Lifting off with Astro

Brian Rinaldi

Who am I?

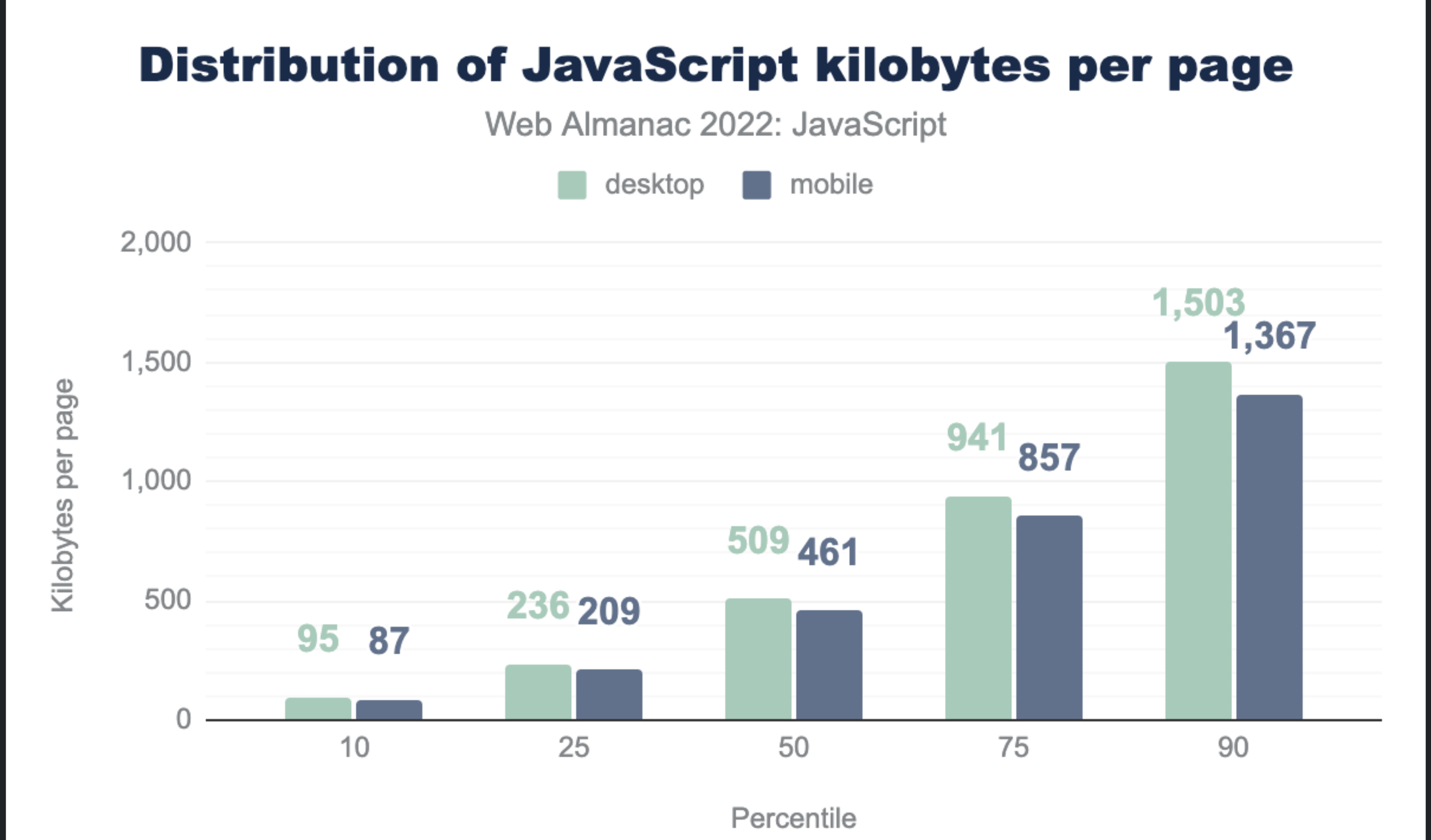
— Developer Experience Engineer at LaunchDarkly
— Co-author of The Jamstack Book from Manning
— Editor of the Jamstacked newsletter from Cooper Press
— Run the CFE.dev community and Orlando Devs


LaunchDarkly

Let's first talk about JavaScript. Addy osmani has written about and talked about the cost of JavaScript every year. JavaScript remains the most expensive resource on your site and has been the same since he first talked about it in 2018. It's not just about the weight of the resources, which is growing, but also the time it takes to process it

From 2021 to 2022, an increase of 8% [in the amount of JavaScript shipped to browsers] for mobile devices was observed, whereas desktop devices saw an increase of 10%. ...The fact remains that more JavaScript equates to more strain on a device's resources.
Web Almanac

And that weight keeps growing every year. According to the web almanac's latest report, it grew another 8% for mobile in the past year. While that is a smaller increase than in some prior years, this ever expanding JavaScript can have a really detrimental impact on lower end devices that much of the world uses.

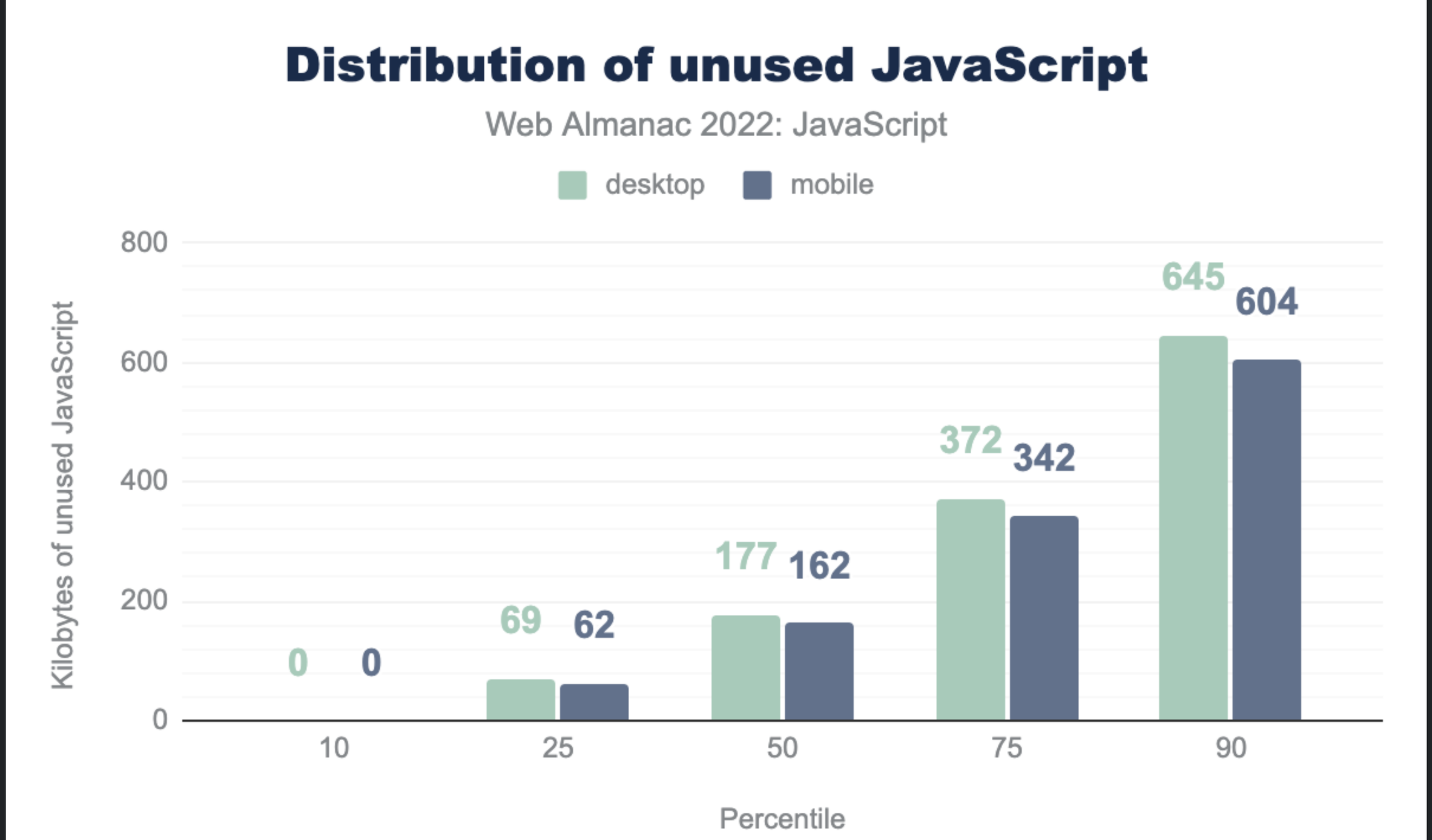


This is the distrbution of JavaScript size in the sites they surveyed for their report and it can be rather shocking to imagine some sites are loading about a megabyte and a half of just JavaScript. One suspects that these same sites will have lots of this JavaScript blocking the main thread, meaning the site is unusable while the user waits.

When contrasted with the total number of bytes loaded for mobile pages at the median, unused JavaScript accounts for 35% of all loaded scripts.

Web Almanac

And here's the sadder part. Much of that JavaScript is entirely unused so this extra weight serves absolutely no purpose.



As you can see, some sites are sending as much as 600k in unused JavaScript. Why is there so much unused JavaScript? Well, in many cases, this is probably related to the ever growing list of third party scripts...

Unnecessary JavaScript

...but beyond unused JavaScript there's a category I am making up that I'm going to call unnecessary JavaScript.

Unnecessary JavaScript is what I refer to as JavaScript being used where JavaScript is not needed. It's not unused, but it is unneeded and it's related to the fact that at some point we also started using tools like React, that were meant for application development, to build everything including sites that were heavy on content with limited interactivity or web app type elements

Not only are new services being built to a self-defeatingly low UX and performance standard, existing experiences are pervasively re-developed on unspeakably slow, JS-taxed stacks.

Alex Russell, The Market for Lemons

You may have read Alex Russell's post The Market for Lemons, which dives deep into the history of how developers were convinced to become more reliant on stacks that delivered increasingly large JavaScript bundles. While you may disagree with some of his rhetoric, much of what he cites as evidence can be backed up with data.

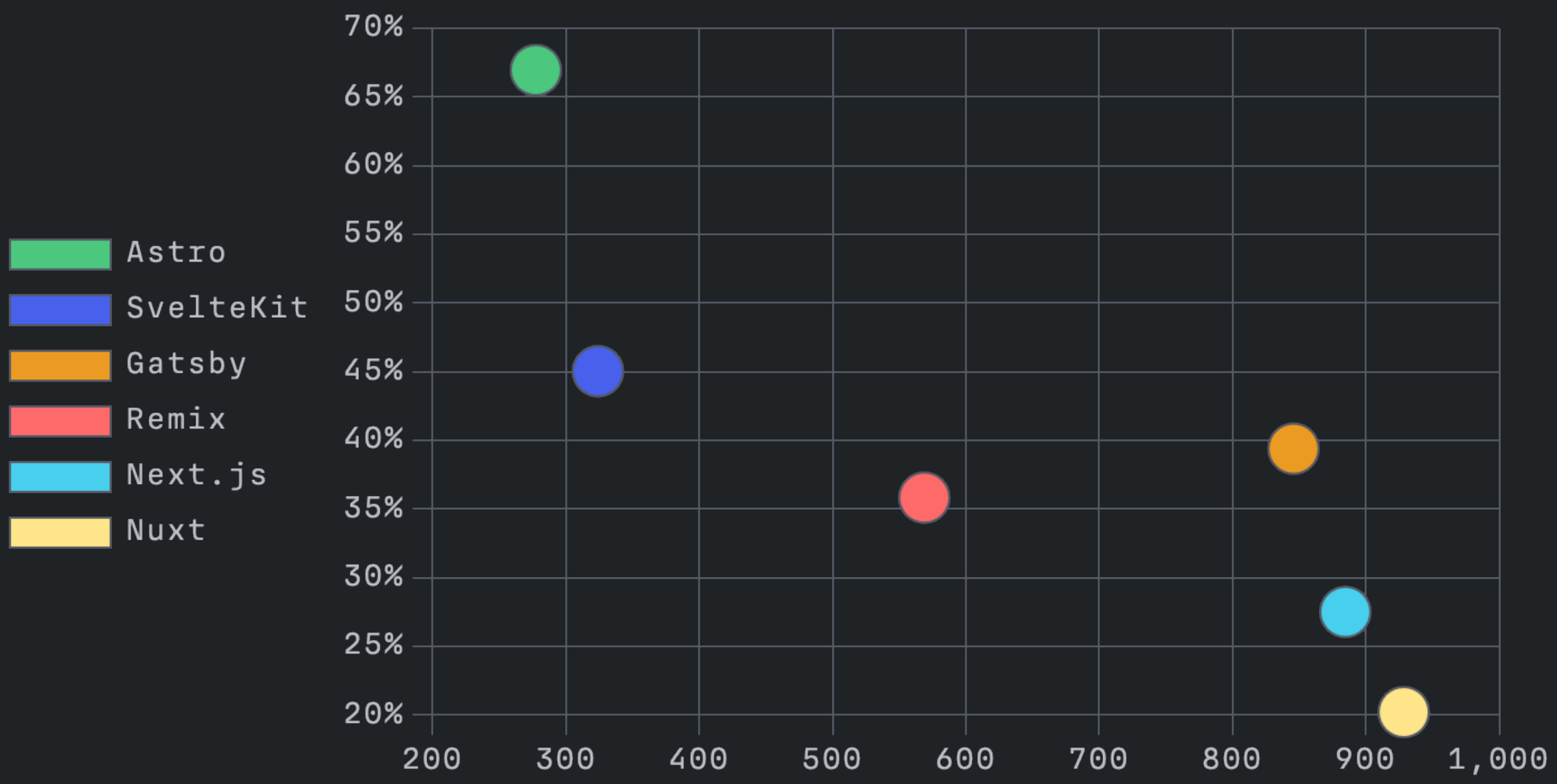
New front-end frameworks like Solid and Qwik are suggesting that React might not have all the answers after all, and on the server Astro, Remix and Next.js (among others) are making us reconsider how much code we really need to ship to the client.
State of JavaScript 2022

Thankfully this has caused a lot of rethinking and we're seeing a ton of innovation around this issue, not just from Astro but from frameworks like Qwik and SolidJS. Even Next.js has been pushing React server pages as a way to remedy this problem.

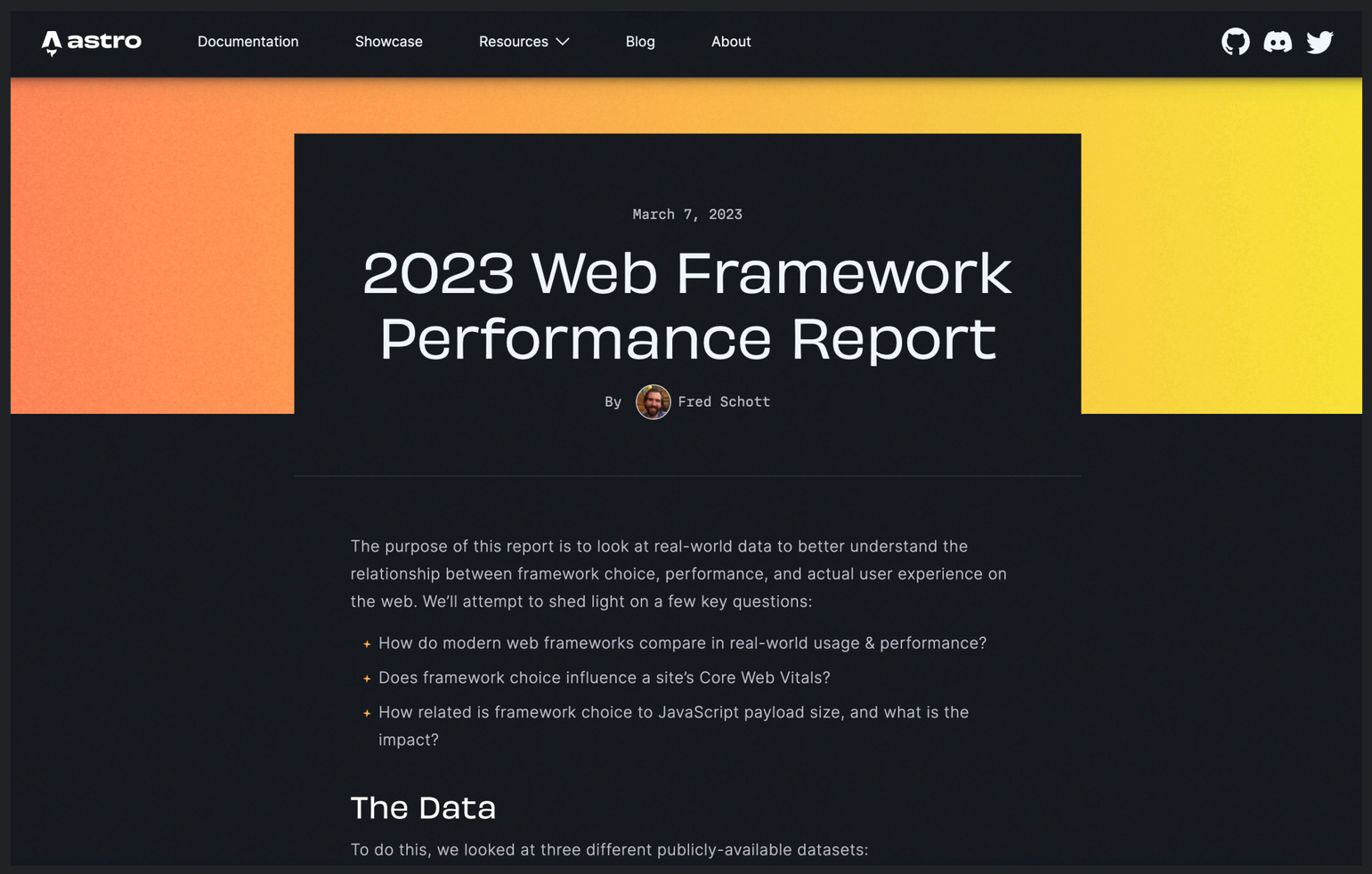
But why are developers hesitant to move? I believe it is because it can be difficult to abandon an existing codebase and the tools you've become accustomed to using. Faced with steep learning curves and/or rewriting their entire codebase, it can be difficult to move, even if you are aware of the problem.

https://astro.build

This problem is a big impetus behind Astro and its architecture. Let's see how.



This is from a report created by the Astro team which shows that Astro sites have the lowest median KB and the highest percentage of sites passing core web vitals. Even lower than tools like SvelteKit and Remix which also aim to address some of these same issues.



https://astro.build/blog/2023-web-framework-performance-report/

The full report goes into more detail including specifics around core web vitals scores and more. However, it's worth keeping in mind that the sample for Astro is very small in comparison to tools like Next.js, and, as we'll see, the types of sites they are building are different by design, so it's worth taking some of these numbers with a grain of salt.

0kb

By default, Astro delivers 0kb of JavaScript to the client. Of course, very few sites today can function with zero JavaScript, but the core architectural philosophy of Astro is designed to allow JavaScript for interactivity but still keep the bundle size to an absolute minimum.

What the heck are islands?

The primary means that Astro uses to achieve this is via a concept called the islands architecture.

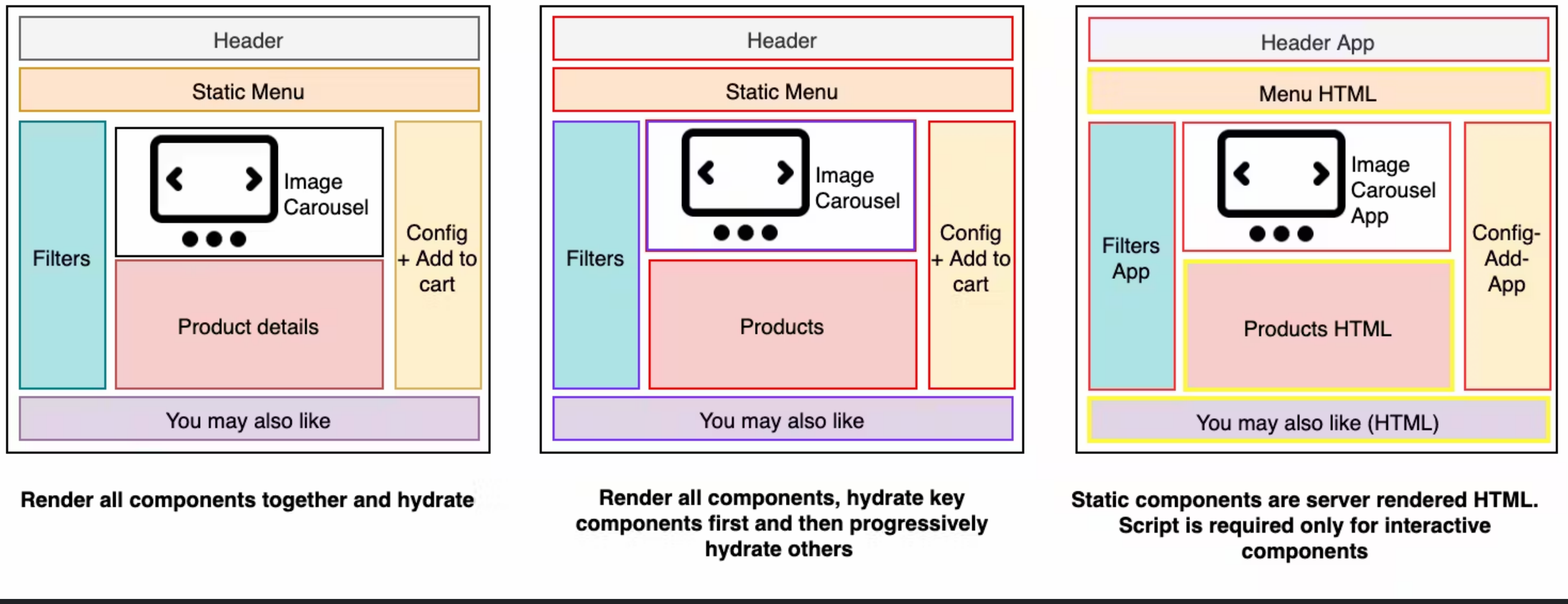
"Islands" === Islands Architecture || Component Islands

The "Component Islands" pattern was coined by Etsy's frontend architect Katie Sylor-Miller in 2019.

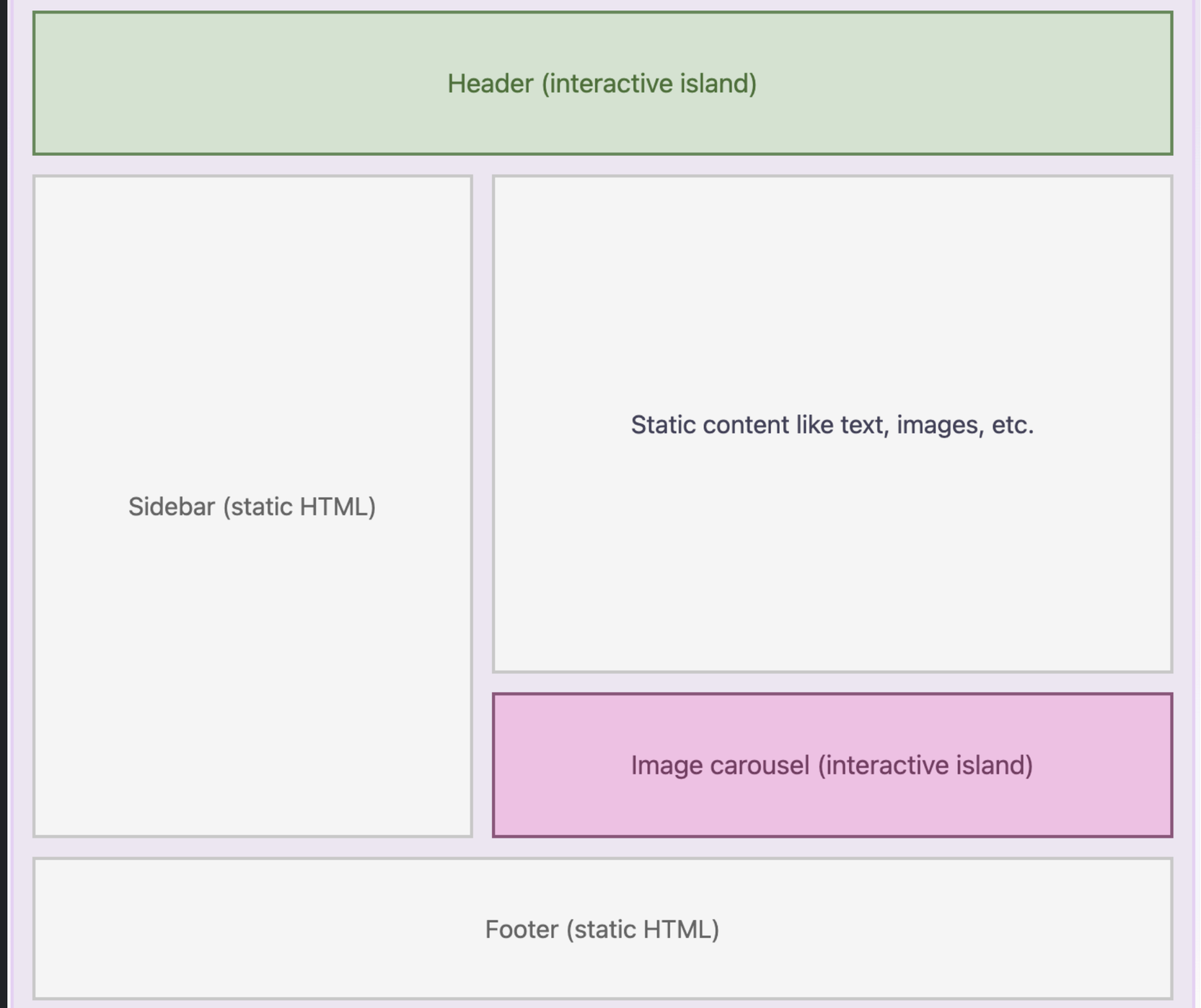
This is a concept that is relatively new, having only originally been coined in 2019, only a couple years before Astro's initial release.

The general idea of an “Islands” architecture is deceptively simple: render HTML pages on the server, and inject placeholders or slots around highly dynamic regions. These placeholders/slots contain the server-rendered HTML output from their corresponding widget. They denote regions that can then be "hydrated" on the client into small self-contained widgets, reusing their server-rendered initial HTML.
Islands Architecture by Jason Miller

As the name implies, the architecture is designed to create islands of interactivity within an otherwise straight HTML output. This means that rather than including full framework JavaScript on every page and for the whole page, it loads only the JavaScript for the dynamic regions on a page.



This diagram from patterns.dev shows how this differs from SSR and partial hydration. In SSR all the components are rendered and hydrated together. In progressive hydration, the components are rendered together but critical components are hydrated first after which other components are progressively hydrated. In both cases, script is required to hydrate all rendered components. But in the islands diagram, only the components marked as interactive will receive the necessary JavaScript and thus hydration.



A simpler example looks like this which shows a standard content page wherein a header component and an image carousel require JavaScript for interactivity and hydration, but the remainder of the content on the page is static HTML.

Hydrating an Island

```
---
// Example: Use a dynamic React component on the page.
import MyAstroComponent from '../components/MyAstroComponent.astro';
import MyReactComponent from '../components/MyReactComponent.jsx';
---
<!-- This component is now interactive on the page! -->
<MyReactComponent client:load />
<!-- This component loads zero js --->
<MyAstroComponent />
```

This is an example of how that works in Astro. The Astro component (don't worry, we'll talk about the different types of components in a bit) requires no JavaScript to render but the React component does and thus has the directive for client load.

Hydration Directives

— **load** – high priority. Elements that will be immediately visible on the page.
— **idle** – medium priority. Elements that do not need to be immediately interactive.
— **visible** – low priority. Typically below the fold elements that become active when in the viewport.
— **media** – low priority. Elements that might only be visible on certain screen sizes.
— **only** – Skip server rendering entirely and only render on the client.

Astro actually offers a ton of flexibility with its hydration directives that allow you to set the appropriate priority for each component all the way to forcing full client-side rendering where necessary.

Create components using other frameworks

```
---
// Example: Mixing multiple framework components on the same page.
import MyReactComponent from '../components/MyReactComponent.jsx';
import MySvelteComponent from '../components/MySvelteComponent.svelte';
import MyVueComponent from '../components/MyVueComponent.vue';
---
<div>
  <MySvelteComponent />
  <MyReactComponent />
  <MyVueComponent />
</div>
```

As you may have noticed from the earlier example, Astro makes it easier to migrate your existing codebase and tools because it can render components from most of the major frameworks including React, Svelte and Vue. You can, if you choose, even mix and match components from different frameworks within the same app or even the same page, though I'm not sure I'd recommend that as a common practice.

Astro components

```
---
import SomeAstroComponent from '../components/SomeAstroComponent.astro';
import someData from '../data/pokemon.json';

// Access passed-in component props, like `<X title="Hello, World" />`
const {title} = Astro.props;
// Fetch external data. top level await.
const data = await fetch('SOME_SECRET_API_URL/users').then(r => r.json());
---
<!-- Your HTML template -->
```

Astro also has its own component syntax that is for HTML only components on the client, though you can import libraries and run JavaScript on the server. As we'll see, you can still even build full sites and even simple web applications using just Astro components. You can still include script tags for basic JavaScript as well.

Astro was designed for building content-rich websites. This includes most marketing sites, publishing sites, documentation sites, blogs, portfolios, and some ecommerce sites.

By contrast, most modern web frameworks are designed for building web applications.

Why Astro?

That being said, Astro is specifically designed for content-focused sites and makes that clear within their marketing and documentation. This doesn't mean you can't build applications, but it isn't designed for apps with a heavy degree of complexity and interactivity. For complex web applicatons, they still recommend a full SPA framework like Next.js.

File-based Routing

```
/pages/about.md // /about
/pages/blog/[post].astro // /blog/post1 /blog/post2
/pages/api/[version]/posts.json.js
/pages/[...path].astro
```

Routing in Astro works how you probably expect. It's a fairly standard file based routing with dynamic and catchall routes for both static and server-rendering including API routes.

MPA vs SPA

Astro is a multi-page application (MPA) framework.

But a key difference is that each route is a seperate page and does not load the new route within the framework skeleton. This means that it does it's rendering and state management on the server rather than on the client. It supports two types of server rendering.

SSG & SSR

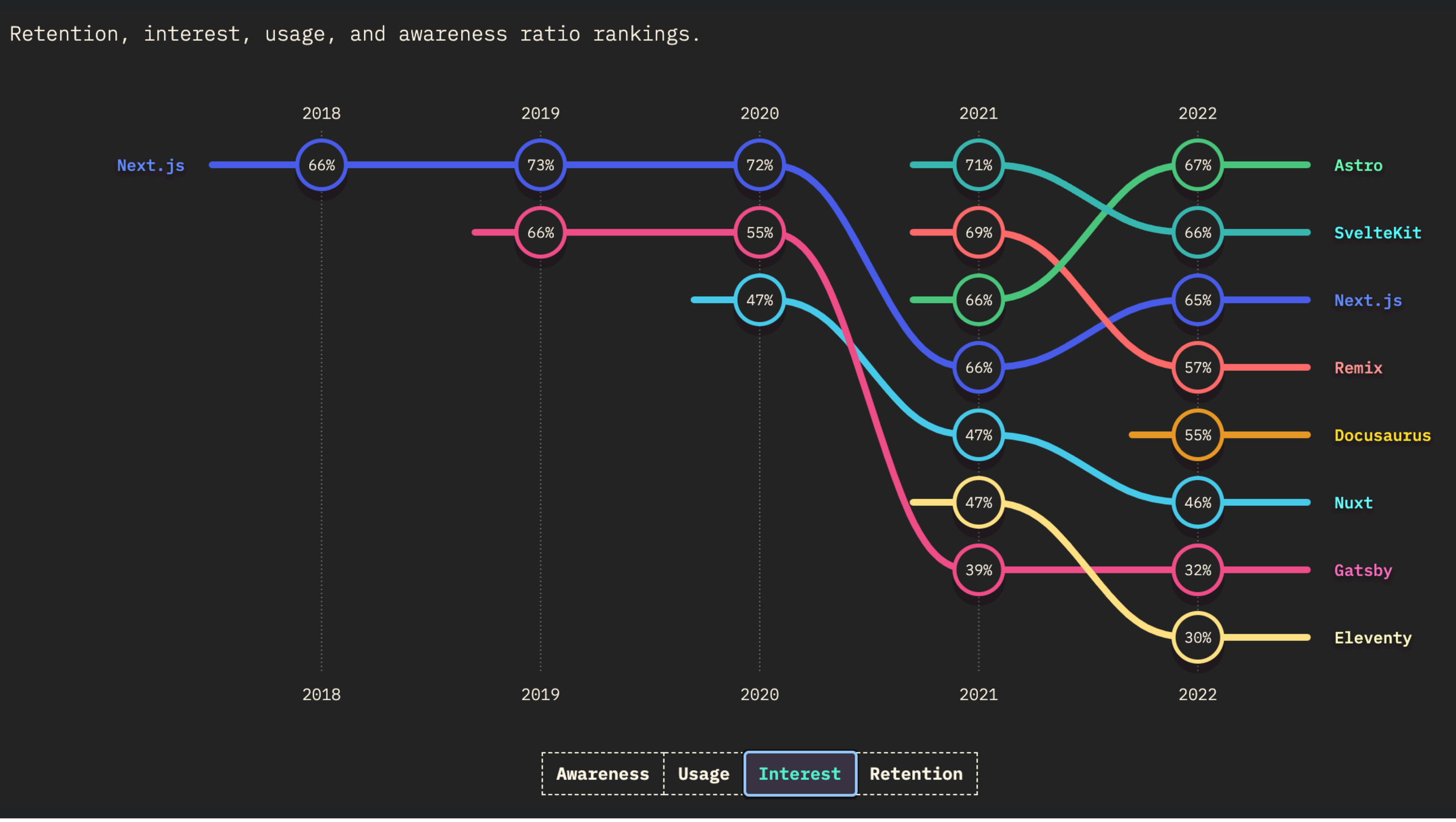
Astro started life as a static-output-only framework. Today is still defaults to static but he guidance tends to favor server-rendering.

You can do static rendering, which is basically server side rendering at build time, or server side rendering at runtime. For SSR there are adapters for many deployment platforms like Netlify, Vercel and more.

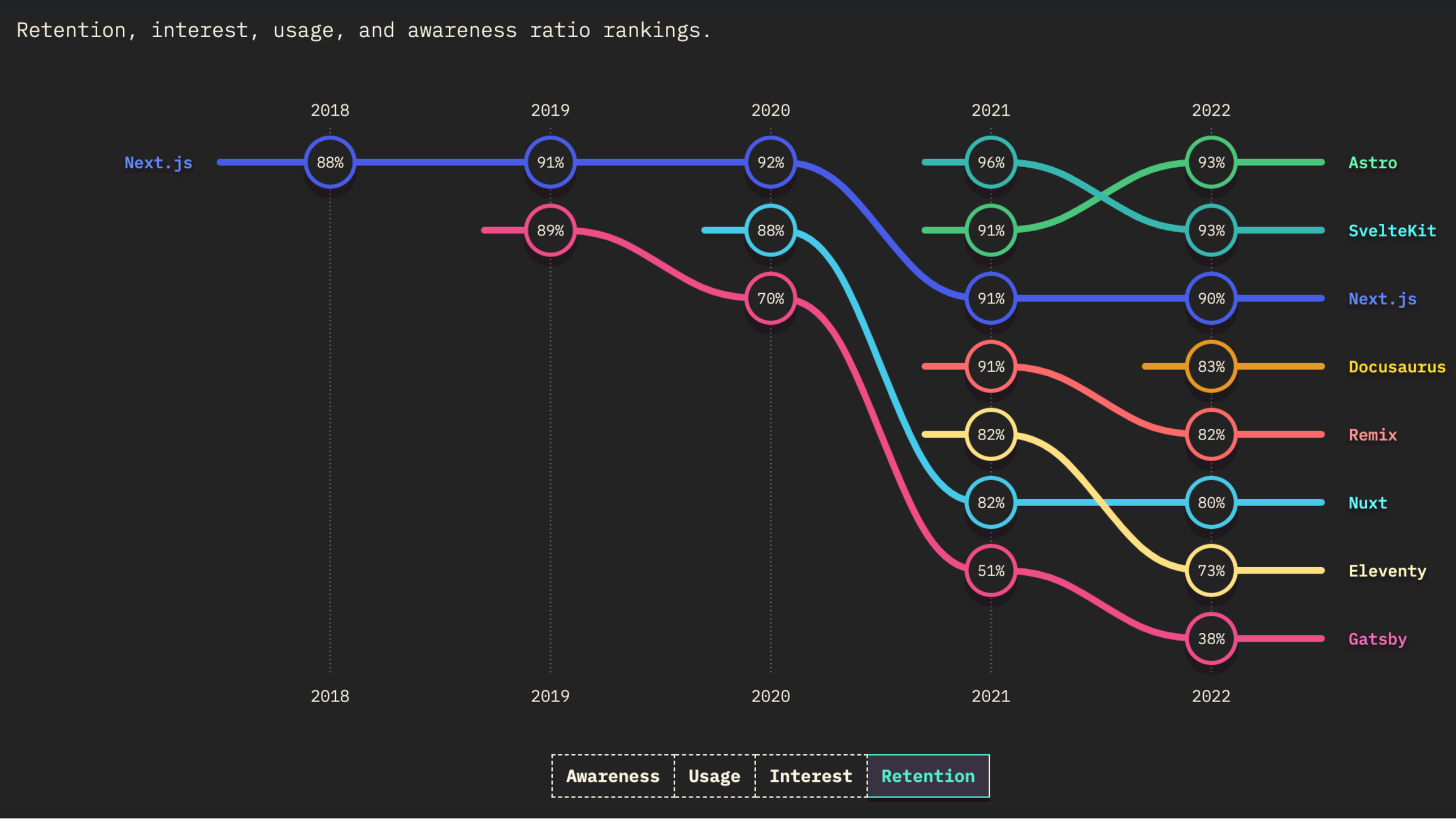
Who's using Astro?

— The Guardian
— Firebase
— NordVPN
— Stackbase
— Trivago

We've starred to see some known companies begin to adopt Astro within their stack but it's worth noting that in many cases these are sub-sites that are content focused or blogs and not for the full company site.



But the state of JS survey shows that developer interest in Astro is extremely high. Higher, in fact, than any other rendering framework.



and the developers who've used Astro like it and plan to keep using it. All of which points to continued rapid growth in adoption.

Demo

now let's dig into a simple demo I created to give you a better sense of how Astro works.

Using LaunchDarkly in Astro (server-side)

Wrapping the SDK in a JavaScript library:

```
import LaunchDarkly from "launchdarkly-node-server-sdk";
const client = LaunchDarkly.init(import.meta.env.LAUNCHDARKLY_SDK_KEY);

export default async function () {
  await client.waitForInitialization();
  return client;
}
```

Using LaunchDarkly in Astro (server-side)

Get the flag value and use it in your server-rendered code:

```
---
// launchdarkly libraries
import ldServer from "../components/LaunchDarkly/ld-server";
import ldUser from "../components/LaunchDarkly/ld-User";

let client = await ldServer();

// each user would be initialized with their info
let user = new ldUser(client);
const myFlag = await user.getFlagValue("featured-category");
---

<p>{myFlag}</p>
```

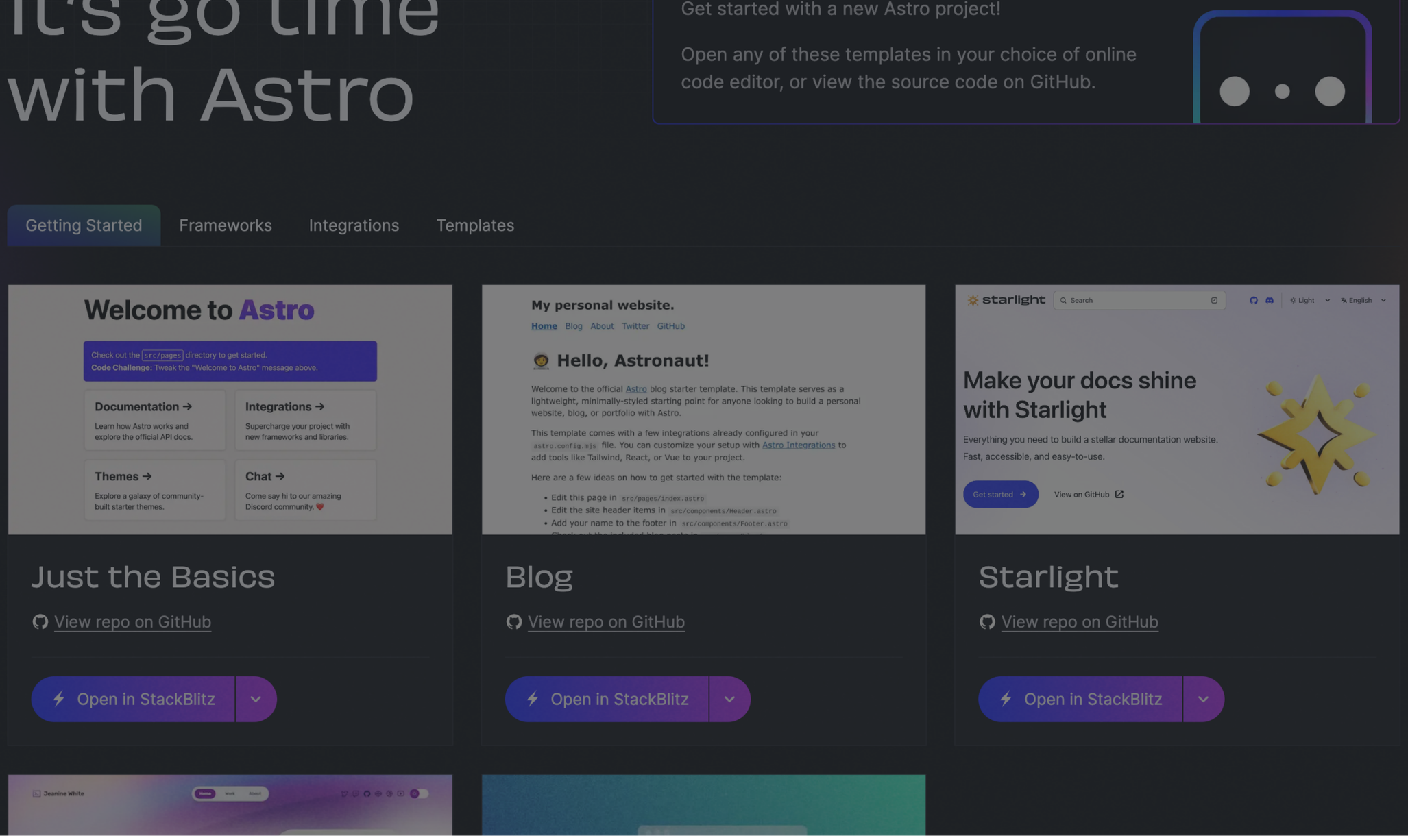
Using LaunchDarkly in Astro (client-side)

On the client, just add a script tag. Be sure to enable hydration.

```
<script>
    import { getFlagValue } from "../lib/ld-client";
    getFlagValue("show-feature", setClientFlag).then(setClientFlag);

    function setClientFlag(val) {
        alert("Flag is: " + val);
    }
</script>
```

Get started - https://astro.new



If you're ready to get started, the easiest way is via one of the many templates on astro.new

Questions?

Email: brinaldi@launchdarkly.com
Mastodon: https://mastodon.xyz/@remotesynth
BlueSky: https://staging.bsky.app/profile/remotesynthesis.com