEEE International Conference on Web Services (ICWS).* IEEE. 2016, pp. 602–608.

[SHB17] Stanislav Shalunov, Gregory Hazel, and Micha Benoliel. *System and method for improving webpage loading speeds.* US Patent App. 14/758,961. Jan. 2017.

[Shi11] Anand Lal Shimpi. *Amazon's Silk Browser Acceleration Tested: Less Bandwidth Consumed, But Slower Performance.* 2011. URL: `https://www.anandtech.com/show/5139/amazons-silk-browser-tested-less-bandwidth-consumed-but-slower-performance` (visited on 07/06/2020).

[SI06] Scott Schiller and Yahoo Inc. *Progressive loading.* [Online; accessed 26. Oct. 2022]. Feb. 2006. URL: `https://patents.google.com/patent/US20070186182A1/en`.

[Sin+16] Harvinder P Singh et al. *Progressive page loading.* US Patent 9,235,559. Dec. 2016.

[Sop22] Szymon Soppa. "Async vs Defer - Which Script Attribute is More Efficient When Loading JavaScript?" In: *Curiosum* (Feb. 2022). URL: `https://curiosum.com/blog/seo-speed-script-tags-async-vs-defer`.

[Sta21] Statista. *Most used web frameworks among developers 2021 | Statista.* [Online; accessed 15. Oct. 2021]. Oct. 2021. URL: `https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web`.

[Ver21] Next.js by Vercel. *The React Framework for Production.* 2021. URL: `https://nextjs.org/` (visited on 01/28/2021).

[Vog18] Lucas Vogel. *Bachelorarbeit: Entwicklung und Evaluation eines Frameworks für Adaptive Web Anwendungen.* TU Dresden, 2018.

[VS22a] Lucas Vogel and Thomas Springer. "An In-Depth Analysis of Web Page Structure and Efficiency with Focus on Optimization Potential for Initial Page Load". In: *International Conference on Web Engineering.* Springer. 2022, pp. 101–116.

[VS22b] Lucas Vogel and Thomas Springer. "User Acceptance of Modified Web Page Loading Based on Progressive Streaming". In: *International Conference on Web Engineering.* Springer. 2022, pp. 391–405.

[VS23a] Lucas Vogel and Thomas Springer. "How Streaming Can Improve the World (Wide Web)". In: *Companion Proceedings of the ACM Web Conference 2023.* 2023, pp. 140–143.

[VS23b] Lucas Vogel and Thomas Springer. "Speed Up the Web with Universal CSS Rendering". In: *International Conference on Web Engineering.* Springer. 2023, pp. 191–205.

[VS23c] Lucas Vogel and Thomas Springer. "Waiter and AUTRATAC: Don't Throw It Away, Just Delay!" In: *International Conference on Web Engineering.* Springer. 2023, pp. 278–292.

[Vse22] Vsevolod Strukchinsky, Vladimir Starkov and contributors. *BEM — Block Element Modifier.* [Online; accessed 9. Nov. 2022]. Sept. 2022. URL: `https://getbem.com`.

[W3C19] W3C. *W3C Standards.* 2019. URL: `https://www.w3.org/standards/` (visited on 2019).

[W3C20] W3C. *World Wide Web Consortium (W3C)*. 2020. URL: `https://www.w3.org/` (visited on 01/01/2020).

[w3o19] w3.org. *W3C HTML*. [Online; accessed 31. Jan. 2022]. Nov. 2019. URL: `https://www.w3.org/html`.

[w3t22] w3techs.com. *Usage Statistics of JavaScript as Client-side Programming Language on Websites, February 2022*. [Online; accessed 1. Feb. 2022]. Feb. 2022. URL: `https://w3techs.com/technologies/details/cp-javascript`.

[Wap21] Wappalyzer. *Find out what websites are built with - Wappalyzer*. [Online; accessed 14. Oct. 2021]. Oct. 2021. URL: `https://www.wappalyzer.com`.

[web21] webpack. *webpack*. 2021. URL: `https://webpack.js.org/` (visited on 01/28/2021).

[wha22] whatwg.org. *HTML Standard - body ok*. [Online; accessed 31. Jan. 2022]. Jan. 2022. URL: `https://html.spec.whatwg.org/multipage/links.html#body-ok`.