Comparative Performance Analysis of React, Vue.js and Svelte

Carlo Pastor Montrucchio

Thesis

Supervisor: Dr Robert Polding

03/05/24

Abstract

This paper compares the performance of three very popular website creation frameworks in the developer world: React, Vue.js and Svelte. This is achieved by creating the exact same webpage using the different frameworks and analysing their rendering performance metrics with a tool called webpagetest.org. The webpage consists of a simple movie webpage that displays a list of forty trending movies with their corresponding details, achieved by requesting the information from The Movie Data Base (TMDB) with an API key that they provide for free.

To be able to be analysed by webpagetest.org, the three webpages were deployed to the web using GitHub Pages:

https://carlo8pastor.github.io/svelte/

https://carlo8pastor.github.io/react/

https://carlo8pastor.github.io/vue/

The conclusion of this paper is that, in addition to great communities, ecosystem and ease of learning, the three frameworks had extremely similar rendering performance metrics. This means that if a beginner developer is overwhelmed by the abundance of web frameworks available, it would not be detrimental to simply pick randomly between React, Vue.js and Svelte without subjecting oneself to further thought and stress (simply 'get the ball rolling'). Another conclusion drawn from the process of creating this paper is that, if you are a beginner developer, starting with Angular may be more challenging due to its steeper learning curve.

Table of Contents

Introduction

When delving into the world of web development, the person at hand is often bombarded by a plethora of options to create their webpage. First, they could come across options such as WordPress, WebFlow and Wix, which are mainly non-code and offer a comfortable solution to less tech savvy people, yet provide many features which make web development extremely comfortable and even useful to developers. In fact, the majority of websites are made with WordPress. Next, there is the option of creating the website from scratch with an HTML, CSS and JavaScript file. However, if the website you are creating is complex, it can be very challenging and time consuming to do it in this manner. Thus the remaining options are, again, an overabundance of frameworks, such as React, Angular, Vue.js, Svelte, Express.js, Django, Ruby on Rails, Flask, Laravel, Bootstrap, Nuxt.js, Next.js and Flutter.

This can make it stressful and difficult for a ‘wannabe developer’ to know where to start from or what to choose. They are overloaded with all the information that the web provides. Thus, being interested in this problem myself and having some experience with React, I wanted to see if there was any noticeable difference between at least three popular frameworks of my choice, not only in performance, but ease of webpage creation. Before doing so, I looked into existing literature.

Literature Review

First Study

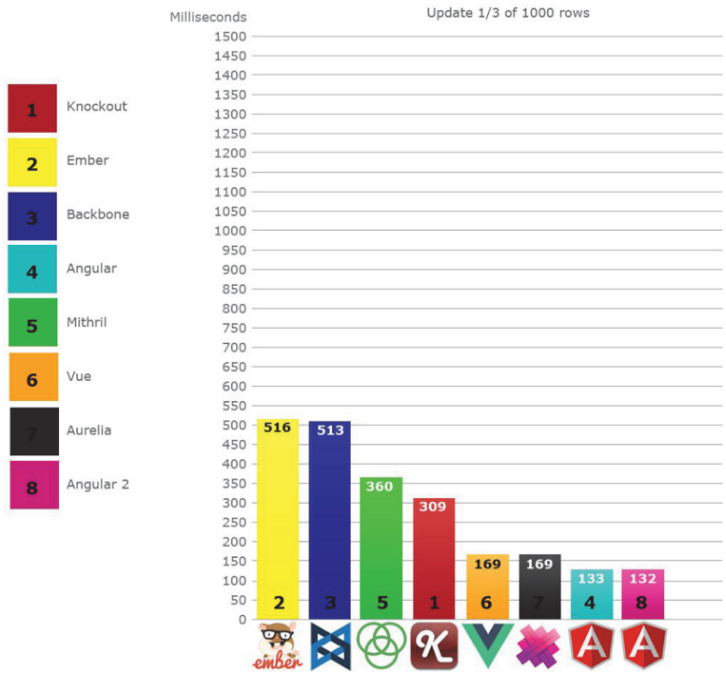
One study relevant to my topic is a 2015 thesis called ‘Speed Performance Comparison of JavaScript MVC Frameworks’ by Alexander Svensson, from the Blekinge Institute of Technology. In this paper he analyses the performance of several popular JavaScript Model-View-Controller (MVC) and MVC-like frameworks. He looked to see which of these would be more efficient in creating, updating and deleting HTML elements, the frameworks being AngularJS (1.5), AngularJS 2.0, Aurelia, Backbone, Ember, Knockout, Mithril and Vue.js. It is worth noting that AngularJS and AngularJS 2.0 are no longer actively maintained, and are not the same thing as the current version of Angular.

His primary objective was to empirically determine the speed performance of these frameworks in real world scenarios. He decided to use the time it takes for each framework to execute Document Object Model (DOM) manipulations as the main criteria for performance evaluation since this is critical for web application responsiveness and user experience. In order to achieve a fair and accurate comparison, Svensson employed Google Chrome’s Timeline tool, a measurement tool that lets you view detailed insights regarding time spent on script execution, rendering, and other browser tasks. In order to view these statistics he created benchmark tests simulating the creation, deletion and updating of a large number of HTML elements, attempting to mimic the dynamic nature of today’s web applications.

His analysis ended up revealing varying differences in the performance of each framework. Overall, AngularJS 2.0 was the best performer, demonstrating a superior efficiency in most of the benchmark tests. This highlighted the advancements made in AngularJS 2.0 in

comparison to its predecessor AngularJS, especially in terms of speed and optimisation for modern web apps. Nonetheless, in some of the tests, AngularJS came very close to AngularJS2.0, and showed a very good performance in tasks related to updating HTML elements, proving that while it may not be the best choice for all its use cases, it remains a viable option for applications where frequent updates to the DOM are required.

Additionally, the study revealed that Vue.js and Mithril were two strong performers, being lightweight frameworks that offer a very nice balance of speed and flexibility. They specifically excelled in the creation and deletion of HTML elements, meaning that they would also be a suitable choice for highly dynamic web applications. Finally, the remaining frameworks; Backbone, Ember and Knockout, showed varying degrees of performance lag compared to the top performers. A good conclusion for these is that their use may be better for projects where their particular features and ecosystem advantages outweigh the need for raw speed performance.



This is a graph from one of Svensson's benchmark tests (updating elements) which helps visualise the study's results: Angulars and Vue.js being the fastest. In this particular benchmark, Aurelia came third. Aurelia also came in second place for creating elements and was almost as fast as Angular in terms of updating elements, however it was the second slowest framework overall.

Second Study

Another study relevant to mine is a 2023 article written by Mangapul Siahaan and Ryan Kenidy named 'Rendering performance comparison of react, vue, next, and nuxt'. As suggested by the title, it presents an analysis and comparison of the rendering performances of React, Vue.js, Next.js and Nuxt.js, all popular JavaScript frameworks. Again, the researchers used various benchmarks and profiling tools to help measure the performance of these frameworks across different scopes, including rendering speed and server-side rendering capabilities, displayed in the table below.

Table 1. Performance Metrics

Metrics	Description	Tools		
		WebPage Test	PageSpeed Insight	GTMetrix
Load Time	Measure the website loading time for a fully rendered website (De Munk & Malavolta, 2021)	✓		✓
First Byte	Measure time taken to receive the first byte (Eslamdoust, 2022)	✓		✓
Start Render	The first content being rendered that can be seen by user even if it's background (Król, 2018)	✓		✓
Speed Index	Measure the speed for the content of the page to be filled (De Munk et al., 2022)	✓	✓	✓
Cumulative Layout Shift	Measure the load time when website element shift until it finishes (Mokhtari et al., n.d., 2022)	✓	✓	✓
First Contentful Paint	Time taken for the first element to be rendered (Tengriano et al., 2022)	✓	✓	✓
Largest Contentful Paint	Time taken for the largest element to be rendered (Justus & Jaeger, 2022)	✓	✓	✓
Total Blocking Time	Amount of time used for blocking the element to be rendered (Hossain et al., 2021)	✓	✓	✓
Time to Interactive	Total time for a website to be fully interactive (Namoun et al., 2020)	✓		✓

After the tests, they found that Vue.js and React were the top performers in most areas, such as Load Time, with Vue.js leading at 0.3s and React closely behind at 0.4s. They also both were better in delivering quick First Byte Response times (0.1s), highlighting their capability in fast data transmission initiation.

Furthermore, Vue.js stood out in Start Render and Speed Index metrics, highlighting its superior ability in quickly displaying content and ensuring a responsive user experience. Next.js and Nuxt.js, being extensions of React and Vue.js respectively, showed slightly longer times in those metrics, potentially impacting user experiences in real-time data rendering scenarios. Additionally, Vue.js and React were impressive with minimal Cumulative Layout Shift (CLS) values (0s each time) which ensure a visually stable experience for users. The two frameworks also demonstrated a better performance in minimising unresponsiveness (Total Blocking Time – TBT) and quick interactivity (Time To Interactive – TTI).

Overall, the study showed that React and Vue.js were better performers, but only by a very small margin. It is worth noting that their average performance only outdid Nuxt and Next in terms of milliseconds, something that a typical user will not notice and may only be beneficial to certain types of projects. In fact, Next and Nuxt had better performances on the First Contentful Paint (FCP) and Largest Contentful Paint (LCP) metrics. Therefore, when choosing what framework to use, again it depends on the project and what performance metrics you want to prioritise. It may be viable to choose Next and Nuxt simply because they provide more comfortability when it comes to creating certain websites, something that one may be willing to trade for a slightly lower performance. The following is one of the summarising tables from the study:

Table 6. Average of Webpage Test, PageSpeed and GTMetrix

Metrics	React	Vue	Next	Nuxt
LT	0.3 s	0.3 s	0.8 s	0.9 s
FB	0.1 s	0.1 s	0.4 s	0.4 s
SR	2.2 s	0.3 s	0.6 s	0.5 s
SI	2.5 s	1.7 s	3.7 s	4 s
CLS	0 s	0 s	0 s	0 s
FCP	2 s	0.7 s	1 s	0.8 s
LCP	6.7 s	7.9 s	6.3 s	6.1 s
TBT	0.1 s	0.1 s	0.4 s	0.7 s
TTI	2.7 s	1.8 s	3 s	3.7 s
Average	1.8 s	1.4 s	1.9 s	1.9 s

Third Study

Next is a 2020 thesis by Matias Levlin that analysed React, Vue.js, Angular and Svelte, four of the most popular JavaScript Frameworks. React came out as the winner, with no discernible weaknesses and many strengths. Its only noted drawback along with Vue.js was its virtual DOM, adding a few milliseconds to simple singular DOM updates. This study also discussed ‘non-technical’ aspects such as framework popularity and community, noting that Svelte, although a small community, may be the easiest and fastest to develop with. Levlin also commented that Angular was the fastest framework when deleting one DOM element, however, he described it as the most “bloated”. In the conclusion, Levlin suggested that Svelte’s approach without a virtual DOM might cause a shift in the web development landscape. Following is a table that summarises the testing and development metrics of the study:

TECHNICAL METRICS	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Add 10,000 (ms)	30.96	25.36	52.75	31.26
Edit one (ms)	16.58	22.23	6.08	0.11
Edit 10,000 (ms)	17.86	20.64	896.76	885.03
Remove one (ms)	16.54	24.51	0.09	0.53
Remove 10,000 (ms)	7.39	33.33	23.83	22.97
Compilation (s)	3.96	3.07	8.70	1.61
Minified size (kb)	6.4	63.5	187.6	3.5
POPULARITY	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Documentation in number of languages	16	8	4	2
Positive interest in StateOfJs 2019 (%)	83.7	74.7	31.6	51.7
Number of Github stars, January 2020	142,850	159,091	29,756	56,789
OTHER METRICS	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Virtual DOM	Yes	Yes	No	No
TypeScript	No	No	Yes	Yes
Release date	2013	2014	2016	2016
OVERALL RATING	React v16.12.0	Vue v2.6.11	Angular v8.2.14	Svelte v3.20.0
Placement	1	2	4	3

Table 8: A complete summary of all testing and development metrics.

Fourth Study

An additional study is a 2019 thesis comparing Angular, React and Vue.js, by Elar Saks. He found Vue.js to be the fastest and React coming in at second place, beating Angular in six tests out of 8. He found that React had the smallest production build with Vue.js and Angular coming after respectively. However, Saks noted one of the study's limitations, this being that the complexity of the application and its size affects the framework's speed performance, and speed is the only metric that was analysed, meaning that it would be necessary to carry out the test again with applications of different sizes and complexities. He mentions that in the case of building large scale applications, Angular might outperform the other frameworks. The following is a table of the results he collected:

Table 6. Frameworks loading speeds

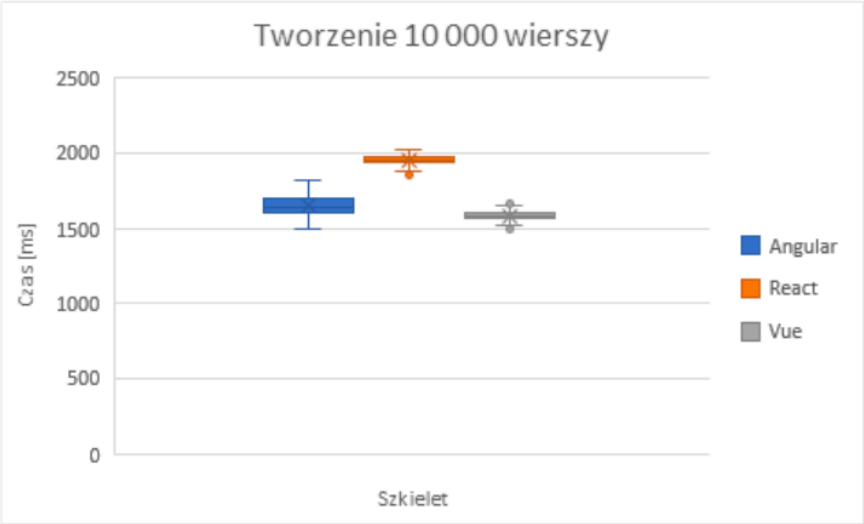
	Angular	React	Vue
Load Page	403	269	246
Create 1000	384	420	232
Re-create 1000	331	190	117
Add 1000	254	370	185
Create 10 000	6361	2866	1447
Re-create 10 000	7427	1176	547
Add 10 000	7623	3854	1436
Remove all	2196	607	505
Total	24978	9752	4715
Multiplier	5.30	2.07	1.00

Fifth Study

This 2022 article by Raimundo N.V. Diniz-Junior et al. analyses React, Vue.js and Angular. In their abstract, they comment that Vue.js was 758% faster than React and 595% than Angular in manipulation time. They also found that Angular occupies more bundle space, with React and Vue.js coming after respectively. They found that React had the best TTI with Vue.js in second place. As can be seen, the results are similar to the previous study except for the manipulation time, where here Angular was faster than React.

Sixth Study

Finally, this 2021 Polish article by Konrad Bielak et al. which also looked at the same three frameworks, found that Vue.js was more efficient than React and Angular, in particular when handling a large amount of data and performing delete and edit operations. Angular came second, doing well in editing and deleting small amounts of data, while React was deemed the least efficient, especially in external functionality. Nevertheless, in their conclusion, they also note (similarly to the Fourth Study) that these results should not be generalised without further research since a different application may yield different results, and that the one they used had very basic functionalities. The following is a graph from one of their benchmark tests:



Rysunek 7: Porównanie czasów tworzenia 10 000 wierszy.

Post Literature Review

After my literature review, I decided to choose React, Vue.js and Angular to try the testing myself. I decided that, similarly to some of the studies, I wanted to analyse render and timing metrics such as load time (LT), time to title (TTT), first contentful paint (FCP), largest contentful paint (LCP), cumulative layout shift (CMS) and time to interactive (TTI), which all are explained in the *Performance Metrics* section. However, regarding Angular (not to be confused with AngularJS), no matter how many hours I spent, I could not get my movie webpage running, whilst having already completed it with React and Vue.js. I thus decided to change Angular to Svelte, leaving React, Vue.js and Svelte as my 3 chosen frameworks.

So, what is React?

React is a very popular JavaScript library, one of the most popular in the development world, designed and developed by Facebook (now Meta) for building user interfaces (UI). It was developed in 2013 and rapidly gained fame for the efficient rendering and flexibility that it provides (Hámori, 2022). The main philosophy of React is in declarative programming, where the developer chooses what the UI should look like for the various states, and updating it is dealt with efficiently upon changing data.

Also at the heart of React's design are reusable components which are self-sufficient in maintaining state information. In fact, the very idea of React is the possibility to create large and complex applications, breaking the UI down into small elements that are called components which act more modestly in the greater ecosystem. Each can be composed together and inherited in data from its parent component by the use of properties or props. State

management within components allows React to only re-render components that have changed as opposed to the whole application, improving the developer experience.

React uses JSX files, which simply means that the user can add both HTML and JS inside the file, making the code easier to understand since it more closely resembles how the UI is structured. It is then transpiled into JavaScript. Additionally, in React version 16.8, they introduced hooks that allow function components to manage state and side effects—something that could only be previously achieved by using a class component (React, n.d.). Using hooks, like useState and useEffect, give a much cleaner and more maintainable code base by providing a more direct API to the React features without changing component hierarchy.

In the web development industry, React is highly valued for its scalability and flexibility. Companies using React include Meta, Instagram, Airbnb, DropBox, UberEats and Pinterest, which deal with content that is dynamic and provide interfaces for the users that demand high levels of interactivity (Fetisov, 2023). Regarding React's ecosystem, there is Redux for state management and Next.js for server-side rendering, which reinforces its abilities to be a good fit for small and large applications. And, of course, another strong aspect of React, beside the rich ecosystem of tools and extensions, is its performance, which is a primary reason for why people continue to use it.

To conclude, with great community support and continuous development by Meta and independent contributors, it is one of the best places for a developer who already has JavaScript experience. All of the aforementioned reasons are how React remains at the forefront of emerging technologies and maintains an extremely large community, something which can easily be observed in the YouTube space.

What is Vue.js?

Vue.js is a progressive JavaScript framework used to build UIs and mostly famous for single-page applications - though not limited to. Developed by Evan You in 2014, it is also a rapidly growing, widely adopted, flexible, detailed and comprehensive framework due to its gentle learning curve and very approachable architecture (Vue.js, n.d.-a). It can be mixed with other libraries or existing projects in such a way that it is a great choice for a pragmatic developer who wants to add interactivity to a webpage without having to redo everything in the project.

The core of Vue.js is that it is designed to be incrementally adoptable. It helps you build powerful applications with modern tooling and supporting libraries, and, as mentioned, it is progressive, meaning that you can use only the parts you need and it is very adaptable. Vue.js also features a very nice reactive data binding system. This makes sure that any change in the data is propagated to all views that depend directly on it, thus keeping state management very easy and intuitive.

Furthermore, Vue.js uses a template syntax to bind the rendered Document Object Model (DOM) to the Vue.js instance data. All Vue.js templates are essentially valid HTML nodes with special attributes for binding, making them parsable by standard-compliant browsers and HTML parsers. These templates are then compiled by Vue.js into virtual DOM render functions (Vue.js, n.d.-c). Performance optimisation within Vue.js is also a core aspect, offering a very reactive tracking of dependencies, which is performed in such a way that the system knows exactly which part of the components really needs to be re-rendered when there is a state change, minimising the number of DOM manipulations (Vue.js, n.d.-b).

Moreover, the ecosystem includes many other tools, such as the use of Vue.js Router for routing in the client-side and Vuex to manage state when developing applications of a complex nature, in addition to the fact that the Vue.js CLI makes it easy for developers to begin projects and adjust them with certain configuration options as they progress (Hussain, 2024).

Overall, the reason why companies and developers like Vue.js is due to its flexibility, gentle learning curve, scalability, detailed documentation, and strong community support, which makes its choice for building progressive web applications a highly reliable one. Finally, like React, Vue.js is also component-based, allowing for a structured way of developing powerful and effective user interfaces while maintaining both code readability and maintainability.

What is Svelte?

Svelte, while slightly newer, is also a component-based JavaScript framework. However, it transfers a lot of work normally done by the browser at runtime to be processed at compile time instead. This, in addition to replacing much of what can be recognised as boilerplate code, produces highly efficient code that updates the DOM. Ever since Rich Harris released it in 2016, it began gaining developer interest due to its unique approach to building user interfaces, and, up until now, has gained more popularity (Markham, 2024). Compared to React or Vue.js, which both allow applying changes through a virtual DOM, Svelte writes the code that surgically updates the DOM every time changes in the application state occur, which could lead to better performance.

The philosophy of Svelte is to offer a much less complex developer experience and gain better performance by compiling the components to vanilla JavaScript at build time. A Svelte application does not have any runtime for the framework. The components are built as independent JavaScript modules that can maintain their own state and reactivity, which results in smaller bundles and faster execution time because there is no abstraction layer between the code and the browser (Harris, 2016).

Additionally, Svelte has quite a clean and simple syntax that aims for developers to write less code yet gain more functionality. An App.svelte file uses HTML, CSS, and JavaScript, which are supposedly improved with the Svelte syntax that provides an opportunity to write reactive statements directly in markup. For example, you can create reactive declarations that update the DOM when there are changes in underlying data simply by assigning a variable.

Furthermore, Svelte has some advanced features such as reactive assignments and stores. This refers to simple writable objects that store the global state of application, which can be subscribed to and managed by components. Svelte also includes transitions, animations, and interactive features to your application with built-in utilities, avoiding bloating your directory size with third-party libraries (Svelte, n.d.).

Overall, Svelte is considered a good choice for projects where speed and low overhead are a consideration. Whether it's for small interactive web components or large, complex applications, it also provides a great development experience for developers. And, although not yet as broad as one would expect from React or Vue.js, it has a very strong and dedicated community that also contributes to beneficial support, resources and tooling. In fact, its community is what caused it to catch my eye in the past, influencing me to choose it as the third framework.

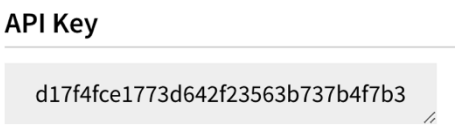
What is The Movie Data Base (TMDB)?

TMDB is how I will be retrieving and displaying the information for my React, Vue.js and Svelte webpages. It is a platform through which developers can get access to movie content from a very large database, which includes all listings for movies and TV shows, in addition to a lot of information regarding titles, cast, crew, plot summaries and posters.

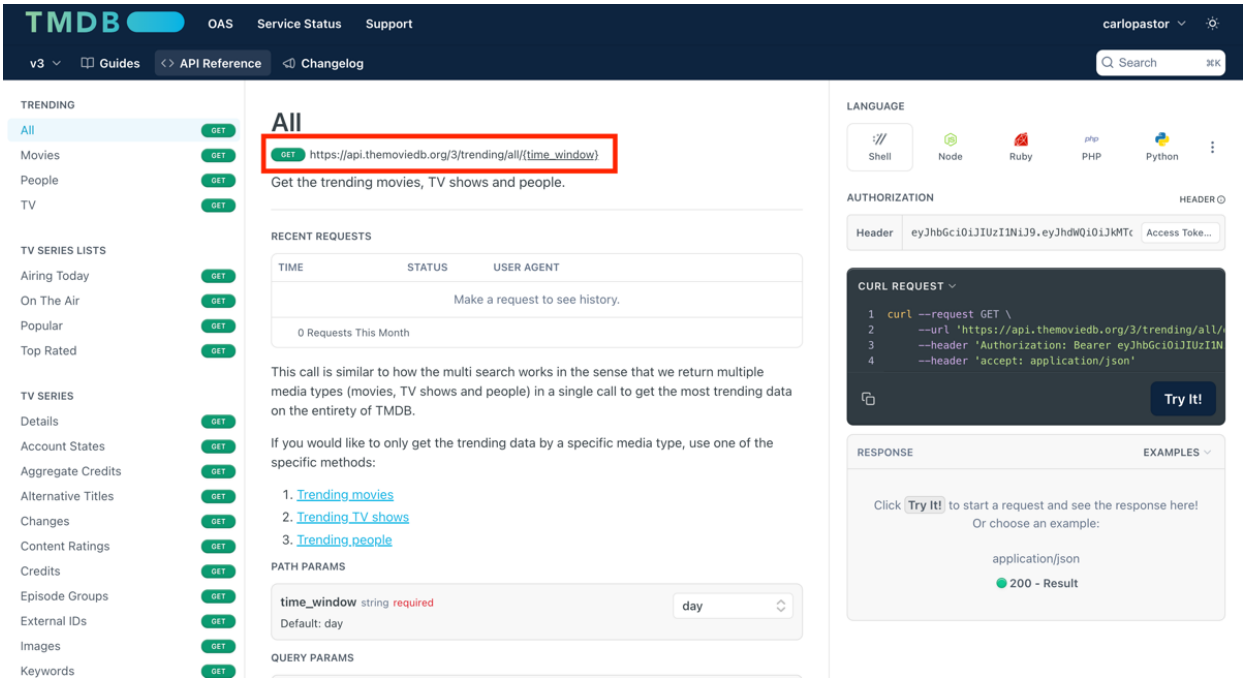
Each version of my webpage will dynamically display forty movies, including their poster path, English title, original title, release date, genres, duration, rating, overview, age rating, cast, director, original language, budget and trailer link.

The data retrieval will be a testament to how each framework handles API calls, data parsing and the rendering of this fetched data on the webpage. Using TMDB for the three frameworks acts as a control variable in the sense that, any performance difference can be attributed to the frameworks involved rather than differences or variations in data sources.

The following is an example of the API Key that TMDB provides once you have created an account and request the key. It is completely free to do so:



The TMDB team also do a great job in helping you utilise their API. They have an 'API Reference' section as seen in the image below, with a menu bar on the left showing you what type of information you can request. Once you click on one of these, you are then provided with instructions on how you can request it. For example, in the image below, they tell you that to get trending movies, tv shows and people, you have to use the link *'https://api.themoviedb.org/3/trending/all/'* in your code.



In the following section, I will be showing my three framework's project directories and some code snippets of how the movies and their details were obtained from TMDB. The full codes can be viewed in their repositories:

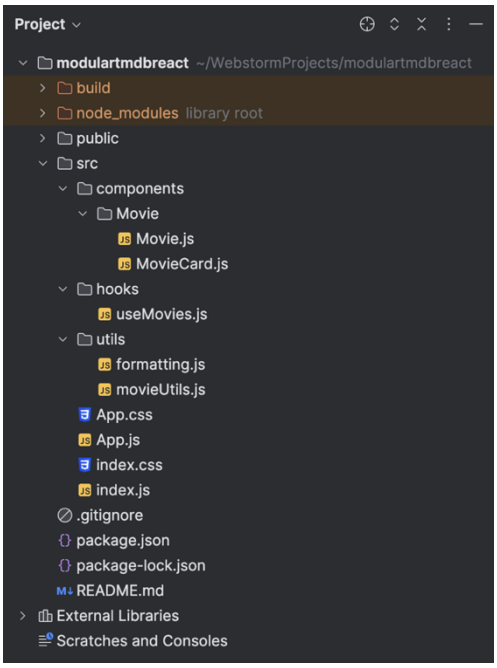
https://github.com/carlo8pastor/react

https://github.com/carlo8pastor/svelte

https://github.com/carlo8pastor/vue

Methodology for React

Project directory:

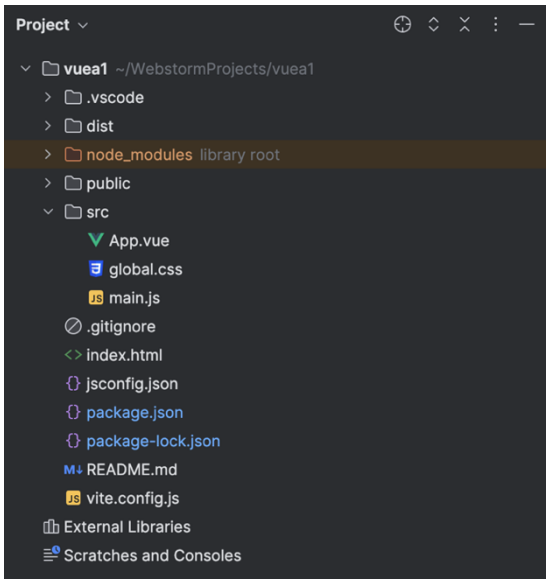


Snippet from movieUtils.js:

```
export const getMoviesWithDetails = async () => {
    const moviesWithDetails = [];


    for (let page = 1; page <= 2; page++) {

        const moviesResult = await fetchAPI(`${BASE_URL}/discover/movie?api_key=${API_KEY}&page=${page}`);


        if (moviesResult && moviesResult.results) {

            const detailPromises = moviesResult.results.map(async (movie) => {

                const details = await fetchAPI(`${BASE_URL}/movie/${movie.id}?api_key=${API_KEY}`);
                const credits = await fetchAPI(`${BASE_URL}/movie/${movie.id}/credits?api_key=${API_KEY}`);
                const videos = await fetchAPI(`${BASE_URL}/movie/${movie.id}/videos?api_key=${API_KEY}`);

                const director = credits && credits.crew ? credits.crew.find(member => member.job === 'Director') : null;
                const trailer = videos && videos.results ? videos.results.find(video => video.type === 'Trailer' && video.site === 'YouTube') : null;

                return {
                    ...movie,
                    director: director ? director.name : 'N/A',
                    cast: credits ? credits.cast.map(actor => actor.name).join(', ') : 'N/A',
                    genres: details ? details.genres.map(genre => genre.name).join(', ') : 'N/A',
                    runtime: details ? formatRuntime(details.runtime) : 'N/A',
                    vote_average: details ? `${details.vote_average}/10` : 'N/A',
                    original_language: details ? getLanguageFullName(details.original_language) : 'N/A',
                    budget: details && details.budget ? `$${details.budget.toLocaleString()}` : 'N/A',
                    trailer: trailer ? `https://www.youtube.com/watch?v=${trailer.key}` : null,
                    adult: details ? details.adult : false,
                };
            });
            const moviesDetails = await Promise.all(detailPromises);
            moviesWithDetails.push(...moviesDetails);
        }
    }
    return moviesWithDetails;
};
```

Methodology for Vue.js

Project directory:

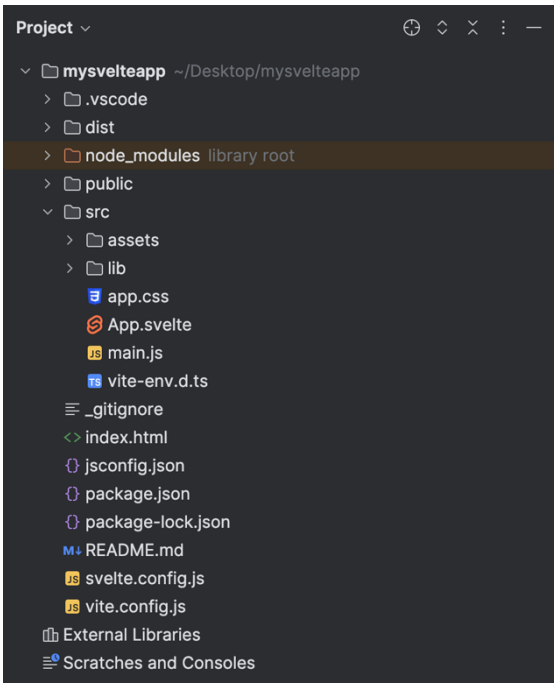


Snippet from App.vue:

```
onMounted(async () => {
  const moviesPage1 = await getMoviesWithDetails(1);
  const moviesPage2 = await getMoviesWithDetails(2);
  const combinedMovies = [...moviesPage1, ...moviesPage2];

  const moviesWithDetailsPromises = combinedMovies.map(async (movie) => {
    const detailsRes = await fetch(`${baseURL}/movie/${movie.id}?api_key=${apiKey}`);
    const detailsJson = await detailsRes.json();
    const creditsRes = await fetch(`${baseURL}/movie/${movie.id}/credits?api_key=${apiKey}`);
    const creditsJson = await creditsRes.json();
    const videosRes = await fetch(`${baseURL}/movie/${movie.id}/videos?api_key=${apiKey}`);
    const videosJson = await videosRes.json();

    const director = creditsJson.crew.find(member => member.job === 'Director');
    const cast = creditsJson.cast.map(actor => actor.name).join(', ');
    const genres = detailsJson.genres.map(genre => genre.name).join(', ');
    const trailer = videosJson.results.find(video => video.type === 'Trailer' && video.site === 'YouTube');

    return {
      ...movie,
      director: director ? director.name : 'N/A',
      cast,
      genres,
      runtime: formatRuntime(detailsJson.runtime),
      vote_average: `${detailsJson.vote_average}/10`,
      original_language: getLanguageFullName(detailsJson.original_language),
      original_title: detailsJson.original_title,
      budget: detailsJson.budget ? `$${detailsJson.budget.toLocaleString()}` : 'N/A',
      trailer: trailer ? `https://www.youtube.com/watch?v=${trailer.key}` : null,
    };
  });

  movieList.value = await Promise.all(moviesWithDetailsPromises);
});

return {
  movieList,
};
},
};
```

Methodology for Svelte

Project directory:



Snippet from App.svelte:

```
onMount(async () => {
  const fetchMovies = async () => {
    const moviesPage1 = await getMoviesWithDetails(1);
    const moviesPage2 = await getMoviesWithDetails(2);

    const combinedMovies = [...moviesPage1, ...moviesPage2];


    const seenIds = new Set();
    const uniqueMovies = combinedMovies.filter(movie => {
      if (seenIds.has(movie.id)) {
        return false;
      }
      seenIds.add(movie.id);
      return true;
    });

    const moviesWithDetailsPromises = uniqueMovies.map(async (movie) => {
      const detailsRes = await fetch(`${BASE_URL}/movie/${movie.id}?api_key=${API_KEY}`);
      const detailsJson = await detailsRes.json();
      const creditsRes = await fetch(`${BASE_URL}/movie/${movie.id}/credits?api_key=${API_KEY}`);
      const creditsJson = await creditsRes.json();
      const videosRes = await fetch(`${BASE_URL}/movie/${movie.id}/videos?api_key=${API_KEY}`);
      const videosJson = await videosRes.json();

      const director = creditsJson.crew.find(member => member.job === 'Director');
      const cast = creditsJson.cast.map(actor => actor.name).join(', ');
      const genres = detailsJson.genres.map(genre => genre.name).join(', ');
      const trailer = videosJson.results.find(video => video.type === 'Trailer' && video.site === 'YouTube');

      return {
        ...movie,
        director: director ? director.name : 'N/A',
        cast,
        genres,
        runtime: formatRuntime(detailsJson.runtime),
        vote_average: `${detailsJson.vote_average}/10`,
        original_language: getLanguageFullName(detailsJson.original_language),
        original_title: detailsJson.original_title,
        budget: detailsJson.budget ? `$${detailsJson.budget.toLocaleString()}` : 'N/A',
        trailer: trailer ? `https://www.youtube.com/watch?v=${trailer.key}` : null,
      };
    });

    const moviesWithDetails = await Promise.all(moviesWithDetailsPromises);
    movieList.set(moviesWithDetails);
  };

  fetchMovies();
});
```

Notes:

React has more folders because, being more familiar with it, I modularised my initial working script into various files to make it more understandable, whilst, on the other hand, Svelte and Vue.js work entirely on App.svelte and App.vue.

Regarding the ease and comfortability of coding, the three frameworks were equally easy to work with, especially compared to my unsuccessful attempt with Angular. React was the first framework I made the webpage with, and, it is worth noting I am slightly biased because I already had some experience with React. I also really like the way both Svelte and Vue.js utilise HTML, JS and CSS in their .svelte and .vue files. For a webpage simple like mine, this was quite convenient.

Another note is that the Svelte code snippet seems longer because after a few days of finishing the webpage, I encountered a problem where the movies did not display due to an error about duplicate ids, which I fixed by adding lines 9-15 in the snippet.

Finally, before continuing to the deployment stage of my webpages, I would also like to point out the challenges I faced when attempting to create the app with Angular, which the reader may find beneficial.

Challenges With Angular

The first is that Angular uses TypeScript. I am not experienced enough to explain the technicalities of TypeScript, but it is basically a stricter form of JavaScript that checks your code for errors before it runs. You need to define types for variables, function parameters and objects, and if everything in your app is not perfect, TypeScript will throw errors. The Integrated Development Environment (IDE) I used, WebStorm, allows you to supress TypeScript errors, however, I still could not get anything to display on the webpage, and I did not know if it was because of the TypeScript errors or other problems with the Angular set up, which brings me to my next point.

Whilst attempting to get the webpage running, I was encountering many concepts like selectors, pipes, declarations, imports, providers, bootstrapping, ngIf and ngFor, in addition to having to configure routes and lot of boilerplate code. No matter how many changes I tried to make to my code files, the webpage continued displaying a blank page, and I simply felt overwhelmed by the environment, not knowing where the errors were coming from or how to fix them. I realised that Angular is better for more experienced developers who truly have a precise understanding of how things work in web development, in addition to the fact that it was too complex and simply not necessary for the small app that I was making.

A final downside regarding Angular is that its project directory is a lot more complicated than other frameworks'. The *app* folder inside the *src* directory has six files: *app.component.css*, *app.component.html*, *app.component.spec.ts*, *app.component.ts*, *app.config.ts* and *app.routes.ts*, in addition to three files in *src* folder itself: *index.html*, *main.ts* and *styles.css*. This is the standard setup for standalone components. There is then the option of using NgModules, and, if you are a beginner, you likely do not even know 1. what standalone components or NgModules mean, 2. which of the two is best for your type of app, and 3. how

to continue after choosing one of the two. An Angular project also has lot more more configuration and dependency files than normal (especially compared to React): *angular.json*, *package.json*, *package-lock.json*, *tsconfig.app.json*, *tsconfig.json, tsconfig.spec.json*, *.editorconfig*. Although these can eventually be understood, they simply add to the more complex feel of Angular which a beginner can feel overwhelmed by.

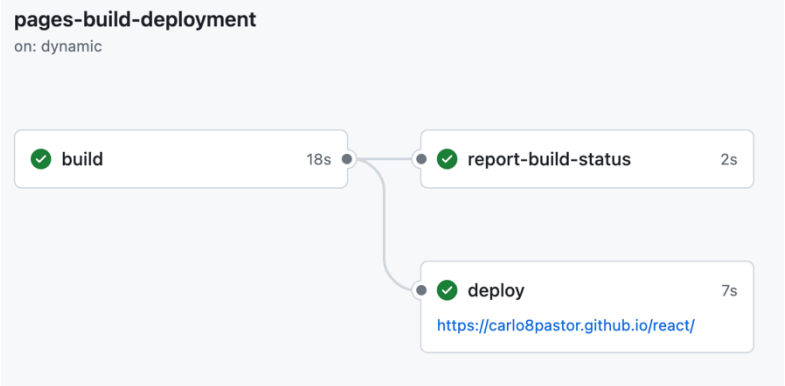
To summarise, I feel completely identified with a sentence from Matias Levlin's thesis, where he commented that the "commonly cited negative aspects of Angular were its perceived bloatedness, complexity, and heavy style of development, not being recommended for smaller development projects".

What is GitHub Pages?

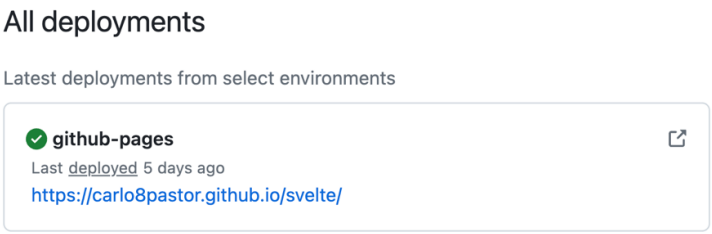
GitHub Pages is how I deployed my apps to the web. It is a static site hosting service that hosts web pages published from a user's GitHub repository. This is a good solution for developers who are already familiar with GitHub and would like to get their projects online in the shortest time possible without worrying about managing sever infrastructure. It supports HTML, CSS and JavaScript, making it ideal for hosting front-end aspects of web applications.

GitHub Pages offers a few advantages. The first is that it is pretty simple. The only thing you have to do is learn how to configure your project, depending on the framework, to be compatible with GitHub Pages, which I explain in the section after this one. In my case, I had to learn how to do it for React, Svelte and Vue.js, but most people will only have to learn how to do it with one framework, and all it takes is following a YouTube video. If you are familiar with git and your framework's terminal commands, deployment is pretty simple as it is only a sequence of these. You can then continue pushing changes to your repository or re-publishing the website with the same commands.

It works with GitHub Actions / workflows, which is also pleasant if you are already familiar with it. After making updates to your website and pushing the changes to GitHub, all you have to do is re-run the deploy terminal command to update the deployed website.



Moreover, the hosting service that GitHub Pages offers is free of charge if your repository is public, can host high traffic, provides HTTPS enforcement, and allows you to add a custom domain. In this study, just like TMDB API, GitHub Pages is also a control variable because my three websites being hosted in the same environment ensures that discrepancies in performance metrics are due to the frameworks themselves rather than the hosting environment.

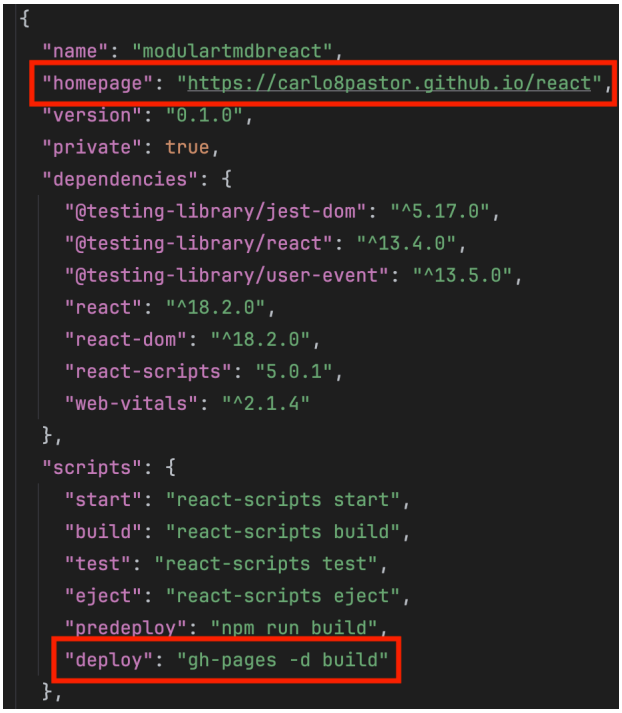


Configuring React With GitHub Pages

After finalising your project in the IDE of your choice, you can go ahead and create a repository for it using GitHub's user interface. To then connect your project to the repository, all you have to do is type the following commands in your project's terminal:

```
$ git init
$ git add .
$ git commit -m "first commit"
$ git remote add origin https://github.com/carlo8pastor/REPONAME.git
$ git push
```

Before deploying your webpage with GitHub pages, you then have to make sure it is compatible. For React, you have to add the following dependencies to the package.json file:



You can then finally run the following in your project's terminal:

```
$ npm run build
$ npm run deploy
```

Configuring Vue.js With GitHub Pages

For Vue.js and Svelte, the initial git and final npm commands remain the same. However, you have to make the following changes to your files:

vite.config.js:

```
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

// https://vitejs.dev/config/
no usages  Carlo
export default defineConfig( config: {
  base: '/vue/',
  plugins: [
    vue(),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    }
  }
})
```

package.json:

```
{
  "name": "vuea1",
  "version": "0.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "deploy": "gh-pages -d dist"
  },
  "dependencies": {
    "vue": "^3.4.21"
  },
  "devDependencies": {
    "@vitejs/plugin-vue": "^5.0.4",
    "gh-pages": "^6.1.1",
    "vite": "^5.1.6"
  }
}
```

Configuring Svelte With GitHub Pages

And the following for Svelte:

vite.config.js:

```
import { defineConfig } from 'vite'
import { svelte } from '@sveltejs/vite-plugin-svelte'

// https://vitejs.dev/config/
no usages  Carlo
export default defineConfig( config: {
  base: process.env.NODE_ENV === 'production' ? '/svelte/' : '/',
  plugins: [svelte()],
});
```

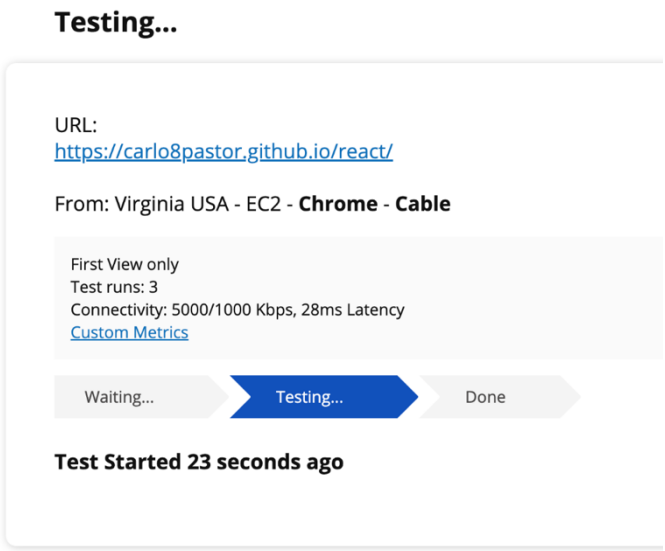
package.json:

```
{
  "name": "vite-svelte-starter",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview",
    "deploy": "npx gh-pages -d dist"
  },
  "devDependencies": {
    "@sveltejs/vite-plugin-svelte": "^3.0.2",
    "svelte": "^4.2.12",
    "vite": "^5.2.8"
  },
  "dependencies": {
    "gh-pages": "^6.1.1"
  }
}
```

What is webpagetest.org?

The final control variable in this project is webpagetest.org, the tool that I will be using to collect performance data on the websites. It is a very effective and user-friendly tool that lets you test webpages with the option of simulating the runs from a few specified locations, devices, browsers and connection types.

As seen in the *Results* section, in most cases, I will simulate a Chrome desktop browser accessed from the location of Dulles, Virginia, USA. The line simulates a 5 Mbps cable with 28 ms of latency, thus being able to simulate a fairly realistic environment of typical user experience under some moderate bandwidth limitations. I will also be carrying out a few mobile simulations, this being Chrome on an Emulated Motorola G (gen 4) tested from Virginia, USA on a 9 Mbps 4G connection with 170ms of latency.



After webpagetest.org runs your webpage three times, it provides a large array of information, including charts, a filmstrip view of your webpage loading in images or video, downloadable .json and. csv files, and more.

Performance Metrics

The specific performance metrics that will be shown in the results are the following:

Timings (ms):

- Visually Complete: The point in time that the content of the page has all visually rendered to the screen. This is important as it is when the user perceives the page as being ready to interact with.
- Last Visual Change: This is when the last visual change occurred during the loading of page. It helps in understanding when the page has become stable.
- Load Time (Onload): The time between the start of the page load to when the onload event is fired. This is important since it helps in the assessment of when all the objects on the page have fully loaded, including scripts and style sheets.
- Load Time (Fully Loaded): The time that has passed until all the resources of the page are loaded, and there has been no network activity for 2 seconds.
- DOM Content Loaded: Measures the time that it takes to load and parse the HTML document, including scripts that are meant to be executed after the parsing of the document, excluding stylesheets, images, and subframes.
- Speed Index: The average time at which visible parts of the page are displayed.
- Time to First Byte (TTFB): The time period from the submission of an HTTP request until the first byte of the response is received from the server by the browser; it generally represents server responsiveness.
- Time to Title: Time until the title of the page is displayed on the screen. The title is important since it tells the user they are on the correct page and serves as a progress indicator since the title loads faster than the remaining content, especially in my case (forty movies and their details).

- Time to Start Render: This is the time the browser takes before it starts painting content on the screen. It is quite important for the perception of speed by the user.
- Busy time CPU: Reveals how computationally expensive it is to load the page. It is the time in which the CPU is processing tasks, excluding idle time.
- 85%, 90%, 95%, 99% visually complete: The time it took the webpage to reach the visual completion percentages, assessed by webpagetest.org
- First Contentful Paint (FCP): The time at which the first text or image is painted.
- Largest Contentful Paint (LCP): The time until the largest text block or image has rendered, something that the user also typically understands as the webpage being complete or close to completion for interaction.
- Time to Interactive (TTI): The time to when the page becomes fully interactive, meaning even the event listeners attached to visible elements are prepared to take in inputs from the user.
- Total Blocking Time (TBT): The sum of all time intervals from FCP to TTI where the main thread was blocked for a duration long enough to prevent input responsiveness.

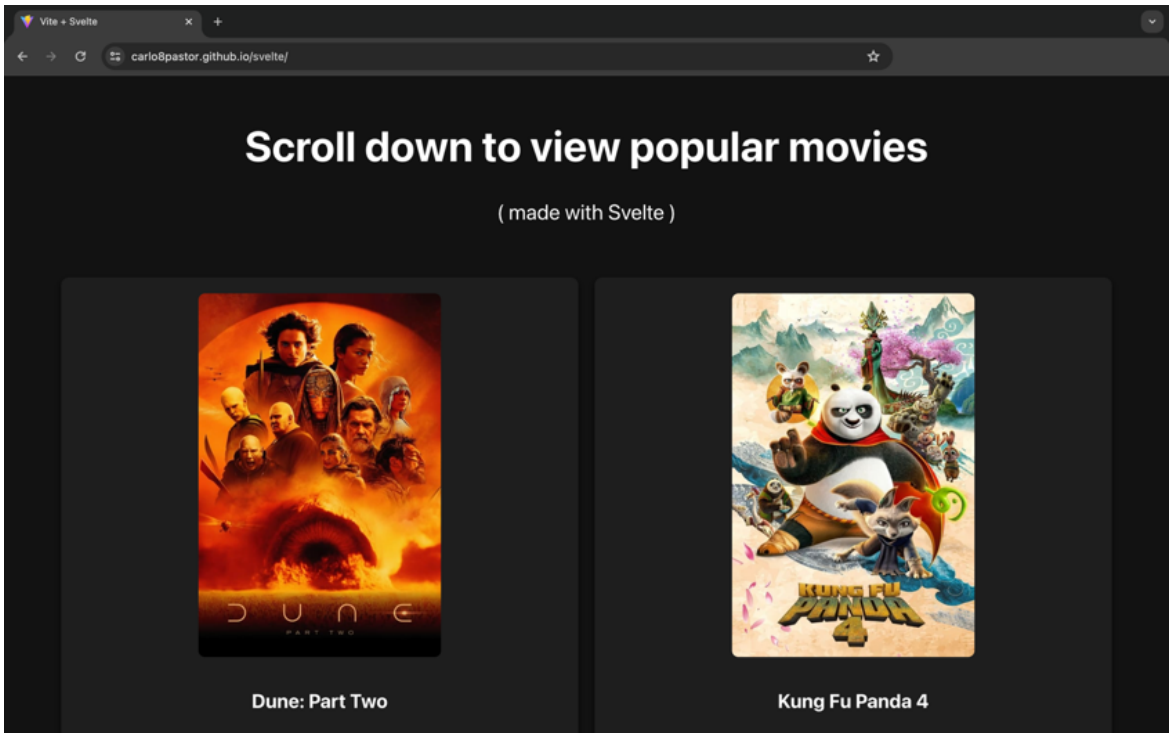
Layout Stability:

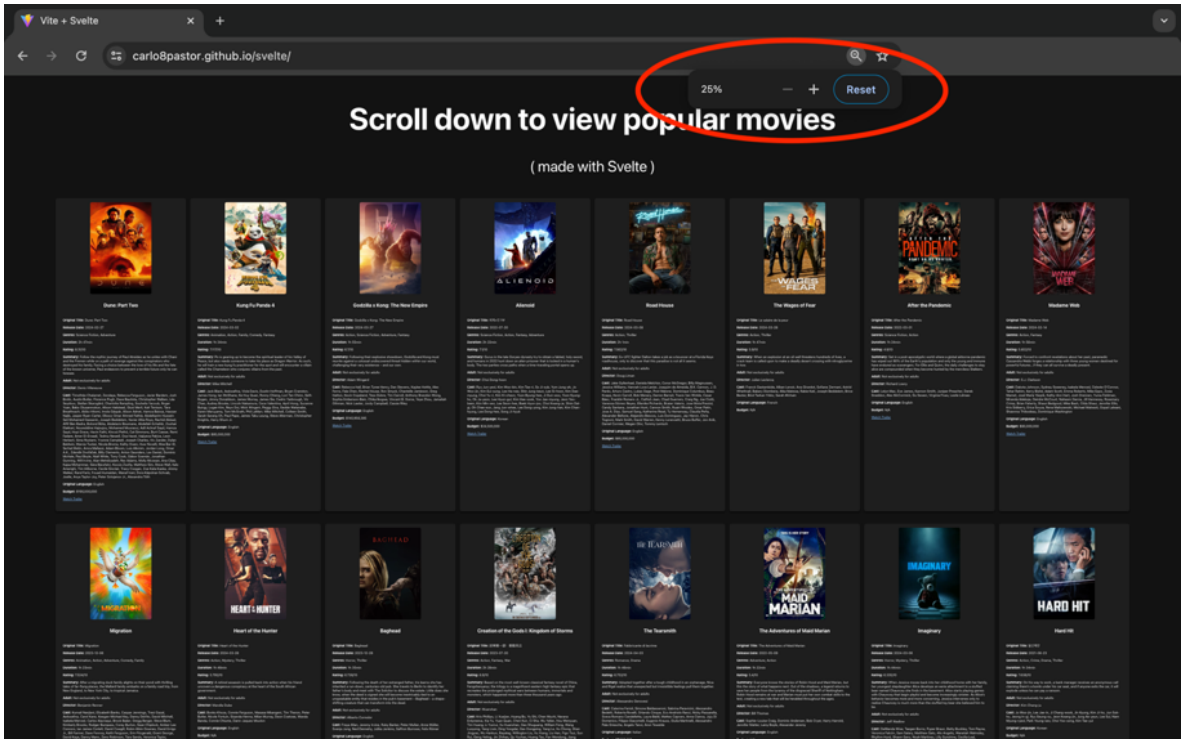
- CLS (Cumulative Layout Shift): A value from 0 to a positive number that represents the sum of all unexpected layout shifts that occur throughout the lifespan of the page. Closer to 0 means the less layout shifts. This metric is important for assessing the visual stability of the page.
- Layout Shifts: This is a similar metric provided by webpagetest.org that shows at what time since the page started to load the layout shifts occurred, in the form of a graph.

Resource Loading:

- Requests (Total, HTML, JS, CSS, Image, Flash, Font, Video, Other): This graph shows the amount and type of HTTP requests made by page.
- Bytes (Total, HTML, JS, CSS, Image, Flash, Font, Video, Other): This graph shows the byte size of the individual fetched resources that make up the page.

Before running the tests, the following are visual examples of what a user will see when loading one of the webpages:





As seen in the first image, once the screen loads, the user will only view the first two movie cards out of the forty. However, the webpage will load the forty cards all at once, so it will take a few seconds before the two visible cards render on the screen. In the second image, I have set the zoom of the webpage to 25%, showing sixteen out of the forty cards to better help the reader understand the size of the webpage.

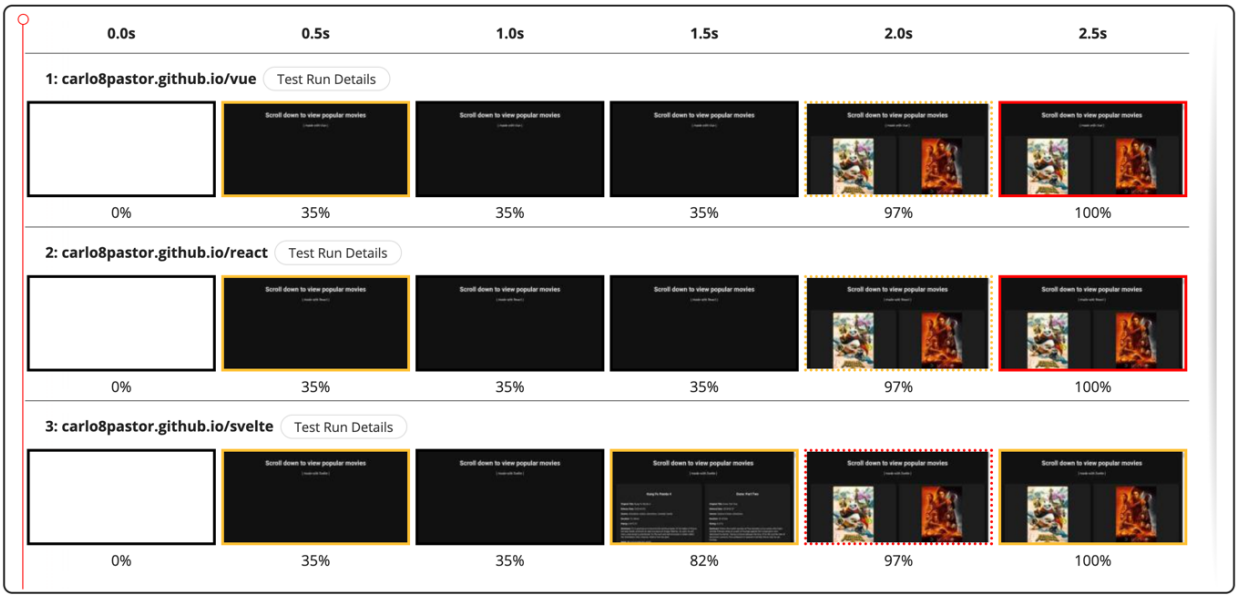
The following are the links to the webpages themselves:

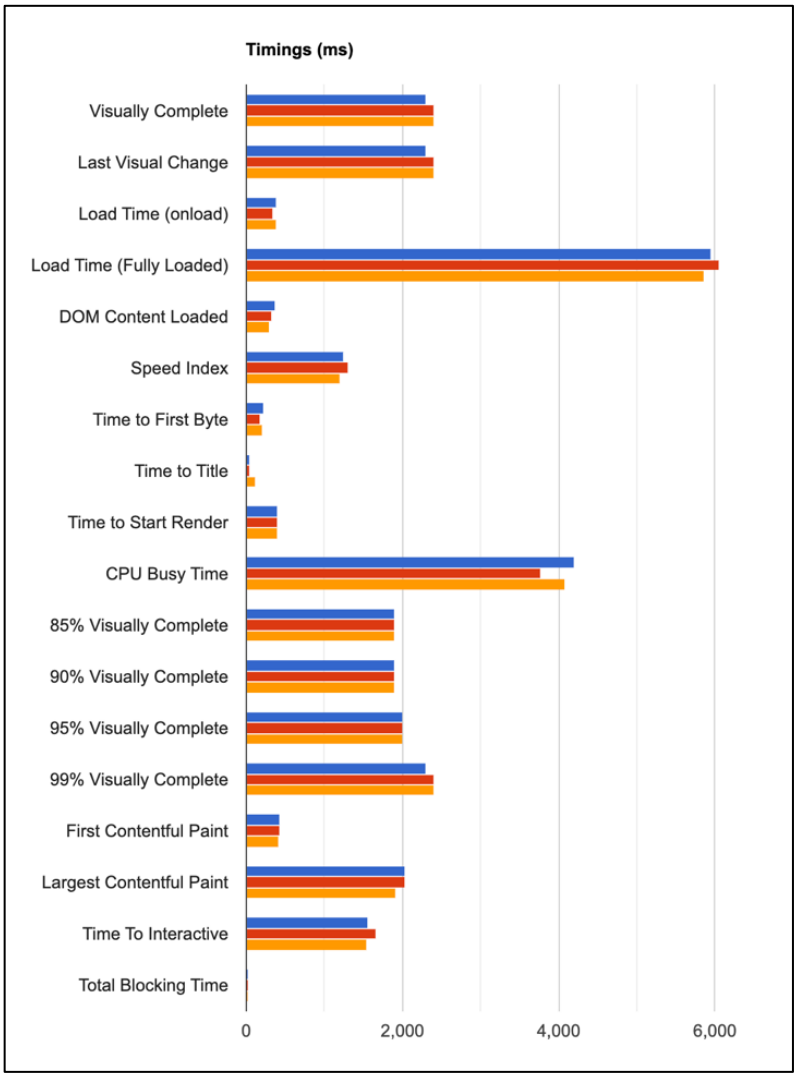
https://carlo8pastor.github.io/svelte/

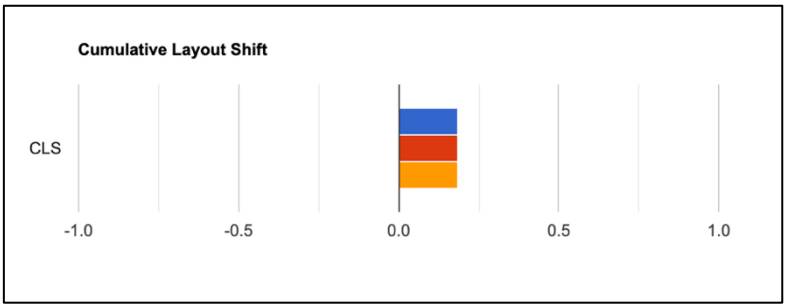
https://carlo8pastor.github.io/react/

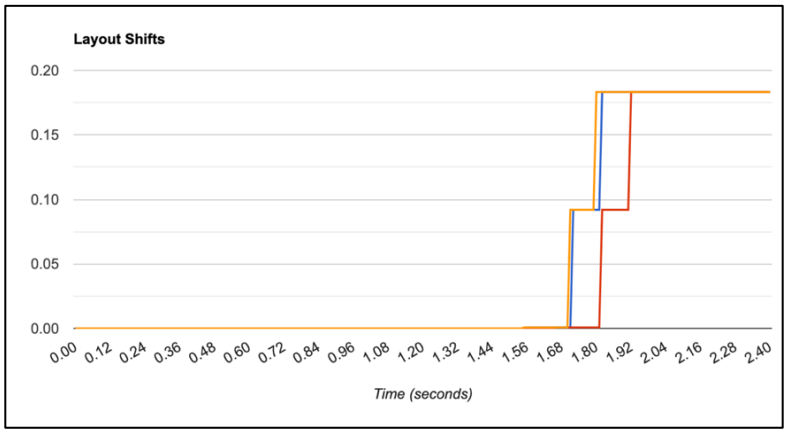
https://carlo8pastor.github.io/vue/

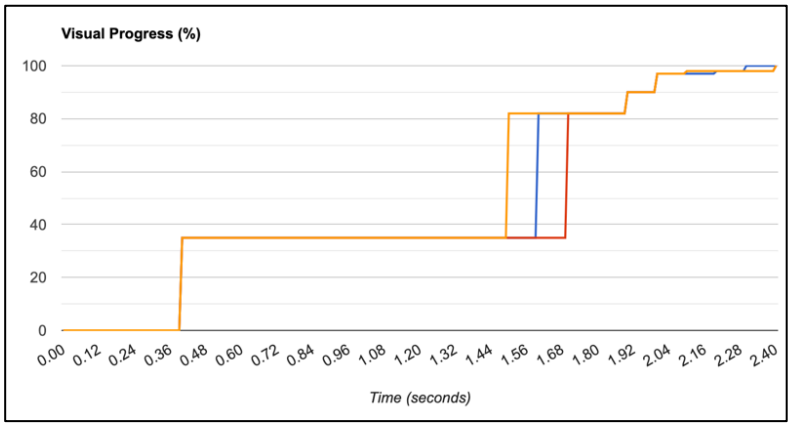
Results - First Comparison (Desktop)

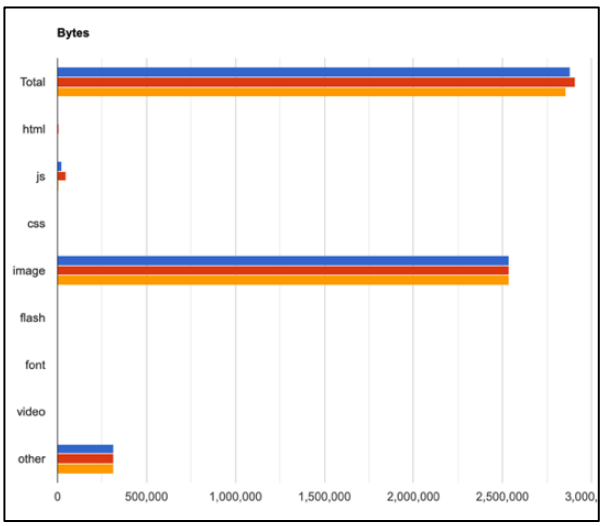


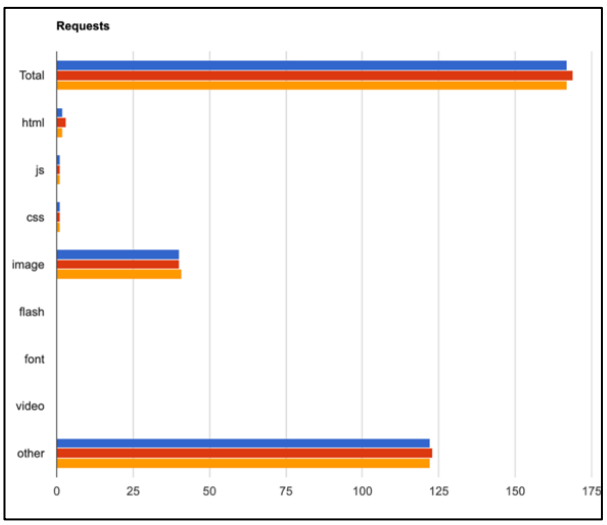



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0

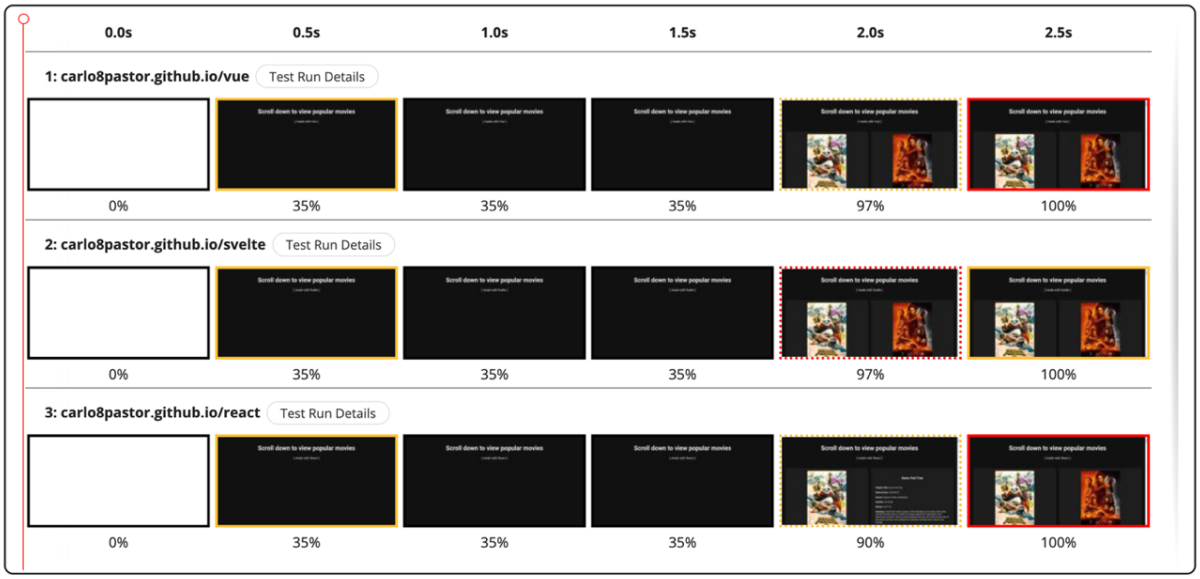

Layout Shifts
0.20
0.15
0.10
0.05
0.00
0.00 0.12 0.24 0.36 0.48 0.60 0.72 0.84 0.96 1.08 1.20 1.32 1.44 1.56 1.68 1.80 1.92 2.04 2.16 2.28 2.40
Time (seconds)

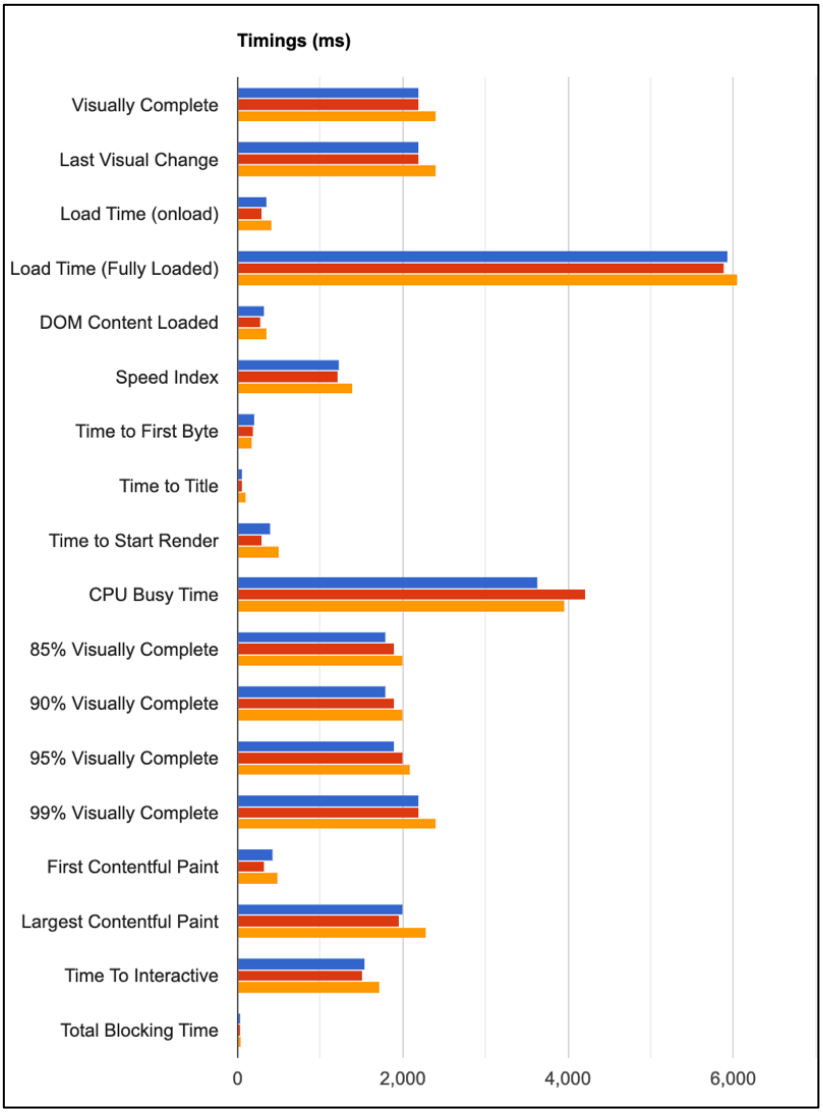

Visual Progress (%)
100
80
60
40
20
0
0.00 0.12 0.24 0.36 0.48 0.60 0.72 0.84 0.96 1.08 1.20 1.32 1.44 1.56 1.68 1.80 1.92 2.04 2.16 2.28 2.40
Time (seconds)

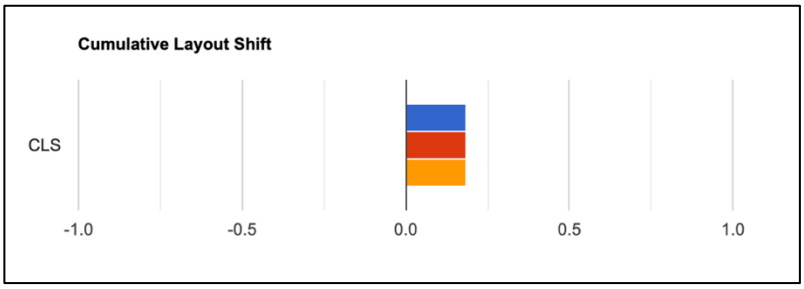

Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,

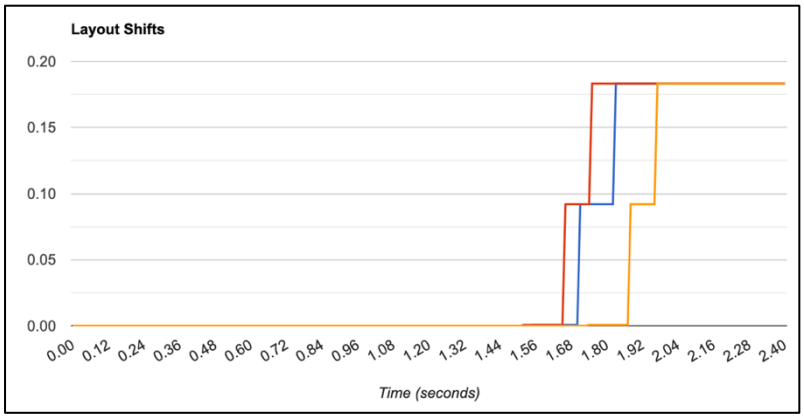

Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150
175

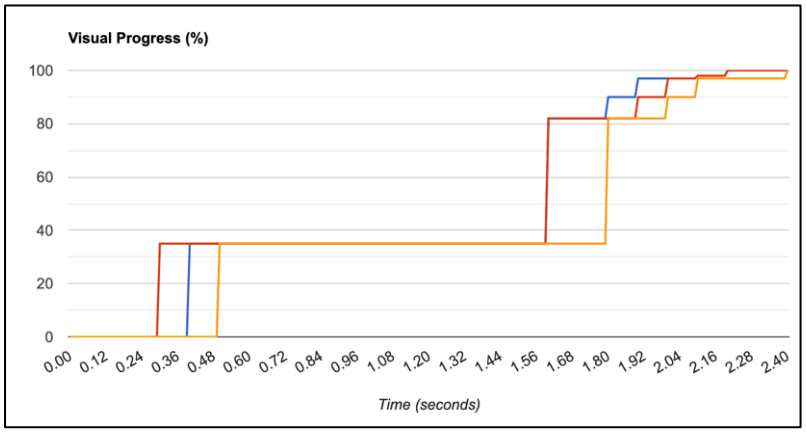
Second Comparison (Desktop)

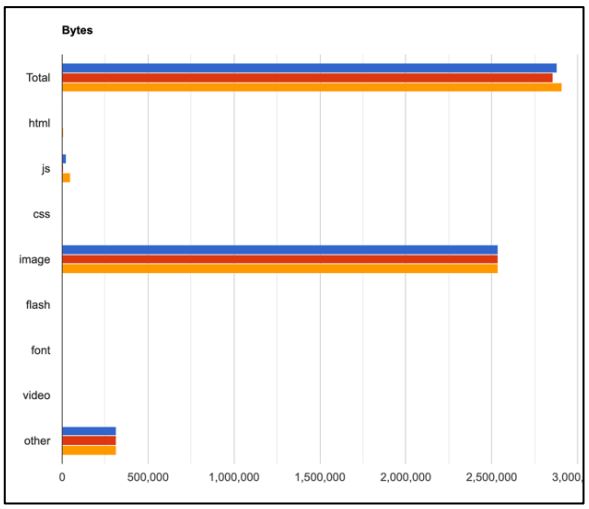


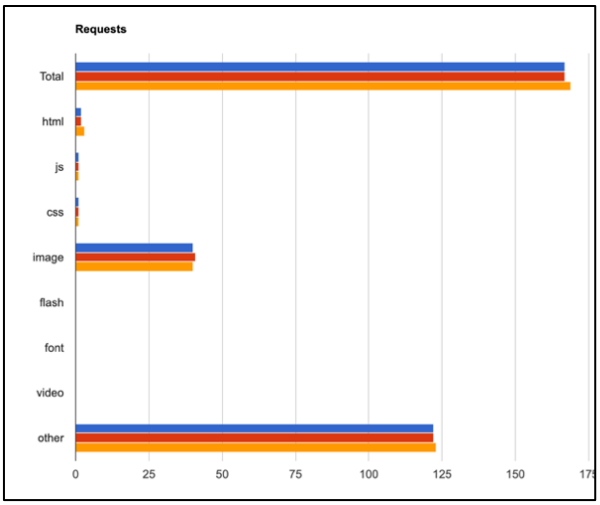



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0

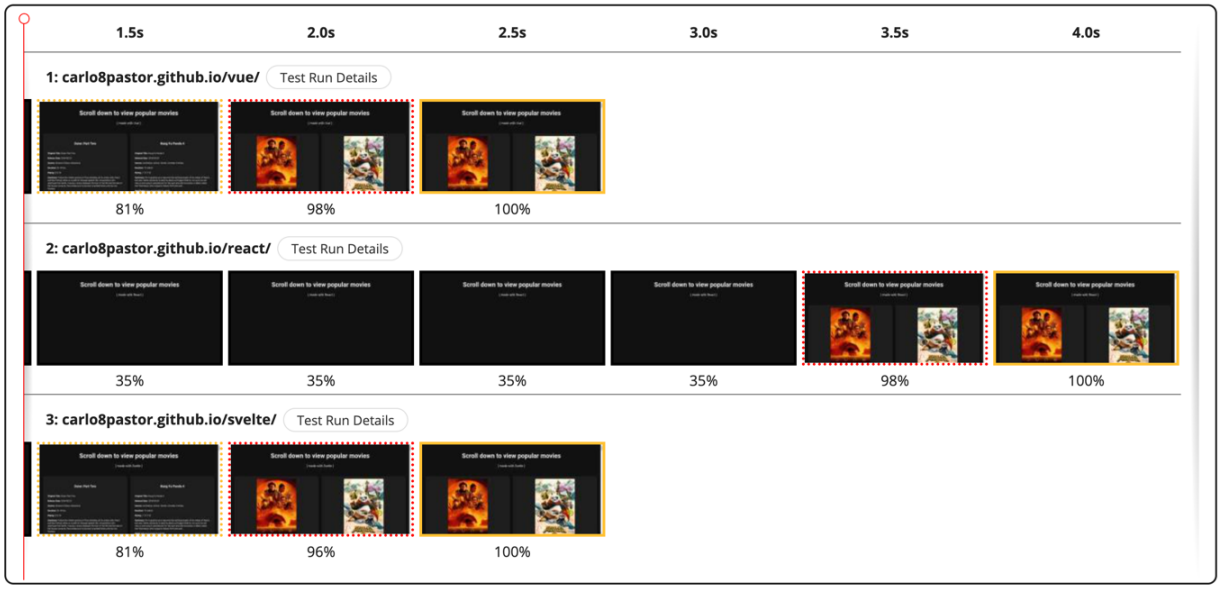

Layout Shifts
0.20
0.15
0.10
0.05
0.00
0.00 0.12 0.24 0.36 0.48 0.60 0.72 0.84 0.96 1.08 1.20 1.32 1.44 1.56 1.68 1.80 1.92 2.04 2.16 2.28 2.40
Time (seconds)

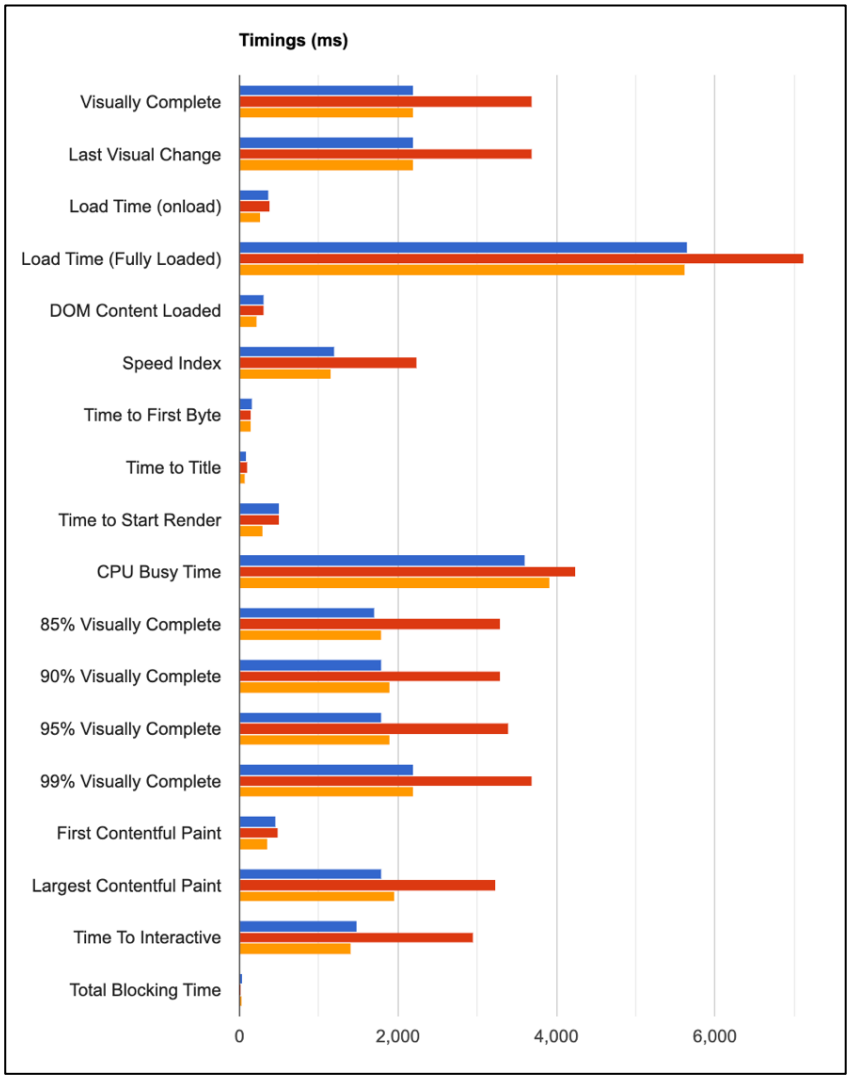

Visual Progress (%)
100
80
60
40
20
0
0.00 0.12 0.24 0.36 0.48 0.60 0.72 0.84 0.96 1.08 1.20 1.32 1.44 1.56 1.68 1.80 1.92 2.04 2.16 2.28 2.40
Time (seconds)

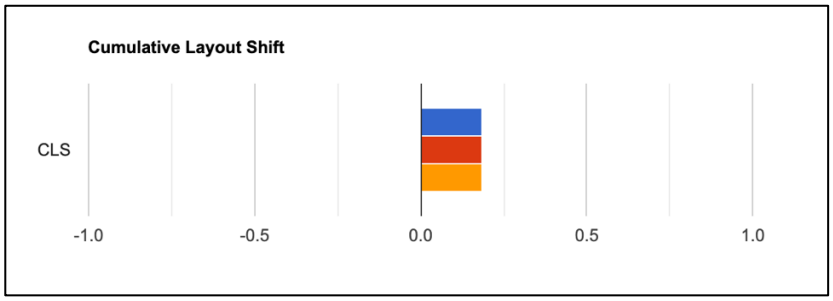

Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000

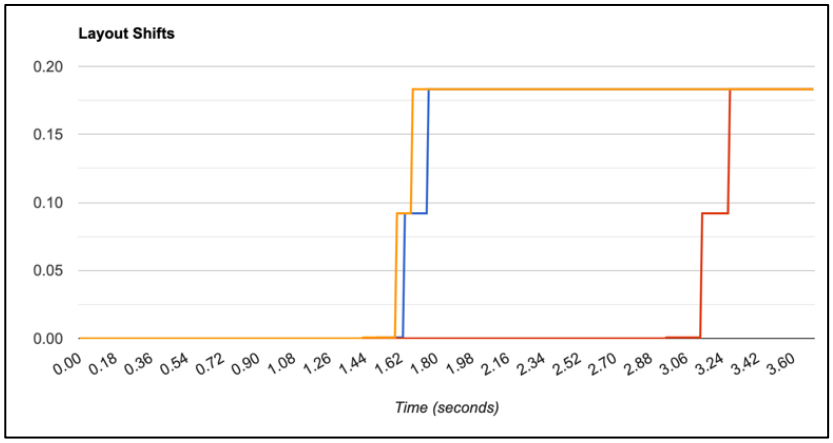

Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150

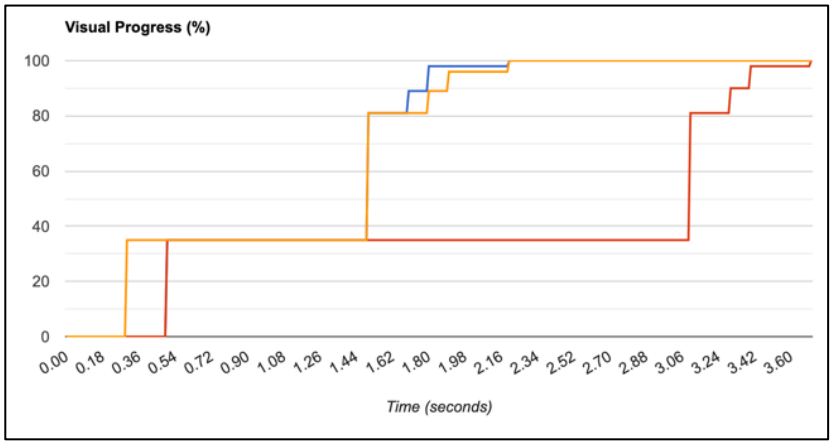
Third Comparison (Desktop)

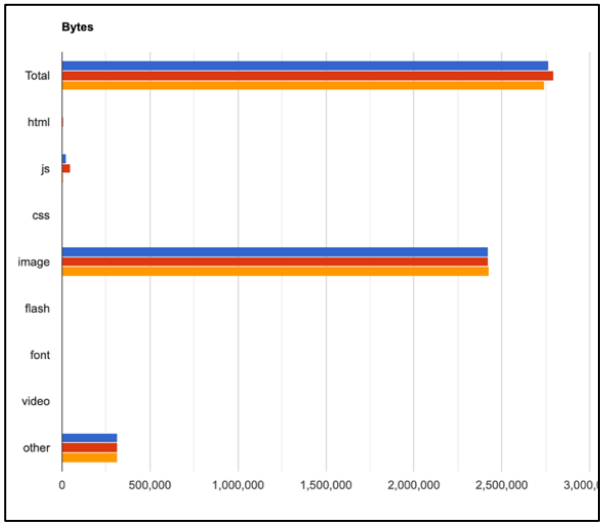


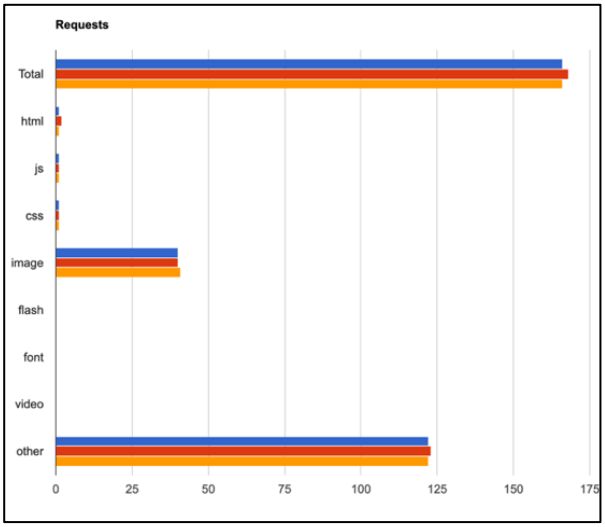



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0

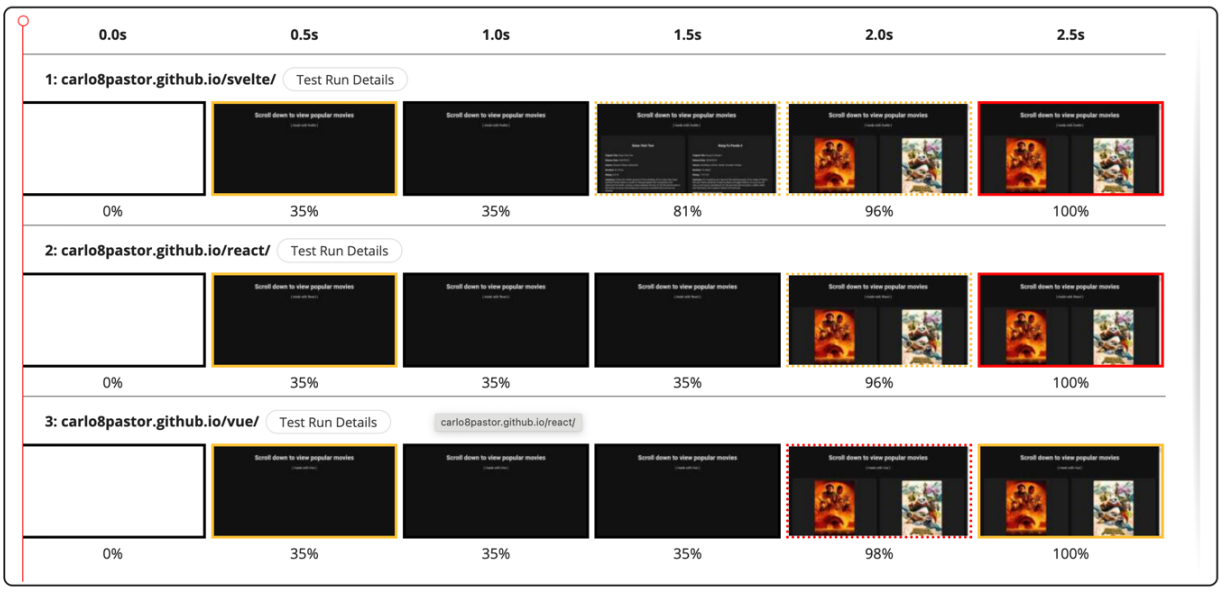

Layout Shifts
0.20
0.15
0.10
0.05
0.00
0.00 0.18 0.36 0.54 0.72 0.90 1.08 1.26 1.44 1.62 1.80 1.98 2.16 2.34 2.52 2.70 2.88 3.06 3.24 3.42 3.60
Time (seconds)

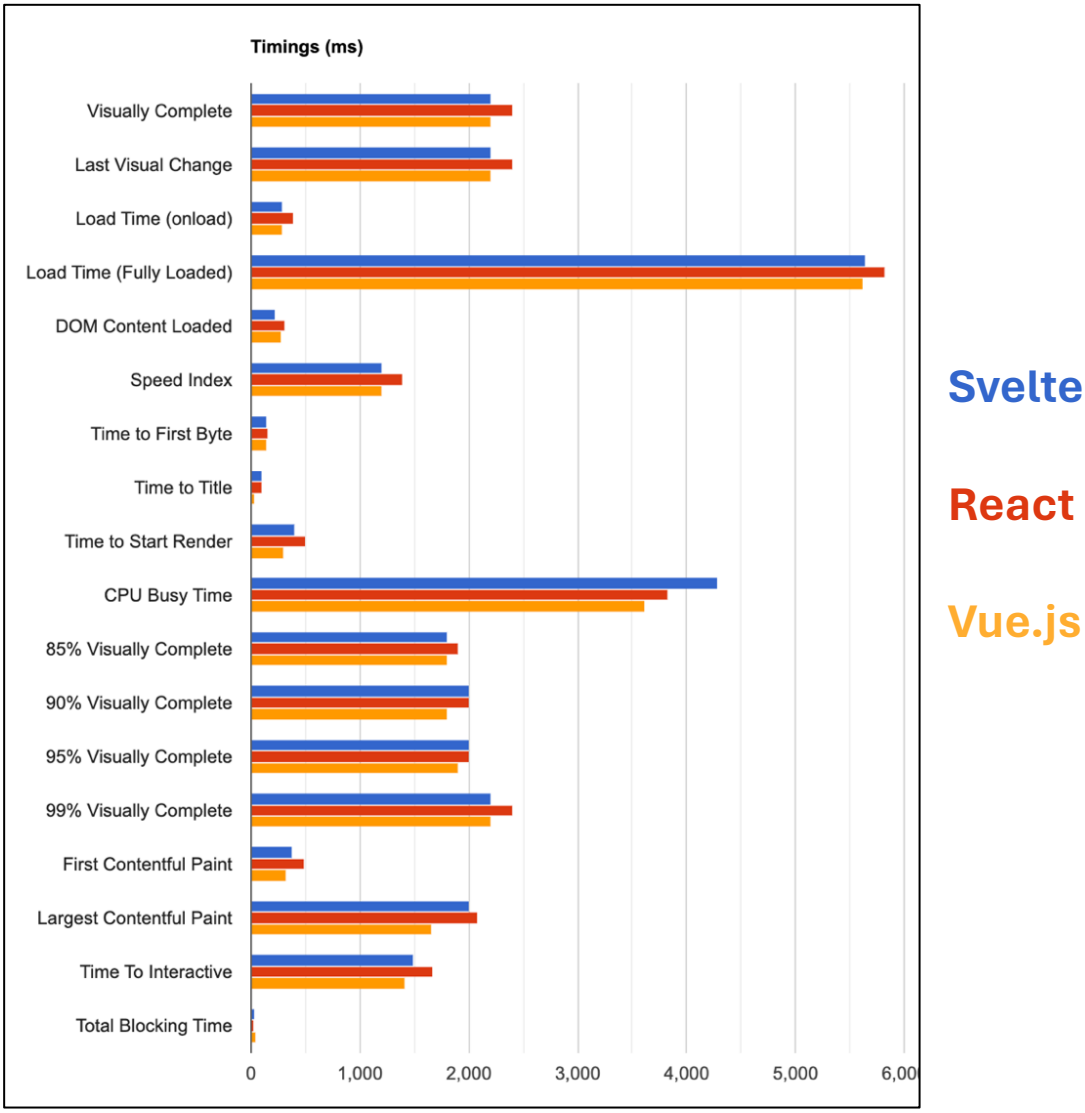

Visual Progress (%)
100
80
60
40
20
0
0.00 0.18 0.36 0.54 0.72 0.90 1.08 1.26 1.44 1.62 1.80 1.98 2.16 2.34 2.52 2.70 2.88 3.06 3.24 3.42 3.60
Time (seconds)

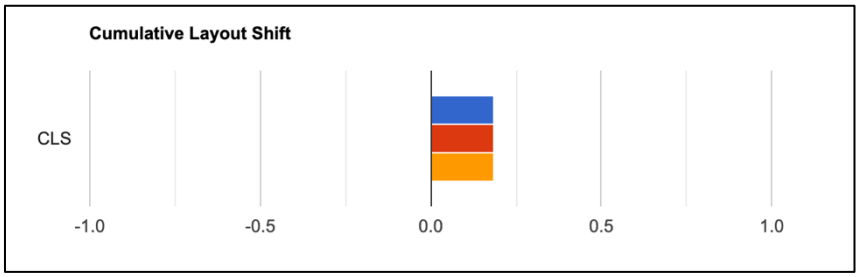

Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000

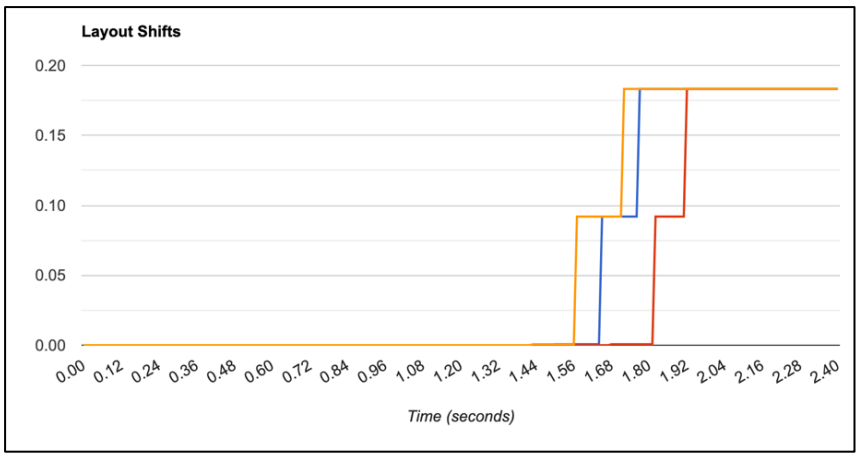

Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150
175

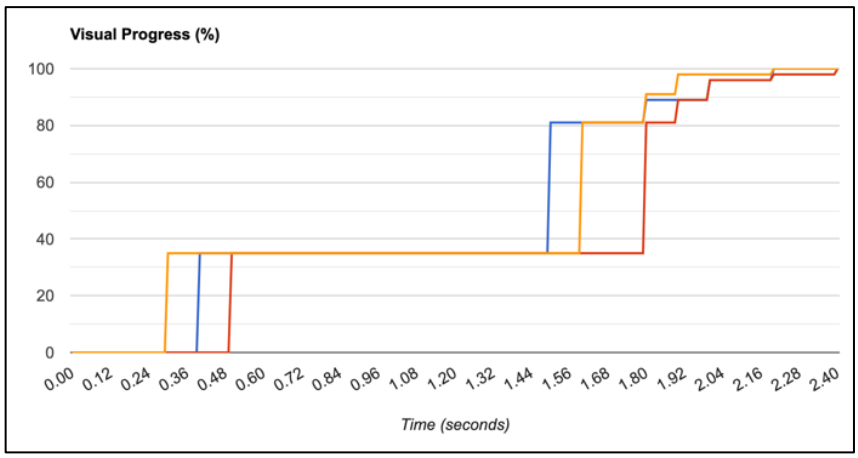
Fourth Comparison (Desktop)

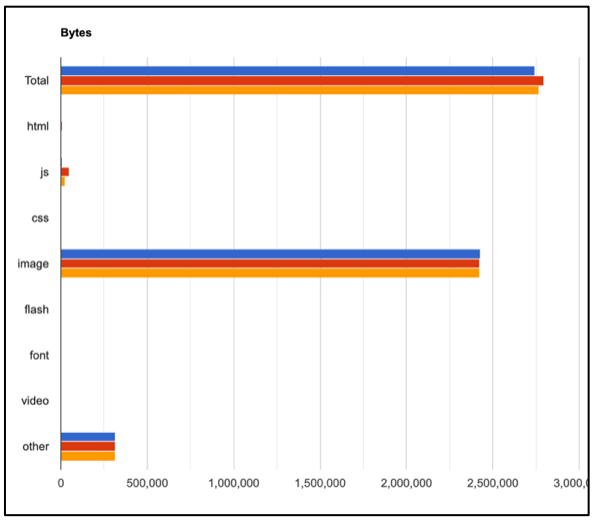


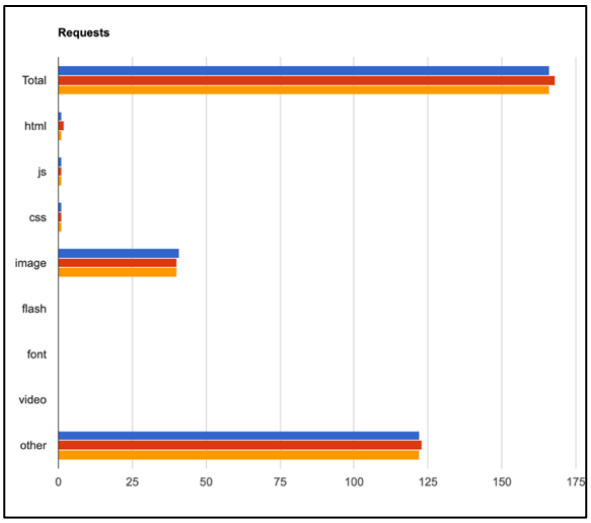



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0

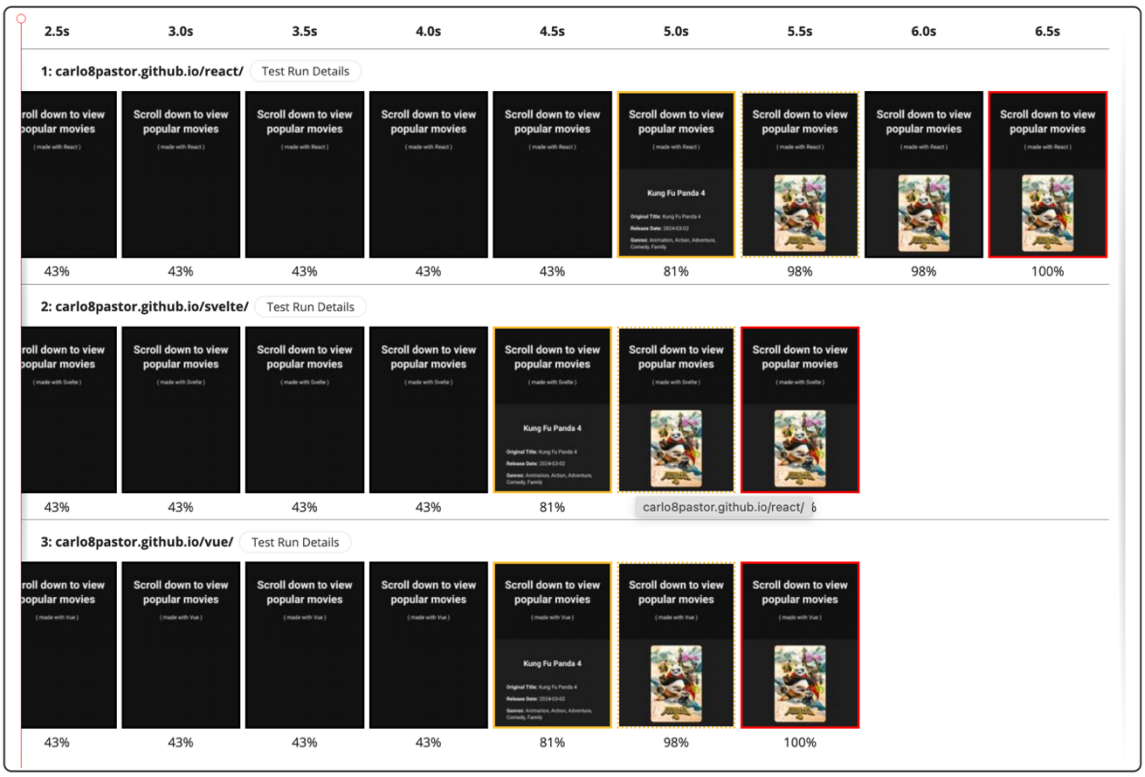

Layout Shifts
0.20
0.15
0.10
0.05
0.00
0.00 0.12 0.24 0.36 0.48 0.60 0.72 0.84 0.96 1.08 1.20 1.32 1.44 1.56 1.68 1.80 1.92 2.04 2.16 2.28 2.40
Time (seconds)

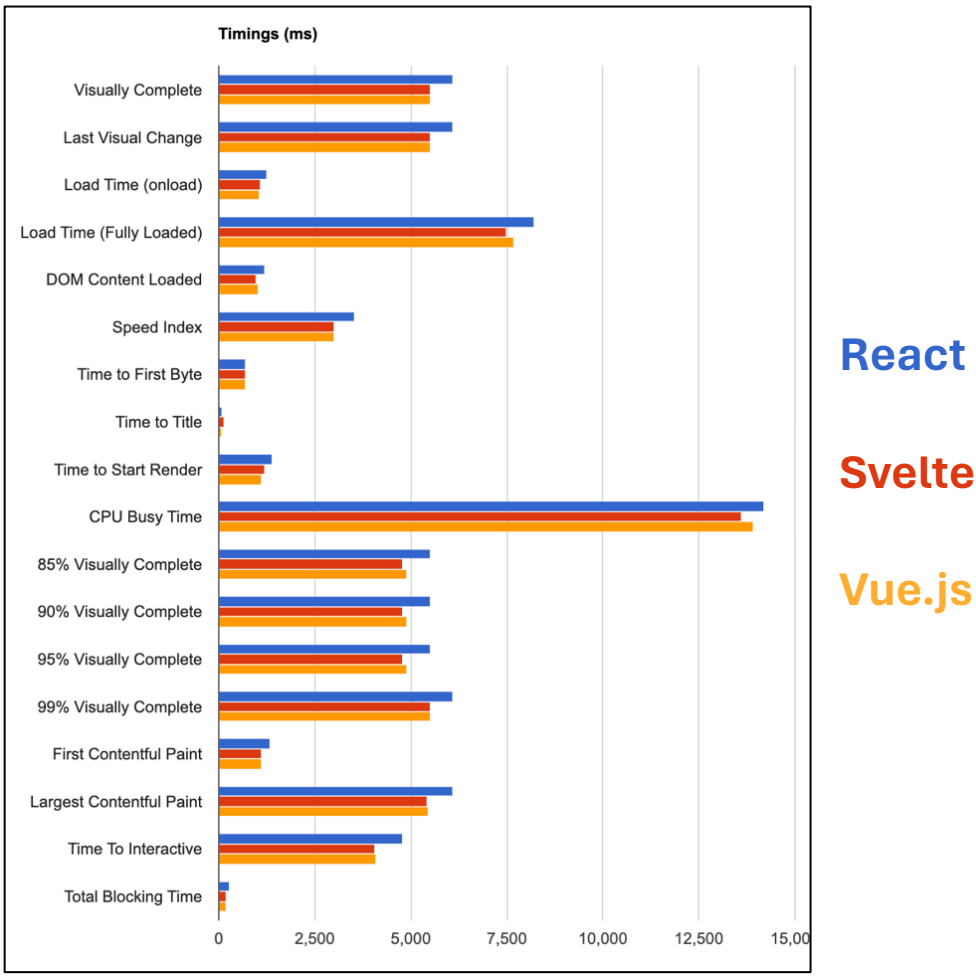

Visual Progress (%)
100
80
60
40
20
0
0.00 0.12 0.24 0.36 0.48 0.60 0.72 0.84 0.96 1.08 1.20 1.32 1.44 1.56 1.68 1.80 1.92 2.04 2.16 2.28 2.40
Time (seconds)

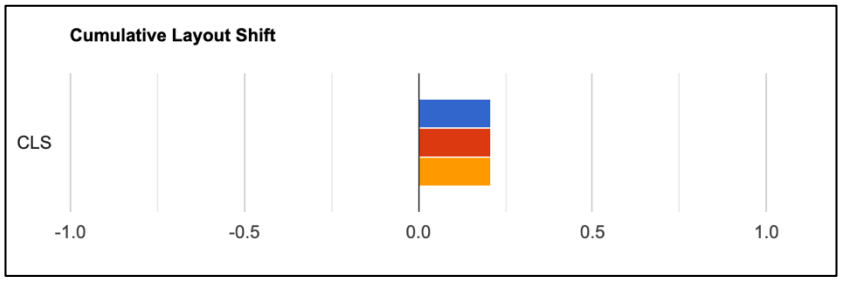

Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000

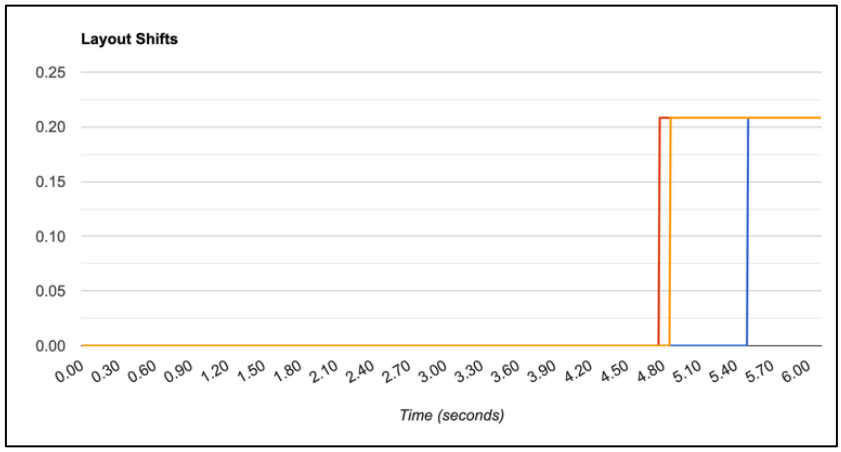

Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150
175

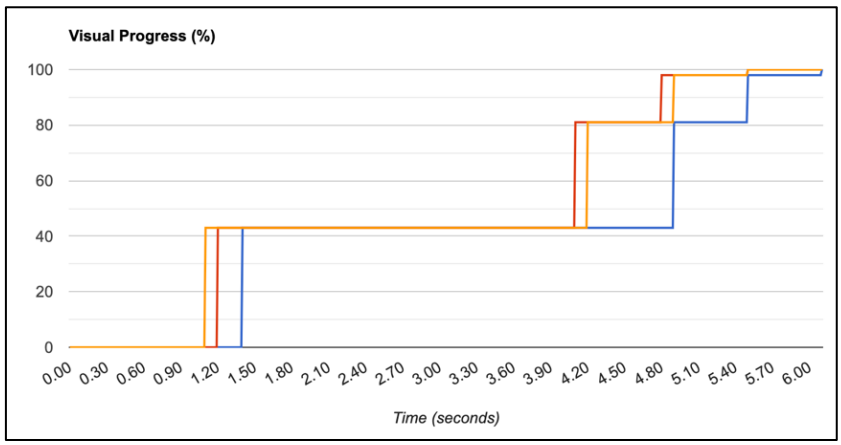
Fifth Comparison (Mobile)

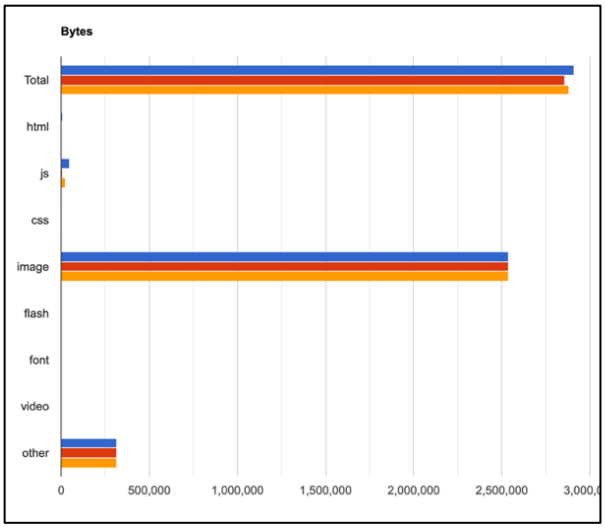


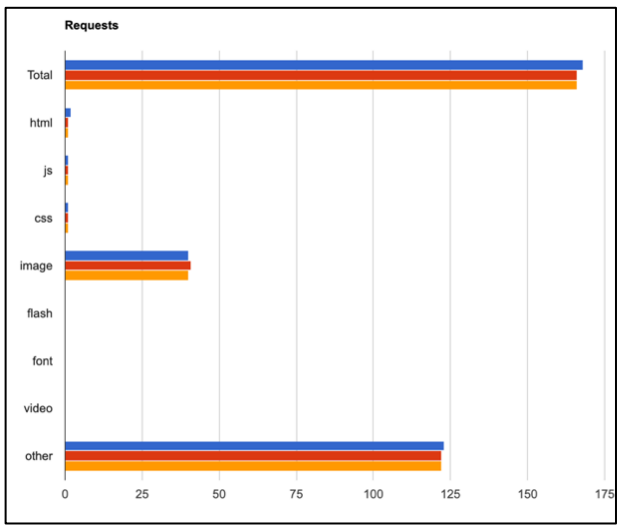



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0


Layout Shifts
0.25
0.20
0.15
0.10
0.05
0.00
0.00 0.30 0.60 0.90 1.20 1.50 1.80 2.10 2.40 2.70 3.00 3.30 3.60 3.90 4.20 4.50 4.80 5.10 5.40 5.70 6.00
Time (seconds)

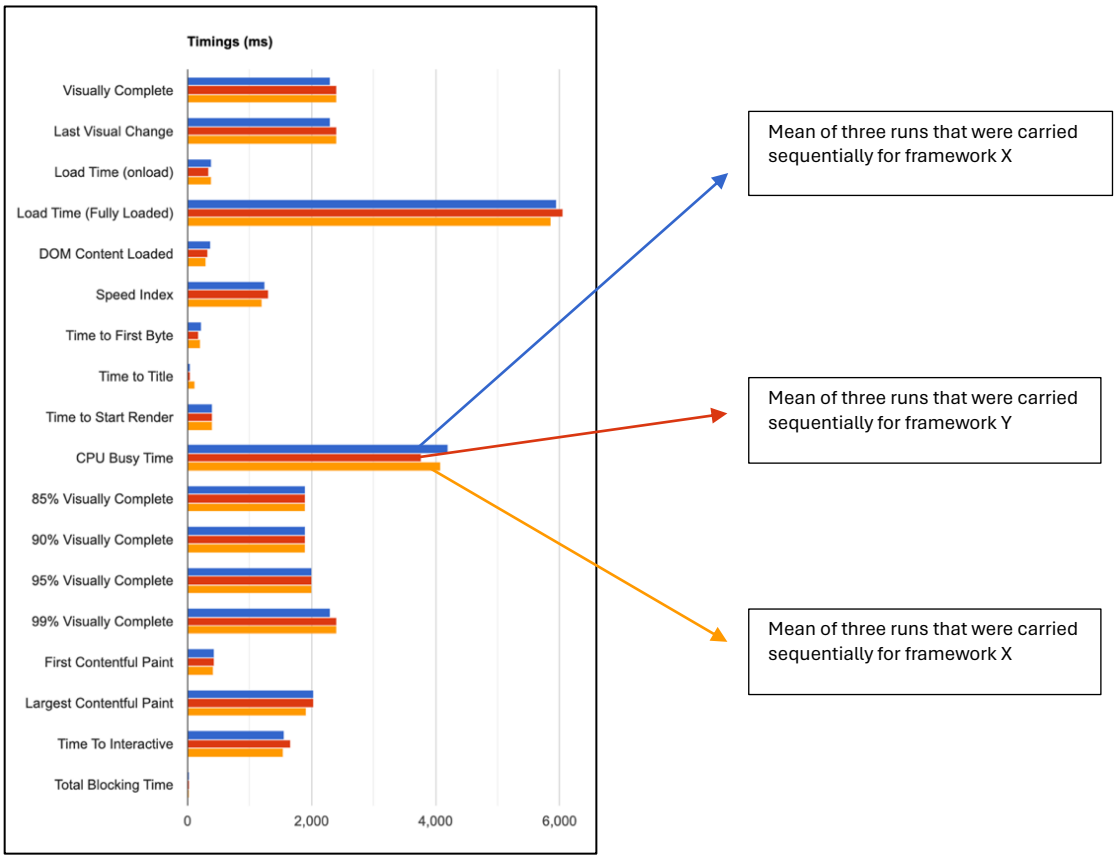

Visual Progress (%)
100
80
60
40
20
0
0.00 0.30 0.60 0.90 1.20 1.50 1.80 2.10 2.40 2.70 3.00 3.30 3.60 3.90 4.20 4.50 4.80 5.10 5.40 5.70 6.00
Time (seconds)


Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000


Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150
175

Note:

As can be seen, I have only included five comparisons. However, it is worth noting the following:



The metrics that are viewed in the graphs are not the metrics for the framework being run only once, they are the mean value taken from the framework being ran three different times. Thus, one comparison sections is not representative of only three runs (one for each framework), rather nine runs in total (three for each framework). This means that the five comparison sections I have included are representative of forty-five different runs in total.

Result Analysis

Desktop simulations:

As seen by the graphs, the frameworks performed quite similarly, with Load Times (Fully Loaded) bordering 6 seconds and LCP's averaging 2 seconds. However, I noticed that in the majority of the comparison's metrics, React was always slightly slower. Most importantly, you can see that in the Third Comparison, it was significantly slower with a load time of 7.131 seconds compared to 5.657s (Vue.js) and 5.623s (Svelte), and an LCP of 3.243s, compared to 1.793s (Vue.js) and 1.963 (Svelte).

Nonetheless, I ended up discovering that this was a problem with webpagetest.org. The way I ran the tests were: React first (3 runs), then Svelte (3 runs), and then Vue.js (3 runs), or, React first, then Vue.js, and then Svelte. I was always running React before the other frameworks. It turns out that for some reason, there is a factor in webpagetest.org that I have no control over, that makes the first-ran framework slower. It is not due to caching because if it were, the second and third runs of the same framework would be quicker, however they are also slow. The way I checked to confirm this was by testing Svelte and Vue.js before React, and, effectively, the same thing occurs. In the *Appendix*, I have included four more runs. You can see that, in the Sixth Comparison and Eighth Comparison, Svelte was the poor performer because I ran it first, and in Seventh Comparison Vue.js was the poor performer because I ran it first.

In the body of this thesis, the five comparisons included were all ran with React first. It seems that only in the third comparison react was largely affected by this uncontrollable webpagetest.org factor. Either way, I have been running tests throughout the whole month of April, and the general results, excluding these outliers, are that the performance metrics are always similar - again, bordering 6s for LT and averaging 2s for LCP.

Mobile simulations:

The Fifth Comparison is the only mobile simulation included in the body of this work. As can be seen, the performance metrics of the frameworks continue to be similar, around 8s for the Load Time (Fully Loaded) and 5.5s for the LCP. I have also included a Ninth Comparison in my appendix, which corroborates these results, and it was tested weeks after the Fifth Comparison, adding reliability. In the Ninth Comparison, you can see that Svelte was also negatively affected by the webpagetest.org unknown factor, causing it to be slightly slower in most metrics since it is the framework that I ran first. However, this does not take away from the fact that the results for mobile remain the same. Additionally, I have a third mobile simulation on my webpagetest.org account that also confirms the frameworks perform similarly, at 8s LT and 5.5s LCP results.

Reliability:

I carried out many tests for the desktop simulation. On my webpagetest.org account, 21 of these tests are saved, meaning that I have a record of the frameworks being ran 63 different times. I also ran tests outside of my webpagetest.org profile to try to account for differences, meaning that the results are backed by more than 63 different runs.

The same applies to the mobile simulation, except that I ran less tests on my account, 9 instead of 21, meaning that the frameworks were ran 27 times in total. This also brings reliability to the mobile results. Additionally, I ran both the desktop and mobile tests at different time periods during the month of April to try account for any possible disparity that could be caused by webpagetest.org. In other words, the tests were ran at intervals, not on the same day.

Finally, all of the control variables mentioned throughout the paper – using the same movie API service and same manner of deploying the webpages to the web (GH pages), help bring validity to the results.

Limitations:

Of course, this study has limitations. The first is that I am assuming that the results that webpagetest.org are providing are accurate. It is a third-party tool that I am not affiliated with and have no true control over to check reliability and authenticity.

Next, the conclusions of this thesis are only applicable to the following versions of the frameworks (the versions I made the webpages with): React: 18.2.0, Svelte: 4.2.12 and Vue.js: 3.4.21. Since versions are continually updated, the conclusions of this study may not be representative of future versions of the same frameworks.

Finally, the conclusions I drew from the testing are only specific to the type of app that I used – a simple webpage displaying 40 movies. As mentioned in similar studies from the literature review, results could vary greatly with an app of different complexity or scale.

Technical notes:

It is unfortunate that I found out about webpagetest.org's unknown factor after having carried out the majority of the tests, however, it is clear that the frameworks all perform very similarly. To be very precise, Svelte was objectively the fastest in Load Time (Fully Loaded), though not astonishingly more than the others. If looked at closely, in the Bytes graphs, the total bytes loaded for Svelte are always lower. In particular, it loads significantly less JS resources (around 5,000 bytes) than React (around 47,000) and Vue.js (around 23,000), where the bar chart for these 5,000 resources appears as 0 to the eyes. This is due to Svelte's unique approach that was explained when discussing the framework. Following this logic, Vue.js remains the second fastest in Load Time, and React the third.

Conclusion

After having 1. Successfully created the same movie webpage with React, Vue.js and Svelte, 2. Deployed them to the web, and 3. Analysed their rendering performance metrics, the conclusion is that:

- All three are quite easy to learn and have large communities where you can find support on forums and YouTube.
- All three perform very similarly in the case of my tested webpage.

Thus, addressing the initial dilemma of a beginner developer being overwhelmed by the abundance of frameworks, my proposed solution would be to randomly pick between either of the three. This prevents you from getting stuck at the thought of all the options and allows you to 'get the ball rolling', and, if after some time you decided to switch framework, the knowledge gained from having started with one of them would allow you to do so easily. Angular is another popular option, however, as detailed in the *Challenges with Angular* section, it may be more challenging to learn and not necessary for a simple webpage.

Now that I am more familiar with the frameworks, my next attempt regarding a study in this topic would be to 1. Make my movie webpage more complex – adding different pages (turning it into a website) and a lot more data, and carry out the same testing, or 2. Test different types of performance metrics in this same webpage or a new one. Being as popular as they are, my guess is that the framework's performances would still continue to be similar and not show any differences that would be noticeable to a typical user. I would thus have to conduct more investigation to be able to make an app where I can really squeeze out the framework's individual pros and cons. Additionally, if I gain more knowledge and become comfortable with Angular, I would also include it in the studies due to its popularity and significant market share.

Finally, this paper should not be taken as the be all and end all for web framework performance, rather one that is informative and can be learned from, and, since webpagetest.org is freely available on the web, the reader is welcome to use webpagetest.org to test my webpages themselves and view the resulting performance metrics first-hand. Due to webpagetest.org's unknown factor that negatively affects the first tested framework, I strongly recommend that the reader tests the first framework two separate times, and only takes the second test into consideration.

References

Main Body:

Fetisov, E., & Talochka, A. (2023, April 11). *Top 30 Companies Using ReactJS Development*. JayDevs. https://jaydevs.com/top-companies-using-react-js/

Frequently Asked Questions. Vue.js. (n.d.-a). https://vuejs.org/about/faq

Harris, R. (2016, November 26). *Frameworks without the framework: Why didn't we think of this sooner?* Svelte. https://svelte.dev/blog/frameworks-without-the-framework

Hooks at a glance. React. (n.d.). https://legacy.reactjs.org/docs/hooks-overview.html

Hussain, S. (2024, April 21). *Mastering vue.js: A comprehensive guide to Vue Cli, Vuex, and Vue Router*. Medium. https://medium.com/cloud-believers/mastering-vue-js-a-comprehensive-guide-to-vue-cli-vuex-and-vue-router-1258e0173843

Hámori, F. (2022, May 31). *The history of react.js on a timeline*. RisingStack Engineering. https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/

Markham, S. (2024, April 23). *Rich Harris on why he created Svelte*. The OfferZen Community Blog. https://www.offerzen.com/blog/rich-harris-on-why-he-created-svelte

Rendering Mechanism. Vue.js. (n.d.-b). https://vuejs.org/guide/extras/rendering-mechanism

Svelte. (n.d.). https://svelte.dev/docs/

Template Syntax. Vue.js. (n.d.-c). https://vuejs.org/guide/essentials/template-syntax.html

Literature Review:

Alexander, S. (2015). Speed Performance Comparison of JavaScript MVC Frameworks.

Bielak, K., Borek, B., & Plechawska-Wójcik, M. (2021). Web application performance analysis using Angular, React and Vue.js frameworks. *Journal of Computer Sciences Institute*, *23*, 77-83.

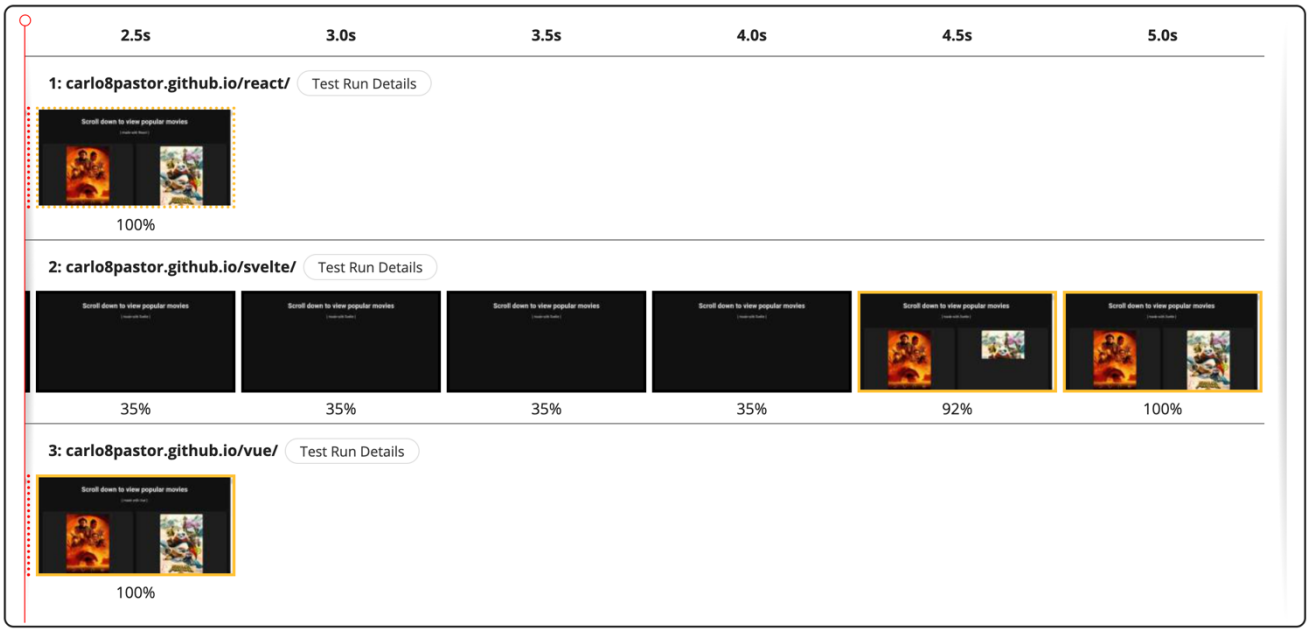
Levlin, M. (2020). DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte.

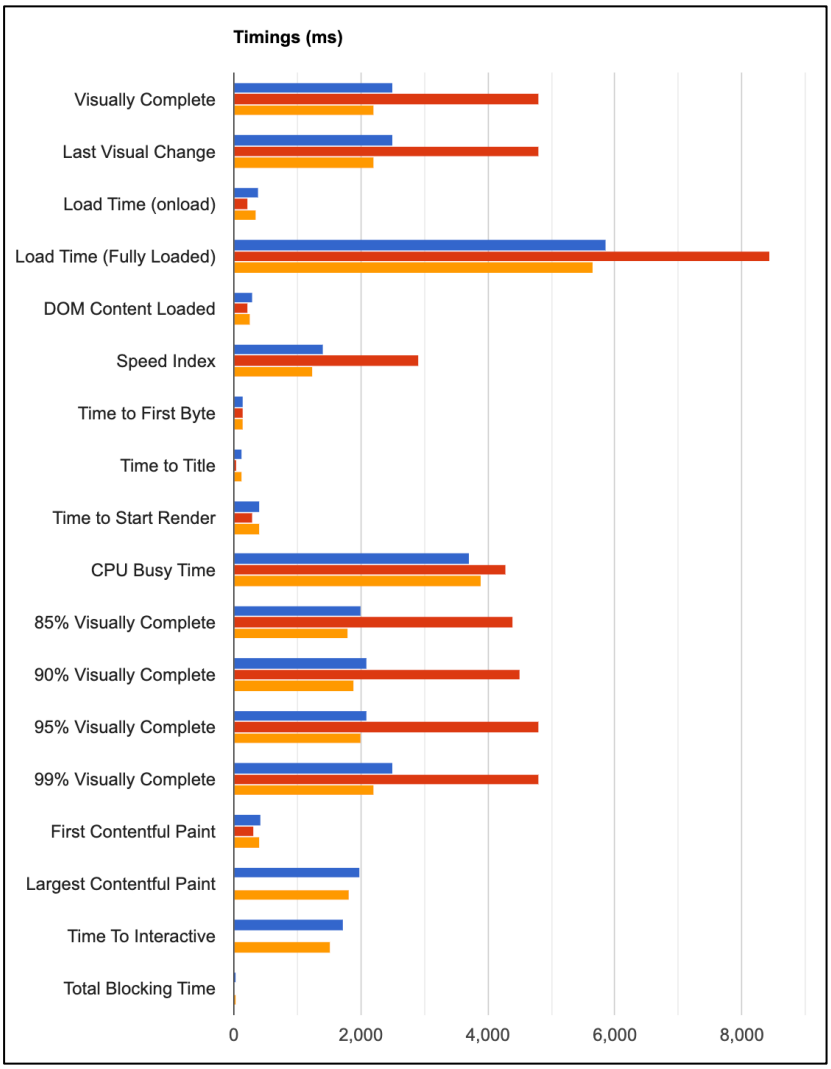
Diniz-Junior, R. N., Figueiredo, C. C. L., Russo, G. D. S., Bahiense-Junior, M. R. G., Arbex, M. V., Dos Santos, L. M., ... & Giuntini, F. T. (2022, October). Evaluating the performance of web rendering technologies based on JavaScript: Angular, React, and Vue. In *2022 XVLIII Latin American Computer Conference (CLEI)* (pp. 1-9). IEEE.

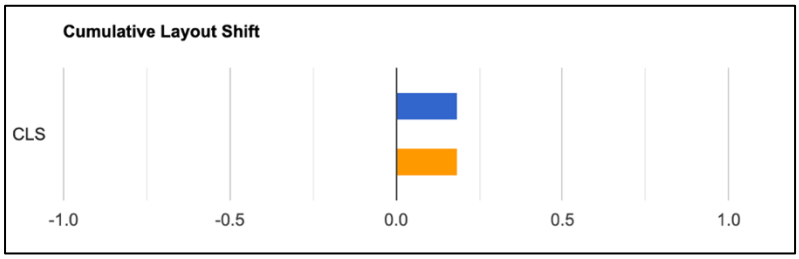
Saks, E. (2019). JavaScript Frameworks: Angular vs React vs Vue.

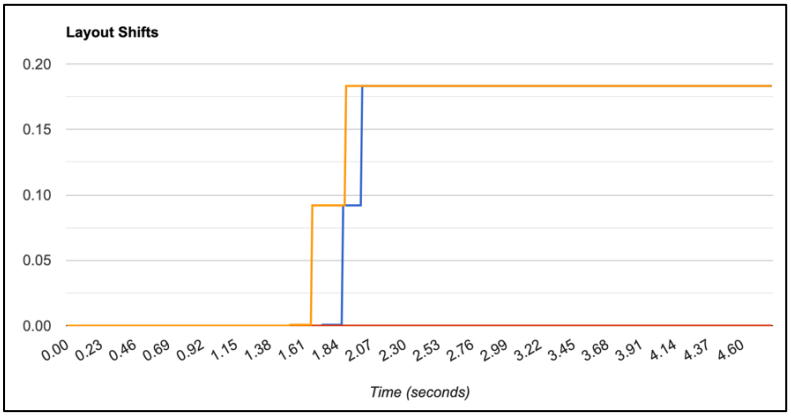
Siahaan, M., & Kenidy, R. (2023). Rendering performance comparison of react, vue, next, and nuxt. *Jurnal Mantik*, *7*(3), 1851-1860.

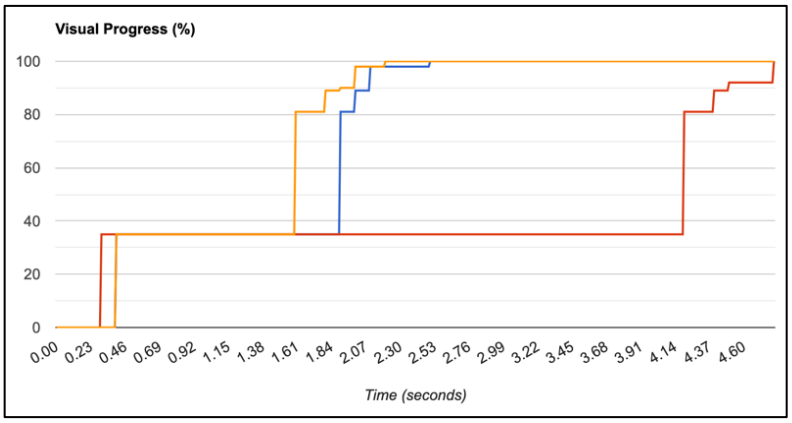
Appendix – Sixth Comparison

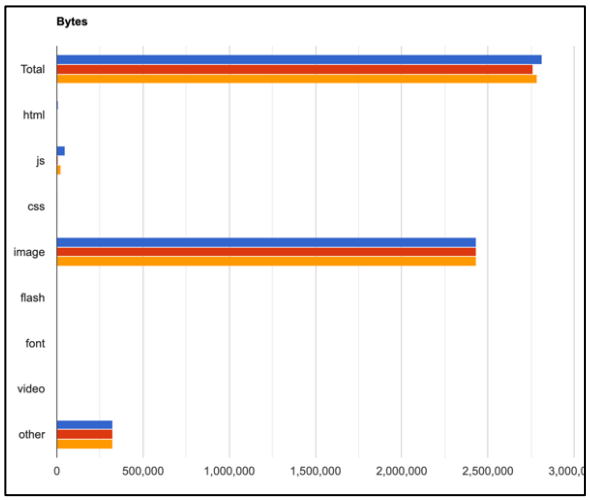


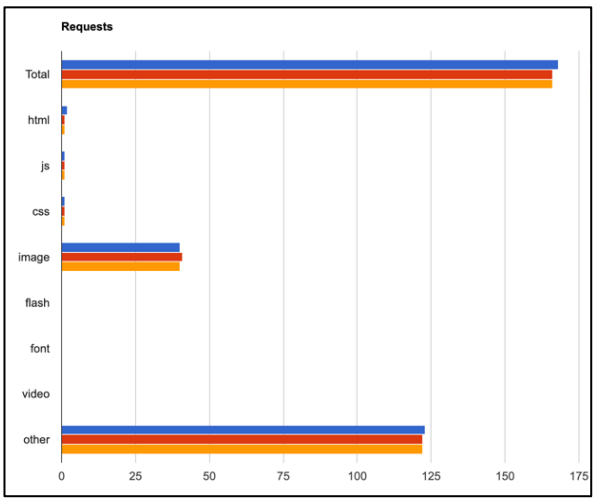



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0

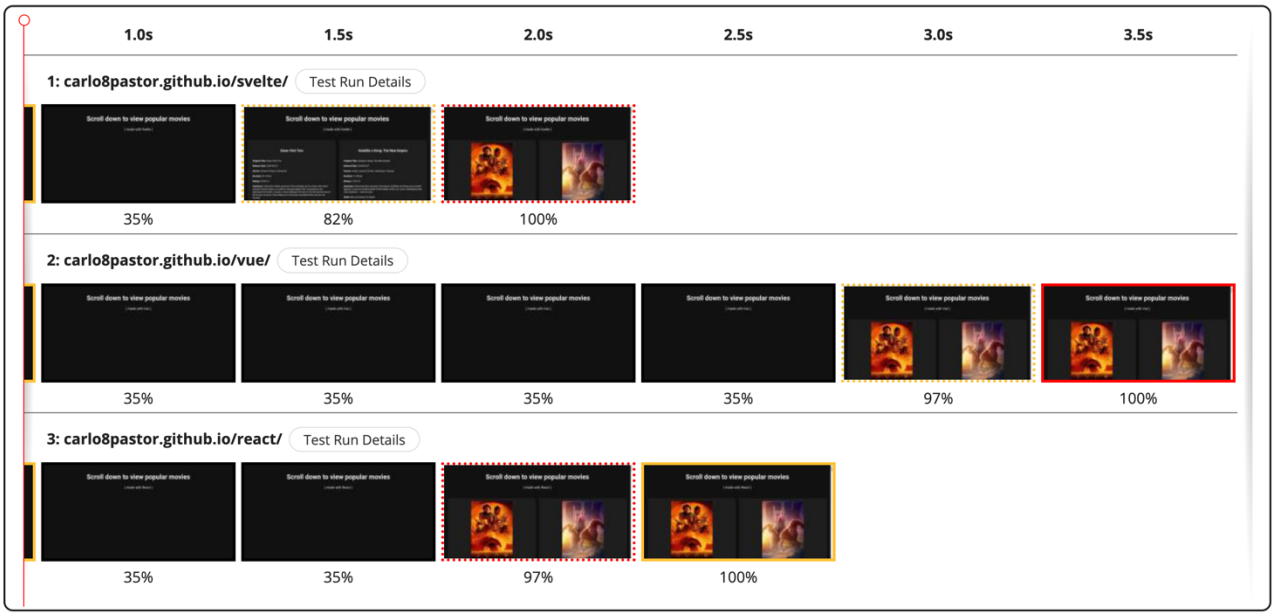

Layout Shifts
0.20
0.15
0.10
0.05
0.00
0.00 0.23 0.46 0.69 0.92 1.15 1.38 1.61 1.84 2.07 2.30 2.53 2.76 2.99 3.22 3.45 3.68 3.91 4.14 4.37 4.60
Time (seconds)

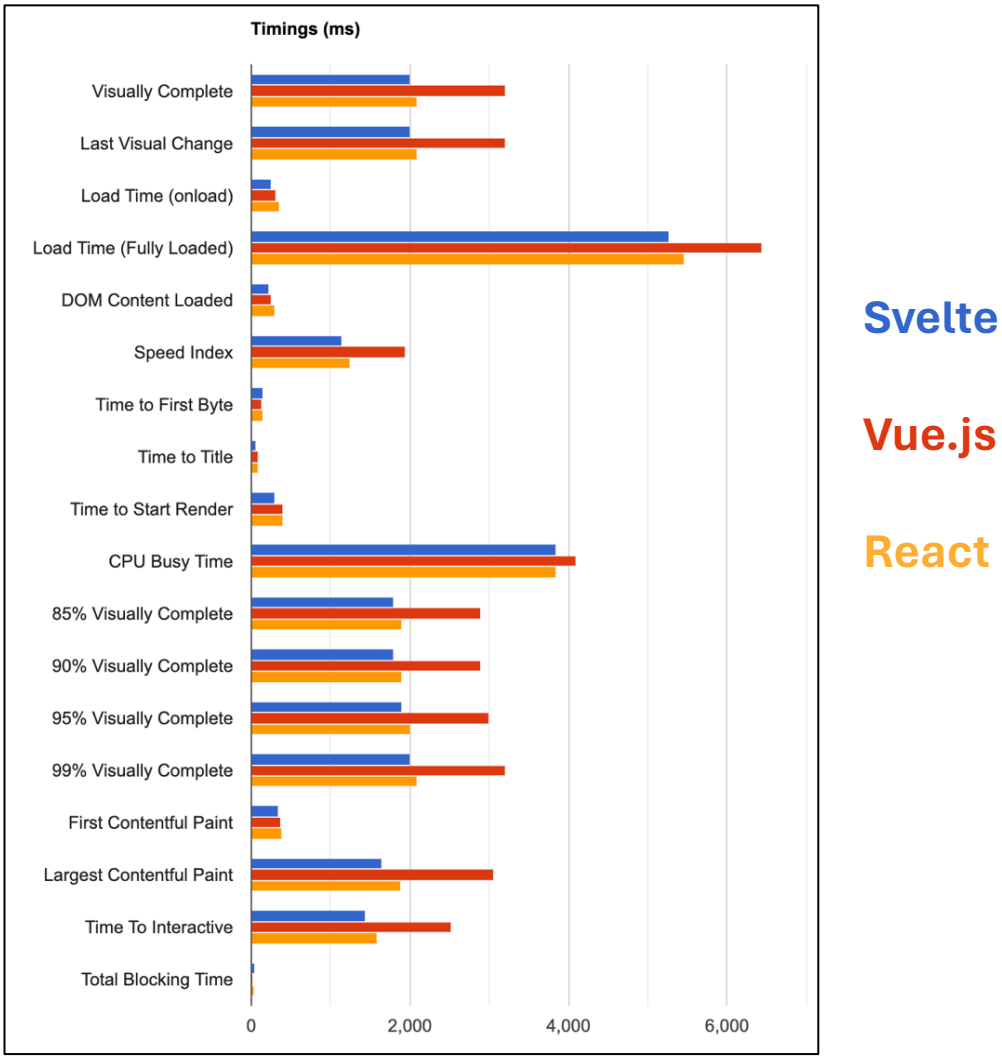

Visual Progress (%)
100
80
60
40
20
0
0.00 0.23 0.46 0.69 0.92 1.15 1.38 1.61 1.84 2.07 2.30 2.53 2.76 2.99 3.22 3.45 3.68 3.91 4.14 4.37 4.60
Time (seconds)

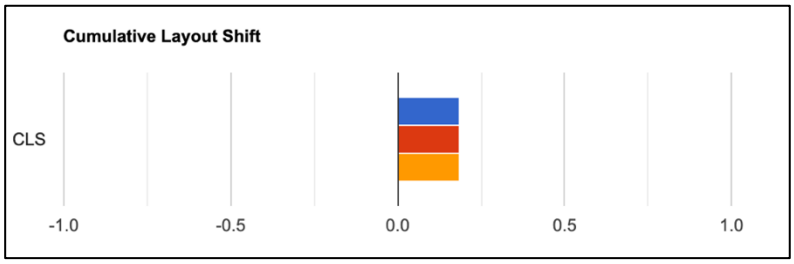

Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000

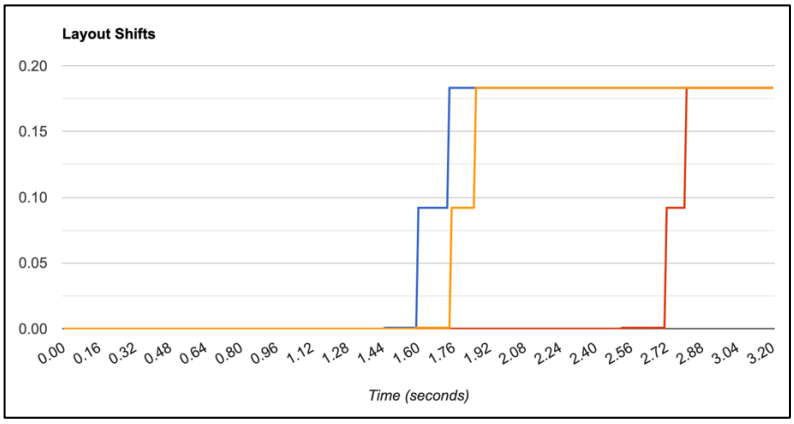

Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150
175

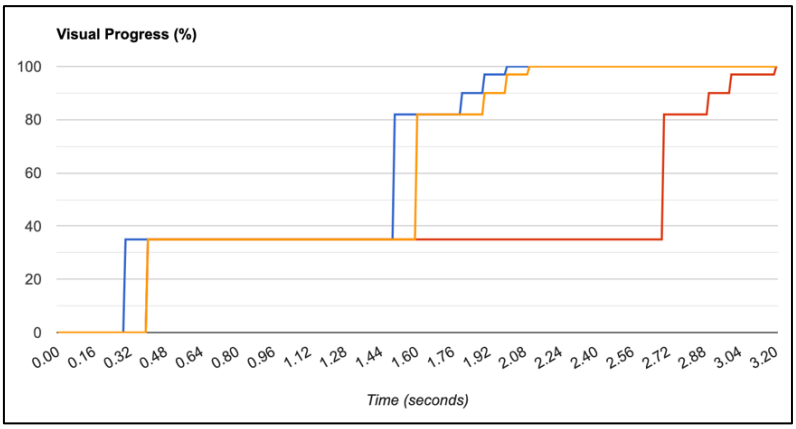
Seventh Comparison

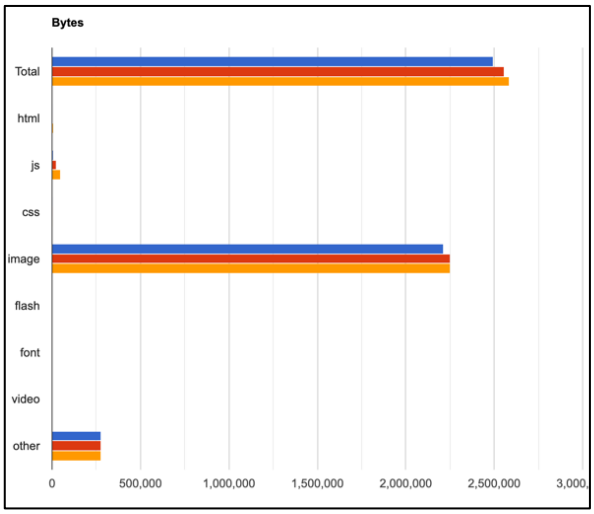


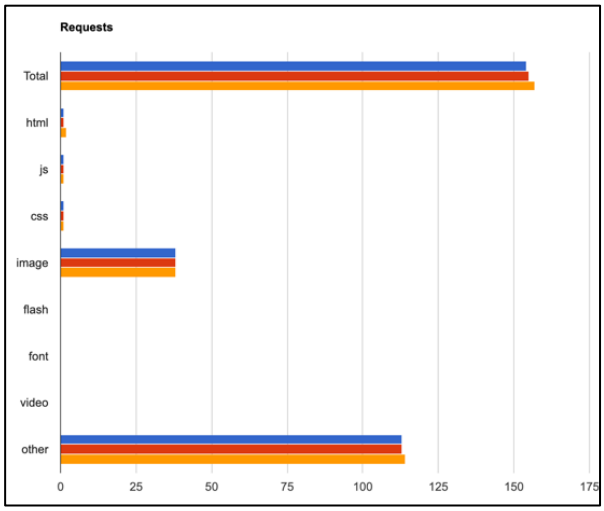



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0

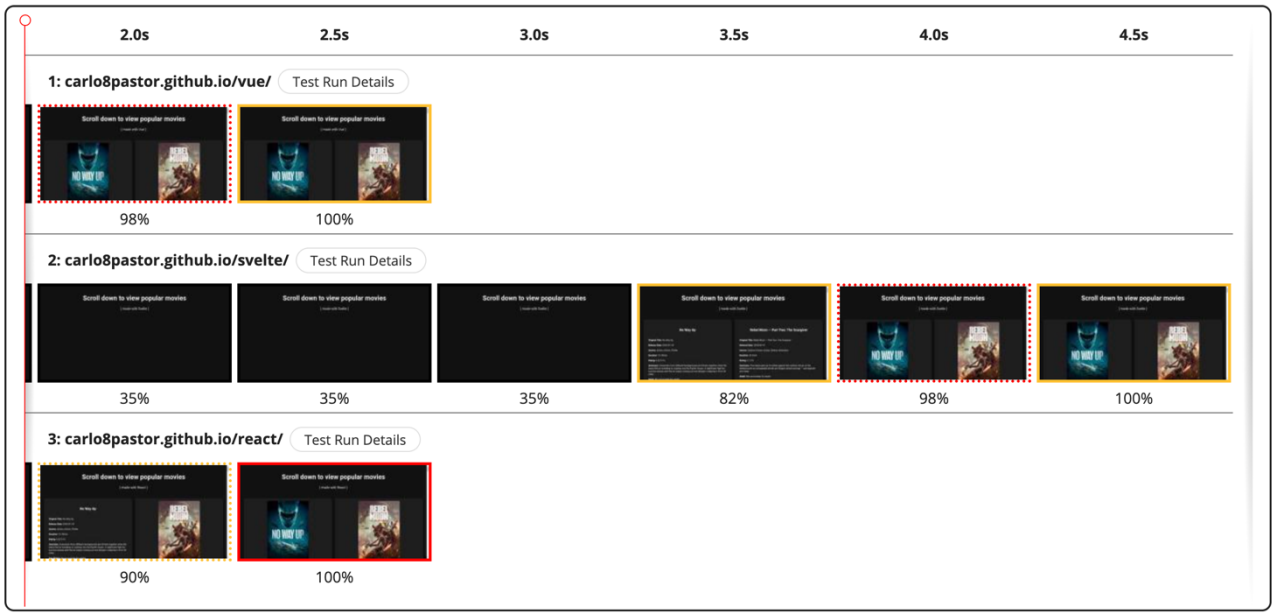

Layout Shifts
0.20
0.15
0.10
0.05
0.00
0.00 0.16 0.32 0.48 0.64 0.80 0.96 1.12 1.28 1.44 1.60 1.76 1.92 2.08 2.24 2.40 2.56 2.72 2.88 3.04 3.20
Time (seconds)

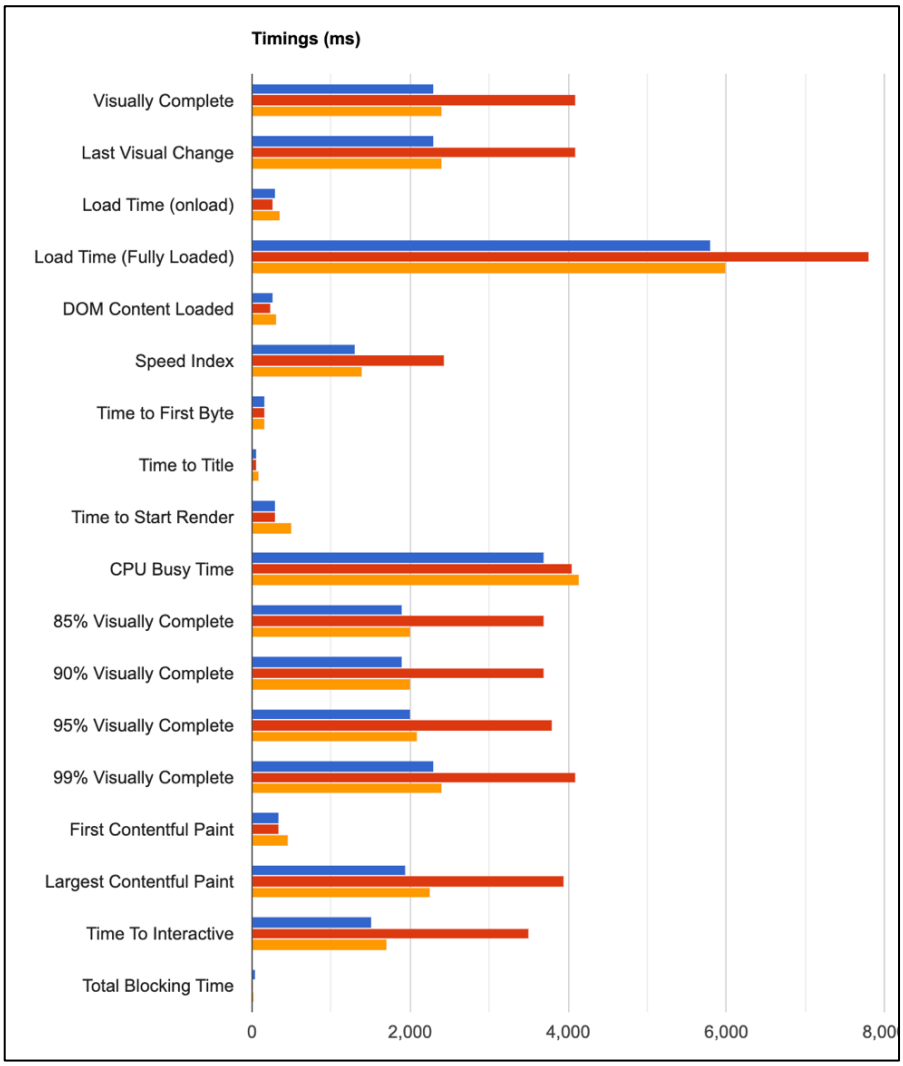

Visual Progress (%)
100
80
60
40
20
0
0.00 0.16 0.32 0.48 0.64 0.80 0.96 1.12 1.28 1.44 1.60 1.76 1.92 2.08 2.24 2.40 2.56 2.72 2.88 3.04 3.20
Time (seconds)


Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,


Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150
175

Eighth Comparison

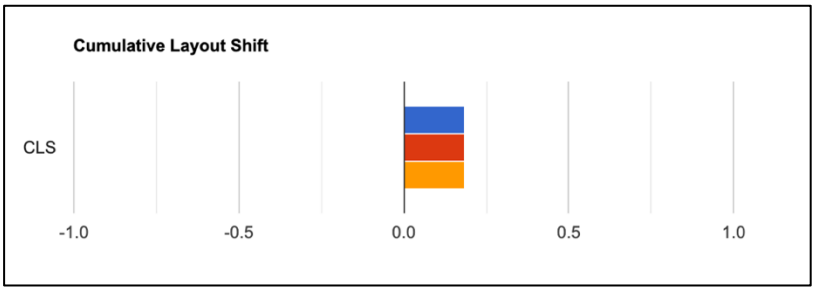


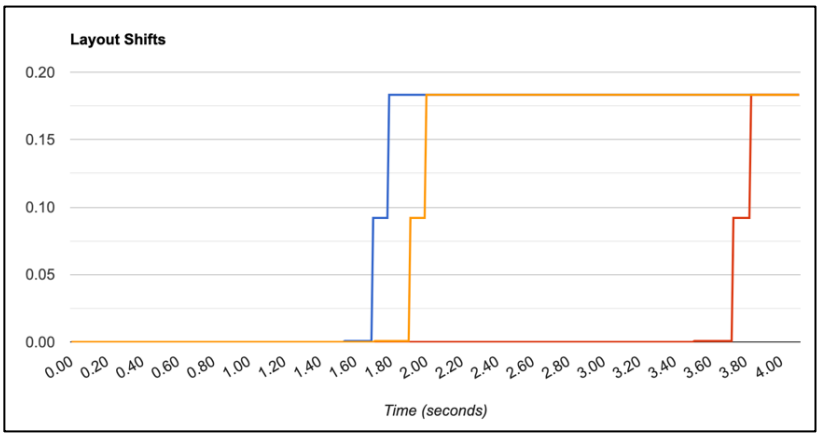


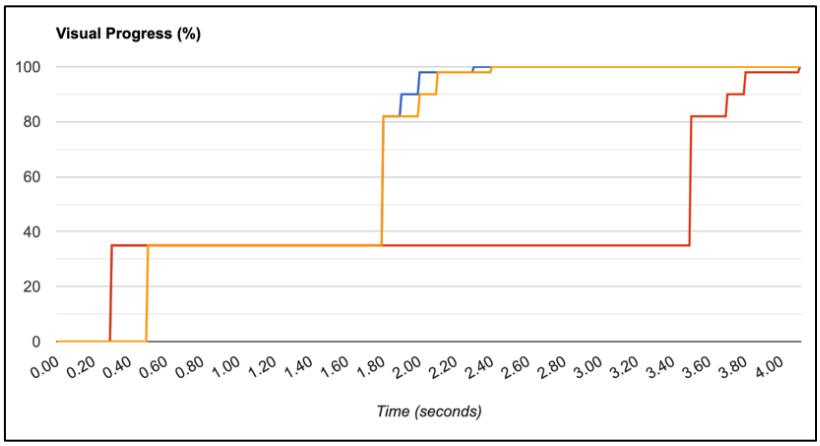
Vue.js

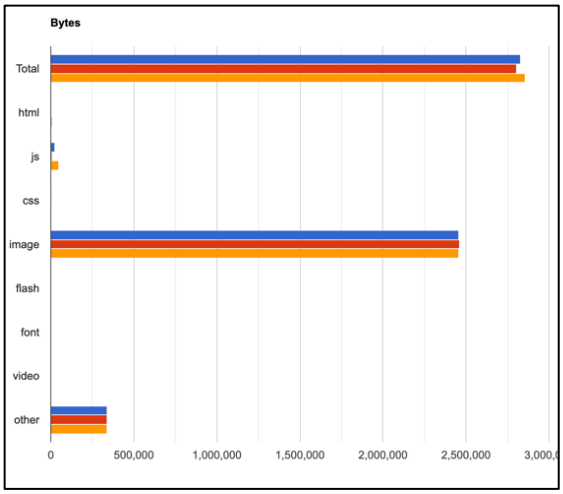
Svelte

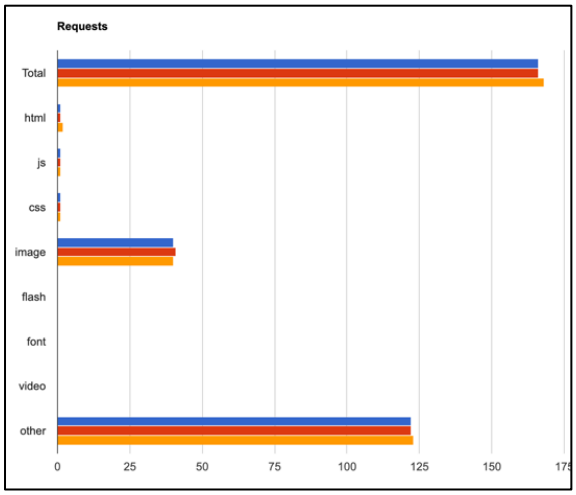
React


Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0

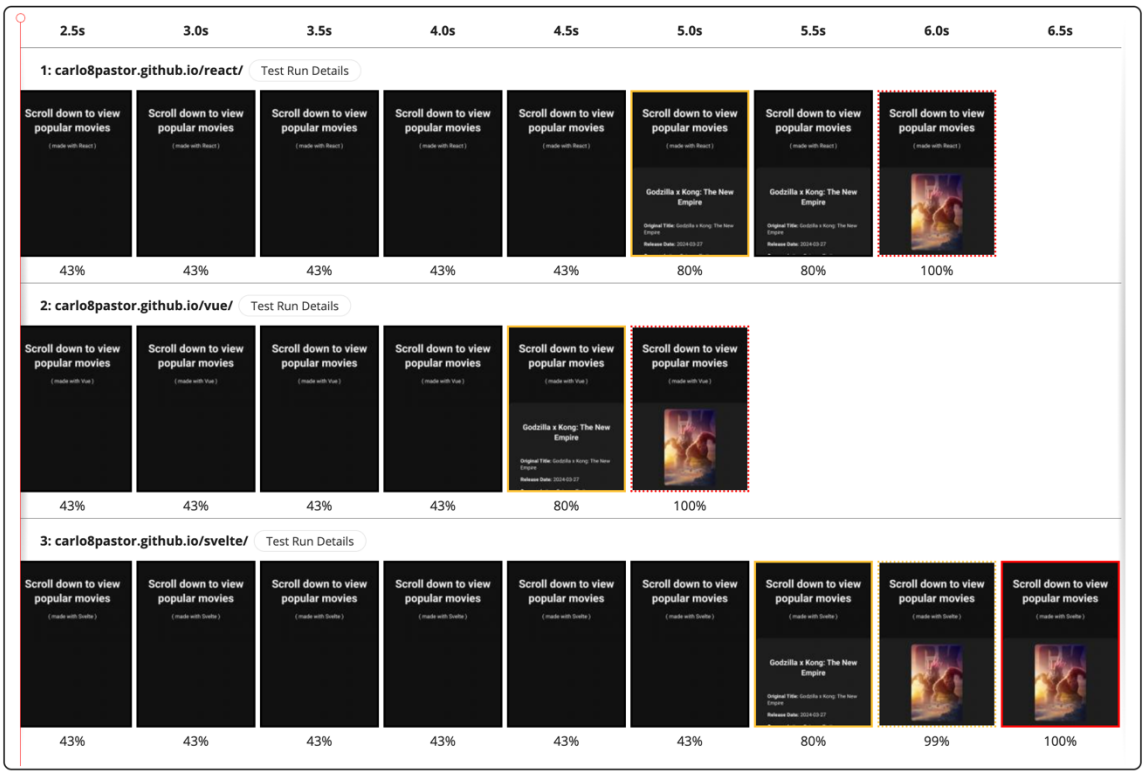

Layout Shifts
0.20
0.15
0.10
0.05
0.00
0.00 0.20 0.40 0.60 0.80 1.00 1.20 1.40 1.60 1.80 2.00 2.20 2.40 2.60 2.80 3.00 3.20 3.40 3.60 3.80 4.00
Time (seconds)

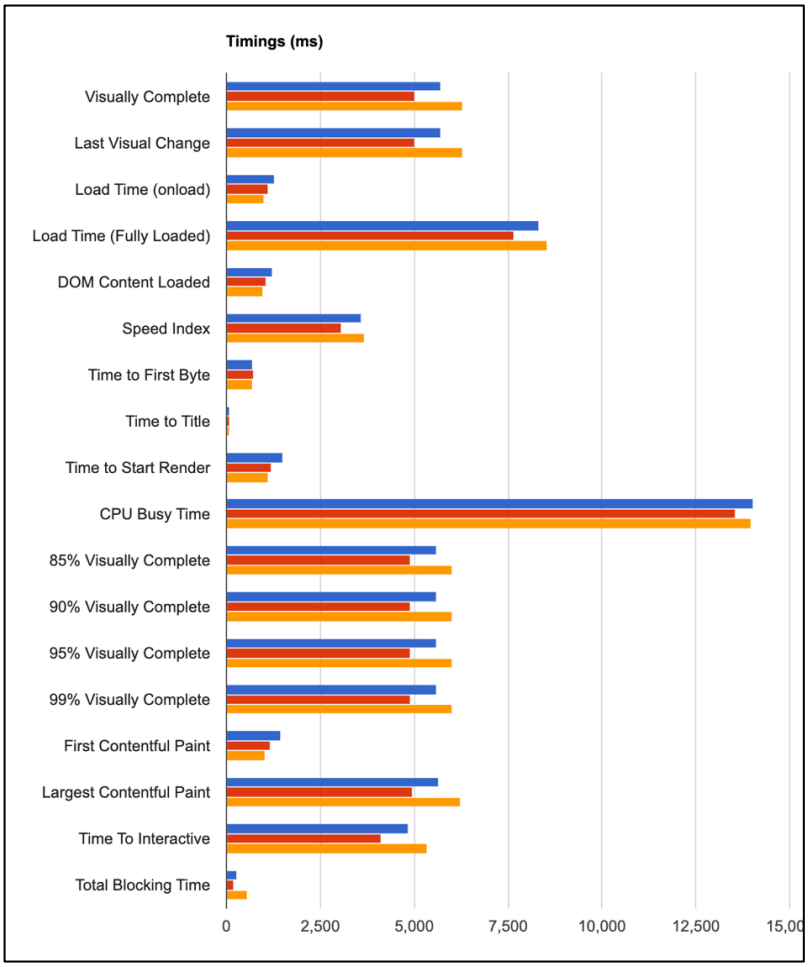

Visual Progress (%)
100
80
60
40
20
0
0.00 0.20 0.40 0.60 0.80 1.00 1.20 1.40 1.60 1.80 2.00 2.20 2.40 2.60 2.80 3.00 3.20 3.40 3.60 3.80 4.00
Time (seconds)

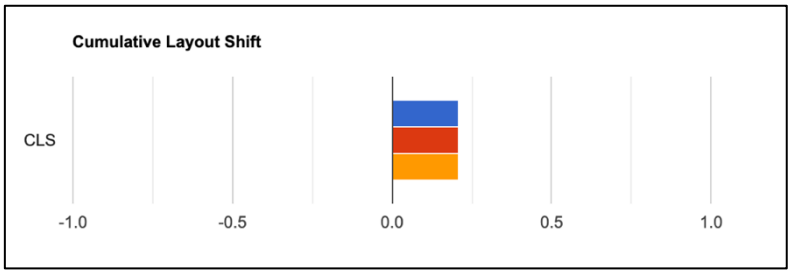

Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000

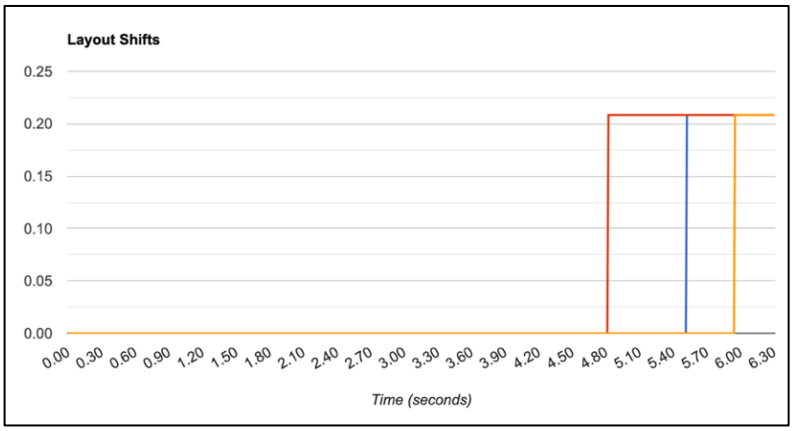

Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150

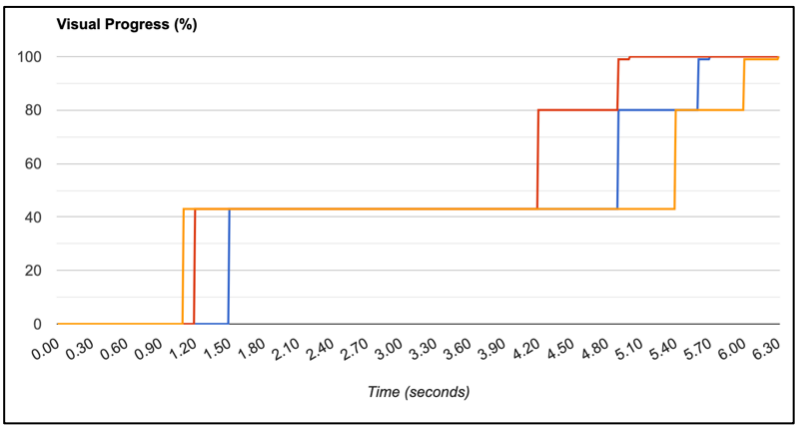
Ninth Comparison

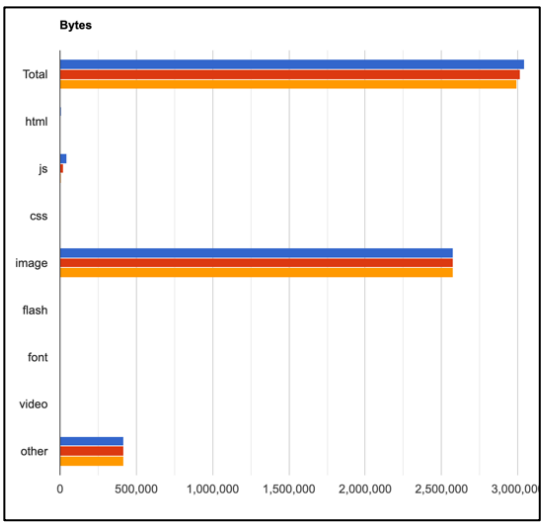


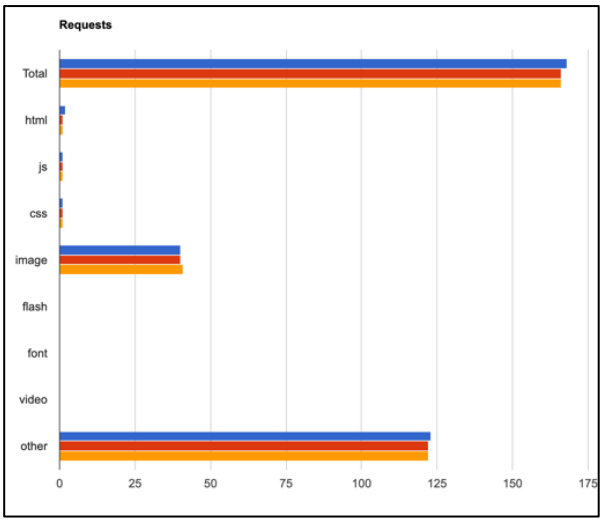



Cumulative Layout Shift
CLS
-1.0
-0.5
0.0
0.5
1.0


Layout Shifts
0.25
0.20
0.15
0.10
0.05
0.00
0.00 0.30 0.60 0.90 1.20 1.50 1.80 2.10 2.40 2.70 3.00 3.30 3.60 3.90 4.20 4.50 4.80 5.10 5.40 5.70 6.00 6.30
Time (seconds)


Visual Progress (%)
100
80
60
40
20
0
0.00 0.30 0.60 0.90 1.20 1.50 1.80 2.10 2.40 2.70 3.00 3.30 3.60 3.90 4.20 4.50 4.80 5.10 5.40 5.70 6.00 6.30
Time (seconds)


Bytes
Total
html
js
css
image
flash
font
video
other
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,00


Requests
Total
html
js
css
image
flash
font
video
other
0
25
50
75
100
125
150
175