The State of Disappearing Frameworks in 2023

Juho Vepsäläinen[a], Arto Hellas and Petri Vuorimaa
Department of Computer Science, School of Science, Aalto University, Espoo, Finland

Keywords: Astro, Code Splitting, Disappearing Frameworks, Fresh, Islands Architecture, JavaScript, Marko, Programming, Qwik, Qwik City, Svelte, SvelteKit, Software Architecture, Web, Web Programming, www.

Abstract: Disappearing frameworks represent a new type of thinking for web development. In the current mainstream JavaScript frameworks, the focus has been on developer experience at the cost of user experience. Disappearing frameworks shift the focus by aiming to deliver as little, even zero, JavaScript to the client. In this paper, we look at the options available in the ecosystem in mid-2023 and characterize them in terms of functionality and features to provide a state-of-the-art view of the trend. We found that the frameworks rely heavily on compilers, often support progressive enhancement, and most of the time support static output. While solutions like Astro are UI library agnostic, others, such as Marko, are more opinionated.

1 INTRODUCTION

Since its beginning in the 90s, the world wide web (Berners-Lee et al., 1992) has become an enormous success. Although initially designed with the delivery of static websites in mind, over time, the web has allowed developers to build cross-platform applications with relative ease (Severance, 2012). The evolution of JavaScript has supported the transition from a website delivery platform to an application platform (Wirfs-Brock and Eich, 2020). With this transition, the demands for the platform have grown over time as the users expect more.

The growing expectations have been intertwined with the evolution of web application development approaches. While traditional web relied on refreshing pages during operation, approaches such as single page applications (SPAs) have raised the bar in responsiveness and interactivity (Severance, 2012; Woychowsky, 2006). As shown by (Vepsäläinen et al., 2023), SPAs come with a cost of their own as they depend on client-side JavaScript and may be challenging to optimize for Search Engine Optimization (SEO) purposes. One particular challenge is the increasing amount of code shipped to the client (HTTP Archive, 2023). Newer approaches, such as disappearing frameworks, try to address these problem points.

To remedy the problems of SPAs, disappearing frameworks shift the focus on shipping a minimal amount of JavaScript to the client; hence the term disappearing (Vepsäläinen et al., 2023). Depending on the implementation, the way the framework disappears may differ, and it is an ongoing space of technical competition. The crux of disappearing frameworks is to take the best ideas from the early web – delivering static content – and combine them with the lessons learned from building SPAs, delivering as little as possible, even zero, JavaScript to the client.

This article surveys the currently available disappearing frameworks to understand possible ways to implement them and to create a map of the emerging space. The overarching research question of this article is as follows: *Which disappearing frameworks exist in the ecosystem, and how can they be characterized in terms of functionality and features while considering their pros and cons?*

The framing responds to the question, "What are the pros/cons of the solutions from a developer and a user perspective relative to the incumbent approaches" proposed in (Vepsäläinen et al., 2023). This more specific question motivates considering the discovered frameworks against popular solutions to understand how they differ from mainstream ones.

To address the question, we first discuss disappearing frameworks in the context of earlier work in Section 2, before providing a technical review and comparison of them in Section 3. We discuss the findings in Section 4 and conclude the findings in Sec-

[a] https://orcid.org/0000-0003-0025-5540

Vepsäläinen, J., Hellas, A. and Vuorimaa, P.
The State of Disappearing Frameworks in 2023.
DOI: 10.5220/0012174000003584
In *Proceedings of the 19th International Conference on Web Information Systems and Technologies (WEBIST 2023)*, pages 232-241
ISBN: 978-989-758-672-9; ISSN: 2184-3252

tion 5 while outlining future research directions.

2 BACKGROUND

As described in (Vepsäläinen et al., 2023), the evolution of the web can be characterized through several phases: formation in the 90s, the rise of SPAs, and re-evaluation of current practices. Although a simplified view, this evolution provides a brief background for understanding the current developments. Early websites and applications could be characterized as multi-page applications (MPAs), which relied heavily on server logic triggered through navigation (Kaluža et al., 2018). In part motivated by a need to reduce loading times (Nah, 2004), SPAs lifted the need to reload the whole page on each content change, improving user experience (UX) (Kaluža et al., 2018) while also fundamentally changing and improving the developer experience (Vepsäläinen et al., 2023). In the current phase of re-evaluating development practices, solutions mixing the benefits of both approaches are being explored, and disappearing frameworks form one of the potential options.

2.1 Libraries and Frameworks

Libraries and frameworks form one of the fundamental divisions in web development. Libraries such as React aim to do a single thing well. At the same time, frameworks such as Angular come with opinions, potentially increasing developers' productivity if they align with the framework's approach and work within its constraints. The border between a library and a framework is occasionally unclear, but this rough definition is sufficient for the present discussion.

To add complexity to the discussion, we also acknowledge the existence and emergence of meta-frameworks such as Astro. Meta-frameworks are headless as they do not rely on a specific user interface (UI) library but let the developer decide which one, or even many, to use.

2.2 Sprinkles Architecture a.k.a. jQuery and Friends

jQuery (2006) is an early example of a successful JavaScript library that has wide usage to this day(w3techs, 2023). jQuery was developed to address browser deficiencies in terms of ergonomics. jQuery could reduce twenty lines of standard DOM API code to a mere three through its chaining design while solving browser-incompatibility issues underneath (Bibeault et al., 2015). The style jQuery adopted could be characterized as "Sprinkles architecture", where JavaScript is sprinkled into the application to add interactivity while following the principle of progressive enhancement (Champeon, 2003). The example below shows what jQuery declarations look like:

```
$(".selector").on("click", () => alert("hi"))
```

The architecture pioneered by jQuery is still relevant, and most recently, several solutions, such as Alpine.js (2019), Sidewind (2019), and htmx (2020) have taken its ideas and moved it to HTML markup itself while still allowing JavaScript to be used when necessary.

2.3 Current Web Application Development Landscape

React, Angular, and Vue.js dominate the current web application development landscape. Still, it is good to keep in mind that they hold only a small portion of the global market, highlighting that modern web applications are still a small subset of the whole web[1]. Overall, these libraries and frameworks rely on components to allow the reuse and composition of code, leverage a templating solution (i.e., JSX) for component definition, and use a process called hydration to make the client-side code alive to the user by enabling event handlers and running component logic (Vepsäläinen et al., 2023).

Although the current technologies enable the creation of complex web-based experiences, it is not always clear what to use and when, as requirements tend to differ depending on the use case (Miller, 2019). For instance, for a simple website, using a complete framework might be too much in terms of complexity. On top of this, there is a development-related cost to consider, as frameworks can take time to configure and learn. Furthermore, by definition, frameworks are collections of opinions; at times, the framework opinions might not match what is required, which leads to additional work – frameworks can make complex tasks possible and possible tasks easy. Still, it is challenging to go against the inherent opinions of the frameworks.

Similarly, there are differences in the performance of frameworks that influence their suitability for specific tasks. As an example, (Ollila et al., 2022) explored the cost of rendering using contemporary frameworks and observed that the cost of React grows radically with the number of components, not to mention the amount of component code that

[1]E.g., the global market share of React is approximately 3.7% (Vepsäläinen et al., 2023).

must be loaded. In particular, leveraging a compiler can yield benefits when optimizing client-side performance (Ollila et al., 2022).

2.4 Islands Architecture

As pointed out by (Vepsäläinen et al., 2023), islands architecture can be considered as a stepping stone towards disappearing frameworks. The idea of islands architecture is to let the developer define which portions of a page are interactive while attaching a loading strategy to each interactive section; these areas are loaded only when they are needed (e.g., a user scrolls to a location on the page that was previously not visible), while not being loaded at all if not needed.

Compared to loading and hydrating the entire page before it becomes accessible to the user, islands are an improvement for the users. Astro framework has popularized the approach, and it is good to recognize islands architecture as a recent development aiming at the same direction (2019) (Miller, 2020; Hallie and Osmani, 2022). We cover one possible implementation when discussing Astro in detail in Section 3.1.

The key differences between server-side rendering (SSR), progressive hydration, and islands architecture are illustrated in Figure 1. Compared to regular hydration, progressive hydration goes further by hydrating key components first and the rest later (Hallie and Osmani, 2022).

2.5 Disappearing Frameworks

The term disappearing frameworks appeared to the public in (O'Shaughnessy, 2018) (2018), and as described by (Carniato, 2021a), disappearing frameworks represent a paradigm-level shift by rephrasing the problem addressed by contemporary frameworks. Rather than addressing the problem of developing complex web applications, disappearing frameworks aim to remove themselves from an application and try to start from close to zero cost in terms of JavaScript shipped to the client (Vepsäläinen et al., 2023). At the same time, disappearing frameworks leverage ideas, such as using components of the earlier generation, but how they are framed differs, especially from a loading point of view.

By shifting focus to the cost of what is shipped, disappearing frameworks reach towards the best practices discovered during the early web. Although the client still has to perform some work, the target is to defer the work performed and avoid it when possible. By reducing the amount of scripting in the client, accessibility is improved, especially in performance-limited contexts, such as mobile devices (Ollila et al., 2022).

The idea of disappearing frameworks is consistent with Rich Harris' transitional web applications (TWAs) from 2021 that aim to capture the best ideas of both the traditional web and the SPAs (Harris, 2021) and practices such as progressive enhancement from 2008 (Gustafson et al., 2008) that encourage developers to think markup and styling first before applying JavaScript logic. TWAs go a step further, and the idea is that such a web application should be able to work without JavaScript enabled and, therefore, contain the necessary fallback mechanisms to work in this case.

Disappearing frameworks respond to the demands of both developers and users as they take the advancements gained during the SPA era and adapt them to the best practices discovered during the earlier practices established for developing MPAs. In other words, the movement aims to remedy the gap in the approaches while delivering better user experiences, especially in contexts with limited bandwidth and computing power. As a side benefit, the shift also aligns with green computing that highlights the need for increased mindfulness of resource usage (Kurp, 2008).

2.6 Developing Web Applications Server-First

While the previous examples have heavily focused on client-side development, there is also an emphasis on server-side functionality. For example, Phoenix LiveView emphasizes what happens at the server, and sites written in LiveView can work without JavaScript, but when enabled, changes are relayed to the client through a WebSocket (Phoenix LiveView, 2023). The approach allows Elixir developers to build complex applications while staying within the Elixir language environment. The approach is interesting because it optimizes for First Meaningful Paint (FMP) while leaving JavaScript under the hood, allowing developers to stay within their preferred environment. Phoenix LiveView is not unique, and there are many implementations for other programming languages beyond Elixir, as listed in (GitHub LiveViews, 2023).

3 TECHNICAL REVIEW

By our definition, a framework is a disappearing framework if the focus is on aiming for zero or near zero cost in terms of JavaScript delivered to the client. The definition rules out older frameworks shipping a

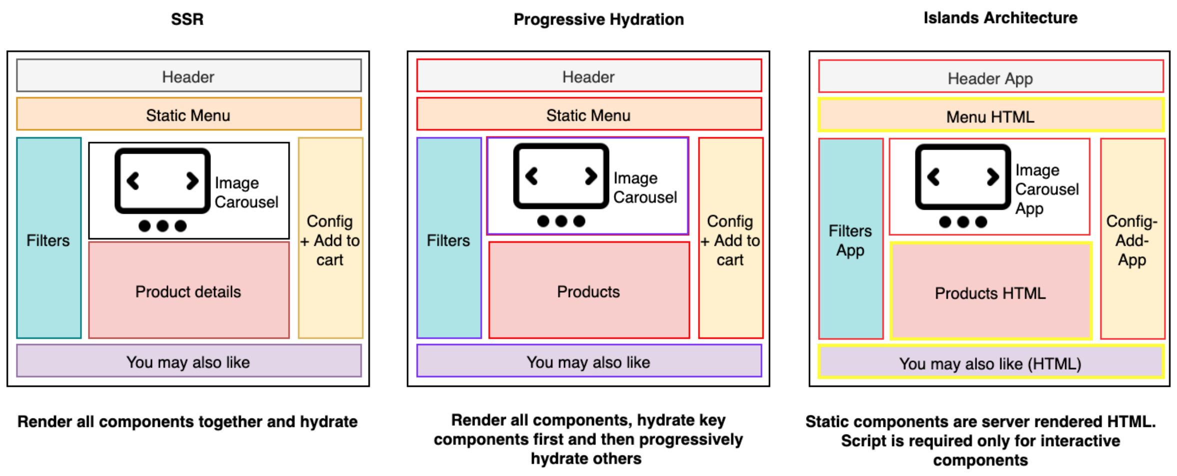

Figure 1: Three examples of rendering of a website. Server-side rendering (SSR) creates the website on the server, sending it to the client. Progressive hydration allows the prioritization of components rendered and shown to the client. Islands architecture, on the other hand, creates a static website on the server with placeholders for islands, which are then retrieved only if needed (Hallie and Osmani, 2022).

runtime and application code to evaluate at the client, giving a good baseline for evaluating newer options. In this article, we explicitly focus on frameworks that self-identify as ones that seek to minimize delivered JavaScript. To scope further, we limit our analysis to solutions that are under active development and have had a recent release.

For the present analysis, we identified ten frameworks, outlined in Table 1, listing the projects and their characteristics while showing our selection of projects to study in detail. The identification of the frameworks was based on existing discussions (e.g. (O'Shaughnessy, 2018; Carniato, 2021b)) and an exploration of conferences focusing on modern web development practices (e.g., Future Frontend 2023 conference[2]).

3.1 Astro

Astro is a meta-framework built with islands architecture in mind. The project aims to allow developer experience (DX) familiar with contemporary frameworks within a static environment (Astro, 2023). Astro achieves this in several ways:

1. Islands are supported out of the box - In other words, developers can decide which component boundaries should be interactive. (Astro, 2023)
2. Server-first API design - To minimize the cost of hydration, work is offloaded to the server as soon as possible. (Astro, 2023)
3. Zero JavaScript by default - Astro doesn't emit any JavaScript by default. (Astro, 2023)
4. Edge-ready - Astro sites can be deployed anywhere, edge included, due to its static nature. (Astro, 2023)
5. Customizable - Numerous extensions exist to expand the capabilities of Astro. (Astro, 2023)
6. UI-agnostic - Astro can host many contemporary JavaScript frameworks. (Astro, 2023)

3.1.1 Astro Islands

Astro implements islands architecture through what it calls Astro islands (Astro Islands, 2023). The following example illustrates how to load a static React component through Astro:

```
---
import MyComponent from "../MyComponent.jsx";
---
<MyComponent />
```

To make the component interactive, you have to mark it as an island while giving it a loading strategy as below (Astro Islands, 2023):

```
---
import MyComponent from "../MyComponent.jsx";
---
<!-- The component is interactive on load -->
<MyComponent client:load />
```

Beyond loading the island immediately, Astro provides other strategies, including **client:idle**, **client:visible**, and **client:media** to mention some (Astro Directives, 2023).

3.1.2 Observations

Astro is the first framework that embraced the islands architecture as a first-class citizen and used it as a part

[2]https://futurefrontend.com/2023/

Table 1: Potential web application frameworks and libraries.

Name	Type	Version	Compiled	Notes	Included
Svelte	Library	4.2.0, 2023-08-11	✓	Complemented by SvelteKit for routing and related functionality	✓
Elder	Framework	1.7.5, 2022-05-31	✓	Built on top of Svelte	
Stencil	Library	4.1.0, 2023-08-21		Focus on authoring Web Components	
Angular	Framework	16.2.3, 2023-08-30	✓	Conventional framework with support for Web Components	
Marko	Framework	5.31.6, 2023-08-25	✓	Implements a DSL on top of HTML, supports streaming	✓
Qwik	Framework	1.2.10, 2023-08-26	✓	Complemented by Qwik City for routing and related functionality	✓
Astro	Framework	3.0.8, 2023-09-04	✓	Focus on islands architecture	✓
îles	Static site generator	0.9.5, 2023-04-07	✓	Implements partial hydration and comes with zero cost by default	
Slinkity	Framework	1.0.0-canary.1, 2023-01-09		Based on 11ty static site generator and Vite bundler, early alpha	
Fresh	Framework	1.4.2, 2023-08-17		Deno-based, edge friendly framework with SSR support	✓

of their marketing effort. Astro is not bound to a specific UI library and provides flexibility in static and dynamic use cases by supporting both. Astro comes with zero cost by default for JavaScript shipped to the client, and the cost is added only by defining islands. It can be argued that what happens beyond that could be potentially costly. Still, at the same time, this could be seen as a pragmatic compromise as the approach allows leveraging what is available in the ecosystem of popular UI libraries such as React.

3.2 Fresh

Fresh calls itself the next-generation web framework, and it claims to have the following features (Fresh, 2023): just-in-time rendering, island-based client hydration (Jiang, 2023), zero runtime overhead, no build step, no configuration, and TypeScript support. In other words, Fresh renders on demand over the edge (JIT) while supporting islands architecture, enabling developers to ship code for interactivity when needed. Due to its edge-oriented approach, it avoids the build step and configuration. By leveraging Deno as the runtime instead of Node.js for its implementation, TypeScript support is gained out of the box.

Based on Fresh documentation, Fresh leverages Preact and JSX for rendering (Fresh Documentation, 2023). Preact is a light (3 kB) implementation of React API, making it an ideal choice for a framework like Fresh. The combination of Deno and Preact also restricts the project as it is not framework-agnostic like Astro. Still, at the same time, the choice is understandable, given the project constraints and focus.

3.3 Svelte and SvelteKit

According to its homepage, Svelte lets developers build "cybernetically enhanced web apps" (Svelte, 2023). Svelte's central claims to fame are writing less code, lacking a virtual DOM, and being genuinely reactive. In technical terms, Svelte relies on a compiler-based approach, and compared to the incumbent frameworks, it claims to avoid the associated cost at the browser.

3.3.1 Svelte Compiler

The description of Svelte aligns well with the idea behind disappearing frameworks. Furthermore, Svelte retains many features of the earlier solutions, including component orientation, and it comes with a templating language. At the same time, the hydration step is achieved through code generated by the Svelte compiler. The Svelte compiler accepts code such as the one below adapted from Svelte documentation (Svelte, 2023):

```
<script>
  let count = 0;
  function handleClick() { count += 1; }
</script>
<button on:click={handleClick}>
  Clicked {count} time{count > 1 ? "s" : ""}
</button>
```

Although Svelte alone is enough for simple applications, it is often complemented by a solution such as Astro or SvelteKit. We already saw Astro at Subsection 3.1 and will discuss SvelteKit next.

3.3.2 SvelteKit

SvelteKit is a framework that builds on top of Svelte and provides core features, such as routing and server-side rendering (SSR). SvelteKit leverages Vite bundler for the added functionality and implements developer-oriented features such as Hot Module Replacement (HMR) for adequate development flow (SvelteKit, 2023).

While SvelteKit can run as a server in production mode, it also supports static site generation (SSG). A SvelteKit site can be hosted on top of popular edge platforms, including Cloudflare Pages, Netlify, and Vercel while allowing developers to adapt to any platform beyond the officially supported ones (SvelteKit, 2023).

From the user's point of view, SvelteKit allows developing sites that work without JavaScript and that enhance progressively if JavaScript is available to provide higher quality user experience (SvelteKit Form Actions, 2023). SvelteKit provides flexibility to the developer by allowing them to decide how (CSR, SSR) and where to render a page (server, client) to support both dynamic and static use cases depending on what is required (SvelteKit Page Options, 2023).

3.3.3 Observations

Svelte and SvelteKit align well with the ideas behind disappearing frameworks. It is not a big surprise, given both projects were initiated by Rich Harris, the developer who introduced the idea of TWAs to the broader public. Given that TWAs are closely aligned with disappearing frameworks, it makes sense that Svelte and related solutions comply well with the target of shipping less JavaScript to the client.

There is some learning curve to Svelte's approach as it implies learning to use Svelte's DSL for defining components. The connection between state and template is mainly intuitive, but learning template control structures requires effort from the developer.

Svelte was one of the earliest JavaScript frameworks built as a compiler; other projects have followed that choice. Compiler-based approaches contrast earlier runtime and bundler-focused frameworks where client cost was not considered as crucial as disappearing frameworks.

3.4 Qwik and Qwik City

Qwik is a web framework built by Miško Hevery (Angular.js), Adam Bradley (Ionic), and Manu Almeida (Gin Framework) (Adservio, 2022). The framework addresses the conflicting requirements of interactivity and page speed (Qwik's magic, 2022). The conflict means you have to compromise in interactivity or speed due to the hydration cost of the current frameworks (Qwik's magic, 2022). According to Qwik documentation (Qwik, 2022a), the framework is solving the quandary using the following means:

1. Resumability over hydration – Instead of hydrating, Qwik can resume code execution on demand.
2. Automatic code splitting – Qwik splits code aggressively due to the approach avoiding manual effort by developers.
3. Tiny runtime – Qwik runtime is only one kilobyte.
4. Compilation over runtime cost – Qwik's optimizer pushes some state management to the server side.

It is these factors that make Qwik unique compared to its competition. Because of its approach, Qwik represents a paradigm-level shift in how to develop web applications. One of the ways it achieves its targets is by focusing on different metrics than its predecessors, namely Time to Interactive (TTI) over Time to Load (TTL).

While many frameworks focus on TTL, the core metric of Qwik is TTI (Tyson, 2022). The developers of Qwik are concerned about how soon a web page can become responsive to user interaction. This shift in perspective likely motivated the approach and the idea of resumability.

3.4.1 Resumability

Resumability allows pages to become interactive on demand based on user intent (Qwik's magic, 2022) while picking up where the server left off (Adservio, 2022). In contrast to hydration-based approaches, there is less work to do for the browser as a part of it has already been performed. As a result, TTI can become low, and the page can quickly respond to user interaction.

The idea of resumability is not new, as illustrated by the example of jQuery (2006) (Qwik's magic, 2022). The difference with jQuery is that working with Qwik is similar to working with React regarding developer experience (DX). Qwik has adopted a similar component style and supports reactive state management out of the box (Qwik's magic, 2022). To leverage resumability, Qwik generates both server and client-side code while serializing using a so-called optimizer (Qwik's magic, 2022)

3.4.2 Optimizer and Automatic Code Splitting

The optimizer is a core part of Qwik generating code using automatic code splitting applied across component and event listeners, meaning Qwik defers loading necessary code as long as possible (Qwik, 2022b).

Given deferring isn't always the preferred behavior, Qwik supports standard preloading strategies (Qwik, 2022b). The strategies are then used by a service worker set up by Qwik to monitor application state (Qwik's magic, 2022).

For code splitting, Qwik has been split architecturally into three parts (Hevery, 2021): view, state, and handlers. Usually, these three parts are coupled and held together in code. The coupling means there are three parts to download, parse, and execute together (Hevery, 2021). Even if only one part is needed based on user interaction, all three must be processed regardless (Hevery, 2021).

Because of the separation of concerns in Qwik, automatic code splitting has become possible at an unprecedented level. A specific dollar-based code convention is used to mark the code splitting boundaries, and then the optimizer can compile the code based on them (Qwik, 2022a). Due to the shift in perspective, Qwik can achieve fine-grained code splitting out of the box that its predecessors couldn't due to their tighter coupling of concerns.

3.4.3 Code Splitting Boundaries, Reactive State, and Components

To better understand how code splitting boundaries work in Qwik, consider the counter-based example below from Qwik documentation (Qwik, 2022a). It also illustrates Qwik's usage components and reactive state, specifically signals.

```
import * as qwik from "@builder.io/qwik";

export default qwik.component$(() => {
  const count = qwik.useSignal(0);
  return (
    <div>
      <span>Count: {count.value}</span>
      <button onClick$={() => count.value++}>
        Click
      </button>
    </div>
  );
});
```

The example is relatively close to the code you would write in React. React created suitable programming interfaces as a pioneer, and Qwik decided to mimic them while adding twists on top.

3.4.4 Qwik City

Qwik City is a meta-framework comparable to Next.js for React (Qwik's magic, 2022). It provides the following features on top of Qwik (Qwik's magic, 2022): directory-based routing, nested layouts, file-based menus, breadcrumbs, support authoring content with .tsx or .mdx file formats, and data endpoints. The technical target of Qwik City is to provide the capabilities of an MPA with the benefits of a SPA (Qwik's magic, 2022). Therefore, navigation-wise, the solution avoids page refreshes commonly encountered in MPAs and allows UX familiar from SPAs.

3.4.5 Observations

Qwik represents a paradigm-level shift in how web applications are developed. It approaches the problem from a different angle. Compared to earlier solutions, it tries to quickly solve the issue of providing highly interactive pages to the client on a tooling level thanks to its compiler-based implementation. At the same time, it has adopted ergonomics familiar to developers from React, easing its adoption.

3.5 Marko

Marko's main claims are familiarity, performance, scalability, and trustworthiness. By familiarity, Marko means that it has been built on top of standard JavaScript, CSS, and HTML with tweaks as a DSL for HTML. Performance is achieved through streaming, partial hydration, optimizing the compiler, and a small runtime. Scalability is reached through component orientation, as the system can be expanded as required. Trustworthiness is gained by the fact that Marko powers high-traffic sites, such as ebay.com. (Marko, 2023)

3.5.1 Marko DSL

To illustrate Marko DSL, the documentation provides the following example showcasing how to loop through an array and map it as an HTML list (Marko, 2023):

```
<!doctype html>
<html>
<head><title>Hello Marko</title></head>
<body>
    <h1>My favorite colors</h1>
    <ul>
        <for|color| of=["red", "tan", "hue"]>
            <li style=`color:${color}`>
                ${color.toUpperCase()}
            </li>
        </for>
    </ul>
    <shared-footer/>
</body>
</html>
```

Table 2: Comparison of disappearing frameworks.

Name	Rendering and hosting approach	Static output	Stance on external UI libraries
Svelte	Compiles only what is needed. Possible to host on popular edge platforms when using SvelteKit (Svelte Adapters, 2023).	Through SvelteKit	Svelte only
Marko	Compiles what is needed and streams results to the client. Possible to host on Cloudflare Workers and platforms supporting Node.js (Marko Server Integrations, 2023).		Marko DSL only
Qwik	Compiles features within split points loaded on demand and ships a minimal runtime for bootstrapping. Possible to host on edge and Node.js platforms (Qwik Deployments, 2023).	✓	Supports React
Astro	Implements islands architecture and provides multiple strategies for loading the islands. It includes a compiler and supports rendering for dynamic use cases on the edge.	✓	Works with React and others
Fresh	Renders on demand on top of the edge. Possible to host on Deno Deploy or through Docker (Fresh Deployment, 2023).		Depends on Preact

3.5.2 Observations

Marko fits the definition of a disappearing framework well, as one of its design goals is a small runtime and optimized rendering. Furthermore, leveraging partial hydration and streaming further improves user experience as a page can be rendered while the user receives data. Marko's DSL has a learning curve that plugins for popular code editors have alleviated.

4 DISCUSSION

Several frameworks have already adopted ideas suitable for disappearing frameworks. Many advertise to ship zero JavaScript to the client by default, which shows that the developers of the frameworks are aware of the problem of increasing delivered JavaScript (HTTP Archive, 2023) and the associated costs.

4.1 Main Observations

Our technical review highlights the following observations: many frameworks are built as compilers, many solutions support static output, and there are different takes on interoperability.

Compared to earlier frameworks, such as React, which leveraged a compiler only for its JSX templating, it seems the new generation relies heavily on compilers. The shift enables the new frameworks to keep the runtime shipped to the client as lean as possible. The shift is supported by (Ollila et al., 2022), as he states that runtime-based approaches are costly and using a compiler helps with client-side performance.

Many solutions support static output, allowing them to work as SSGs (Petersen, 2016; Camden and Rinaldi, 2017). For cases that do not support static output, a solution like ssr-to-html can be used to extract it, but at the same time, the results may not be ideal.

Astro is an example of a UI library agnostic framework, while others enforce a specific approach (Astro, 2023). Qwik meets somewhere in the middle by leveraging JSX and providing compatibility with React (Qwik React, 2023). Marko is an extreme example with its custom DSL.

4.2 Methods for Improving Existing Frameworks

As replacing a framework can be costly, research seeks ways to optimize the use of existing frameworks. In (Vogel and Springer, 2023), the authors introduce two new open-source frameworks that can delay JavaScript code without breaking it. Through these kinds of tools, it becomes possible to leverage modern development practices on aging codebases.

4.3 Summary of Results

The rendering and hosting approaches, support for static output, and the stance on external UI libraries for the reviewed frameworks are outlined in Table 2.

5 CONCLUSION

Disappearing frameworks are a new trend in web development. In this paper, we addressed the research question *Which disappearing frameworks exist in the* *ecosystem, and how can they be characterized in terms of functionality and features while considering their pros and cons?* by conducting a technical review of frameworks that comply with the definition of disappearing frameworks by aiming to delivery zero or near zero JavaScript to the client. We observe that many approaches rely on a compiler while also considering the state in which client-side JavaScript has been completely disabled. Several options also support static output, allowing easy hosting. In addition, solutions such as Astro provide limited interoperability with earlier UI libraries.

In this article, we addressed one of the questions proposed by (Vepsäläinen et al., 2023) and gained a further understanding of the space of disappearing web frameworks. Additional research is needed to understand how the frameworks perform relative to the mainstream frameworks and each other. Performance studies should not be limited only to the performance experienced by the client, as developer experience can also significantly influence the adoption of new technologies, as hinted by (Ferreira et al., 2022). In addition, it would be worthwhile to evaluate the code authored using the frameworks in detail to understand differences in understandability, complexity, and the number of lines of code, for example.

REFERENCES

Adservio (2022). Qwik – The Post-Modern Framework — adservio.fr. https://www.adservio.fr/post/qwik-the-post-modern-framework. [Accessed 15-Nov-2022].

Astro (2023). Getting Started — docs.astro.build. https://docs.astro.build/en/getting-started/. [Accessed 13-Apr-2023].

Astro Directives (2023). Template Directives Reference — docs.astro.build. https://docs.astro.build/en/reference/directives-reference/. [Accessed 17-Apr-2023].

Astro Islands (2023). Astro Islands — docs.astro.build. https://docs.astro.build/en/concepts/islands/. [Accessed 17-Apr-2023].

Berners-Lee, T., Cailliau, R., Groff, J.-F., and Pollermann, B. (1992). World-wide web: the information universe. *Internet Research*.

Bibeault, B., De Rosa, A., and Katz, Y. (2015). *jQuery in Action*. Simon and Schuster.

Camden, R. and Rinaldi, B. (2017). *Working with Static Sites: Bringing the Power of Simplicity to Modern Sites*. " O'Reilly Media, Inc.".

Carniato, R. (2021a). Understanding transitional javascript apps. [Accessed 29-Sep-2022].

Carniato, R. (2021b). Understanding Transitional JavaScript Apps — dev.to. https://dev.to/this-is-learning/understanding-transitional-javascript-apps-27i2. [Accessed 13-Apr-2023].

Champeon, S. (2003). Progressive enhancement and the future of web design. http://www.webmonkey.com/03/21/index3a.html. [Accessed over The Wayback Machine, 15-May-2023].

Ferreira, F., Borges, H. S., and Valente, M. T. (2022). On the (un-) adoption of javascript front-end frameworks. *Software: Practice and Experience*, 52(4):947–966.

Fresh (2023). fresh - The next-gen web framework. — fresh.deno.dev. https://fresh.deno.dev/. [Accessed 19-Apr-2023].

Fresh Deployment (2023). Deployment — fresh docs — fresh.deno.dev. https://fresh.deno.dev/docs/concepts/deployment. [Accessed 19-Apr-2023].

Fresh Documentation (2023). Introduction — fresh docs — fresh.deno.dev. https://fresh.deno.dev/docs/introduction. [Accessed 19-Apr-2023].

GitHub LiveViews (2023). GitHub - liveviews/liveviews: Phoenix LiveView workalikes for different languages and frameworks — github.com. https://github.com/liveviews/liveviews. [Accessed 27-Apr-2023].

Gustafson, A., Overkamp, L., Brosset, P., Prater, S. V., Wills, M., and PenzeyMoog, E. (2008). Understanding progressive enhancement. [Accessed 29-Sep-2022].

Hallie, L. and Osmani, A. (2022). Islands Architecture — patterns.dev. https://www.patterns.dev/posts/islands-architecture/. [Accessed 29-Sep-2022].

Harris, R. (2021). Have single-page apps ruined the web? — transitional apps with rich harris, nytimes. [Accessed 29-Sep-2022].

Hevery, M. (2021). Your bundler is doing it wrong — dev.to. https://dev.to/builderio/your-bundler-is-doing-it-wrong-ic0. [Accessed 14-Nov-2022].

HTTP Archive (2023). State of javascript. https://httparchive.org/reports/state-of-javascript. [Accessed 15-May-2023].

Jiang, A. (2023). A Gentle Introduction to Islands — deno.com. https://deno.com/blog/intro-to-islands. [Accessed 27-Apr-2023].

Kaluža, M., Troskot, K., and Vukelić, B. (2018). Comparison of front-end frameworks for web applications development. *Zbornik Veleučilišta u Rijeci*, 6(1):261–282.

Kurp, P. (2008). Green computing. *Communications of the ACM*, 51(10):11–13.

Marko (2023). Marko — markojs.com. https://markojs.com/. [Accessed 13-Apr-2023].

Marko Server Integrations (2023). Server Integrations — Marko — markojs.com. https://markojs.com/docs/server-integrations-overview/. [Accessed 19-Apr-2023].

Miller, J. (2019). Application Holotypes: A Guide to Architecture Decisions - JASON Format — jasonformat.com. https://jasonformat.com/application-holotypes/. [Accessed 10-Jan-2023].

Miller, J. (2020). Islands architecture. [Accessed 29-Sep-2022].

Nah, F. F.-H. (2004). A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163.

Ollila, R., Mäkitalo, N., and Mikkonen, T. (2022). Modern web frameworks: A comparison of rendering performance. *Journal of Web Engineering*.

O'Shaughnessy, P. (2018). Disappearing Frameworks — medium.com. https://medium.com/samsung-internet-dev/disappearing-frameworks-ed921f411c38. [Accessed 11-Apr-2023].

Petersen, H. (2016). From static and dynamic websites to static site generators. *University of Tartu, Institute of Computer Science*.

Phoenix LiveView (2023). Phoenix.LiveView - Phoenix LiveView v0.18.18 — hexdocs.pm. https://hexdocs.pm/phoenix_live_view/Phoenix.LiveView.html. [Accessed 27-Apr-2023].

Qwik (2022a). Overview - Qwik — qwik.builder.io. https://qwik.builder.io/docs/overview/. [Accessed 14-Nov-2022].

Qwik (2022b). Resumable - Qwik — qwik.builder.io. https://qwik.builder.io/docs/concepts/resumable/. [Accessed 14-Nov-2022].

Qwik Deployments (2023). Deployments - Qwik — qwik.builder.io. https://qwik.builder.io/docs/deployments/. [Accessed 19-Apr-2023].

Qwik React (2023). Qwik React - Qwik — qwik.builder.io. https://qwik.builder.io/docs/integrations/react/. [Accessed 19-Apr-2023].

Qwik's magic (2022). Qwik's magic is not in how fast it executes, but how good it is in avoiding doing any work — devm.io. https://devm.io/javascript/qwik-javascript-hevery. [Accessed 15-Nov-2022].

Severance, C. (2012). JavaScript: Designing a language in 10 days. *Computer*, 45(2):7–8.

Svelte (2023). Svelte - Cybernetically enhanced web apps — svelte.dev. https://svelte.dev/. [Accessed 13-Apr-2023].

Svelte Adapters (2023). Adapters - Docs - SvelteKit — kit.svelte.dev. https://kit.svelte.dev/docs/adapters. [Accessed 19-Apr-2023].

SvelteKit (2023). Introduction - Docs - SvelteKit — kit.svelte.dev. https://kit.svelte.dev/docs/introduction. [Accessed 13-Apr-2023].

SvelteKit Form Actions (2023). Form actions - Docs - SvelteKit — kit.svelte.dev. https://kit.svelte.dev/docs/form-actions. [Accessed 13-Apr-2023].

SvelteKit Page Options (2023). Page options - Docs - SvelteKit — kit.svelte.dev. https://kit.svelte.dev/docs/page-options. [Accessed 13-Apr-2023].

Tyson, M. (2022). Intro to Qwik: A superfast JavaScript framework — infoworld.com. https://www.infoworld.com/article/3676577/intro-to-qwik-a-superfast-javascript-framework.html. [Accessed 14-Nov-2022].

Vepsäläinen, J., Hellas, A., and Vuorimaa, P. (2023). The rise of disappearing frameworks in web development. In *International Conference on Web Engineering*, pages 319–326. Springer.

Vogel, L. and Springer, T. (2023). Waiter and autratac: Don't throw it away, just delay! In Garrigós, I., Murillo Rodríguez, J. M., and Wimmer, M., editors, *Web Engineering*, pages 278–292, Cham. Springer Nature Switzerland.

w3techs (2023). Usage Statistics and Market Share of jQuery for Websites, May 2023 — w3techs.com. https://w3techs.com/technologies/details/js-jquery. [Accessed 08-May-2023].

Wirfs-Brock, A. and Eich, B. (2020). JavaScript: the first 20 years. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–189.

Woychowsky, E. (2006). *AJAX: Creating web pages with asynchronous JavaScript and XML*, volume 8. Prentice Hall Upper Saddle River, NJ, USA.