

An Analysis On the Performance of Virtual Document Object Models

Lucas A. Gusmão¹

¹Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUCMG)
Rua Claudio Manoel 1.162, Funcionarios, Belo Horizonte, MG – Brasil

Abstract. *JavaScript frameworks such as React, Vue, and Angular have become popular in the web development community, especially for building single-page applications. React and Vue implement a concept known as virtual DOM (Document Object Model), which coordinates UI (User Interface) updates when the application’s state is changed in a fast and efficient manner. Svelte, another JavaScript framework, proposes a different model of directly updating the DOM, and it claims to be faster and more efficient than the virtual DOM approach. This paper compares the efficiency of Svelte’s approach and the virtual DOM to determine which one offers better performance and efficiency. The results obtained suggest that both Vue and Svelte performed well in most of the metrics analyzed, as opposed to React, which offered worse performance in almost all metrics. This indicates that the virtual DOM architecture can provide good performance depending on its framework’s implementation, as exemplified by the results yielded by Vue in this study.*

Keywords: JavaScript frameworks, user experience, web performance, Document Object Model

Bachelor’s in Software Engineering - PUC Minas

Trabalho de Conclusão de Curso (TCC) - Undergraduate Thesis

Content advisor (TCC I): Cleiton Tavares - cleitontavares@pucminas.br

Content advisor (TCC I): Simone de Assis - simone@pucminas.br

Academic advisor (TCC I): Laerte Xavier - laertexavier@pucminas.br

Advisor (TCC II): Hugo Bastos de Paula - hugo@pucminas.br

Belo Horizonte, May 14, 2023.

1. Introduction

With the evolution of the web platform, JavaScript front-end frameworks that aid in developing the UI (User Interface) of applications have become widely used by developers [Greif and Benitte 2020]. The main objective of JavaScript front-end UI frameworks is to simplify the development of web UI components, such as state management, data binding, and lifecycle methods/functions [Novac et al. 2021a, Novac et al. 2021b]. Angular, React, and Vue are the most widely used frameworks for web development in most parts of the world [Novac et al. 2021a]. One common concept in two of the three aforementioned JavaScript front-end frameworks is the virtual DOM (Document Object Model). The virtual DOM acts as a “virtual” representation of the application’s UI in memory,

which keeps it in sync with its state, by diffing the actual DOM of the website with its virtual counterpart [Javeed 2019].

Svelte - a JavaScript UI framework - has presented itself as an alternative to mainstream libraries like React and Vue with the promise of providing a component toolchain that “surgically” updates the DOM whenever it is needed instead of keeping a virtual representation of it [Harris 2018]. The framework has gathered the community’s attention, being ranked first in satisfaction and interest, while still being low in usage compared to its alternatives [Greif and Benitte 2020]. Some articles have made comparisons between JavaScript front-end frameworks like React, Angular, Vue - and even Svelte itself - in aspects such as performance and developer experience [Novac et al. 2021a, Novac et al. 2021b, Tripon et al. 2021], but **no research has been done to see if Svelte’s way of dynamically updating the DOM offers better performance when compared to the virtual DOM approach used by frameworks like React and Vue** [Novac et al. 2021a, Novac et al. 2021b, Tripon et al. 2021].

UX (User Experience) has been a growing concern for companies and developers [Sam Meder and Chang 2017, Palmer 2016, Osmani 2018, Friedman 2014]. Performance plays a key role in UX and can also greatly impact users’ website engagement, so much so that some aspects of it are taken into account in modern UX evaluation models, such as Google’s Web Vitals [Walton 2020]. Besides user experience, web applications with bad performance can cause an impact on the energy consumption of devices that access it, which causes degradation of their batteries and hardware [Chan-Jong-Chu et al. 2020]. Because of these issues, it’s important to understand the impact of frameworks and libraries on application performance.

This paper’s goal is to **analyze the impact of the virtual DOM on the performance of web applications**, by measuring and comparing different performance metrics of three identical web applications built using three different technologies: React and Vue, which use a VDOM (Virtual Document Object Model) to keep the UI up-to-date with the state, and Svelte, which uses a different approach. In order to reach this goal, the following specific objectives have been proposed: 1. understand which DOM updating approach is faster; 2. understand which DOM updating approach is more efficient; 3. understand how much memory and time is spent on scripting and painting on each DOM updating approach. The reasoning behind using both React and Vue for the comparison on the VDOM side of the study is so that two virtual DOM implementations are taken into account. That way, performance losses cannot be attributed to a framework’s specific implementation.

The result from this research is a detailed report of performance data, collected from three identical web applications built with different front-end technologies. The comparison in this report should help answer whether the concept of a virtual DOM is still beneficial for web performance or if the evolution of other models proposed by web frameworks has outgrown its benefits. The final result is an experiment that can be useful in aiding developers and engineers when making a choice of which technologies to incorporate into their web applications.

This paper is divided into the following sections: Section 2 aims to present the theoretical foundation, i.e. concepts that are used in the study. Section 3 discusses the

related work. Section 4 describes the methodology used in the study. Section 5 describes and shows the artifacts developed for the study. Section 6 presents the results of each metric. Section 7 provides a discussion of the results, and Section 8 concludes the study.

2. Theoretical Foundation

This section presents concepts that are addressed throughout the study. Section 2.1 defines the concept of User Experience on the web. Section 2.2 goes over the concept of the Virtual DOM. Section 2.3 provides a definition of Google Lighthouse, and Section 2.4 provides an overview of the frameworks that are analyzed in this paper.

2.1. User Experience

User experience can be generally defined as the interaction between a user and a product or service. It considers the user’s perceptions of efficiency, utility, ease of use, and aesthetics [Law et al. 2009]. ISO 9241, which addresses ergonomics of human-system interaction, defines user experience as “Users’ perceptions and responses include the users’ emotions, beliefs, preferences, perceptions, comfort, behaviors, and accomplishments that occur before, during, and after use.” [ISO 2019]. On the web, user experience can be affected by both technical aspects such as performance, responsiveness, and accessibility, and non-technical aspects, such as aesthetics and ease of use.

2.2. Virtual DOM

The virtual DOM (or VDOM) is a “virtual” representation of the application’s UI in memory, which is used to keep the application’s UI in sync with its state [Novac et al. 2021b, Javeed 2019]. Each UI element is represented by a node inside of a tree-like structure, similar to the website’s actual DOM. Whenever the application’s state is updated, it will propagate through the nodes in the VDOM. Then, the actual DOM is diffed from the VDOM, so that the changes made can be applied to it in an efficient manner. Figure 1 illustrates the functionality of the virtual DOM.

2.3. Front-end Frameworks

Svelte is a component-driven framework for building the front-end of web applications [Svelte 2022]. It functions as a compiler of Svelte component files, written with JavaScript, CSS, and HTML. The framework provides a template syntax for things such as conditional rendering of elements, data binding, loops, etc. It’s based on a reactive paradigm similar to that of React, but it aims to keep a smaller footprint of library code on the final bundle, generating code that dynamically updates the DOM at compile-time instead of relying on a VDOM like React or Vue [Harris 2016, Tripon et al. 2021]

React is a JavaScript framework for building user interfaces based on components [React 2022]. UI components in React are written in JSX (JavaScript Syntax Extension). They can be created either with classes (known as class components) or functions (known as functional components). Components can have states that store data, and when they get updated, those updates are reflected in the VDOM of the application, which is then compared to the web page’s actual DOM and dynamically updated. All of this logic is executed at runtime [Novac et al. 2021a, Javeed 2019].

Vue is a JavaScript framework for building web UIs that is based on the model-view-viewmodel (MVVM) architecture [Vue.js 2022]. Vue components are written in

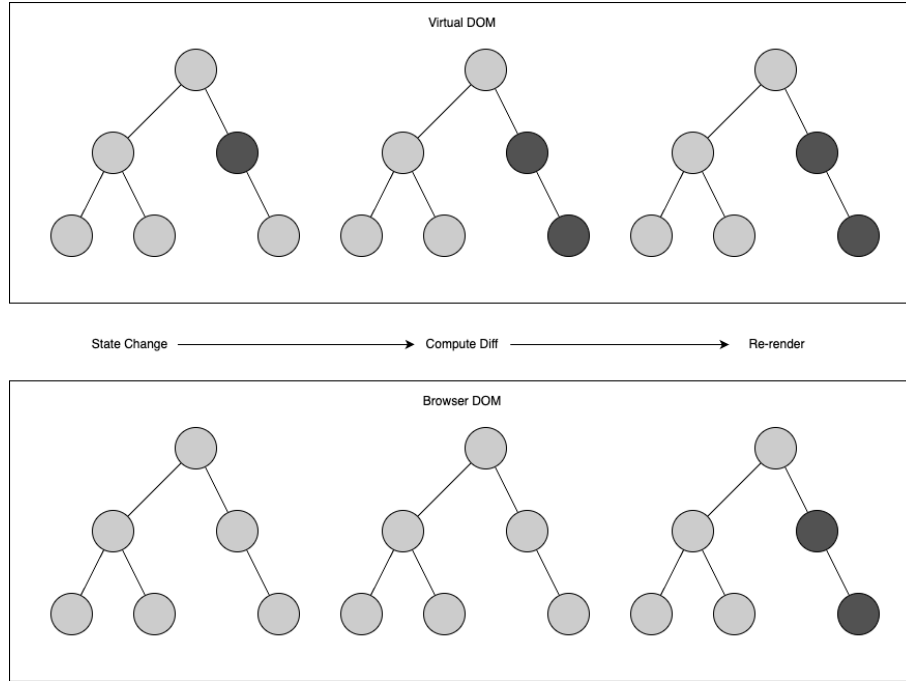


Figure 1. Virtual DOM. [Eisenman 2017]

JavaScript, HTML, and CSS, and they are provided with a “data” object that can be used to store any data. Similarly to React, it also uses a VDOM to efficiently and dynamically update the template when the component’s data is changed and is also executed at runtime [Novac et al. 2021a, Novac et al. 2021b].

2.4. Google Lighthouse

Google Lighthouse is an open-source tool for measuring quality metrics of web pages [Lighthouse 2022]. The metrics that can be obtained by it are separated into five categories: performance, accessibility, best practices, SEO (search engine optimization), and Progressive Web App. It can be run on any web page through the DevTools panel on any Chromium-based browser (i.e. Chrome, Edge, Brave, etc) or programmatically through its official “lighthouse” npm (Node Package Manager) package [Tripon et al. 2021, Chan-Jong-Chu et al. 2020]. Lighthouse is developed and actively maintained by Google, LLC.

3. Related Work

This section introduces the related work from other authors that are relevant to this paper’s subject. It features comparisons between JavaScript frameworks and performance analyses of web technologies and platforms.

Novac et al. (2021a) made an analysis of React and Vue in terms of performance. In order to compare both frameworks, the authors developed two identical calculator applications with them and analyzed and compared their loading, scripting, and rendering speeds using the profiler tool on Chrome DevTools. The conclusion of the paper is that both frameworks achieve similar performance and speed. Therefore the choice between the two comes down to other aspects, such as project requirements and preference. The

study is relevant to this paper in the sense that it provides a technical comparison between the two frameworks, which is what our study aims to do in order to compare different techniques of keeping the DOM up-to-date with the application's state.

Tripon et al. (2021) made a study comparing the Angular and Svelte frameworks in terms of performance, both during development and runtime. For the comparison, two "To-Do" SPAs (single-page applications) were developed using both frameworks analyzed in the study. Google Lighthouse was chosen to measure the applications' performance during runtime, and the metrics compared were: First Contentful Paint, Largest Contentful Paint, Total Blocking Time, Speed Index, Time to Interactive, and Cumulative Layout Shift. It was noted that during development Svelte was faster to compile and generate the JavaScript bundles for the application when compared to Angular. During runtime, Svelte also succeeded in obtaining better scores than Angular in most metrics. The study is relevant to this paper in the sense that it also provides a technical comparison between two frameworks - one of which is also analyzed in our study - and also serves as an example of a good set of metrics to use when measuring a website's performance.

Novac et al. (2021b) compared the performance of two different JavaScript frameworks: Angular and Vue. The study involved the development of two "To-Do" web applications using both addressed frameworks and comparing them in terms of performance. The applications were measured using the profiler tool available on Chrome DevTools, and the loading, rendering, and painting speed metrics were analyzed between both frameworks. The study found that Vue was capable of delivering better performance in all metrics compared to Angular. The conclusion given is that Vue is more efficient than Angular at runtime due to its shorter loading times. This study is relevant to our paper due to the fact that it compares two different JavaScript front-end frameworks in terms of performance, providing a theoretical framework of metrics that can be used to compare such technologies.

Nakajima et al. (2022) developed a playground tool named Jact for comparing JavaScript frameworks in terms of source code and runtime performance. Jact allows users to run, edit and share JavaScript code in their web browsers, and also allows them to preview the web pages generated by their source code and measure their runtime performance. In order to measure performance, Jact embeds user-generated code into a hundred iframes and measures loading time and time to trigger events. Measurements are handled by tasks that are created by the users of the tool, so that specific features of their web pages can be tested and compared with implementations made with different frameworks. In order to test the tool, a comparison is made between vanilla JavaScript code, jQuery, and Vue. The results showed that the Vue code has the most negative impact on performance among the web pages compared. This paper is relevant to our study because it also compares different JavaScript frameworks in terms of performance, albeit with a different goal - validating the tool developed in the article.

De Macedo et al. (2021) proposed a comparison of the performance and energy efficiency of JavaScript and WASM (WebAssembly) - a fast, safe, portable low-level byte-code for web applications. In order to compare them, three versions of nine algorithms were written. The three versions of each were: written in C, written in C and compiled to WASM, and written in JavaScript. The algorithms were executed and measured by a Python script. The script collected CPU usage (Joules), DRAM (Joules), GPU (Joules),

and time (seconds) measurements and exported the median and mean values to a CSV file. After the data was collected, the study pointed out that C was the fastest and most energy-efficient language out of the three, but WebAssembly provides slightly better performance than JavaScript. This work is relevant to our study because it also proposes a comparison of different web technologies in terms of performance.

4. Procedure and Methodology

This study aimed to perform a controlled experiment in order to do a quantitative analysis of the impact of VDOM on the performance of web applications. It can be classified as so due to the fact that it proposes the creation of a controlled environment of study objects (i.e. three identical web applications), and its analysis is based on aspects that can be measured in a quantitative manner, in this case, through a script. Section 4.1 describes the environment in which the study was performed. Section 4.2 presents the metrics that were used for the analysis and Section 4.3 presents the steps executed in the study.

4.1. Environment

In order to measure the impact of the VDOM, a comparison was made between three “To-Do” single-page applications that were built using three different technologies: React, Vue, and Svelte. The applications have two main functions: adding tasks and marking tasks as completed. In order to measure noticeable performance differences caused by DOM updates, each application starts with a thousand items added to the list, and one new item is automatically added every 100ms. This ensures that the rendering performance can be measured as new items added to the list trigger mutations to the DOM.

Each application was executed locally, in order to prevent network speed from affecting the measures. Google Lighthouse was used to measure the FCP (First Contentful Paint), SI (Speed Index), and TTI (Time To Interactive) metrics while the profiler tool in Chrome DevTools was used to measure memory usage, scripting, and painting time metrics [Lighthouse 2022].

Google Lighthouse can be used mainly through two different means: using the GUI present on the DevTools panel on any Chromium-based web browser (i.e. Chrome, Edge, Brave, etc.), or programmatically through their official package on npm [Chan-Jong-Chu et al. 2020]. This study chose to develop a node script that runs Lighthouse through the npm module in order to directly export the metrics to a JSON file. Each application was measured 10 times and the mean value of each metric was considered, in order to avoid unprecise results caused by unrelated performance variations that might happen during the execution of the tool.

The profiling tool was executed through Puppeteer, a Node.js library that provides an API to control the Google Chrome browser programmatically. Both the browser and the script were run on a 2021 MacBook Pro running macOS 13.2.1 with the base model M1 Pro chip and 16GB of RAM.

4.2. Metrics

The following metrics have been chosen in order to compare both approaches to updating the DOM:

1. **First Contentful Paint:** refers to how long, in seconds, the browser takes to render the first piece of DOM content. Lighthouse considers results from 0 to 2.5s as good, 2.5s to 4s as moderate, and over 4s as slow. This metric is useful to measure the page's initial overhead on the browser's memory [Tripon et al. 2021].
2. **Speed Index:** measures how quickly, in seconds, content is visually displayed as the page loads. Lighthouse captures a video of the page loading and computes time between the visual progression of frames. Results from 0 to 3.4s are considered fast, 3.4s to 5.8s are moderate, and over 5.8s are slow. This metric is useful to measure the time it takes for the DOM to handle updates done to it [Hoßfeld et al. 2018].
3. **Time to Interactive:** measures how long a page takes to become fully interactive, i.e. when the page displays useful content, its event handlers are registered for most visible elements and the page responds to user interaction within 50ms. Lighthouse considers results from 0 to 0.38s as fast, 3.9s to 7.3s as moderate, and over 7.3s as slow. This metric is used to measure the interactivity of the site [Tripon et al. 2021].
4. **Scripting time, painting time, and JS heap size:** These three values indicate the amount of time during page load that was spent registering events in JavaScript, translating the HTML into visual elements, and displaying them on the screen, and the memory heap size used by the website's JavaScript code. The purpose of using these values in our comparison is to see which aspect is more impacted by each DOM updating approach. They are measured using the profiler tool on Chrome DevTools [Novac et al. 2021b].

4.3. Steps

This subsection outlines the four steps devised in order to execute the proposed experiment. In the first step, three identical single-page applications were developed using React, Vue, and Svelte. The three applications were initialized with Vite - a front-end build tool that facilitates the creation of websites with different frameworks - and were all written in TypeScript. In order to avoid differences in performance not related to the frameworks, the three sites followed the same DOM structure and used the same CSS stylesheet. For the second step, a JS script was developed in order to measure the web applications. The script used the official "lighthouse" npm package from Google that programmatically executes the Lighthouse measuring tool. Alongside "lighthouse", the "chrome-launcher" package was used to programmatically launch a clean Chrome instance for the performance measurement, and "puppeteer" was used to automatically collect profiler metrics.

For the third step, the performance measuring script was executed 10 times on each application and the mean values of each measurement were saved on a JSON file. In the fourth step, the values obtained in the previous step were statistically analyzed through a Jupyter Notebook project¹ using SciPy and Matplotlib to generate charts and data visualizations for them. The normality of each dataset was then checked with the Shapiro-Wilk test and parametric and non-parametric ANOVA and Kruskal-Wallis tests were used to verify if the results are statistically different among frameworks.

¹<https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2022-2-tcci-5308100-pes-lucas-gusmao/blob/master/Instrumentos/OutrosInstrumentos/StatisticalAnalysis.ipynb>

5. Development

This section presents the software artifacts obtained throughout the experiment’s development and describes their functionalities.

5.1. To-Do Applications

Three identical “To-Do” web applications were developed using React, Vue, and Svelte. The applications have an identical HTML structure, use the same CSS stylesheets, and were generated using the “react-ts”, “vue-ts” and “svelte-ts” Vite templates. Figure 2 shows a screenshot of the Svelte application running on Chrome. For measurement purposes, each application starts with one thousand items added to the list, and a script on the applications’ template automatically adds one item every 100ms.

The applications are comprised of a single module with two attributes: `itemName` and `items`, and three methods/functions: `addItem()`, `checkItem()`, and `onFormSubmit()`. The React application contains one extra function: `onNameChange`, which updates the `itemName` value whenever the user types into the input field. This is necessary due to the fact that React doesn’t have a way to bind values directly to state variables, unlike Vue’s `“v-model”` and Svelte’s `“bind”` attributes.

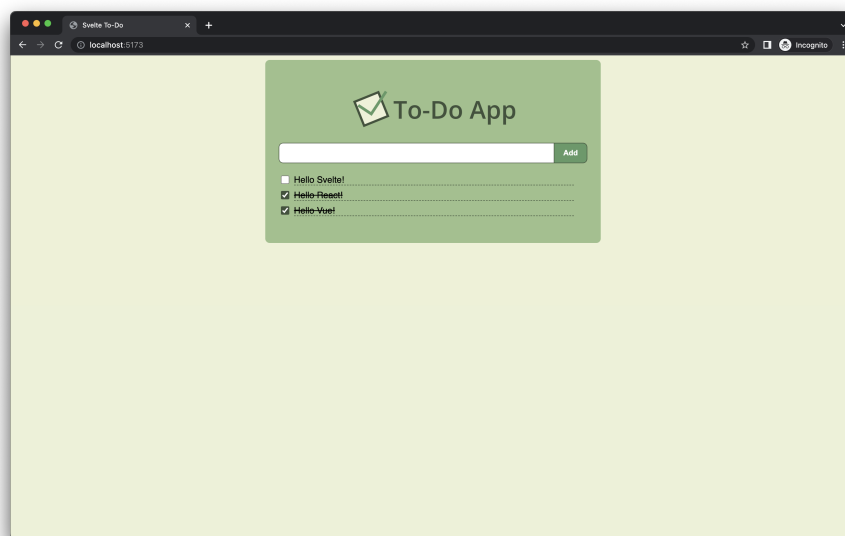


Figure 2. Svelte To-Do

5.2. Measuring Tool

An automatic tool was developed to automatically collect the desired metrics for this study. The tool was written in JavaScript and executed on Node.js v16.15.1 and used three main libraries to accomplish its goals. The first one, “chrome-launcher”, was used to automatically start and finish fresh instances of the Chrome browser, guaranteeing that no aspects such as extensions or usage data will interfere with the metrics. The second library - “lighthouse” - automatically runs Google Lighthouse on Chrome and returns the results in a JS object. The third and final library - “puppeteer” - takes over Chrome after it is launched and collected the remaining metrics by automatically running the browser’s

profiler tool. After Lighthouse and the profiler are executed ten times, the script compiles the data in a JavaScript array and saves it in a “results.json” file.

6. Results

After measuring the applications, the average, median, and standard deviation of each metric were calculated. All metrics presented low variance, which indicates that their averages provide a good picture of the overall results. Table 1 presents the average, median, and standard deviation values of each metric.

Metric	React	Vue	Svelte
FCP (ms)	Avg. = 7732 Median = 7704 Std. Deviation = 75	Avg. = 3607 Median = 3386 Std. Deviation = 2	Avg. = 3387 Median = 3386 Std. Deviation = 2
SI (ms)	Avg. = 5087 Median = 5090 Std. Deviation = 14	Avg. = 2390 Median = 2404 Std. Deviation = 31	Avg. = 2559 Median = 2559 Std. Deviation = 2
TTI (ms)	Avg. = 8132 Median = 8136 Std. Deviation = 105	Avg. = 3415 Median = 3415 Std. Deviation = 2	Avg. = 2989 Median = 2987 Std. Deviation = 7
Scripting (ms)	Avg. = 2538 Median = 2531 Std. Deviation = 32	Avg. = 144 Median = 142 Std. Deviation = 6	Avg. = 158 Median = 162 Std. Deviation = 12
Painting (ms)	Avg. = 66 Median = 65 Std. Deviation = 9	Avg. = 142 Median = 143 Std. Deviation = 5	Avg. = 166 Median = 170 Std. Deviation = 11
JS Heap Size (KB)	Avg. = 8.883,60 Median = 7.407,62 Std. Deviation = 2.033,74	Avg. = 5.849,68 Median = 5.434,30 Std. Deviation = 827,13	Avg. = 7.104,29 Median = 7.775,00 Std. Deviation = 1.268,64

Table 1. Average, median, and std. deviation of each metric in each framework

First Contentful Paint

The “First Contentful Paint” metric indicates the time a site takes to display its first piece of content. Lighthouse considers values up to 2,5 s as fast, up to 4,0 s as moderate, and above 4,0 s as slow. Both Vue and Svelte provided fast results, while React fell into the slow category. Figure 3 shows a boxplot chart of the results from each application.

Speed Index

Speed Index measures how quickly content is rendered in a website as it loads. Lighthouse considers values from 0 to 3.4s as fast, 3.4s to 5.8s as moderate, and over 5.8s as slow. Similarly to the previous metric, Vue and Svelte fell into the fast category while React fell into slow. Figure 4 presents a boxplot chart with the results.

Time to Interactive

Time to Interactive evaluates how long a page takes to be interactive after it is first displayed. Lighthouse considers values from 0 to 3.8s as fast, 3.8s to 7.3s as moderate, and over 7.3s as slow. Just like the previous metrics, Vue and Svelte were classified fast while React was classified as slow. Figure 5 presents a boxplot chart with the results.

Scripting, painting and JS Heap Size

These values indicate how long it takes for the browser to register JavaScript events, render visual elements from the template, and the amount of memory used by the JavaScript

on the site. Since these metrics are collected from the browser’s profiler, there is no standard for qualitatively categorizing them, but it can be noted that React performed poorly in scripting at the same time that it outperformed the other libraries in painting time. Regarding JS heap size, React and Svelte presented similar means while Vue performed significantly better. Figures 6, 7, and 8 show boxplot charts with each result.

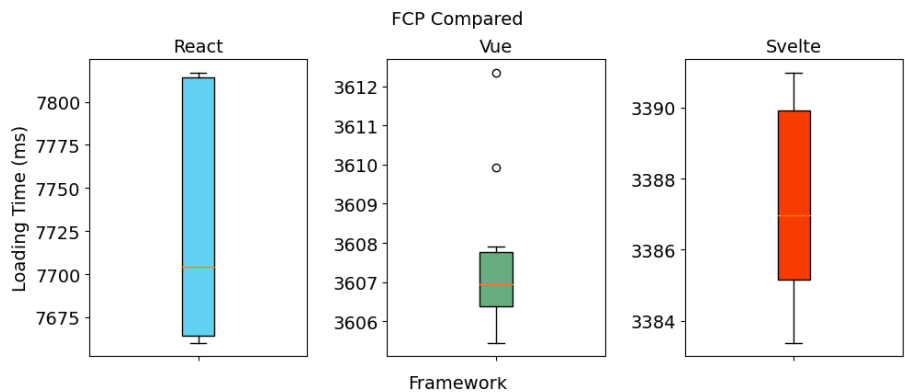


Figure 3. FCP values

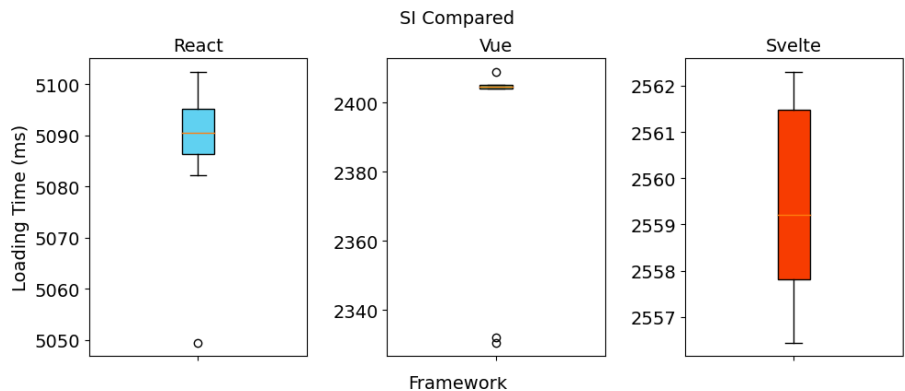


Figure 4. SI values

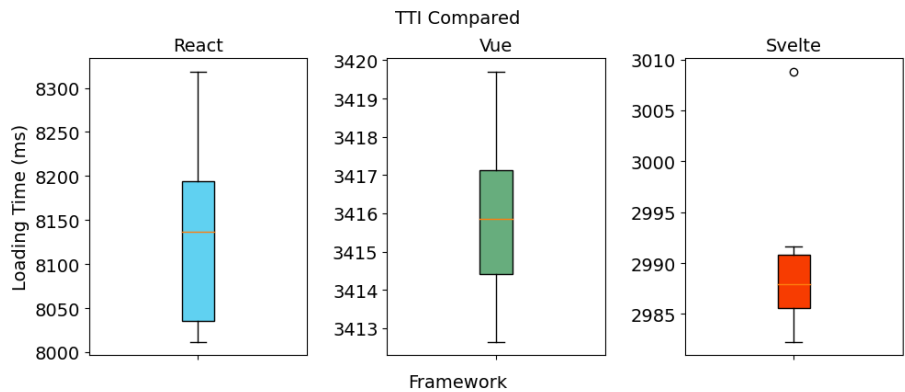


Figure 5. TTI values

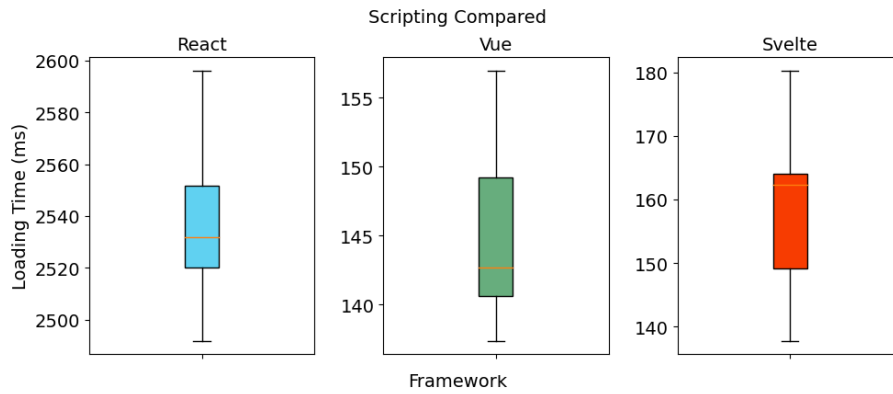


Figure 6. Scripting values

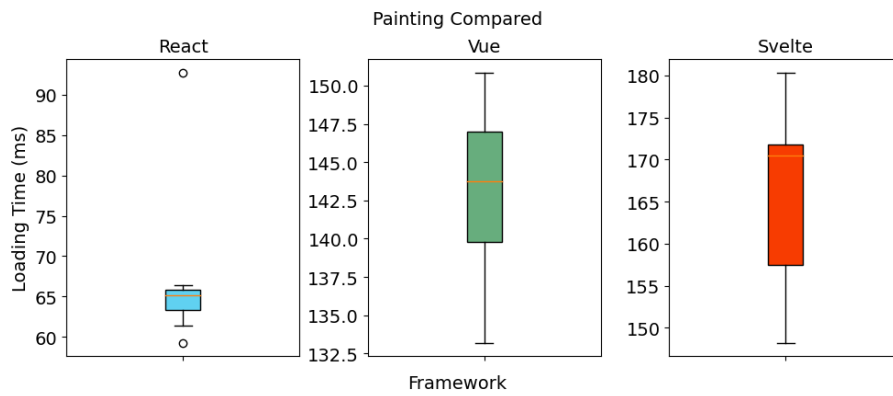


Figure 7. Painting values

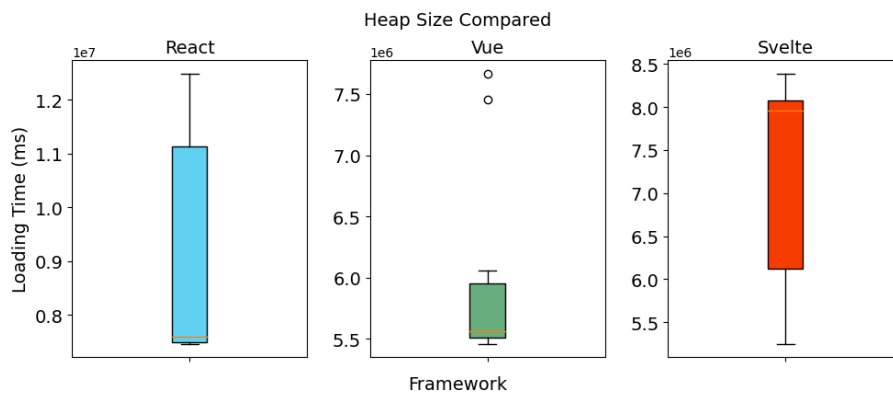


Figure 8. JS heap size values

6.1. Statistical Analysis

For the purpose of statistical analysis, all results obtained in the previous sections were utilized.

The statistical analysis was performed using the SciPy Python library. First, a Shapiro-Wilk test was carried out to determine the normality of the data. Then, non-parametric (Kruskal-Wallis) and parametric (ANOVA) analysis of variance was used to compare the results obtained from each sample. A p-value of less than 0.05 (corresponding to 95% confidence) was considered to indicate statistically significant differences.

Table 2 presents the results of each metric's statistical test, which reflect the impact of the Technology factor on the variables. These results indicate statistically significant differences ($p - value \geq 0.05$) in all variables based on the technology factor.

Metric	Test	React & Vue	React & Svelte	Vue & Svelte	React & Vue & Svelte
FCP (ms)	Kruskal-Wallis	1.57e-4	1.57e-4	1.57e-4	2.49e-6
SI (ms)	Kruskal-Wallis	1.57e-4	1.57e-4	1.57e-4	2.49e-6
TTI (ms)	Kruskal-Wallis	1.57e-4	1.57e-4	1.57e-4	2.49e-6
Scripting (ms)	Kruskal-Wallis	1.57e-4	1.57e-4	1.26e-2	1.54e-5
Painting (ms)	Kruskal-Wallis	1.57e-4	1.57e-4	2.85e-4	3.21e-6
JS Heap Size (bytes)	Anova	3.69e-4	3.05e-2	1.74e-2	3.49e-4

Table 2. Statistical tests and results

7. Discussion

After collecting the data from the applications and performing the statistical tests, some interesting aspects have been identified in each framework. This section goes over the performance implications of the results and discusses threats to the validity of this work.

Figure 9 shows a radar chart with each framework's strongest and weakest points.

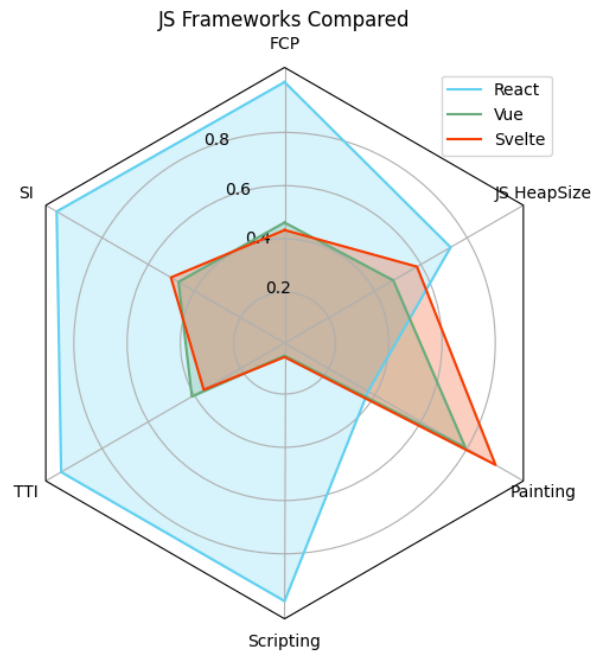


Figure 9. Radar chart of frameworks and metrics (normalized)

The results show that React performs worse than both Vue and Svelte in almost all metrics analyzed in this study. Both Svelte and Vue presented similarly good results in metrics that can be qualitatively classified (i.e. FCP, SI, and TTI), but Vue performed

slightly better than Svelte in terms of JS Heap Size. It's also notable that React performed better in terms of painting when compared to Vue and Svelte.

Previous studies have performed similar analyses with JavaScript frameworks, including the ones analyzed in this study, and yielded both similar and contrasting results [Almeida and de Paula 2022, Ollila et al. 2022]. One of the factors that could explain the better performance of Vue and Svelte in comparison to React is the different optimization techniques utilized by the two former frameworks. Vue tries to maintain more prop stability by only automatically updating child components when one of its parent props has changed. Svelte does something similar during compile-time instead of runtime. Meanwhile, React's functional components can be unnecessarily rerendered anytime a prop or state variable is updated [Vue 2023, Harris 2016, React 2022]. Another factor that could explain React's overall worse performance is the fact that it has existed for a long time as a front-end framework and strives to maintain backward compatibility.

This indicates that the Virtual DOM model is still capable of performing well at keeping track of UI state in web applications and that the performance issues encountered with React could be attributed to other factors in the library's implementation.

7.1. Threats to Validity

This section goes over some threats to validity that have been identified in this study, alongside the strategies that were adopted to minimize them.

Internal Validity

Internal validity refers to how non-controlled aspects of the experiment can affect its results. The main aspects that could influence the results are hardware and software instabilities that could affect the measurements. In order to mitigate these issues, multiple measurements were carried out in a local server environment on the same machine, running the same script on the same operating system and environment.

Conclusion Validity

Conclusion validity refers to the possibility of assuming general conclusions from the experiment's results. All of the frameworks analyzed provide a big plethora of features that can impact performance (such as state management, client-side routing, and server-side rendering, for example) which were not explored in the test applications developed in this study. However, considering that the goal of this paper is to analyze the impact of libraries on the rendering of UI elements on websites, these features are outside of the scope of the conclusions assumed from this study. In order to make sure that the rendering performance was evaluated, the applications developed for the experiment were designed to trigger multiple mutations to the DOM.

External Validity

External validity refers to the possibility of generalizing the experiment's results. The figures obtained in the study are useful in comparing the frameworks against each other since they were collected from identical applications developed with different technologies. One possible threat to the external validity of this experiment is the fact that it was only run on one environment (machine, OS, web browser, etc). However, considering the

fact that the majority of the install base of modern web browsers consists of Chromium-based browsers (i.e. Chrome, Edge, Opera, Brave, etc) [Statcounter 2023], the results of this study can be assumed to be relevant to the majority of web users today.

8. Conclusion and Future Work

This study performed a performance comparison of different JavaScript frameworks that implement two different architectures for keeping the application’s UI in sync with its state. In order to compare these two approaches, two frameworks with the VDOM (React and Vue) and one without the VDOM (Svelte) were chosen to be evaluated, and three identical “To-Do” web applications were developed and compared in terms of speed, visual progression, and memory use.

For the data collection, a script in Node.js was developed, utilizing the Lighthouse measuring tool and Puppeteer in order to automatically collect the metrics at runtime. After the metrics were collected, a Shapiro-Wilk test was executed to determine if the values had a normal distribution, and then the Kruskal-Wallis and ANOVA tests were used to compare the results and determine if there is a statistical difference between them. The tests indicated significant statistical differences (p-value < 0.05) in all variables based on the technology factor. One hypothesis is that the performance advantages of both Vue and Svelte can be due to optimization strategies used by both frameworks in terms of determining when components should be rerendered. The end results suggest that the VDOM does not hinder performance as much as Svelte claims it does and that the poor performance exhibited by React can possibly be attributed to other aspects of the framework’s implementation.

One potential area of future work is to investigate more aspects of these libraries’ performances in different environments and web browsers. Lighthouse is a powerful measuring tool, but it is limited to Chromium-based browsers. A browser-agnostic measurement tool that can obtain the same metrics as Lighthouse would open up the possibility of comparing the results in other relevant browsers such as Safari and Firefox.

Replication Package

All of the files required to reproduce this study can be found at:
<https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2022-2-tcci-5308100-pes-lucas-gusmao>

References

- Almeida, P. H. M. and de Paula, W. G. (2022). Análise comparativa das características de performance dos javascript frameworks react e vue.
- Chan-Jong-Chu, K., Islam, T., Exposito, M. M., Sheombar, S., Valladares, C., Philippot, O., Grua, E. M., and Malavolta, I. (2020). Investigating the correlation between performance scores and energy consumption of mobile web apps. In *Proceedings of the Evaluation and Assessment in Software Engineering*, EASE ’20, page 190–199, New York, NY, USA. Association for Computing Machinery.
- De Macedo, J., Abreu, R., Pereira, R., and Saraiva, J. (2021). On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet? In *2021 36th*

- IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 255–262.
- Eisenman, B. (2017). *Learning react native*. 2nd Edition. O’Reilly Media, Inc. November 2017.
- Friedman, V. (2014). Improving smashing magazine’s web performance: A case study. <https://www.smashingmagazine.com/2014/09/improving-smashing-magazine-performance-case-study/>. Accessed: 2022-09-10.
- Greif, S. and Benitte, R. (2020). State of js 2020. <https://2020.stateofjs.com/en-US/technologies/front-end-frameworks/>. Accessed: 2022-09-10.
- Harris, R. (2016). Frameworks without the framework: why didn’t we think of this sooner? <https://svelte.dev/blog/virtual-dom-is-pure-overhead>. Accessed: 2022-09-10.
- Harris, R. (2018). Virtual dom is pure overhead. <https://svelte.dev/blog/virtual-dom-is-pure-overhead>. Accessed: 2022-09-10.
- Hoßfeld, T., Metzger, F., and Rossi, D. (2018). Speed index: Relating the industrial standard for user perceived web performance to web qoe. In *2018 Tenth International Conference on Quality of Multimedia Experience (QoMEX)*, pages 1–6.
- ISO (2019). Iso 9241-210: Ergonomics of human system interaction.
- Javeed, A. (2019). Performance optimization techniques for reactjs. In *2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–5.
- Law, L.-C., Roto, V., Hassenzahl, M., Vermeeren, A., and Kort, J. (2009). Understanding, scoping and defining user experience: A survey approach. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 719–728.
- Lighthouse, G. (2022). Google lighthouse. <https://developer.chrome.com/docs/lighthouse/overview/>. Accessed: 2022-11-08.
- Nakajima, N., Matsumoto, S., and Kusumoto, S. (2019). Jact: A playground tool for comparison of javascript frameworks. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 474–481.
- Novac, C. M., Novac, O. C., Sferle, R. M., Gordan, M. I., BUJDOSÓ, G., and Dindelegan, C. M. (2021a). Comparative study of some applications made in the vue.js and react.js frameworks. In *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–4.
- Novac, O. C., Madar, D. E., Novac, C. M., Bujdosó, G., Oproescu, M., and Gal, T. (2021b). Comparative study of some applications made in the angular and vue.js frameworks. In *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–4.
- Ollila, R., Mäkitalo, N., and Mikkonen, T. (2022). Modern web frameworks: A comparison of rendering performance.

- Osmani, A. (2018). A netflix web performance case study. <https://medium.com/dev-channel/a-netflix-web-performance-case-study-c0bcde26a9d9>. Accessed: 2022-09-10.
- Palmer, O. (2016). How does page load time impact engagement? <https://www.optimizely.com/insights/blog/how-does-page-load-time-impact-engagement/>. Accessed: 2022-09-10.
- React (2022). React. <https://reactjs.org/>. Accessed: 2022-11-08.
- Sam Meder, V. A. and Chang, J. (2017). Driving user growth with performance improvements. <https://medium.com/pinterest-engineering/driving-user-growth-with-performance-improvements-cfc50dafadd7>. Accessed: 2022-09-10.
- Statcounter (2023). Browser market share worldwide. <https://gs.statcounter.com/browser-market-share>. Accessed: 2023-05-13.
- Svelte (2022). Svelte. <https://svelte.dev/>. Accessed: 2022-11-08.
- Tripon, T.-D., Adela Gabor, G., and Valentina Moisi, E. (2021). Angular and svelte frameworks: a comparative analysis. In *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–4.
- Vue (2023). Vue - performance. <https://vuejs.org/guide/best-practices/performance.html>. Accessed: 2023-04-29.
- Vue.js (2022). Vue.js. <https://vuejs.org/>. Accessed: 2022-11-08.
- Walton, P. (2020). Web vitals. <https://web.dev/vitals/>. Accessed: 2022-09-10.