The Potential of Serverless Edge-powered Islands for Web Development

Juho Vepsäläinen*, Petri Vuorimaa and Arto Hellas

Aalto University, Department of Computer Science, Finland
E-mail: juho.vepsalainen@aalto.fi
**Corresponding Author*

Received 01 July 2024; Accepted 24 November 2024

Abstract

Web developers face two significant challenges when developing their applications and websites: latency and payload size. Given that web services rely on servers, the related communication incurs a cost in terms of latency. In contrast, the payload passed to the client incurs a communication cost, not to mention the computational cost to the client. The concept of serverless edge computing, built on top of content delivery networks (CDNs), is an approach that has begun to gain the attention of web developers for its promise of lower latencies due to its efficiencies in communication thanks to globally distributed networks and replication. Islands architecture is a technical approach that addresses payload size by giving developers easy ways to defer and potentially even avoid the cost of loading content. Combined, these two approaches form edge-powered islands and, in this article, we examine how the combination can help to address these two notable costs web developers have to consider in their daily work. Our findings indicate that edge-powered islands can provide a way to introduce interactivity to otherwise static websites while wrapping dynamic portions of a page within

Journal of Web Engineering, Vol. 24_1, 1–38.
doi: 10.13052/jwe1540-9589.2411

islands to gain the benefits of static approaches in more dynamic contexts, such as storefronts. In addition, islands can provide loading benefits even for more application-like websites, such as social networks, and give web developers an additional control layer in their development work.

Keywords: E-commerce, edge-powered islands, islands architecture, performance, serverless edge computing, web applications, web development, World Wide Web.

1 Introduction

World Wide Web was introduced in the early 1990s as a global information interchange system [3]. Since then, the web has transformed into the most significant available application platform thanks to the advent of web applications, as the platform reaches two-thirds of the global population these days [18, 45]. To enable applications on top of the web, the platform had to be retrofitted to suit the purpose while the amount of dynamic functionality on websites has been growing. This growth has led to an increase in the size of web applications, where the median page weight has tripled in the last decade [17].

1.1 Increased Page Weight Leads to Waste

The increased page weight is a problem as slower sites lose conversions [2], not to mention the environmental impact [18] and the reduced usability with mobile phones and in countries with poor connectivity. Because of these reasons, there is a clear need for more effective solutions. Prior research has suggested approaches for removing dead or unused code [8, 23], precomputing parts of content on the server before delivery [32, 50], and delaying the execution of code until it is needed [53]. Latest web application frameworks also increasingly provide functionality that minimizes the amount of code the user initially needs to load [49, 51].

1.2 Serverless Edge Computing Represents a Potential Solution to Latency

The problem of latency compounds the problem of page weight further as the web relies on a client–server architecture where content is often

fetched from an external source. Content delivery networks (CDNs) [34, 48] formed an early solution to the problem of static content and, later on, edge computing [42] took the approach further by allowing developers to develop their code on top of the infrastructure. Edge computing decreases latency by allowing computation closer to the client [4] and, therefore, it offers an interesting value proposition for web developers. When combined with the ideas from serverless computing, we arrive at the concept of serverless edge computing that promises scaling benefits, especially for use cases with strict latency requirements [21] while providing a simple way for developers to provision their code [31].

1.3 Islands Architecture Allows Deferring Loading

Islands architecture is a recent (2019) approach for reducing the amount of code that needs to be initially loaded [19]. Complementing micro-front-ends [46], islands architecture [19, 29] allows building dynamic functionality on top of a static backdrop, for example, by loading the dynamic functionality only when the island comes or is close to the browser viewport. As a relatively new approach, there is little experience using islands architecture [15], but, simultaneously, there is a growing interest in frameworks, such as Astro, that implement islands architecture.

1.4 Edge-powered Islands – Edge Computing Combined with Islands Architecture

Combined, serverless edge computing and islands architecture form edge-powered islands. Through this combination, two significant costs encountered by web developers, latency and payload size, can be addressed effectively. This article explores this direction and its potential for web development. Therefore, we have formalized the research question of this article as follows: "What are the benefits of combining the islands architecture with serverless edge computing?"

1.5 Research Approach

To address the question we first consider the background and earlier related work in Sections 2 and 3 as we trace the main forces behind this combination before experimenting to test the edge-powered islands in practice.

To address the fact that web applications come in many shapes and sizes, we decided to leverage the idea of web application holotypes[1] introduced by Miller [28] and then simplified by Carniato [6] to evaluate edge-powered islands through five types: portfolio, content, storefront, social network, and immersive. It can be argued that the list of holotypes is not fully exhaustive. Still, simultaneously, this evaluation should give us an initial idea of where to get the most benefit out of edge-powered islands.

The idea of holotypes and the holotypes considered in the context of this discussion is covered in Section 4. Each holotype is considered separately in the following Sections 5, 6, 7, 8, and 9. The implications of our findings are discussed in Section 10 and concluded in Section 11.

2 Serverless Edge Computing

While CDNs brought servers close to the client and enabled low latencies [34, 48], edge computing does the same for computation as pointed out by Shi et al. [43]. Because of this, it is natural for CDN-oriented companies to develop edge infrastructure on top of their CDN offerings. In recent years, companies such as Akamai, Alibaba, Amazon, Cloudflare, Deno, Equinix, Fastly, Fly.io, and Netlify have brought their services to the market, especially the web space, which is expanding. Service offerings tend to differ, and there are underlying differences in implementations and how many capabilities they expose to developers.[2] At the time of writing, no uniform programming interface that works on all platforms exists, and some adaptation is required when moving from one platform to another. To harmonize the development, the WinterCG community group was established in 2022 to avoid platform fragmentation and provide standard APIs for developers.

2.1 Edge Computing Platforms are Programmable CDNs

Satyanarayanan [42] characterizes edge computing platforms as programmable CDNs because they allow developers to define what happens on a request instead of serving only static assets. At the most superficial level, programmability allows developers to use edge computing platforms as middleware for shaping requests towards application servers. Taken further,

[1] Holotypes are typically used in zoology to categorize species [36], and the same idea seems to work well for researching web applications.

[2] For example, Cloudflare provides a strict sandbox where developers can write their code while Fly.io relies on a container-based approach.

edge computing can replace a traditional server altogether as it can perform necessary computations directly on the edge and close to the user without the need of a conventional server. The distributed nature of edge computing brings new challenges, as conventional technologies, such as databases, have not been designed with large-scale distribution in mind.

2.2 Serverless Edge Computing Builds on Top of the Edge and Serverless Computing

Regardless of the computing approach, the provisioning and utilization of computing resources should be considered. How these aspects are handled depends on the general approach and the amount of automation in the process, as highlighted by Kounev et al. [21] in their discussion of what makes serverless computing genuinely serverless. As an analogy, consider owning versus renting a car. In the former approach, you assume full responsibility for the vehicle; in the latter, the responsibility is limited. Different computing approaches are precisely about this as they offer different responsibility models at various price points and offerings. It is up to developers to determine which offering matches their use case, while vendors push their benefits to gain business and utilize their resources. In the extreme case, the developer would have to set up their hardware and maintain a significant portion of the stack while considering scaling and other complex problems.

The function as a service (FaaS)[3] model [21] is interesting because serverless providers expose a function as a unit of computation and a programming interface using the approach. Although the model has a narrow scope, it allows vendors to provide benefits such as automatic scaling and a granular billing model, as it can be billed per execution [21]. In this article, we focus on serverless edge computing [21] since it inherits many of the benefits of serverless computing while shaping it further on the level of infrastructure to gain scaling benefits, especially for use cases with low latency requirements. As highlighted by Raith et al. [40], numerous serverless edge computing platforms are available, and the space is evolving rapidly as the approach is gaining maturity and best practices are established.

Fly.io forms an exciting counterpoint to serverless edge computing as it is a service built around the idea of software containers [20] that developers provide for the platform to deploy on top of their global infrastructure.

[3]FaaS is not the only example of serverless, and my other models with different levels of capabilities exist, but that goes beyond this discussion. The various models are covered in great detail in [21].

Therefore, the service is not serverless but fulfills the essential criteria behind an edge platform. In the context of this discussion, we won't cover Fly.io in detail, but it would be an attractive target to study in detail as a comparison point.

2.3 Beyond Edge Computing

Although edge computing provides substantial benefits, Nastic et al. [31] note that to fully unlock the potential of edge computing, you have to combine it with cloud computing. On a general level, fog computing [57] captures the combination and imagines edge computing as an extension of the cloud. The combination comes with new challenges related to service level objectives (SLOs), data management, reliability, application development, artificial intelligence, and edge intelligence, as pointed out by Nastic et al. [31]. In this article, we focus on a narrow slice of the field and touch specifically on application development concerns of the challenges by highlighting how islands architecture can allow developers to partition their logic on top of the serverless edge.

2.4 Latency Benefits of Edge Computing over Cloud Computing

As highlighted by the measurements of Charyyev et al. [9], edge computing can provide considerable latency benefits to the end-users as, in their measurements, edge servers offer lower latency to 92% compared with the cloud. Interestingly, Charyyev et al. [9] noted that in regions abundant with data centers, the cloud can match the performance of the edge, meaning it matters where your users are. The critical point is that edge computing can provide meaningful benefits for use cases where latency makes a difference. In the context of this article, we won't measure the difference between the edge and the cloud separately but rather accept that the approach comes with latency benefits while being more limited as a development target due to a reduced amount of control.

2.5 Conclusion

The way we define it, serverless edge computing could be seen as an approach that combines the benefits of serverless with CDNs to gain global scaling while providing developers with an easy provisioning model. However, having the ability to compute on the edge is not enough for most practical problems, as databases may have to be accessed. For this reason, the space

is under active research as solutions are explored for specific issues, such as efficient data access on the edge, not to mention aspects like application development practices, which we touch on in the next section through the specific example of islands architecture.

3 Islands Architecture

Websites, particularly web applications, require a certain amount of interactivity to be useful. Although it would be possible to develop applications using a server-driven approach where the state is persisted on the server and updated on page refresh as in the early multi-page application (MPA) [44] model, that does not match the contemporary expectations of users. Partly due to this reason, single-page applications (SPAs) [44] gained popularity as they allowed the development of complex web applications although with a dependency on JavaScript and challenges for search engine optimization (SEO) [49].

The main difference between early and later approaches to web application development is how dynamic functionality is treated. In the early approaches, servers did most of the work, while the responsibilities were shifted to the client over time. Over the past decades, multiple attempts have been made to capture dynamic functionality within specific containers. The most important of the attempts were portals and widget-based web architectures before arriving at islands architecture.

3.1 Portal-based Web Architectures

So-called portal-based approaches form a path of development that has been attempted several times during the history of the web. Portals capture interactivity within specific elements while the remainder of a page can remain static, as in the classic web, and MPAs as described by Wege [55]. Portals [55] from the early 2000s implement the approach through the concepts of portlets and portlet containers. The approach was based on Java, J2EE to be specific, and relied on Java-based servers that ran the associated logic. In other words, the approach was highly coupled to a specific server environment.

3.2 Widget-based Web Architectures

Widget-based approaches were another attempt at formalizing containers that capture interactivity. Early widget approaches allowed blurring of the

boundaries between local and web-based applications and bypassing the strict security sandbox of the standard web by working on the operating system level as pointed out by Mendes [27]. Mäkelä et al. [26] mentions mapping applications as a potential example and this has materialized in the form of the usage of the `iframe` element that allows embedding content from third parties, although with some security risk [47]. It is good to note that iframes are relatively old as they were introduced as early as 1997 [39]. Besides the security concerns illustrated by Tian et al. [47], the main drawback of the approach is the lack of customizability, although that can be solved to a limited extent. Still, it can be challenging to achieve a high level of integration using iframes due to technical constraints.

Web components form another, more flexible way to model widgets as they give more flexibility to the developer, as noted by Krug and Gaedke [22]. A substantial benefit of the approach is that it is not coupled to a specific technology stack [22]. W3C Widgets [56], and OpenSocial [30] are earlier approaches and, as pointed out by Krug [22], have heavier technical constraints than Web Components.

3.3 Introduction of Islands Architecture at Etsy

In 2019, Etsy, an e-commerce platform, had to support both user and vendor sites with differing interactivity requirements and different technologies, contributing to friction in the development process as code could not be shared easily [19]. To solve the problem, Etsy introduced islands architecture to capture islands of interactivity associated with a loading strategy [19]. In Etsy's case, islands were modeled using JSX and Preact[4] while the rest of the markup was still rendered using PHP as earlier [19].

Islands architecture was popularized in 2020 by Miller [29] and later by Hallie and Osmani [15]. Compared to the earlier attempts, islands architecture is technology agnostic on the server side. At the same time, at the client, there is a light dependency on JavaScript as it is required to model loading strategies. Although the approach can be implemented independently, it began to gain popularity as Astro meta-framework[5] (2021) implemented the architecture out of the box and made it easy for developers to adopt while

[4] Preact is a light version of the popular React library.

[5] By meta-framework, we mean a framework that can work with other libraries to leverage their ecosystems. For example, Astro allows developers to use React, Vue, and others within the same project through adapters.

allowing usage of other solutions, such as React, Vue, or Svelte, and letting developers leverage their component libraries.

3.4 Elements of Islands Architecture

Essentially, islands allow capturing interactive functionality on an otherwise static page [19] by letting developers mark dynamic portions as islands to be loaded using a specific strategy. Splitting interactive functionality to islands enables developers to define loading strategies, such as "load island when in view"[6] to conserve bandwidth and to keep initial page payload under control [19]. Around these axioms, various implementations have appeared to support adopting the idea, and these have been collected in Table 1.

Table 1 Islands architecture-related frameworks and libraries

Name	Type	Version	Description
is-land	Library	4.0.0	Small (5 kB) runtime with SSR examples for Lit, Svelte, Vue, and Preact.
Astro	Framework	4.16.6	Meta-framework with broad support for user interface libraries, including options such as Lit, Svelte, Vue, React, and Preact.
Capri	Framework	5.2.3	Meta-framework with a strong focus on static sites and broad support for user interface libraries, including React, Preact, Svelte, Vue, and SolidJS.
Fresh	Framework	1.7.1	Deno-focused islands architecture framework leveraging Preact.

Given the interest in the approach is constantly growing, more implementations exist, although some have already been abandoned. The awesome-islands[7] link collection captures several more, including articles and other related resources.

3.5 Benefits and Challenges of Islands Architecture

As summarized by Vepsäläinen et al. [49], the benefits of islands architecture include the avoidance of top-down rendering, improved performance as there is less JavaScript to ship, less problematic SEO due to static content, better accessibility and discoverability by default, and component orientation that

[6] In practice, more strategies exist, and partially, it is up to the imagination of developers to figure out how they prefer to load their interactive code. Most importantly, islands architecture can help to avoid work altogether by not triggering specific portions of application logic.

[7] https://github.com/lxsmnsyc/awesome-islands

encourages reuse. It can be argued that composition might not be trivial in all cases due to island boundaries, as SEO issues may arise within them.

Given islands architecture is still relatively new, there is not much experience in using it, and it is possible that it is not the right choice for highly interactive use cases, such as a heavily personalized social media applications, as it would require a large number of islands, perhaps defeating the purpose [15]. Therefore, there is likely some sweet spot for islands, and it seems natural to use islands to sprinkle interactivity to otherwise static pages as in the case of Etsy [19].

Although the basic idea of islands architecture is simple, frameworks have implemented several improvements to consider more complex use cases. For example, Fresh 1.2 added support for nested islands, allowing static and dynamic content to be nested while implementing a story to enable the application to state to be shared between islands easily [7]. In Fresh 1.5, the framework added support for partials that allow retaining a part of a page during navigation to avoid reloading related flashes as users access pages [14]. Starting from Astro 4, the framework introduced specific client-side tooling to ease the development of islands directly in the browser [13].

It is important to note that islands are, by definition, JavaScript-dependent. In other words, they should not be used for critical functionality without considering users who haven't enabled JavaScript in their browsers. This limitation highlights the purpose of islands, as they should be used to capture dynamic or secondary functionality.

The idea of islands can be taken further by changing the approach entirely. For example, the Qwik framework goes further with its implementation of resumability as pointed out by Vepsäläinen et al. [50], and the framework takes care of code-splitting internally [51]. Due to its architecture, Qwik automatically creates islands of functionality and loads them on demand by default while allowing developers to control the behavior further [51]. In this article, we focus on the implications of islands architecture in its pure form, as the architecture can be implemented easily in existing contexts while gaining its benefits while acknowledging that the feature could be implemented transparently within a web framework.

3.6 Connection of Islands to Micro-frontends

Micro-frontends form a related approach as micro-frontends allow developers to decompose the frontend into individual, semi-independent micro-applications as defined by Taibi and Mezzalira [46]. Furthermore,

micro-frontends enable teams to independently develop sections of an application in a full stack manner [46]. As put by Miller [29], although the approaches look similar, the composition of independent units occurs specifically via HTML, which may not always be true for micro-frontends. While micro-frontends could be considered a development pattern, islands are a technical one related to performance. Conversely, performance may be an issue in the micro-frontend approach due to the cost of the first-page load, as pointed out by Prajwal [37].

Due to this difference, the approaches come with different tradeoffs. As hinted by Taibi and Mezzalira [46] and Prajwal et al. [37], there may be duplicated code across micro-frontends, and it is not always clear when to develop abstractions as there is a cost involved in creating wrong abstractions. Another issue mentioned by Taibi and Mezzalira [46] concerns the maturity of micro-frontends, as the approach has not been fully defined yet and has not been investigated in detail yet.

3.7 Conclusion

Islands architecture represents the latest take on widget and portal-style web architectures. Although the basic idea of encapsulating interactivity remains the same, the approach is more technology-agnostic than earlier ones. With the advent of JavaScript frameworks like Astro, islands architecture has become accessible to an increasing amount of developers. As a new approach, there is still experimentation in the space.

To show how islands architecture differs from other similar technologies, Table 2 captures their main characteristics. It is good to note that islands architecture is meant explicitly to improve the loading behavior of an application, while other approaches have partially differing targets. It may also be beneficial to use multiple approaches within the same application. For example, third-party widgets may still help embed external content like maps or video.

4 Web Application Holotypes

As described by Miller [28], it is easy to broadly generalize web application performance based on single anecdotes or statistics. The target of web application holotypes[8] is to capture common categories for web applications

[8]The term holotype comes from zoology and has been defined by [36] as “a single type specimen upon which the description and name of a new species is based.”.

Table 2 Comparison of islands architecture against earlier approaches

Approach	Backend	Frontend	Purpose
Islands architecture	Not specified	JavaScript is needed for loading the islands	Optimization of application loading behavior
Micro-frontends	Not specified	Depends on the implementation	Allowing multiple teams to develop separate sections of the same application independently using their preferred technologies
Portals	Java	Optionally JavaScript as needed [35]	Optimization of application loading behavior and potentially sharing portlets across applications [55]
Widgets	Not specified	Standard web technologies (CSS, HTML, JavaScript). Using iframes is an option [47].	Sharing functionality with third-parties [26] and potentially wrapping specific functionality beyond a web browser within a container of its own [27].

to produce specific recommendations and insights [28]. The original holotypes of Miller [28] contained twelve to consider, and Carniato [6] simplified the list to five, namely portfolio, content, storefront, social network, and immersive applications. Table 3 explains these five holotypes in greater detail.

Table 3 This slightly modernized table based on Carniato [6] illustrates different holotypes and their specific features in terms of examples, interactivity, session depth, values, routing, rendering, hydration, and frameworks to show how they differ from each other

Holotype	Portfolio	Content	Storefront	Social Network	Immersive
Example holotype	Personal blog	BBC, CNN	Amazon, Etsy [19]	Facebook, Instagram, X	Figma, Google Docs
Interactivity	Minimal	Linked articles	Purchase	Multi-point, real-time	Everything is interactive
Session depth	Shallow	Shallow	Shallow to medium	Extended	Deep
Values	Simplicity	Discoverability	Load performance	Dynamicism	Immersiveness
Routing	Server	Server, HTML swap	HTML swap, hybrid	Hybrid, client	Client
Rendering	Static	Static, SSR	Static, SSR	SSR	CSR
Hydration	None	Progressive, partial	Partial, resumability	Any	None (CSR)
Example frameworks	11ty	Astro, Elder	Marko, Qwik	Next, Remix	Create React App, Vite

The list is broad enough to give us some insight into the applicability of edge-powered islands, as it covers enough of both dynamic and static cases to demonstrate the ideal use cases and limitations for the architecture.

It is good to remember that many websites are composites of multiple holotypes, as they might have separate sections with specific purposes. That said, holotypes provide a natural way to consider their technical requirements in isolation and give us a way to evaluate how islands might work for a particular use case.

5 Portfolio Sites

As an example of the portfolio holotype, Carniato [6] mentions a personal blog with minimal interactivity requirements. As further noted by Carniato [6], a small blog can rely entirely on static site generation (SSG) and comes with little technical complexity. Given performance and loading time are essential metrics for portfolio sites, techniques like SSG are a natural choice. Relying on static techniques comes with limitations; therefore, edge-powered islands may be a good way to go beyond purely static. For example, edge-powered islands could be used to develop standard blog features, such as commenting. In the case of commenting, comments could exist in a dynamic widget rendered within an island as the user loads the page or decides to load the comments by intent using a specific user interface control to defer loading.

5.1 Benchmarking Edge-powered Islands at Portfolio Sites

To benchmark edge-powered islands at portfolio sites, we set up a small example[9] that consists of a blog with generated content and a comments section per blog page. It is the comments section that is an ideal use case for an island, and we benchmarked it as follows:

1. There are four variants to compare: vanilla, Disqus,[10] lazy Disqus, and islands. The variants are described below to help you better understand how they have been implemented.
2. All variants are implemented using TypeScript.

[9]The benchmark source is publicly available at https://github.com/bebraw/islands-benchmark to review and improve further.

[10]Disqus is a commonly used third-party platform for commenting that is easy to integrate using a `script` tag.

3. All variants use ES2015 templates for templating.
4. All of the variants have been hosted on top of Cloudflare platform and have been implemented as Cloudflare Workers.
5. Images are kept out of scope to keep the test case simple and to avoid loading costs. Similarly, styling has been kept to a minimum apart from minimal aesthetic tweaks.
6. To illustrate the benefits of using islands, we included twenty short pre-existing comments for the vanilla and islands variants to show the difference between the two.

5.2 Description of Test Variants

The test variants were chosen to give a comprehensive view of how islands architecture compares to other approaches and what intermediate solutions, e.g., lazy loading, might exist. A Disqus variant relying on a third-party service was included to provide a contrast, as it is a commonly used service for commenting.

The so-called vanilla variant uses a Cloudflare worker to render an HTML page, including page content, a commenting section, and a standard HTML form to send a new comment to give a baseline.[11] When adding a new comment, the corresponding Cloudflare worker endpoint will store the latest comment in Cloudflare KV[12] database. Assuming the request went through, the endpoint will force a simple redirect so the new comment is visible on the page.

The Disqus variant is the simplest of the four, as it has been implemented using a third-party script embed, which takes care of rendering comments and the commenting portion. This embedding has been set up per instructions on the Disqus platform, and in this case, a Cloudflare worker is used only for the initial rendering of the page. The lazy Disqus variant goes a notch further by adding an HTML button that loads the Disqus script only after the button is pressed, thereby deferring the loading action.

The island variant builds on the idea of a loading button. Upon pressing, it triggers a custom Cloudflare worker endpoint returning HTML, including comments related to a post and a form to add a new comment. Here, the commenting logic follows the same logic as the vanilla variant. As a further

[11] It would have been possible to use a different provider for static hosting, but that would have made the comparison more difficult due to differences in infrastructure and resulting latencies.

[12] https://developers.cloudflare.com/kv/

optimization, avoiding a full page refresh after adding a comment would be possible, but this is unnecessary for this benchmark. Also, the payload could be optimized using a format other than HTML.

Figure 1 shows how the variants differ on a high level, as there are subtle differences in their implementations and how HTML and JavaScript are used. The vanilla variant relies only on the server for its implementation, while the Disqus and the island variants require some JavaScript.

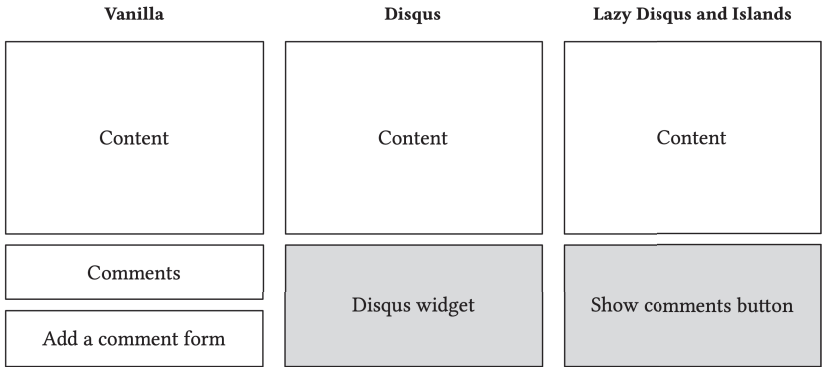

Figure 1 The figure shows how the portfolio test variants were implemented at a high level. Each variant had a content section below, in which commenting-related logic was implemented. The white portions of the image signify the usage of HTML, while the gray ones have been implemented using JavaScript.

5.3 Measurement of Performance

To measure performance, we used Playwright[13] with Google Lighthouse,[14] a tool that is widely used within the industry. Lighthouse reports plenty of data, and for this benchmark, we chose to showcase the following values[15]: First contentful paint (FCP), largest contentful paint (LCP), cumulative layout shift (CLS), total blocking time (TBT), interaction to next paint (INP), and bytes transferred.

Heričko et al. [16] recommend running Lighthouse five times to reduce the variability of the results, and this is confirmed by Lighthouse

[13] https://playwright.dev/

[14] https://developer.chrome.com/docs/lighthouse/

[15] Walton [54] explains the meaning of these values in greater detail in his online article. Each value measures a specific portion of website loading behavior, and Lighthouse can also consider dynamic behavior, as we will see in Section 7.

documentation [24], which suggests measuring the median of five runs leads to twice as stable measurement results. We used ten runs to play it safe and add more stability to our test results. We ran Lighthouse tests using two profiles included with the tool: lr-desktop and lr-mobile. The profiles were run using the Chrome 130.0.6723.31 browser. The mobile profile throttles the connection speed to match a slow 4G connection and a low-power mobile device with a 4x slowdown compared to the machine's regular speed. The desktop profile does not have these constraints.

5.4 Benchmark Results

From Table 4 illustrating our desktop results, it is visible that the Disqus variant has to transfer the most data, roughly 709 kB, while the other variants are much lighter at the two-kilobyte range. The large size informs us of the poor optimization of the third-party Disqus widget. The islands variant stands out as the lightest, and the lazy Disqus variant follows this behavior as it defers work. The vital thing to note is that the difference between the vanilla and the islands variants depends on the number of comments visible on the page, and assuming there are no comments, their results would likely be close to the same. In other words, the more content a page has, the bigger the savings in the island approach.

Mobile behavior seems to follow desktop behavior, as visible in Table 5. Throttling of mobile computing power and bandwidth is visible in the results as the time-related values are slower. The main difference is that TBT and INP are exceptionally high for the Disqus variant, implying that a large amount of data to load can impact the value, especially on mobile. This observation supports the importance of optimizing the initial load and deferring work.

Table 4 Disqus variant stands apart from the others, particularly due to the amount of bytes it transfers and in high INP. The islands variant has to transfer the least data as it defers the work, and the same behavior is visible in the lazy Disqus variant

Variant	FCP	LCP	CLS	TBT	INP	Bytes Transferred
Vanilla (average)	357.7	357.7	0	0	357.7	1766
Vanilla (median)	361.5	361.5	0	0	361.5	1766
Disqus (average)	448.2	448.2	0	4.1	1237.3	709658.7
Disqus (median)	440	440	0	0.5	1063.5	709658.5
Lazy Disqus (average)	357.3	357.3	0	0	357.3	1662.2
Lazy Disqus (median)	394	394	0	0	394	1643.5
Islands (average)	351.3	351.3	0	0	351.3	1622.6
Islands (median)	333.5	333.5	0	0	333.5	1619.5

Table 5 The mobile results follow the desktop ones. The notable difference is that TBT and INP are particularly high for the Disqus variant. This slowdown likely has to do with the fact that the third-party Disqus widget is heavy to load, and in a more limited context, there is a concrete cost included

Variant	FCP	LCP	CLS	TBT	INP	Bytes Transferred
Vanilla (average)	973.1	973.1	0	0	973.1	1770.2
Vanilla (median)	1038.5	1038.5	0	0	1038.5	1772.5
Disqus (average)	991.2	991.2	0	309.1	6753.3	709658.6
Disqus (median)	991	991	0	311.5	6757.5	709642
Lazy Disqus (average)	952.8	952.8	0	0	952.8	1660.1
Lazy Disqus (median)	911.5	911.5	0	0	911.5	1645.5
Islands (average)	929.6	929.6	0	0	929.6	1635.8
Islands (median)	912.5	912.5	0	0	912.5	1624

6 Content Sites

Based on Carniato [6], a content website differs from a portfolio site by having stricter requirements for discoverability and linking content together. A part of the complexity comes from the expected workflows that a content site might have to support, as there might be a specific way to author content without worrying about a site's technical portions. Content management systems (CMSs) encapsulating these types of flows within the solution are a good example.

Compared to portfolio sites, content sites are technically more diverse as they have more substantial requirements around authoring content and explicitly allowing users to discover it quickly [6]. Content sites can vary from small systems to extensive ones, such as those maintained by national broadcasters. The scope alone may rule out solutions amenable to portfolio sites and point towards more robust and technically complex options. While loading times are essential, content sites have constraints above that increase complexity and force some dynamic functionality to be implemented, at least on the authoring side.

As with portfolio sites, islands can capture dynamic functionality and help to maintain a boundary between static and dynamic. Furthermore, islands may be used to deliver part of the content, assuming there are heavy requirements on how fast content should become available, not to mention possibilities for aggregation. For example, an island could contain a list of the site's top ten most-read articles while being rendered on the server and aggregating the listing based on site analytics.

Content sites typically rely on some form of advertising to fund their operation, showing third-party advertisements to their users. A common approach is to use a third-party script that handles bidding [1] and then shows the winning ad to the user. As this type of bidding typically occurs on a third-party server that resolves the logic, islands are not a direct fit for the problem, but lazy loading can still be a good idea to speed up initial page loading. Islands could still work for ads handled by the content platform, where bidding can be avoided.

6.1 Using Edge-powered Islands at Content Sites

Figure 2 shows how islands could be used in collaboration with edge-side includes (ESI)[16] [33, 38] to segment a news page into smaller portions to load while gaining benefits of caching at different levels. While the page content is static, dynamic portions are overlaid on top of it while addressing mutability and customization requirements. For example, ESI is suggested for site navigation in this particular case. Navigation will likely change faster than the content itself and should be bound with the content at the edge to gain caching benefits. Page sections related to account handling, most read articles, and the latest articles are natural fits for islands as they are secondary content that can be loaded on demand after the initial loading of the page.

The critical thing to note is that ESI and islands are complementary techniques that give control over what kind of content is bound together and when. Interestingly enough, Rabinovich et al. [38] proposed using client-side includes (CSI) as early as 2003 to move ESI to the client, and one could argue that islands are an implementation of this old idea with modern development ergonomics.

In this use case, islands allow secondary work to be deferred. As in Section 5, the savings depend on the amount of data that can be loaded later, and for text content, that might be quite a trivial amount. The true benefit might be in caching as, due to the separation, there is less to cache and less chance that cached data would be invalidated. As an added benefit, the islands could be personalized, and this use case is more visible in Section 8 where using islands with social networks is considered.

[16] ESI allows composition of pages from smaller fragments at the edge using a specific syntax to bind content from multiple sources together to gain caching benefits since often different parts of a page age at different speeds [33, 38].

6.2 Benchmarking Edge-powered Islands at Content Sites

To benchmark edge-powered islands with content sites, we followed the testing procedure used in Section 5 while implementing a minimally styled version of the view shown in Figure 2. To keep the benchmark simple, we implemented SSR and islands variants. The SSR variant rendered all of the content on the edge, while the islands variant deferred loading the most read and latest posts so that the contents of these islands were loaded only after the initial DOM structure had been rendered. Another simplification we made was to test only against the mobile profile, as using it provided enough data to consider.

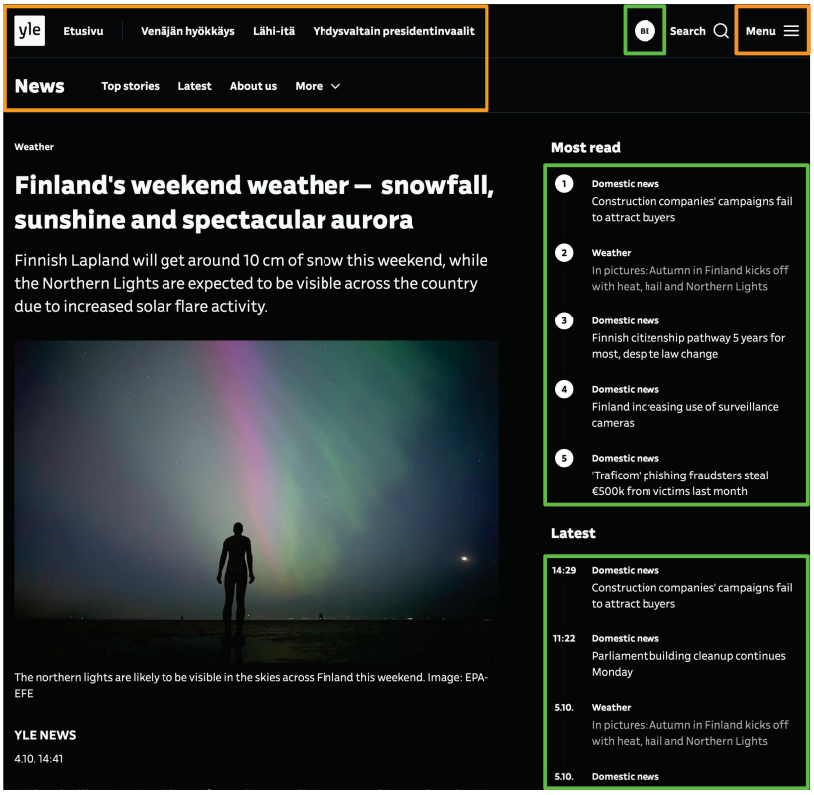

Figure 2 The figure illustrates how edge-side includes (ESI) (marked with orange) and islands (marked with green) could be used to segment a news-related page into smaller portions to load either on the server or client. It is important to note how the techniques complement each other and give developers the means to define what is loaded, where, and when.

Table 6 shows that the islands variant transfers roughly 50% more data while being about 10% faster in FCP and LCP metrics than the SSR variant. The difference is due to more efficient compression for the SSR case. In practice, the difference could be more minor, assuming a more efficient way to transfer island data is used instead of the HTML used in the islands variant. The islands variant also contains some layout shift that could be eliminated, for example, by fixing element heights.

Table 6 The mobile results show that the islands variant transfers roughly 50% more data while being about 10% faster in FCP and LCP metrics than the SSR variant. The islands variant shows some layout shift, likely due to the dynamic loading behavior causing a reflow

Variant	FCP	LCP	CLS	TBT	INP	Bytes Transferred
SSR (average)	862.6	862.6	0	0	862.6	2066.6
SSR (median)	877	877	0	0	877	2061.5
SSR with islands (average)	809.3	809.3	0.15	0	809.3	3420.2
SSR with islands (median)	791.5	791.5	0.16	0	791.5	3411

7 Storefronts

Storefronts, such as Amazon or Etsy, have specific requirements related to discoverability and performance, as pointed out by Miller [28]. While storefronts should be possible for search engines to discover, they must also support potentially complex purchasing-related workflows to fulfill their primary purpose. Unlike content sites, storefronts present unique challenges, including storefront-specific logic and complexity. Simultaneously, client-facing portions must be highly optimized for revenue reasons as highlighted by Saleem and Di [12, 41]. Therefore, loading times should be optimized while keeping content, for example, inventory amounts, up to date, meaning there may be conflicting goals to optimize against. Storefronts may vary from simple shops selling a few products to complex and global ones like Amazon.

Compared to content sites, storefronts go a notch further as they have to support discovery and purchasing-related flows. For this reason, storefronts usually provide robust search interfaces that allow users to browse products before choosing which ones to buy. If the product to sell is simple, it may be enough to use a third-party provider and an associated integration, such as a JavaScript library, that completes the payment flow and related logistics.

Islands architecture may be leveraged at different levels of a storefront to capture dynamic functionality. Assuming a storefront implements a discovery flow against its inventory, islands can be used to capture the dynamic portions

of a search interface and results handling, in particular, since results depend on user input.

7.1 Benchmarking Edge-powered Islands at Storefronts

For storefronts, perhaps the most meaningful benchmark involves comparing full SSR against SSR with islands on top for dynamic functionality. For this purpose, we created a small discovery view where products may be fetched from a database based on user input.[17] To be exact, we followed the constraints below:

1. There are two variants to compare: SSR and SSR with islands.
2. All variants are implemented using TypeScript.
3. All variants use ES2015 templates for templating.
4. In the SSR variant, the idea is to return a new page with the search results after the user has given input and submitted it to the server to our Cloudflare endpoint. This has been set up to mimic how the Amazon storefront works.
5. In the SSR with islands variant, we update the browser query to include the user query and refresh specifically the results island after the user has sent their query to our Cloudflare endpoint, loading only a portion of the page related to the search results.
6. Both variants include an additional 1.4 kB HTML markup payload beyond the minimal markup needed for implementing the search functionality. Otherwise, the islands variant would be larger due to runtime requirements, making it difficult to show the potential benefits.
7. To keep the implementation simple, the database has been integrated into a TypeScript module as a simple data structure to search against since there are no write requirements. In practice, an implementation like this would rely on a database, adding latency to the response.
8. Both variants have been hosted on top of Cloudflare platform and have been implemented as Cloudflare Workers.
9. Images are kept out of scope to keep the test case simple and to avoid loading costs. Similarly, styling has been kept to a minimum apart from minimal aesthetic tweaks.

To capture an essential portion of how storefronts work, we implemented a simple search view that allows the users to query product data. As illustrated

[17]The benchmark source is publicly available at https://github.com/bebraw/islands-benchmark to review and improve further.

by Figure 3, the SSR variant relies entirely on the server for logic, while in the SSR with islands variant, the island handles user queries dynamically without a page refresh.

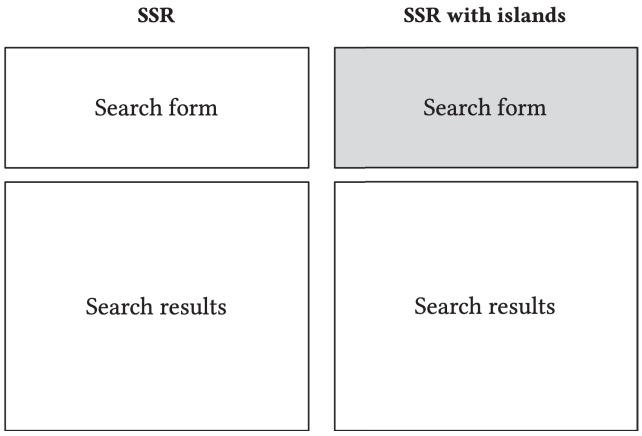

Figure 3 The figure shows how the storefront test variants were implemented at a high level. The SSR variant relied entirely on the server for logic. In contrast, in the SSR with the islands variant, the island took over the search-related responsibilities to avoid full-page refreshes during searching. The island is marked gray in the figure as it relies on JavaScript to update search results dynamically.

7.2 Measurement of Performance

For measuring performance, we used a similar setup as in Section 5 earlier and leveraged Google Lighthouse to capture FCP, LCP, CLS, TBT, INP, and bytes transferred while running tests ten times to reduce the variability of the results. The main difference is that, instead of Playwright, we used Puppeteer[18] to simulate a browser to enable the usage of Lighthouse user flows to measure the dynamic search functionality in aggregate. Since Puppeteer is bundled with an older version of Chrome browser, version 129.0.6668.100 was used instead. Compared to the regular way to use Lighthouse, user flows allow capturing what happens as the user navigates on the website instead of only page load, making the feature valuable for this particular benchmark [25]. To keep this benchmark simple, we used only the mobile profile.

[18]https://pptr.dev/

7.3 Benchmark Results

It can be seen from Table 7 that, in our test case, where pure SSR implementation of the search view is compared against an island-powered one, the SSR with islands variant is consistently faster as it has to transfer fewer data. However, additional island logic is included in the initial payload, meaning a marginal cost is associated with using islands. The SSR variant transferred roughly twice as much data as the islands variant due to the full page refresh required for the search action used in our benchmark.

Table 7 The mobile results show that the islands variant transfers significantly fewer data than the SSR variant while performing better, especially in FCP, LCP, and INP metrics. This is logical because the SSR variant has more work due to a full page refresh. The islands variant avoids most of the work by being able to update the search results

Variant	FCP	LCP	CLS	TBT	INP	Bytes Transferred
SSR (average)	1756.1	1781.8	0	5.6	1807	3384
SSR (median)	1749	1775	0	1	1800	3335
SSR with islands (average)	921.3	921.3	0	0	921.3	1971.7
SSR with islands (median)	919	919	0	0	919	1977.5

8 Social Networks

Compared to the earlier holotypes, social networks come with heavier requirements for interactivity while having strict demands for SEO to allow discoverability through search engines [6, 28]. Social networks can be highly technical, embodying complex content reading and writing requirements. The content delivered by its users must be effectively provided to other users. Due to this, social networks often leverage SSR as their rendering technique [6] to address the problem of constantly changing pages. Additional complexity is provided by standard functionality related to notifications. For example, to keep the network active, social networks have to track content relevant to their users and activate them to feed network effects. While loading times are important, they may not be as essential as the other holotypes. It also may be enough that the content is eventually consistent as there may not be strict requirements when it arrives to the clients.

Due to the heavy demands for interactivity, social networks seem like a questionable fit for edge-powered islands. However, it may be possible to apply the technique selectively in specific places of a social network application. Even then, it can be argued that the main benefits of deferring work and potentially avoiding it altogether may not be easy to reach.

To illustrate how islands could be used in the context of social networks, Figure 4 shows how a generic social network application could be decomposed as islands of interactivity. Through decomposition, an application could be loaded selectively, emphasizing its most important functionalities, such as content feed, while loading the rest of the functionality later. The implication is that islands could likely be developed as small entities of their own, consistent with the ideas behind micro-frontends.

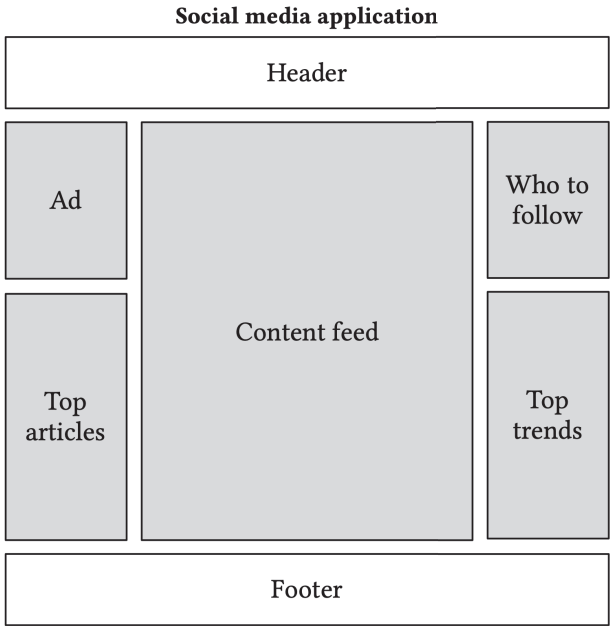

Figure 4 The figure shows how social networks could leverage islands. The identified islands were content feed, ad block, top articles, who to follow, and top trends marked with gray in the image. The site header and footer were considered static content and marked with white color.

8.1 Benchmarking Edge-powered Islands at Social Networks

For benchmarking, we leveraged the setup from Section 5 and set up SSR and islands variants as per Figure 4. Given the dynamic nature of social media applications, this is a crude approximation, as the content feed should be dynamic by definition and require some JavaScript code for proper implementation. In our test case, the whole page is static for the SSR case, while for the islands case the dynamic portions marked gray in Figure 4 are

loaded only after the initial loading of the page has been completed. For the islands variant, the implementation could be improved further by making the content feed alive by pushing new items from the server dynamically using WebSockets, for example. Our tests were done using Lighthouse's mobile profile, which is enough to capture the difference between the two variants.

Table 8 shows that the islands variant transfers roughly 100% more data than the pure SSR variant due to HTML compression. The island payloads are delivered in HTML for simplicity and could be optimized further by using a more efficient format. Interestingly, the islands variant performs roughly 10% better for FCP and LCP metrics, while it comes with a cost at CLS. The layout shift is caused by loading the island content and implies that the content size should likely be fixed to a specific height using CSS, for example.

Table 8 The mobile results show that the islands variant transfers roughly 100% more data in total than the SSR variant but performs roughly 10% better than SSR in FCP, LCP, and INP metrics. The islands variant exhibits layout shifts that could be eliminated by fixing island heights using CSS

Variant	FCP	LCP	CLS	TBT	INP	Bytes Transferred
SSR (average)	938.9	938.9	0	0	938.9	1858.3
SSR (median)	910	910	0	0	910	1860
SSR with islands (average)	809.6	809.6	0.56	0	809.6	4118.2
SSR with islands (median)	809.5	809.5	0.75	0	809.5	3865

9 Immersive Applications

Immersive applications go beyond social networks regarding interactivity requirements [5]. Immersive applications are essentially dynamic and could be considered the opposite of the portfolio holotype, as those can be largely static. For immersive applications, SEO requirements are non-existent, and loading times do not matter noticeably since you will likely need to load a considerable chunk of the application code for usage of the application to make sense. The main exception to this rule is the landing page and documentation of an immersive application as those fall behind other holotypes and, for those islands, may be used as described in the earlier sections. Another exception is navigation, as often immersive applications have general navigation that may be rendered statically and then a dynamic working area for the application itself. Islands may still be helpful for large applications if it is possible to segment them so that smaller portions may be loaded at a time. However, this may require some architecture-level thinking from developers.

10 Discussion

As seen in our discussion of holotypes, edge-powered islands represent a technique that can expand the usefulness of otherwise static approaches and use cases. In other words, they give developers more control over how their functionality is delivered to the clients. Islands may have limitations for use cases where it is not straightforward to separate static functionality from dynamic; for example, an immersive editor such as Figma is an excellent example of a use case where it would be challenging to leverage islands.

10.1 Comparison of Evaluated Holotypes

Based on our observations, edge-powered islands form a technique that can add interactivity to otherwise static websites or avoid SSR in cases where parts of the content delivered to the client are static. Islands may complement other techniques, such as ESI, to give developers more control over what to load and when. Precisely, edge-powered islands can encapsulate dynamic portions of a page that change often while providing loading benefits. Islands can be applied with specific strategies to defer or avoid loading them until their functionality is truly needed. Table 9 captures our main observations in greater detail to give you a complete picture of the technique's usefulness.

10.2 Benefits of Edge-powered Islands in Practice

As mentioned, edge-powered islands give developers a clear means for wrapping interactive functionality and controlling loading. Lazy loading, i.e., triggering the loading of a script based on user interaction, can provide an intermediate solution that is easy to implement, as islands require some level of server-side programming. Besides giving a new control layer for developers, islands are technology agnostic, unlike earlier widget attempts, and therefore allow experimentation with related ideas, like micro-frontends [11].

Our portfolio holotype benchmark results discussed in Section 5 show that islands can provide clear benefits even in a simple use case. Lazy loading alone gave a sizable benefit to FCP in our use case. Although islands and lazy-loading variants had roughly equal FCP scores, their behavior would be different when the user interacts with the application as there would be less work to do in the island variant due to the light server-side implementation compared to the third-party one provided by Disqus and used in our example.

For content sites discussed in Section 6, islands allowed us to send fewer initial data to the client and defer the loading of secondary content. Combined

Table 9 The table shows how edge-powered islands fit each of the six holotypes covered in this article while considering our observations. The most static websites benefit from the technique the most, while sites with clearly identifiable dynamic portions also benefit

Holotype	Fit	Observations
Portfolio sites	Good fit for interactive portions, such as comments.	In the benchmark at Section 5, it was shown that wrapping interactive functionality, such as commenting, can benefit INP, thereby improving site performance. In addition, only leveraging a client-side technique, such as loading a script based on user interaction, can be beneficial.
Content sites	Edge-driven islands can capture frequently updating content and therefore the fit is limited.	The content site case shares the benefits with the portfolio sites holotype. We discussed through a news page how the technique could be applied to defer work and found islands can complement ESI well. The benchmark in Section 6 showed that using islands can help to reduce the amount of data loaded initially.
Storefronts	Given the easily encapsulated requirements of storefronts, edge-powered islands are an excellent fit for their dynamic portions, such as search.	In the benchmark at Section 7, it was shown that leveraging islands can have potential benefits for search views as they can help to avoid the cost of refreshing a page.
Social networks	Potential fit at the dynamic portions of the application.	For social media applications, islands allow splitting up the application into specific portions to load separately to speed up initial loading of the application as shown in Section 8. There may be a small size penalty related to the island bootstrapping code that has to be provided to the client for the approach to work.
Immersive applications	Limited fit due depending on application structure and whether it can be segmented into islands.	It is possible that islands could be used for applications with multiple separate views that can be easily divided as islands.

with ESI, islands give a good amount of control for developers to decide what to load and when.

The storefront case discussed in Section 7, shows that islands allow avoiding work related to showing search results over a naïve SSR-based approach. Although the test case was partly synthetic, the difference can

be more significant than was demonstrated in practice, given that individual pages can be quite heavy due to additional logic and third-party dependencies.

Islands architecture has potential benefits even for more dynamic use cases, such as social networks discussed in Section 8 and immersive applications discussed in Section 9. The main point to highlight is that if it is possible to split up an application into smaller portions, islands can be used to speed up the initial loading of the application. It is possible similar benefits could be gained while navigating within an application depending on its architecture. Using islands might become problematic if the application is monolithic by nature and cannot be split into smaller parts easily.

The critical thing to keep in mind is that islands work as primitives that give further control over caching and loading. Islands seem ideal for short-lived, potentially customized portions of a page where some amount of server processing may be required. From the server's point of view, the endpoints exposed to islands can be kept technically simple and may leverage standard caching techniques themselves.

10.3 Limitations of Edge-powered Islands

Edge-powered islands seem helpful for cases where functionality can be wrapped into static and dynamic, and, therefore, promote static rendering techniques with their caching benefits while pushing SSR to the edges of a website. That said, the more dynamic and complex the use case becomes, the fewer chances to benefit from this split. Although you could likely implement large applications using islands, the need to load the code soon for the application to be useful would likely decrease the benefits somewhat.

An important caveat of the island approach is that it relies on JavaScript to trigger the loading of the islands. The reliance on JavaScript can be problematic when the user has disabled JavaScript on their browser as the functionality is unreachable. That said, the impact of this issue is mitigated by the fact that the ideal use case of content-driven websites' dynamic functionality is secondary. For a storefront, it may make sense to define fallbacks using a standard `noscript` element to allow the usage of essential functionality, such as a shopping cart.

Inter-island communication is another aspect of interactive applications, such as a storefront, as separate islands may share the same state or use a derived part of it. To solve this problem, a global bus or a similar solution may be considered, adding complexity to the approach. A related issue is so-called nested islands initially solved by Fresh framework [7].

10.4 Approaches That Complement Edge-powered Islands

As seen in Section 5, occasionally, a technically more straightforward option may be to use client-side lazy loading to gain loading benefits. Generally, doing less work and deferring it is a good option for all the holotypes discussed. The main question is what kind of payloads are needed and when. A related question is how the payloads are delivered, and that is where streaming [52] may be a good option. Another way to improve application performance might be to use enhanced tooling [53] or new techniques, such as resumability [50].

As noted in Section 6, older techniques such as ESI may be helpful with islands as they allow composition at the edge to gain caching benefits since content often has different lifecycles. Interestingly, CSI by Rabinovich et al. [38] from 2003 is close to the idea of islands by allowing ESI on the client side. However, CSI has different development ergonomics than modern island implementations, even if the concept is similar at a high level.

Islands architecture is simple to implement, providing one way to develop a primitive boundary between static and dynamic. Therefore, adopting islands architecture in a project may be the first step towards consciously defining what is static and what is not. Perhaps the question should be, how can adopting islands architecture improve the state of your application? As shown in this article, the architecture may have its uses depending on how dynamic your use case is and how easy it is to separate static from dynamic.

10.5 Edge Computing Can Replace Traditional Servers

We could use edge infrastructure as a complete replacement for traditional server infrastructure to allow for easy provisioning of our code. Although that was not the study's target, we could show one benefit to the edge – relative ease of deployment. That said, there are likely benefits to using islands architecture, even in a more traditional context, as it gives developers a way to control how the dynamic portions of their applications are loaded since the technical implementation is so simple, given only a minimal amount of JavaScript is required. Meta-frameworks, such as Astro, make the task even more accessible as then the idea of islands is baked in.

10.6 Threats to Validity

In our study, we analyzed edge-driven islands through holotypes to capture their benefits in different scenarios, as websites can differ wildly in their

requirements for dynamic functionality. There were likely gaps in our selection of holotypes to study, and it is essential to consider each application separately. It is even possible that, within a single application, there are certain parts where using islands makes more sense than others; therefore, the approach can be used opportunistically to gain benefits. Lazy-loading seems like an easy, intermediate solution that avoids server-side effort.

In our benchmarks, we used only the Cloudflare platform. In technical terms, Cloudflare was a black box to us as the implementation is primarily hidden apart from technical descriptions available online [10]. Since we tested only from a single location, the results only indicate possible performance and do not capture global experience. For this comparison, it was an acceptable compromise as the target was to evaluate techniques relative to each other.

There is some uncertainty in the technical implementations as it is possible not all best practices were always followed, although the technical simplicity of the implementations reduces the risk. We considered using a framework like Astro for the implementation, but that would have represented yet another black box as we don't understand the technical details of Astro in great detail. Therefore, keeping our implementation simple was an acceptable way to go. Implementing an additional variant using Astro might be a worthwhile endeavor to capture the cost/benefit of a framework on some level in a future study.

The test cases were partially synthetic, so they might not have captured all real-world use case details. Simultaneously, it is difficult to arrive at a single case that would capture the complexity of real-world use cases, as web applications tend to vary by their requirements. In this benchmark, we decided to focus on the core issue, the performance of islands architecture when combined with edge computing, and minimize external factors contributing noise to the results.

11 Conclusion

In this article, we sought answers to the following question: "What are the benefits of combining the islands architecture with serverless edge computing?". We found early proof supporting the usefulness of combining the approaches to address the question. The main observed benefit is

performance, as in our test case islands architecture allowed deferring work to the future and even avoiding it.[19]

It is important to note that the approach's applicability depends on the web application and whether there are opportunities to perform at least some of the work later. For use cases where the application is mostly or partially static in terms of content, there are likely good ways to leverage islands to capture and control how and when the interactive portions are loaded. The more dynamic and complex the use case becomes, the more difficult it is to gain the benefits as using islands ties into application design and how an application can be separated into smaller portions to treat as islands.

11.1 Future Work

In this article, we provided early evidence of the usefulness of the islands architecture for web development. There are still open questions related to the approach, and we have summarized several of those below:

1. Is there a practical limit to the number of islands an individual page can have?
2. What are good ways to coordinate data-related concerns between islands? In other words, what should we do when different islands have to share some of the same data?
3. When does it make sense to use some other approach instead of islands? In other words, what are the sweet spots for using the islands architecture?
4. What benefits do islands-oriented web frameworks provide over implementing the island pattern yourself?
5. Are the performance benefits islands visible when using a more traditional server or cloud-based environment?
6. Which technical approaches complement the islands architecture and how?
7. If edge infrastructure is left out of the picture and a traditional server/client architecture is used, what do the discovered benefits look like then?

[19]The benchmark source is available at https://github.com/bebraw/islands-benchmark to encourage experimentation and research.

There is growing interest in islands architecture, and with the introduction of islands-oriented frameworks like Astro or Fresh, it seems like an approach with some future. By definition, it reminds us of earlier approaches, for example, portals/portlets [55] and web widgets [26]. Still, such is the nature of web development, where good ideas are repeatedly discovered in different forms, and islands architecture is an example of this process of rediscovery.

References

[1] Shalinda Adikari and Kaushik Dutta. Real time bidding in online digital advertisement. In *New Horizons in Design Science: Broadening the Research Agenda: 10th International Conference, DESRIST 2015, Dublin, Ireland, May 20-22, 2015, Proceedings 10*, pages 19–38. Springer, 2015.

[2] Marcus Basalla, Johannes Schneider, Martin Luksik, Roope Jaakonmäki, and Jan Vom Brocke. On latency of e-commerce platforms. *Journal of Organizational Computing and Electronic Commerce*, 31(1):1–17, 2021.

[3] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann. World-wide web: the information universe. *Internet Research*, 1992.

[4] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[5] Ryan Carniato. Understanding Transitional JavaScript Apps. https://dev.to/this-is-learning/understanding-transitional-javascript-apps-27i2, 2021. [Accessed 17-05-2024].

[6] Ryan Carniato. Patterns for Building JavaScript Websites in 2022. https://dev.to/this-is-learning/patterns-for-building-javascript-websites-in-2022-5a93, 2022. [Accessed 13-05-2024].

[7] Luca Casonato. Fresh 1.2 – welcoming a full-time maintainer, sharing state between islands, limited npm support, and more. https://deno.com/blog/fresh-1.2, 2023. [Accessed 30-08-2023].

[8] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. Jscleaner: De-cluttering mobile webpages through javascript cleanup. In *Proceedings of The Web Conference 2020*, pages 763–773, 2020.

[9] Batyr Charyyev, Engin Arslan, and Mehmet Hadi Gunes. Latency comparison of cloud datacenters and edge servers. In *GLOBECOM 2020-2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[10] Architectures · Cloudflare Reference Architecture docs. https://developers.cloudflare.com/reference-architecture/, 2024. [Accessed 17-05-2024].

[11] Peter Bacon Darwin, James Culveyhouse, and Igor Minar. Cloudflare Workers and micro-frontends: made for one another. https://blog.cloudflare.com/better-micro-frontends/, 2022. [Accessed 30-08-2023].

[12] Davide Di Fatta, Dean Patton, and Giampaolo Viglia. The determinants of conversion rates in sme e-commerce websites. *Journal of Retailing and Consumer Services*, 41:161–168, 2018.

[13] Matthew Phillips et al. Astro 4.0. https://astro.build/blog/astro-4/, 2023. [Accessed 11-01-2024].

[14] Marvin Hagemeister. Fresh 1.5: Partials, client side navigation and more. https://deno.com/blog/fresh-1.5, 2023. [Accessed 11-01-2024].

[15] Lydia Hallie and Addy Osmani. Islands Architecture. https://www.patterns.dev/posts/islands-architecture/, 2022. [Accessed 29-Sep-2022].

[16] Tjaša Heričko, Boštjan Šumak, and Saša Brdnik. Towards representative web performance measurements with google lighthouse. In *Proceedings of the 2021 7th Student Computer Science Research Conference*, page 39, 2021.

[17] Page Weight — 2022 — The Web Almanac by HTTP Archive. https://almanac.httparchive.org/en/2022/page-weight, 2022. [Accessed 28-08-2023].

[18] Robert Istrate, Victor Tulus, Robert N Grass, Laurent Vanbever, Wendelin J Stark, and Gonzalo Guillén-Gosálbez. The environmental sustainability of digital content consumption. *Nature Communications*, 15(1):3724, 2024.

[19] Allie Jones. Mobius: Adopting JSX While Prioritizing User Experience. https://www.etsy.com/codeascraft/mobius-adopting-jsx-while-prioritizing-user-experience, 2021. [Accessed 30-08-2023].

[20] Mikael Koskinen, Tommi Mikkonen, and Pekka Abrahamsson. Containers in software development: A systematic mapping study. In *International conference on product-focused software process improvement*, pages 176–191. Springer, 2019.

[21] Samuel Kounev, Nikolas Herbst, Cristina L Abad, Alexandru Iosup, Ian Foster, Prashant Shenoy, Omer Rana, and Andrew A Chien. Serverless computing: What it is, and what it is not? *Communications of the ACM*, 66(9):80–92, 2023.

[22] Michael Krug and Martin Gaedke. Smartcomposition: enhanced web components for a better future of web development. In *Proceedings of*

the 24th International Conference on World Wide Web, pages 207–210, 2015.

[23] Tofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, Waleed Hashmi, Lakshmi Subramanian, and Yasir Zaki. Muzeel: A dynamic javascript analyzer for dead code elimination in today's web. *arXiv preprint arXiv:2106.08948*, 2021.

[24] lighthouse/variability.md at main. https://github.com/GoogleChrome/lighthouse/blob/main/docs/variability.md, 2022. [Accessed 11-Oct-2022].

[25] lighthouse/docs/user-flows.md at main. https://github.com/GoogleChrome/lighthouse/blob/main/docs/user-flows.md, 2024. [Accessed 25-06-2024].

[26] Eetu Mäkelä, Kim Viljanen, Olli Alm, Jouni Tuominen, Onni Valkeapää, Tomi Kauppinen, Jussi Kurki, Reetta Sinkkilä, Teppo Kansala, Robin Lindroos, et al. Enabling the semantic web with ready-to-use web widgets. *FIRST*, 293:56–69, 2007.

[27] Paco Azevedo Mendes. *Evaluation of widget-based approaches for developing rich internet applications*. PhD thesis, University of the Witwatersrand, 2010.

[28] Jason Miller. Application Holotypes: A Guide to Architecture Decisions. https://jasonformat.com/application-holotypes/, 2019. [Accessed 13-05-2024].

[29] Jason Miller. Islands Architecture. https://jasonformat.com/islands-architecture/, 2020. [Accessed 17-05-2024].

[30] Juliana Mitchell-Wong, Ryszard Kowalczyk, Albena Roshelova, Bruce Joy, and Henry Tsai. Opensocial: From social networks to social ecosystem. In *2007 Inaugural IEEE-IES Digital EcoSystems and Technologies Conference*, pages 361–366. IEEE, 2007.

[31] Stefan Nastic, Philipp Raith, Alireza Furutanpey, Thomas Pusztai, and Schahram Dustdar. A serverless computing fabric for edge & cloud. In *2022 IEEE 4th International Conference on Cognitive Machine Intelligence (CogMI)*, pages 1–12. IEEE, 2022.

[32] Ravi Netravali and James Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 249–266, 2018.

[33] Stephane Nouvellon. Edge-Side-Includes with Cloudflare Workers — blog.cloudflare.com. https://blog.cloudflare.com/edge-side-includes-with-cloudflare-workers/, 2018. [Accessed 06-10-2024].

[34] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[35] Creating Java Portlets. https://docs.oracle.com/cd/E17904_01/portal.1111/e10238/pdg_java_intro.htm, 2024. [Accessed 27-03-2024].

[36] holotype – quick search. https://www.oed.com/search/dictionary/?scope=Entries&q=holotype, 2024. [Accessed 19-06-2024].

[37] Y Prajwal, Jainil Viren Parekh, and Rajashree Shettar. A brief review of micro-frontends. *United International Journal for Research and Technology*, 2(8), 2021.

[38] Michael Rabinovich, Zhen Xiao, Fred Douglis, and Chuck Kalmanek. Moving {Edge-Side} includes to the real {Edge—the} clients. In *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*, 2003.

[39] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. Html 4.01 specification. *IETF HTML WG*, 1997.

[40] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27(3):50–64, 2023.

[41] Hussain Saleem, M Khawaja Shaiq Uddin, Syed Habib-ur Rehman, Samina Saleem, and Ali Muhammad Aslam. Strategic data driven approach to improve conversion rates and sales performance of e-commerce websites. *International Journal of Scientific & Engineering Research*, 10(4):588–593, 2019.

[42] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[43] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[44] V Solovei, Olga Olshevska, and Y Bortsova. The difference between developing single page application and traditional web application based on mechatronics robot laboratory onaft application. *Automation of technological and business processes*, 10(1), 2018.

[45] Internet and social media users in the world 2023. https://www.statista.com/statistics/617136/digital-population-worldwide/, 2023. [Accessed 28-08-2023].

[46] Davide Taibi and Luca Mezzalira. Micro-frontends: Principles, implementations, and pitfalls. *ACM SIGSOFT Software Engineering Notes*, 47(4):25–29, 2022.

[47] Ke Tian, Zhou Li, Kevin D Bowers, and Danfeng Yao. Framehanger: Evaluating and classifying iframe injection at large scale. In *Security and Privacy in Communication Networks: 14th International Conference, SecureComm 2018, Singapore, Singapore, August 8-10, 2018, Proceedings, Part II*, pages 311–331. Springer, 2018.

[48] Sipat Triukose, Zhihua Wen, and Michael Rabinovich. Measuring a commercial content delivery network. In *Proceedings of the 20th international conference on World wide web*, pages 467–476, 2011.

[49] Juho Vepsäläinen, Arto Hellas, and Petri Vuorimaa. The rise of disappearing frameworks in web development. In *International Conference on Web Engineering*, pages 319–326. Springer, 2023.

[50] Juho Vepsäläinen, Miško Hevery, and Petri Vuorimaa. Resumability-a new primitive for developing web applications. *IEEE Access*, 2024.

[51] Juho Vepsäläinen, Arto Hellas, and Petri Vuorimaa. The state of disappearing frameworks in 2023. In *WEBIST 2023: 19th International Conference on Web Information Systems and Technologies*. Springer, 2023.

[52] Lucas Vogel and Thomas Springer. How streaming can improve the world (wide web). In *Companion Proceedings of the ACM Web Conference 2023*, pages 140–143, 2023.

[53] Lucas Vogel and Thomas Springer. Waiter and autratac: Don't throw it away, just delay! In Irene Garrigós, Juan Manuel Murillo Rodríguez, and Manuel Wimmer, editors, *Web Engineering*, pages 278–292, Cham, 2023. Springer Nature Switzerland.

[54] Philip Walton. Web Vitals. https://web.dev/articles/vitals, 2024. [Accessed 12-02-2024].

[55] Christian Wege. Portal server technology. *IEEE Internet Computing*, 6(3):73–77, 2002.

[56] Scott Wilson, Florian Daniel, Uwe Jugel, and Stefano Soi. Orchestrated user interface mashups using w3c widgets. In *Current Trends in Web Engineering: Workshops, Doctoral Symposium, and Tutorials, Held at ICWE 2011, Paphos, Cyprus, June 20-21, 2011. Revised Selected Papers 11*, pages 49–61. Springer, 2012.

[57] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, pages 37–42, 2015.

Biographies

Juho Vepsäläinen is a Doctoral Researcher at Aalto University, Finland. His current research interests include web development, web performance, and green computing.

Petri Vuorimaa is a Professor at Aalto University, Finland. His current research and teaching interests include web applications, web technologies, and science.

Arto Hellas is a Senior University Lecturer at Aalto University, Finland. His current research interests include understanding and improving teaching and learning in digital and hybrid learning environments.