Mega-fast or just super-fast? Performance differences of mainstream JavaScript frameworks for web applications

Andreas Nicklaus
Hochschule der Medien Stuttgart
an067@hdm-stuttgart.de

Prof. Dr. Fridtjof Toenniessen & Stephan Soller

Abstract

An essential initial step in every modern web application project is the selection of an appropriate web development framework. Often, detrimental decisions are made based on sentiment rather than a proper assessment of the framework's performance vs. the project requirements and resources.

This thesis presents a study of a model web application created identically with seven mainstream JavaScript web development frameworks: Angular, Astro, Next.js, Nuxt, React, Svelte and Vue.js.

Performance measurements are done with Lighthouse and Playwright tools to identify strengths and weaknesses of the frameworks. To this end, classic page load times and the load and update times of JavaScript components are retrieved among other data. Additionally, two new suitable derivative metrics are evaluated: the "Observed Visual Change Duration" and a "loadEventEnd" metric.

The results show no clear-cut general advantage of a single web development framework. Component update times indicate Nuxt as the fastest web development framework. Next.js is the slowest one in this context. Similarly, Google Chrome appears to be the fastest client browser. Desktop Safari is the slowest one for updating the DOM after user input.

1 Introduction

With the evolution of the world wide web, development of websites reached a higher complexity, both of the page content and the functionality. This complexity resulted in higher demand for technical sophistication in networking, hosting services and development tools. Although modern frameworks provide technical advancements to increase the speed of page and content generation and arguably a better developer experience, there is no apparent way to objectively determine a "best framework" in terms of development ease and speed.

When it comes to user experience and perceived performance however, there are plentiful collections of metrics and criteria to choose from so as to determine the performance of websites, not frameworks. The strife always is to somehow better the web site performance since it has a palpable influence on search engine results, user acceptance and ultimately project success. Hence, there are business interests and financial incentives to invest resources into performance optimization (Li et al., 2010; Zhou et al., 2013). Past research, existing tools and guides give direction to optimize websites according to stakeholders' and users' expectations. Yet in most cases, the only focus on specific websites or specific frameworks or give general advice.

The lack of research on the effect of the framework on website performance indicates a need for research on the topic. Relying on marketing material for the choice of framework is questionable because most modern frameworks claim to be fast, easy to use and performance efficient. This suggests that each would be a great choice. Thus, comparing frameworks presents a challenge because no ideal set of metrics for this use case is apparent and there are no publicly accessible replicas of web applications built with different frameworks available. Therefore, a comparative study between versions of the same website built with different web development frameworks is needed. With this data, an informed choice might be made for projects in the future.

The goals of this thesis are to propose a set of metrics that allow comparing mainstream JavaScript (JS) frameworks for web applications, to provide a comparative study between selected frameworks and to create a tool to compare the rendering performance a web page as a whole and of dynamic components within a page.

2 Application and testing environment

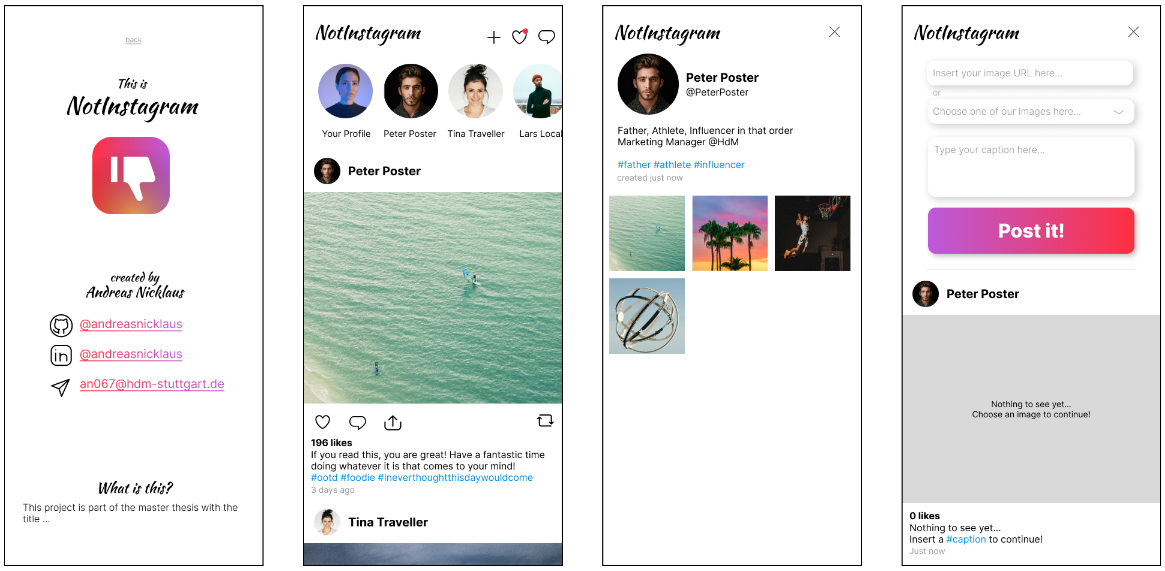
Frameworks. One of the choices for the setup of the study is which frameworks to implement the application in and compare. The selected frameworks have to support the designed web application without the help of another tool or framework unless intended by the developers of the framework. Plus, the frameworks have to use JavaScript (JS) in order to narrow down the scope of the study. TypeScript frameworks are allowed because they support JS (Bierman et al., 2014).

Basis for the framework selection are the rankings of most-used, most-liked and most-interesting web development frameworks and tools (Devographics, 2024). The following frameworks were selected for this framework:

- Angular
- Astro
- Next.js
- Nuxt
- React
- Svelte
- Vue.js

Others like Preact, Solid and Qwik were considered to be included in this study, but were dropped because of negative sentiment or low usage among developers that have experience with the tools.

Web application. The web application used for this study is designed to be the subject of comparisons between frameworks. Its look is derived from the Android app of Instagram (Instagram from Meta, 2024) and it has four pages (see figure 1). The four pages cover three generally valid page types identified in the design process. The About page is a "Static page" as it does not change its content after the initial response from the web server. No additional data query is needed to build the finished DOM structure. The Feed page and the Profile page are "Delayed pages". Their defining characteristic is that the DOM cannot be fully built from the initial HTML document, but needs data queries to complete before all content can be displayed. These data queries are triggered immediately after the initial page request. The Create page is the only "Dynamic page". Its initial features indicate it being either a static or delayed page, depending on the implementation, and it has dynamic components that update through user input. Mutations to the DOM are therefore not only triggered by the initial page request but a user interaction. The time of such changes is therefore not predictable.



This project is part of the master thesis with the title ...

If you read this, you are great! Have a fantastic time doing whatever it is that comes to your mind! #ootd #foodie #Ineverthoughtthisdaywouldcome
3 days ago

Father, Athlete, Influencer in that order
Marketing Manager @HdM
#father #athlete #influencer

Nothing to see yet...
Choose an image to continue!

Nothing to see yet...
Insert a #caption to continue!

(a) About page (b) Feed page (c) Profile page (d) Create page

Figure 1: Screenshots of the "NotInstagram" application's pages

Components. These four pages are comprised of 15 components, most of which are wrappers to encapsulate image components, styled text or iterations over lists with subcomponents. However, two components stand out because of their special purpose and implementation differences between frameworks.

1. The MediaComponent is designed to present both internal and external image and video sources in a single component. It is used to display Profile images and Post content. Its main purpose is to decide - based on the passed source string - how to project the multimedia file onto the DOM. As such, a decision for enhanced image or video elements had to be made per framework during the implementation of the application. Svelte, Astro, Next.js and Nuxt provide such an enhanced image component. In contrast, video elements are inserted to the DOM as-is, but the browser behaviour is adapted identically for all frameworks using attributes on the `<video>` element and JavaScript. In addition, the import of local images differs between frameworks because the load behaviour differs. As such, some frameworks require importing all local images in order to select the requested image.

2. Astro does not natively support dynamic components as needed in the Create page of the application. The intended solution is to implement so-called "Islands" using another framework. React is chosen for its high usage rate among web developers (Devographics, 2024). As a result, two implementations are compared in this study:

Using the React components that are needed for Astro Islands everywhere, even if the component in question is not dynamic, and creating duplicate native Astro components for when a component is not required to be dynamic. One additional React component "CreateForm" was created in order to encapsulate React subcomponents and six components were implemented in React because they are part of the form and the Post preview on the Create page.

Hosting. In order to test the end-products of the frameworks, at least one web server is needed to host the application. Network delays are an obvious source of rendering and performance issues (Grigorik, 2013). For this reason, the tests for this study are performed on two different web servers: An online hosting service and the local testing machine.

1. **Vercel** was chosen for hosting the applications on distant servers based on its popularity, capabilities for Server-side Rendering (SSR), support for both a free and paid version and its simple integration into CI/CD pipelines. Each Vercel project was connected to a Github repository, one per framework. Only required project configuration options were changed per project on the plattform to ensure its state as "as-is".

2. A **local host** was chosen to minimize the effect of network delay and related delays, e.g. domain name resolving, in this study. The application is hosted on the testing machine. A HP Envy x360 with an AMD Ryzen 5 5500U processor and 16 GB RAM is used here. The OS on the device is Windows 11 Home (Version 10.0.22631) during testing. The application is built before every test and hosted using either built-in commands for the framework or using the `serve` command.

Metrics. To identify strengths and weaknesses of the frameworks, eleven metrics were chosen to test the frameworks in three categories (see table 1). The Page Load Time (PLT) covers the classic load time of web pages and is specified to outline the load speed from `requestStart` to the last change to the page. The Component Load Time (CLT) is defined as the time frame in which any changes to the DOM with JS can be identified and the rendering process of JS components are shown. The Component Update Time (CUT) is defined as the time between a user interaction and a DOM mutation. This time frame describes the speed of feedback to the user that the interaction has been registered and something is happening as well as the speed until that something finishes happening. Especially DOM mutation times are expected to show differences between frameworks and implementations as the HTML elements and internal implementation change from one framework to another.

Test tools. The requirements for testing tools - created by hosting the application on two different web servers and by the list list of metrics - are fulfilled by the Lighthouse CLI and Playwright. They are set up so as to provide results both in human-readable and machine-readable format enabling easy debugging and automatic creation of aggregate metrics.

1. Using the **Lighthouse CLI**, a script for starting the web server and running Lighthouse tests on the web application is executed. These tests run 20 times and only cover the performance measurements of Lighthouse. Reports are created in both HTML and JSON format in order to debug the tests and create the mean average of every measurement.

	PLT	CLT	CUT
Total Byte Weight	x		
Time To First Byte	x		
Time To Interactive	x	x	
Total Blocking Time	x	x	
LoadEventEnd	x	x	
DomContentLoaded	x		
Last Visual Change	x		
Largest Contentful Paint	x		
Observed Last Visual Change		x	
Observed First Visual Change		x	
DOM Mutation Times		x	x

Table 1: Assignment of metrics to the metric categories: Page Load Time (PUT), Component Load Time (CLT) and Component Update Time (CUT)

2. Tests with **Playwright** focus on the measurement of DOM mutations and the adherence to time budgets. To that end, a JS script is injected into the browser context before tests. This recording script initializes a MutationObserver on a specific HTML element that is created by the framework. This way, all DOM mutations such as element addition, element removal and attribute change are recorded with an identifier of the element and the time of the mutation.

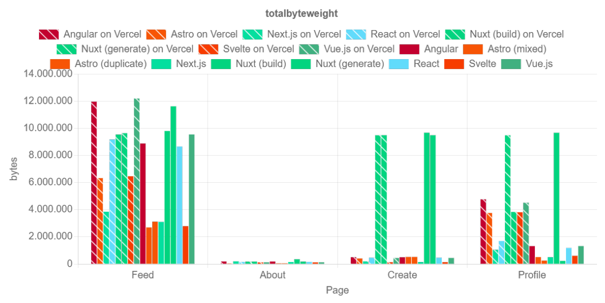
The respective metrics covered by Lighthouse and Playwright are seen in table 2.

Lighthouse	**Playwright**
Total Byte Weight (TBW)	domContentLoaded
Time To First Byte (TTFB)	loadEventEnd
Time To Interactive (TTI)	User Input Times
Total Blocking Time (TBT)	Mutation Times
Largest Contentful Paint (LCP)	
First Visual Change (FVC)	
Observed First Visual Change (OFVC)	
Observed Last Visual Change (OLVC)	

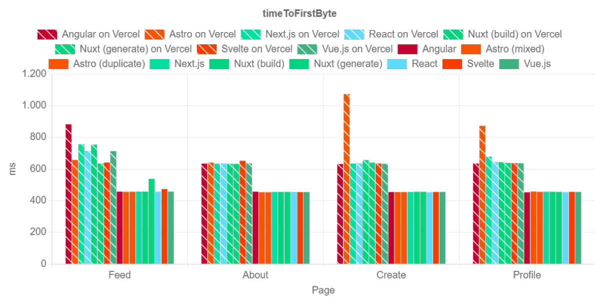
Table 2: Assignment of metrics to the test tools

3 Results

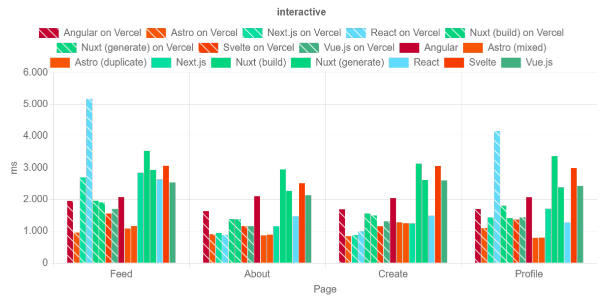
Metrics for the page load and for the component load times show no clear generally applicable evidence for a single framework being faster than the others. Such a distinction can only be made on a per-metric basis. Figure 2 presents the averages of measurements from the Lighthouse reports per page and framework.



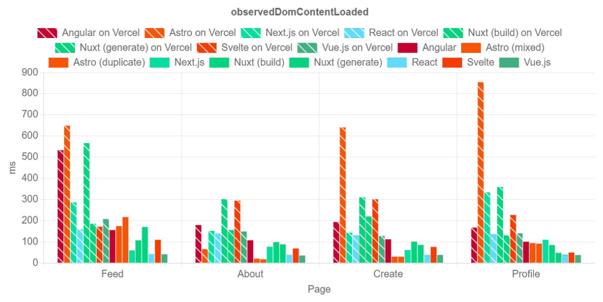
(a) Total Byte Weight (TBW)



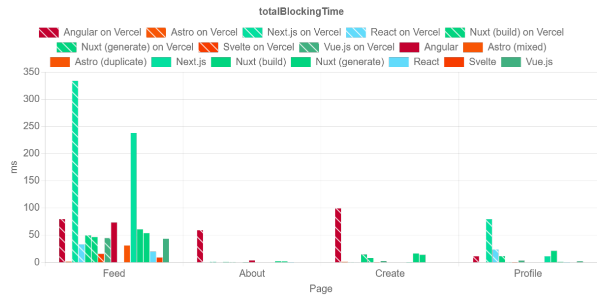
(b) Time To First Byte (TTFB)



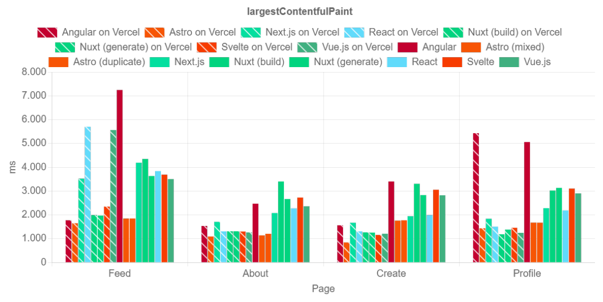
(c) Time To Interactive (TTI)



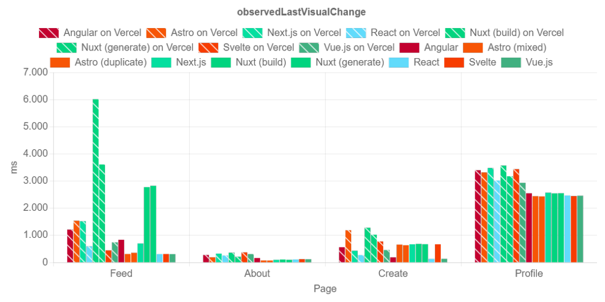
(d) Observed DomContentLoaded



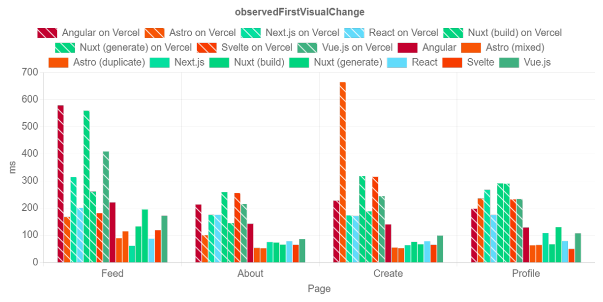
(e) Total Blocking Time (TBT)



(f) Largest Contentful Paint (LCP)



(g) Observed Last Visual Change (OLVC)



(h) Observed Last Visual Change (OLVC)

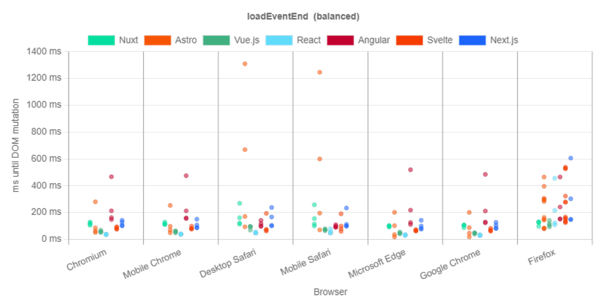
Figure 2: Lighthouse test results in Google Chrome

Page Load Time. Next.js, Astro and Svelte are the leading frameworks in TBW and Svelte, Next.js, Vue.js and especially Astro have fast results in their TTI. In addition, Astro, Angular, Svelte, Nuxt and Vue.js stand out through little fluctuations in TTI across the four pages and the test repetitions. The results of measurements for the TBT also favor Astro and Svelte. In contrast, Astro and Svelte perform poorly in DomContentLoaded and balanced LoadEventEnd (see figure 3a). These metrics are strengths of Vue.js, React and Nuxt. The balanced LoadEventEnd is the difference between LoadEventEnd and the requestStart (see equation 1). Vue.js and React are also the fastest frameworks in OLVC. The TTFB does not support a ranking of frameworks. Instead, it is more dependent on the page content and the host, which influence the results more than the framework. However, Astro, Next.js and Angular stand out through slow results in this metric. The balanced LoadEventEnd highlights Vue.js and React positively, but also demonstrates a high dependency on the browser.

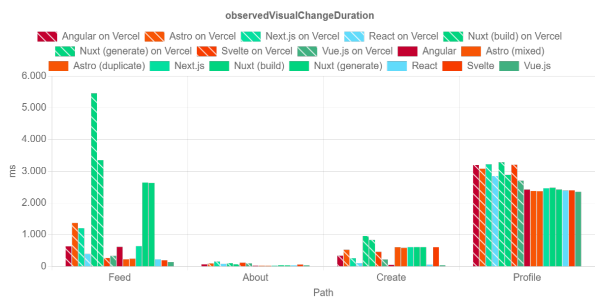
$$loadEventEnd_{balanced} = loadEventEnd_{raw} - requestStart \tag{1}$$

Component Load Time. The metrics for the Component Load Time have similar characteristics as with the Page Load Time. The OFVC of the applications are early in Astro, React and Next.js, which indicates a strength of React-based frameworks. React, Vue.js and Angular also naturally have a short Observed Visual Change Duration (OVCD) (see figure 3b), which is unsurprising. The OVCD is defined as the time difference between OFVC and OLVC (see equation 2). The recordings of early DOM mutations are also very fast for Astro, Vue.js and React, whereas recordings are missing completely for Angular (see figure 3c). This is most likely due to a faulty initialization of the MutationObserver that is responsible for recording mutation times.

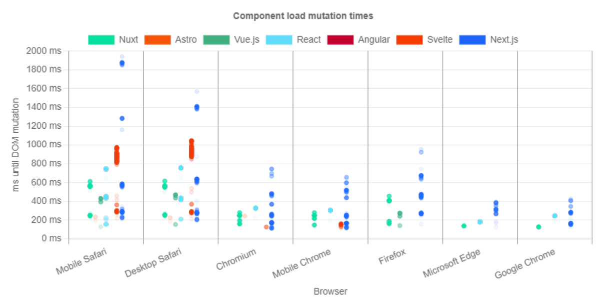
$$\begin{multline} observedVisualChangeDuration = \\ observedLastVisualChange - observedFirstVisualChange \end{multline} \tag{2}$$



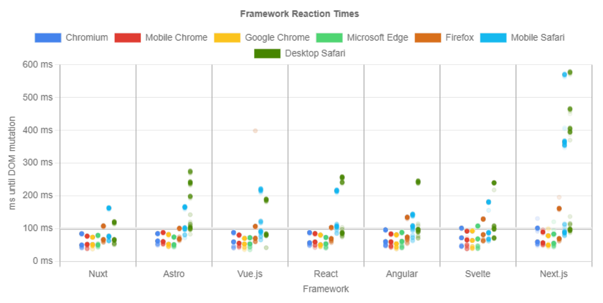
(a) Balanced loadEventEnd timings



(b) Observed Visual Change Duration (OVCD)



(c) Component load mutation times



(d) Recorded DOM mutation timings after user actions

Figure 3

Component Update Time. In contrast, the measurements made for the Component Update Time suggest clear rankings of the frameworks and of the used browsers (see figure 3d and table 3). The times of the DOM mutations are quite similar to each other except in Mobile Safari and Desktop Safari. In these browsers, Next.js is the slowest and Nuxt is the fastest framework. Across all pages and frameworks, the ranking of browsers from fastest to slowest is Google Chrome, Microsoft Edge, Chromium, Mobile Chrome, Firefox, Mobile Safari and Desktop Safari. This means that time budgets are most easily kept to in Google Chrome and hardest in Desktop Safari. The ranking of frameworks is - from fastest to slowest - Nuxt, Angular, Vue.js, React, Astro/Svelte and Next.js. In addition, Nuxt, Vue.js and Svelte are economical with DOM mutations after user interaction, whereas the other frameworks update the DOM after user interaction in more different ways. This ranking can influence the choice of framework for user input heavy applications. For this kind of application, Nuxt, Angular, Vue.js and React present themselves as the best choices relating to Component Update Time.

4 Summary

The results of the study are inconclusive in relation to load times for both pages and components. The measurements only show general advantages of single frameworks for the component update time.

Nuxt is the fastest framework in regards to component update time, whereas Next.js is the slowest. Likewise, Google Chrome turns out to be the fastest browser for component updates These updates are slowest in Desktop Safari.

However, test results fluctuate between repetitions. For this reason, future work should focus on making these results more reliable and statistically interpretable. Reliability might be achieved through repetition of the study with more test runs and the inclusion of other hosting environments. Additional pages might outline dependencies of the performance on the type of web page. Continuations of this study could also expand the user actions to other interactions than filling a form, for instance navigation between pages.

This study also revealed that the used algorithms for start and end of the recording are suboptimal for the goals. The start is delayed with periodical initialization attempts and the ending of the time frame for recording is manually set. For these reasons, early mutations (fast loading components), slow mutations (slow loading components) and periodically mutating components, e.g. a digital clock, cannot be recorded properly.

	Angular	Astro	Next.js	Nuxt	React	Svelte	Vue.js
Chromium	44	51	47	39	44	38	51
	69	**71**	**75**	**66**	**58**	**74**	**77**
	95	89	108	94	85	95	104
Firefox	54	63	59	59	54	60	52
	89	**99**	**142**	**83**	**84**	**94**	**82**
	123	142	235	108	181	129	103
Desktop Safari	77	87	79	51	84	70	47
	123	**170**	**304**	**86**	**169**	**164**	**136**
	172	270	493	124	280	283	200
Mobile Chrome	44	49	47	42	44	45	46
	67	**69**	**94**	**61**	**67**	**81**	**69**
	90	85	143	82	82	116	89
Mobile Safari	52	78	73	47	67	56	52
	106	**154**	**196**	**110**	**126**	**126**	**133**
	152	254	372	167	183	208	206
Microsoft Edge	43	44	46	37	41	40	40
	70	**64**	**73**	**61**	**62**	**74**	**61**
	90	80	134	85	75	102	79
Google Chrome	41	43	41	34	40	39	37
	62	**57**	**69**	**60**	**59**	**64**	**61**
	84	72	99	77	77	89	77
Browser Average	51	59	56	44	53	50	46
	84	**98**	**136**	**75**	**89**	**97**	**88**
	115	142	226	105	138	146	123
Weighted Browser Average	45	48	45	35	45	42	36
	69	**74**	**107**	**60**	**75**	**78**	**70**
	94	104	167	80	110	118	93

	Framework
Browser	minimum with framework in browser
	average with framework in browser
	maximum with framework in browser
Browser Average	average of minima across browsers
	total average across browsers
	average of maxima across browsers

Table 3: Minimum, average and maximum of recorded mutation times after user input in milliseconds (fastest times are highlighted green, slowest red). Weights are based on browser usage quota (StatCounter, 2024).

A Acronyms

CI/CD Continuous Integration and Continuous Delivery.

DOM Document Object Model.

FVC First Visual Change.

HTML Hypertext Markup Language.

JS JavaScript.

JSON JavaScript Object Notion.

LCP Largest Contentful Paint.

LVC Last Visual Change.

OFVC Observed First Visual Change.

OLVC Observed Last Visual Change.

OVCD Observed Visual Change Duration.

SSR Server-side Rendering.

TBT Total Blocking Time.

TBW Total Byte Weight.

TTFB Time To First Byte.

TTI Time To Interactive.

B References

Bierman, G., Abadi, M., and Torgersen, M. (2014). Understanding typescript. In Jones, R., editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg. Springer Berlin Heidelberg.

Devographics (2024). State of javascript 2023. `https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/`. accessed 07/29/2024.

Grigorik, I. (2013). *High Performance Browser Networking*. O'Reilly Media, Inc., 1005 Gravensetin Highwy North, Sebastopol, CA 95472.

Instagram from Meta (2024). Instagram. `https://www.instagram.com/`. accessed 08/02/2024.

Li, Z., Zhang, M., Zhu, Z., Chen, Y., Greenberg, A., and Wang, Y.-M. (2010). Webprophet: automating performance prediction for web services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 10, USA. USENIX Association.

StatCounter (2024). Quick start. `https://gs.statcounter.com/`. accessed 07/18/2024.

Zhou, M., Giyane, M., and Nyasha, M. (2013). Effects of web page contents on load time over the internet. *International Journal of Science and Research (IJSR)*, pages 2319–7064.

Github repository: All code and additional material can be found under `https://github.com/andreasnicklaus/master`.