



2022

Web Almanac

HTTP Archive's annual
state of the web report



Table of Contents

Part I. Page Content

Chapter 1: CSS	1
Chapter 2: JavaScript	71
Chapter 3: Markup	113
Chapter 4: Structured Data	131
Chapter 5: Fonts	171
Chapter 6: Media	207
Chapter 7: WebAssembly	243
Chapter 8: Third Parties	253
Chapter 9: Interoperability	275

Part II. User Experience

Chapter 10: SEO	301
Chapter 11: Accessibility	339
Chapter 12: Performance	377
Chapter 13: Privacy	423
Chapter 14: Security	449
Chapter 15: Mobile Web	491
Chapter 16: Capabilities	515
Chapter 17: PWA	531

Part III. Content Publishing

Chapter 18: CMS	557
Chapter 19: Jamstack	589
Chapter 20: Sustainability	609

Part IV. Content Distribution

Chapter 21: Page Weight	651
Chapter 22: CDN	673
Chapter 23: HTTP	695

Appendices

Methodology	709
Contributors.....	719

Part I Chapter 1

CSS



Written by Rachel Andrew

Reviewed by Chris Lilley and Jens Oliver Meiert

Analyzed and edited by Rick Viscomi

Introduction

CSS is the language used to lay out and format web pages and other media. It is one of the three main languages of the web, joining HTML, which is used for structure, and JavaScript for behavior.

The past few years have seen a flurry of new CSS features. Many of these have taken inspiration from things developers were already doing with JavaScript or in preprocessors, while others provide methods of doing things that were impossible a few years ago. Having new features available is one thing, but are developers actually using them in their production web pages and applications? It is this question we will try to answer with data.

In this chapter, we use the data to find out what developers actually use in production, rather than the features most talked about on Twitter, showcased at conferences, or found in clever demos. We can see which of the new features are being adopted, which old techniques are falling out of use, and the legacy techniques that are stubbornly remaining in our stylesheets.

Usage

Each year, we see that CSS grows in size, and 2022 was no exception.



Figure 1.1. Distribution of the stylesheet transfer size by page.

Other than the 25th percentile, which dropped a percentage point, each percentile showed a small increase in size. At the 90th percentile the increase was almost 7%, a similar increase to that seen between 2020¹ and 2021². Mobile stylesheets remain slightly smaller than those served to desktop.

The desktop page with the greatest CSS weight was slightly smaller than last year at 62,631 KB. The largest mobile stylesheet had risen from 17,823 KB to 78,543 KB—thankfully this was an exception.

1. <https://almanac.httparchive.org/en/2020/css>
 2. <https://almanac.httparchive.org/en/2021/css>

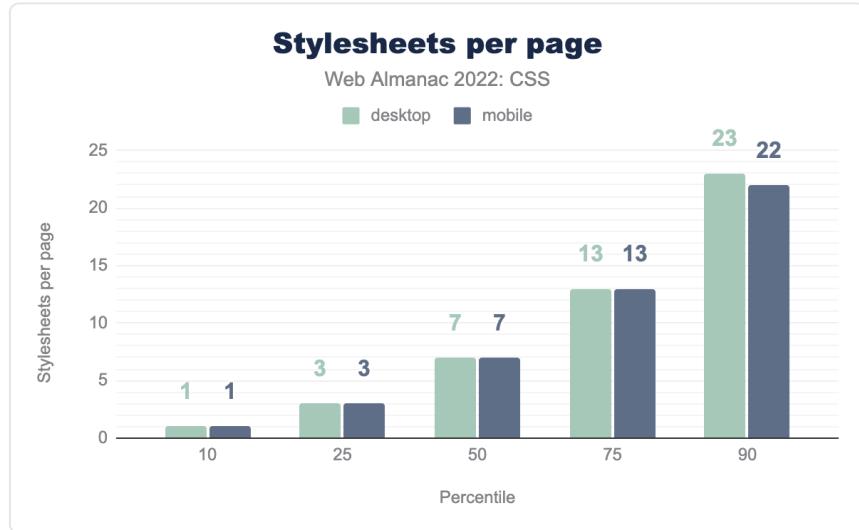


Figure 1.2. Distribution of the number of stylesheets per page.

The number of stylesheets per page has remained almost identical to 2021, with an increase of one for mobile at the 50th percentile.

Last year the record was broken for the number of stylesheets loaded by a single page at 2,368. This year we found one site loading 1,387 stylesheets on mobile, still a significant amount.



Figure 1.3. Distribution of the total number of style rules per page.

Taking a look at the number of style rules in a page showed an increase across all percentiles; the lower percentiles showing more rules for mobile, the higher percentiles more for desktop. These increases are substantial. Desktop rules for the 50th percentile increased by 130 rules, and the 90th percentile by 202.



Figure 1.4. Distribution of the number of rules per stylesheet.

We can see from the total number of stylesheets loaded, that typically people are breaking their CSS down into multiple stylesheets. At the 50th percentile this works out as 31 rules per stylesheet, growing to 276 rules on desktop and 285 rules for mobile at the 90th percentile.

Selectors and the cascade

2022 saw a shake-up with regard to the cascade with `@layer` landing in all engines. This new at-rule enables the grouping of selectors into layers, the order of precedence of the layers can then be managed.

It's a little early to see widespread usage of this new method of managing the cascade, but let's take a look at how selector usage has evolved.

Class names

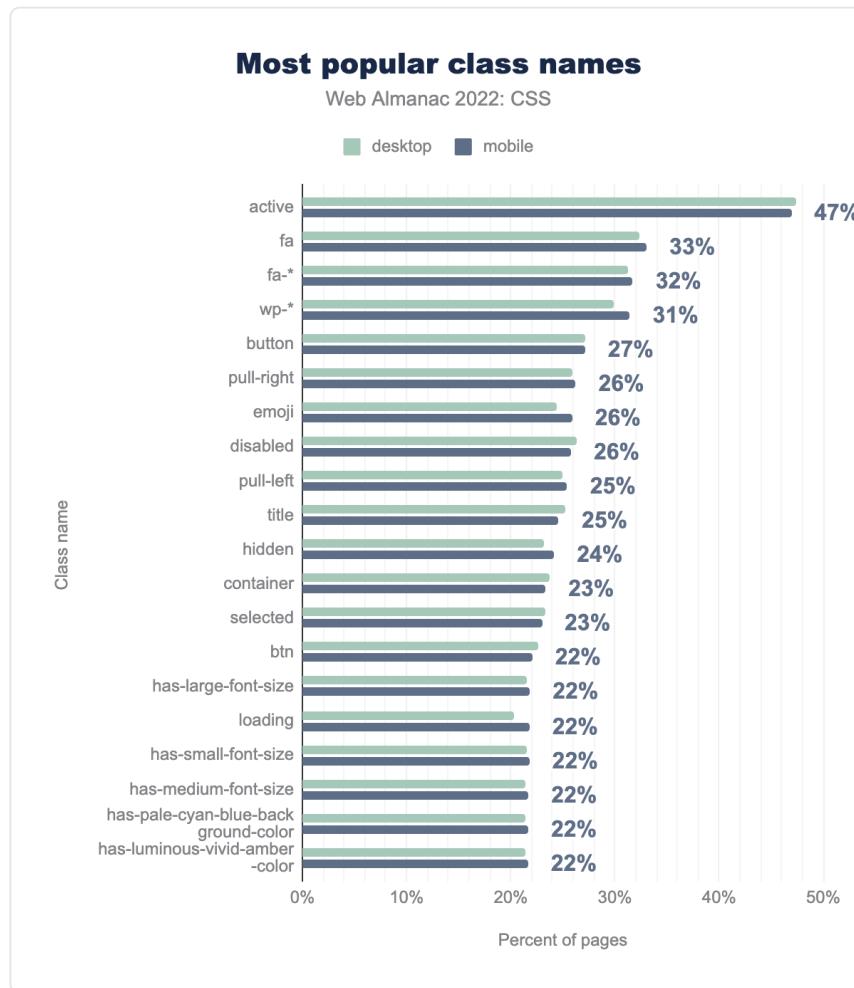


Figure 1.5. The most popular class names by the percent of pages.

As in 2020 and 2021 the most popular class name on the web is `active`. The `fa`, `fa-*` prefixes for Font Awesome still coming second and third. However, `wp-*` class names have crept up the rankings, moving to fourth place. They now show up on 31% of pages, having been at 20% in 2021. We also see class names such as `has-large-font-size` appearing, these are used in the new WordPress Block Editor.

`clearfix` has disappeared from the top 20, it is now found on only 10% of pages, a very clear

indication that float-based layouts are vanishing from the web.

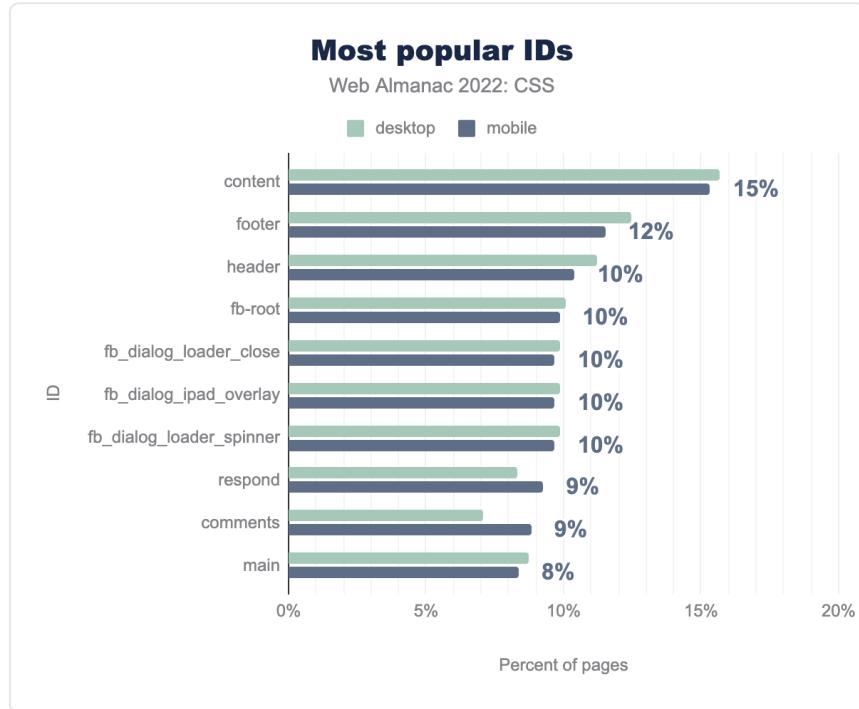


Figure 1.6. The most popular ID names by percent of pages.

The name `content` is once again the most popular ID name, followed by `footer`, and `header`. The IDs starting with `fb_` indicate use of Facebook widgets. In 2021 IDs beginning with `rc-`, indicating use of Google's reCAPTCHA system were seen on 7% of pages, and are still seen with the same frequency, despite being pushed out of the top ten by the Facebook ID names.

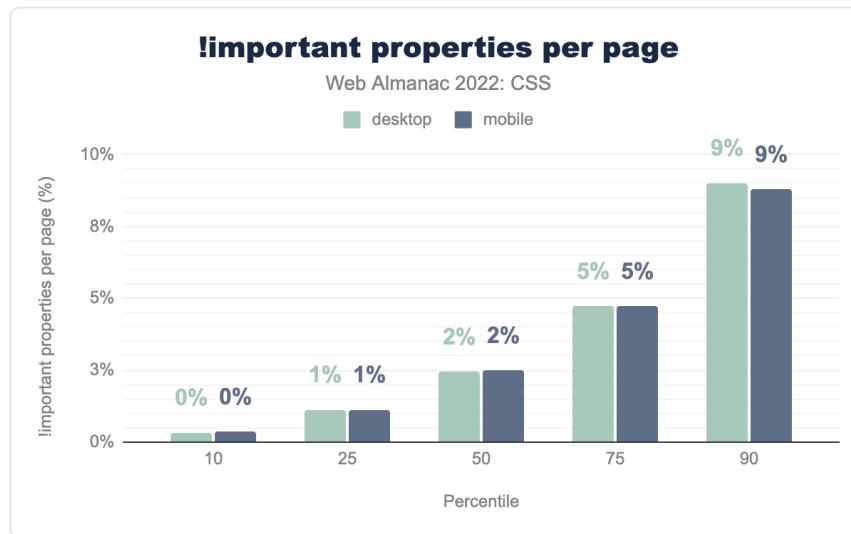
!important

Figure 1.7. The distribution of the number of `!important` properties per page.

The use of `!important` has slightly increased for the top two percentiles this year. As `@layer` usage takes hold, it will be interesting to see how this impacts the use of this property, typically used to deal with specificity issues.

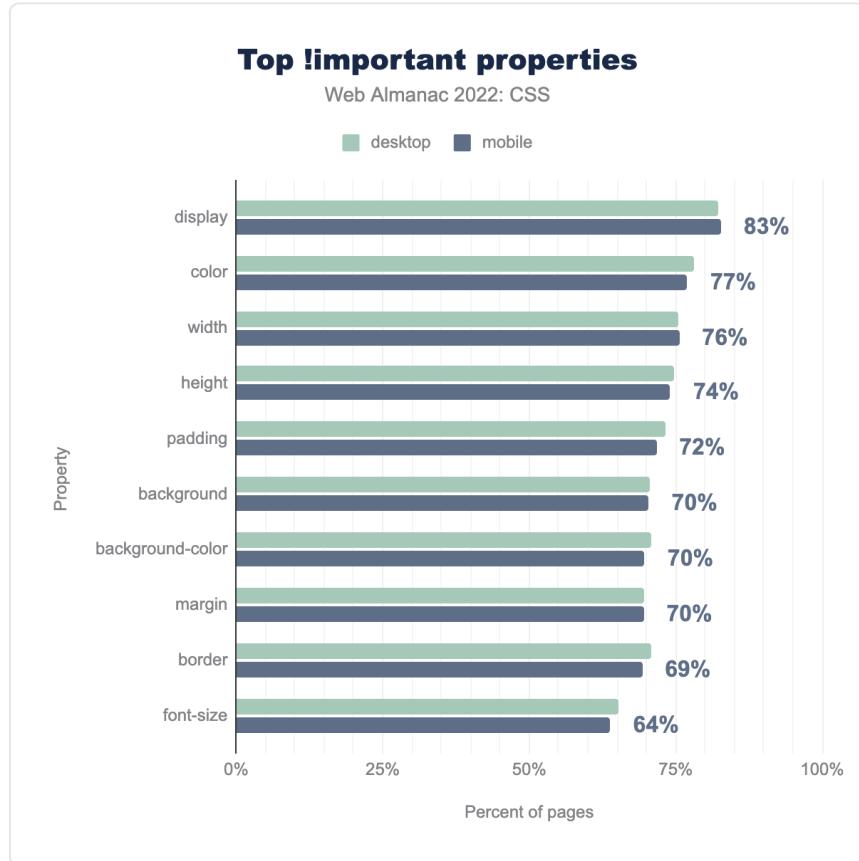


Figure 1.8. The top properties that `!important` is applied to by percent of pages.

In terms of what `!important` is applied to, the top properties remain unchanged. However, `position` has fallen out of the top ten, to be replaced with `font-size`.

Selector specificity

Percentile	Desktop	Mobile
10	0,1,0	0,1,0
25	0,1,2	0,1,3
50	0,2,0	0,2,0
75	0,2,0	0,2,0
90	0,3,0	0,3,0

Figure 1.9. Distribution of the median specificity per page.

Except for desktop at the 25th percentile, median specificity values are exactly the same as last year, remaining constant over the past two years. These values indicate the flattened specificity created by methodologies such as BEM³.

3. <https://en.bem.info/methodology/quick-start/>

Pseudo-classes and -elements

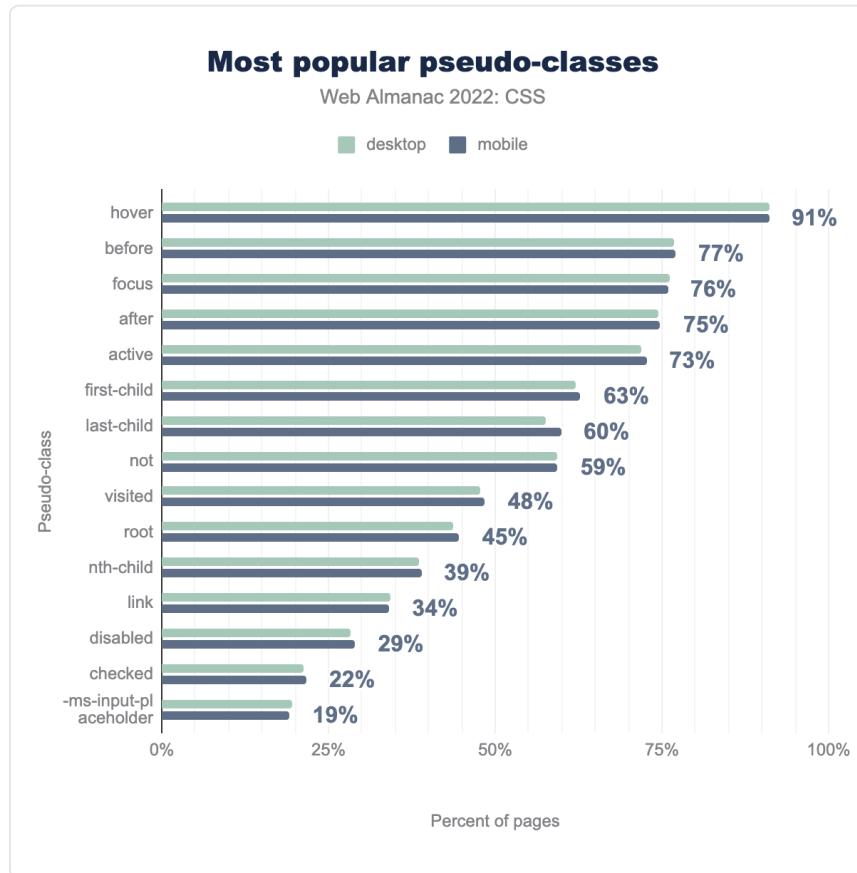


Figure 1.10. Most popular pseudo-classes by percent of pages.

Once again the user-action pseudo-classes `:hover`, `:focus`, and `:active` are in the top three spots. The negation pseudo-class `:not()` also continues its rise in popularity, along with `:root`, likely used to create custom properties.

Last year it was noted that `:focus-visible`, a way to style elements in focus in a way that better matches user expectations, appeared in less than 1% of pages. The property has been available in all three major engines since March 2022, and is now found on 10% of desktop and 9% of mobile pages.

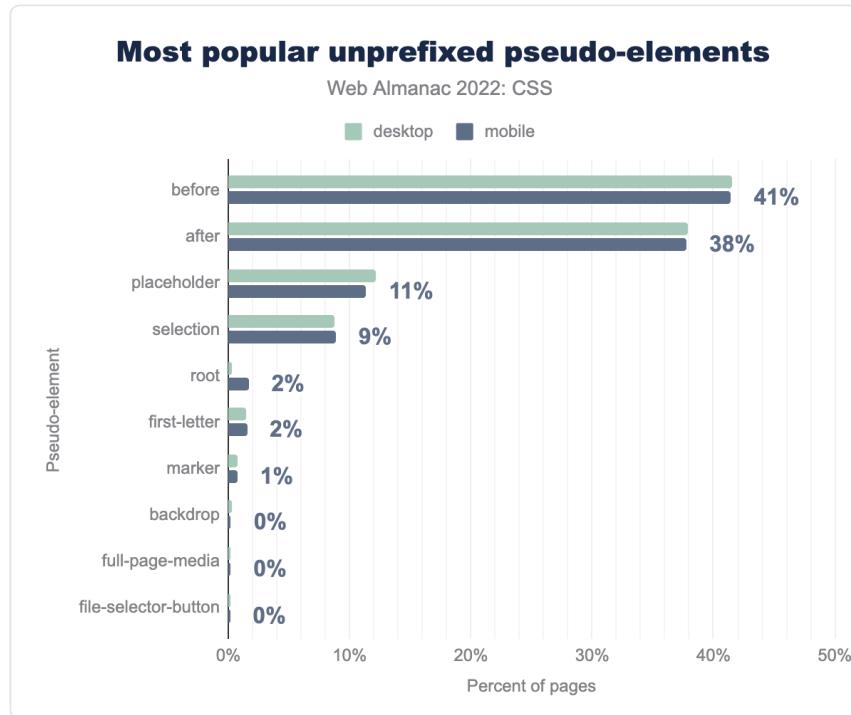


Figure 1.11. Most popular pseudo-elements by percent of pages.

We filter out any prefixed, and therefore browser-specific, pseudo-elements. These are typically used to select interface components or parts of browser chrome, and we are interested in the pseudo-elements developers are actually using.

The use of `::before` and `::after` has increased since last year. These are used to insert generated content into the document. By checking usage of the `content` property, it is possible to see that this is most often being used to insert an empty string, used for styling purposes. Generated content is one way to style a grid area without needing to add an element; perhaps this has contributed to the rise in usage of these properties?

Use of the `::marker` pseudo-element has now made 1%, showing that people are slowly starting to take advantage of the ability to select and style list markers.

Attribute selectors

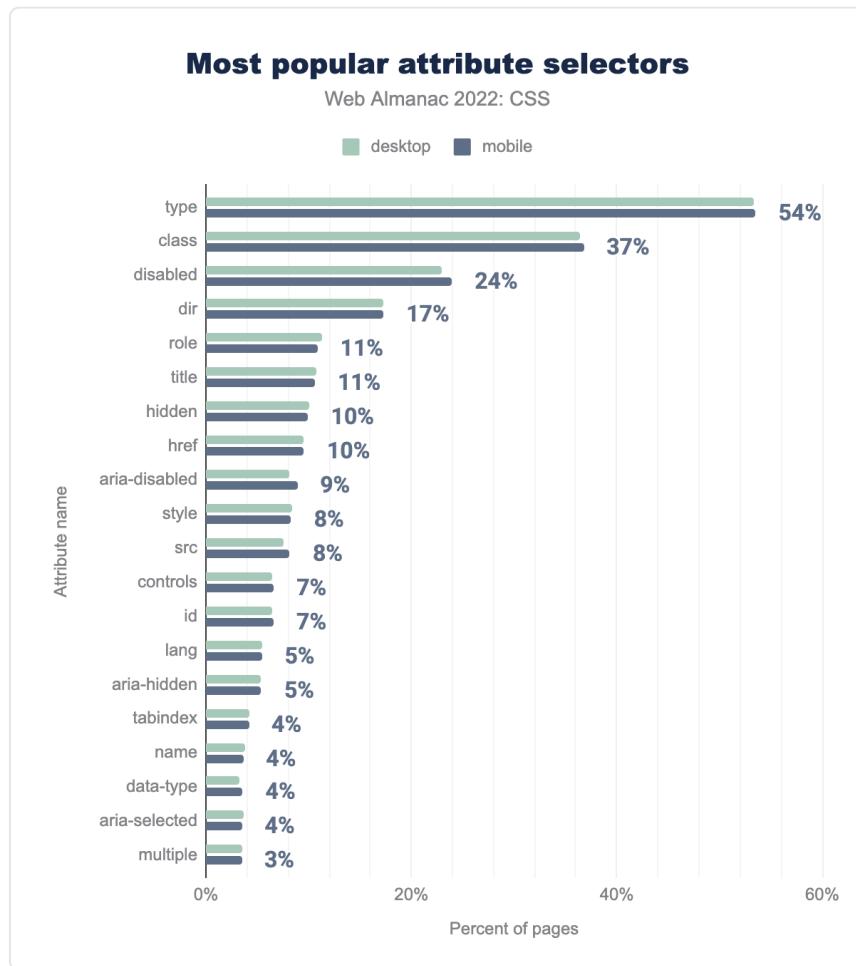


Figure 1.12. Most popular attribute selectors by percent of pages.

The most popular attribute selector is `type`, found on 54% of pages. The next most popular attribute selectors are `class` on 37%, `disabled` on 25%, and `dir` on 17% of pages.

Values and Units

CSS provides multiple ways to specify values and units, either in set lengths, or calculations

based on global keywords.

Length

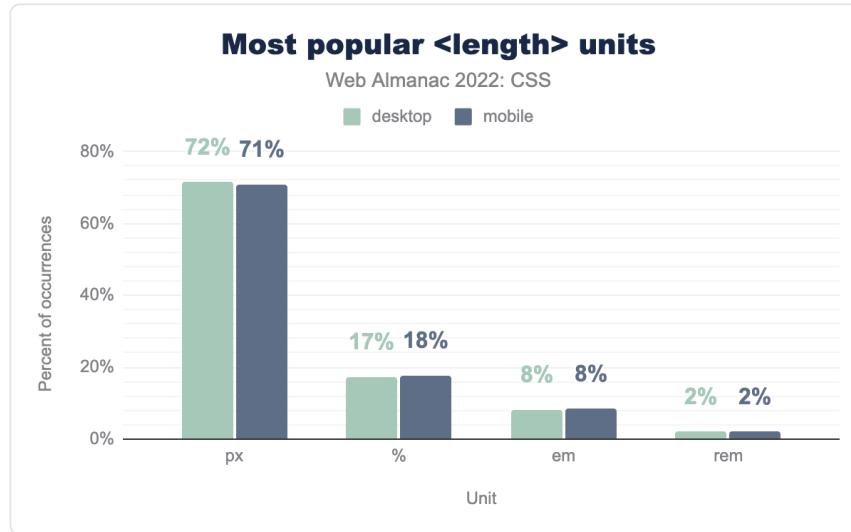


Figure 1.13. Most popular <length> units by percent of pages.

Pixel lengths remain the most popular at 71%, the same percentage as in 2021. The spread of usage remains roughly the same too.

Property	px	<number>	em	%	rem	pt
font-size	(▲2%) 71%	2%	(▼1%) 15%	5%	(▲1%) 6%	(▼1%) 2%
border-radius	(▼1%) 64%	(▼1%) 20%	3.13%	(▲1%) 11%	(▲2%) 2%	0%
line-height	(▼5%) 49%	(▲4%) 35%	12.94%	(▼1%) 2%	(▲1%) 1%	0%
border	(▼1%) 70%	28%	2%	0%	0%	0%
text-indent	(▼5%) 26%	(▲13%) 65%	(▼4%) 5%	(▼3%) 5%	0%	0%
vertical-align	(▼26%) 3%	(▼9%) 3%	(▲39%) 94%	0%	0%	0%
gap	(▲4%) 25%	(▼6%) 10%	(▲32%) 33%	0%	(▼31%) 32%	0%
margin-inline-start	(▼31%) 7%	(▲3%) 49%	(▲30%) 44%	0%	0%	0%
grid-gap	(▲5%) 68%	(▼1%) 10%	(▼2%) 7%	0%	(▼1%) 15%	0%
margin-block-end	(▼1%) 3%	(▲54%) 85%	(▼53%) 12%	0%	0%	0%
padding-inline-start	(▼4%) 29%	(▲11%) 16%	(▼10%) 53%	0%	(▲3%) 3%	0%
mask-position	(▲1%) 1%	(▲3%) 3%	(▼14%) 36%	(▲10%) 60%	0%	0%

Figure 1.14. Distribution of length types per property.

The up and down arrows on this chart show the change from the results in 2021⁴. As seen last year, in the majority of cases there is a shift away from using pixels, in favor of other length units. Once again, the `vertical-align` property saw a huge drop in pixel and `<number>` use, and a large rise in `em` use.

4. <https://almanac.httparchive.org/en/2021/css#fig-15>

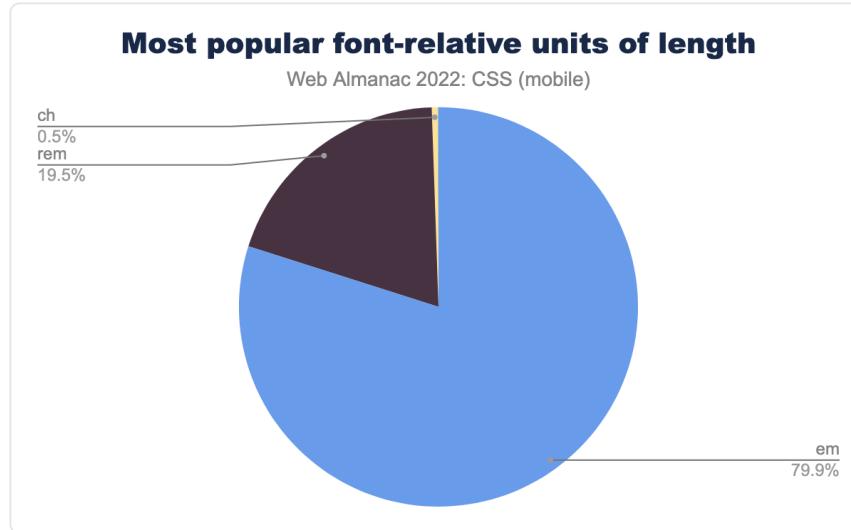


Figure 1.15. The most popular font-relative length units.

While `em` remains the most popular method of sizing fonts, the swing to `rem` continues with a small (just under two point) increase over last year.

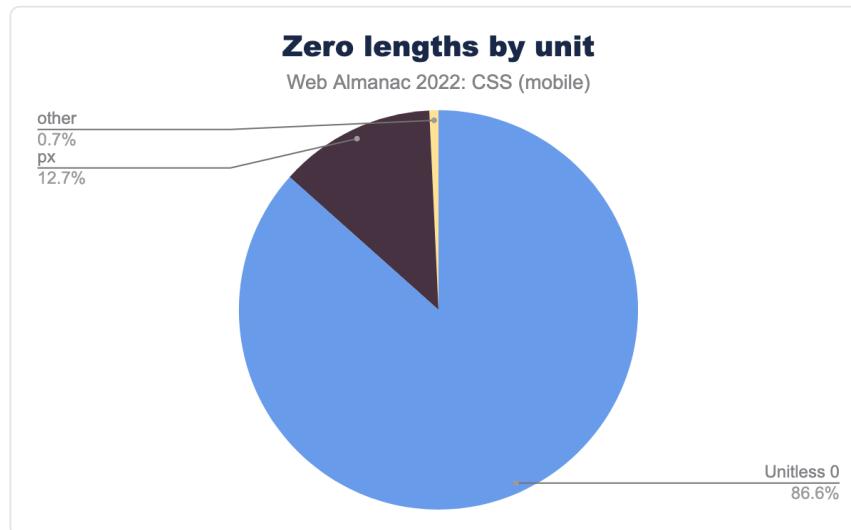


Figure 1.16. The units (or lack thereof) used on zero-length values.

There are a few properties that allow bare `<number>` units (for example, `line-height`), but

`<length>` values have a special case where a length of zero does not require a unit. When we looked at all zero-length values, almost 87% of them omitted the unit, this is a small decrease from last year. Nearly all of those zero lengths that included a unit used pixels (0px).

Calculations

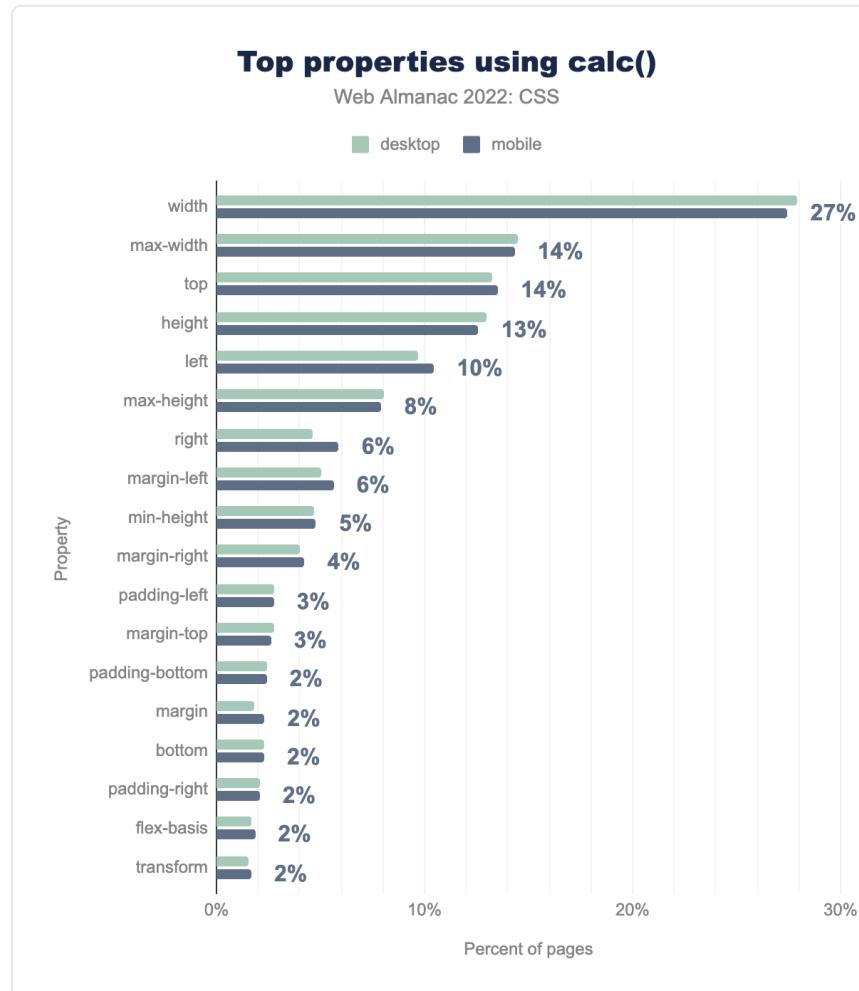


Figure 1.17. The most popular properties using `calc()` functions.

As in previous years, the most popular use of `calc()` is in values for width. This use has dropped 12% points, however, `max-width` has increased in popularity by 9 points.

Top units used in calc()

Web Almanac 2022: CSS

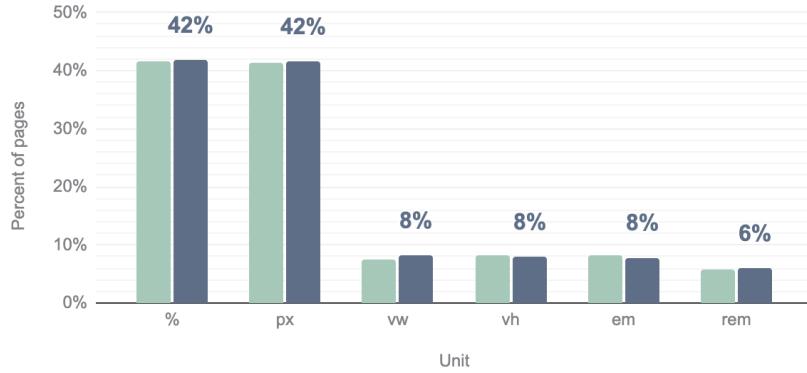
desktop
 mobile


Figure 1.18. The most popular length units used in `calc()` functions.

The percentage of sites using pixels in calculations has decreased 9 points, it is now level with `%` usage at 42%. There is a significant increase in usage for other values, the viewport units `vw` and `vh` both increased from 2% to 8% this year, `em` increased the same amount, and use of `rem` doubled from 3% to 6%.

Top operators used in calc()

Web Almanac 2022: CSS

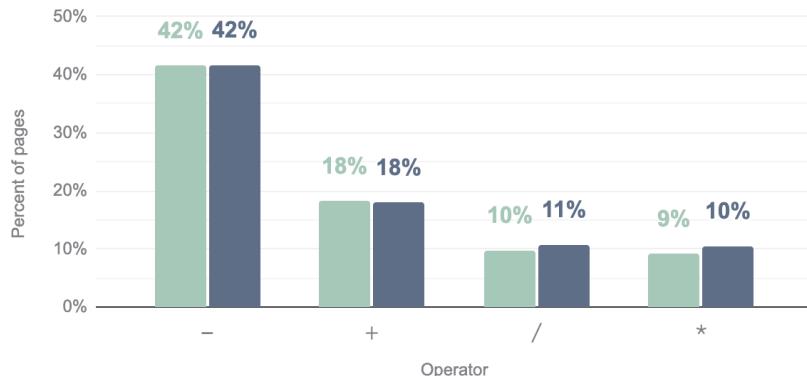
desktop
 mobile


Figure 1.19. The most popular operators used in `calc()` functions.

Subtraction remains the clear favorite in terms of calculation operators, but all four top values saw a drop since 2021, other than addition, which remained the same.

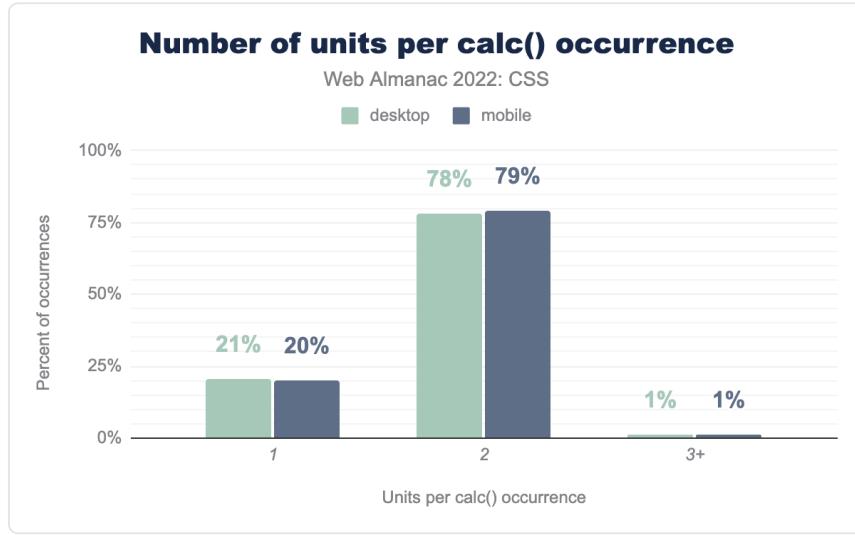


Figure 1.20. The number of unique units used in `calc()` values.

As last year, `calc()` values tend to be fairly simple. The majority using two values, such as the common use case of subtracting a fixed length such as pixels from a percentage. There was a small rise in one unit values, and a small drop in two units.

Global keywords

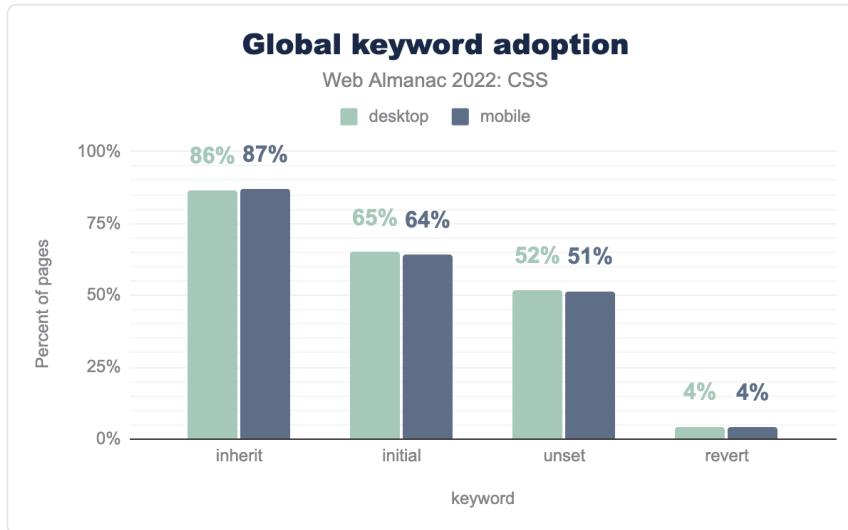


Figure 1.21. Usage of global keyword values.

Last year the use of global keywords had risen significantly, in 2022 `inherit` is found in the same percentage of pages, however the other three values have increased in use. The newer value of `revert` has increased from 1% to 4%.

Custom Properties

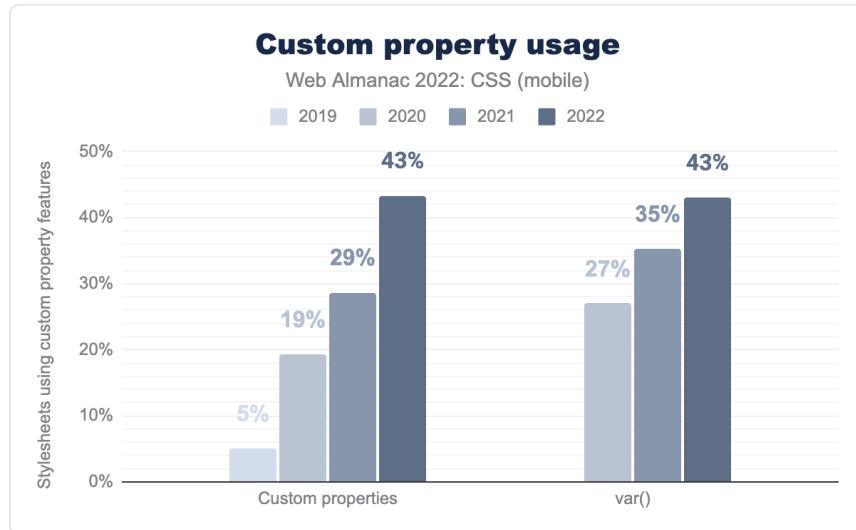


Figure 1.22. Usage of custom properties over the past four years.

Custom properties (sometimes known as CSS variables) have seen a huge surge in use, the growth between 2021 and 2022 is no exception. 43% of pages, for both desktop and mobile are using custom properties and have at least one `var()` function.

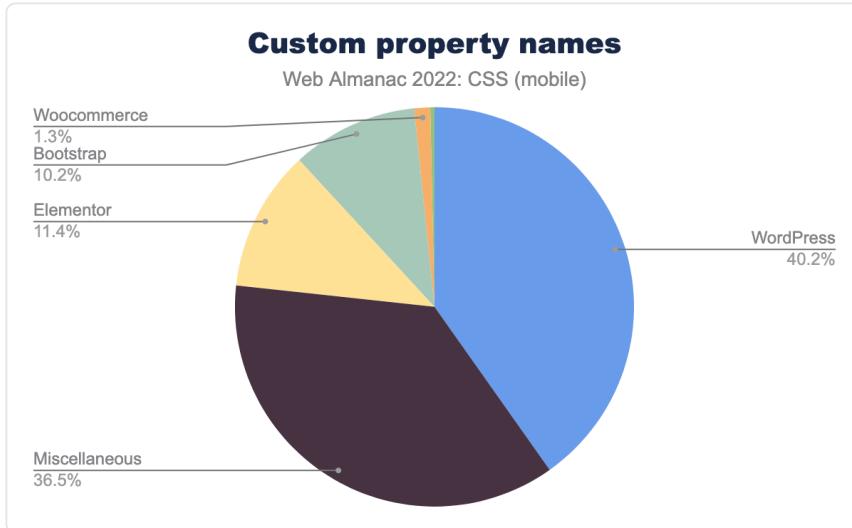


Figure 1.23. Source of common custom property names.

As seen last year, WordPress is the driver for the most common custom property names, these are easily identifiable by the `-wp-*` prefix. Following these, we once again found a lot of color names `-white`, `-blue`, and so on, used to assign a particular shade of that color.

Types

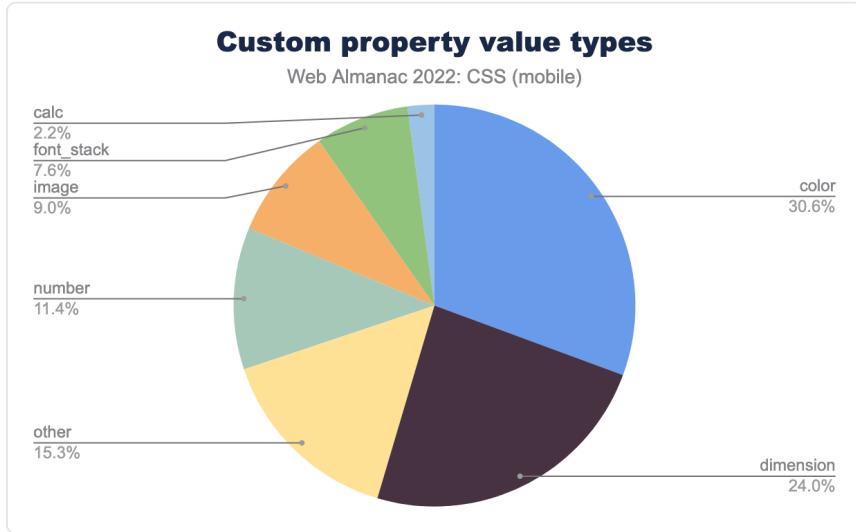


Figure 1.24. Distribution of custom property value types.

The value of a custom property includes a type. For example, `--red: #EF2143` is assigning a color value to `--red`, whereas `--multiplier: 2.5` is assigning a number value. The types have changed a little since last year. We know that setting a color is the most common use of custom properties, and the amount of pages on which color types are found is increasing. However, in terms of the share of usage, this has dropped from 40% to 30%. Entering this distribution is `calc()`, and images as a value type.

Properties

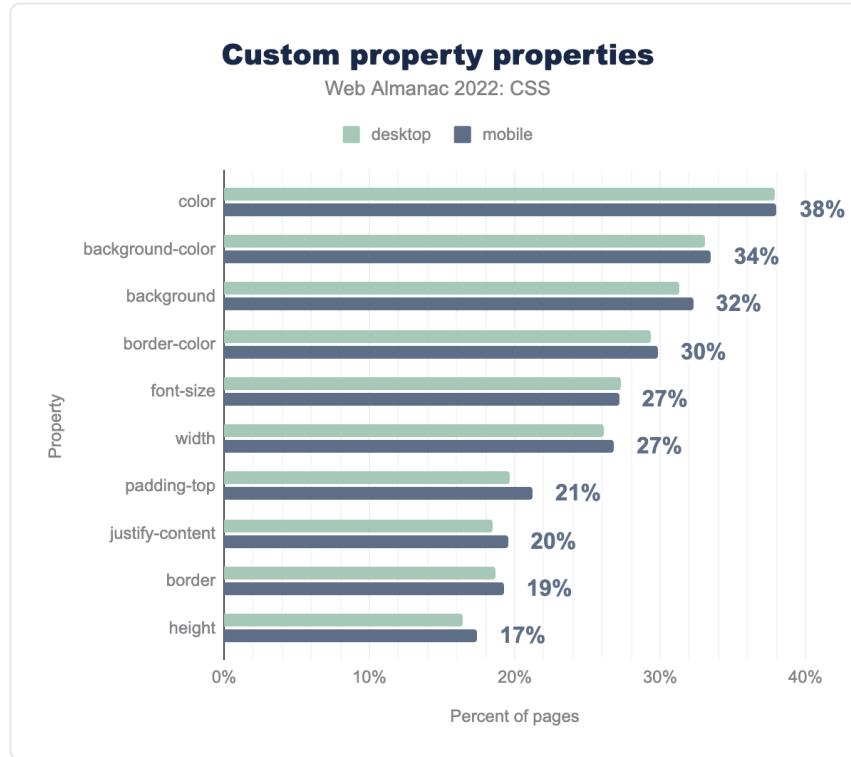


Figure 1.25. The most popular custom property properties by percent of pages.

While the number of pages including these properties has increased, the properties that have custom properties as a value have remained in roughly the same order as last year. Custom properties are most likely to be used for `color`, unsurprisingly as creating color schemes is an obvious use of this functionality. Using the `var()` function to set `font-size` has moved from 10th place to 5th in the list however, and setting the alignment value of `justify-content` has moved into the top ten. In 2021 5% of mobile, and 4% of desktop pages were using custom properties to set this alignment value, this has jumped to 20%. From the data it looks as if some of this increase is due to WordPress usage, 5% of pages use the `-navigation-layout-justify` custom property, for example.

Functions

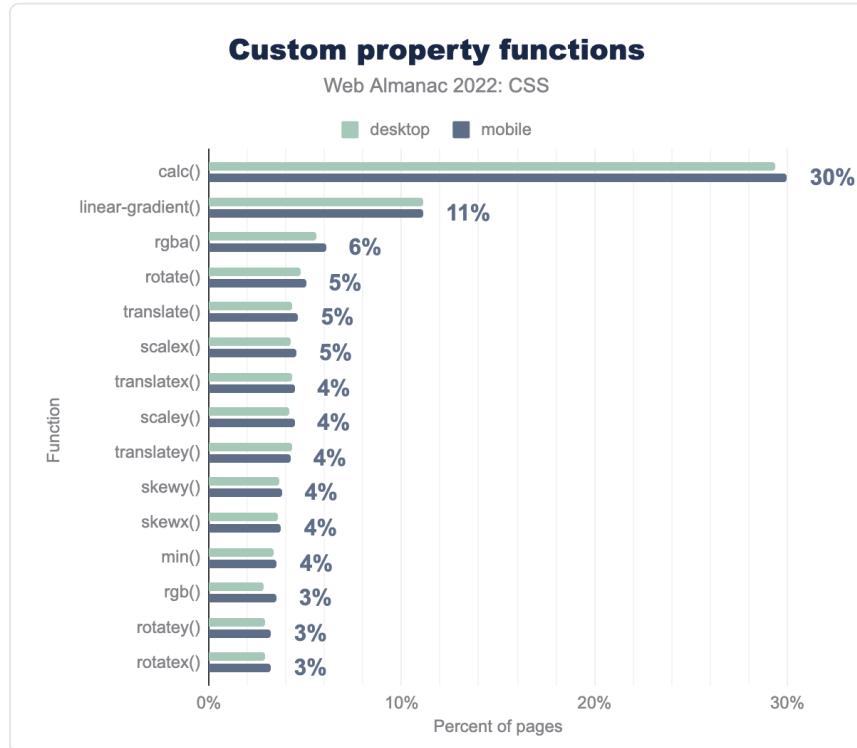


Figure 1.26. The most popular custom property functions by percent of pages.

We saw that `calc()` has started to be notable as a value type for custom properties, and it is by far the most commonly seen function used in this way. It is followed by `linear-gradient()` and the `rgba()` function used to set RGB color values with an alpha channel. After this are the various functions used for transitions and animations, showing a growing use of custom properties in this area.

Complexity

It's possible to include custom properties in the values of other custom properties. Consider this example⁵ from the 2020 Web Almanac:

5. <https://almanac.httparchive.org/en/2020/css#complexity>

```
:root {
  --base-hue: 335; /* depth = 0 */
  --base-color: hsl(var(--base-hue) 90% 50%); /* depth = 1 */
  --background: linear-gradient(var(--base-color), black); /* depth = 2 */
}
```

As the comments in the previous example show, the more that these sub-references are chained together, the greater the depth of the custom property.

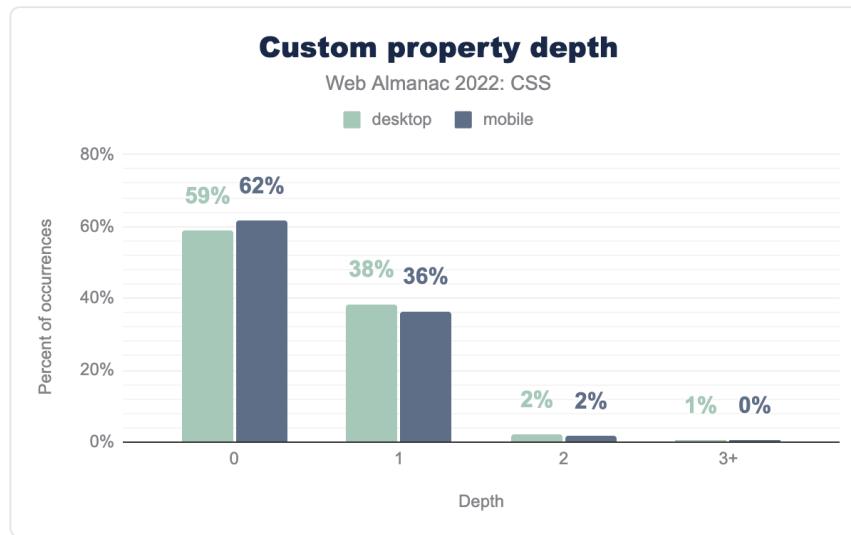


Figure 1.27. The distribution of custom property depth.

As seen in 2021, the vast majority of custom properties had a depth of zero, meaning that they did not include the values of other custom properties in their value. There has been a small increase in the number of properties with a depth of one, and a small decrease in the number with a depth of two. However, it does not seem from the data that our use of custom properties has become much more complex in the past year.

Colors

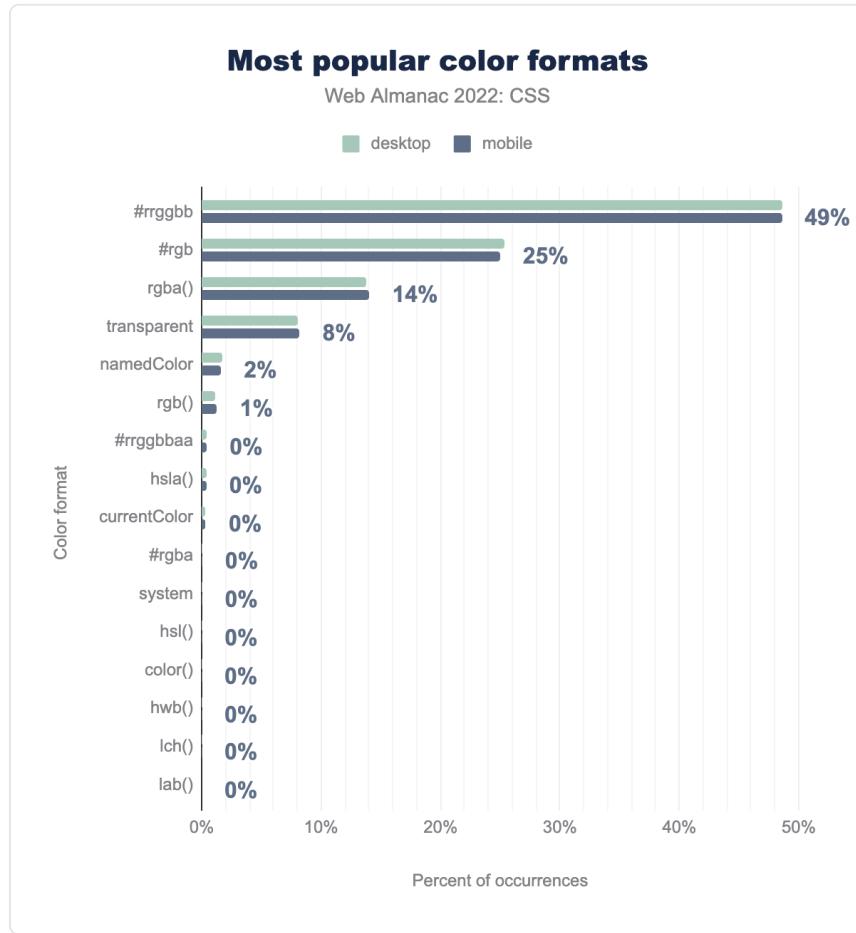


Figure 1.28. The most popular color formats by percent of occurrences.

The use of the time-honored six-digit `#RRGGBB` syntax remains unchanged since 2021, being used in half of color declarations. Despite the widespread availability of eight-digit `#RRGGGBAA` hex, the `rgba()` form is the most widely used way to add an alpha component, likely because it was implemented in browsers much earlier.

The usage of other values showed a similar story; the web community hasn't yet started to take advantage of other color formats, even widely supported ones such as `hsl()`.

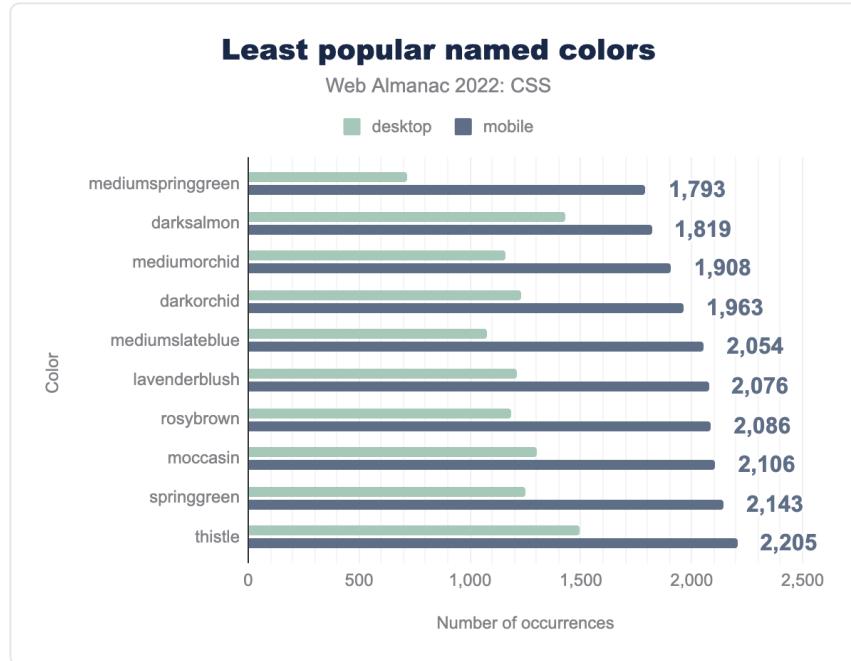


Figure 1.29. The least popular named colors by number of occurrences.

8% of pages use the keyword `transparent`, making it the most popular named color. 2% of pages use other named colors, `white` being the most popular followed by `black`. At the other end of the scale `mediumspringgreen` languishes as the least popular color.

Alpha support and use

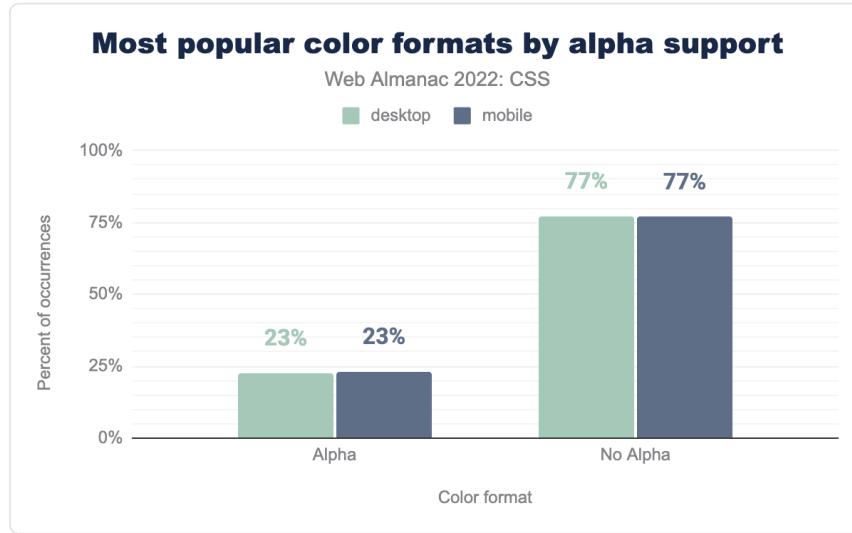


Figure 1.30. The most popular color formats by alpha support.

The `rgba()` function is the third most popular color format, used substantially more than the `rgb()` form, presumably in order to make use of alpha channel support. We looked at the occurrences of values with and without alpha support, to find that 77% of color formats used do not have support for an alpha channel.

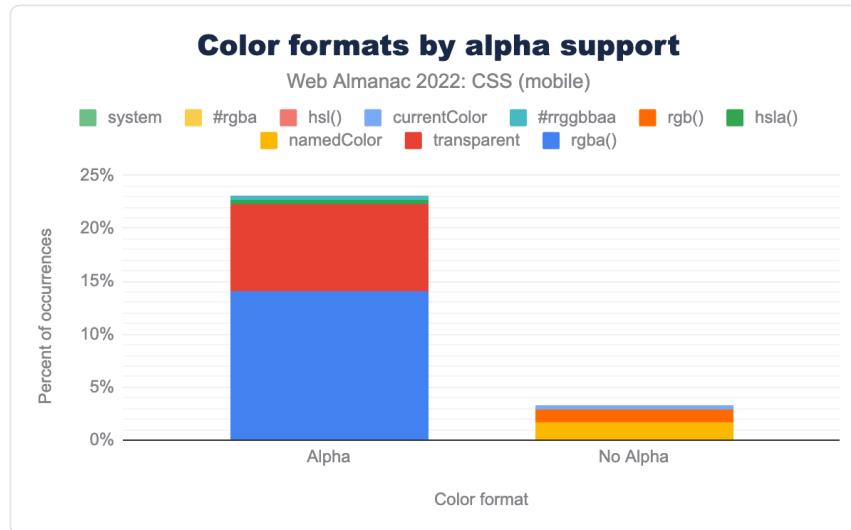


Figure 1.31. Distribution of color formats by alpha support.

As we would expect from other data, `rgba()` is the most popular alpha-supporting format in use, followed by the `transparent` keyword. Other formats such as `hsla()` barely feature.

New color properties and values

There are interesting things happening in the world of color. In addition to new color spaces, we have a number of color-related properties and values. We wondered if any of these were making an impact on the data.

The `accent-color` property lets you add your brand color as an accent color to notoriously hard-to-style form elements such as checkboxes, radio buttons, and range sliders. Perhaps due to the fact it has only been available in all engines since March this year, it still shows less than 0.3% usage.

Another property becoming available in all engines this year is `color-scheme`, a property that lets you specify in which color schemes (light or dark) a component can be rendered. This property is, somewhat surprisingly, so far only found in 0.2% of pages.

Gradients and Images

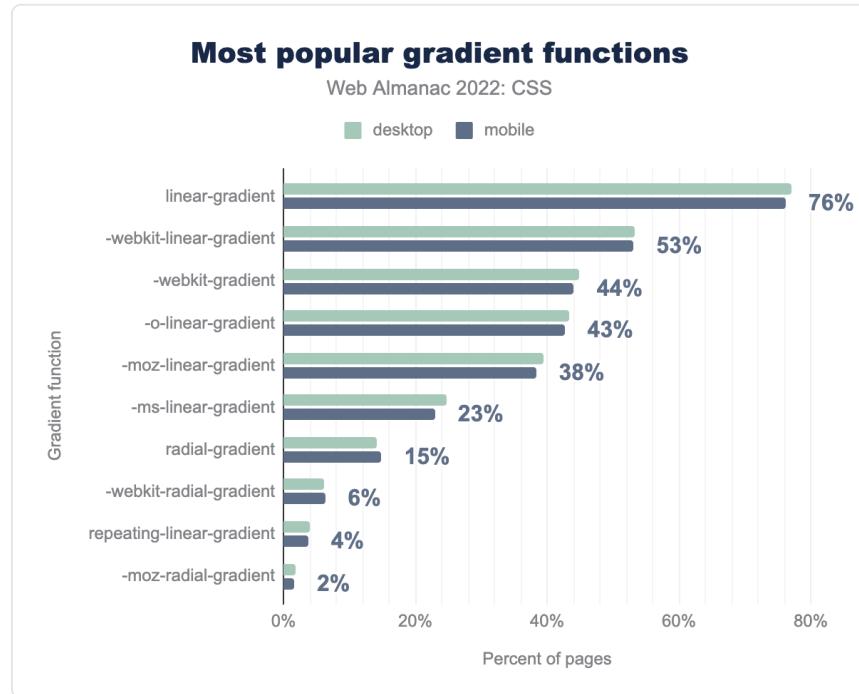


Figure 1.32. The most popular gradient functions by percent of pages.

Linear gradients continue as the leading choice, appearing on a slightly higher percentage of pages than in 2021, however gradient use stays pretty much the same for the last two years. There is still a very high frequency of prefix use when it comes to the `linear-gradient` property, despite this having been supported unprefixed in all engines for over nine years.

Image formats

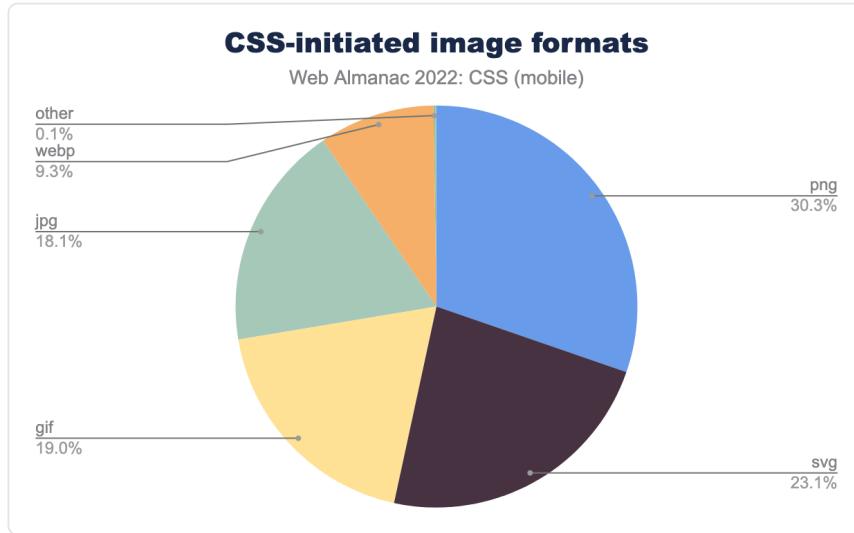


Figure 1.33. Image formats as loaded from CSS.

This chart breaks down the image formats of images loaded from CSS. It does not include images loaded from HTML, just those that appear in a style rule. There has been a significant swing away from PNG—down from 44% to 30%—with SVG and WebP each seeing an increase of 6 percentage points.

Number of images in CSS

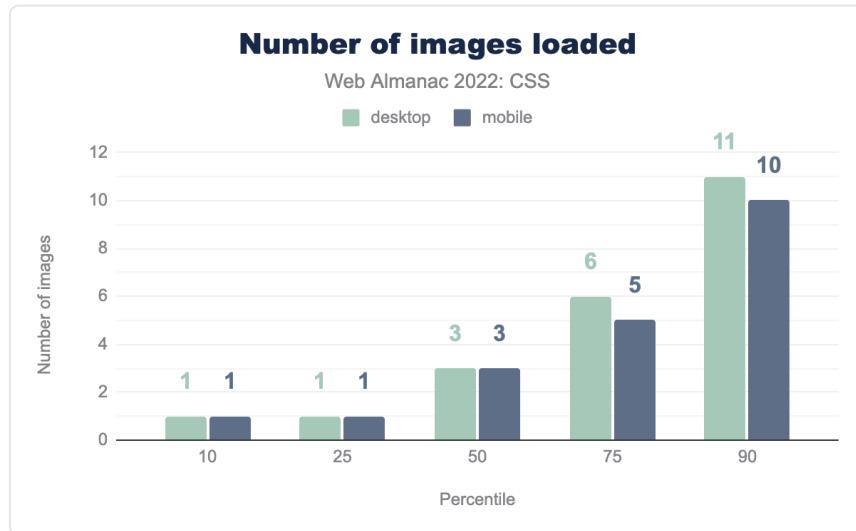


Figure 1.34. Distribution of number of images loaded from CSS.

The number of images loaded from CSS remains the same as in 2021. CSS doesn't cause many image loads: the lower two percentiles came in at one image each, and even the 90th percentile hovered around 10 images, across all image types.

Weight of images in CSS

While CSS doesn't cause many image loads, the weight of those images is important. The data showed that image weight has increased from 2021, despite the fact that the number of images has stayed the same.

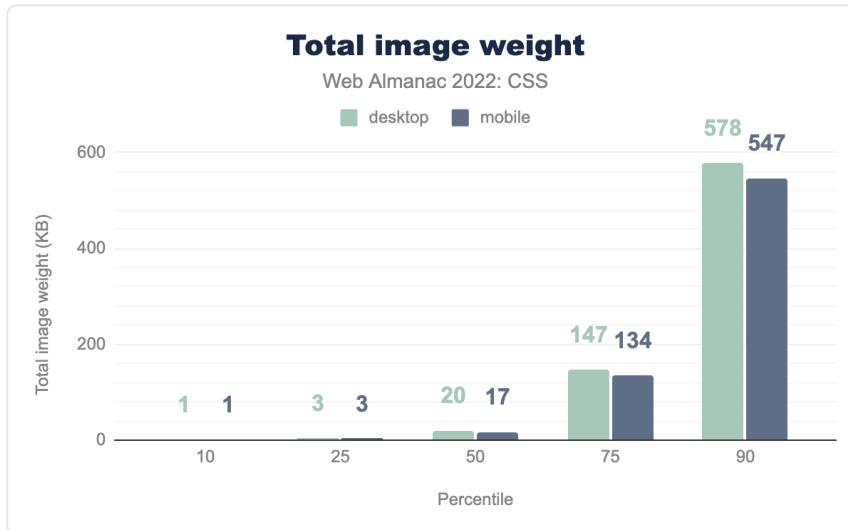


Figure 1.35. Distribution of total weight of images loaded from CSS.

The median page, on mobile, has increased image weight by 1KB to 17KB. At the upper end of the chart however, at the 90th percentile we see an increase of 67KB on mobile and 42KB on desktop. As in 2021, the weight is consistently lower on mobile, an indication that developers are trying to serve smaller images to mobile contexts.

Pixel size of images in CSS

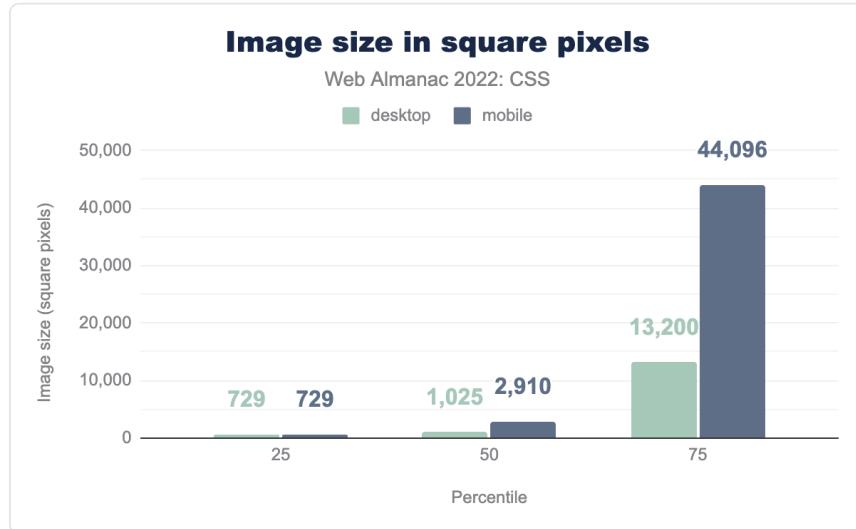


Figure 1.36. Distribution of sizes of images loaded from CSS.

This is an interesting chart which shows that at the lower end of the chart people are serving images of around the same size to desktop and mobile, at the 50th and 75th percentile pages are serving far larger images to their mobile users than they do to desktop. What the data shows is that people are serving much wider images to their mobile users, perhaps to try to account for tablets in landscape mode.

Layout

We have many options to choose from when doing layout on the web, and most sites will be using a variety of these methods. A simple search of the data, looking for property and value combinations to detect layout methods in use, gives us the following table.

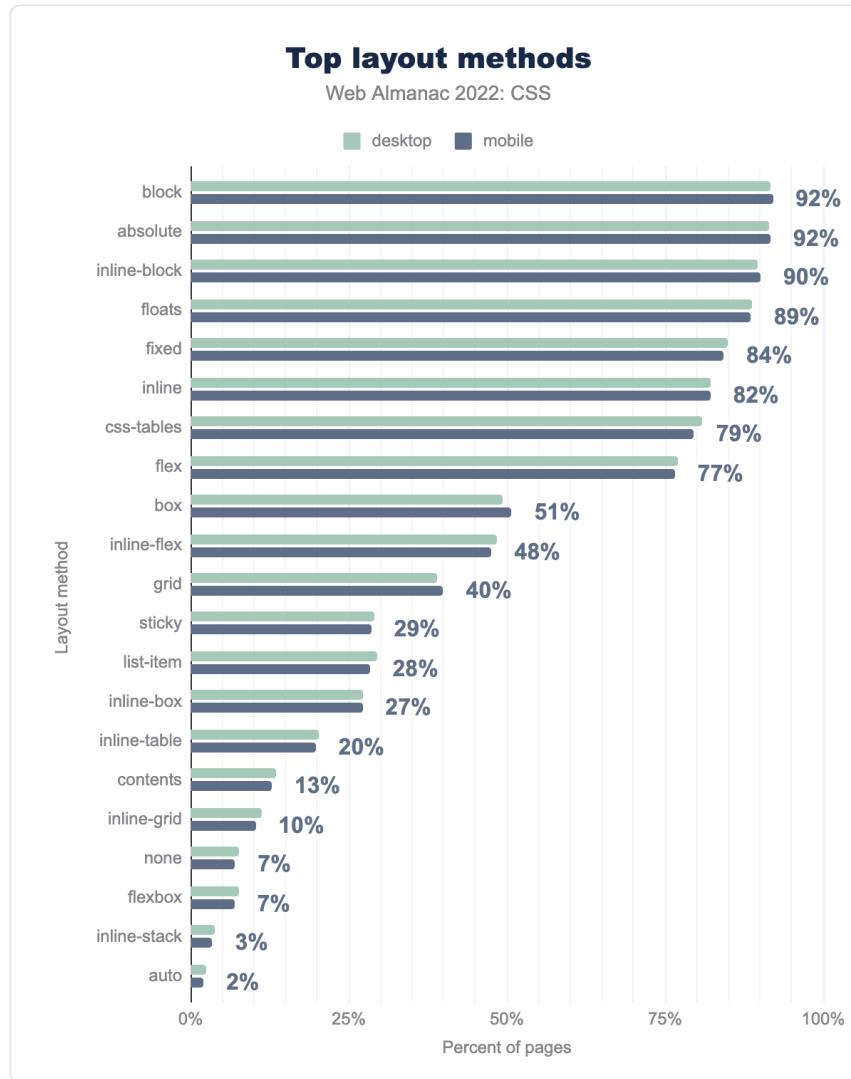


Figure 1.37. Layout methods by percent of pages.

This chart doesn't tell us the main layout method used on a page. It indicates that a property or value appears in the CSS for those pages. For example, 51% of pages are using the old 2009 version of flexbox, with `display: box`. It's likely this has been added for backwards compatibility, perhaps via a tool such as Autoprefixer.

Flexbox and grid adoption

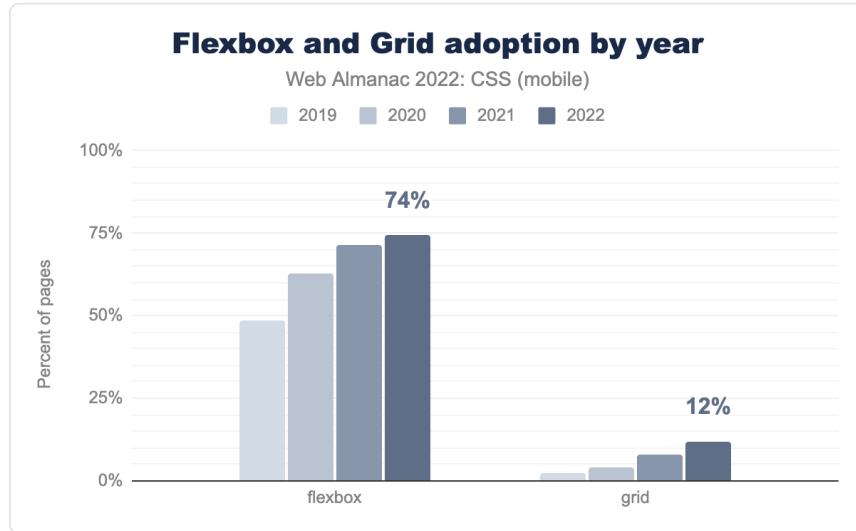


Figure 1.38. Flexbox and grid adoption over the past four years.

Flexbox and grid usage continue to grow. In 2021, flexbox adoption was 71%—it's now at 74%. Grid has jumped from 8% to 12%. Note that, in contrast to the previous section, what is measured here is the percentage of pages that are actually using flexbox or grid for layout, as opposed to the pages that simply have some sort of flexbox or grid property in their stylesheet.

Grid adoption is reasonably slow. We feel this may be due to the prevalence of frameworks being used for layout, many of which have based their layouts on flexbox.

We also took a look at a couple of values of `flex` and `grid` properties that are newer to us, to see how adoption of these new features was developing.

The value of `content` for the `flex-basis` property is an explicit instruction for the browser to look at the intrinsic content size of the item, rather than any width set on it. It's a newer value, at the time of writing not available in the release version of Safari. Currently, only 0.5% of mobile and 0.6% of desktop sites use this value.

The `subgrid` value for `grid-template-rows` and `grid-template-columns` is, at the time the queries were run, only supported in Firefox. Perhaps unsurprisingly, it appears in only 211 mobile and 212 desktop pages in the entire dataset. As the value is part of the Interop 2022 project, we will be interested to see how support grows once this becomes interoperable.

Box sizing

92%

Figure 1.39. The percentage of pages that set `box-sizing: border-box`.

The web has overwhelmingly voted to reject the original W3C box model in favor of `box-sizing: border-box`. The number of pages using this property and value combination has risen slightly again to over 90% of pages.

44%

Figure 1.40. The percentage of pages that declare `box-sizing: border-box` on the `*` selector.

Almost half of all pages analyzed apply `border-box` sizing to every element on the page via the universal selector (`*`).

Around 22% of pages use `border-box` on checkboxes and radio buttons. We see a lot of `.wp-` classes again, showing that WordPress is responsible for the use on 20% of pages analyzed.

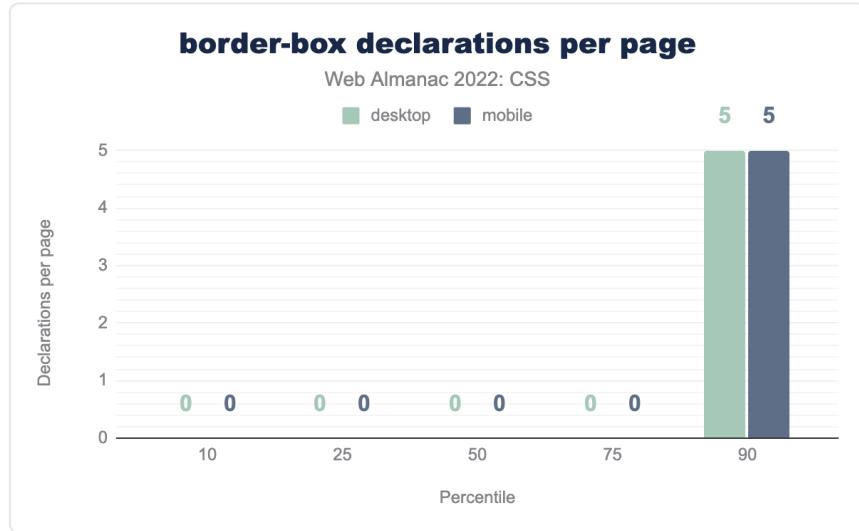


Figure 1.41. Distribution of the number of `border-box` declarations per page.

The median mobile page declares `border-box` 22 times. At the 90th percentile, it's declared an overwhelming 101 times. Note that previous years' queries had a bug affecting this metric. Correcting for that, the results in 2021 are comparable.

Multicolumn

23%

Figure 1.42. The percentage of pages using multi-column layout.

Use of multi-column⁶ layout has increased once again, it's now found on 23% of pages, a rise of 3 points since 2021.

6. https://developer.mozilla.org/docs/Web/CSS/CSS_Columns

The `aspect-ratio` property

2%

Figure 1.43. The percentage of pages using the `aspect-ratio` property.

The new `aspect-ratio` property is used on 2% of pages. This became interoperable towards the end of 2021, so it will be interesting to see usage of this property grow over time.

Transitions and animations

The `animation` property appears on 77% of mobile pages (the same as last year) and a slight increase on desktop to 76.8%. The `transition` property is even more popular, it's found on 85% of mobile and 85.6% of desktop pages. The desktop frequency has dropped slightly by around 4 percentage points since 2021.

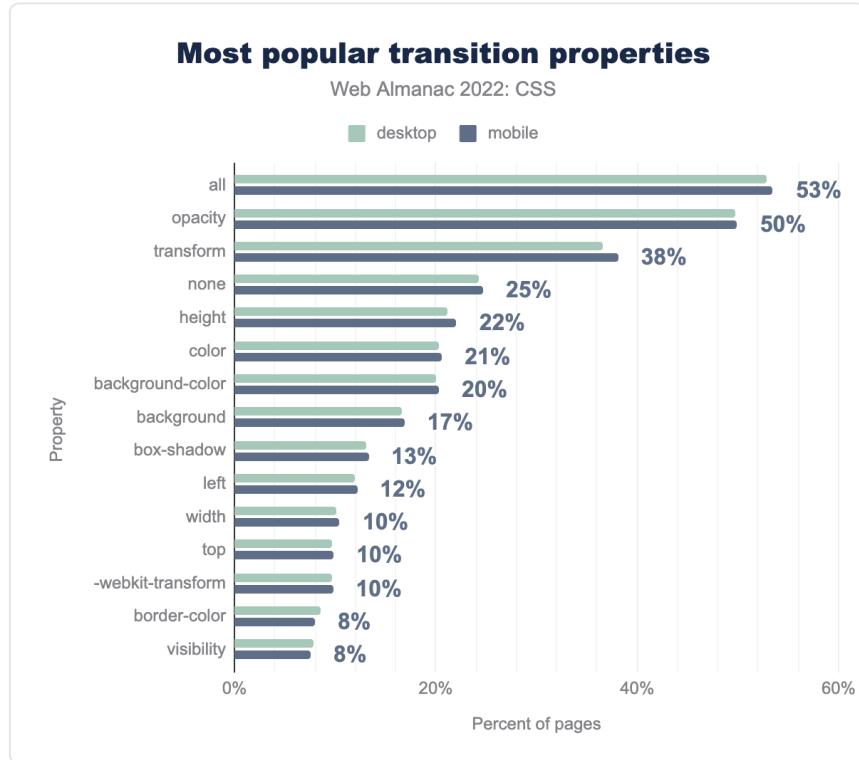


Figure 1.44. The most popular `transition` properties by percent of pages.

As seen last year, the most common usage is to apply transitions to all animatable properties with the `all` keyword. This usage has grown to 53%—up 7 percentage points—followed by `opacity` at 50% of pages.

Distribution of transition durations

Web Almanac 2022: CSS

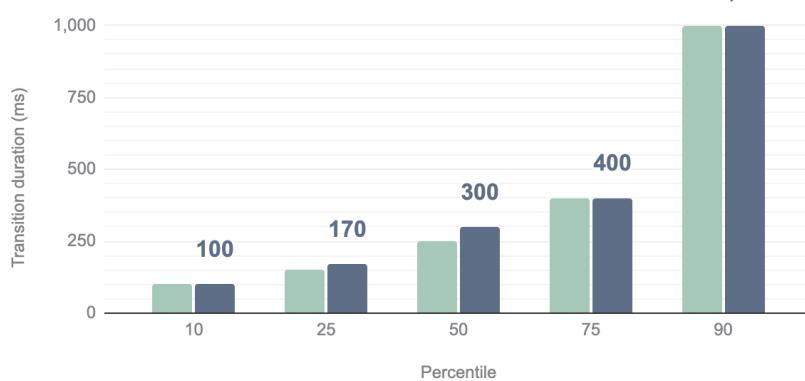
desktop
mobile


Figure 1.45. Distribution of transition durations.

Looking at the duration of transitions, we see a change from last year. In 2021, at the 90th percentile the median transition duration was half a second, this has now jumped to 1 second. We see increases across all top four percentiles.

Distribution of transition delays

Web Almanac 2022: CSS

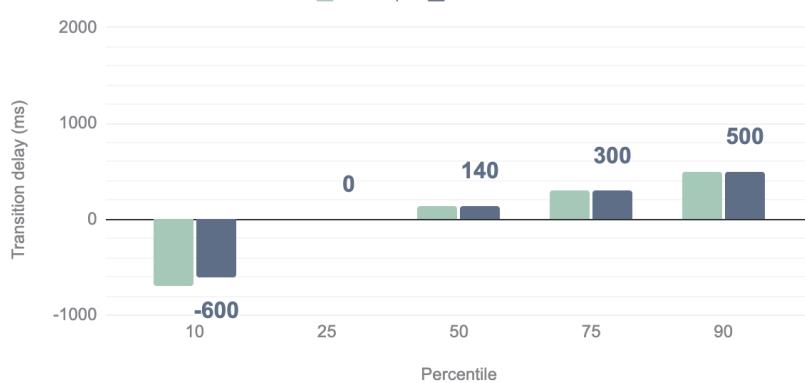
desktop
mobile


Figure 1.46. Distribution of transition delays.

The distribution of transition delays has also changed. The 90th percentile delay has dropped from 1.7 seconds to half a second. Though the 10th percentile median delay is now over half a negative second. This is seen when a transition starts partway through the resulting animation.

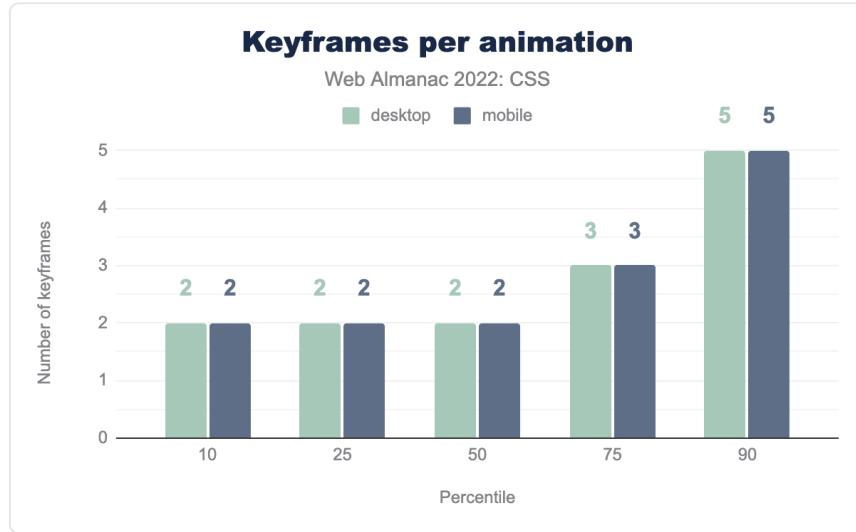


Figure 1.47. Distribution of keyframes per animation.

We also looked at the average number of keyframes used per animation, and found one site that used an astonishing 6,995 keyframes. This was unusual however, and even at the 90th percentile, the number of keyframes per animation is five on both desktop and mobile.

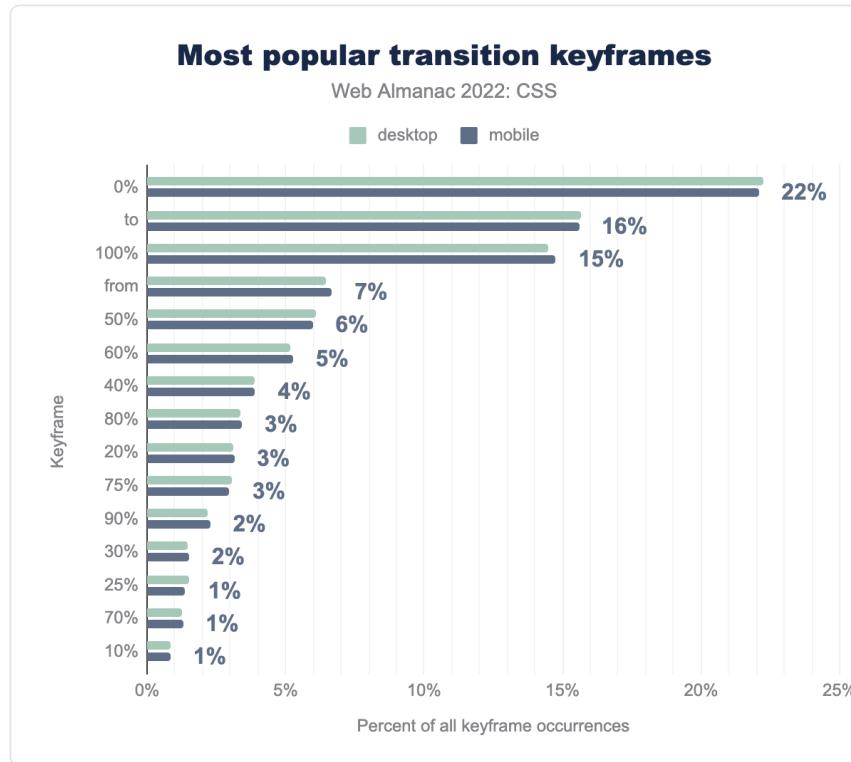


Figure 1.48. The most popular transition keyframes by percent of occurrences.

As you might expect the most popular stops are at 0% to and from 100%, followed by 50%. Developers generally set these stops at 10% intervals, only 1% of pages use 33%, for example.

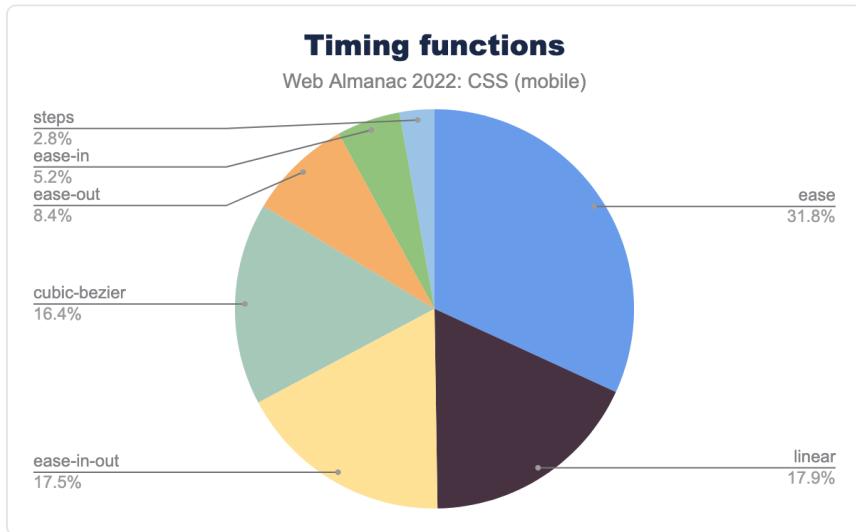


Figure 1.49. Distribution of timing functions.

There has been little change in the distribution of timing functions used during transitions when compared to 2021. As then, the clear leader is `ease`.

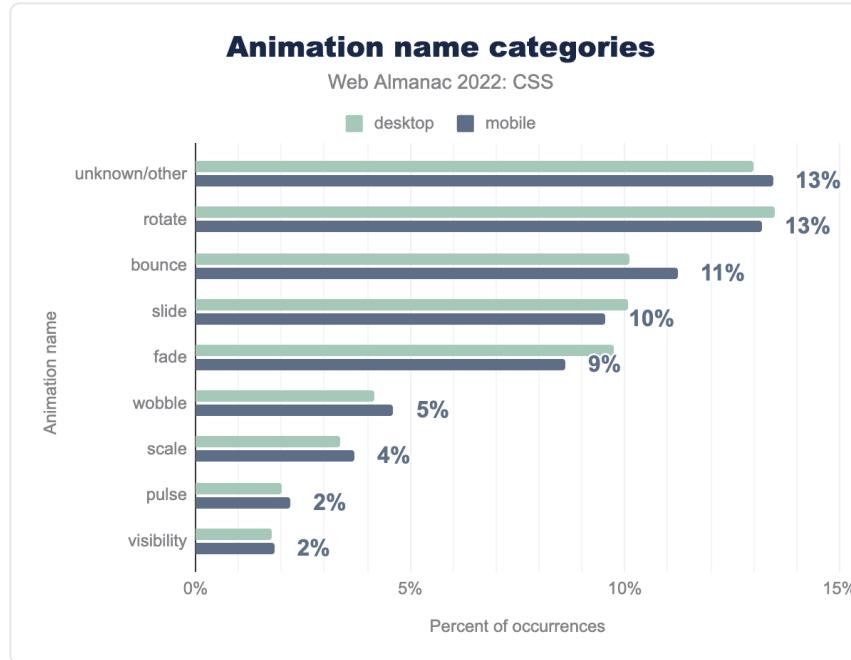


Figure 1.50. Types of animations as identified by animation name.

To understand what developers are using animations for, we took a look at the names used for the animation classes. For example, anything with `spin` in the class name is deemed to be rotate. Rotate animations were the most popular, as in 2021. However the percentage has dropped from 18% to 13%, with bounce animations moving from 5th place to 3rd place in the list.

As last year, the high showing for unknown/other is due to a prevalence of the class name `a`, which we can't map to a specific animation type.

Visual Effects

18%

Figure 1.51. The percentage of pages using blend modes.

We looked at some visual effects being used in CSS. For example, 18% of desktop pages define

styles on the `background-blend-mode` or `mix-blend-mode` properties.

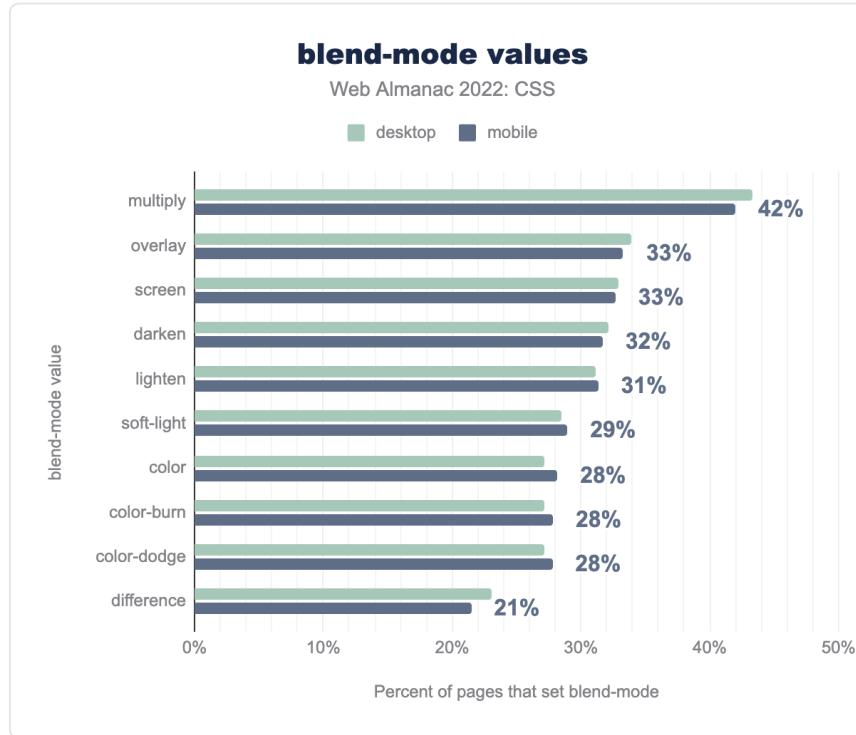


Figure 1.52. Most popular blend modes used on pages that set blend mode.

The most frequently seen value for blend modes was `multiply`, seen on 42% of pages. However there is a fair distribution of other values too.

Around 18% of pages were using a custom property `var(--overlay-mix-blend-mode)`, a specific name that must come from a library or tool of some sort.

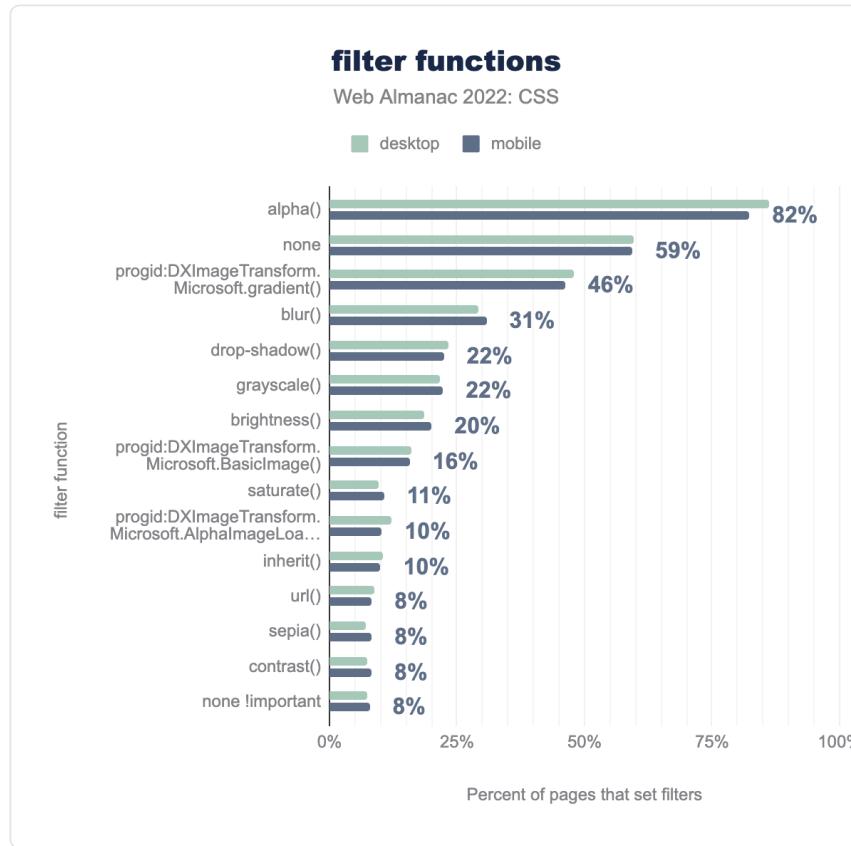


Figure 1.53. Most popular filter functions used on pages that set filters.

Of the percentage of pages that have set filters to apply graphical effects, 82% are using the `alpha()` value, which is non-standard and used for Internet Explorer 8 and below. We also see a high usage of the `Microsoft.gradient()` filter.

Of the standard values⁷, 31% of pages use `blur()`, making it the most popular value after `none`.

7. <https://developer.mozilla.org/docs/Web/CSS/filter>

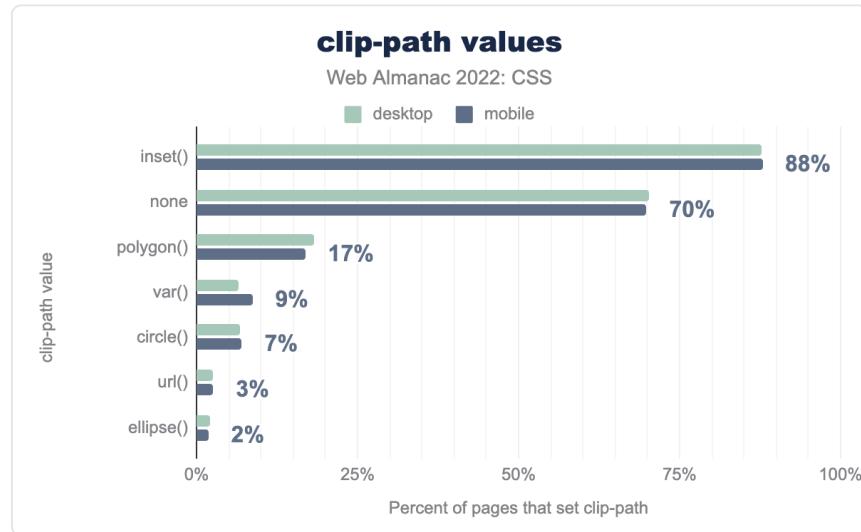


Figure 1.54. Popular `clip-path` values in pages that set `clip-path()`.

In pages that use `clip-path` to clip an element, the vast majority are using `inset()`, the value that simply insets the box of the element, 88% of pages using `clip-path` have used this function.

After that, and the value `none`, most developers have chosen to use `polygon()`, which is the value that gives the most flexibility to define your own path.

Responsive design

While many developers are eagerly anticipating container queries⁸, and new layout methods such as flexbox and grid can often enable a design to work well on multiple screen sizes, media queries⁹ are used in the majority of pages for responsive design.

When developers write media queries, they most often test the width of the viewport. `max-width` and `min-width` were the most popular queries by far, the same as in 2020 and 2021. There was no ranking change in the third and fourth place results either.

8. https://developer.mozilla.org/docs/Web/CSS/CSS_Container_Queries
 9. https://developer.mozilla.org/docs/Web/CSS/Media_Queries/Using_media_queries

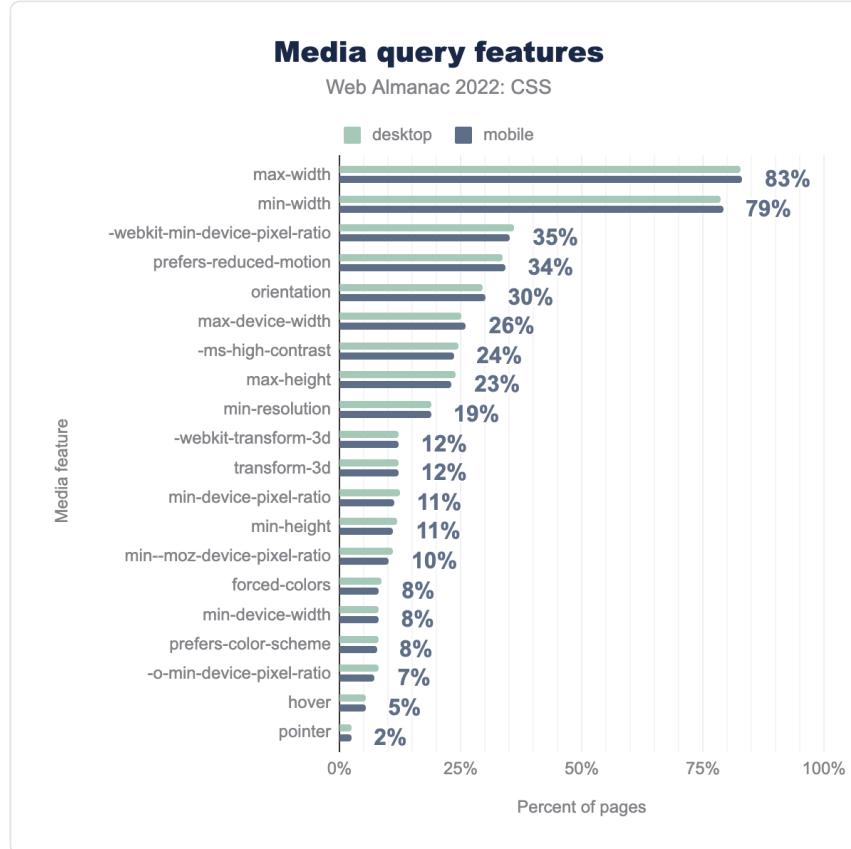


Figure 1.55. Popular media query features.

The `prefers-reduced-motion` media query, however, which was noted in 2021 as rising in the rankings, has now edged out `orientation` to take the fourth spot. This is due to a 2% rise for `prefers-reduced-motion` but also a drop of 4% for `orientation`.

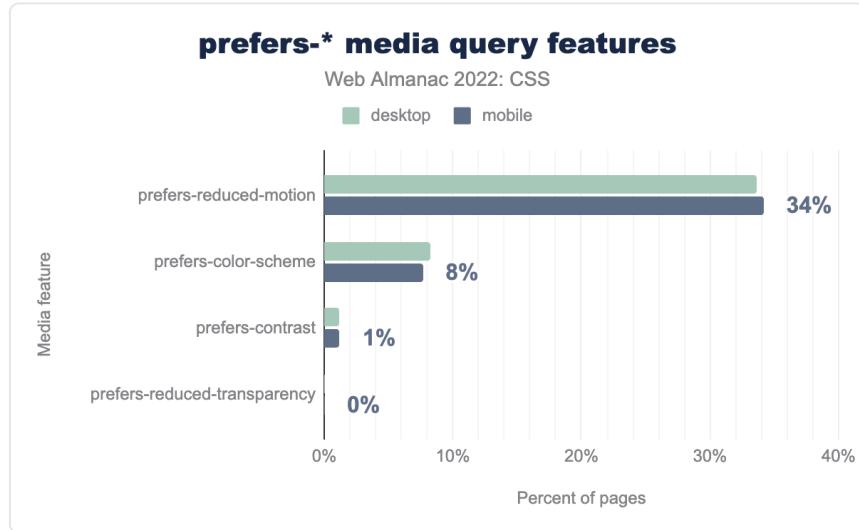


Figure 1.56. Use of user preference features by percent of pages.

If we just look at the `prefers-*` user preference features, we can see that `prefers-reduced-motion` is by far the most popular, due to good browser support plus the prevalence of animations and transitions on the web. The `prefers-color-scheme` feature, checking to see if the user has set a preference for a light or dark scheme, has increased in use slightly, as the use of dark mode on websites and applications becomes more popular.

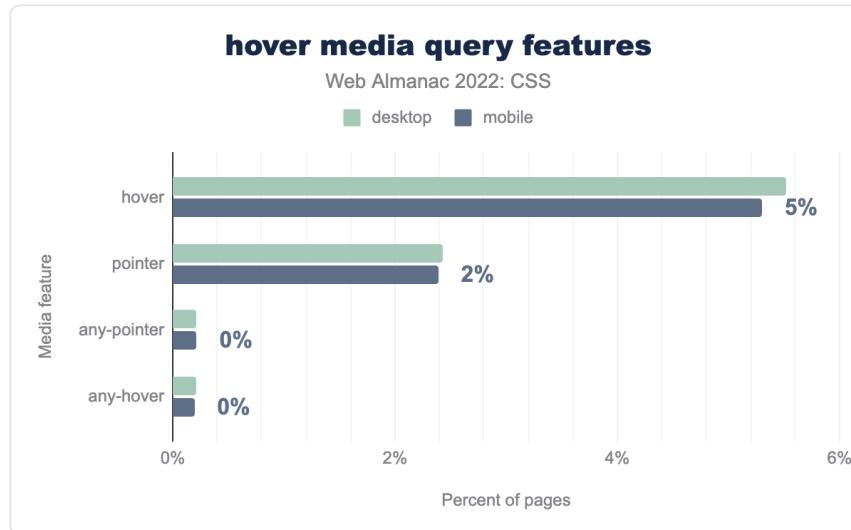


Figure 1.57. Use of hover and pointer media features.

The `hover` and `pointer` media features help developers test the capabilities of the device, and the way the user might be interacting with it. They are a better way to discover if a user is using a touchscreen, for example, than screen size alone given the number of large tablets and touchscreen laptops in use.

Both `hover` and `pointer` now appear in the top ten features. The less useful `any-pointer` and `any-hover` see very little use. Using `any-pointer` allows you to determine if a user has access to a fine pointer such as a mouse or trackpad, even if `pointer` indicates they are currently using the touchscreen. Asking a user to switch is definitely not ideal, though a combination of these features could give you a good understanding of the environment a user is working in.

Common breakpoints

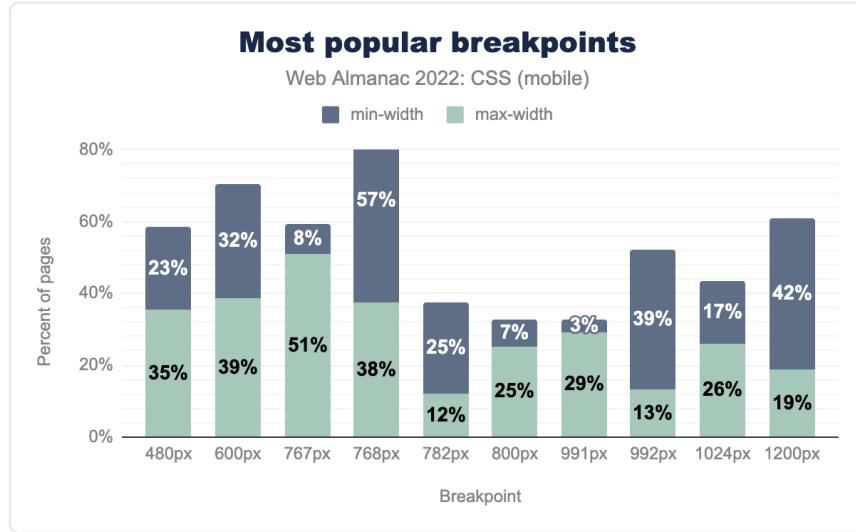


Figure 1.58. Distribution of the most popular breakpoints.

As in the past two years, common breakpoints have changed little. The chart follows the same shape, and the most common breakpoint being a `max-width` of 767px and `min-width` of 768px. As noted in 2021, this corresponds with an iPad in portrait mode.

Once again, breakpoints are overwhelmingly set in pixel values, we haven't converted other values to pixels for the chart. The first `em` value is again `48em`, found at position 78.

Properties changed in queries

We looked at the properties that appear within media query blocks, to see which properties people were changing based on breakpoint.

Most popular properties used in media queries

Web Almanac 2022: CSS

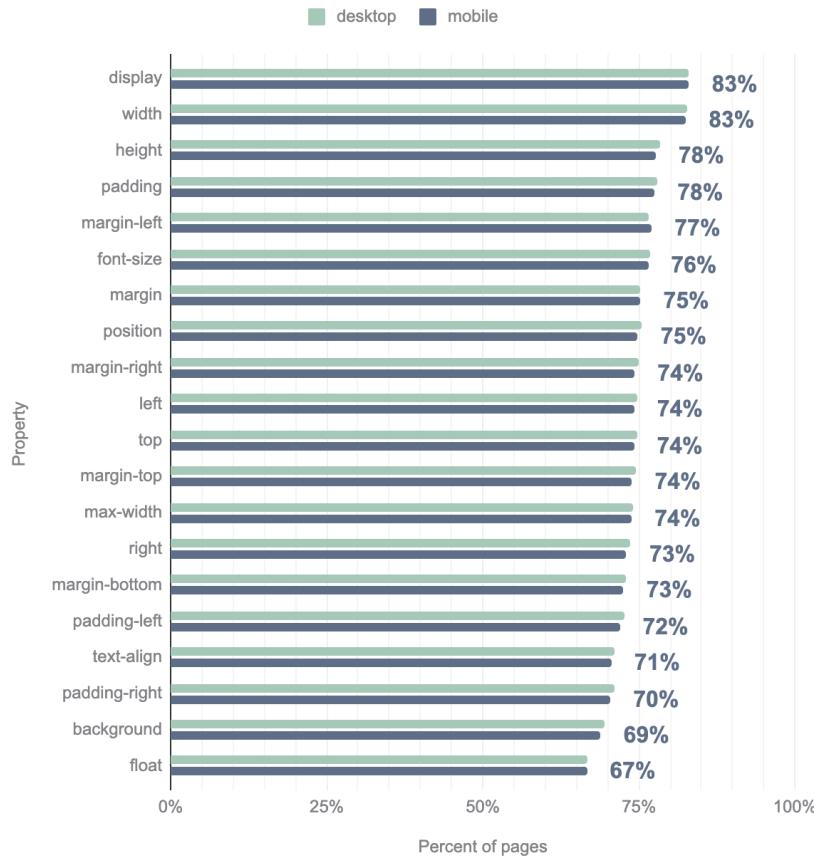


Figure 1.59. Most popular properties found in media query blocks.

The `display` property is still top of the chart for properties changed within media queries, however there has been some reshuffling in the rankings. These are not as dramatic as they might seem. The `color` property has vanished from the chart, however this only represents a change from 74% to 67%. It is joined however by a reduction in usage of `background-color` for 65% to 63%, which makes us wonder if some framework, or perhaps WordPress has stopped using this in a stylesheet.

Another interesting point to note is that in 2020 `font-size` appeared in 73% of media

blocks, and was fifth on the list. In 2021, it showed up in 60% of blocks, appearing at 12th. This year it has gained ground, back to 76% and sixth place.

Feature Queries

Features queries, used for testing for support of a CSS feature, were found on 40% of mobile pages and 38% of desktop pages. This was down from a figure of 48% in 2021. This may indicate that support for common features tested has become great enough for people not to worry about testing for the feature before use.

The number of feature query blocks per page is 4 at the 75th percentile, and at the 90th percentile 7 for desktop and 8 for mobile. We did find one site however with 1,722 feature query blocks.

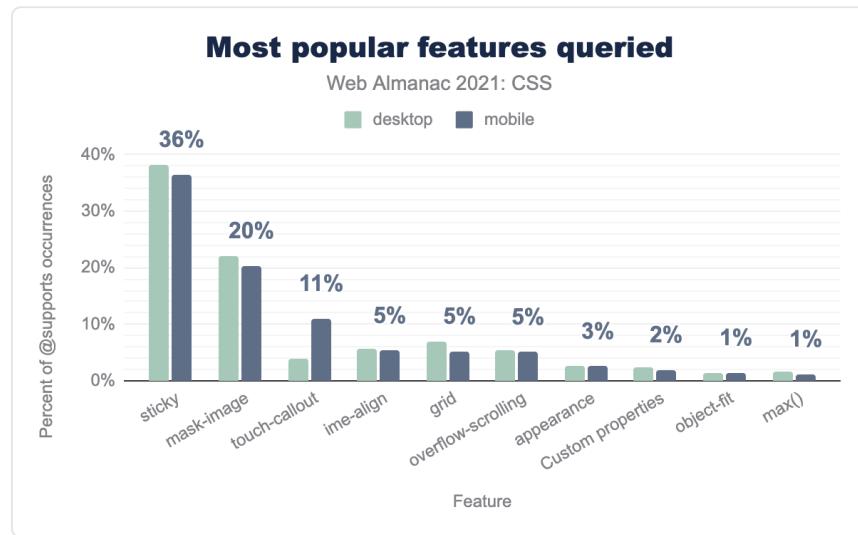


Figure 1.60. Most popular features tested for with feature queries.

As with last year, the most popular feature tested for in feature queries was `position: sticky`, however this has fallen from 53% to 36% of occurrences, perhaps due to the improved browser support for this feature.

Non-standard features show up strongly in these tests, with `touch-callout` (`-webkit-touch-callout`) and `ime-align` (`-ms-ime-align`). The former has grown in usage from 5% to 11%, while `ime-align` has dropped from 7% to 5%.

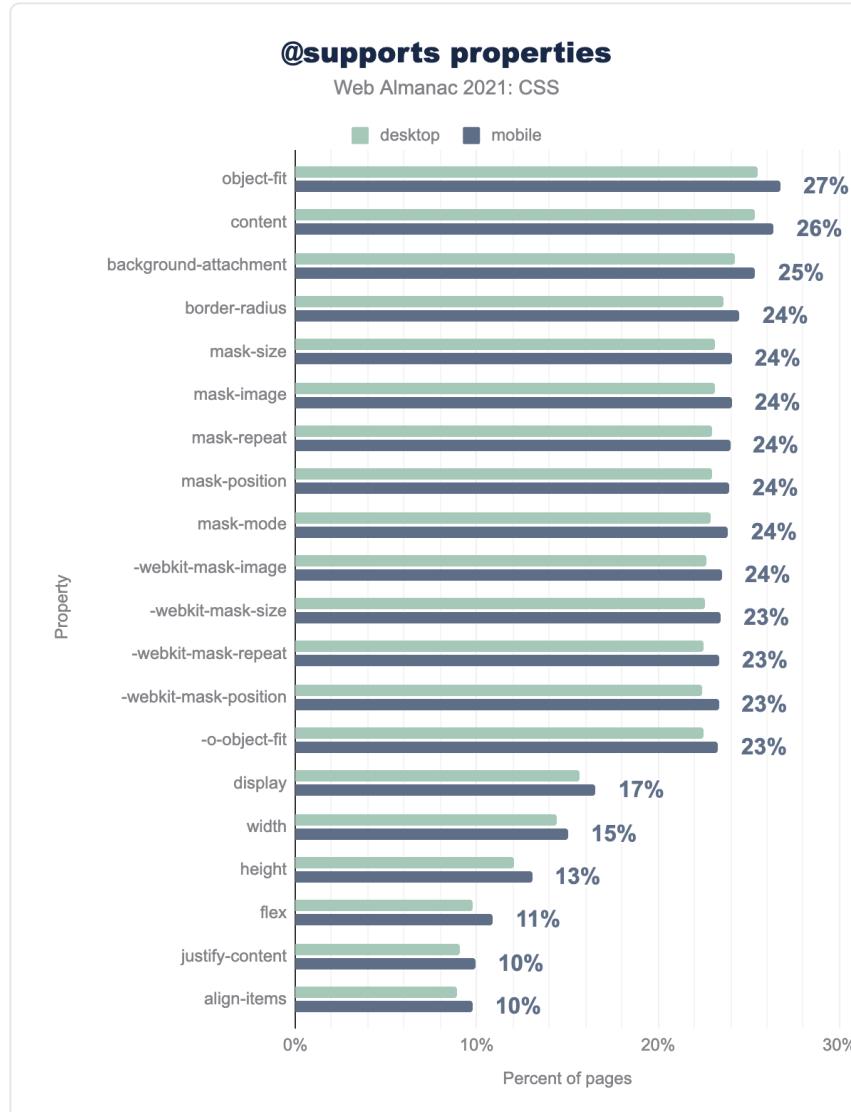


Figure 1.61. Properties used inside feature query blocks by percent of pages.

Having tested for support, which properties are then used inside these feature query blocks? The `object-fit` property came out top, the `mask-*` properties making a good showing, along with their `-webkit-mask-*` counterparts. This is likely due to the lack of interoperability for masking until recently, with the properties still requiring a `-webkit` prefix for Chrome.

While the `display` property features in the top 20, you have to go a long way down the list to find any grid properties. The `grid-template-columns` property being found in 2% of feature query blocks.

Internationalization

English is described as a horizontal top to bottom language, because sentences are written horizontally, starting at the top of the page. The script direction runs left-to-right (LTR). Arabic, Hebrew, and Urdu are also horizontal top to bottom languages, but have a script direction of right-to-left (RTL). There are also languages that are written vertically, from top to bottom, such as Chinese, Japanese, and Mongolian. CSS has evolved to better cope with these different writing modes and script directions.

Direction

The number of pages using the `direction` property to set CSS either on the `<body>` or `<html>` element remained unchanged from 2021, with 11% of pages setting it on `<html>` and 3% on `<body>`. It's recommended to use HTML¹⁰, rather than CSS to set `direction`, so a lower number here matches that best practice.

Logical and physical properties

Logical or flow-relative properties such as `border-block-start` and values such as `start` for `text-align` are useful for internationalization, as they follow the flow of text rather than being tied to the physical dimensions of the screen. Browser support for these properties is now excellent, so we wondered whether we would see more adoption.

10. <https://www.w3.org/International/questions/qa-html-dir>

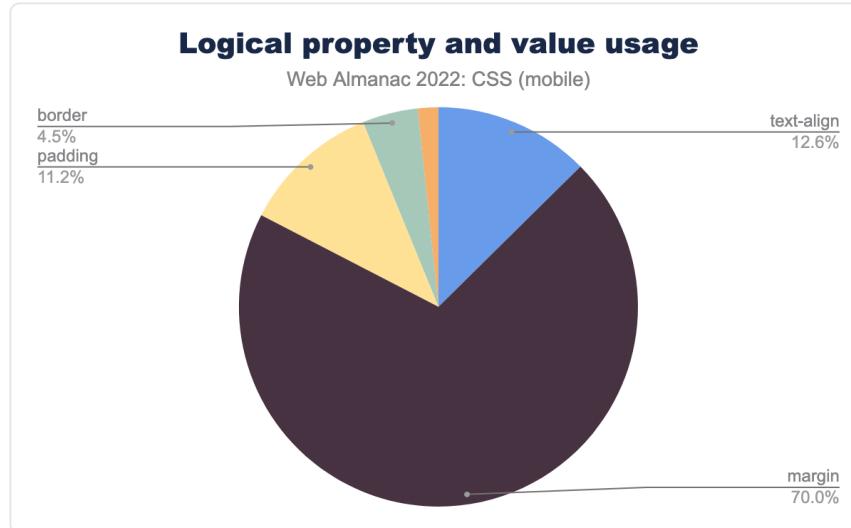


Figure 1.62. The distribution of logical properties used.

Logical property usage has increased slightly from 2021, up from 4% to 5%. However, the chart for 2022 looks very different to the one for 2021. Overwhelmingly, people are using logical properties to set `margin` properties, up to 70% from 26%. The most popular `margin` properties are `margin-inline-start` and `margin-inline-end`, found on 9% of total pages. These are particularly useful for making sure that spacing between a label and following field, for example, works in the same way in a LTR and RTL script.

Ruby

Once again we checked for usage of CSS Ruby¹¹, a collection of properties used for interlinear annotation, which are short runs of text alongside the base text.

0.2%

Figure 1.63. The percentage of mobile pages using CSS Ruby.

Its usage is still tiny, but has increased from 2021. In only 8,157 desktop pages and 9,119 mobile pages were found to be using it—less than 0.1% of all pages analyzed. This year, 16,698 desktop and 21,266 mobile pages—or 0.2% of all pages analyzed—were using it.

11. https://developer.mozilla.org/docs/Web/CSS/CSS_Ruby

CSS in JS

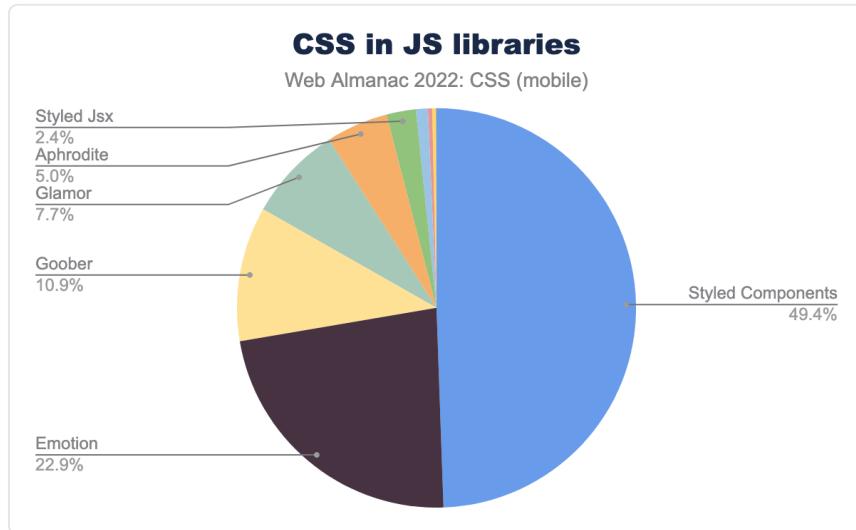


Figure 1.64. Usage of CSS in JS libraries.

The use of CSS-in-JS has not increased from last year, staying at 3%. This usage is almost all from libraries, the most popular of which is Styled Components. This library has dropped in share from 57% to 49%, with a new library entering the mix at almost 11%. Goober¹² describes itself as “a less than 1KB css-in-js solution”, and is certainly making some inroads among people who like this type of thing.

Houdini

There is still very little usage of Houdini¹³ on the open web. Looking at the number of pages using animated custom properties shows only a small increase since 2021. We also looked at usage of the Houdini Paint API. We do find instances of this in use on the web. By looking at the names of worklets used, much of this is the smooth corners¹⁴ worklet, indicating that people are using it as a progressive enhancement, given that this can fall back nicely to a regular `border-radius`.

12. <https://goober.js.org/>

13. https://developer.mozilla.org/docs/Web/CSS/CSS_Houdini

14. <https://css-houdini.rocks/smooth-corners/>

Sass

Preprocessors like Sass can be seen as a good indicator of what developers want to be able to do with CSS, but can't. And, with CSS increasing in power, a common question from developers is whether we need to use Sass at all. We can see from the rise in custom properties usage, that one common preprocessor use—the ability to have variables or constants—now has a built-in CSS equivalent.

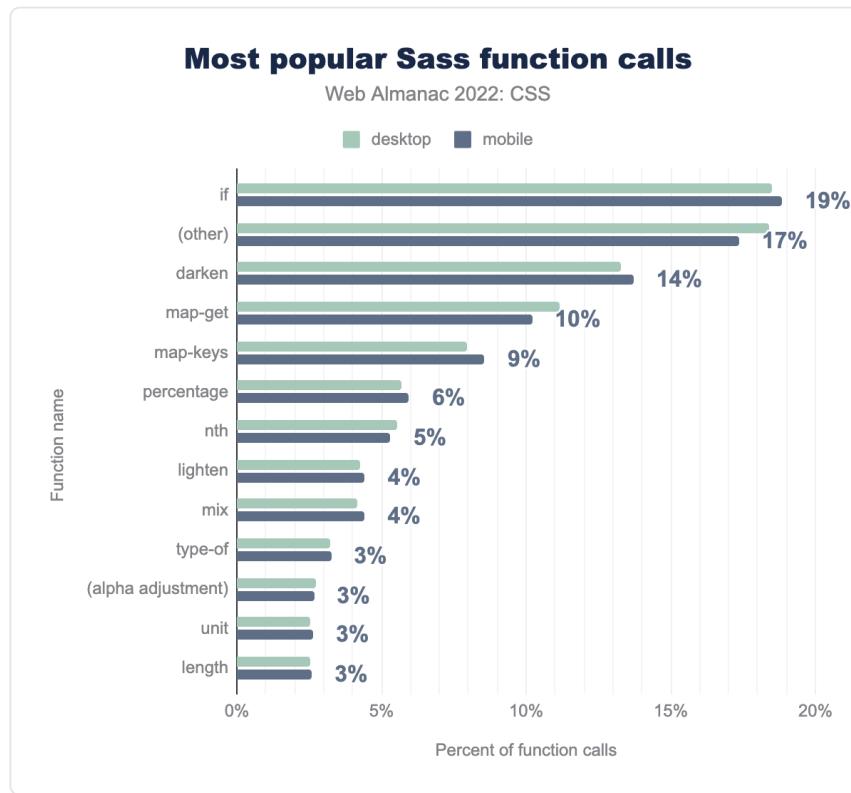


Figure 1.65. Most popular Sass function calls by percent of calls.

Looking at the function calls shows that color functions are still a very popular use of Sass, something that may well soon be replaced with new CSS color functions¹⁵ such as `color-mix()`. There are some changes from last year. The `darken` function has dropped 2 percentage points to 14% and third place. The `lighten` function has, however, gained a point.

15. <https://www.w3.org/TR/css-color-5/>

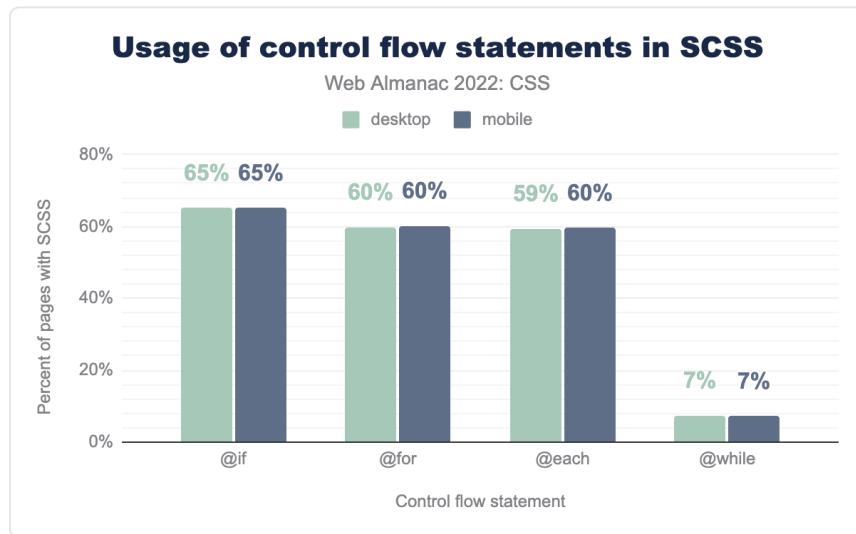


Figure 1.66. Distribution of control flow statements on SCSS.

Looking at control flow statements we see a small increase in `@for` and `@each`, however `@while` has increased from 2% to 7%.

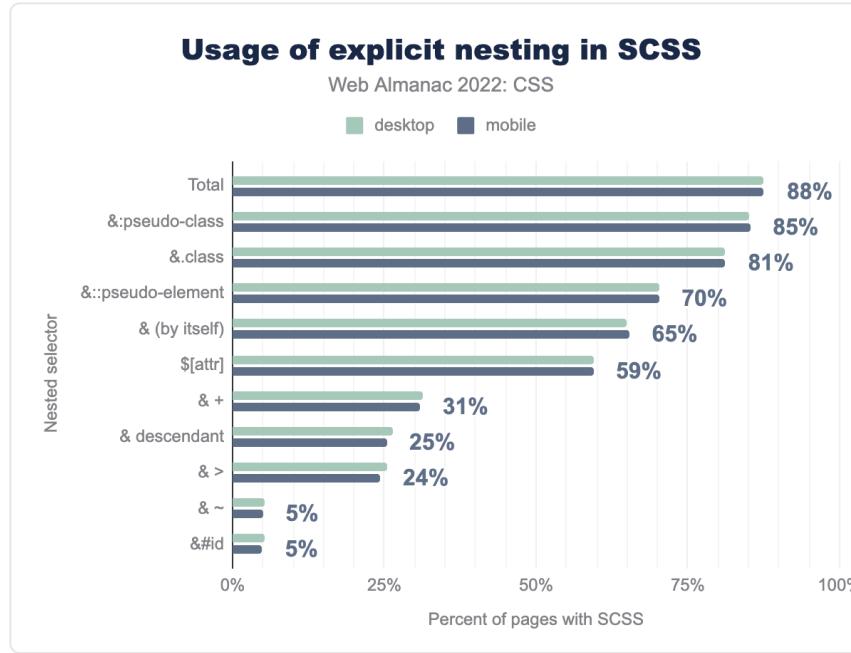


Figure 1.67. Use of explicit nesting in SCSS by percent of pages using SCSS.

Nesting is also interesting, given that a future spec for CSS Nesting is currently in development and discussion at the CSS Working Group. Nesting in SCSS sheets is very common, and can be identified by looking for the `&` character. As with last year pseudo-classes such as `:hover`, and classes such as `.active` make up most cases of nesting. All usage increased slightly, however `& descendant` increased 7 percentage points from 18% to 25%. Implicit nesting is not measured in this survey, as it does not use special characters.

CSS for print

5%

Figure 1.68. The percentage of desktop pages with print-specific styles.

We wondered whether developers were creating print stylesheets to provide a better printed experience, and only 5% of desktop and 4% of mobile sites were doing so.

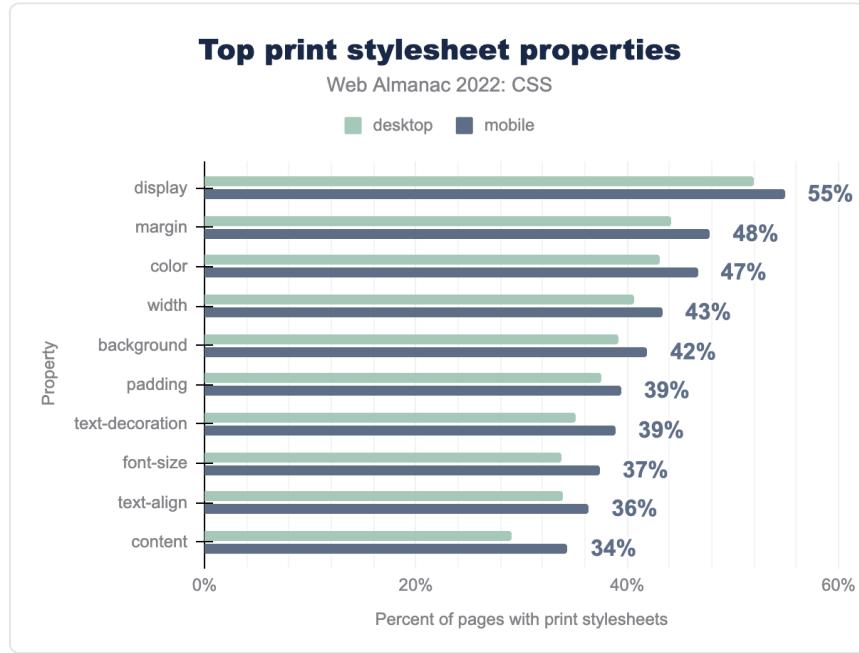


Figure 1.69. The top properties found in print styles on pages that have a print stylesheet.

Of the pages using print styles, over half changed the value of `display`—perhaps to simplify a grid or flex layout for print. We also see people changing colors, tweaking margin and padding, and setting the `font-size`. At 34% is the `content` property, used to insert generated content.

Print is a fragmented medium; the content is fragmented into pages, and we have a set of fragmentation properties that aim to give some control over how these breaks happen. For example, developers usually want to avoid a heading being the last thing on a page, or a caption being disconnected from the figure it relates to.

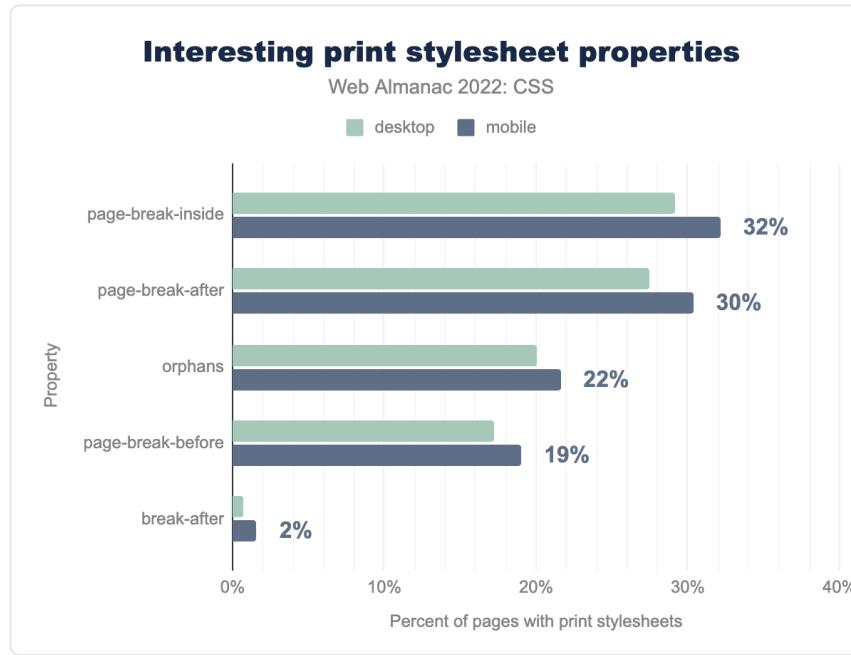


Figure 1.70. Fragmentation properties used in print stylesheets.

We see in this chart that many developers are using the old fragmentation properties of `page-break-inside`, `page-break-after`, and `page-break-before`, rather than the new properties such as `break-before`, which has very low usage.

The `orphans` property appears in 22% of print stylesheets, despite lacking support in Firefox. This property defines the number of lines that should be left at the bottom of a page before a fragmentation break. The `widows` property (which sets the number of lines on their own after a fragmentation break) is seen with around the same frequency. It is likely that people are setting the same value for both.

Paged media

There is an entire specification for dealing with Paged Media, and CSS for print. However, this has been poorly implemented in browsers. To find a good implementation of these features you need to use a print-specific user agent.

There is some browser support for the `@page` rule and its pseudo-classes, and we did find developers using these to set different page properties for the first page, and the left and right pages of a spread.

Pseudo-class	Desktop	Mobile
<code>:first</code>	5,950	7,352
<code>:right</code>	1,548	2,115
<code>:left</code>	1,554	2,101

Figure 1.71. Number of pages found using `@page` spread pseudo-classes.

Of people using these pseudo-classes, usage was mostly to set the page margins, and also the size of the page.

Meta

This section rounds up some general information about CSS usage, for example how often declarations are repeated, and common mistakes in CSS.

Declaration repetition

In 2020 and 2021, analysis was done to determine the amount of “declaration repetition”. This aims to identify how efficient a stylesheet is by looking for the number of declarations using the same property and value.

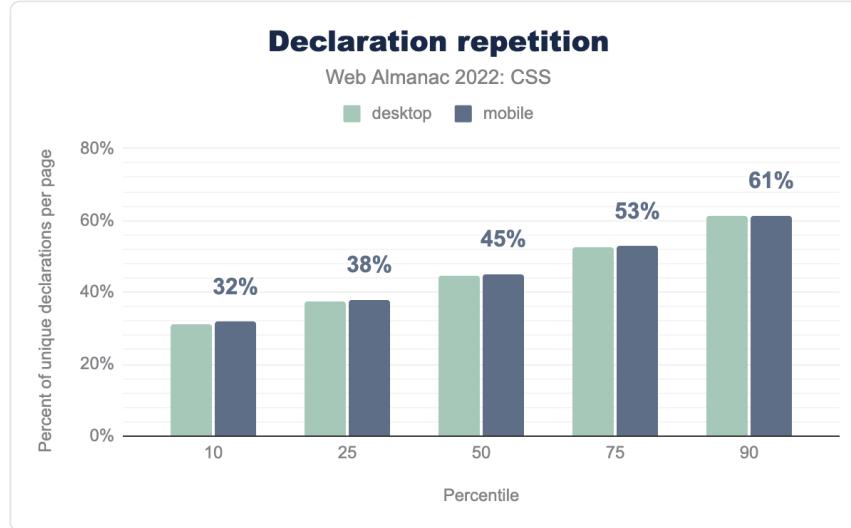


Figure 1.72. Distribution of repetition.

In 2021 it was reported there was a slight drop in repetition, this year there is a slight rise. This metric does therefore seem fairly stable year-on-year.

Shorthands and longhands

In CSS, a shorthand property is one that can set a number of longhand properties in one declaration. For example, the shorthand property `background` can be used to set all of the following longhand properties:

- `background-attachment`
- `background-clip`
- `background-color`
- `background-image`
- `background-origin`
- `background-position`
- `background-repeat`
- `background-size`

When developers mix shorthand properties like `background` and longhand properties like `background-size` in a stylesheet, it is always best to have the longhands come after the shorthands. We looked at instances of this to see which longhands were most common.

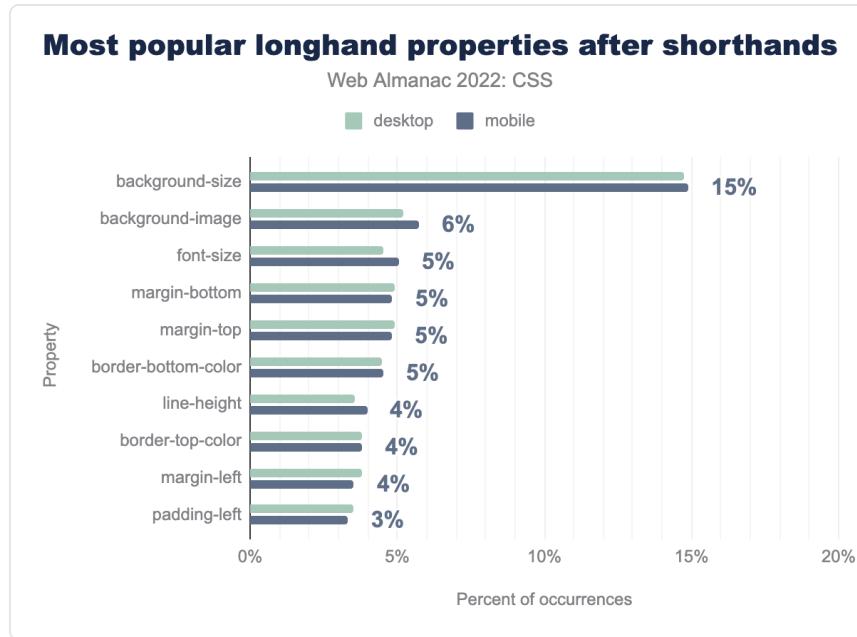


Figure 1.73. The most popular longhand properties that come after shorthands.

As in 2020 and 2021, `background-size` came out top of the chart, and there was little difference to be seen from 2021.

Unrecoverable syntax errors

As in previous years, we use the Rework¹⁶ engine for CSS parsing. An unrecoverable error is one where the error is so bad, the full stylesheet is unable to be parsed by Rework. Last year, 0.94% of desktop pages, and 0.55% of mobile pages contained an unrecoverable error. This year, 13% of desktop and 12% of mobile pages had such an error. This seems like a large jump, however due to some changes in methodology (adding size thresholds) it is likely that not all of the instances are unrecoverable errors.

16. <https://github.com/reworkcss/css>

Nonexistent properties

As in previous years we checked for declarations that had valid syntax, but referred to properties that don't actually exist. This includes spelling errors, malformed vendor prefixes, and things developers have just made up.

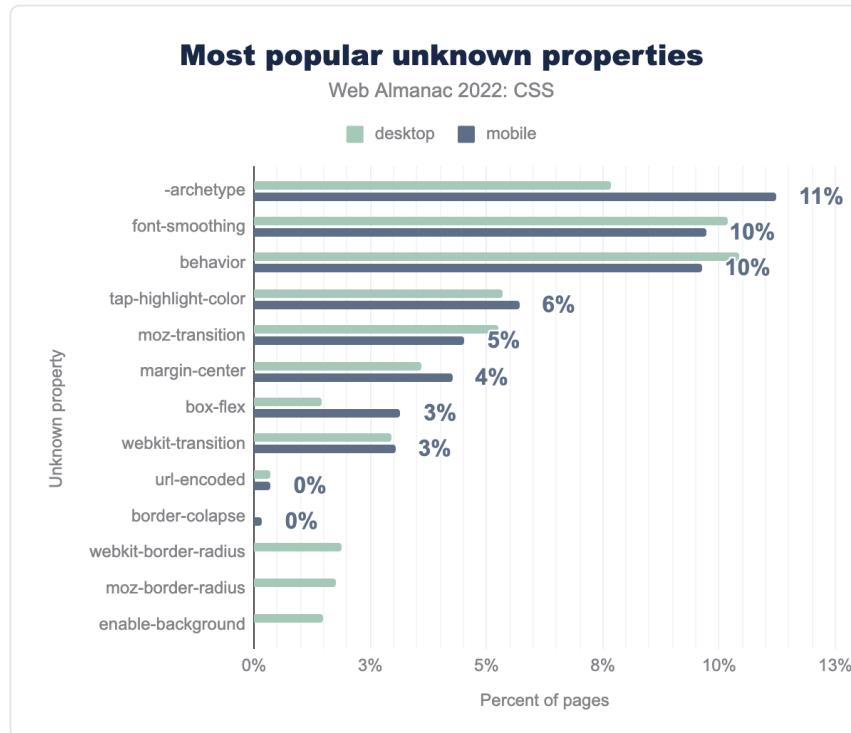


Figure 1.74. The most frequently seen unknown properties.

The top mystery property is `-archetype`, which is now appearing in 11% of cases of stylesheets with nonexistent properties. This property has jumped from 4% last year to 11% to take the top spot. The second property is `font-smoothing` with a drop of 4% points from last year. This appears to be an unprefixed version of `-webkit-font-smoothing` that does not actually exist. The use of the malformed `webkit-transition` (which should be `-webkit-transition`) has dropped from 14% to 3%. This makes us think it was perhaps getting into a large number of stylesheets via a framework or other third party that has since been updated to fix the problem.

Conclusion

CSS continues to evolve at a rapid pace, however we can see from the data that new features are adopted quite slowly, even when they have been in all major engines for several years. There are a few highly requested features, such as container queries, landing in browsers as of this writing. It will be interesting to see whether the uptake for these features will match the demand for them.

Something that has been apparent in this data is how much popular platforms, in particular WordPress, can impact usage statistics. We can see WordPress class and custom property names clearly in the data, but what is harder to see are the properties and values used by classes added to the majority of WordPress sites. If WordPress adopts a new feature, as part of one of these standard classes, we should expect to see a sudden uptick in usage.

As noted in last year's conclusion, the data tells a story of gradual, steady adoption of new features (such as grid layout) or best practices (such as using logical rather than physical properties). We look forward to seeing how these changes develop in the years to come.

Author

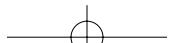
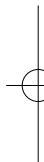
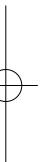


Rachel Andrew

 @rachelandrew  rachelandrew  <https://rachelandrew.co.uk>

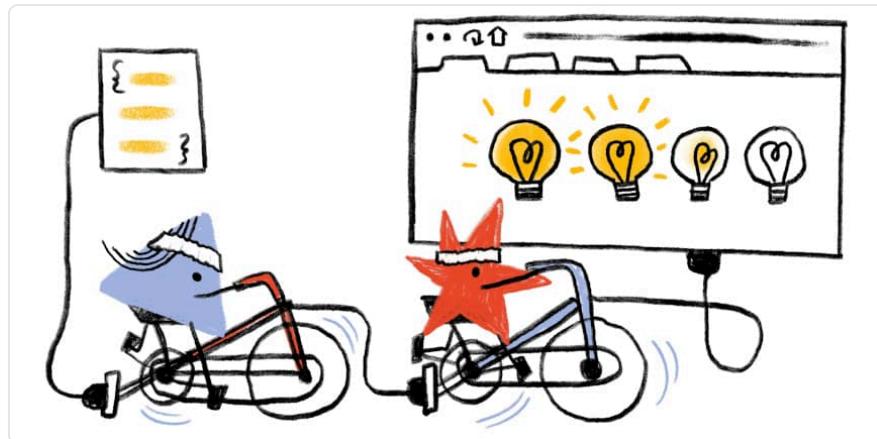
Rachel Andrew works for Google as a technical writer, working on web.dev¹⁷ and the Chrome Developers site¹⁸. She is a front and back-end web developer, author and speaker, author or co-author of 22 books including *The New CSS Layout*¹⁹ and a regular contributor to a number of publications both on and offline. Rachel is a Member of the CSS Working Group, and can be found posting photos of her cats on Twitter as @rachelandrew.

17. <https://web.dev>
18. <https://developer.chrome.com/>
19. <https://abookapart.com/products/the-new-css-layout>



Part I Chapter 2

JavaScript



Written by Jeremy Wagner

Reviewed by Minko Gechev, Pankaj Parkar, Nishu Goel, Houssein Djirdeh, Kevin Farrugia, and Barry Pollard

Analyzed by Nishu Goel and Kevin Farrugia

Edited by Abel Mathew and Rick Viscomi

Introduction

JavaScript is a powerful force that provides the lion's share of interactivity on the web. It drives behaviors from the simple to the complex, and is making more things possible on the web than ever before.

Yet, the increased usage of JavaScript to deliver rich user experiences comes at a cost. From the moment JavaScript is downloaded, parsed, and compiled, to every line of code it executes, the browser must orchestrate all kinds of work to make everything possible. Doing too little with JavaScript means you might fall short of fulfilling user experience and business goals. On the other hand, shipping too much on JavaScript means you will be creating user experiences that are slow to load, sluggish to respond, and frustrating to users.

This year, we'll once again be looking at the role of JavaScript on the web, as we present our findings for 2022 and offering advice for creating delightful user experiences.

How much JavaScript do we load?

To begin, we'll assess the amount of JavaScript web developers ship on the web. After all, before improvements can be made, an assessment of the current landscape must be performed.

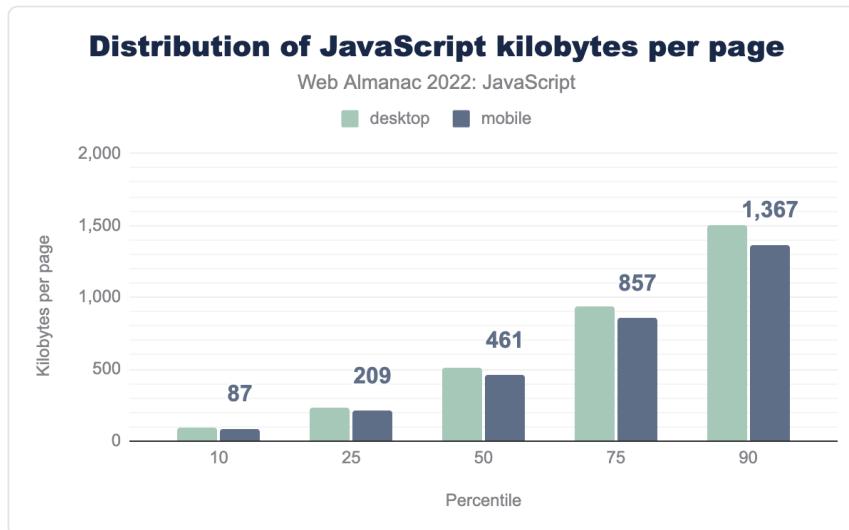


Figure 2.1. Distribution of the amount of JavaScript loaded per page.

As was the case last year, this year marks yet another increase in the amount of JavaScript shipped to browsers. From 2021²⁰ to 2022, an increase of 8% for mobile devices was observed, whereas desktop devices saw an increase of 10%. While this increase is less steep than in previous years, it's nonetheless the continuation of a concerning trend. While device capabilities continue to improve, not everyone is running the latest device. The fact remains that more JavaScript equates to more strain on a device's resources.

20. <https://almanac.httparchive.org/en/2021/javascript#how-much-javascript-do-we-load>

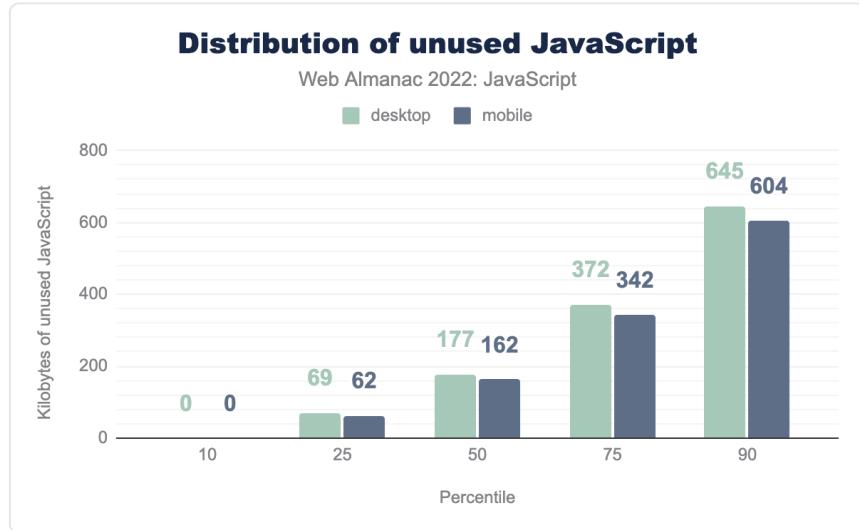


Figure 2.2. Distribution of the amount of unused JavaScript bytes.

According to Lighthouse²¹, the median mobile page loads 162 KB of unused JavaScript. At the 90th percentile, 604 KB of JavaScript are unused. This is a slight uptick from last year, where the median and 90th percentile of unused JavaScript was 155 KB and 598 KB, respectively. All of this represents a very large amount of unused JavaScript, especially when you consider that this analysis tracks the transfer size of JavaScript resources which, if compressed, means that the decompressed portion of used JavaScript may be a lot larger than the chart suggests.

When contrasted with the total number of bytes loaded for mobile pages at the median, unused JavaScript accounts for 35% of all loaded scripts. This is down slightly from last year's figure of 36%, but is still a significantly large chunk of bytes loaded that go unused. This suggests that many pages are loading scripts that may not be used on the current page, or are triggered by interactions later on in the page lifecycle, and may benefit from dynamic `import()` to reduce startup costs.

JavaScript requests per page

Every resource on a page will kick off at least one request, and possibly more if a resource makes additional requests for more resources.

Where script requests are concerned, the more there are, the more likely you'll not just load

21. <https://web.dev/unused-javascript/>

more JavaScript, but also increase contention between script resources that may bog down the main thread, leading to slower startup.

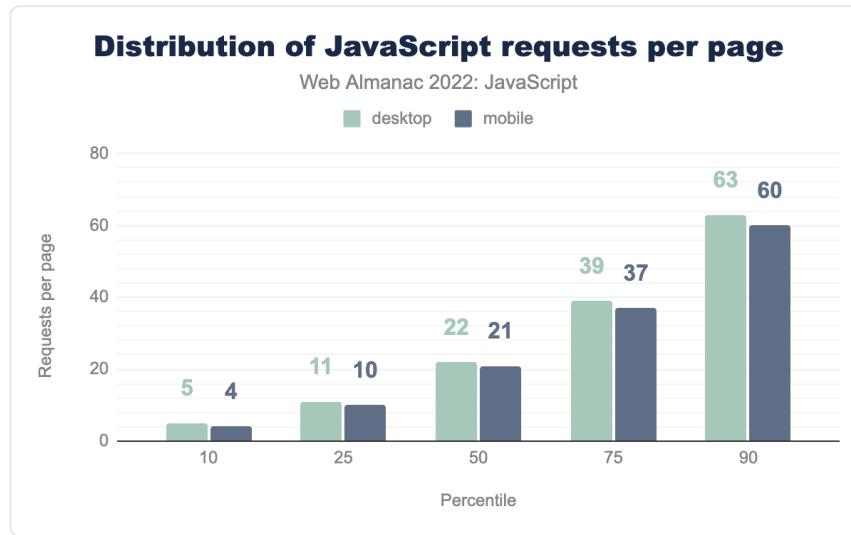


Figure 2.3. Distribution of the number of JavaScript requests per page.

In 2022, the median mobile page responded to 21 JavaScript requests, whereas at the 90th percentile, there were 60. Compared to last year, this is an increase of 1 request at the median and 4 requests at the 90th percentile.

Where desktop devices in 2022 are concerned, there are 22 JavaScript requests at the median, and 63 at the 90th percentile. Compared to last year, this is an increase of 1 JavaScript request at the median, and 4 at the 90th percentile—the same increase as noted for mobile devices.

While not a large increase in the number of requests, it does continue the trend of increased requests year over year since the Web Almanac's inception in 2019.

How is JavaScript processed?

Since the advent of JavaScript runtimes such as Node.js, it has become increasingly common to rely on build tools in order to bundle and transform JavaScript. These tools, while undeniably useful, can have effects on how much JavaScript is shipped. New to the Web Almanac this year, we're presenting data on the usage of bundlers and transpilers.

Bundlers

JavaScript bundlers are build-time tools that process a project's JavaScript source code and then apply transformations and optimizations to it. The output is production-ready JavaScript. Take the following code as an example:

```
function sum (a, b) {  
    return a + b;  
}
```

A bundler will transform this code to a smaller, but more optimized equivalent that takes less time for the browser to download:

```
function n(n,r){return n+r}
```

Given the optimizations bundlers perform, they are a crucial part of optimizing source code for better performance in production environments.

There are a wealth of choices when it comes to JavaScript bundlers, but one that pops into mind often is webpack²². Fortunately, webpack's generated JavaScript contains a number of signatures (`webpackJsonp`, for example) that make it possible to detect if a website's production JavaScript has been bundled using webpack.

22. <https://webpack.js.org/>

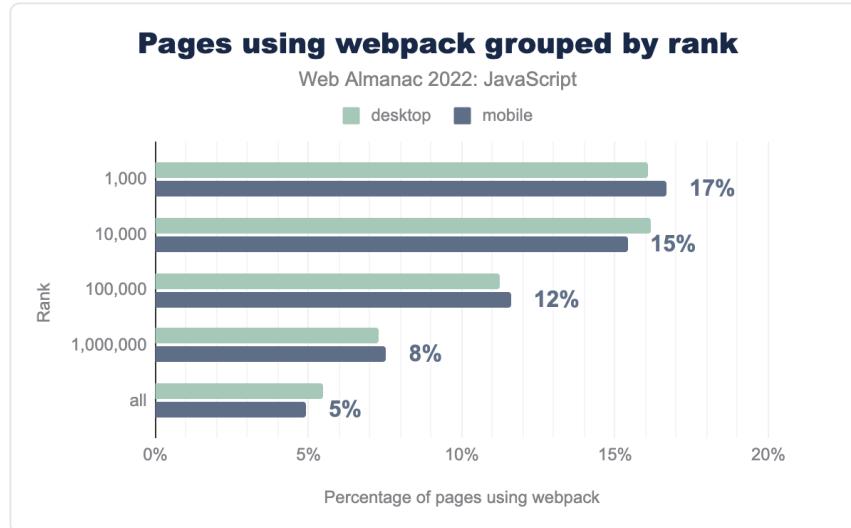


Figure 2.4. Pages that use webpack-bundled JavaScript by rank.

Of the 1,000 most popular websites, 17% use webpack as a bundler. This makes sense, as many of the top pages HTTP Archive crawls are likely to be high-profile ecommerce sites that use webpack to bundle and optimize source code. Even so, the fact that 5% of the all pages in the HTTP Archive dataset use webpack is a significant statistic. However, webpack isn't the only bundler in use.

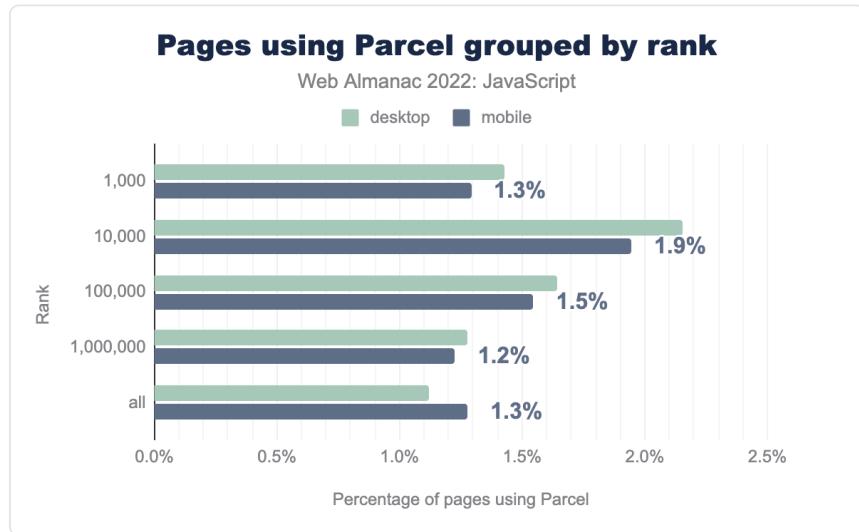


Figure 2.5. Pages that use Parcel-bundled JavaScript by rank.

Parcel²³ is a noteworthy alternative to webpack, and its adoption is significant. Parcel's adoption is consistent across all ranks, accounting for a range of 1.2% to 1.9% across rankings.

While HTTP Archive is unable to track the usage of all bundlers in the ecosystem, bundler usage is significant in the overall picture of JavaScript in that they're not only important to the developer experience, but the overhead they can contribute in the form of dependency management code can be a factor in how much JavaScript is shipped. It's worth checking how your overall project settings are configured to produce the most efficient possible output for the browsers your users use.

Transpilers

Transpilers are often used in toolchains at build-time to transform newer JavaScript features into a syntax that can be run in older browsers. Because JavaScript has evolved rapidly over the years, these tools are still in use. New to this year's Web Almanac is an analysis of the usage of Babel²⁴ in delivering widely compatible, production-ready JavaScript. The singular focus on Babel specifically is due to its wide usage in the developer community over alternatives.

23. <https://parceljs.org/>

24. <https://babeljs.io/>

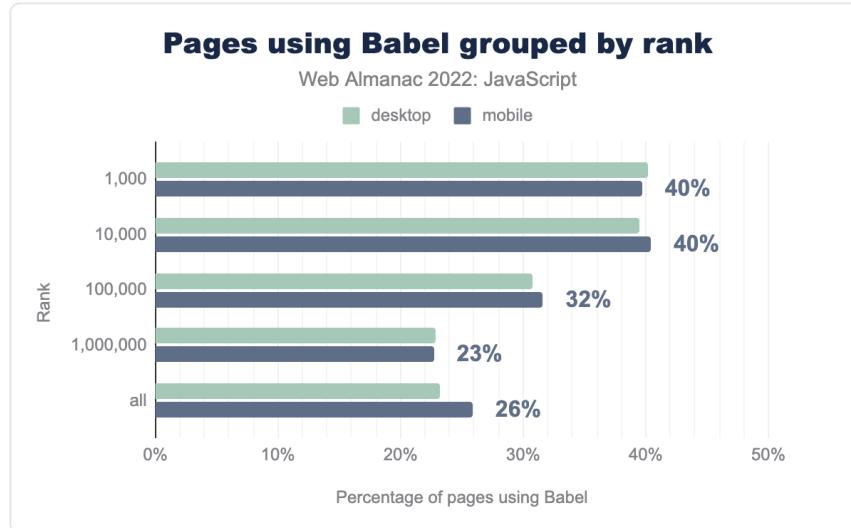


Figure 2.6. Pages that use Babel by rank.

These results are not a surprising development when you consider how much JavaScript has evolved over the years. In order to maintain broad compatibility for a certain set of browsers, Babel uses transforms²⁵ to output compatible JavaScript code.

Transforms are often larger than their untransformed counterparts. When transforms are extensive or duplicated across a codebase, potentially unnecessary or even unused JavaScript may be shipped to users. This can adversely affect performance.

Considering that even 26% of pages ranked in the top million are transforming their JavaScript source code using Babel, it's not unreasonable to assume that some of these experiences may be shipping transforms they don't need. If you use Babel in your projects, carefully review Babel's available configuration options²⁶ and plugins to find opportunities to optimize its output.

Since Babel also relies on Browserslist²⁷ to figure out whether it needs to transform certain features to a legacy syntax, be sure to also review your Browserslist configuration to ensure that your code is transformed to work in the browsers your users actually use.

How is JavaScript requested?

The manner in which JavaScript is requested may also have performance implications. There

25. <https://babeljs.io/docs/en/babel-plugin-transform-runtime#why>

26. <https://babeljs.io/docs/en/options>

27. <https://github.com/browserslist/browserslist>

are optimal ways you can request JavaScript, and in some cases, there are far less optimal methods. Here, we'll see how the web is shipping JavaScript overall, and how that aligns with performance expectations.

`async`, `defer`, `module`, and `nomodule`

The `async` and `defer` attributes on the HTML `<script>` element control the behavior of how scripts load. The `async` attribute will prevent scripts from blocking parsing, but will execute as soon as they are downloaded, so may still block rendering. The `defer` attribute will delay execution of scripts until the DOM is ready so should prevent those scripts from blocking both parsing and rendering.

The `type="module"` and `nomodule` attributes are specific to the presence (or absence) of ES6 modules being shipped to the browser. When `type="module"` is used, the browser expects that the content of those scripts will contain ES6 modules, and will defer the execution of those scripts until the DOM is constructed by default. The opposite `nomodule` attribute indicates to the browser that the current script does not use ES6 modules.

Feature	Desktop	Mobile
<code>async</code>	76%	76%
<code>defer</code>	42%	42%
<code>async</code> and <code>defer</code>	28%	29%
<code>module</code>	4%	4%
<code>nomodule</code>	0%	0%

Figure 2.7. Percentage of pages using `async`, `defer`, `async` and `defer`, `type="module"`, and `nomodule` attributes on `<script>` elements.

It's encouraging that 76% of mobile pages load scripts with `async`, as that suggests developers are cognizant of the effects of render blocking. However, such a low usage of `defer` suggests that there are opportunities being left on the table to improve rendering performance.

As noted last year²⁸, using both `async` and `defer` is an antipattern that should be avoided as the `defer` part is ignored and `async` takes precedence.

The general absence of `type="module"` and `nomodule` is not surprising, as few pages seem

28. <https://almanac.httparchive.org/en/2021/javascript#async-and-defer>

to be shipping JavaScript modules. As time goes on, the usage of `type="module"` in particular may increase, as developers ship untransformed JavaScript modules to the browser.

Looking at the percentage of overall scripts across all the sites, presents a slightly different view:

Feature	Desktop	Mobile
<code>async</code>	49.3%	47.2%
<code>defer</code>	8.8%	9.1%
<code>async</code> and <code>defer</code>	3.0%	3.1%
<code>module</code>	0.4%	0.4%
<code>nomodule</code>	0%	0%

Figure 2.8. Percentage of scripts using `async`, `defer`, `async` and `defer`, `type="module"`, and `nomodule` attributes on `<script>` elements.

Here we see a much smaller use of both `async` and `defer`. Some of these scripts may be being inserted dynamically after the initial rendering, but it's also likely a good proportion of pages are not setting these attributes on a lot of their scripts that are included in the initial HTML and so are delaying rendering.

`preload`, `prefetch`, and `modulepreload`

Resource hints such as `preload`, `prefetch`, and `modulepreload` are useful in hinting to the browser which resources should be fetched early. Each hint has a different purpose, with `preload` used to fetch resources needed for the current navigation, `modulepreload` the equivalent for preloading scripts that contain JavaScript modules²⁹, and `prefetch` used for resources needed in the next navigation.

29. <https://developer.mozilla.org/docs/Web/JavaScript/Guide/Modules>

Resource hint	Desktop	Mobile
<code>preload</code>	16.4%	15.4%
<code>prefetch</code>	1.0%	0.8%
<code>modulepreload</code>	0.1%	0.1%

Figure 2.9. Percentage of pages using various resource hints.

Analyzing the trend of resource hint adoption is tricky. Not all pages benefit from them, and it's unwise to make a blanket recommendation to use resource hints broadly, as their overuse has their own consequences—especially where `preload` is concerned. However, the relative abundance of `preload` hints on 15% of mobile pages suggests that many developers are aware of this performance optimization, and are trying to use it to their advantage.

`prefetch` is tricky to use, though it can be beneficial for long, multi-page sessions. Even so, `prefetch` is entirely speculative, so much so that browsers may ignore it in certain conditions. This means some pages may waste data by requesting resources which go unused. It really “just depends”.

The lack of use of `modulepreload` makes sense, since adoption of the `type="module"` attribute on `<script>` elements is similarly low. Even so, apps that ship JavaScript modules without transformations could benefit from this resource hint, as it will not fetch just the named resource, but the entire module tree. This could help in certain situations.

Let's dig into an analysis of how many of each resource hint type is used.

Distribution of prefetch adoption for JavaScript resources per page

Web Almanac 2022: JavaScript

desktop mobile

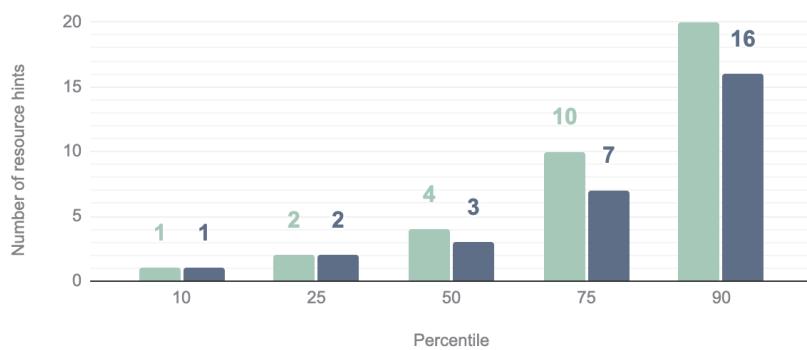


Figure 2.10. Distribution of `prefetch` adoption for JavaScript resources per page.

Adoption of `prefetch` here is somewhat surprising, with three `prefetch` hints for JavaScript resources per page. However, the number of these hints at the 75th and 90th percentiles suggests that there may be a fair amount of waste in the form of unused resources for page navigations that never occur.

Distribution of preload adoption for JavaScript resources per page

Web Almanac 2022: JavaScript

desktop mobile

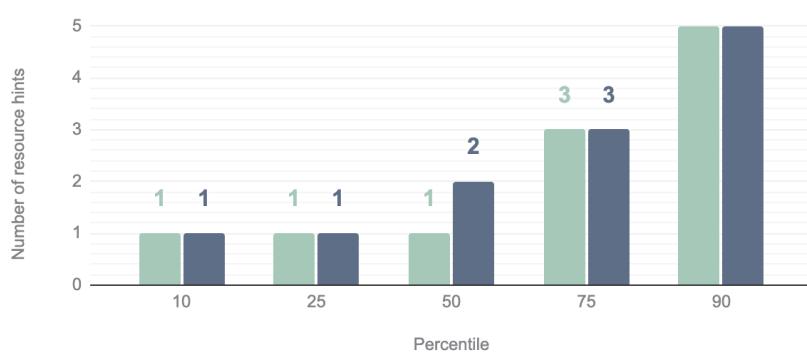


Figure 2.11. Distribution of `preload` adoption for JavaScript resources per page.

Remember—this analysis tracks how many resource hints are used for JavaScript resources on pages that use one or more `preload` hints. The median page is delivering two `preload` hints for JavaScript, which isn't bad on its face, but it often depends on the size of the script, how much processing scripts can kick off, or whether the script fetched via `preload` is even needed for the initial page load.

Unfortunately, we see five `preload` hints for JavaScript resources at the 90th percentile, which may be too much. This suggests that pages at the 90th percentile are especially reliant on JavaScript, and are using `preload` to try and overcome the performance issues that result.

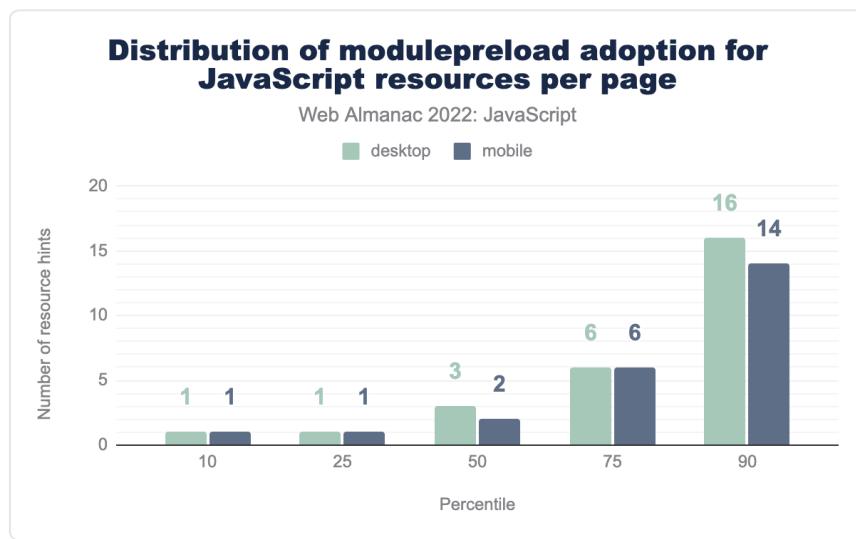


Figure 2.12. Distribution of `modulepreload` adoption for JavaScript resources per page.

With `modulepreload`, we see a staggering 6 hints at the 75th percentile, and 14 at the 90th percentile! This suggests that, while pages using one or more `modulepreload` hints at upper percentiles are shipping untransformed ES6 modules directly to the browser, the need for so many resource hints suggests an overreliance on JavaScript at the upper range.

Resource hints are great tools for optimizing how we load resources in the browser, but if there's one piece of advice you can heed when using them, it's to use them sparingly, and for resources that may not be initially discoverable; for example, a JavaScript file initially loaded in the DOM that requests another one. Rather than preloading loads of scripts, try to whittle down the amount of JavaScript you're shipping, as that will lead to a better user experience rather than preloading gobs of it.

JavaScript in the `<head>`

An old and often-touted best practice for performance has been to load your JavaScript in the footer of the document to avoid render blocking of scripts and to ensure the DOM is constructed before your scripts have a chance to run. In recent times, however, it has been more commonplace in certain architectures to place `<script>` elements in the document `<head>`.

This can be a good way to prioritize the loading of JavaScript in web applications, but `async` and `defer` attributes should be used where possible to avoid render blocking of the DOM. Render blocking is when the browser must halt all rendering of the page in order to process a resource that the page depends on. This is done to avoid unpleasant effects such as the flash of unstyled content³⁰, or JavaScript runtime errors that can occur when the DOM isn't ready for a script that depends on DOM readiness.

A large, bold, dark blue percentage sign, consisting of the digits '77' followed by a percent symbol '%'. The font is a sans-serif style.

Figure 2.13. The percentage of mobile pages that have render-blocking scripts in the document `<head>`.

We found that 77% of mobile pages have at least one render-blocking script in the document `<head>`, whereas 79% of desktop pages do this. This is a concerning trend, because when scripts block rendering, page content is not painted as quickly as it could be.

30. https://en.wikipedia.org/wiki/Flash_of_unstyled_content

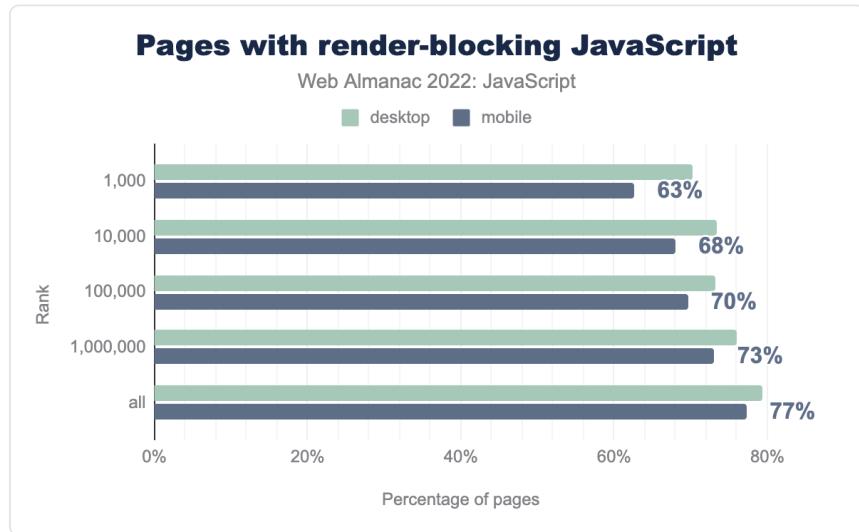


Figure 2.14. Pages by rank that have render-blocking scripts in the document `<head>`.

When looking at the problem by ranked pages, we see a similarly troubling pattern. In particular, 63% of the top 1,000 websites accessed on mobile devices ship at least one render blocking script in the `<head>`, and the proportion of pages increases as we proceed through the ranks.

There are solutions to this: using `defer` is a relatively safe bet that will unblock the DOM from rendering. Using `async` (when possible) is a good option, and will allow scripts to run immediately, but those scripts must not have any dependencies on other `<script>` elements, otherwise errors could occur.

Where possible, render-critical JavaScript can be placed in the footer and preloaded so the browser can get a head start on requesting those resources. Either way, the state of render-blocking JavaScript in tandem with how much JavaScript we ship is not good, and web developers should make more of an effort to curb these issues.

Injected scripts

Script injection is a pattern where an `HTMLScriptElement` is created in JavaScript using `document.createElement` and injected into the DOM with a DOM insertion method. Alternatively, `<script>` element markup in a string can be injected into the DOM via the `innerHTML` method.

Script injection is a fairly common practice used in a number of scenarios, but the problem with

it is that it defeats the browser's preload scanner³¹ by making the script undiscoverable as the initial HTML payload is parsed. This can affect metrics such as Largest Contentful Paint (LCP)³² if the injected script resource is ultimately responsible for rendering markup, which itself can kick off long tasks to parse large chunks of markup on the fly.

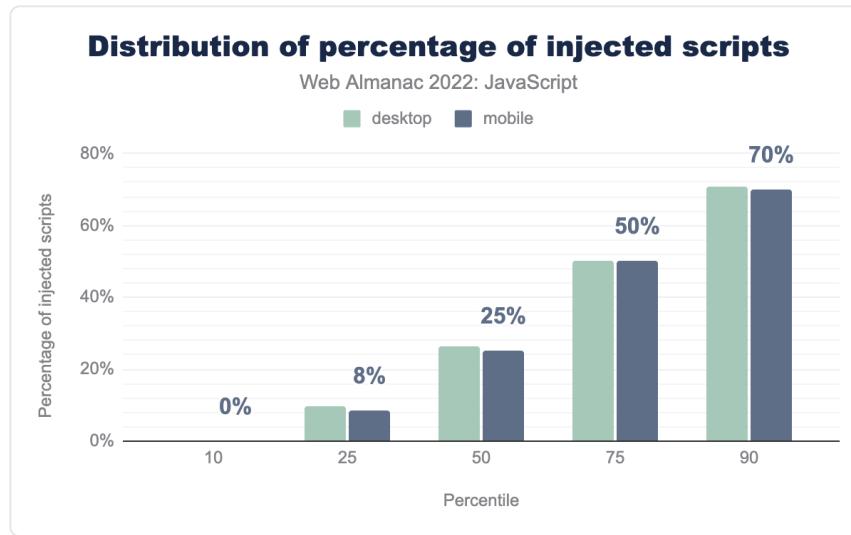


Figure 2.15. Distribution of percentage of injected scripts across various percentiles.

At the median, we see that 25% of a page's scripts are injected, as opposed to leaving them discoverable in the initial HTML response. More concerning is that the 75th and 90th percentiles of pages inject 50% and 70% of scripts respectively.

Script injection has the potential to harm performance³³ when used to render page content the user consumes, and should be avoided in these cases whenever necessary. That script injection is so prevalent in today's web is a concerning trend. Modern frameworks and tooling may rely on this pattern, which means that some out-of-the-box experiences may rely on this potential anti-pattern to provide functionality for websites.

First-party versus third-party JavaScript

There are two categories of JavaScript that websites often ship:

- First-party scripts that power the essential functions of your website and provide

31. <https://web.dev/preload-scanner/#injected-async-scripts>

32. <https://web.dev/lcp/>

33. <https://www.igvita.com/2014/05/20/script-injected-async-scripts-considered-harmful/>

interactivity.

- Third-party scripts provided by external vendors that satisfy a variety of requirements, such as UX research, analytics, providing advertising revenue, and embeds for things such as videos and social media functions.

While first-party JavaScript may be easier to optimize, third-party JavaScript can itself be a significant source of performance problems, as third-party vendors may not prioritize the optimization of their JavaScript resources over adding new features to serve additional business functions for their clients. Additionally, UX researchers, marketers, and other non-technical personnel may be hesitant to give up functionality or sources of revenue that these scripts provide.

In this section, we'll analyze the breakdown of first-party and third-party code, and comment on the current state of how websites today are divvying up how they load JavaScript—and where from.

Requests

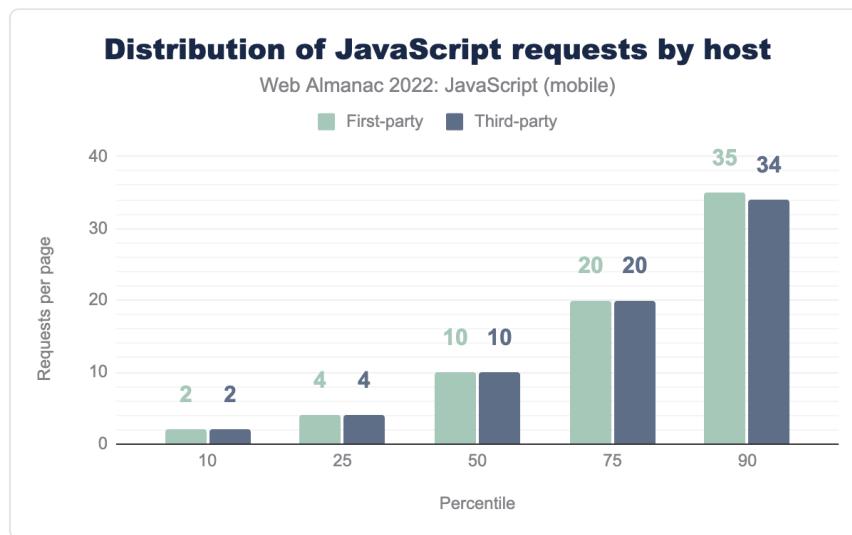


Figure 2.16. Distribution of first- versus third-party JavaScript requests by host.

Here, we see a sobering picture. Regardless of the percentile, it seems that all observed hosts are serving an equivalent amount of first and third-party scripts. The median host serves 10 of each type, the 75th percentile serves 20 of each type, and the 90th percentile host serves 34

third-party scripts!

This is a problematic and worrying trend. Third-party scripts are responsible for all sorts of damage when it comes to performance. Third-party scripts may do a number of things, such as running expensive timers that orchestrate multitudes of tasks, attach their own event listeners that add extra work which can delay interactivity, and some video and social media third-parties ship exorbitant amounts of scripts to power the services they provide.

Steps for mitigating third-party scripts is often more of a cultural affair than an engineering one. If you're shipping excessive third-party scripts, conduct an audit of each script, what they do, and profile their activity to find out what performance problems they're incurring.

If you're doing considerable UX research, consider collecting your own field data (if the origin sends a proper `Timing-Allow-Origin` header) to make informed decisions to avoid the performance problems that some third-party scripts can cause. For every third-party script you add, you're not just incurring loading costs, but also costs during runtime where responsiveness to user input is crucial.

Bytes

So we know that hosts are shipping a lot of third-party scripts, but what's the byte cost of first-versus third-party scripts?

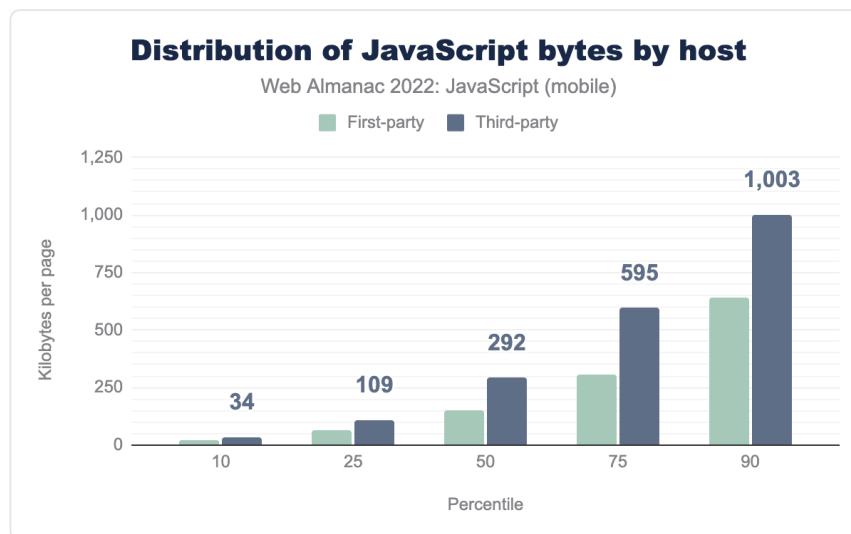


Figure 2.17. Distribution of first- versus third-party JavaScript bytes by host.

At nearly every percentile, the amount of bytes third-party scripts ship exceeds that of first-party scripts. At the 75th percentile, it appears that third-party script payloads are twice that of first-party scripts. At the 90th percentile, it appears that the amount of third-party scripts sent over the wire is nearly one megabyte.

If you find your website's first versus third-party script payloads is similar to the graph above, it is key that you should work with your engineering organization to try and get this number down. It can only help your users if you do.

Dynamic `import()`

Dynamic `import()` is a variant of the static `import` syntax that can be run anywhere in a script, whereas static `import` expressions must be run at the top of a JavaScript file and nowhere else.

Dynamic `import()` allows developers to effectively “split” off chunks of JavaScript code from their main bundles to be loaded on-demand, which can improve startup performance by loading less JavaScript upfront.



Figure 2.18. The percentage of mobile pages using dynamic `import()`.

A staggeringly low 0.34% of all observed mobile pages currently use dynamic `import()`, while 0.41% of desktop pages use it. That said, it's common for some bundlers to transform the dynamic `import()` syntax into an ES5-compatible alternative. It's very likely that the feature is in wider use, just less so in production JavaScript files.

It's tricky, but a balance can be struck, and it involves gauging the user's intent. One way of deferring loading of JavaScript without delaying interactions is to preload³⁴ that JavaScript when the user signals intent to make an interaction. One example of this could be to defer loading JavaScript for the validation of a form, and preload that JavaScript once the user has focused a field in that form. That way, when the JavaScript is requested, it will already be in the browser cache.

Another way could be to use a service worker to precache JavaScript necessary for interactions when the service worker is installed. Installation should occur at a point in which the page has fully loaded in the page's `load` event. That way, when the necessary functionality is requested,

34. https://developer.mozilla.org/docs/Web/HTML/Link_types/preload

it can be retrieved from the service worker cache without startup costs.

Dynamic `import()` is tricky to use, but more widespread adoption of it can help shift the performance cost of loading JavaScript from startup to a later point in the page lifecycle, presumably when there will be less network contention for resources. We hope to see increased adoption of dynamic `import()`, as the amount of JavaScript we see loaded during startup is only increasing.

Web workers

Web workers³⁵ are a web platform feature that reduces main thread work by spinning up a specialized JavaScript file without direct access to the DOM on its own thread. This technology can be used to offload tasks that could otherwise overwhelm the main thread by doing that work on a separate thread altogether.



Figure 2.19. The number of mobile pages using web workers.

It's heartening to see that 12% of mobile and desktop pages currently use one or more web workers to relieve the main thread of work that could potentially make the user experience worse—but there's a lot of room for improvement.

If you have significant work that can be done without direct access to the DOM, using a web worker is a good idea. While you have to use a specialized communication pipeline³⁶ to transfer data to and from a web worker, it's entirely possible to make your web pages much more responsive to user input by using the technology.

However, that communication pipeline can be tricky to set up and use, though there are open source solutions that can simplify this process. `comlink`³⁷ is one such library that helps with this, and can make the developer experience around web workers much more enjoyable.

Whether you manage web workers on your own or with a library, the point is this: if you have expensive work to do, gauge whether or not it needs to happen on the main thread, and if not, strongly consider using web workers to make the user experience of your websites as good as it possibly can be.

35. https://developer.mozilla.org/docs/Web/API/Web_Workers_API/Using_web_workers

36. https://developer.mozilla.org/docs/Web/API/Web_Workers_API/Using_web_workers#transferring_data_to_and_from_workers_further_details

37. <https://www.npmjs.com/package/comlink>

Worklets

Worklets are a specialized type of worker that allows lower-level access to rendering pipelines for tasks such as painting and audio processing. While there are four types of worklets, only two—paint worklets³⁸ and audio worklets³⁹—are currently implemented in available browsers. One distinct performance advantage of worklets is that they run on their own threads, freeing up the main thread from expensive drawing and audio processing work.

0.0013%

Figure 2.20. The percentage of mobile pages that register at least one paint worklet.

With worklets being such niche technologies, it's not surprising that they're not widely used. Paint worklets are an excellent way of offloading expensive processing for generative artwork onto another thread—not to mention a great technique for adding a bit of flair to the user experience. For every 1 million websites, only 13 of them use a paint worklet.

0.0004%

Figure 2.21. The percentage of mobile pages that register at least one audio worklet.

Adoption of audio worklets is even lower: only four in a million websites use it. It will be interesting to see how adoption of these technologies trends over time.

How is JavaScript delivered?

An equally important aspect of JavaScript performance is how we deliver scripts to the browser, which includes a few common—yet sometimes missed—opportunities for optimization, starting with how we compress JavaScript.

Compression

Compression is an often-used technique that applies largely to text-based assets, such as HTML, CSS, SVG images, and yes, JavaScript. There are a variety of compression techniques

38. https://caniuse.com/mdn-api_css_paintworklet

39. https://caniuse.com/mdn-api_audioworklet

that are widely used on the web that can speed up the delivery of scripts to the browser, effectively shortening the resource load phase.

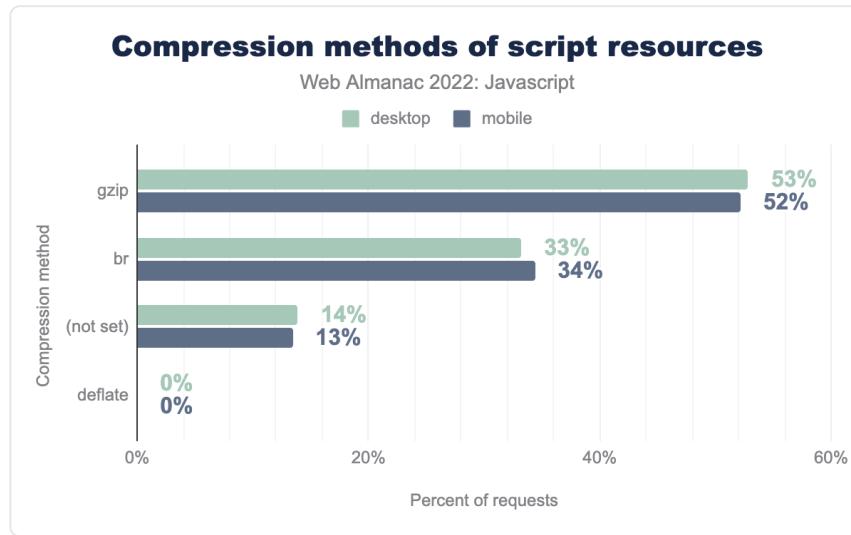


Figure 2.22. Compression of JavaScript by method.

There are a few compression techniques that can be used to reduce the transfer size of a script, with the Brotli⁴⁰ (br) method being the most effective⁴¹. Despite Brotli's excellent support in modern browsers⁴², it's still clear that gzip⁴³ is the most preferred method of compression. This is likely due to the fact that many web servers use it as the default.

When something is the default, that default sometimes remains in place rather than being tuned for better performance. Given that only 34% of pages observed are compressing scripts with Brotli, it's clear that there's an opportunity on the table to improve the loading performance of script resources, but it's also worth noting that it is an improvement over last year's adoption at 30%.

40. <https://github.com/google/brotli>
 41. <https://www.smashingmagazine.com/2016/10/next-generation-server-compression-with-brotli/>
 42. <https://caniuse.com/brotli>
 43. <https://www.gzip.org/>

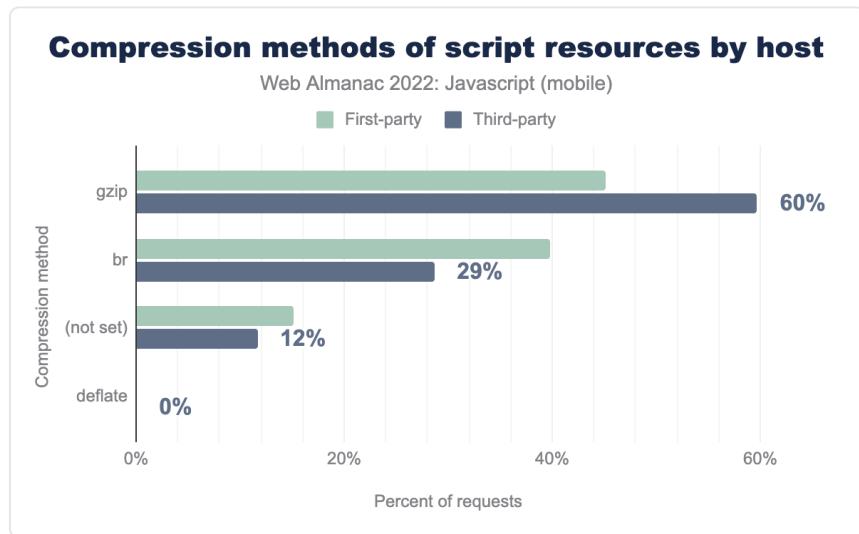


Figure 2.23. Compression methods of script resources by host.

The problem is made worse by third-party script providers, which still deploy gzip compression more widely than Brotli at 60% versus 29%, respectively. Given that third-party JavaScript is a serious performance issue on the web today, the resource load time of these resources could be reduced by deploying third-party resources using Brotli instead.

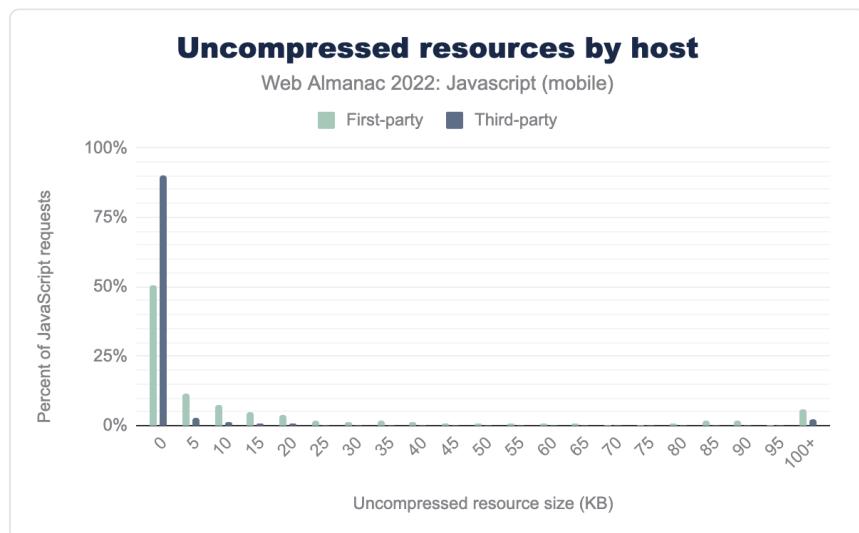


Figure 2.24. Uncompressed resources by size.

Thankfully, we're seeing that it's only mostly the smallest resources, specifically those third-party scripts that have payloads smaller than 5 KB, that are being delivered without compression. This is because compression yields diminishing returns when applied to small resources, and in fact, the added overhead of dynamic compression may cause delayed resource delivery. There are, unfortunately, some opportunities across the spectrum to compress larger resources, such as some first-party scripts with payloads over 100 KB.

Always check your compression settings to ensure you're delivering the smallest possible script payloads over the network, and remember: compression speeds up resource delivery. Those scripts, once delivered to the browser, will be decompressed and their processing time will not change due to compression. Compression is not a good excuse to deliver huge script payloads that can make interactivity worse during startup.

Minification

Minification of text assets is a time-tested practice for reducing file size. The practice involves removing all of the unnecessary spaces and comments from source code in order to reduce their transfer size. A further step known as uglification is applied to JavaScript, which reduces all of the variables, class names, and function names in a script to shorter, unreadable symbols. Lighthouse's Minify JavaScript⁴⁴ audit checks for unminified JavaScript.

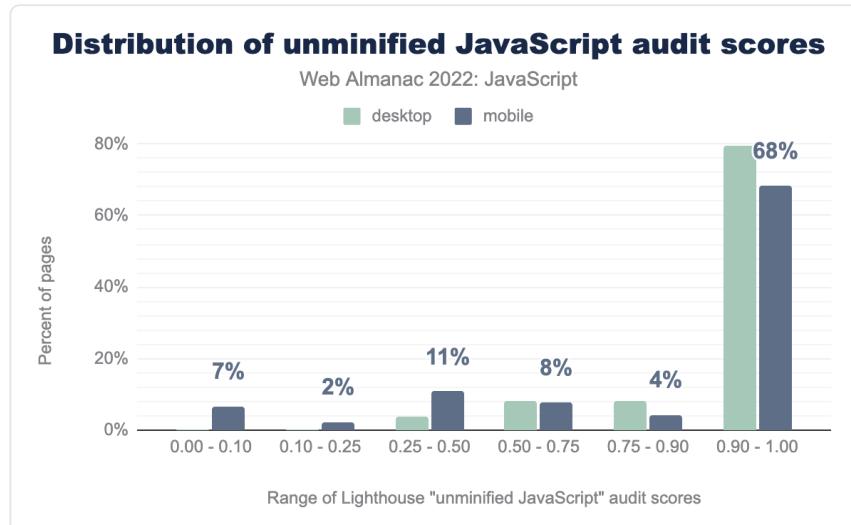


Figure 2.25. Distribution of unminified JavaScript audit scores.

44. <https://web.dev/unminified-javascript/>

Here, 0.00 represents the worst score whereas 1.00 represents the best score. 68% of mobile pages are scoring between 0.9 and 1.0 on Lighthouse's minified JavaScript audit, whereas the figure for desktop pages is 79%. This means that on mobile, 32% of pages have opportunities to ship minified JavaScript, whereas that figure for desktop pages is 21%.

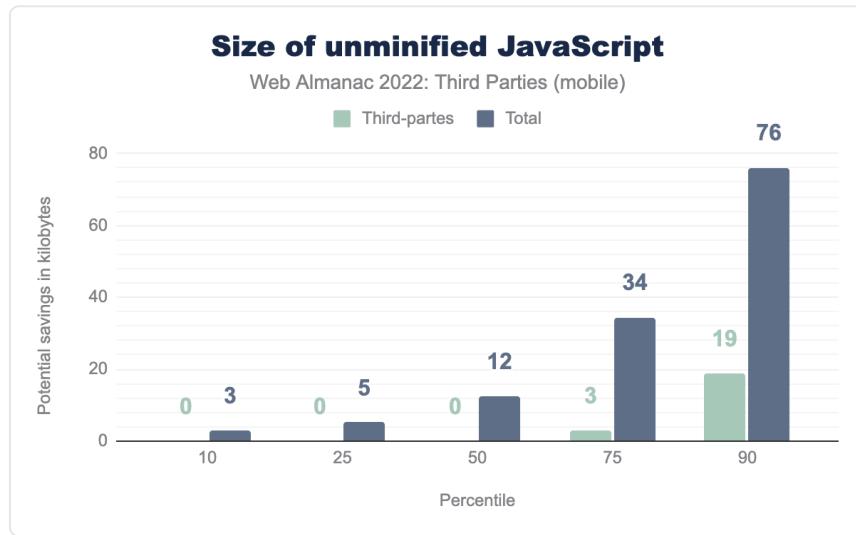


Figure 2.26. Distribution of the potential savings by minifying JavaScript.

At the median, we see that pages are shipping around 12 KB of JavaScript that can be minified. By the time we get to the 75th and 90th percentiles, however, that number jumps quite a bit, from 34 KB to about 76 KB. Third-parties are pretty good throughout, up until we get to the 90th percentile, however, where they're shipping around 19 KB of unminified JavaScript.

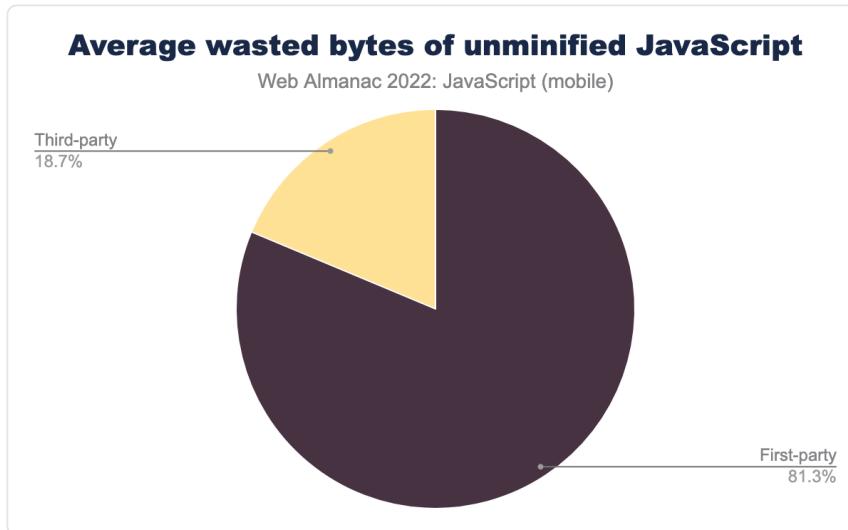


Figure 2.27. Average wasted bytes of unminified JavaScript.

Given the data we just presented, wasted bytes of unminified JavaScript isn't too surprising when you look at the average. First parties are overwhelmingly the biggest culprits in shipping unminified JavaScript at just over 80%. The remainder are just under 20% that could be doing a bit more to ship less bytes over the wire.

Minification addresses one of the first principles of web performance: ship less bytes. If you're failing the Lighthouse audit for unminified JavaScript, check your bundler's configuration to ensure your first party code is as streamlined for production as it can be. If you notice a third-party script that's unminified, it might be time to have a chat with that vendor to see what they can do to fix it. Refer to the Third Parties chapter for an even deeper look into the state of third parties on the web.

Source maps

Source maps⁴⁵ are a tool that web developers use to map minified and uglified production code to their original sources. Source maps are used in production JavaScript files, and are a useful debugging tool. Source maps can be specified in a comment pointing to a source map file at the end of a resource, or as the `SourceMap` HTTP response header.

45. https://firefox-source-docs.mozilla.org/devtools-user/debugger/how_to/use_a_source_map/index.html

14%

Figure 2.28. The percentage of mobile pages specifying source map comments to publicly accessible source maps.

14% of JavaScript resources accessed through mobile devices deliver a source map comment to a source map that is publicly accessible, whereas 15% of JavaScript resources accessed through desktop devices deliver them. However, the story is quite different for pages using a source map HTTP header.

0.12%

Figure 2.29. The number of mobile pages specifying source map headers.

Only 0.12% of requests for JavaScript resources on mobile devices used a source map HTTP header, whereas the number for desktop devices is 0.07%.

From a performance perspective, this doesn't mean much. Source maps are a developer experience enhancement. What you should avoid, however, is the use of inline source maps, which insert a base64 representation of the original source into a production-ready JavaScript asset. Inlining source maps means that you're not just sending your JavaScript resources to users, but also their source maps, which can lead to oversized JavaScript assets that take longer to download and process.

Responsiveness

JavaScript affects more than just startup performance. When we rely on JavaScript to provide interactivity, those interactions are driven by event handlers that take time to execute. Depending on the complexity of interactions and the amount of scripts involved to drive them, users may experience poor input responsiveness.

Metrics

Many metrics are used to assess responsiveness in both the lab and the field, and tools such as Lighthouse, Chrome UX Report (CrUX), and HTTP Archive track these metrics to provide a data-driven view of the current state of responsiveness on today's websites. Unless otherwise

noted, all of the following graphs depict the 75th percentile—the threshold for which Core Web Vitals are determined to be passing⁴⁶—of that metric at the origin level.

The first of these is First Input Delay (FID)⁴⁷, which records the input delay of the very first interaction made with a page. The input delay is the time between which the user has interacted with the page and when the event handlers for that interaction begin to run. It's considered a load responsiveness metric that focuses on the first impression a user gets when interacting with a website.

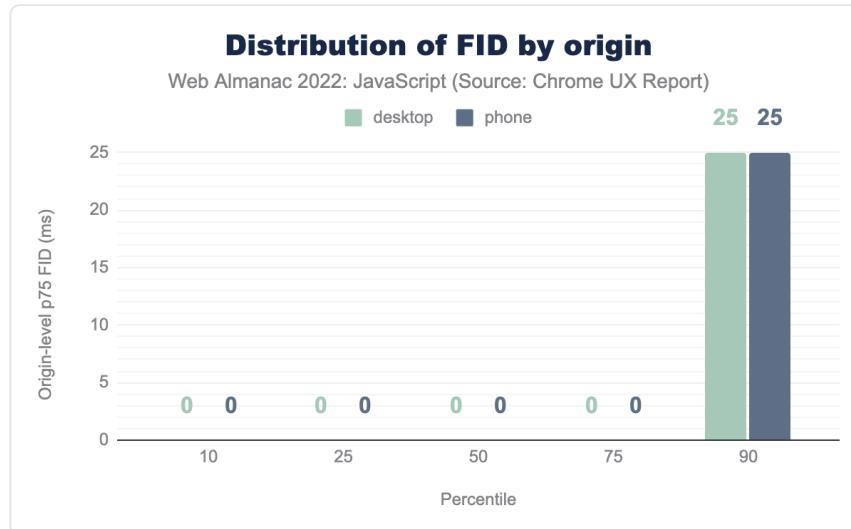


Figure 2.30. The distribution of websites' 75th percentile FID values.

This chart shows the distribution of all websites' 75th percentile FID values. The median website has a FID value of 0 ms for at least 75% of both desktop and phone user experiences. This “perfect FID” experience even extends into the 75th percentile of websites. Only when we get to the 90th percentile do we start to see imperfect FID values, but only 25 ms.

Given that the “good” FID threshold is 100 ms⁴⁸, we can say that at least 90% of websites meet this bar. In fact, we know from the analysis done in the Performance chapter that 100% of websites actually have “good” FID experiences on desktop devices, and 92% on mobile devices. FID is an unusually permissive metric.

46. <https://web.dev/vitals/#core-web-vitals>
 47. <https://web.dev/fid/>
 48. <https://web.dev/fid/#what-is-a-good-fid-score>

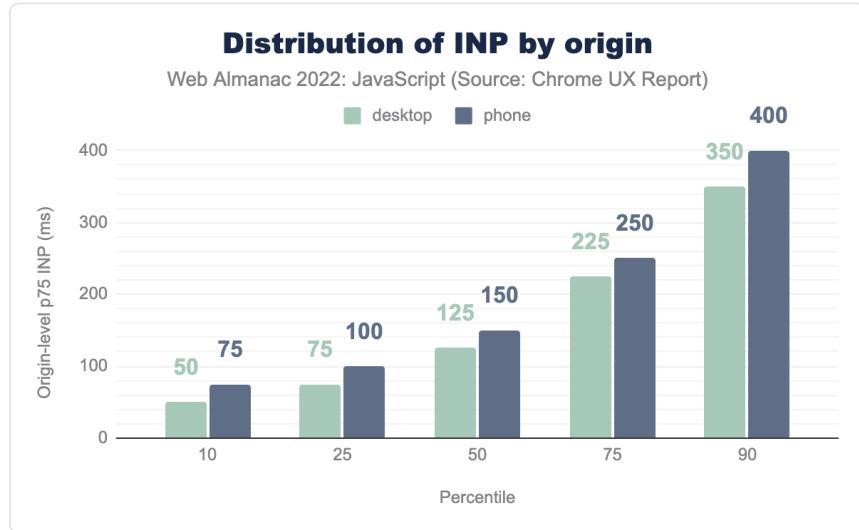


Figure 2.31. The distribution of websites' 75th percentile INP values.

In order to get a comprehensive view of page responsiveness across the entire page lifecycle, though, we need to look at Interaction to Next Paint (INP)⁴⁹, which assesses all keyboard, mouse, and touch interactions made with a page and selects a high percentile of interaction latency that's intended to represent overall page responsiveness.

Consider that a “good” INP score is 200 milliseconds⁵⁰ or less. At the median, both mobile and desktop score below this threshold, but the 75th percentile is another story, with both mobile and desktop segments well within the “needs improvement” range. This data, quite unlike FID, suggests that there are many opportunities for websites to do everything they can to run fewer long tasks⁵¹ on pages, which are a key contributor to less-than-good INP scores.

49. <https://web.dev/inp/>
 50. <https://web.dev/inp/#what's-a-%22good%22-inp-value>
 51. <https://web.dev/long-tasks-devtools/>

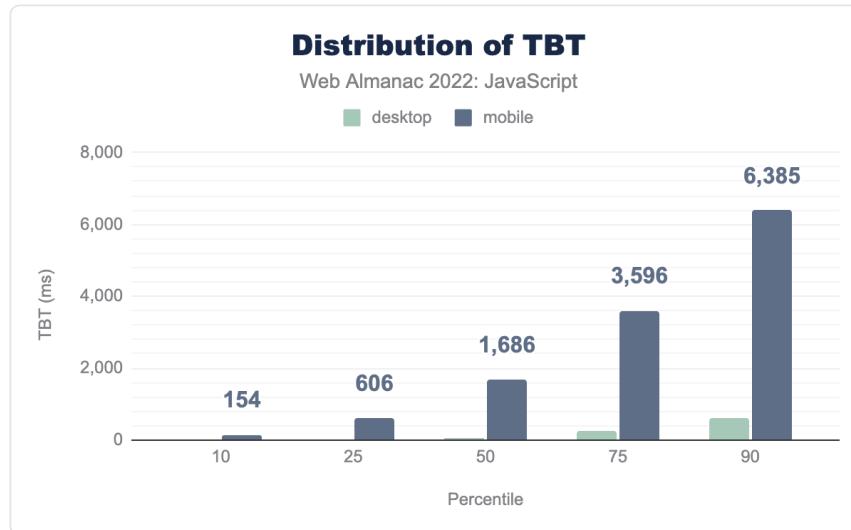


Figure 2.32. The distribution of pages' lab-based TBT values.

Dovetailing into long tasks, there's the Total Blocking Time (TBT)⁵² metric, which calculates the total blocking time of long tasks during startup.

Note that unlike the preceding stats on FID and INP, TBT and TTI (below) are not sourced from real-user data. Instead, we're measuring synthetic performance in simulated desktop and mobile environments with device-appropriate CPU and network throttling enabled. As a result of this approach, we get exactly one TBT and TTI value for each page, rather than a distribution of real-user values across the entire website.

Considering that INP correlates very well with TBT⁵³, it's reasonable to assume that high TBT scores may produce poorer INP scores. Using our synthetic approach, we see a wide gulf between desktop and mobile segments, indicating that desktop devices with better processing power and memory are outperforming less capable mobile devices by a wide margin. At the 75th percentile, a page has nearly 3.6 seconds of blocking time, which qualifies as a poor experience.

52. <https://web.dev/tbt/>
 53. https://github.com/GoogleChromeLabs/chrome-http-archive-analysis/blob/main/notebooks/HTTP_Archive_TBT_and_INP.ipynb

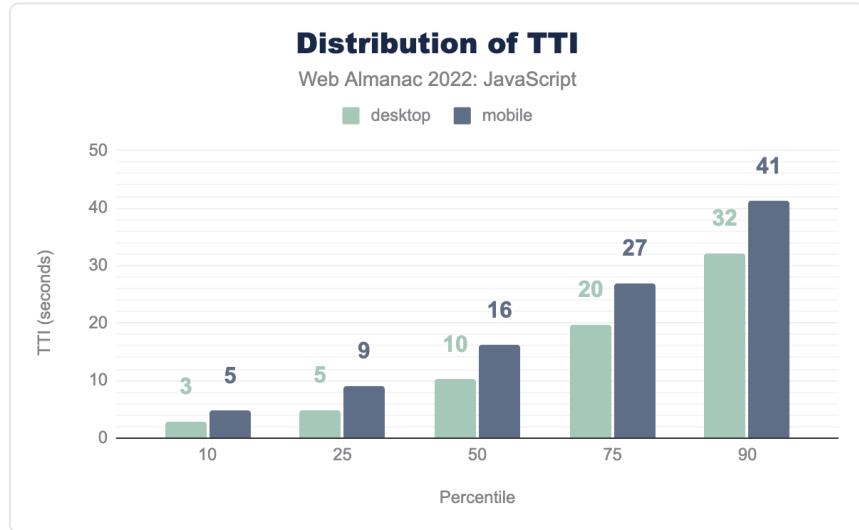


Figure 2.33. The distribution of the TTI scores by origin and percentile.

Finally, we come to Time to Interactive (TTI)⁵⁴, which is considered “good” if the metric comes in at under 5 seconds. Given that only the 10th percentile barely slips in under the 5 second mark, most websites in our simulated environment are relying on JavaScript to such an extent that pages are unable to become interactive within a reasonable timeframe—especially the 90th percentile, which takes a staggering 41.2 seconds to become interactive.

Long tasks/blocking time

As you may have gleaned from the previous section, the principal cause of poor interaction responsiveness is long tasks. To clarify, a long task is any task that runs on the main thread for longer than 50 milliseconds. The length of the task beyond 50 milliseconds is that task’s blocking time, which can be calculated by subtracting 50 milliseconds from the task’s total time.

Long tasks are a problem because they block the main thread from doing any other work until that task is finished. When a page has lots of long tasks, the browser can feel like it’s sluggish to respond to user input. In extreme cases, it can even feel like the browser isn’t responding at all.

54. <https://web.dev/tti/>

Distribution of number of long tasks per page

Web Almanac 2022: JavaScript

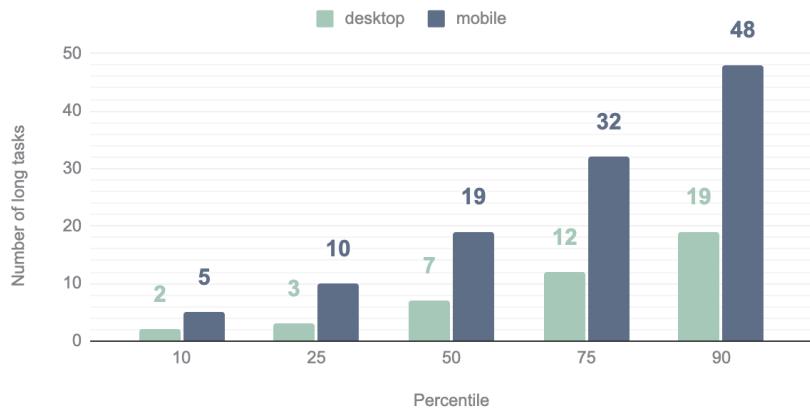


Figure 2.34. The distribution of the number of long tasks per page.

The median page encounters 19 long tasks on mobile and 7 long tasks on desktop devices. This makes sense when you consider that most desktop devices have greater processing power and memory resources than mobile devices, and are actively cooled.

However, the picture gets much worse at higher percentiles. Long tasks at the 75th percentile per page are 32 and 12 on mobile and desktop, respectively.

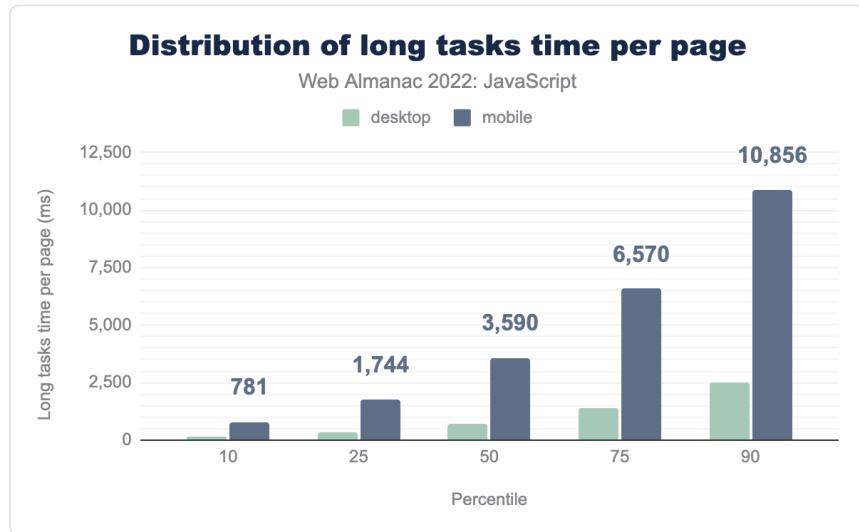


Figure 2.35. Distribution of long tasks time per page.

It's not enough to know how many long tasks there are per page—we need to understand the total time those tasks are taking on pages. The median mobile page has 3.59 seconds of time dedicated to long tasks, whereas desktop pages have far less at 0.74 seconds.

The 75th percentile suggests a much worse picture for those on mobile devices, coming in at nearly 6.6 seconds of processing time per page dedicated to handling long tasks. This is a lot of time the browser is spending on intense work that could be optimized or even possibly be moved to web workers on a different thread. In any case, these results spell trouble for the mobile web and responsiveness.

Scheduler API

Scheduling JavaScript tasks has historically been deferred to the browser. There are newer methods such as `requestIdleCallback` and `queueMicrotask`, but these APIs schedule tasks in a coarse way, and—especially in the case of `queueMicrotask`—can cause performance issues if misused.

The Scheduler API has recently been released, and gives developers finer control over scheduling tasks based on priority—though it is currently only limited to Chromium-based browsers⁵⁵.

55. https://caniuse.com/mdn-api_scheduler_posttask

0.002%

Figure 2.36. The percentage of mobile pages using the Scheduler API.

Only 20 per million (0.002%) mobile pages are currently shipping JavaScript that uses the Scheduler API, whereas 30 per million (0.003%) desktop pages do. This is not surprising, considering the lack of documentation on this very new feature, and its limited support. However, we expect this number to increase as documentation on the feature becomes available, and especially if it is used in frameworks. We believe that the adoption of this important new feature will eventually result in better user experience outcomes.

Synchronous XHR

AJAX—or usage of the `XMLHttpRequest` (XHR) method to asynchronously retrieve data and update information on the page without a navigation request—was a very popular method of creating dynamic user experiences. It has largely been replaced by the asynchronous `fetch` method, but XHR is still supported in all major browsers⁵⁶.

XHR has a flag that allows you to make synchronous requests. Synchronous XHR⁵⁷ is harmful for performance because the event loop and main thread is blocked until the request is finished, resulting in the page hanging until the data becomes available. `fetch` is a much more effective and efficient alternative with a simpler API, and has no support for synchronous fetching of data.

2.5%

Figure 2.37. The percentage of mobile pages using synchronous XHR.

While synchronous XHR is only used on 2.5% of mobile pages and 2.8% of desktop pages, its continued use—no matter how small—is still a signal that some legacy applications may be relying on this outdated method that harms the user experience.

Avoid using synchronous XHR, and XHR in general. `fetch` is a much more ergonomic alternative that lacks synchronous functionality by design. Your pages will perform better without synchronous XHR, and we hope someday to see this number fall to zero.

56. https://caniuse.com/mdn-api_xmlhttprequest

57. https://developer.mozilla.org/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests#synchronous_request

document.write

Before the introduction of DOM insertion methods (`appendChild` and others, for example), `document.write` was used to insert content at the position the `document.write` was made in the document.

`document.write` is very problematic. For one, it blocks the HTML parser, and is problematic for a number of other reasons the HTML spec itself warns against its use⁵⁸. On slow connections, blocking document parsing to append nodes in this way creates performance problems that are entirely avoidable.

18%

Figure 2.38. The number of mobile pages using `document.write`.

A staggering 18% of pages observed are still using `document.write` to add content to the DOM in lieu of proper insertion methods, whereas 17% of desktop pages are still doing so. The explanation for this could be legacy applications that haven't been rewritten to use the preferred DOM methods to insert new nodes into the document, and even some third-party scripts that still use it.

We hope to see a decline in this trend. All major browsers explicitly warn against using this method. While it isn't deprecated just yet, its existence in browsers in the years to come isn't guaranteed. If `document.write` calls are in your website, you should prioritize removing them as soon as possible.

Legacy JavaScript

JavaScript has evolved considerably over the last several years. The introduction of new language features has turned JavaScript into a more capable and elegant language that helps developers to write more concise JavaScript, resulting in less JavaScript loaded—provided those features haven't been unnecessarily transformed into a legacy syntax by using a transpiler such as Babel.

Lighthouse currently checks for Babel transforms that may be unnecessary on the modern web, such as transforming use of `async` and `await`, JavaScript classes⁵⁹, and other newer, yet widely supported language features.

58. [`https://html.spec.whatwg.org/multipage/dynamic-markup-insertion.html#document.write\(\)`](https://html.spec.whatwg.org/multipage/dynamic-markup-insertion.html#document.write())

59. [`https://developer.mozilla.org/docs/Web/JavaScript/Reference/Classes`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Classes)



Figure 2.39. The percentage of mobile pages that ship legacy JavaScript.

Just over two thirds of mobile pages are shipping JavaScript resources that are being transformed, or otherwise contain unnecessary legacy JavaScript.

Transformations can add a lot of extra bytes to production JavaScript for the sake of compatibility, but unless there is a necessity to support older browsers, many of these transforms are unnecessary, and can harm startup performance. That so many pages on mobile—and 68% of pages on desktop—are shipping these transforms is concerning.

Babel is doing much to solve this problem out of the box, such as through the compiler assumptions feature⁶⁰, but Babel is still driven by user-defined configurations, and can only do so much in the presence of outdated configuration files.

As mentioned above, we strongly encourage developers to carefully review their Babel⁶¹ and Browserslist⁶² configurations to ensure that the minimum amount of transforms are applied to code in order for them to work in required browsers. Doing so can result in large reduction of bytes shipped to end users. Developers have a lot of work to do in this area, and we hope to see this figure decline over time now that the language's evolution has relatively stabilized.

How is JavaScript used?

There's more than one way to build a web page. While some may opt to use the web platform directly, it's undeniable that the trend in the web developer industry is to reach for abstractions that make our work easier to do and reason about. As is the case with previous years, we'll be exploring the role of libraries and frameworks, as well as how frequently those libraries and frameworks present security vulnerabilities that can make the web a riskier place for users.

Libraries and frameworks

Libraries and frameworks are a huge part of the developer experience—one that has the potential to harm performance through framework overhead. Though developers have largely accepted this trade-off, understanding what libraries and frameworks are commonly used on

60. <https://babeljs.io/docs/en/assumptions>
61. <https://babeljs.io/docs/en/configuration>
62. <https://github.com/browserslist/browserslist>

the web is extremely important, as it informs our understanding of how the web is built. In this section, we'll be taking a look at the state of libraries and frameworks across the web in 2022.

Library usage

To understand the usage of libraries and frameworks, HTTP Archive uses Wappalyzer to detect the technologies used on a page.

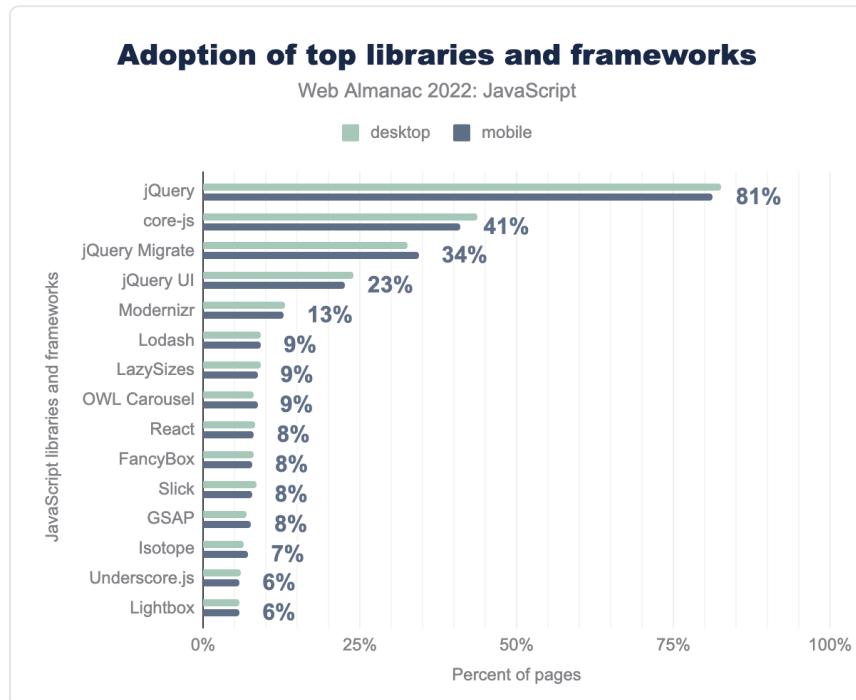


Figure 2.40. Adoption of top libraries and frameworks.

It's still no surprise that jQuery is by far the most used library on the web today. Part of that is because WordPress is used on 35% of sites, but even so, the majority of jQuery usage occurs outside of the WordPress platform.

While jQuery is relatively small and reasonably quick at what it does, it still represents a certain amount of overhead in applications. Most of what jQuery offers is now doable with native DOM APIs⁶³, and may be unnecessary in today's web applications.

63. <https://youmighthnotinneedjquery.com/>

The usage of core-js is also not surprising, as many web applications transform their code with Babel, which often uses core-js to fill in missing gaps in APIs across browsers. As browsers mature, this figure should drop—and that would be a good thing indeed, as modern browsers are more capable than ever, and shipping core-js code could end up being wasted bytes.

React usage notably remained the same from last year at 8%, which may be a signal that adoption of the library has plateaued due to an increasing amount of choices in the JavaScript ecosystem.

Libraries used together

It's not an uncommon scenario to see multiple frameworks and libraries used on the same page. As with last year, we'll examine this phenomenon to gain insight into how many libraries and frameworks have been used together in 2022.

Libraries	Desktop	Mobile
jQuery	10.19%	10.33%
jQuery, jQuery Migrate	4.30%	4.94%
core-js, jQuery, jQuery Migrate	2.48%	2.80%
core-js, jQuery	2.78%	2.74%
jQuery, jQuery UI	2.40%	2.07%
core-js, jQuery, jQuery Migrate, jQuery UI	1.18%	1.36%
jQuery, jQuery Migrate, jQuery UI	0.88%	0.99%
GSAP, Lodash, Polyfill, React	0.48%	0.93%
Modernizr, jQuery	0.87%	0.86%
core-js	0.92%	0.85%

Figure 2.41. Analysis of libraries and frameworks used together on observed pages.

It's clear though that jQuery has some serious staying power, with some combination of it, its UI framework, and its migration plugin occurring in the top seven spots, with core-js having a prominent role in library usage as well.

Security vulnerabilities

Given the wide proliferation of JavaScript on today's web, and with the advent of installable JavaScript packages, it's no surprise that security vulnerabilities exist in the JavaScript ecosystem.

57%

Figure 2.42. The percentage of mobile pages that download a vulnerable JavaScript library or framework.

While 57% of mobile pages serving up a vulnerable JavaScript library or framework is significant, this figure is down from last year's figure of 64%. This is encouraging, but there's quite a bit of work to be done to lower this figure. We hope that as more security vulnerabilities are patched, developers will be incentivized to update their dependencies to avoid exposing their users to harm.

Library or framework	Desktop	Mobile
jQuery	49.12%	48.80%
jQuery UI	16.01%	14.88%
Bootstrap	11.53%	11.19%
Moment.js	4.54%	3.91%
Underscore	3.41%	3.11%
Lo-Dash	2.52%	2.44%
GreenSock JS	1.65%	1.62%
Handlebars	1.27%	1.12%
AngularJS ⁶⁴	0.99%	0.79%
Mustache	0.44%	0.57%

Figure 2.43. The percentage of pages having known JavaScript vulnerabilities among the top ten most commonly used libraries and frameworks.

64. <https://angularjs.org>

With jQuery being the most popular library in use on the web today, it's no surprise that it and its associated UI framework represents a fair amount of the security vulnerabilities that users are exposed to on the web today. This could likely be that some developers are still using older versions of these scripts which don't take advantage of fixes to known vulnerabilities.

A notable entry is Bootstrap, which is a UI framework that helps developers to quickly prototype or build new layouts without using CSS directly. Given the release of newer CSS layout modes such as Grid or Flexbox, we may see usage of Bootstrap decrease over time, or in lieu of that, see developers update their Bootstrap dependencies to ship more safe and secure websites.

Regardless of what libraries and frameworks you use, be sure to regularly update your dependencies wherever possible to avoid exposing your users to harm. While package updates do result in some amount of refactoring or code fixes from time to time, the effort is worth the reduction in liability and increase in user safety.

Web components and shadow DOM

For some time, web development has been driven by a componentization model employed by numerous frameworks. The web platform has similarly evolved to provide encapsulation of logic and styling through web components and the shadow DOM. To kick off this year's analysis, we'll begin with custom elements⁶⁵.

A large, bold, blue percentage value of 2.0%.

Figure 2.44. The percentage of desktop pages that used custom elements.

This figure is down a bit from last year's analysis of custom element usage on desktop pages, which was 3%. With the advantages that custom elements provide and their reasonably broad support in modern browsers, we're hoping that the web component model will compel developers to leverage web platform built-ins to create faster user experiences.

A large, bold, blue percentage value of 0.39%.

Figure 2.45. The percentage of mobile pages that used shadow DOM.

65. <https://developers.google.com/web/fundamentals/web-components/custom-elements>

Shadow DOM⁶⁶ allows you to create dedicated nodes in a document that contain their own scope for sub-elements and styling, isolating a component from the main DOM tree. Compared to last year's figure of 0.37% of all pages using shadow DOM, adoption of the feature has remained much the same, with 0.39% of mobile pages and 0.47% of desktop pages using it.

0.05%

Figure 2.46. The percentage of mobile pages that use templates.

The `template` element helps developers reuse markup patterns. Their contents render only when referenced by JavaScript. Templates work well with web components, as the content that is not yet referenced by JavaScript is then appended to a shadow root using the shadow DOM.

Roughly 0.05% of web pages on both desktop and mobile are currently using the `template` element. Though templates are well supported in browsers, their adoption is currently scant.

0.08%

Figure 2.47. The percentage of mobile pages that used the `is` attribute.

The HTML `is` attribute is an alternate way of inserting custom elements into the page. Rather than using the custom element's name as the HTML tag, the name is passed to any standard HTML element, which implements the web component logic. The `is` attribute is a way to use web components that can still fall back to standard HTML element behavior if web components fail to be registered on the page.

This is the first year we are tracking usage of this attribute, and unsurprisingly, its adoption is lower than custom elements themselves. Due to the lack of support in Safari, this means that browsers on iOS and Safari on macOS can't make use of the attribute, possibly contributing to the attribute's limited usage.

Conclusion

The state of JavaScript is largely continuing the way trends would have suggested last year. We're shipping more of it, for sure, but we're also trying to mitigate the ill effects of excessive

66. <https://developers.google.com/web/fundamentals/web-components/shadowdom>

JavaScript through increased usage of techniques such as minification, resource hints, compression, and even down to the libraries we use.

The state of JavaScript is a constantly evolving phenomenon. It's clear that we have an increased reliance on it more than ever, but that spells trouble for the collective user experience of the web. We need to do all we can—and more—to stem the tide of how much JavaScript we ship on production websites.

As the web platform matures, we're hoping that we see increased direct adoption of its various APIs and features where it makes sense to do so. For those experiences that require frameworks for a better developer experience, we're hoping to see additional optimizations and opportunities for framework authors to adopt new APIs to help them develop on both a better developer experience and better experiences for users.

Let's hope that next year signals a shift in the trend. In the meantime, let's continue to do all we can⁶⁷ to make the web as fast as we possibly can, while keeping an eye on both lab⁶⁸ and field⁶⁹ data along the way.

Author



Jeremy Wagner

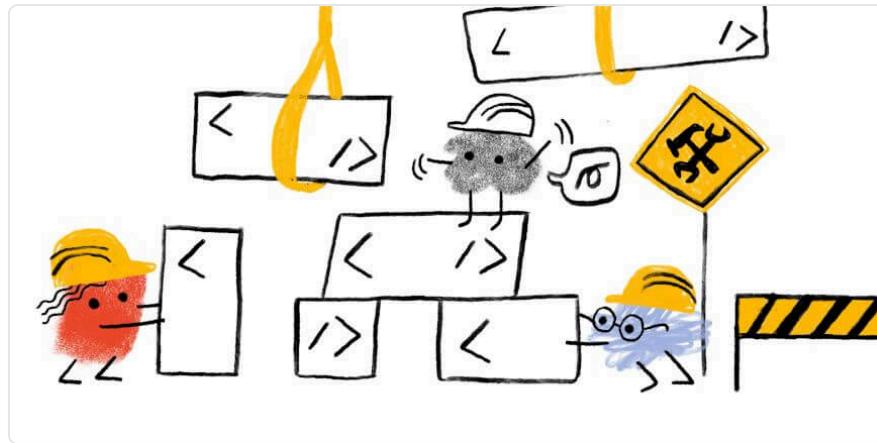
Twitter: @malchata | GitHub: malchata | Website: <https://jlwagner.net/>

Jeremy Wagner is a technical writer for Google on performance and Core Web Vitals. He has also written for A List Apart, CSS-Tricks, and Smashing Magazine. Jeremy will someday relocate to the remote wilderness where sand has not yet been taught to think. Until then, he continues to reside in Minnesota's Twin Cities with his wife and stepdaughters, bemoaning the existence of strip malls.

67. <https://web.dev/fast/>
68. <https://web.dev/lab-and-field-data-differences/#lab-data>
69. <https://web.dev/lab-and-field-data-differences/#field-data>

Part I Chapter 3

Markup



Written by Jens Oliver Meiert

Reviewed by Brian Kardell and Simon Pieters

Analyzed by Rick Viscomi

Edited by Barry Pollard

Introduction

As the 2020 chapter said⁷⁰, without HTML there are no web pages, no web sites, no web apps. You can say that without HTML, there's no Web. That makes HTML one of the most important web standards, if not the most important web standard.

Accordingly, like every year, we used the millions of pages in our data set—7.9 million in the mobile set, 5.4 million in the desktop set, with overlap—to also look at HTML. This chapter doesn't cover “everything” there is about HTML, so we explicitly encourage you to also analyze the data we gathered⁷¹ and to share your own conclusions—and when you do, tag them: #htmlalmanac⁷².

70. <https://almanac.httparchive.org/en/2020/markup#introduction>
 71. https://docs.google.com/spreadsheets/d/1grkd2_1xSV3jvNK6ucRQOOL1HmGTsScHuwA8GZuRLHU/edit
 72. <https://twitter.com/hashtag/htmlalmanac>

Document data

There's much to be curious about when it comes to how we write HTML. We can ask lots of questions, but when it comes to HTML in general, let's have a look at how our HTML is sent to our browsers, before we even get into the contents of the markup itself.

Doctypes

Doctype	Desktop	Mobile
<code>html</code>	88.1%	90.0%
<code>html -//w3c//dtd XHTML 1.0 transitional//en http://www.w3.org/tr/xhtml1/dtd/ xhtml1-transitional.dtd</code>	4.7%	3.9%
<code>No doctype</code>	3.0%	2.7%
<code>html -//w3c//dtd XHTML 1.0 strict//en http://www.w3.org/tr/xhtml1/dtd/xhtml1-strict.dtd</code>	1.2%	1.1%
<code>html -//w3c//dtd HTML 4.01 transitional//en http://www.w3.org/tr/html4/loose.dtd</code>	0.9%	0.6%
<code>html -//w3c//dtd HTML 4.01 transitional//en</code>	0.4%	0.4%

Figure 3.1. Doctype usage.

Let's start with doctypes—which one is the most popular? But you know the answer to this one: It's the short, simple, boring standard HTML doctype, that is, `<!DOCTYPE html>`.

90%

Figure 3.2. Mobile using the standard HTML doctype.

90% of all mobile pages use it—as the mobile data set is largest, this chapter will usually work with that data. Next most popular is XHTML 1.0 Transitional (3.9%, down from 4.6% in 2021⁷³). After that it's no doctype being set at all at 2.7%, up from 2.5% last year.

73. <https://almanac.httparchive.org/en/2021/markup#doctypes>

Compression

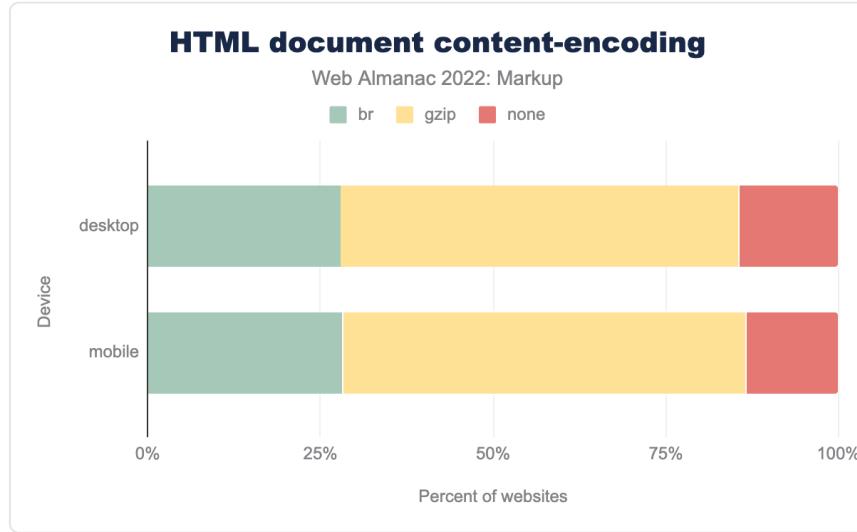


Figure 3.3. HTML document content encoding.

Are HTML documents being compressed? How many? How? 86% of them are—with 58% (down 5.8% since last year) overall being gzip-compressed, and 28% (up 6.1%) being compressed using Brotli. Overall, slightly more documents are being compressed, and compressed more effectively.

Languages

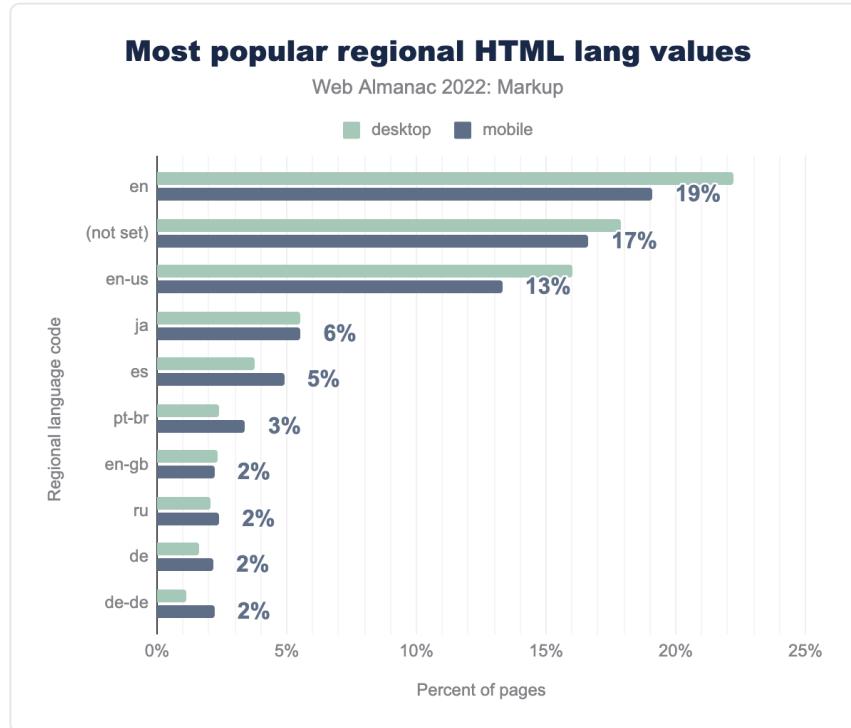


Figure 3.4. Most popular regional HTML lang values.

What about languages? In our data set, 35% of pages used a `lang` attribute mapping to English; 17% had no language set; and you already see the difficulties—the sample is likely biased and also not as big as to reflect all of the world, and no `lang` attribute being used is not equaling no language being set so, this isn't something our data would be useful for.

Conformance

Do documents conform with the HTML specification—i.e., are they valid? A quick way for you to tell is by using a tool like the W3C markup validation service⁷⁴.

We didn't and we couldn't check this yet. So why include this section?

The reason to at least mention conformance is that if you don't check on conformance, if you

74. <https://validator.w3.org/>

don't validate, there's a good chance—in practice, effectively a 100% chance⁷⁵—you end up writing at least some fictitious and fantasy (and therefore wrong) HTML. But HTML isn't fiction or fantasy—it's a hard technical standard with clear rules on what works and what doesn't.

For a professional, it's good to know these rules. It's good work to produce code that works and that doesn't contain anything superfluous, too. And both of that—learning and not shipping anything non-working or superfluous—is why conformance matters, and why validation matters.

We don't have conformance data to share in the Web Almanac yet, but that doesn't mean the point is any less important. And if you haven't focused on conformance yet—start validating your HTML output. Maybe one of the next editions of the Web Almanac will have some positive news to share because of you.

Document size

HTML payload and document size are a staple in this series—we've looked at this information since 2019. But the trend is clear, and while it follows a common theme that other chapters will confirm, too, it's not a great one:

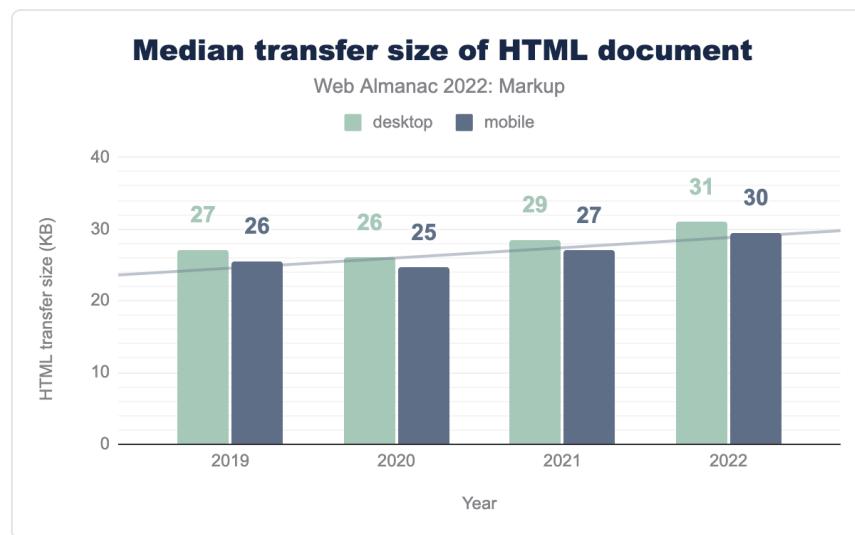


Figure 3.5. Median transfer size of HTML document.

After some brief relief in 2020, document size has continued growing in 2021, and again in

75. <https://meiert.com/en/blog/valid-html-2022/>

2022, with a median transfer size of 30 kB in our mobile data set.

One way to counter this trend is to write HTML, the HTML way (and not the XHTML way)⁷⁶, as that would already result in smaller HTML transfer size. Disclosure: Your author here likes to come up with HTML writing classifications, and enjoys promoting minimal HTML.

Elements

If you're not including the `svg` and `math` elements—because they're specified outside of HTML—the current HTML specification currently consists of 111 elements.

Elements, not tags, because we're not referring to mere start or end tags, like `` or `</ins>`. And some people count HTML elements differently, but most important is to be clear about how you're counting⁷⁷.

What can we observe?

Element diversity

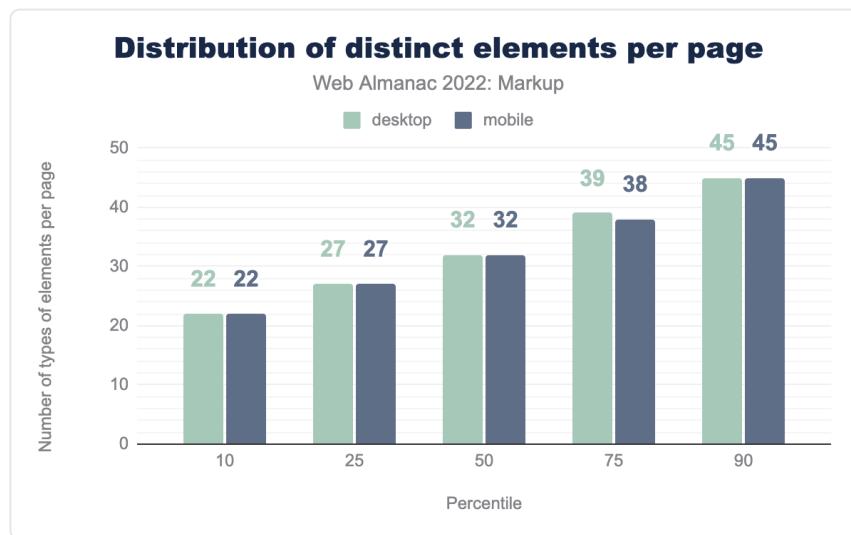


Figure 3.6. Distribution of distinct elements per page.

The first thing we can note is that developers use slightly more different elements per page

76. <https://css-tricks.com/write-html-the-html-way-not-the-xhtml-way/>

77. <https://meiert.com/en/blog/the-number-of-html-elements/>

now, with a median of 32 different elements per document.

The median is up from 31 elements in 2021⁷⁸, and 30 elements in 2020⁷⁹. As this is a trend throughout, it may be a tender sign that developers put HTML elements to better use, by using more of them for what they're there for.

Alas, there's another trend which aligns with an increasing document size, and that's a growing number of elements per page in total:

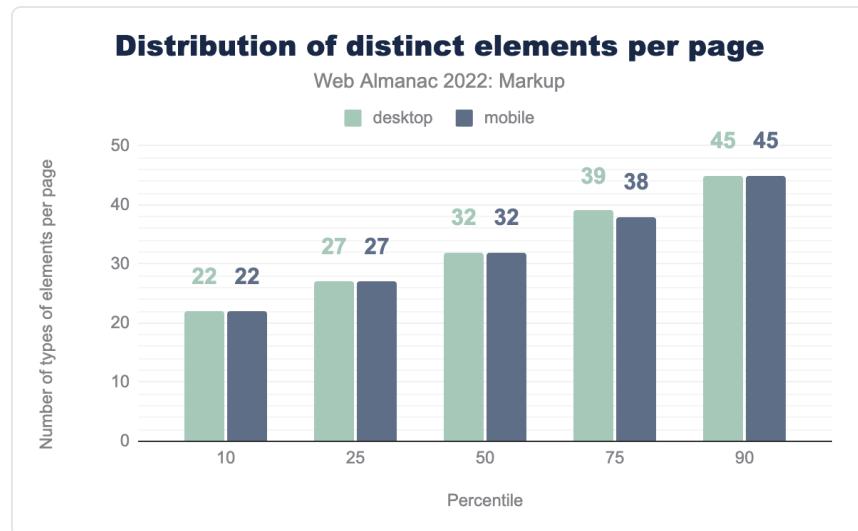


Figure 3.7. Distribution of elements per page.

The median is currently at 653 elements per page, up from 616 in 2021, and 587 in 2020—all per the respective mobile data set. Do we publish more content, requiring more elements to hold them (something like, more paragraphs per text, more `p` elements)? Or is this just another sign of an unchecked `div` pandemic? Our data doesn't answer this but it is probably due to both—and more—reasons.

Top elements

The following elements are used most frequently:

78. <https://almanac.httparchive.org/en/2021/markup#element-diversity>
 79. <https://almanac.httparchive.org/en/2020/markup#element-diversity>

2019	2020	2021	2022
<i>div</i>	<i>div</i>	<i>div</i>	<i>div</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>span</i>	<i>span</i>	<i>span</i>	<i>span</i>
<i>li</i>	<i>li</i>	<i>li</i>	<i>li</i>
<i>img</i>	<i>img</i>	<i>img</i>	<i>img</i>
<i>script</i>	<i>script</i>	<i>script</i>	<i>script</i>
<i>p</i>	<i>p</i>	<i>p</i>	<i>p</i>
<i>option</i>	<i>link</i>	<i>link</i>	<i>link</i>
<i>i</i>	<i>meta</i>	<i>i</i>	
<i>option</i>	<i>i</i>	<i>meta</i>	

Figure 3.8. Most used elements.

The `div` element is—by far—the most popular element: We found 2,123,819,193 occurrences in the mobile data set, and 1,522,017,185 of them in our desktop data set.

29%

Figure 3.9. Percentage of elements which are `div` elements.

Divitis⁸⁰ is real.

If you wonder about the odd one out, the `i` element, it stands to reason that this is still largely due to Font Awesome⁸¹ and its arguable misuse of this element. The element has also a bad reputation because during XHTML times, everyone suggested to use `em` instead—but that advice wasn't sound, and `i` elements have their use cases.

When it comes to what elements are being used on the most documents, the list looks a little different:

80. <https://en.wiktionary.org/wiki/divitis>

81. <https://fontawesome.com/>

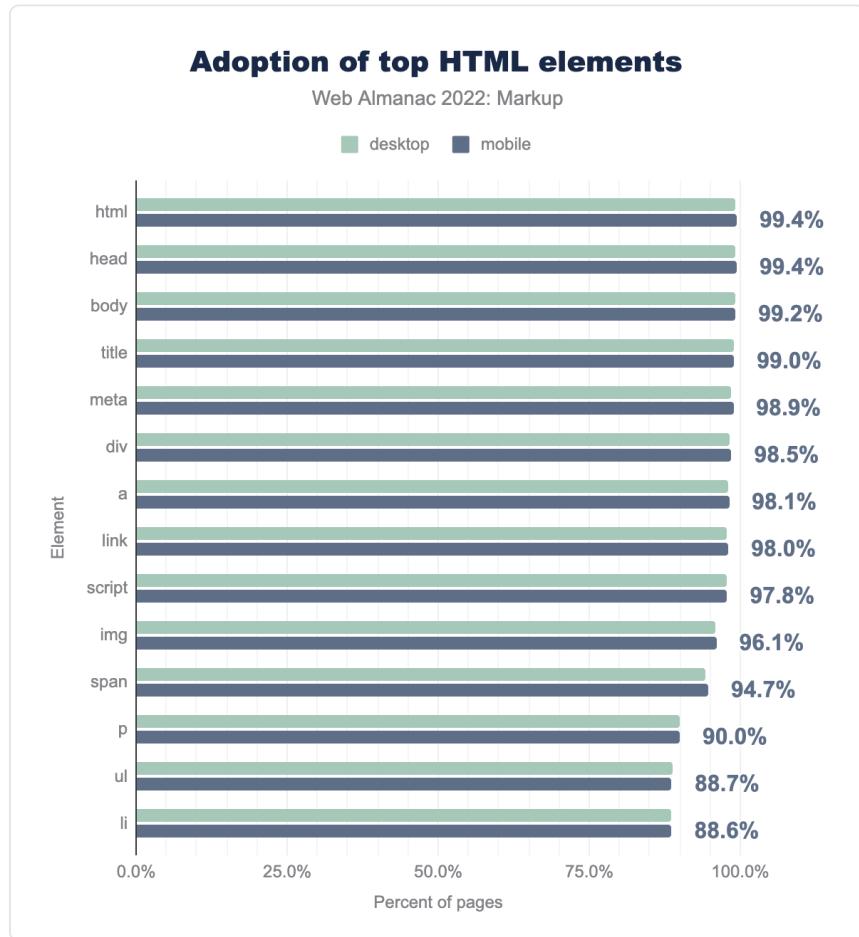


Figure 3.10. Adoption of top HTML elements.

It's not a surprise that nearly every document uses `html`, `head`, or `body` tags—they are automatically inserted in the DOM and that is what is being counted here. That the numbers are slightly less than 100% is due to a small number of pages that break detection by overriding the JavaScript APIs we use—for example, MooTools⁸² overriding the `JSON.stringify()` API.

It's a lot more surprising to miss `title` on 1% of all sampled documents—this element is not optional, and not being inserted in the DOM, and its omission an indicator for lack of conformance checking.

82. <https://mootools.net/>

The elements that then follow are old friends—especially `a`, `img`, and `meta` have been popular elements ever since Ian Hickson's seminal HTML study back⁸³ in 2005.

What's the least used HTML element that's part of the current standard, you ask? That's `samp`, with a mere 2,002 findings in our mobile set.

Custom elements

Custom elements⁸⁴—elements we can loosely identify by their inner-name use of a hyphen—also made it into our samples again. This year, however, the Top 10 is entirely dominated by Slider Revolution⁸⁵:

Custom element	Desktop	Mobile
<code>rs-module-wrap</code>	2.1%	2.3%
<code>rs-module</code>	2.1%	2.3%
<code>rs-slides</code>	2.1%	2.3%
<code>rs-slide</code>	2.1%	2.3%
<code>rs-sbg-wrap</code>	2.0%	2.2%
<code>rs-sbg-px</code>	2.0%	2.2%
<code>rs-sbg</code>	2.0%	2.2%
<code>rs-progress</code>	2.0%	2.2%
<code>rs-layer</code>	1.8%	2.0%
<code>rs-mask-wrap</code>	1.8%	2.0%

Figure 3.11. Most used custom elements.

That's impressive—but gives us little to work with other than saying that Slider Revolution is used on roughly 2% of all sampled pages.

What are the next popular custom elements that are not part of Slider Revolution?

83. <https://web.archive.org/web/20060203035414/http://code.google.com/webstats/index.html>
 84. <https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements-core-concepts>
 85. <https://www.sliderrevolution.com/>

	Custom element	Desktop	Mobile
<code>pages-css</code>		1.1%	2.0%
<code>wix-image</code>		1.1%	2.0%
<code>router-outlet</code>		0.7%	0.5%
<code>wix-iframe</code>		0.4%	0.7%
<code>ss3-loader</code>		0.5%	0.5%

Figure 3.12. Most used custom elements not starting with `rs-`.

This is more diverse: `pages-css`, `wix-image` and `wix-iframe` come from the Wix website builder. `router-outlet` originates in Angular. And `ss3-loader` seems to be related to Smart Slider.

Obsolete elements

Are obsolete elements still a thing? Given that not-validating is still a thing, yes.

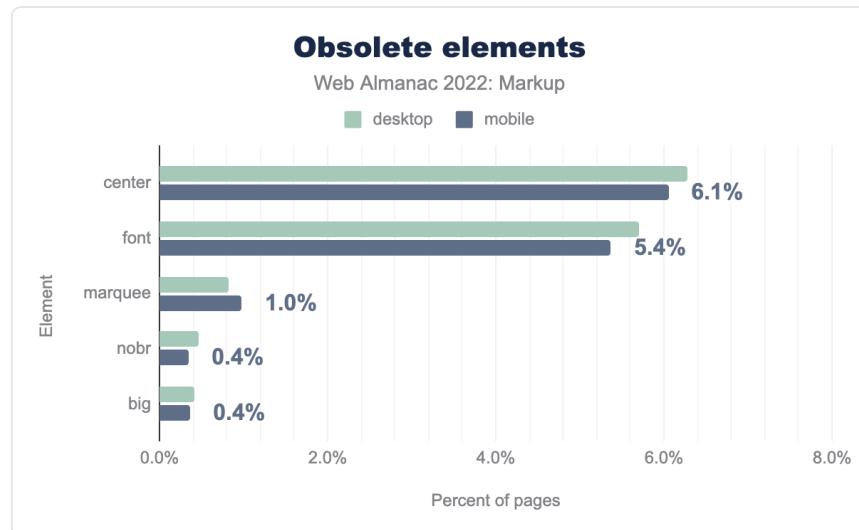


Figure 3.13. Obsolete elements.

On 6.1% of pages, you still find `center` elements (hi Google homepage⁸⁶), and on 5.4% of pages, you find `font` elements. Use of both elements went down (down 0.5% in both cases), fortunately, while `marquee`, `nobr`, and `big` didn't witness significant changes.

`center` and `font` make for the lion's share (81.2%) of all obsolete elements, per our analysis:

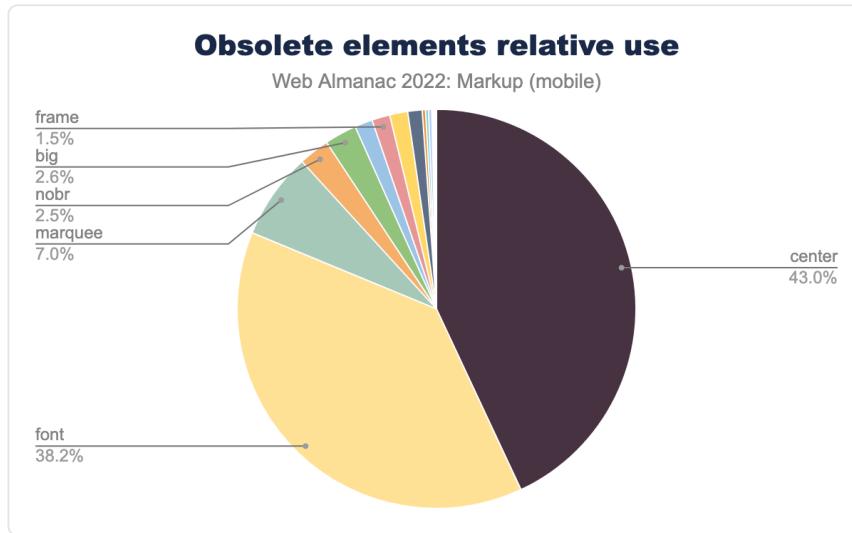


Figure 3.14. Obsolete elements relative use.

Attributes

If elements are the bread of HTML, then attributes are the butter. What can we learn here?

Top attributes

The most popular attribute, by far, was and still is `class`:

86. <https://www.google.com/>

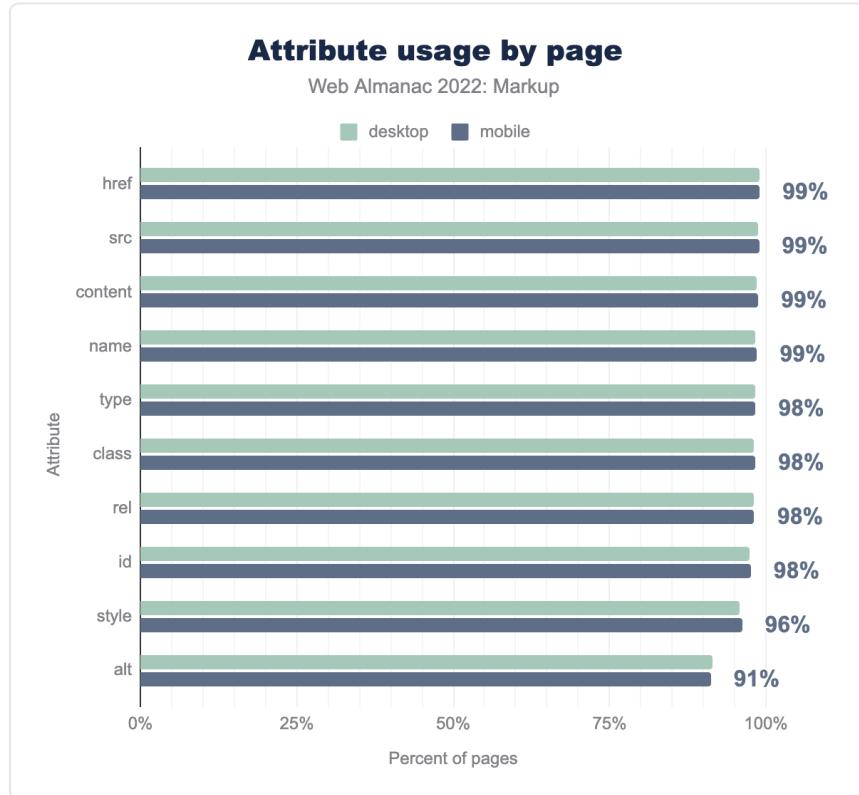


Figure 3.15. Attribute usage.

This order isn't any different from what we've seen last year, but there are some changes:

- `class` ($\downarrow 0.3\%$), `href` ($\downarrow 0.9\%$), `style` ($\downarrow 0.6\%$), `id` ($\downarrow 0.2\%$), `type` ($\downarrow 0.1\%$), `title` ($\downarrow 0.3\%$), and `value` ($\downarrow 0.5\%$) are all used a little less than before.
- `src` ($\uparrow 0.3\%$) and `alt` ($\uparrow 0.1\%$) are used more than before—tentatively good news for accessibility!
- `rel` usage hasn't changed significantly.

Are there attributes we find on (nearly) every document? Yes:

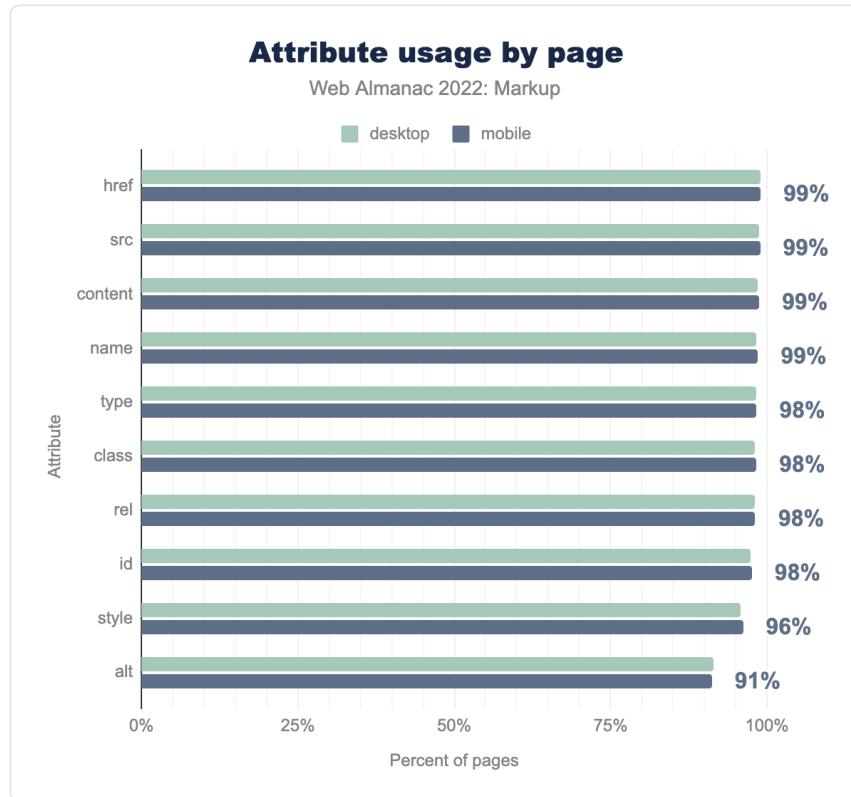


Figure 3.16. Attribute usage by page.

`href`, `src`, `content` (metadata), and `name` (metadata, form identifiers) are present on nearly every document in our sample.

`data-*` attributes

For `data-*` attributes—which allow authors to embed their own custom metadata—we also pulled new information.

This changed only little compared to last year's `data-*` attributes stats. Here are some changes to call out:

- `data-id` is still the most popular `data-*` attribute, with a 0.7% increase compared to 2021.

- `data-element_type`, though its position stayed the same, gained 0.7% as well.
- `data-testid` ranked #6 before, gained 0.3%, and jumped to #4.
- `data-widget_type` ranked #8, gained 0.4% popularity, and also gained two spots, taking #6 in 2022.

`data-element_type` and `data-widget_type` relate to Elementor⁸⁷, while `data-testid` is coming from Testing Library⁸⁸.

Let's have a look at how often we find `data-*` attributes on our pages:

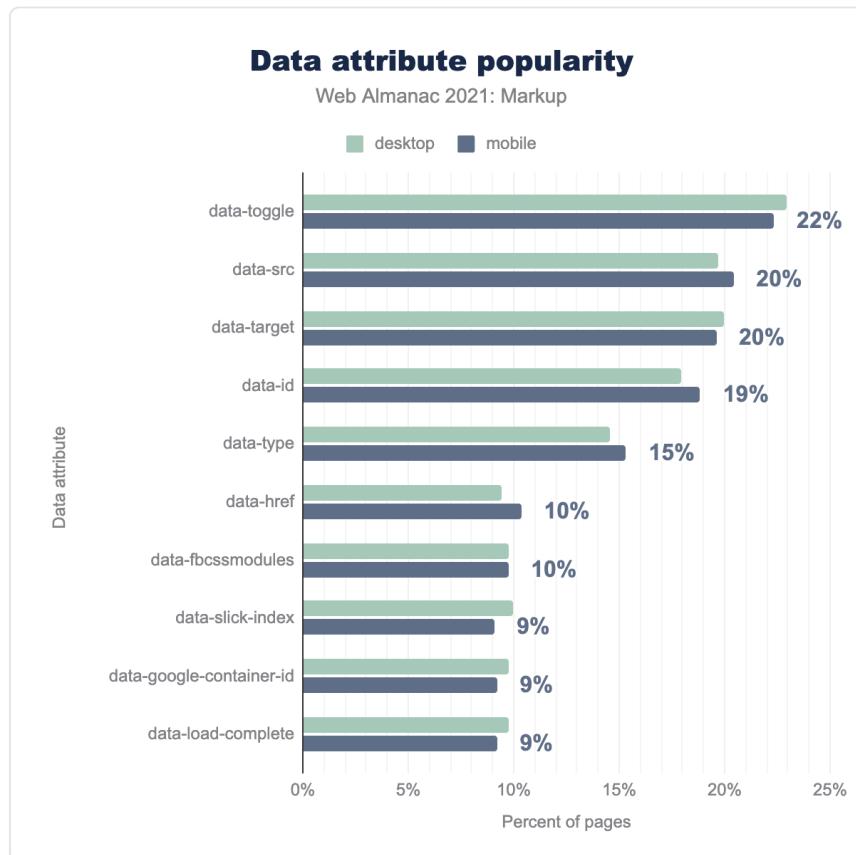


Figure 3.17. Data attribute popularity.

87. <https://developers.elementor.com/>

88. <https://testing-library.com/>

Their popularity is high! Per the chart above close to every fourth document uses `data-*` attributes. But the overall data show that 88% of documents use at least one `data-*` attribute. That's quite some adoption.

Social markup

Last year's edition introduced a section on social markup⁸⁹, special markup which makes it easier for social platforms to identify and display the respective metadata. Here's the 2022 update:

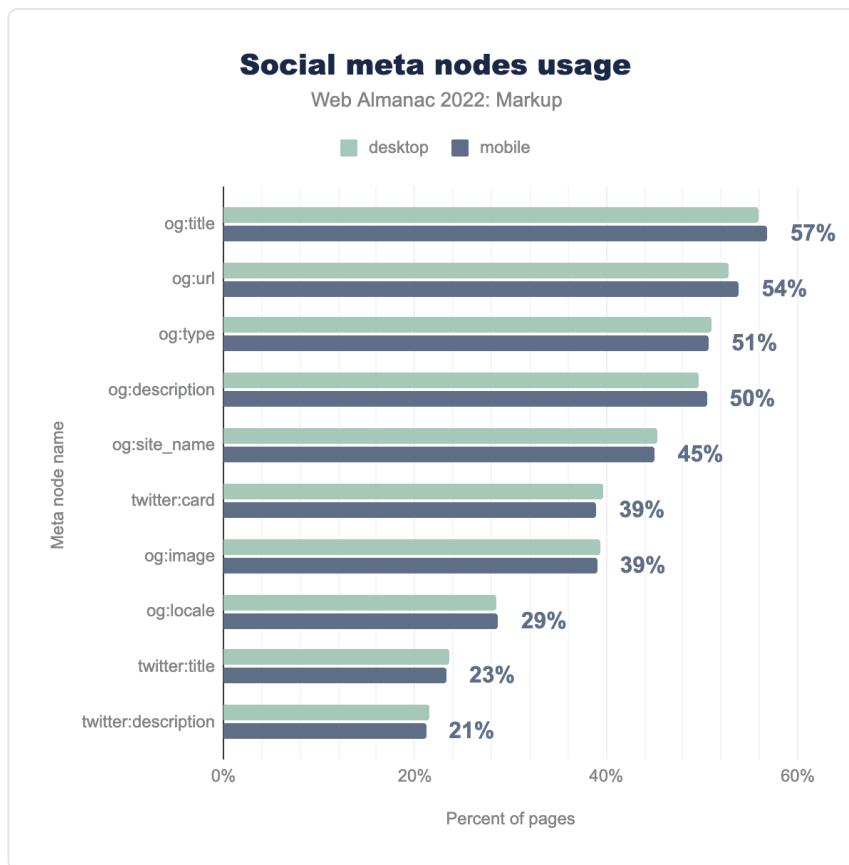


Figure 3.18. Social meta nodes usage.

Do you need all of this metadata? That depends on your requirements. But if these requirements are about showing title, description, and image, you don't seem to need nearly as

89. <https://almanac.httparchive.org/en/2021/markup#social-markup>

much. You may be able to do with `twitter:card`, `og:title`, `og:description` (hooked up to standard `description` metadata), and `og:image`. The author and many others have described options for minimal social markup⁹⁰.

Conclusion

This was a glance at HTML in 2022.

The conclusion is brief: Going from year to year, it's hard to say what important trends were started or reversed. Document size seems to keep growing—at least from 2020 to 2021 to 2022. The number of elements per page goes up every year too. There may be slightly more `alt` attributes now, but that's relative to itself and we can't tell whether more images now do have an appropriate `alt` attribute set—nor whether its text is really meaningful⁹¹.

But with all of this, the Web Almanac will help. We're going to look at HTML again—next year, the year after next, and the year after that. And we'll go into more detail again and we'll look back at more years.

What perhaps we'll also be able to do is to look at conformance too. Not everyone may care about this at this time in our field. But we're all professionals, and it seems at least relevant to know whether overall, we produce work that corresponds to the underlying standard(s)⁹². After all, this shouldn't be a chapter about fantasy HTML—it should be one about HTML that actually works. It's one of the most important web standards.

Author



Jens Oliver Meiert

[@j9t](https://twitter.com/@j9t) [j9t](https://github.com/j9t) <https://meiert.com/en/>

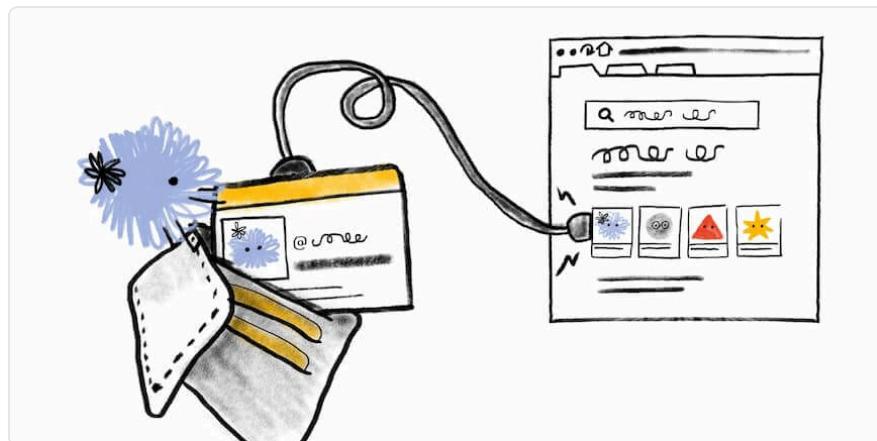
Jens Oliver Meiert is an engineering lead and author (*The Web Development Glossary*⁹³, *Upgrade Your HTML*⁹⁴), who works as an engineering manager at LivePerson⁹⁵. He specializes in HTML and CSS minimization and optimization. Jens regularly writes about the craft of web development on his website, meiert.com⁹⁶.

90. <https://meiert.com/en/blog/minimal-social-markup/>
 91. <https://html.spec.whatwg.org/multipage/images.html#alt>
 92. <https://html.spec.whatwg.org/multipage/>
 93. <https://leanpub.com/web-development-glossary>
 94. <https://www.amazon.com/dp/B094W54R2N/>
 95. <https://www.liveperson.com/>
 96. <https://meiert.com/en/>



Part I Chapter 4

Structured Data



Written by Andrea Volpini and Allen O'Neill

Reviewed by Rob Teitelman and Jono Alderson

Analyzed by Rick Viscomi

Edited by Jasmine Drudge-Willson

Introduction

This is the second year that the Web Almanac has included a chapter on structured data. Last year's content gave a solid grounding⁹⁷ in the concept of structured data, outlining the reason it exists, the most frequently used types, and how it benefits organizations. This year we compared data gathered in 2022 with the previous data from last year, so were able to monitor trends that occurred within that period.

Despite many advances in machine learning and in particular the field of "natural language progressing", data still needs to be presented in a machine-readable format. Structured data assists in information discoverability in web search, data linkage and archival purposes. By implementing structured data on websites, engineers and web content creators facilitate:

- making website data more widely available for automated discovery and linking

97. <https://almanac.httparchive.org/en/2021/structured-data#key-concepts>

- the open availability of data for public research
- ensuring the quality of the organization's data is maintained when the data leaves its origin

Organizations of all sizes and types want their content to be discovered on the web. Search engines such as Google and Bing emphasize data discoverability by promoting the use of structured data. From an SEO point of view, it is advantageous to present data in an easy to find and parse manner. Some of these advantages will be discussed in the use cases and key concepts sections within this chapter.

Last year's introduction⁹⁸ pointed out that "when machines can reliably extract structured data, at scale, we enable new and smarter types of software, systems, services and businesses". This year's chapter includes sections that explore recently published research on structured data, open source frameworks and tools that assist the generation of high-quality structured data.

This year we provide the first year over year comparison of metrics such as the presence of different structured data types as well as the growth of those structured data types, and examines the evolving benefits of using structured data. Having a baseline of data from 2021 allows us to gain insights into how the use of structured data has changed over the intervening period and observe interesting trends, for example the growth of TikTok in the period.

Data caveats

Structured data can appear in many forms, and may be more visible in certain domains, and their corresponding websites, over others. For example, compare a news website with an ecommerce website. In general, a news site shows the most important breaking news on its home page, therefore the structured data relating to the news articles may be present on the main website landing page attached as data-snippets to the individual article headlines. In comparison, structured data in ecommerce pertains to individual products and, as such, is mostly present within a website's product catalog itself, and in many ways, "hidden" from a high level search of the main navigation and promotional parts of the website. This is the key caveat that we need to be aware of in relation to the structured data chapter and report.

Due to the fact that the technology used to harvest data from websites only scratches the surface of sites (ie: the home pages), and does not go into depth on a full crawl of the site, we are unable to get a full picture of the extent of structured data usage in sites where such data is by necessity, contained deep within the site. In future years we hope to take a sample of sites across different domains and go deep to rectify this issue and give additional insight into

98. <https://almanac.httparchive.org/en/2021/structured-data#introduction>

domain-specific use of structured data.

The high level caveats from last year's chapter still remain, namely:

- **Auto-generated structured data:** This is where technologies such as content creation systems auto-generate structured data snippets based on templates. In this case any template-based error will inevitably populate across all data presented.
- **Data format overlaps:** Structured data can be presented in a number of different ways, including JSON-LD, RDF etc. This means that we may see overlap, for example, between a Facebook meta tag and the same tag presented in a different manner in the RDFa section. As analysis is tightly based on queries created for the baseline in 2021, we expect the impact of cleaning/normalization and data flattening should carry through for like analysis.

Key concepts

As structured data is a rich and complex area, it is important to explore and explain some key concepts of the topic before diving head-first into further analysis.

Linked data

By adding structured data to web pages, and providing URI links to the entities the pages contain/reference, we create *linked data*. This structured data is then interlinked, making it more useful through semantic queries.

Adding linked data to describe web page content enables machines to treat web pages as databases. At a large scale, this contributes to the semantic web⁹⁹. The semantic web links data together through The *Resource Description Framework (RDF)*. This is a framework for representing information on the web using URIs to define entities and the relationships between them.

A relationship between entities in the RDF data model is known as a *semantic triple*. With a semantic triple¹⁰⁰ (or just *triple*), we can codify a statement about data. These expressions follow

99. https://en.wikipedia.org/wiki/Semantic_Web

100. https://en.wikipedia.org/wiki/Semantic_triple

the form of subject–predicate–object (e.g., “Allen knows John”).

To be able to retrieve and manipulate RDF data, we can use an RDF Query Language such as SPARQL¹⁰¹, the standard RDF query language.

As will be discussed later, this semantic web creates many opportunities for business and technology.

Open data

Linked data may also be *open data*, described as *Linked Open Data*. Open data, as the name implies, is data that is openly accessible to anyone for any purpose. This data is licensed under an open license.

Open data is the first of the 5 stars of open data¹⁰², a deployment scheme suggested by Tim Berners-Lee. According to the open data handbook¹⁰³, to score the maximum five stars, data must (1) Be available on the Web under an open license, (2) Be in the form of structured data, (3) Be in a non-proprietary file format, (4) Use URIs as its identifiers, (5) Include links to other data sources (see data linking).

While structured data is the second star in the 5 star open data plan, linked data should fulfill requirements for all 5 stars of open data.

Semantic search engines, rich results and beyond

A semantic search engine is one which performs semantic search¹⁰⁴. This is different from lexical search where search engines look for exact or close matches to words or strings of text.

Semantic search aims to understand the user’s intent and the context of the search terms in order to improve the accuracy of search. An example would be a structured data entity of “local business: hairdresser” versus “TG Locks n Lashes”; the latter is a business name, and while it tells the creative name of the hair salon as a key-word, it does little to help the search engine to understand what the business does. By using structured data, the website can better help the search engine understand the context of its information, and thus enable the engine to offer better search results in the context of the query asked by the search user. Google and Bing are excellent examples of semantic search engines.

Google uses semantic search technologies to serve relevant information from the Google Knowledge Graph¹⁰⁵ which is a knowledge base used to serve search results in an infobox. This

101. <https://www.w3.org/TR/sparql11-query/>

102. <https://5staridata.info/en/>

103. <https://opendatahandbook.org/>

104. https://en.wikipedia.org/wiki/Semantic_search

105. <https://blog.google/products/search/introducing-knowledge-graph-things-not/>

infobox is known as a knowledge panel¹⁰⁶, and can be seen in many results. This knowledge box can be enabled or enhanced by structured data.

Another search result that is made possible by structured data combined with linked data is the rich result¹⁰⁷. These results display richer features in search results, and come in the form of Events, FAQs, How-tos, Job listings and many more¹⁰⁸. Implementing structured data to make web pages eligible for rich results could increase clickthrough rate¹⁰⁹. The image below illustrates how structured data with business details for a Hair Studio allows the search engine to easily extract and display information about the business, highlighting it and optimizing SEO.

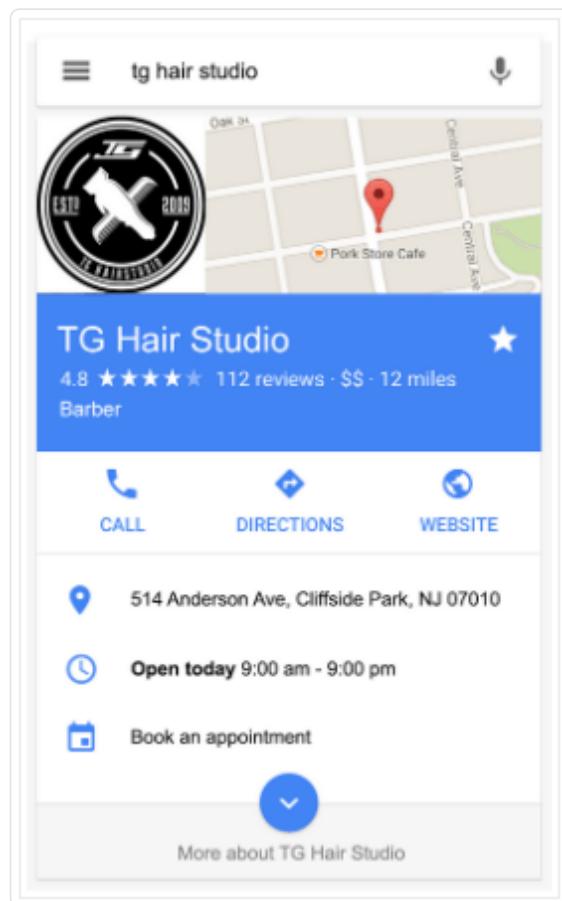


Figure 4.1. Structured data surfaced in a web search.

106. <https://support.google.com/knowledgepanel/answer/9163198>

107. <https://developers.google.com/search/docs/advanced/structured-data/search-gallery>

108. <https://developers.google.com/search/docs/advanced/structured-data/search-gallery>

109. <https://www.searchenginejournal.com/how-important-is-structured-data/257775/>

Beyond knowledge panels and web page rich results, structured data can also enable answers to factual queries¹¹⁰ in search. A factual query search can get multiple signals from different structured data sources and support more precise results. Here, structured data implementation, and the technologies that allow for it, provide faster and more reliable access to information in order to improve user experience.

The combination of SEO importance, higher click-through rates, improved user experience and machine-readable data being accessible for analysis illustrate significant benefits to implementing structured data. Understanding these key concepts will help both content providers and technical personnel who construct sites how to implement better navigation and understand the function of automated data consumption from web pages.

Structured Data research

For this year's chapter we were interested in investigating what—if any—academic research has been carried out in the area of structured data, or if structured data was documented as being used to assist in development of state-of-the-art technologies and services.

To look for published research, we used academic search tools such as Google Scholar¹¹¹, ConnectedPapers¹¹² and University-based citation databases. We not only looked for recent publications, but also older research that continues to be cited.

The results of our search showed that there is not a lot of highly cited recent work conducted into generating, managing and building structured web data. However, research on the application of structured web data (“The Semantic Web”¹¹³) like knowledge graphs, recommendation engines, information retrieval, chatbots and explainable AI has been conducted in the past twelve months and continues to grow.

Web structured data shares a synergetic relationship with the field of machine learning by providing consistent data with appropriate Uniform Reference Indicator (URI) vocabulary which can be used to generate machine readable labels¹¹⁴. Our searches and background reading have shown that structured data has considerably reduced the work and time input to generate high quality web data for training machine learning algorithms.

On a practical level, we highlight three areas that structured data has improved:

- Knowledge graphs

110. <https://gofishdigital.com/blog/answering-questions-structured-data/>

111. <https://scholar.google.com/>

112. <https://www.connectedpapers.com/>

113. https://en.wikipedia.org/wiki/Semantic_Web

114. <https://developers.google.com/machine-learning/crash-course/framing/ml-terminology>

- Question Answering over Knowledge Graphs
- Explainable AI

Knowledge graphs

Structured web data provides fixed vocabularies between entities and objects as a domain-specific language, which are generally stored in a RDF format. Knowledge graphs using RDF have proven to be great tools for querying relationships between entities. As an example, Wikidated 1.0 is an evolving knowledge graph which uses web structured data to store Wikipedia's revision history. Its corresponding paper¹¹⁵ talks through the process of aggregating revisions to a page as a set of additions and deletions of the RDF tuple. The authors have open sourced their method to convert wikipedia dumps into knowledge graphs. Applied research carried out by doordash engineering demonstrates that using knowledge graphs can dramatically improve search performance¹¹⁶.

Question Answering over Knowledge Graphs

Question answering systems enable end users to find answers to their questions. When built upon a knowledge graph, a question answering system makes it possible to access the rich and structured data stored in knowledge graphs. Query languages such as SPARQL¹¹⁷ are often used to query the information stored as RDF triples in knowledge graphs.

However, writing SPARQL queries can be tedious and challenging for end-users. Therefore, natural language questions (NLQs) are an attractive solution that allows overcoming the numerous complexities of querying knowledge graphs. This work proposes a KG-based question answering system (KGQAS) that consists of two main phases: 1) an offline phase, and 2) a semantic parsing phase.

While the offline phase aims to convert natural language questions into formal query patterns in a semi-automated way, the semantic parsing phase leverages machine learning to build a prediction model. The model is trained on the output of the first phase. It enables predicting the most appropriate query pattern for a given question. For evaluation, SalzburgerLand KG is used as a practical use case. It's a real-world knowledge graph that is built using the schema markup vocabulary and its primary focus is structured data automation that describes touristic entities of the region of Salzburg, Austria.

115. <https://arxiv.org/abs/2112.05003>

116. <https://doordash.engineering/2020/12/15/understanding-search-intent-with-better-recall/>

Explainable AI

Explainable AI focuses on explaining decisions of an AI model. Most AI models are not openly available to the public, and so do not provide rationale for the decisions they make. Owing to knowledge graphs built on top of semantic web; harder to find relationships between entities can be found. These are then used as 'ground truth' to trace back the results of the model. The most common approach is to map network inputs or neurons to classes of an ontology or entities of a web structured data.

References:

- Knowledge graphs: Wikidata 1.0: An Evolving Knowledge Graph Dataset of Wikidata's Revision History^{[118](#)}
- Question Answering Over Knowledge Graphs: Question Answering Over Knowledge Graphs: A Case Study in Tourism^{[119](#)}
- Explainable AI using structured data: Semantic Web Technologies for Explainable Machine Learning Models: A Literature Review^{[120](#)}

Open source use of Structured Data

Three projects of note that rely heavily on the use of structured data are the following:

- **Open Source Metadata Framework (OMF)** - The OMF aims to collect data about Open Source documentation / metadata which are typically stored in a structured data format that will be used to describe the documentation. The idea is that the OMF will act as a sophisticated card catalog type of system for the numerous Open Source documentation projects that exist.
- **DBpedia** is a set of datasets, tools and services related to structured web data. It contains more than 228 million freely-available entities to date. The main DBpedia Knowledge Graph encompasses clean data from Wikipedia. DBPedia is available in all supported Wikipedia languages and averages over 600k file downloads per year. Some open source tools that are built on top of DBpedia provide data access, versioning, quality control, ontology visualization and linking infrastructures.

118. <https://arxiv.org/abs/2112.05003>

119. <https://ieeexplore.ieee.org/abstract/document/9810255>

120. https://www.researchgate.net/profile/Mathias-Pfaff/publication/336578867_Semantic_Web_Technologies_for_Explainable_Machine_Learning_Models_A_Literature_Review/links/5daaf99a6fdccc99d91d120/Semantic-Web-Technologies-for-Explainable-Machine-Learning-Models-A-Literature-Review.pdf

- **Wikidata** stores structured data from Wikimedia projects like Wikipedia. It is a document-oriented database, which focuses on storing structured web data.

References:

- Open Source Metadata Framework¹²¹
- DBpedia¹²²
- WikiData¹²³

Use cases

The implementation of structured data is widely beneficial in numerous areas, some of which will be focused on in this section. It is important to note that many of these areas are overlapping, such is the nature of linked and structured data.

Data linking

Having structured and linked data, while using identifiers to designate places, events, people, concepts, etc, the data can be cited by other data sources and therefore make their metadata descriptions more accessible. This data is then more shareable and reusable.

With data linking, we collect information from different sources to create richer and more useful data. This is possible thanks to structured data, whose global, unique identifiers allow machines to read and understand the relationship between different types of data. This has the use of creating a more connected web of relationships.

Search Engine Optimization & discoverability

Search engine optimization (SEO¹²⁴) is the area focusing on building the content of a web page so that it has better results from search engines. Naturally, this is highly important for discoverability as a successful implementation of SEO may allow for a page to appear higher on the search engine results page (SERP¹²⁵). The SERP is where the titles, URLs, and meta descriptions are displayed from a search query.

121. <http://www.ibiblio.org/osrt/omf/>

122. <https://en.wikipedia.org/wiki/DBpedia>

123. <https://en.wikipedia.org/wiki/Wikidata>

124. https://www.webopedia.com/definitions/seo/#How_does_SEO_work

125. <https://www.webopedia.com/definitions/serp/>

By adding structured data to web pages, we can optimize a web page for search engines, as well as have extra content visible from the SERP. This extra content can come in many forms, some of which has been discussed previously, namely Knowledge Panels, Rich Snippets and Related Questions.

Having this added discoverability, enabled by structured data, is essential for increasing traffic to a web page from search engines. It follows that businesses and ecommerce pages would find great value in these technologies, which will be discussed in the following section.

Ecommerce & business

The implementation of structured data for ecommerce web pages is incredibly beneficial for those involved with the business. There are numerous structured data types which are widely used for these businesses for SEO.

LocalBusiness¹²⁶ is a structured data type which may return a Google knowledge panel with details entered in the structured data type during relevant search queries (e.g. “popular restaurants in Dublin”). This type also may have business hours, different departments within a business, reviews for the business, which could all be returned from a maps app search query as well.

Product¹²⁷, the structured data type, works similarly to LocalBusiness in that it allows for a search query to return rich results. These results can include price, availability, reviews, ratings, and even images in the search results. These added elements can make the product far more likely to receive attention from the search. Product attributes can help link products together and better respond to search queries, increasing discoverability.

These are just a couple of examples of use cases for structured data in ecommerce, but there are many more structured data types¹²⁸ that an ecommerce page can benefit from implementing.

A year in review

Structured data is underpinned by formats and standards that describe a meta-level schema into which publishers can fit and present data in a pre-defined manner. RDFa, OpenGraph, JSON-LD and other established formats have been used in the analysis for this chapter.

126. <https://developers.google.com/search/docs/advanced/structured-data/local-business>

127. <https://developers.google.com/search/docs/advanced/structured-data/product>

128. <https://developers.google.com/search/docs/advanced/ecommerce/include-structured-data-relevant-to-ecommerce>

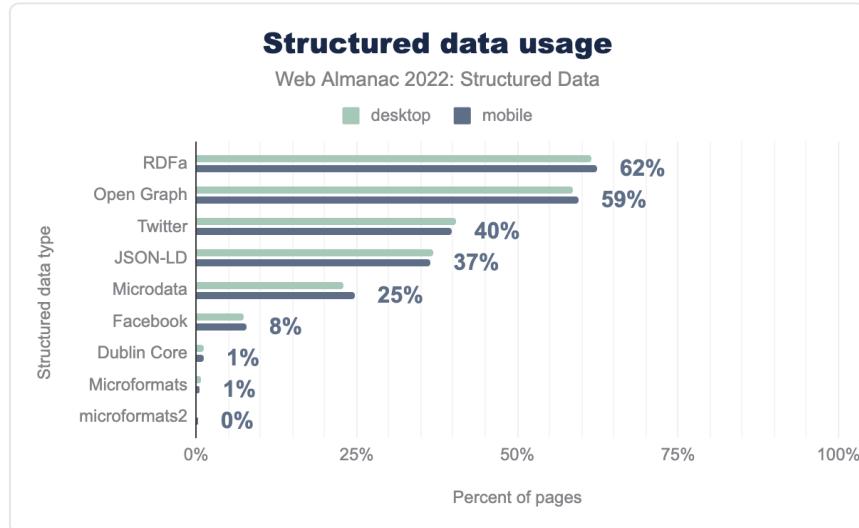


Figure 4.2. Structured data types

RDFa and Open Graph remain in the majority with 62% and 57% of mobile pages, respectively. Structured data types are seen consistently across mobile and desktop pages, with Microformats and microformats2 differing the most from other structured data types we examined in this chapter. Microformats are 86% as prominent on mobile pages, whereas microformats2 are 171% as prominent on mobile pages. These two structured data types make up a small percentage of those found in our set.

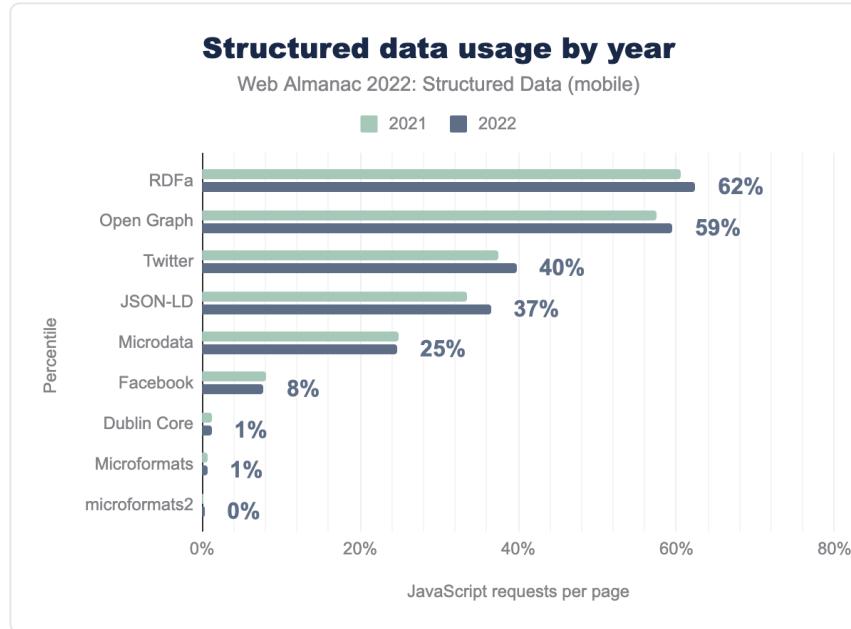


Figure 4.3. Structured data usage by year on mobile

A general increase in these widely-used structured data types can be seen, including Twitter meta tags (which has increased from 37% to 40%) and JSON-LD (which has increased coverage from 34% overall in 2021 to 37% overall in 2022). There is a slight decrease in usage for some of the less prevalent structured data types such as Microdata, Facebook meta tags, Dublin Core and Microformats. Desktop movements were very similar.

The below table lists the major changes to structured data formats in the last year. Only types with changes have been listed.

Data type	Change
RDFa	<p>Although there are no changes in the base format of RDFa, version 3 of the _Data Catalog Vocabulary (DCAT) contained a significant update. DCAT is an “RDF vocabulary designed to facilitate interoperability between data catalogs published on the Web”. This is significant due to the increased availability of open datasets on the web. Being able to describe the entire contents of a dataset greatly increases the discoverability, and thus usefulness, of a public dataset and makes federated search and distribution more likely.</p> <p>References:</p> <ul style="list-style-type: none"> • DCAT: https://www.w3.org/TR/2022/WD-vocab-dcat-3-20220510 • Google dataset search engine¹²⁹ • Google dataset structured data format guide¹³⁰
JSON-LD	<p>Updates and additions in the past year were minor. Of these, most were related to maintenance and minor expansion of context, for example “adding OnlineBusiness as a subtype of Organization and OnlineStore as a subtype of OnlineBusiness”.</p> <p>References:</p> <ul style="list-style-type: none"> • https://schema.org/docs/releases.html

Figure 4.4. Changes between 2021 and 2022 in data type formats.

Overall there has been little change in the definitions of the major data types as the table outlines, however some formats have been advanced in specific domains.

Let's delve a little deeper into each type.

129. <https://datasetsearch.research.google.com/>
 130. <https://developers.google.com/search/docs/advanced/structured-data/dataset>

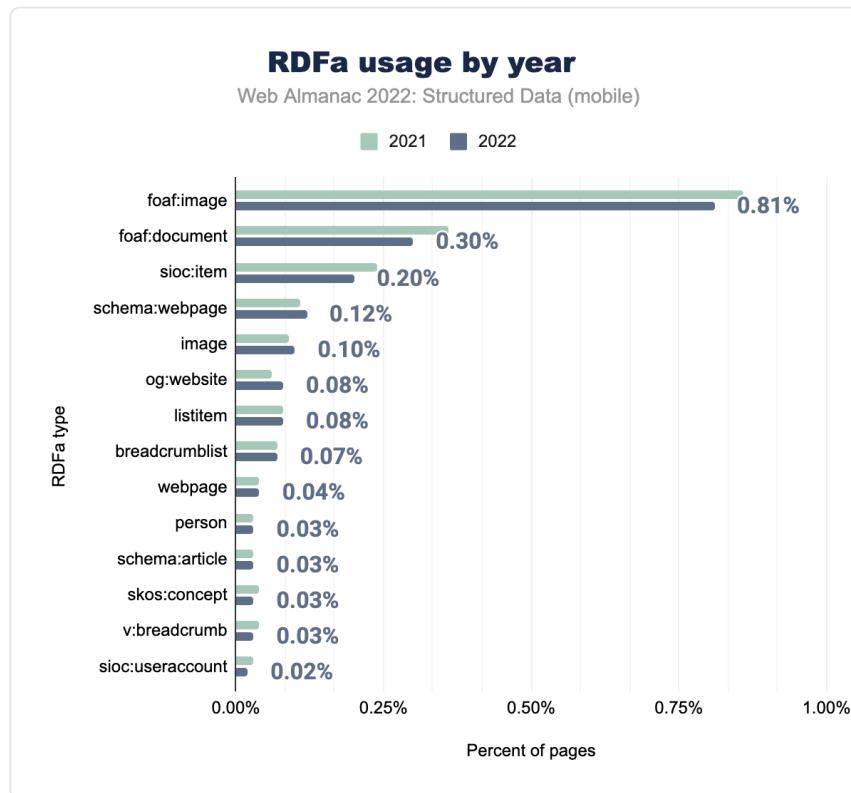
RDFa

Figure 4.5. RDFa usage by year on mobile

When evaluating the types of RDFa, `foaf : image` remains present on more pages than any other type, though it has shown a decrease in the percent of pages in our set since 2021. This applies to the next two types, `foaf : document` and `sioc : item`, with small decreases in usage. Many of the other types show a slight increase in usage, as RDFa has seen as a whole.

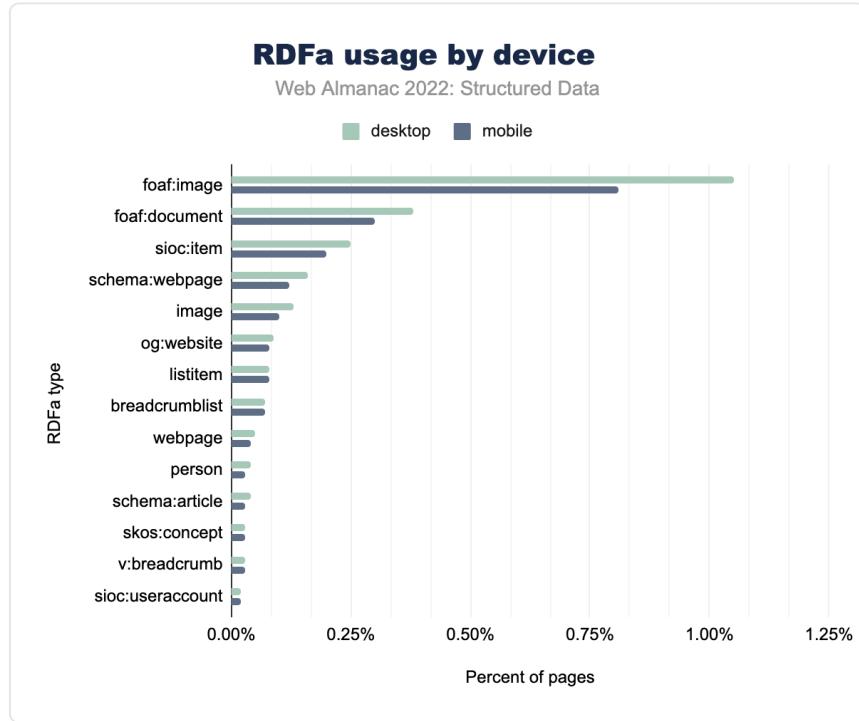


Figure 4.6. RDFa usage by device

RDFa remains more prominent on desktop with `foaf:image` appearing on 1% of desktop pages, compared to 0.81% on mobile pages. Other RDFa types saw a slight increase in appearance on desktop pages over mobile, with the exception of `og:website` reaching ahead with 0.08% on mobile pages and 0.07% on desktop pages.

Dublin Core

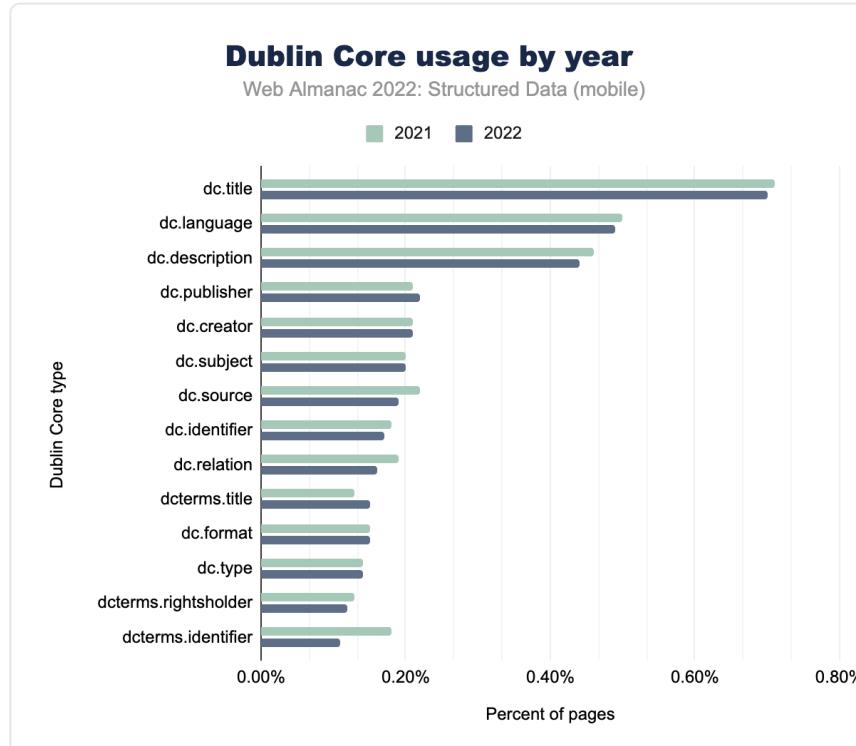


Figure 4.7. Dublin Core usage by year (mobile)

Dublin Core attribute type usage remains very similar across the most prominent attribute types. A notable exception is `dcterms.identifier`, going from 0.11% in 2021 to 0.18% in 2022 for mobile pages. Though small in percentage, this totals to a usage count of nearly 15,000 in our set. This increase was also seen for desktop pages, though not as substantial, going from 0.14% in 2021 to 0.18% in 2022.

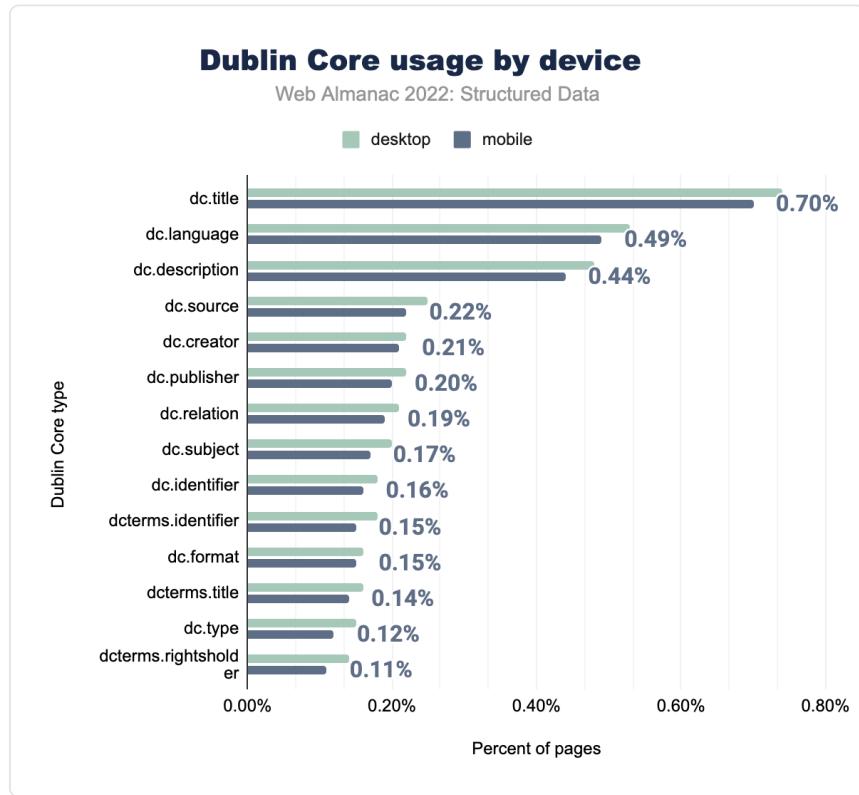


Figure 4.8. Dublin Core usage by device

Other than that, Dublin Core types are similar between mobile and desktop pages, sharing the same slight increase in appearances compared to the previous year.

Open Graph

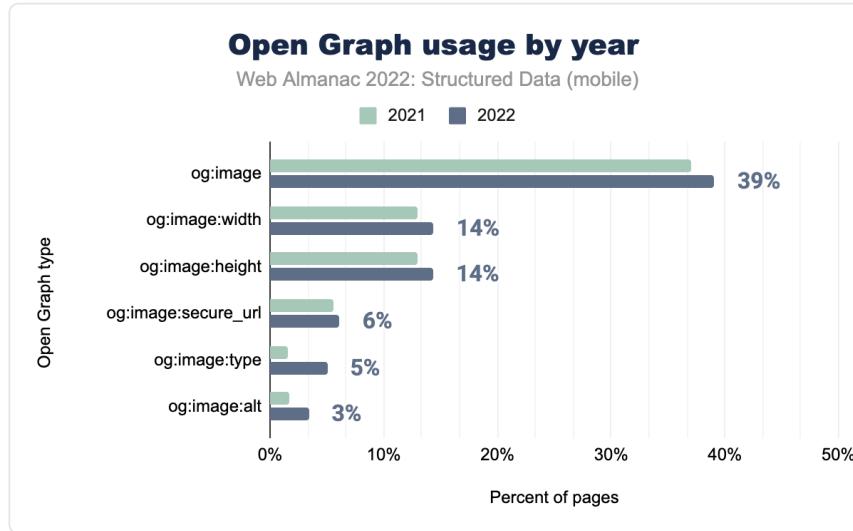


Figure 4.9. Open Graph usage by year (mobile)

Open Graph tags have seen a widespread increase in use. The most common of these tags is `og:title` appearing in over half of all pages in our set, joined by `og:url` and `og:type`. Most of these increases are small, with `og:image:type` as an exception which more than tripled on mobile pages since 2021. This is matched by desktop, going from 1.6% to 5.4% over the course of the year.

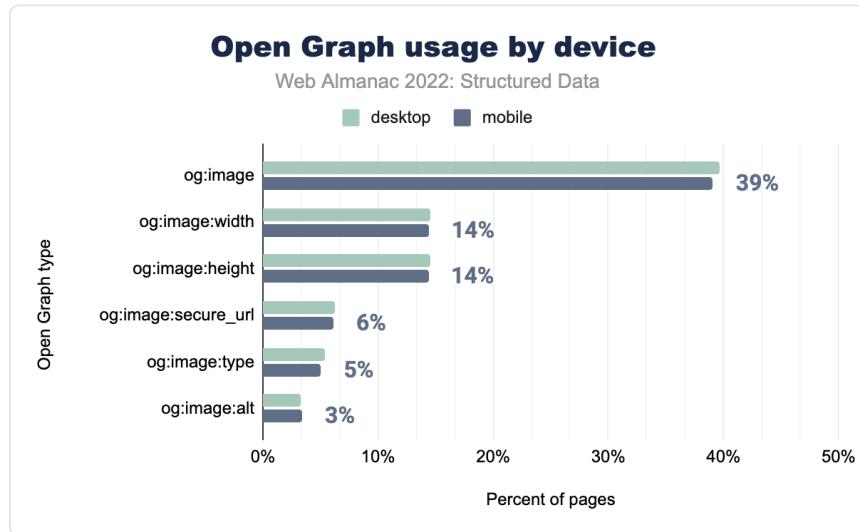


Figure 4.10. Open Graph usage by device

We have seen an increase in use for each Open Graph type in the top 10 for both mobile and desktop, resulting in Open Graph's relative growth of 1.5% since 2021.

Twitter

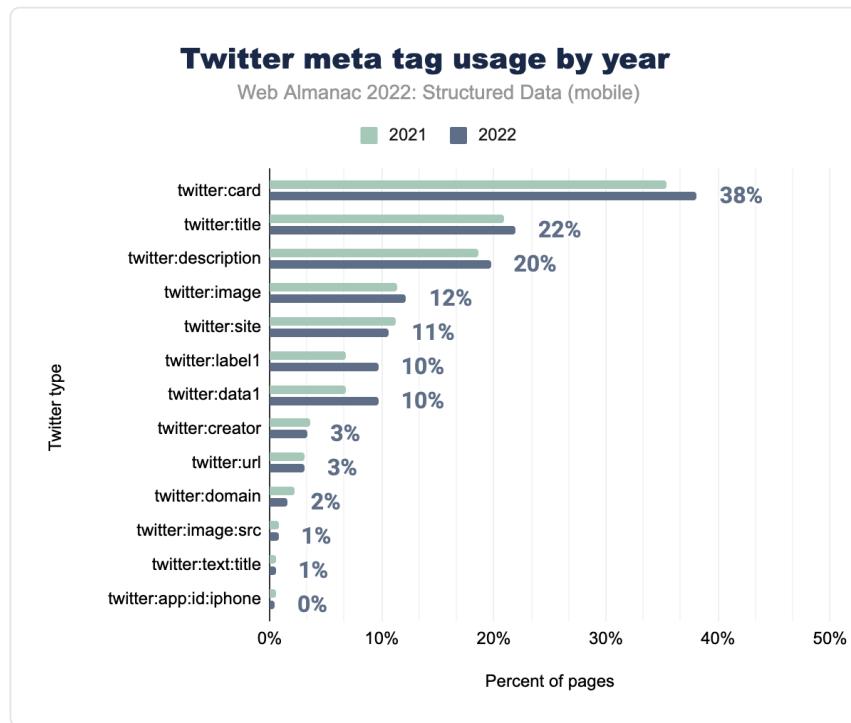


Figure 4.11. Twitter meta tag usage by year (mobile)

Twitter meta tags once again follow the pattern of a general increase in usage, more specifically in the common tags of `twitter:card`, `twitter:title`, `twitter:description` and `twitter:image`. A notable increase can be seen for `twitter:label1` and `twitter:data1`, both at 7% in 2021 to 10% in 2022 for mobile pages.

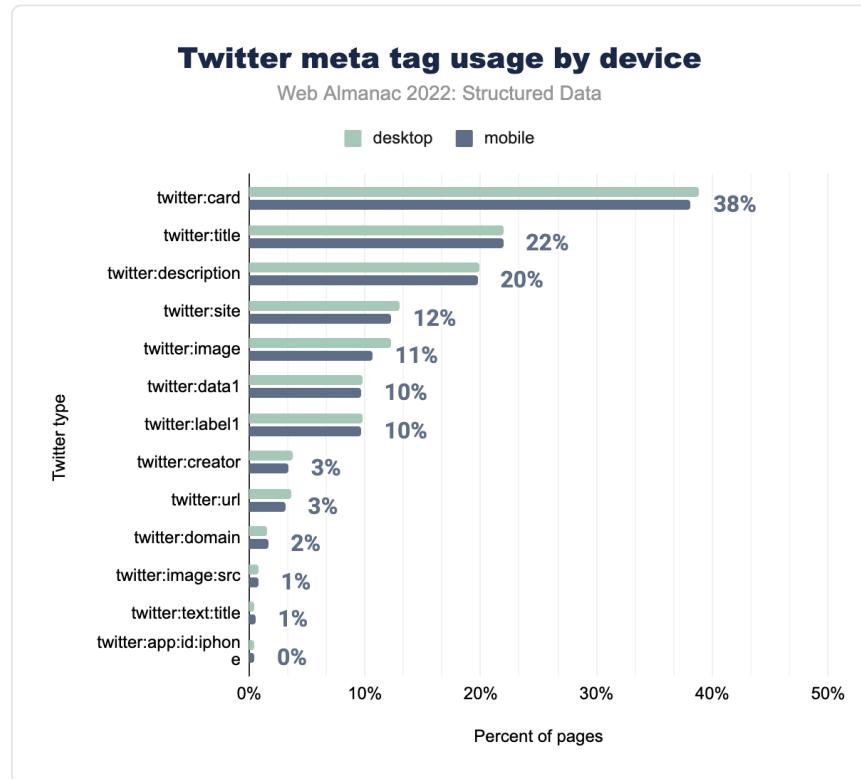


Figure 4.12. Twitter meta tag usage by device

Twitter meta tags such as `twitter:site` and `twitter:image` have a larger presence on desktop sites, though the majority of these meta tags share the same prevalence between mobile and desktop, as well as year-to-year. Some of the less common tags saw a slight decrease in use this year, though Twitter meta tag usage maintains an overall increase from last year.

Facebook

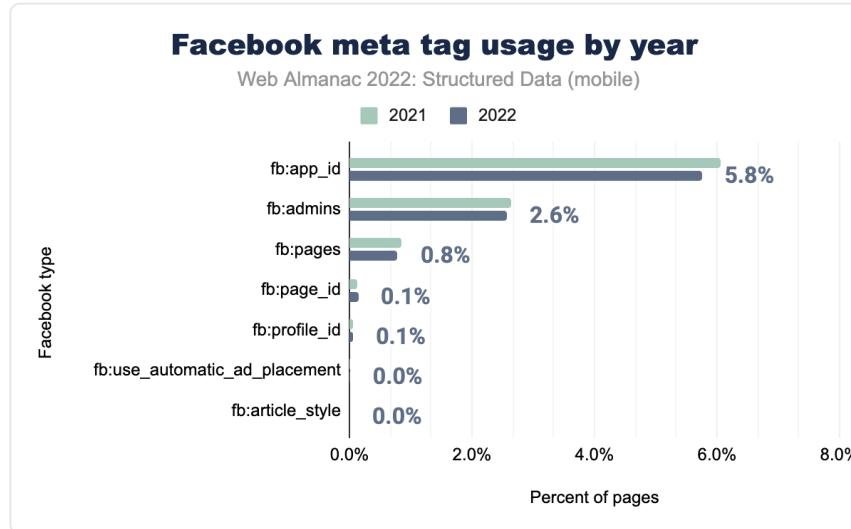


Figure 4.13. Facebook meta tag usage by year (mobile)

Out of all of the Facebook tags here, there are only three with significant numbers of appearances. These are the same as the top three in 2021, namely `fb:app_id`, `fb:admins` and `fb:pages` at 5.8%, 2.6% and 0.8% on mobile, all a slight decrease from last year.

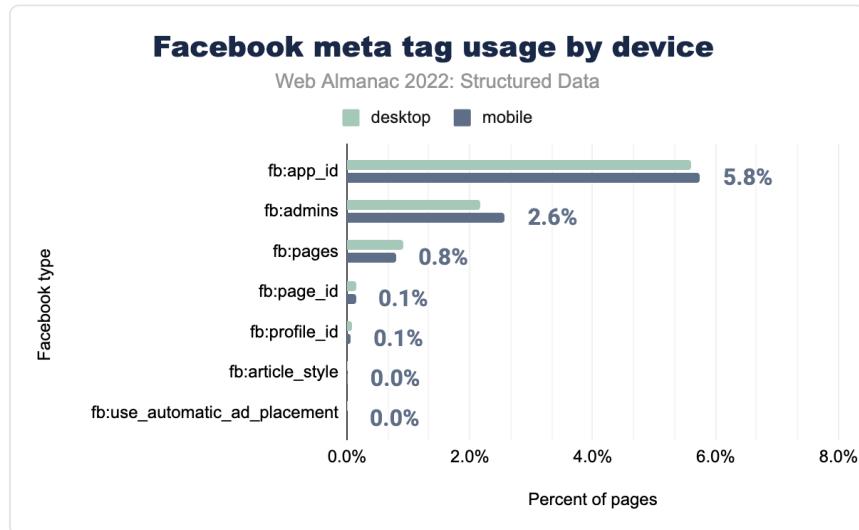


Figure 4.14. Facebook meta tag usage by device

This is true for desktop pages too, with the exception of `fb:pages` at a slight increase from 0.90% in 2021 to 0.92% in 2022. The meta tag `fb:pages_id` sees a slight increase on mobile and desktop pages alike, but overall facebook meta tag usage has seen a decline for both mobile and desktop pages since last year.

Microformats and microformats2

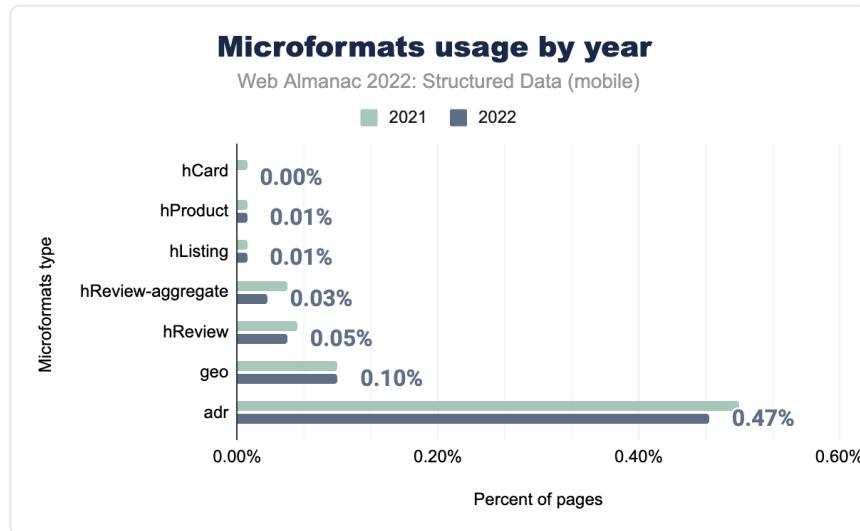


Figure 4.15. Microformats usage by year (mobile)

Microformats have remained very similar in usage numbers since 2021, with `adr` (appearing on 0.47% of pages in our set) still being the most common on the list.

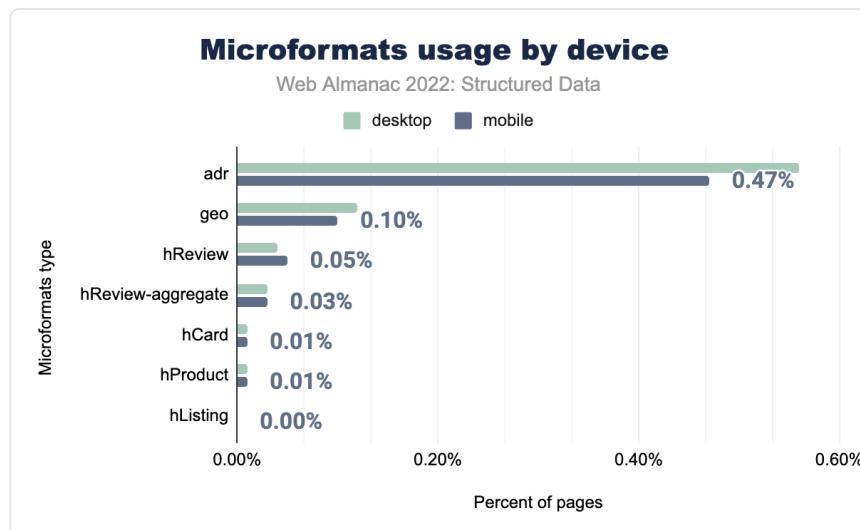


Figure 4.16. Microformats usage by device

Both mobile and desktop share a mix of increased and decreased usage between microformat types, though both averaging out to less than last year's numbers. Some types which factor into this decrease are `hReview` (going from 0.06% to 0.05% on mobile pages and 0.06% to 0.04% on desktop pages) and `hReview-aggregate` (going from 0.06% to 0.04% on both mobile and desktop pages).

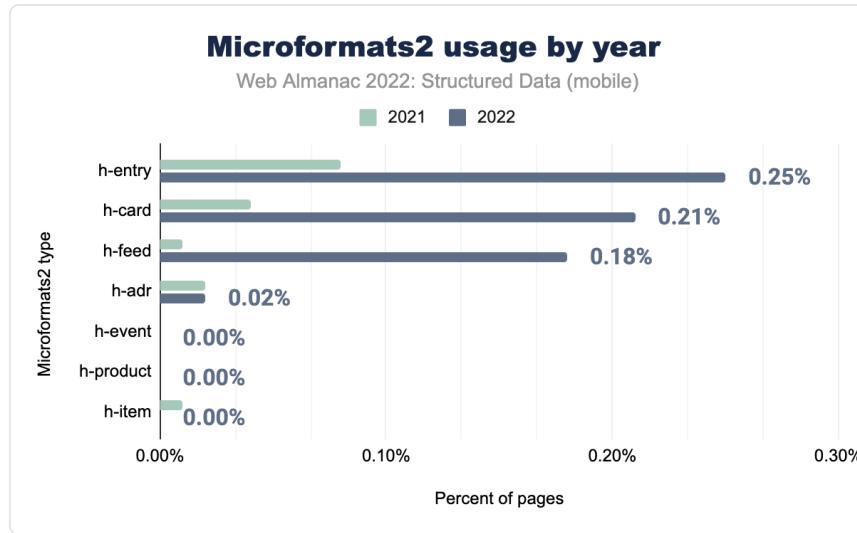


Figure 4.17. Microformats2 usage by year

Meanwhile, microformats2 attributes have skyrocketed since 2021. The properties of `h-entry`, `h-card` and `h-feed` have shown huge increases in our set of pages, which account for the fact that microformats2 attributes have almost tripled in our set since the previous year.

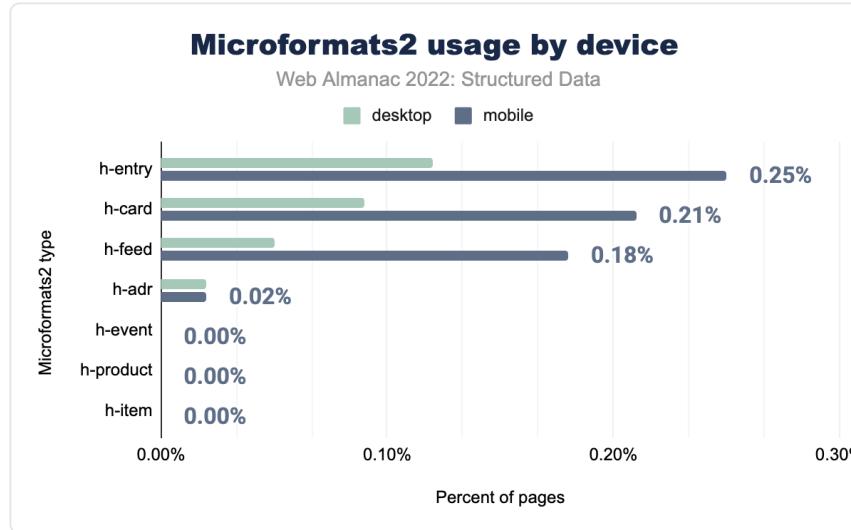


Figure 4.18. Microformats2 usage by device

This growth is seen more drastically on mobile pages, though desktop pages do follow the same pattern. Other than that, `h-adr` remains the exact same across both years and both platforms at 0.02% of pages.

Microdata

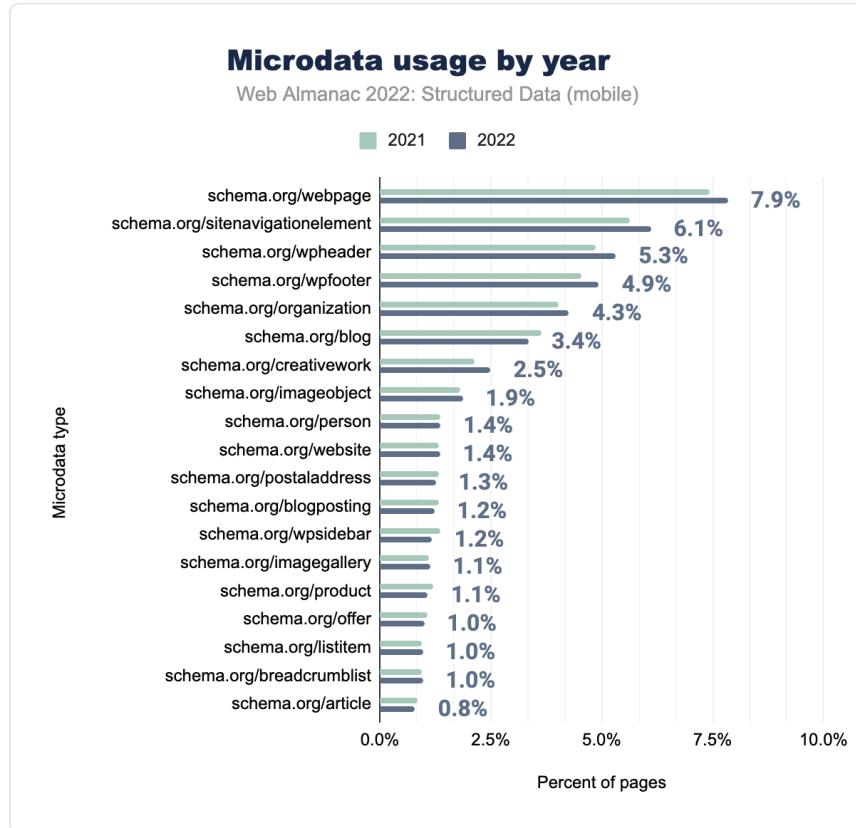


Figure 4.19. Microdata usage by year (mobile)

Most of the properties for Microdata have not seen much change, with a slight increase in some of the more common properties such as `webpage`, `sitenavigationelement` and `wpheader` appearing in 7.9%, 6.1% and 5.3% of mobile pages respectively.

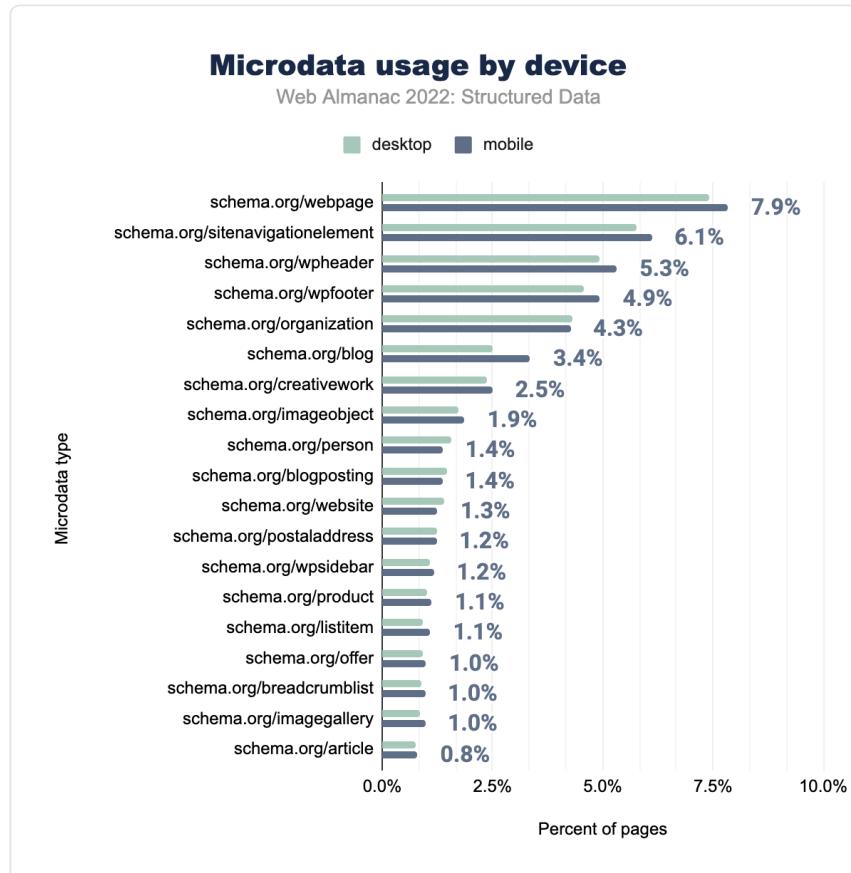


Figure 4.20. Microdata usage by device

These increases are common for desktop as well, with slight decreases elsewhere such as `wpsidebar` (going from 1.4% to 1.2% on mobile pages and going from 1.3% to 1.1% on desktop pages), resulting in minimum change over the last year for both mobile and desktop pages as a whole.

JSON-LD

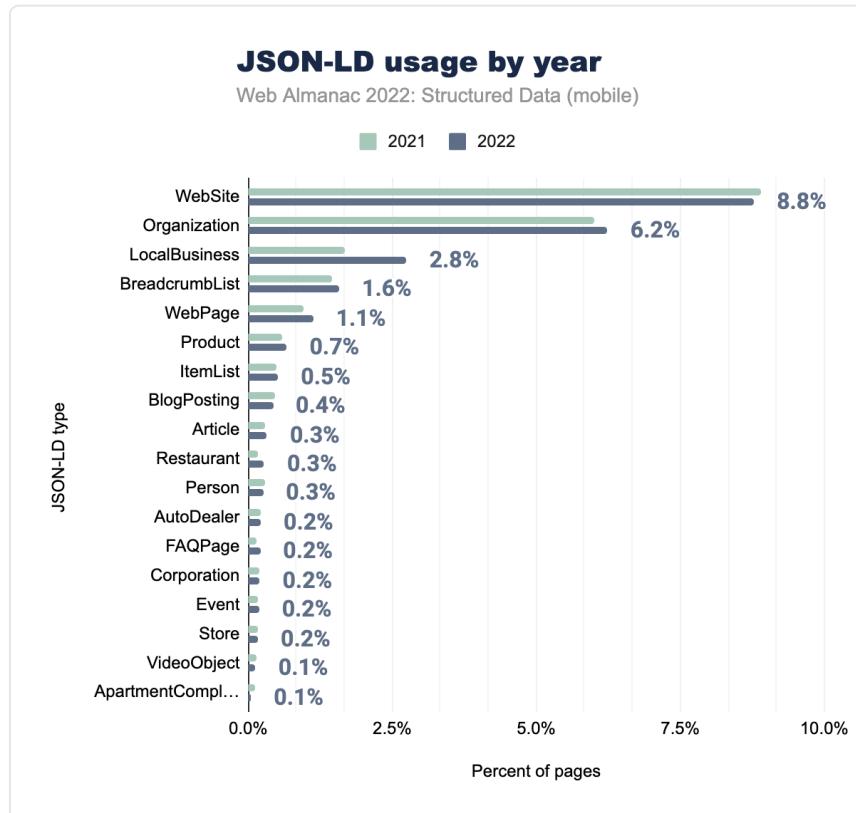


Figure 4.21. JSON-LD usage by year (mobile)

JSON-LD types continue to be mostly similar with a few notable increases over the previous year. Namely, these are `LocalBusiness` (which has increased to 2.8% of pages in our set) and `Restaurant` (which has increased to 0.3% of pages in our set).

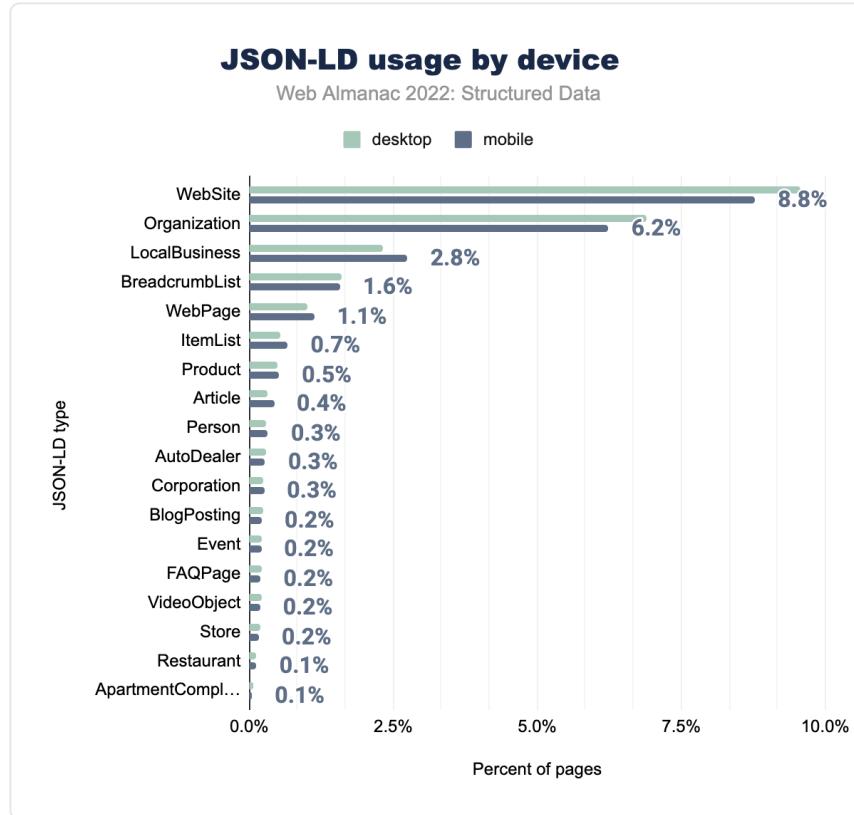


Figure 4.22. JSON-LD usage by device

These increases are enough to make JSON-LD types have the 2nd biggest positive change since 2021, going from appearing on 33.5% to 36.5% of mobile pages in our set and going from 34.1% to 36.9% on desktop pages.

JSON-LD Relationships

When evaluating JSON-LD, we can focus on the most recurring patterns of relationships among the different classes. More than with other syntaxes, JSON-LD expresses the value of graphs in structured data. An `Article`, for example, is frequently characterized by a linked `image` and the entity type `Person` to represent its *author*. Quite similarly, we would see that `BlogPosting` is also connected with `image` but as a frequent relationship with the `Organization` that serves as `Publisher`.

Some types are purely syntactic like `BreadcrumbList` that is used exclusively to connect

different items (`itemListElement`) of a site navigation's system or a `Question` that is typically linked with its answer (`acceptedAnswer`). Other elements deal with meanings: a `LocalBusiness` typically is linked to an `address` and to the opening hours (`openingHoursSpecification`).

With this analysis we want to share a birds-eye overview of the most common types of relationships between entities and the subtle differences between let's say `Article` and `BlogPosting`.

Here below we can see the common links between the different types, based on how frequently they occur within all structure/relationship values. Some of these structures are typically part of larger relationship chains.

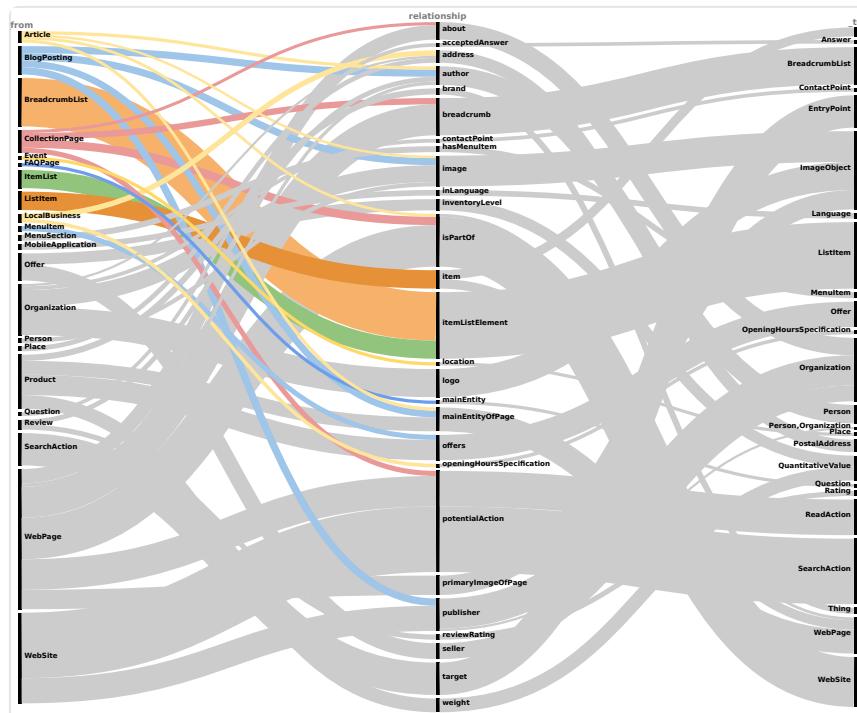


Figure 4.23. JSON-LD entity relationship as a Sankey diagram.

The analysis also provides an overview of the verticals behind these constructs: from news and media to e-commerce, from local businesses to events, and so on.

Here below we can see the same data interactively with the source attribute on the left and the target class on the right.

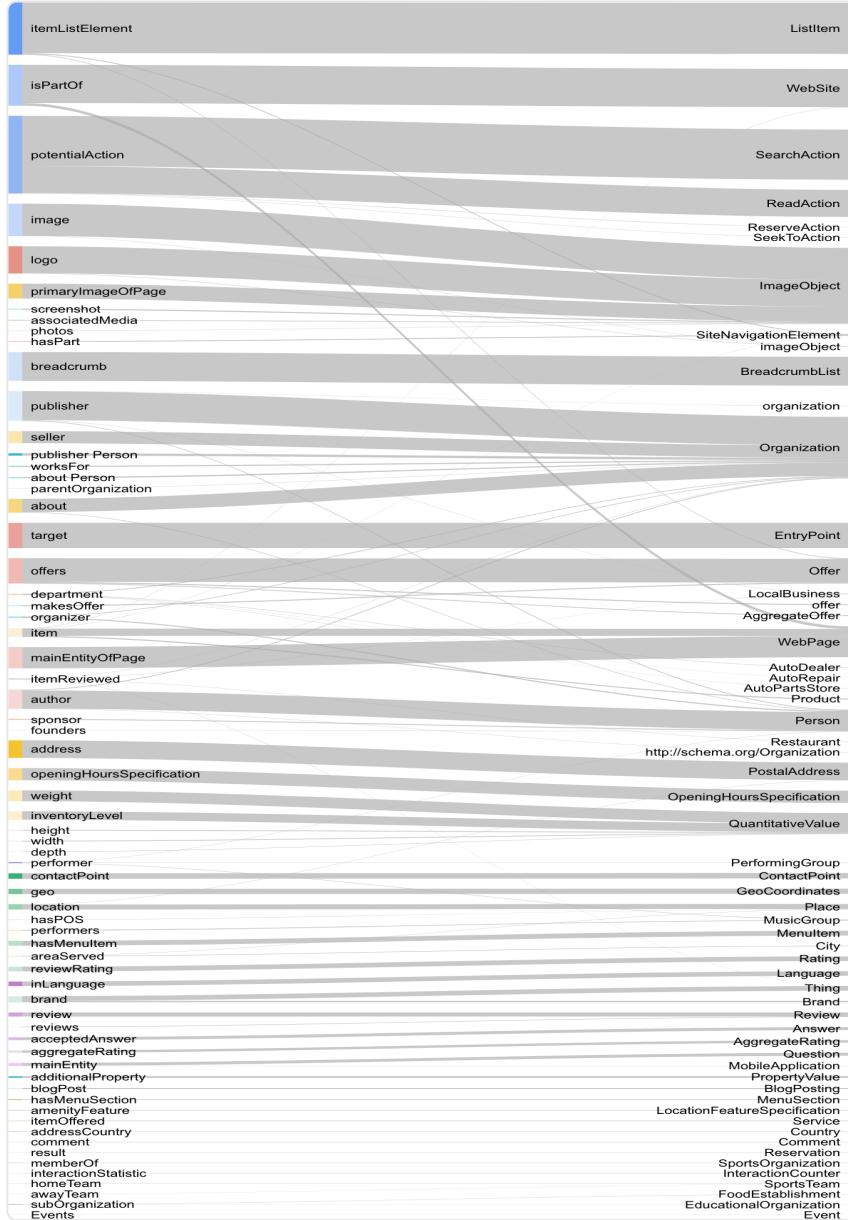
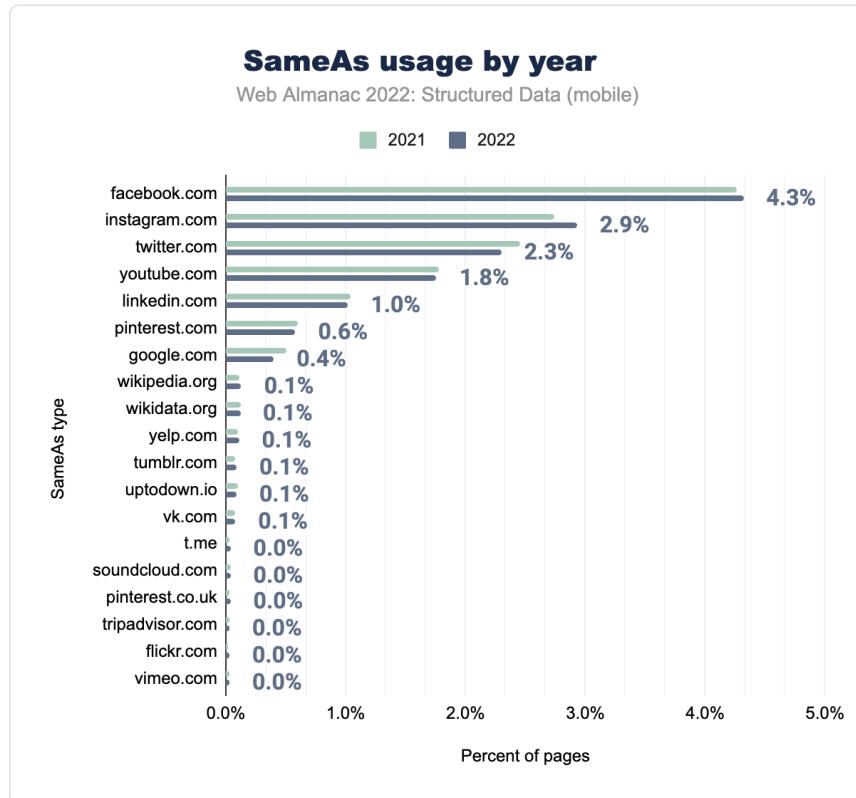


Figure 4.24. Sankey Chart.

The main limitation of this analysis is represented by the fact that we can evaluate relationship chains at the homepage level.

SameAsFigure 4.25. `SameAs` usage by year (mobile)

As was the case in 2021, the most common values of the `sameAs` property are social media platforms. These include `facebook.com` (at 4.32% on mobile and 4.94% on desktop), `instagram.com` (at 2.93% on mobile and 3.34% on desktop) and `twitter.com` (at 2.30% on mobile and 2.86% on desktop). The former two of which have seen a slight increase on mobile from 2021, with all 3 increasing on desktop.

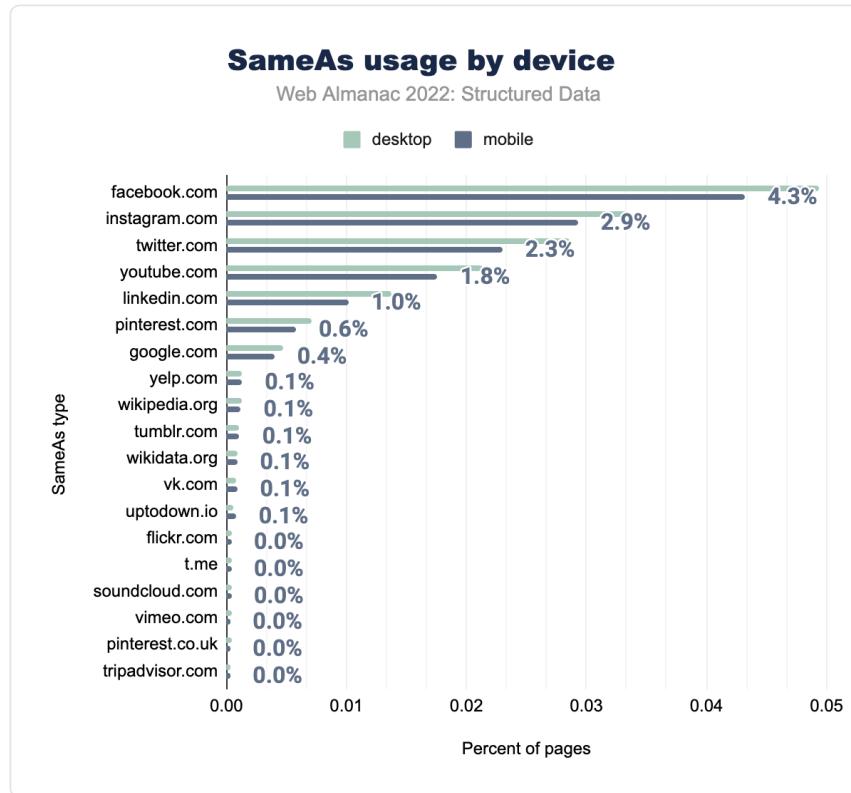


Figure 4.26. *SameAs* usage by device

The rest of the list includes information sources such as wikipedia.org (at 0.13% on both mobile and desktop) and yelp.com (at 0.11% on mobile and 0.13% on desktop), both at an increase from the previous year.

Further JSON-LD insights - relative changes

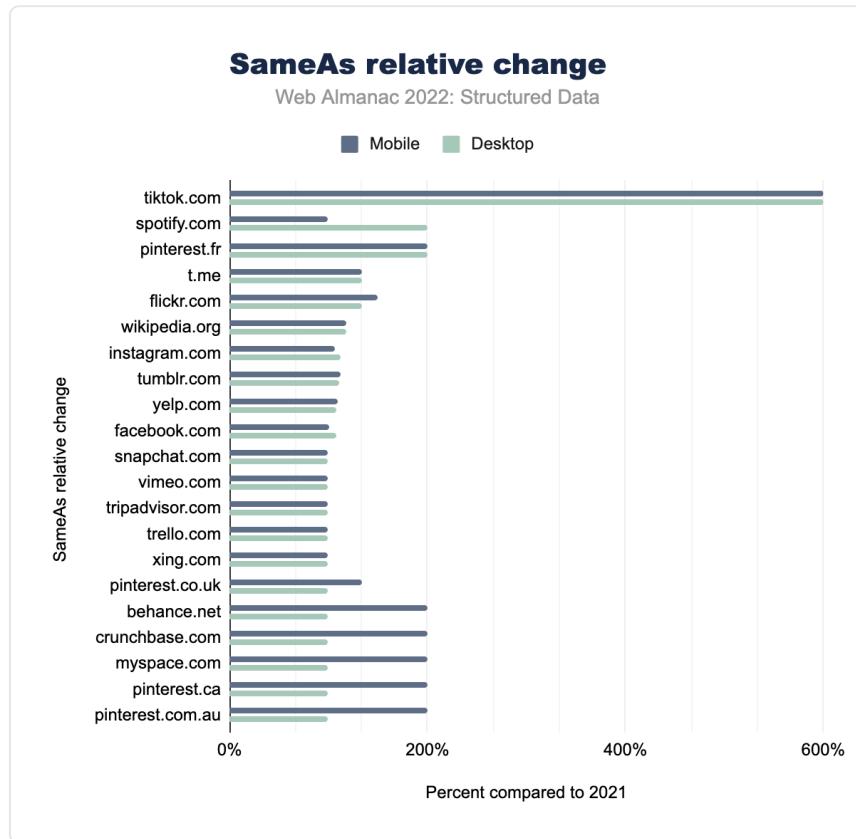


Figure 4.27. *SameAs relative change*

It is insightful to look at the `SameAs` entries and how they change over time. TikTok has seen a huge increase with 2022 showing its appearance on six times as many pages relative to our set compared to 2021. This change is equal for both desktop and mobile pages. Pinterest, and the various domain names it has, make up for 3 of the top 5 largest growth for mobile pages in 2022. Mobile overall has seen a larger increase for `SameAs` entries than desktop, with Spotify being an exception with its desktop page appearances being doubled compared to 2021.

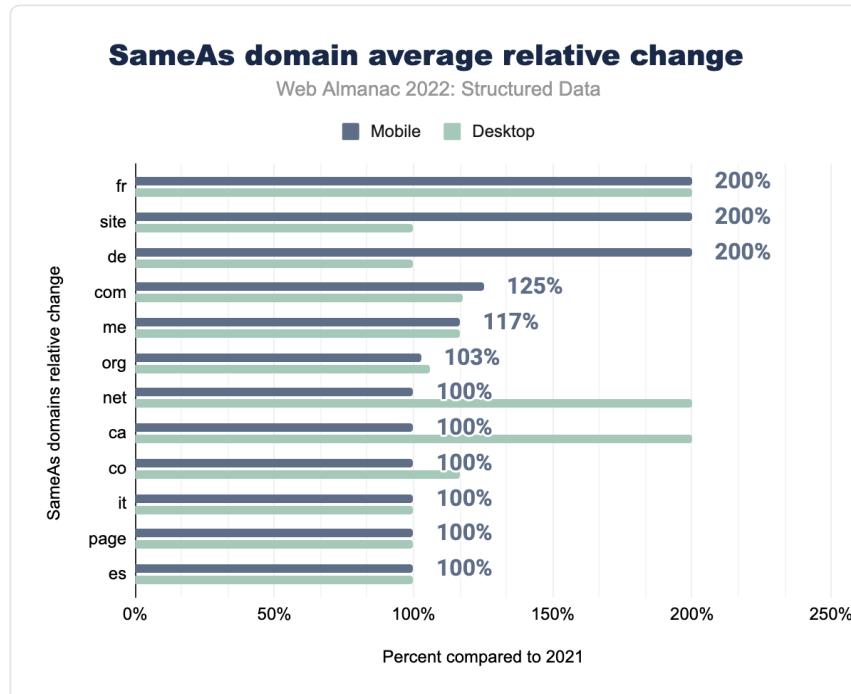


Figure 4.28. `SameAs` domain average relative change

Looking at the domain names of the `SameAs` entries, and how they change over time, also may give valuable insight. In the largest desktop page growth we see `.ca`, `.net` and `.fr`, with the latter also being up there in the top for mobile page increases. As this is an average, the amount of entries is not accounted for. In both years `.com` is far larger in numbers than all other entries, but the average change is 125% for mobile pages and 117% for desktop pages. The Canadian and French domain averages are heavily boosted by Pinterest, which—as mentioned above—has seen widespread increases from last year. In fact, 7 out of the top 10 `SameAs` domain growers have Pinterest in their entries, sometimes being their only entry.

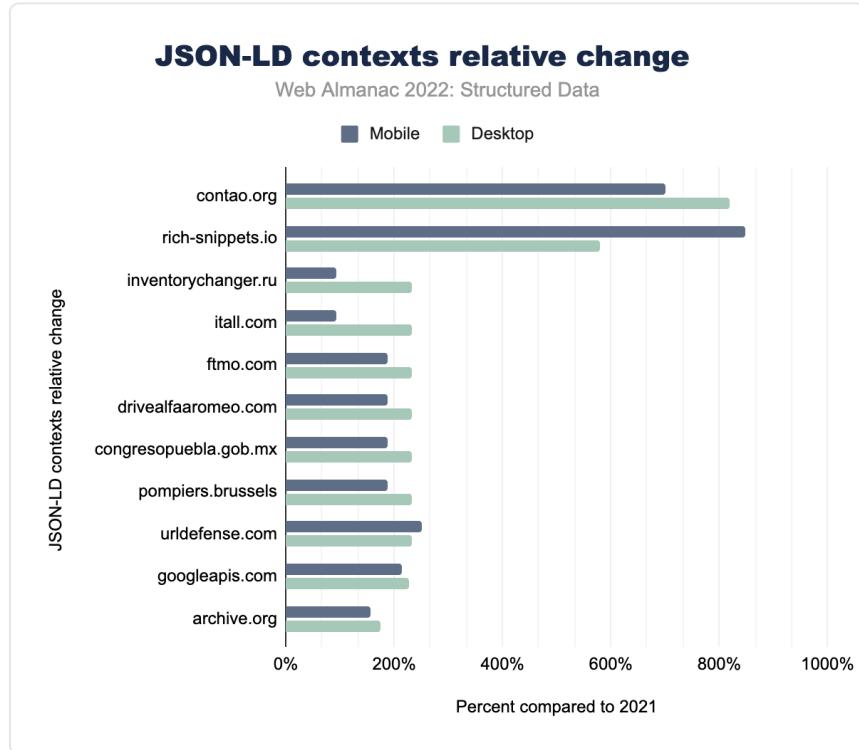


Figure 4.29. JSON-LD contexts relative change

For JSON-LD contexts, schema.org is by far the largest contributor, with over 6,000 times as many appearances for desktop pages and over 3,500 times as many appearances for mobile pages than the second largest contributor, googleapis.com. With that said, googleapis.com's appearances more than doubled for both desktop and mobile pages, compared to schema.org's more modest increase to 108% of last year's numbers. In terms of top growers, contao.org and rich-snippets.io take the top spots with contao.org's desktop page increase of 819% and rich-snippets.io's mobile increase of 849%. Contao.org ranks 4th in total entries, while rich-snippets.io ranks at 8th.

Conclusion

A lot has been outlined about how structured data affects the web and, by extension, our experience. This year we also focused on comparing how the adoption of structured data has changed over a year. We could see, for example, the general increase in some classes like LocalBusiness (particularly for Restaurants) or FAQ and the slight decrease in usage for some of

the less relevant structured data types that come from `foaf` or the microdata syntax.

Although far from being a comprehensive list, we can see structured (linked) data bringing several advantages to:

- Ecommerce pages
- Business pages
- Researchers
- Social media sites
- Users

Through SEO, increased discoverability, general data linking, and rich results drive the adoption. Implementing semantic markup within web pages results in a more connected and accessible web.

With more information and insight available than ever, it is essential to understand the capabilities and limitations of specific techniques or choices when trying to increase web page traffic. Adding fake reviews or misleading data in the hopes of improving SEO will be detected, resulting in fewer benefits from those mentioned above and poorer discoverability and performance from search engines.

As already seen in the previous year, while SEO remains a crucial driver for adopting structured data, a growing landscape of use cases is emerging beyond search engines. Website owners are adding data in diverse scenarios to make their systems interoperable, train their content recommendation systems, and gain new insights from web pages.

Although this is only the second year for this chapter in the Web Almanac, we are excited to see how these trends continue and change, along with the state of structured data on the web. With all of the benefits structured data brings, we expect an increasing implementation of these technologies.

Authors

**Andrea Volpini**

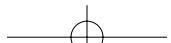
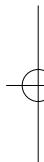
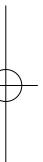
🐦 @cyberandy 🌐 cyberandy 🌐 <https://wordlift.io/blog/en/entity/andrea-volpini>

Andrea Volpini is the CEO of WordLift, and is currently focusing on the semantic web, SEO and artificial intelligence.

**Allen O'Neill**

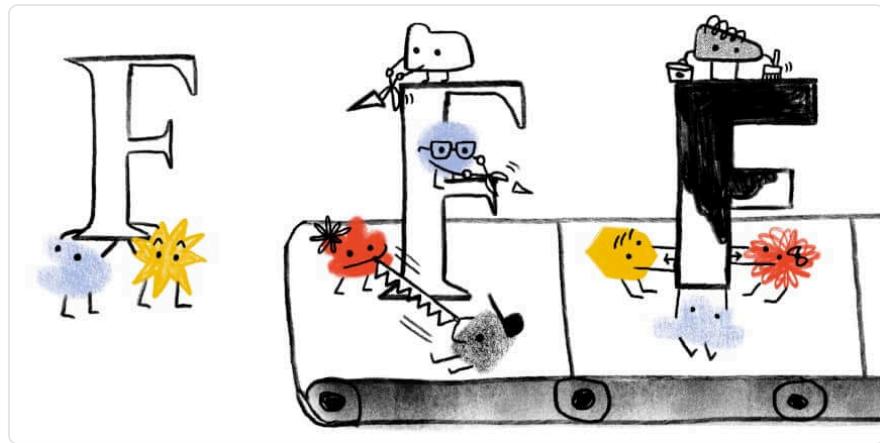
🐦 @DataBytesAI 🌐 DataBytzAI 🌐 allenoneill 🌐 <https://webdataworks.io/>

Allen is founder and CTO for 'The DataWorks', delivering AI-driven web-data solutions to top tier organizations worldwide. His core focus is designing innovative technology solutions at scale, and his primary background is in enterprise systems.



Part I Chapter 5

Fonts



Written by Bram Stein

Reviewed by Alex N. Jose, José Solé, Roel Nieskens, and Chris Lilley

Analyzed by Bram Stein and Kanmi Obasa

Edited by Shaina Hantsis

Introduction

We've come a long way since the early days of web fonts. We went from a handful of web-safe fonts to a typographic explosion of hundreds of thousands web fonts. The technology and ease of use is almost unrecognizable: from elaborate "bullet-proof" font loading strategies with several font formats to simply including a WOFF2 file.

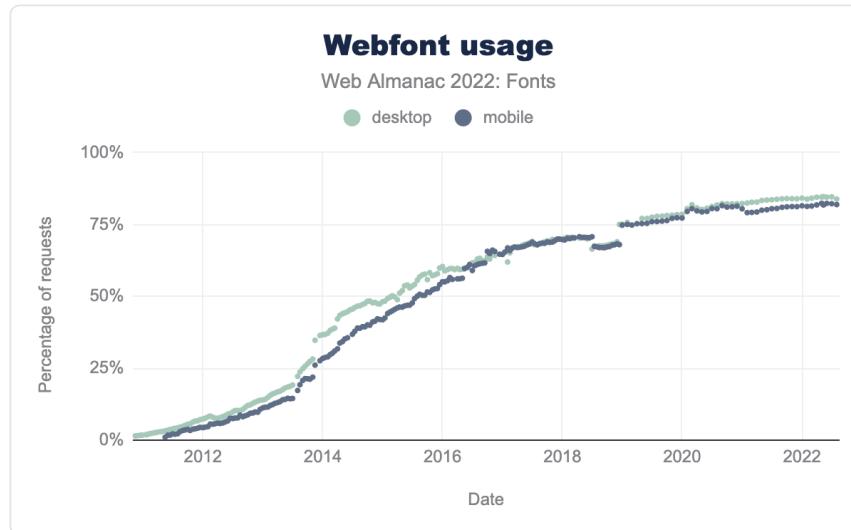


Figure 5.1. Webfont usage.

This progression in web fonts shows. Web font usage continues to grow. In the 2020 Fonts¹³¹ chapter, 82% of all desktop sites used web fonts. In the two years since then, usage has increased to about 84%. The numbers are slightly lower for mobile, but represents a similar growth.

While we've made tremendous progress we're not quite there yet. Large percentages of the world's population can't use web fonts because their writing systems are either too large or too complex to be delivered as a (small) web font. Fortunately, the W3C Fonts Working Group¹³² is working hard on the Incremental Font Transfer¹³³ web standard that will hopefully solve this.

There was no Fonts chapter in 2021, but we hope we can make up for that this year. We took a slightly different angle this year by taking a closer look at what is inside font files and how fonts are used in CSS. We of course also returned to the "classics" such as services, `font-display`, and resource hints usage. Finally, we wrap up the chapter with two special focus sections on *variable fonts* and *color fonts*—because we think they are great.

131. <https://almanac.httparchive.org/en/2020/fonts>

132. <https://www.w3.org/Fonts/WG/>

133. <https://www.w3.org/TR/IFT/>

Performance

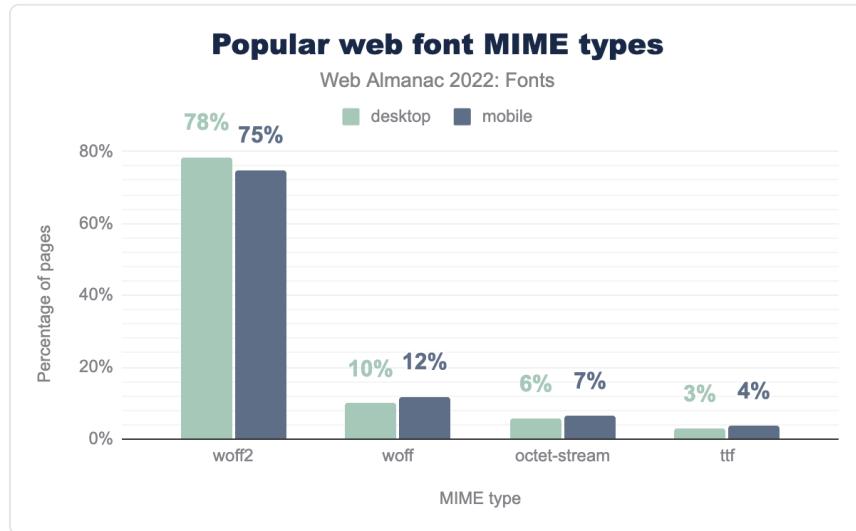


Figure 5.2. Popular web font MIME types.

Surprisingly, not a lot has changed in the types of fonts served. About 75% of all font files are served as WOFF2¹³⁴, about 12% as WOFF and the remainder as either octet-stream or TrueType Font—and then a whole bunch of random MIME types. This is fairly similar to the results in the 2020 Fonts chapter¹³⁵. Fortunately, SVG and EOT font usage has almost disappeared completely.

As noted in 2019, 2020: WOFF2 offers the best compression and should be the preferred format. In fact, we think it is also time to proclaim:

Use only WOFF2 and forget about everything else.

This will simplify your CSS and workflow massively and also prevents any accidental double or incorrect font downloads. WOFF2 is now supported everywhere¹³⁶. So, unless you need to support really ancient browsers, just use WOFF2. If you can't, consider not serving any web fonts to those older browsers at all. This will not be a problem if you have a robust fallback strategy in place. Visitors on older browsers will simply see your fallback fonts.

134. <https://www.w3.org/TR/WOFF2/>

135. <https://almanac.httparchive.org/en/2020/fonts#formats-and-mime-types>

136. <https://caniuse.com/woff2>

Hosting

Where do people get their fonts? Do they self host, or use a web font service? Both? Let's take a look at the numbers.

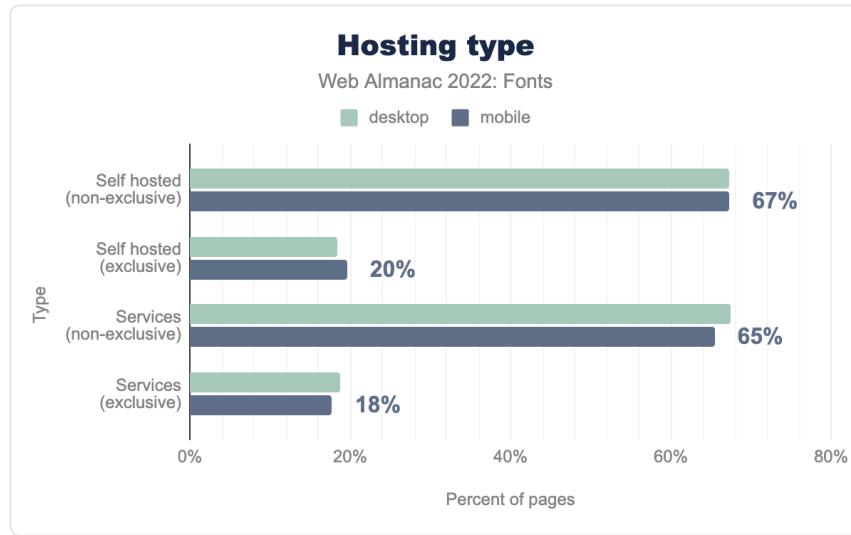


Figure 5.3. Hosting type.

In general, it is a mixture: 67% self host and use a service. Only 19% only use self hosting exclusively. We expect this number to go up in the coming years for two reasons: there is no longer a performance benefit to using a hosted service after the introduction of cache partitioning¹³⁷, and European courts are slowly becoming highly skeptical of European-based companies using Google Fonts¹³⁸.

We can further split this data by service. Perhaps not surprisingly, Google Fonts¹³⁹ is the most popular web font service with nearly 65% of all web pages using it. Free is hard to beat indeed.

137. <https://developers.google.com/web/updates/2020/10/http-cache-partitioning>

138. https://www.theregister.com/2022/01/31/website_fine_google_fonts_gdpr/

139. <https://fonts.google.com/>

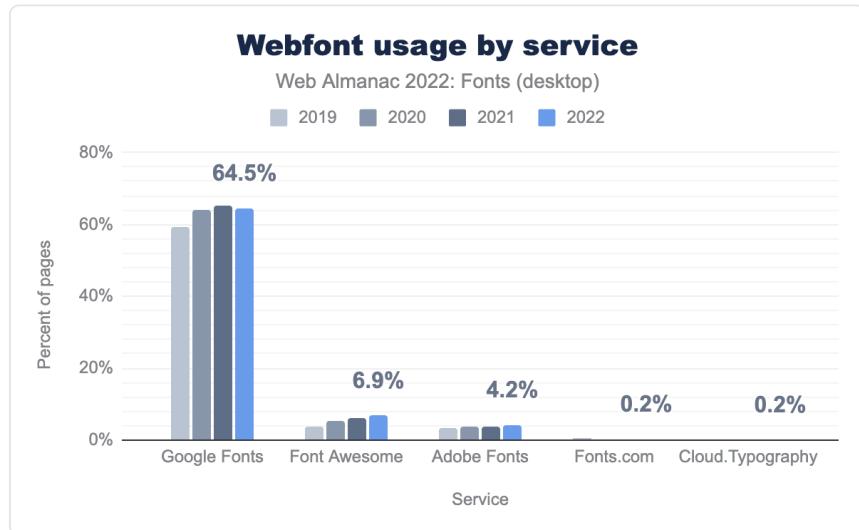


Figure 5.4. Webfont usage by service.

The next runner-up is the Font Awesome¹⁴⁰ web service, which is used on nearly 7% of sites. That is an amazing achievement with only a single font family! In third place is the Adobe Fonts¹⁴¹ web service, which is used on 4.2% of sites. Trailing far behind are the Fonts.com¹⁴² and Cloud.Typography¹⁴³ services, both present on 0.2% of sites.

Looking back at previous years, we can see that Google Fonts usage declined for the first time this year! It's hard to say if this is due to the aforementioned cache partitioning, GDPR, or something else entirely. The decline is only slight, so it will be interesting to see if the trend continues next year.

In contrast, both Font Awesome and the Adobe Fonts service grew significantly over the last few years. Font Awesome service usage grew 86% from 2019 to 2022, while Adobe Fonts usage grew by 24% in the same period.

Note that the services data is measured differently compared to the 2020 and 2019 font chapters. Those chapters looked at the number of requests to a service, whereas the 2022 data looks at pages using the services. Thus the data in 2022 is more accurate as it isn't influenced by the number of fonts loaded on a site. For example, the drop in Google Fonts usage noted in the 2020 chapter was most likely caused by Google Fonts switching to *variable fonts* and thereby significantly reducing the number of requests to their service.

140. <https://fontawesome.com/>

141. <https://fonts.adobe.com/>

142. <https://www.fonts.com/>

143. <https://www.typography.com/webfonts>

File sizes

Compression is a great way to reduce the amount of data that needs to be downloaded, but it has its limits. To better understand what influences font file sizes, let's take a look at the median font sizes across all fonts.

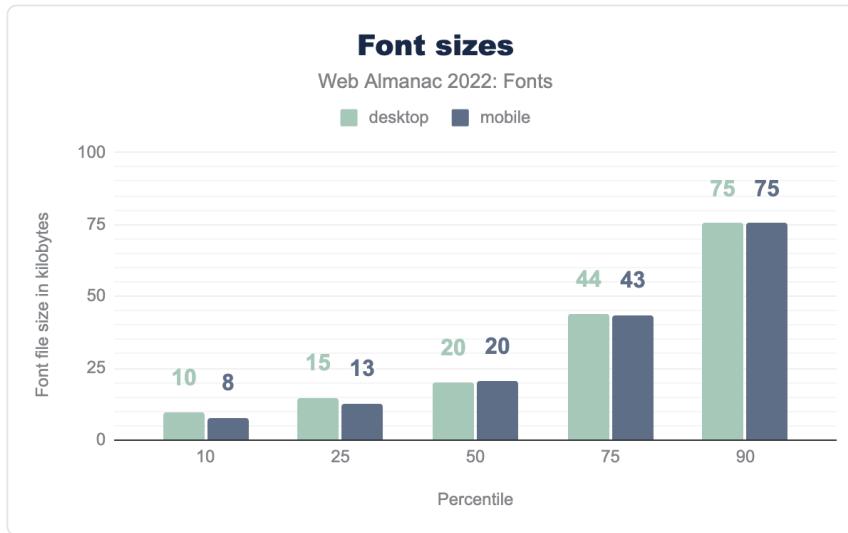


Figure 5.5. Font sizes.

The median font size is about 20 kilobytes. That is pretty good. However, as we have seen earlier, font services account for nearly 70% of all font requests. Services like Google Fonts and Adobe Fonts have teams dedicated to optimizing the fonts as much as possible, so the median font sizes are likely heavily skewed by these services.

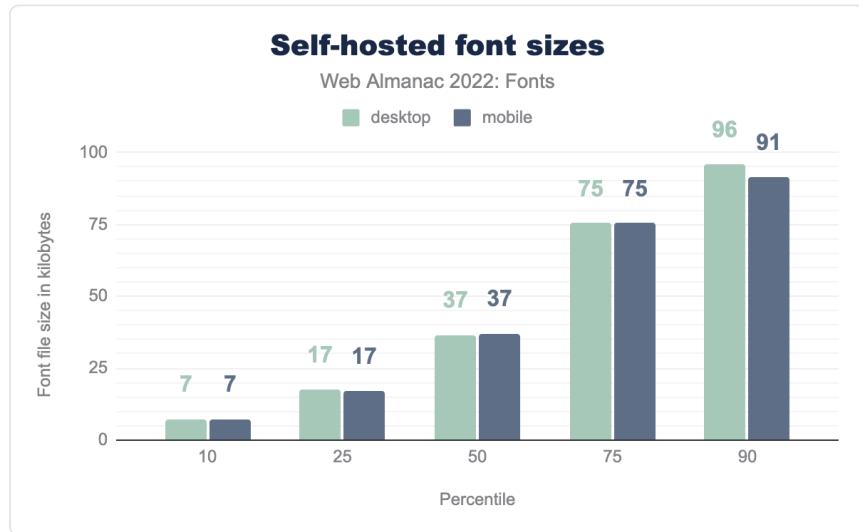


Figure 5.6. Self-hosted font sizes.

Looking at self-hosted font sizes paints a quite different picture. The median font size nearly doubles to about 40 kilobytes. What is going on here? If we revisit the chart of popular web font MIME types, and remove all requests coming from web font services we get some insight into what might be going on.

A lot of websites using self-hosted fonts are still using WOFF instead of WOFF2. It's not clear if the fonts on these sites were never updated since WOFF2 was introduced, or if not enough developers know about WOFF2. Regardless, it's an easy optimization that could benefit a lot of sites.

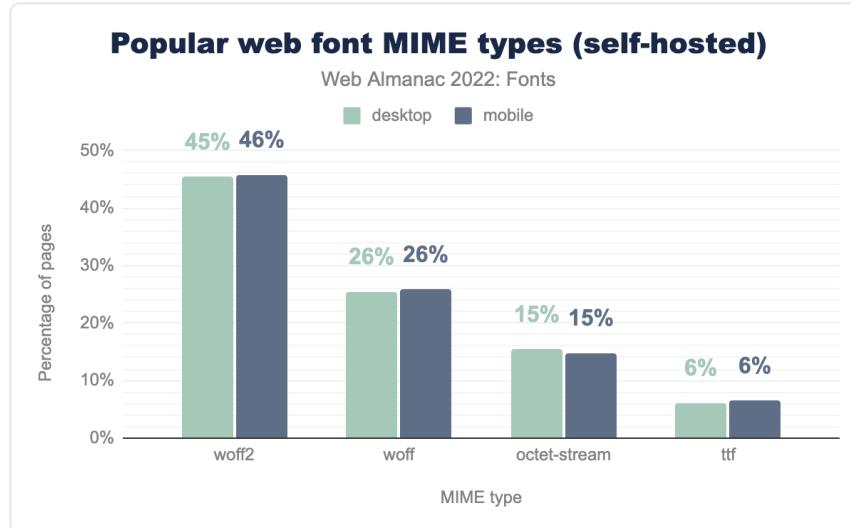


Figure 5.7. Popular web font MIME types (self-hosted).

However, the difference in median font size between services and self-hosted is too large to be explained by a lower usage of WOFF2. While WOFF2 offers excellent compression, compression alone does not explain the large difference in median font sizes. The web font services must be performing other types of optimizations that aren't being done (enough) for self hosted fonts. To find the answer, we'll need to take a look at the insides of a font.

OpenType table sizes

A typical font is essentially a tiny relational database¹⁴⁴ with each table storing data like glyph shapes, glyph relationships, and metadata. For example, there are tables to store the vector Bézier curves that make up glyphs—the characters in the font. There are also tables for relating glyphs to one another, that store things like kerning and ligature relationships (i.e. swap these two glyphs with this one when they are used together, like the famous *fi* ligature).

A reasonable way to measure how much of an impact a table has on overall file size is to multiply its median size by the number of fonts that include that table.

¹⁴⁴. <https://simoncozens.github.io/fonts-and-layout/opentype.html>

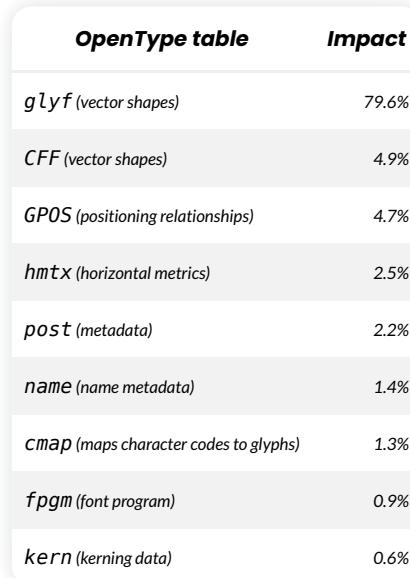


Figure 5.8. Impact (median file size × number of requests as percentage of total).

The top ten highest impact tables starts with the `glyf`, `CFF`, `GPOS`, and `hmtx` tables. These contain the data for the Bézier curves that make up the outlines of all glyphs (`glyf` and `CFF`), OpenType positioning features (`GPOS`) and horizontal metrics (`hmtx`). This is great because these tables are directly related to the number of glyphs in the font. Reduce the number of glyphs in the font by removing glyphs you don't need and you will dramatically reduce its file size.

Figuring out what you need and what you don't need¹⁴⁵ is the hard part though. You might accidentally remove glyphs, or break OpenType features that you need to render text correctly. Instead of subsetting manually using, for example, font tools¹⁴⁶, you can use tools like subfont¹⁴⁷ or glyphhanger¹⁴⁸ to automatically create a “perfect” subset based on the content on your site. However, be mindful whether the license of your font permits such modifications.

It is interesting to note that the `name` and `post` tables are in the top 10. These two tables primarily contain metadata that is important for desktop fonts, but not necessary for web fonts. This is an indication a lot of web fonts contain metadata that can be stripped without consequences, such as name table entries, glyph names in the `post` table, non-Unicode `cmap` entries, etc. We would love to see a universal set of recommendations (or even a pngcrush¹⁴⁹-like

145. <https://bramstein.com/writing/web-font-anti-patterns-subsetting.html>

146. <https://fonttools.readthedocs.io/en/latest/subset/index.html>

147. <https://github.com/Munter/subfont>

148. <https://github.com/zachleat/glyphhanger>

149. <https://pmt.sourceforge.io/pngcrush/>

tool) that can be used by foundries and web developers to remove every last unnecessary byte from a web font.

Outline formats

You might have noticed the OpenType sizes table contains two entries for vector glyph outline data: `glyf` and `CFF`. There are actually four competing ways to store vector outlines in OpenType: TrueType (`glyf`), Compact Font Format (`CFF`), Compact Font Format 2 (`CFF2`), and Scalable Vector Graphics (`SVG`; not to be confused with the old SVG font format). There are also three image based formats—we will talk about two of them in the color fonts section.

The OpenType specification is what you could charitably call “a compromise”. Several competing approaches to do mostly the same thing were added to the specification because there was no consensus. If you’re interested in how this came to be, *The Font Wars*¹⁵⁰ by David Lemon is a great read. We’ll see this pattern of competing approaches repeated again and again in the sections on variable and color fonts (though with different actors). At the end of the day, having multiple ways to store vector outlines mostly works, but it does place a heavy additional burden on type designers and implementations—not to mention increasing the attack surface area for exploits.

Type designers can choose the outline format they prefer. Looking at the distribution of outline formats, it is pretty clear what type designers have chosen. The overwhelming majority (91%) of fonts use the `glyf` outline format, while 9% use the `CFF` outline format. There is some `SVG` color font usage as well, but less than 1% (not pictured).

¹⁵⁰. <https://www.pastemagazine.com/design/adobe/the-font-wars/>

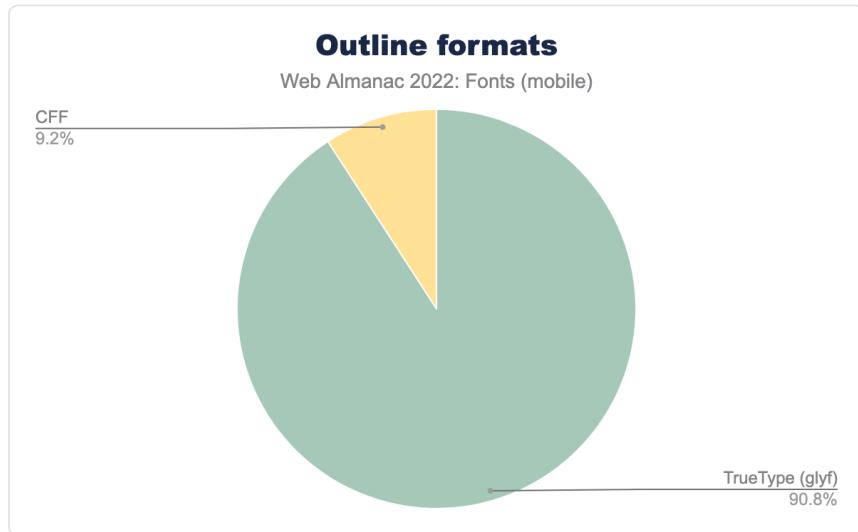


Figure 5.9. Outline formats.

The OpenType specification lists the differences between `glyf` and `CFF`:

- The `glyf` format uses quadratic Bézier curves while `CFF` (and `CFF2`) uses cubic Bézier curves. This matters to some type designers, but not to users of the font.
- The `glyf` format has more control over hinting—making small adjustments so that the glyph outlines snap to the correct pixels at smaller sizes—while `CFF` has most of its hinting built into the text rasterizer.
- The `CFF` format claims to be a more efficient format, resulting in smaller font sizes.

The last claim is interesting. Is `CFF` smaller?

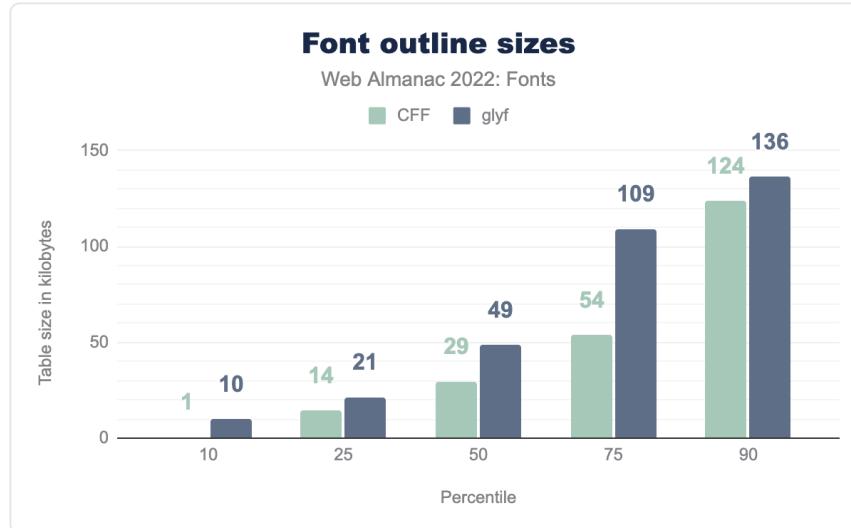


Figure 5.10. Font outline sizes.

On average, CFF does indeed produce a smaller table size. However, the reality is more nuanced, as it doesn't take compression into account—the table sizes are recorded after the font has been uncompressed.

As can be seen in the WOFF2 evaluation report¹⁵¹, the median glyf compression is at 66% while the median CFF compression is at 50% (from uncompressed to compressed using WOFF2).

¹⁵¹. <https://www.w3.org/TR/2016/NOTE-WOFF2ER-20160315/#brotli-adobe-cff>

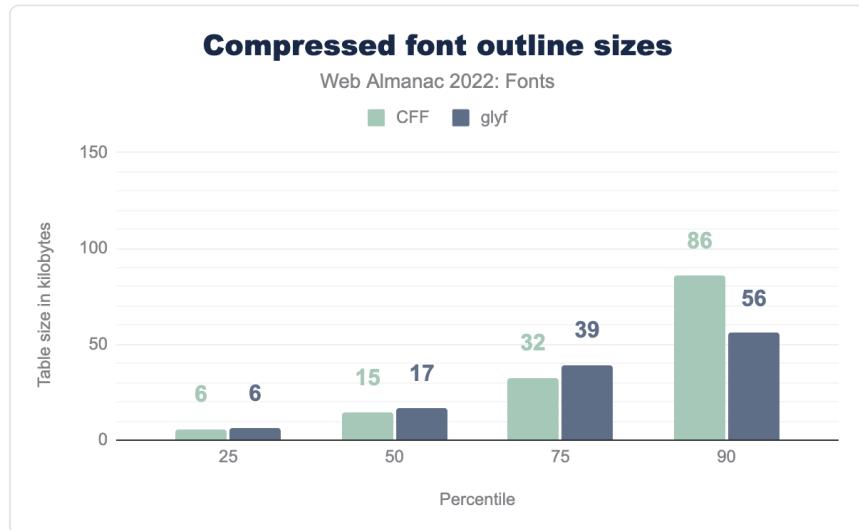


Figure 5.11. Compressed font outline sizes.

Applying compression paints a very different picture. The median file size differences are negligible, and large fonts are much smaller when using `glyf` outlines!

In other words, even if `CFF` starts out smaller, it compresses much less than `glyf`, so it all evens out in the end. In fact—for larger files—it appears using the `glyf` format produces smaller sizes.

Resource hints

Resource hints are special instructions to the browser to load or render a resource before it normally would. Browsers normally only load web fonts when they know a font is used on the page. In order to know that, it needs to have parsed both the HTML and CSS. However, if you, as a web developer, know that a font will be used, you can use resource hints to tell the browser to load fonts much earlier.

There are several types of resource hints that are relevant to web fonts: `dns-prefetch`, `preconnect`, and `preload`—in order of the lowest to highest impact. Ideally you would like to preload your most important fonts, but depending on where they are hosted that may not always be possible.

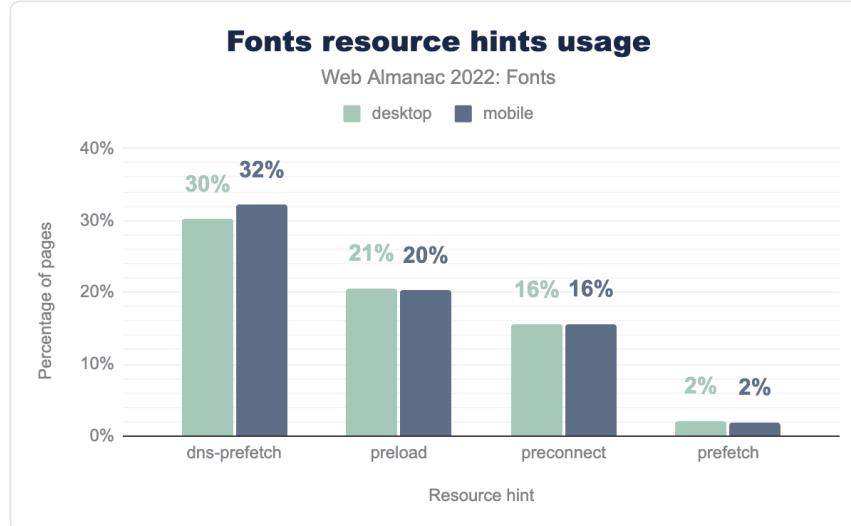


Figure 5.12. Fonts resource hints usage.

There hasn't been any large change in the use of `dns-prefetch` hints since 2020, but there has been a fairly significant increase in the use of `preload` (from 17% in 2020 to 20% in 2022) and `preconnect` (from 8% in 2020 to 15% in 2022). This is great!

As mentioned in the 2020 chapter¹⁵², `preload` and `preconnect` resource hints have the single largest impact on your font loading performance. In most cases it is as simple as adding a link element to your head. For example, if you use Google Fonts:

```
<link rel="preconnect" href="https://fonts.gstatic.com">
```

If you self-host your fonts you can go even further and provide hints to the browser to preload your most important fonts—your primary text font for example. That way the browser can load it early and it will likely be available when text rendering starts.

`font-display`

For many years most browsers did not render text until web fonts had loaded. On slow connections, this would often result in several seconds of invisible text even though the text content had already loaded. This behavior is called *block*, because it blocks rendering of the

152. <https://almanac.httparchive.org/en/2020/fonts#resource-hints>

text. Other browsers showed fallback fonts right away and swapped them when the web font loaded. When a fallback font is replaced by a web font, this is called *swap*.

To give web developers more control over font loading, the `font-display` descriptor was introduced to tell the browser how it should behave while web fonts are loading. It defines four different values of what to do while fonts are loading. These values are implemented using different combinations of *block* and *swap* behavior.

- `swap`: block for a very short period and always swap.
- `block`: block for a short period and always swap.
- `fallback`: block for a very short period and swap within a short period.
- `optional`: block for a very short period and no swap period.

There is also `auto`, which leaves the decision up to the browser—all modern browsers use `block` as the default value.

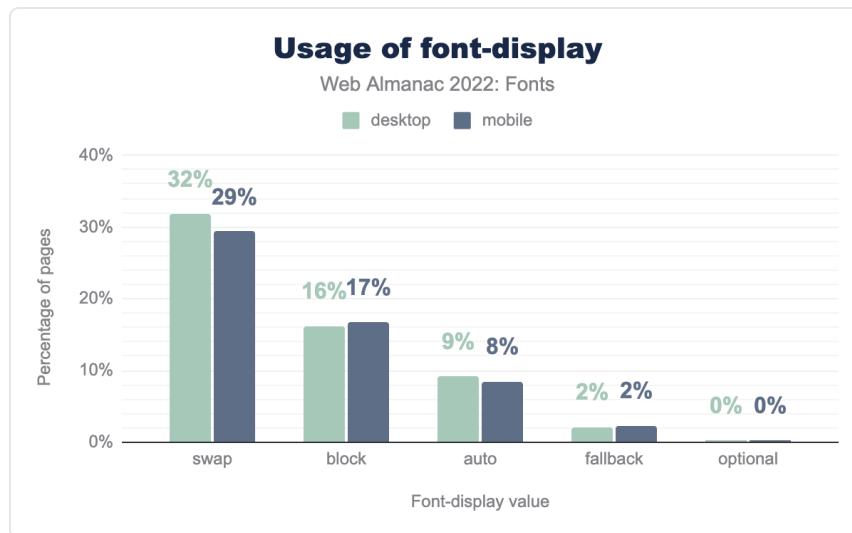


Figure 5.13. Usage of `font-display`.

Use of `font-display: swap` has grown to an impressive 30% (from 11% in 2020¹⁵³). A fair chunk of this increase can most likely be attributed to Google Fonts making swap the recommended value in 2019¹⁵⁴. It is also interesting to see the `block` value overtaking `auto`

153. <https://almanac.httparchive.org/en/2020/fonts#racing-to-first-paint>

154. <https://www.youtube.com/watch?v=YJGCZCaIZkQ&t=1880s>

as the second most used value. We are not sure why developers are intentionally degrading the performance of their site, but it is an interesting, if not slightly worrying, development.

Our guess is that developers—or their customers—really dislike seeing a flash of fallback fonts. Using `font-display: block` is an easy “fix” for that problem. However, there is a better solution on the horizon. In the near future you can use CSS font metric overrides to tweak your fallback fonts to approximate the metrics of your web fonts. This will reduce the jarring reflow of text when a fallback font is swapped with a web font.

The CSS `ascent-override`, `descent-override`, `line-gap-override`, and `size-adjust` descriptors go into the `@font-face` rule and can be used to override the metrics in any font. You can use these descriptors with `local()` to create a customized fallback¹⁵⁵ font that roughly matches your web font—he, finally a good use for `local()`.

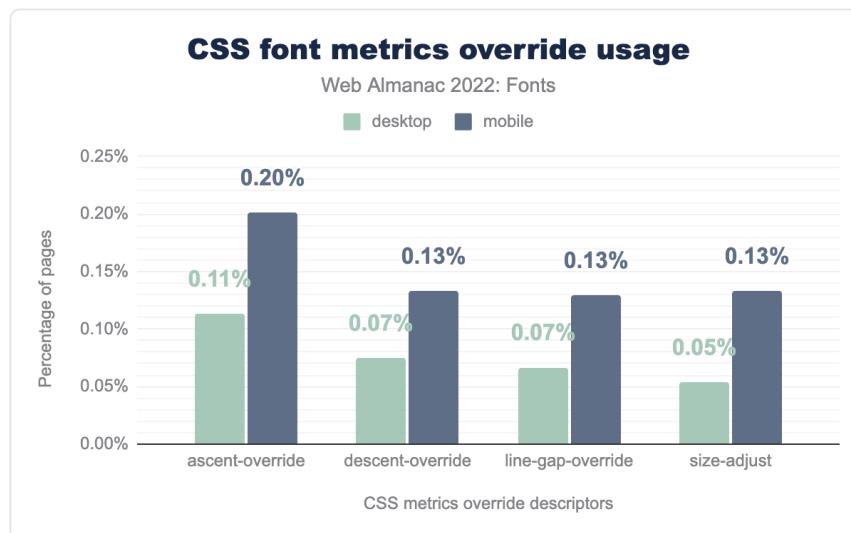


Figure 5.14. CSS font metrics override usage.

These `@font-face` descriptors are very new, but are already seeing some use. To make them even more useful developers need two things:

1. A set of consistent fallback fonts that are available in all browsers and on all platforms. They could even be variable fonts. Imagine the possibilities.
2. Tools to automatically match fonts by tweaking its size and metrics. Doing this by hand is very time-consuming, so a tool is a must. This is not intended as a replacement for the web font, but merely as a temporary fallback while the web

155. https://developer.mozilla.org/docs/Web/CSS/@font-face/ascent-override#overriding_metrics_of_a_fallback_font

fonts are loading (or as a stand-in if the fonts don't load or the browser is very old).

We're slowly getting there with tools such as Font Style Matcher¹⁵⁶ and Perfect-ish Font Fallback¹⁵⁷, but unfortunately, fallback fonts are still very much platform dependent.

Font usage

Performance is important, but it is also interesting to see how fonts are being used on the web. For example, what are the most popular fonts and foundries? Are people using OpenType features? Let's take a look at the data.

156. <https://meowni.ca/font-style-matcher/>
157. <https://www.industrialempathy.com/perfect-ish-font-fallback/>

Families & foundries

Family	desktop	mobile
<i>Roboto</i>	14.5%	1.5%
<i>Font Awesome</i>	10.5%	12.8%
<i>Noto Sans</i>	10.1%	8.0%
<i>Open Sans</i>	5.9%	7.7%
<i>Lato</i>	3.6%	3.9%
<i>Poppins</i>	3.0%	4.0%
<i>Montserrat</i>	2.5%	3.1%
<i>Source Sans Pro</i>	1.6%	1.9%
<i>icomoon</i>	1.3%	1.5%
<i>Proxima Nova</i>	1.0%	1.0%
<i>Raleway</i>	1.0%	1.3%
<i>Noto Serif</i>	0.8%	1.0%
<i>Ubuntu</i>	0.7%	0.9%
<i>NanumGothic</i>	0.7%	0.3%
<i>Oswald</i>	0.6%	0.8%
<i>PT Sans</i>	0.6%	0.8%
<i>GLYPHICONS Halflings</i>	0.5%	0.6%
<i>Rubik</i>	0.4%	0.4%
<i>eicons</i>	0.4%	0.6%
<i>revicons</i>	0.4%	0.5%

Figure 5.15. Most used fonts.

With Google Fonts' massive impact on web font serving, it is no surprise the most used font on the web is Roboto. Roboto was created by Google as a system font for its Android operating system. This also explains the huge discrepancy between Roboto's use on mobile compared to desktop sites. On Android, Google Fonts will use the system installed version instead of

downloading it as a web font.

Font Awesome takes the number two spot, which is an impressive accomplishment for what is essentially a single font family. Font Awesome, combined with Icomoon, Glyphicons, eicons, and revicons make up nearly 18% of all web fonts used on websites! Icon fonts are problematic from an accessibility point of view¹⁵⁸, so it is worrying to see this being so popular.

18%

Figure 5.16. Sites using icon web fonts.

A special note should also be made of Proxima Nova at 1% usage. It is the only commercial, non-icon, font in the top 20. That's an amazing achievement by Mark Simonson Studio¹⁵⁹.

Another interesting fact is that a large portion of the top families are open source. This can be credited to Google Fonts who have either commissioned these fonts or included existing open source fonts in their library.

Vendor	desktop	mobile
Google	30.5%	17.7%
Font Awesome	12.3%	15.6%
Łukasz Dziedzic	3.6%	4.3%
Indian Type Foundry	3.0%	4.1%
Julieta Ulanovsky	2.5%	3.1%
Adobe	1.6%	1.9%
Ascender Corporation	1.6%	2.0%
Icomoon	1.3%	1.5%
Mark Simonson Studio	1.3%	1.3%
ParaType Inc.	1.0%	1.4%

Figure 5.17. Most popular font foundries.

158. <https://fontawesome.com/docs/web/dig-deeper/accessibility>

159. <https://www.marksimonson.com/>

The list of most popular type foundries (or in some cases type designers) is equally fascinating. Besides large companies like Google, Adobe, Ascender (Monotype), it is good to see several smaller companies and even several individuals having such a large impact.

OpenType features

OpenType features are often referred to as a font's superpowers. And of course, the fonts with superpowers are often unrecognized. OpenType features are hard to discover and use. Fortunately, there are web tools, such as Wakamai Fondu¹⁶⁰, that clearly show you which features there are, what they do, and how to use them.



Figure 5.18. Fonts that include OpenType features.

Some OpenType features are for stylistic purposes only, while others are required to render text correctly. You might often see these two mentioned as discretionary and required features. Almost 44% of all fonts have either discretionary or required OpenType features. So, there's a good chance the font you are using also has super powers!

Discretionary features can be used to, for example, replace two adjacent glyphs with a ligature to improve legibility. It's also common for OpenType features to offer alternative versions of glyphs, for example by adding swashes.

A significant number of fonts have discretionary OpenType features¹⁶¹. The most common discretionary feature is, not surprisingly, ligatures. This is followed by a whole range of features that modify numerals like fractions, proportional numerals, tabular numerals, lining numerals, and ordinals. Superscripts are also somewhat common.

¹⁶⁰ <https://wakamafondue.com/>
¹⁶¹ https://fonts.google.com/knowledge/introducing_type/introducing_alternate_glyphs

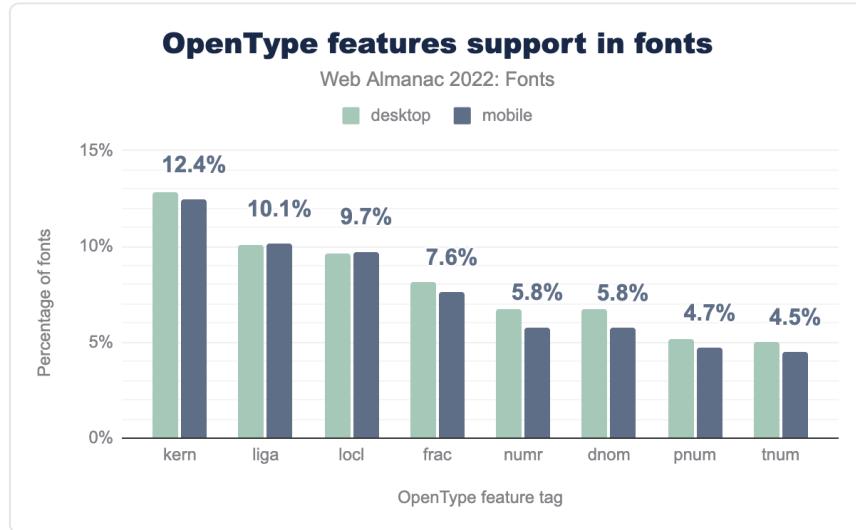


Figure 5.19. OpenType features support in fonts.

If we look at required OpenType features, there are two that stand out from everything else: *kerning* and *localized forms*. Localized forms are used to specify alternate glyphs that are required or preferred by some languages. This implies that a good amount of fonts support multiple languages, which is a great sign of progress for internationalization.

Kerning is the process of slightly increasing or decreasing the space between any combination of two glyphs to make the space between them seem more even. Kerning is enabled by default on all browsers, so as long as the font supports kerning it will be enabled.

34%

Figure 5.20. Fonts including kerning data.

Only 34% of all web fonts have kerning data stored as either an OpenType feature, or using the older `kern` table. Nearly all fonts need kerning to look correct, so we would have expected this number to be much higher than it is. One explanation is that when web fonts started taking off, browser support for kerning wasn't very good, so many early web fonts did not include kerning data to save on space. Nowadays, all browsers support kerning so there is no reason fonts should not have kerning data in them.

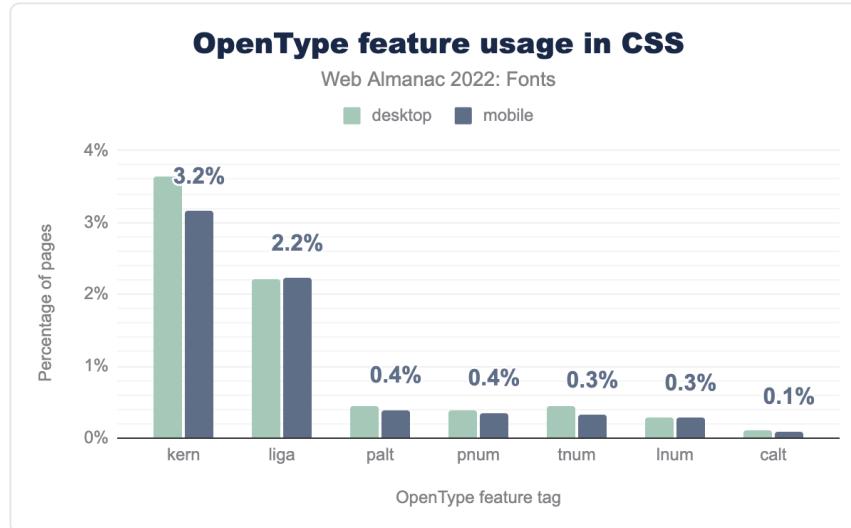


Figure 5.21. OpenType feature usage in CSS.

What's even more interesting is that kerning is also the most common feature tag used in the `font-feature-settings` property. Nearly 3.2% of sites manually enable (or worse, disable) kerning. There is no need for that; it is enabled by default. In fact, there is no need to ever change kerning settings through `font-feature-settings` or the higher level `font-kerning` property. Disabling kerning won't make your site faster, but your typesetting will be poorer for it.

The other most used features are roughly in line with what type designers actually include: ligatures and various numerals. It is interesting to see the `palt` (proportional alternates) feature in this list, as it is primarily used for CJK fonts (which themselves aren't common because they are usually too large to be used as web fonts, even with WOFF2 compression). Like kerning, the `calt` feature (contextual alternates) is enabled by default and should not be explicitly enabled or disabled. There are many other useful OpenType features such as stylistic sets, character variants, small caps, and swashes that have low usage, but have the potential to really enhance your typography. Our recommendation is to drop your fonts in Wakamai Fondu¹⁶² and explore all the hidden superpowers.

¹⁶² <https://wakamafondue.com/>

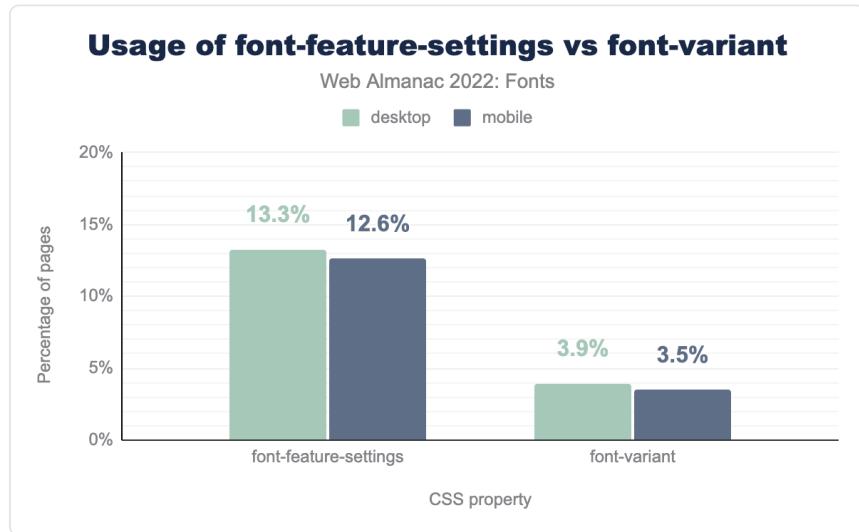


Figure 5.22. Usage of `font-feature-settings` vs `font-variant`.

Overall, usage of OpenType feature usage is quite low on the web. We were hoping most people are using the high-level `font-variant` properties to enable OpenType features, but their usage is even lower than `font-feature-settings`. The `font-feature-settings` property is used on 12.6% of sites, while the `font-variant` properties are used on only 3.5% of sites.

This is disappointing. Not only are people not using the OpenType features present in fonts, they are also primarily using the error-prone `font-feature-settings` property instead of the high-level `font-variant` property. You need to be extra careful with the `font-feature-settings` property, as it will reset any OpenType feature you didn't explicitly list to its default value¹⁶³.

All of the most commonly used `font-feature-settings` values have `font-variant` equivalents that are more readable, and do not unset other OpenType features as a side effect. While support for these high-level features wasn't that great in the past they are well supported¹⁶⁴ these days—except for the recently introduced `font-variant-alternates` property.

163. <https://pixelambacht.nl/2022/boiled-down-fixing-variable-font-inheritance/>

164. <https://caniuse.com/?search=font-variant>

Usage of CSS font-variant values

Web Almanac 2022: Fonts

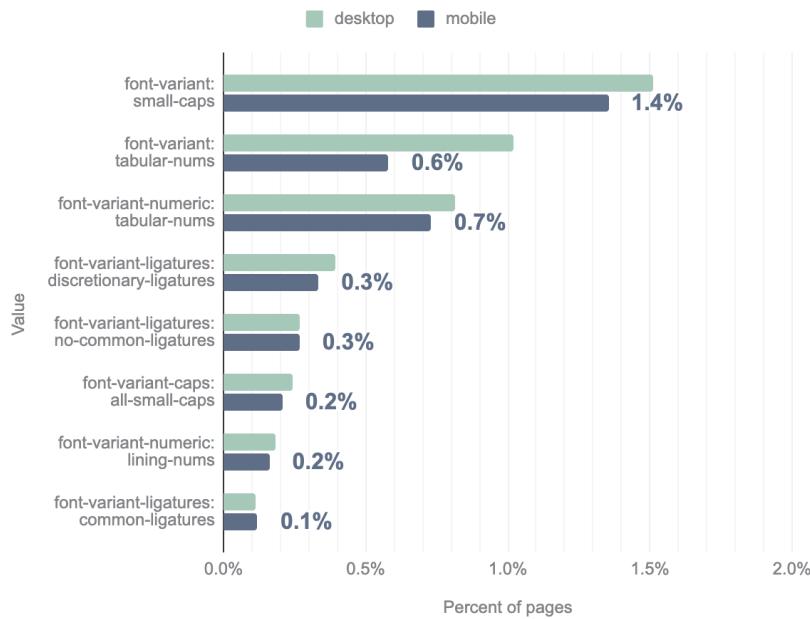


Figure 5.23. Usage of CSS font-variant values.

The most used `font-variant` value is `small-caps` at 1.4% of pages. This is surprising, because small caps are only supported by 0.7% of fonts. That means that most people using `font-variant: small-caps` and `font-variant-caps` will get faux small caps that are generated by the browser instead of small caps created by the type designer! In the future, you can avoid faux small caps by using the `font-synthesis-small-caps` property¹⁶⁵.

The other values roughly match what is provided by the fonts themselves. Even though use of the `font-variant` properties is low, we expect—or rather hope—that these numbers will go up over time and use of `font-feature-settings` becomes relegated to use with custom or proprietary OpenType features.

Writing system and languages

To understand what kind of fonts are being made and used, we thought it would be interesting

^{165.} <https://drafts.csswg.org/css-fonts-4/#font-synthesis-small-caps>

to look at what languages fonts support. For example, do people make a lot of German, Vietnamese, or Urdu fonts? Unfortunately, it is hard to answer this question because a lot of languages share the same writing system. That means they might share the same character set, but might have only been explicitly designed for one specific language. So instead of languages, we'll take a look at writing systems.

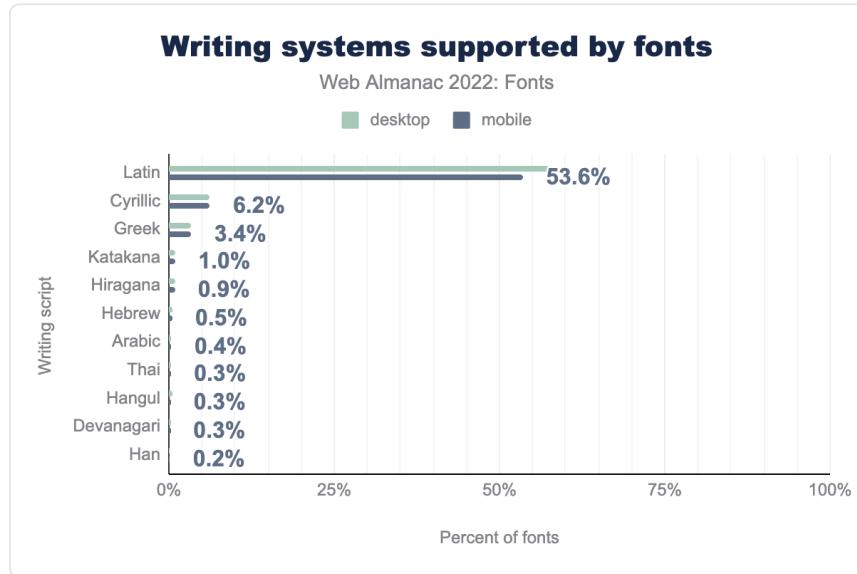


Figure 5.24. Writing systems supported by fonts.

Not surprisingly, the Latin system is in the lead with a whopping 53.6% of all fonts supporting (a significant) subset of the Latin writing system¹⁶⁶. This includes a lot of languages used in the western world, like English, French, and German. However, it also covers languages in Asia, such as Vietnamese and Tagalog. The number two and three spots are taken by Cyrillic and Greek. Again, this is not a surprise, they are commonly used and a reasonably small amount of work to add to a font due to their limited number of extra glyphs that need to be added. Katakana and Hiragana (Japanese) are at 1% and 0.9% respectively—a combined 1.9%. Note that about 10% of fonts failed to meet the minimum threshold of a writing system (not pictured). This includes fonts that, for example, only support a small number of characters or ones that are heavily subsetted.

Sadly, other writing systems are much less prevalent. For example, Han (Chinese) is the 2nd most used writing system in the world¹⁶⁷ (after Latin), but only supported by 0.2% of web fonts. Arabic is the third most used writing system, but again, only supported by 0.4% of web fonts.

166. https://en.wikipedia.org/wiki/List_of_languages_by_writing_system#Latin_script

167. <https://www.worldatlas.com/articles/the-world-s-most-popular-writing-scripts.html>

The reason that some of these writing systems are not used as web fonts¹⁶⁸ is that they are very large due to the sheer number of glyphs they have to support, and the difficulty in subsetting them correctly.

While services like Google Fonts and Adobe Fonts support these writing systems, they are using proprietary technologies that simply are not available for self-hosting. However, the W3C Fonts Working Group is working on a new standard called Incremental Font Transfer¹⁶⁹ that enables web developers to self-host large fonts. We hope to see other writing systems catch up with Latin once this technology becomes widely available.

Generic font families

We already touched on fallback fonts while talking about `font-display`. Sometimes you don't need web fonts at all though, for example in UI elements or form inputs. One of our favorite ways to avoid using web fonts is to use the new generic `system-ui` family name which maps to the font family used by the operating system. Did you know there are several other generic families? There is `ui-monospace`, `ui-sans-serif`, `ui-serif`, and `ui-rounded` as well, if you want to use an operating system font, but have slightly more specific needs.

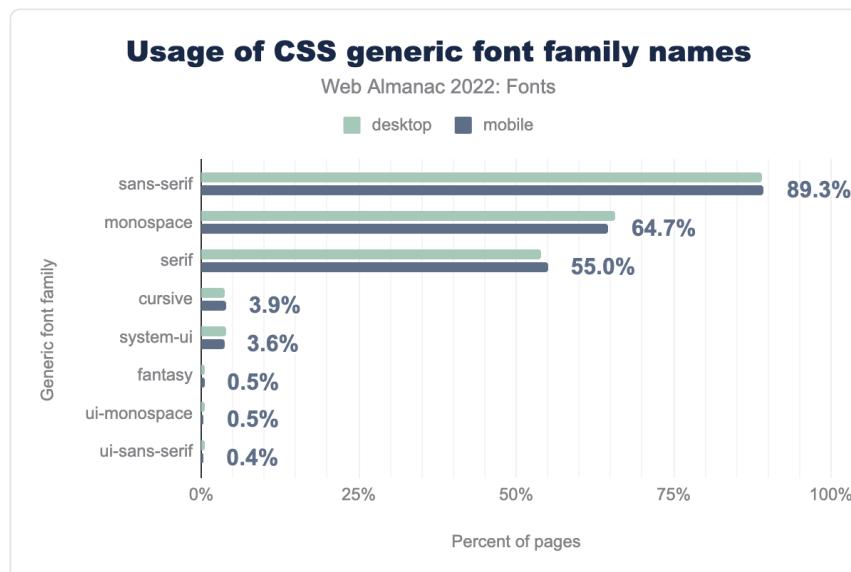


Figure 5.25. Usage of CSS generic font family names.

168. <https://www.w3.org/TR/PFE-evaluation/#fail-large>

169. <https://www.w3.org/TR/IFT/>

While these are fairly new, they are already seeing significant use. The usual suspects, `sans-serif`, `monospace`, and `serif` obviously take the lead as they have been around since the first version of the CSS specification.

The most popular, and well-known, is `system-ui` at 3.6%, followed by `ui-monospace` at 0.5% and `ui-sans-serif` at 0.4%. It isn't clear what the 0.5% of requests for `fantasy` were hoping for, as that generic is so under-specified as to be effectively useless.

We hope to see more use of these generic family names next year. They are great for UI elements, forms, or really anything where you want to evoke a native feel. As an added benefit, they are also great for performance as they are guaranteed to use a locally installed font.

Font smoothing

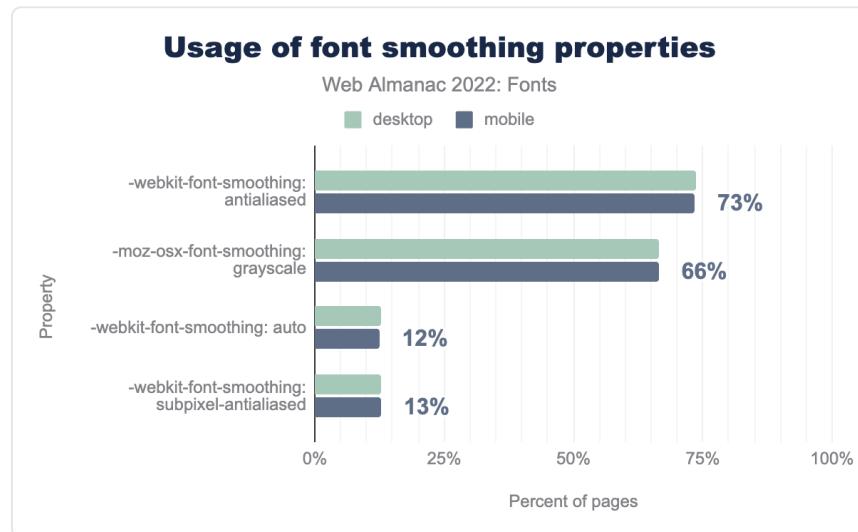


Figure 5.26. Usage of font smoothing properties.

And now for a complete surprise—to us anyway: people really like to specify their MacOS-only font smoothing preferences¹⁷⁰. For example, the `-webkit-font-smoothing: antialiased` value is used on 73.4% of all sites. This is surprising because it—and its siblings `-moz-osx-font-smoothing`, and `font-smoothing`—are not even official CSS properties. This might make them the most used unofficial CSS properties!

Our hunch this is a combination of CSS frameworks including these properties and a dislike of

170. <https://developer.mozilla.org/docs/Web/CSS/font-smooth>

how fonts are rendered slightly bolder on macOS—variable font grades to the rescue! It would be interesting to return to these properties in the 2023 Fonts chapter. Perhaps it is also time to put these properties on a standards track? The demand is clearly there.

Variable fonts

Variable fonts allow type designers to combine multiple styles of a family into a single font file. They do this by defining one or more design axes, such as weight (thin, regular, and bold) or width (condensed, normal, and expanded). Instead of storing each style as individual font files, a variable font interpolates them from a handful of “master” instances.

For example, even if the type designer did not explicitly create a semibold style, using a variable font with a weight axis, the text rendering engine will simply interpolate a semibold style for you (and any other weight you might need, assuming the variable font’s weight axis supports that range). Not only do variable fonts increase typographic expressiveness, they also offer a major benefit for web developers in terms of file size savings when several font variations are used. Variable fonts will however be larger than a single variation.

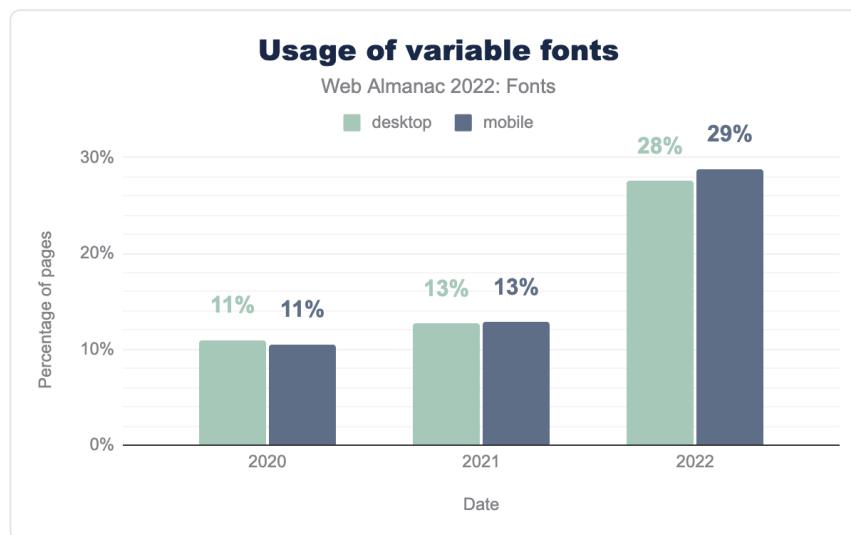


Figure 5.27. Usage of variable fonts.

Usage of variable fonts has nearly tripled since the last measurement in the Almanac’s 2020 Fonts chapter! Nearly 29% of websites use variable fonts. Most of this growth seems to have happened in the last year, with an amazing 125% growth.



Figure 5.28. Variable fonts used that are served by Google Fonts.

This impressive growth in usage can be explained by splitting the request data by host. Google Fonts accounts for 97% of variable fonts served, while everyone else accounts for only 3%. Even with Google's massive influence on web font serving, this growth can only be explained by replacing popular existing static styles with variable versions, rather than the introduction of completely new variable fonts.

As a result, Google Fonts and their users are probably seeing huge benefits in performance. Variable fonts are usually smaller than using multiple static instances. For example, if a website uses more than two or three styles from the same family, a variable font is smaller in file size and only takes a single request. It doesn't take a lot: using regular, bold, and light weights are usually sufficient reasons to use a variable font. As an added benefit, with a variable font you can also tweak the axes to suit your needs—semi-demi-bold anyone?

Regardless of a single actor being responsible for the growth, it is an amazing achievement, and a good indicator of the usefulness of variable fonts to optimize your site's performance.

Variable fonts also have two competing formats: the variable extensions of the `glyf` format and the Compact Font Format 2 (CFF2) format. The main differences between the `glyf` format and `CFF2` are the same as its `CFF` predecessor: different types of Bézier curves, more automated hinting, and a claim about smaller file sizes.



Figure 5.29. Variable fonts using the `glyf` outline format.

So which format should you use? Fortunately, for developers, type designers, and browser vendors the situation is quite simple. Out of all variable fonts served, 99.99% use the variable `glyf` format. Even if you exclude Google Fonts and other font services from the calculation the number changes to a whopping 99.98%. Nobody is using `CFF2`.

Our recommendation is to avoid `CFF2`-based variable fonts (for now, at least). Browsers and operating systems only recently added support for `CFF2`, and some browsers still don't support it. The only tangible benefit of using `CFF2` over `glyf` based variable fonts is the supposed file size savings, but as we've seen in the performance section this claim is dubious at

best.

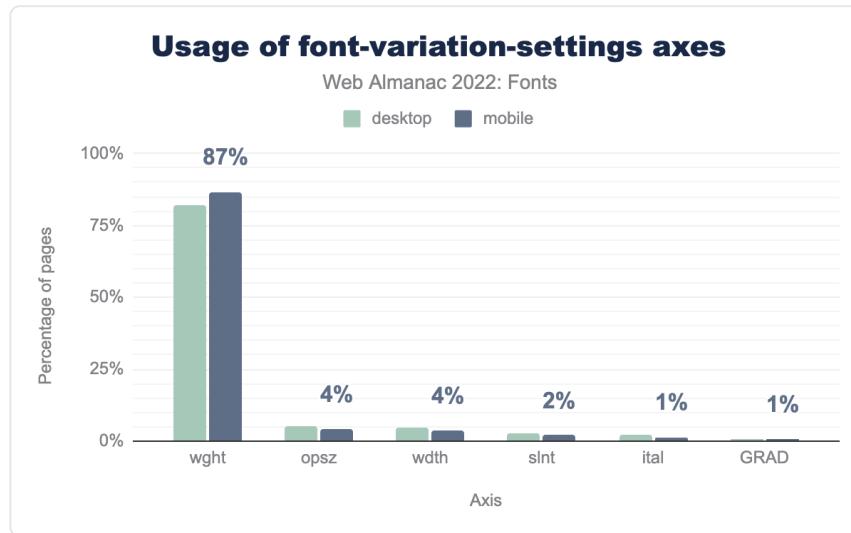


Figure 5.30. Usage of font-variation-settings axes.

So, how are people using variable fonts? Not surprisingly the weight axis is the most popular value used with the `font-variation-settings` property, followed by optical sizes, width, slant, italic, and grades.

This somewhat surprised us, because there is no need to use the low-level `font-variation-settings` property to set a custom weight axis value. You can simply use the `font-weight` property with a custom value, for example, `font-weight: 550` for a weight between 500 and 600.

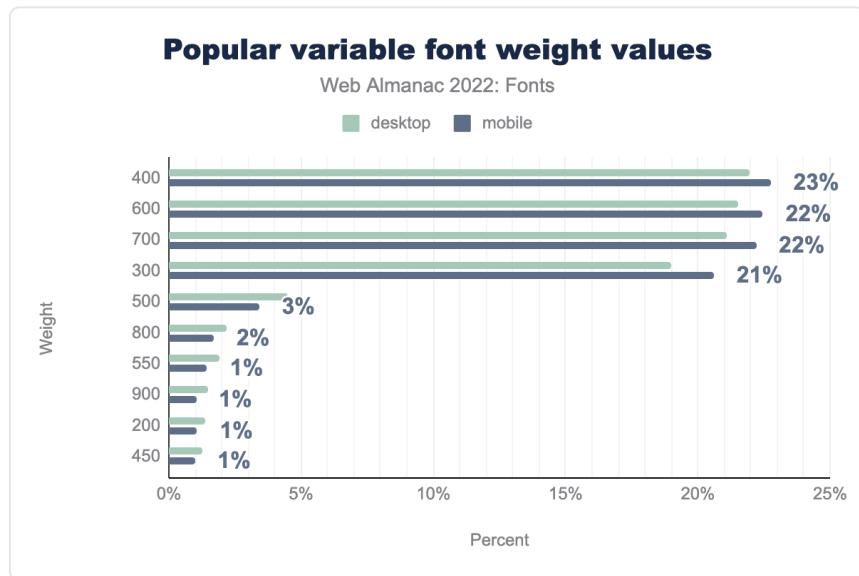


Figure 5.31. Popular variable font weight values.

Even more puzzling is that the most popular weight axis values are the “default” CSS weight values that have been around since the early days of CSS! Only 16% of the weight values are custom values along the weight range.

The most popular “custom” weight value is `550` at only 1.5% of use, followed by `450` at 1% use. Similar results can be seen for the optical size, width, italic, and slant axes, which can be set using the high-level `font-optical-sizing`, `font-stretch`, and `font-style` properties. Using the higher level properties will make your CSS more readable and avoid accidentally disabling an axis—a common source of errors with the low-level property.

One of the highly promoted benefits of variable fonts is animating one or multiple axes. Despite the high usage of variable fonts, very few people are actually animating them through CSS transitions or keyframes. Out of the entire HTTP Archive dataset, only a couple hundred websites do any sort of animation involving variable fonts.

To us, it appears that variable fonts are primarily used for their performance benefits, and less so for their ability to make typographic adjustments. While that’s great, we do hope to see more people use variable fonts to their full typographic potential in the coming years.

Color fonts

Color fonts are pretty much what you would expect: fonts with built in colors. Though the technology was originally created for emoji fonts, there are now more text color fonts than emoji fonts.

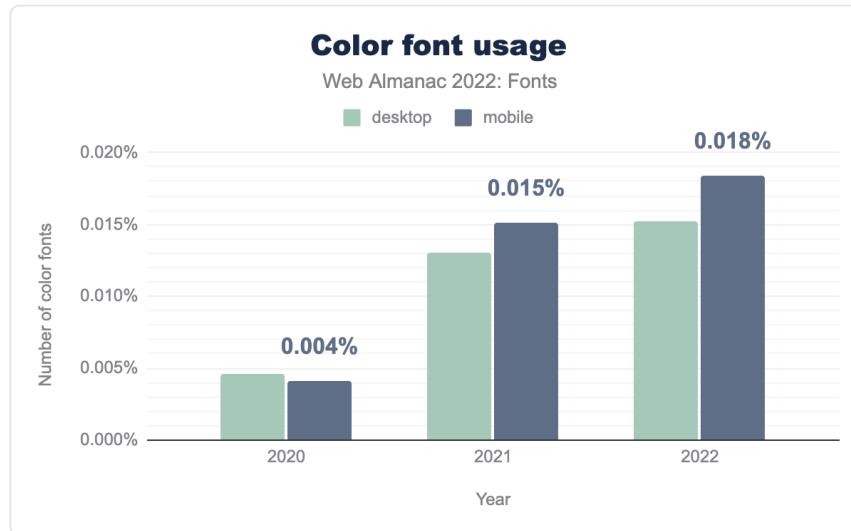


Figure 5.32. Color font usage.

Color fonts usage has grown quite a bit since the last Fonts chapter in 2020. Usage went from 0.004% of pages using color fonts in 2020 to about 0.018% in 2022. While those numbers are still very small, there is a clear growth in their usage.

However, compared to the growth in variable font usage, the limited uptake of color fonts is somewhat disappointing. While color fonts are a relatively new addition to the OpenType specification (2015), variable fonts are an even more recent addition (2016).

The primary factors that have severely hindered color font adoption (and might continue to do so) is the ongoing standards “battle” for the one true color font format, and the lack of support in browsers for the CSS that allows you to select and edit color font palettes—until recently.

There are currently four competing color font formats: two based on vector outlines (`SVG` and `COLR`) and two on images (`CBDT` and `sbix`). The `COLR` format re-uses the existing glyph outline and adds solid colors and layering to them. The most recent version, dubbed `COLRv1` introduced gradients, compositing and blending modes as well. Due to its re-use of existing glyph outlines, the `COLR` format also supports variable fonts, so you can have animated color

fonts¹⁷¹. The `SVG` format takes a different approach and essentially embeds an SVG image for each glyph in the font. Unfortunately, the `SVG` format does not support variable fonts, and is unlikely to do so in the future. Both `CBDT` and `sbix` embed images for each glyph and they only differ in the supported image formats.

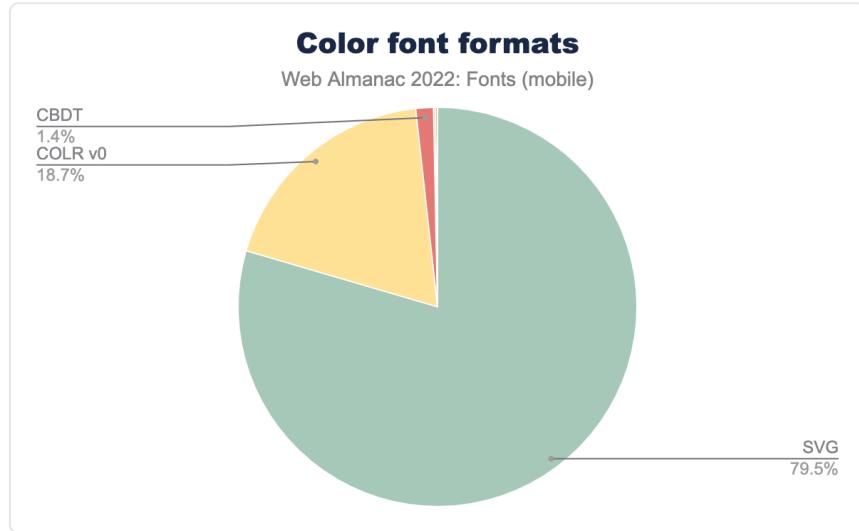


Figure 5.33. Color font formats.

Taking a look at usage data paints an interesting picture: 79% of color font usage is using `SVG`, 19% uses `COLRv0`, and 2% uses `CBDT`.

We can safely conclude that the image based formats are not popular, and for good reasons: the embedded images don't scale well, and their file sizes are not appropriate for web usage.

The split between the vector color font formats however is more nuanced. While `SVG` seems to have the upper hand at the moment, `COLR` still has significant usage. The `COLR` format has a lot going for it: it is supported by all browsers, it can be used in variable fonts, and it is easy to implement. For those reasons alone, we expect it to become the most popular format. A more cynical take is that it will become the most popular format because Google is refusing to implement `SVG` support in Chrome and Android. Interestingly, Apple is refusing to implement `COLRv1`, because a lot of `COLRv1` features are already supported by the `SVG` format. Unfortunately, web developers are caught in the middle of this "color fonts war". We hope this situation is soon resolved and we can all start using color fonts.

The CSS specification has been updated to support color fonts to allow selection and

171. <https://www.typearture.com/variable-fonts/>

customization of palettes¹⁷². Palettes are custom color schemes stored in the font by the type designer. The CSS `font-palette` property allows you to select a palette from the font and the `@font-palette-values` rule allows you to create new palettes or override existing ones. One of the more obvious use cases of this technology is to have light and dark mode palettes built right into the color font. There is a lot of unexplored potential there.



Figure 5.34. Bradley Initials¹⁷³ using COLR v1 and multiple palettes by David Jonathan Ross¹⁷⁴.

Unfortunately, usage of these CSS properties is currently nonexistent. This is likely because support for these properties was only recently added to browsers combined with the limited number of color fonts.

One of the main drivers behind the development of color font technology was emoji. However, there are only a couple dozen web fonts that have color emoji. Most color fonts are for writing text, not emoji. There could be several explanations for this:

- Every OS already includes their own color emoji font, so users don't feel the need to use anything else.
- There are a large number of emoji and it takes a lot of effort—and money—to create fonts for them.
- Emoji fonts are generally quite large and not as suitable as web fonts.

172. <https://css-tricks.com/colrv1-and-css-font-palette-web-typography/>

173. <https://tools.djr.com/misc/bradley-initials/>

174. <https://djr.com/>

Still, it would be nice to see some more diversity in emoji fonts. With the introduction of the COLR v1 format we're likely to see more emoji fonts in the future.

Again, all of this is based on very low usage numbers, but there appear to be some developing trends. We're not quite ready to declare 2023 the year of color fonts, but it seems clear we'll see significant color font growth in the coming years, especially as the industry settles on a single recommended color font format and browser support for color fonts improves. Google Fonts has also just added the first batch of color fonts¹⁷⁵ to their library, which will surely have an impact.

Conclusion

Looking back over this chapter and the previous years it stands out to us how much of an impact web font services have had—and likely will continue to have. For example, Google Fonts alone is responsible for most of web font usage, most of the popular web fonts, and most of variable fonts usage. That's an impressive feat.

While we strongly believe that self-hosting is the future for web fonts, it can not be denied that using a web font service makes a lot of sense for a lot of developers. They are easy to use, provide good out-of-the-box performance—though not the best—and for the most part you do not need to worry about font licensing. It is a good tradeoff.

On the other hand, self-hosting is now easier than ever, and will give you the best performance, more control, and no privacy headaches. If you plan to self-host, be sure to use WOFF2, resource hints, and `font-display`. Combined, they will have the biggest impact on the font loading performance of your site.

Variable fonts have taken off in a spectacular fashion in the last couple of years—thanks Google! While most people seem to be using them for performance reasons, we believe this is a case where adoption will drive innovation. We can't wait to see what kind of fun and downright crazy typography we'll see in the coming years.

We're cautiously optimistic about color fonts as well. Usage is finally growing. The technology has been there for a while, but the disagreements over color font formats and the limited CSS support have hindered adoption. We hope these will be resolved soon and we'll start seeing some real growth.

It is clear that web font usage will continue to grow and evolve over time. We're curious to see what the future holds. Technologies like Incremental Font Transfer¹⁷⁶ will unlock web fonts for

175. <https://material.io/blog/color-fonts-are-here>

176. <https://www.w3.org/TR/IFT/>

more writing systems, enabling billions of people to start using web fonts for the first time. That's exciting!

Author



Bram Stein

Twitter: [@bram_stein](https://twitter.com/bram_stein) GitHub: [bramstein](https://github.com/bramstein) Website: <http://www.bramstein.com/>

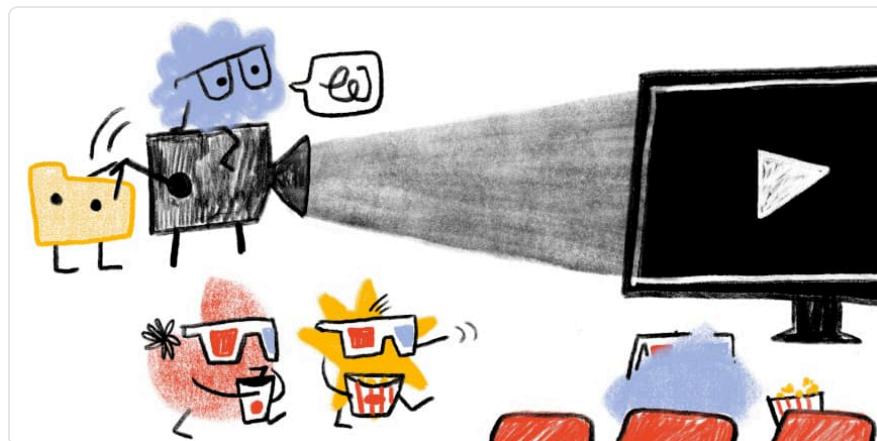
Bram Stein is a developer and product manager. He cares a lot about typography and is happiest working at the intersection between design and technology. He is the author of the Webfont Handbook¹⁷⁷ by A Book Apart and the FontFace Observer¹⁷⁸ library. He also speaks about typography and web performance at conferences around the world.

177. <https://abookapart.com/products/webfont-handbook>

178. <https://fontfaceobserver.com>

Part I Chapter 6

Media



Written by Eric Portis and Akshay Ranganath

Reviewed by Nicolas Hoizey and Yoav Weiss

Analyzed by Eric Portis and Akshay Ranganath

Edited by Michael Lewittes

Introduction

Despite being hypertext, the web is extremely visual. In fact, images and videos are an essential part of the web user experience. They're also undergoing tremendous innovation, and the Web Almanac gives us a unique opportunity to survey both how far the visual web has come—as authors adopt new technologies such as AVIF, wide color, adaptive bitrate streaming, and lazy loading—and how far it still has to go: I'm looking at you, animated GIF.

Let's dive right in.

Images

Images account for a huge portion of the typical website's page weight. We see from the Page Weight chapter that the median website's total weight in June of 2021 was 2,019 kilobytes (on mobile), and 881 of those kilobytes were images. That's more than HTML (30 KB), CSS (72 KB),

JavaScript (461 KB) and fonts (97 KB) combined.

99.9%

Figure 6.1. Pages that generated at least one request for an image resource.

Almost every page serves up some kind of an image, even if it's just a background or favicon.

70%

Figure 6.2. Mobile pages whose LCP responsible element has an image.

On the vast majority of pages—70% on mobile, and 80% on desktop—the most impactful resource is an image. Largest Contentful Paint¹⁷⁹ (LCP) is a web performance metric that identifies the largest element above the fold. Most of the time that element has an image.

It's hard to overstate the importance of images on the web. So, what can we say about the web's images?

Image resources

Let's start with the resources themselves. Bitmap images are made of pixels. How many pixels do the web's images typically have?

A note on single-pixel images

Client 1x1 images	
Mobile	7.3%
Desktop	7.0%

Figure 6.3. Resources loaded by elements that contain just a single pixel.

A suspiciously large number of them are 1×1. These s don't contain any image content

¹⁷⁹. <https://web.dev/lcp/>

at all. Instead, they're being used for two purposes: for layout (as spacer GIFs¹⁸⁰) or as tracking beacons¹⁸¹.

Any newly authored website should use CSS for layout and the Beacon API¹⁸² for tracking. Lots of existing content will use tracking pixels and spacer GIFs forever, but it's disheartening that the desktop number here is unchanged from last year¹⁸³, and that the mobile number has only shrunk by a tiny amount. Old habits¹⁸⁴ die hard¹⁸⁵!

Wherever possible, we excluded these not-really-an-image ``s from our analysis.

Image dimensions

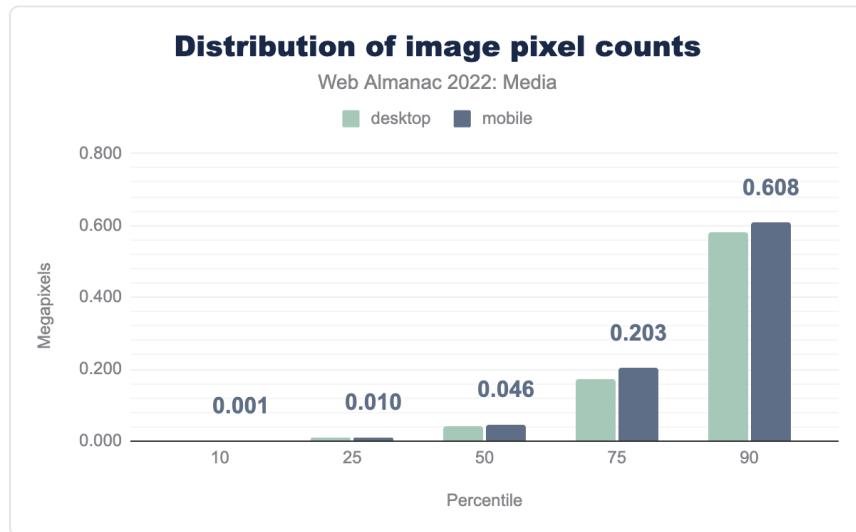


Figure 6.4. Distribution of image pixel counts.

Moving on to images that contain more than one pixel: Most of them are fairly small, but the majority of pages also contain at least one big image.

“Megapixels” aren't the most intuitive measure of image size. For perspective, at a 4:3 aspect ratio, the median pixel count of 0.046MP works out to a 248×186 image.

That may seem small, but the median page includes at least one `` that contains almost 10

180. https://en.wikipedia.org/wiki/Spacer_GIF

181. https://en.wikipedia.org/wiki/Web_beacon

182. https://developer.mozilla.org/docs/Web/API/Beacon_API

183. <https://almanac.httparchive.org/en/2021/media#fig-5>

184. <https://developers.facebook.com/docs/meta-pixel/implementation/marketing-api#initialize-img>

185. <https://spacergif.org/stats/>

times more pixels than the median `` element.

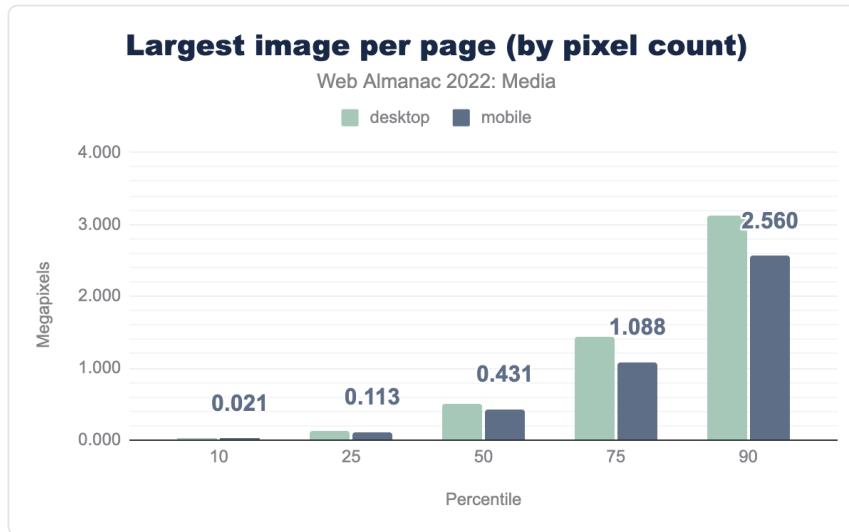


Figure 6.5. Largest image per page (by pixel count).

At the aspect ratio of 4:3, 0.431MP works out to 758×569. Considering the mobile crawler has a (typical) 360px-wide viewport, it's likely that many of these large images end up painted across almost the whole viewport and at high densities.

In short: most images are small, but most pages include at least one big image.

Image aspect ratios

What sorts of aspect ratios are common on the web?

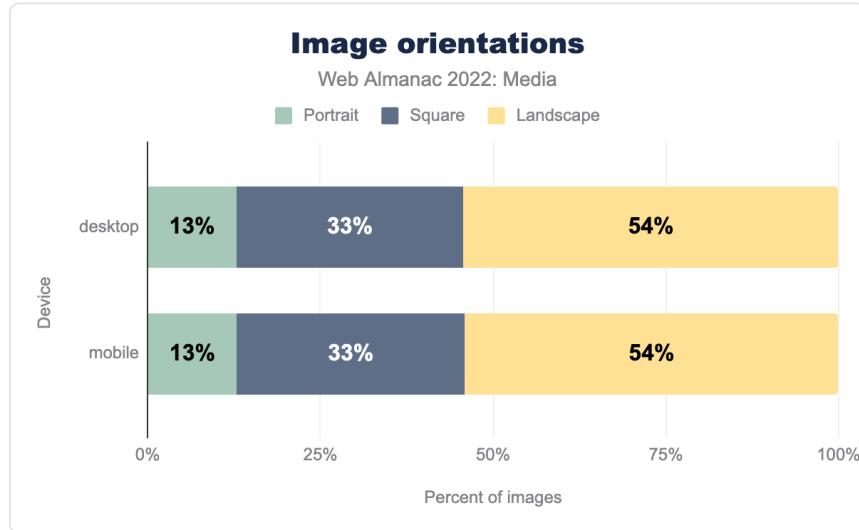


Figure 6.6. Image orientations.

Much like last year¹⁸⁶, most images are landscape-oriented, and there is virtually no difference between the mobile and desktop numbers. Similar to 2021, this feels like a huge missed opportunity. As many-an-Instagrammer knows, portrait-oriented images render larger on mobile screens¹⁸⁷ at full-width than either square or landscape-oriented images do, and drive higher engagement¹⁸⁸. Even when the source material is landscape-oriented, we can and should try to tailor images for mobile screens, using art direction¹⁸⁹.

186. <https://almanac.httparchive.org/en/2021/media#aspect-ratios>
 187. <https://uxdesign.cc/the-powerful-interaction-design-of-instagram-stories-47cdeb30e5b6>
 188. <https://www.dashhudson.com/blog/best-picture-format-instagram-dimensions>
 189. <https://web.dev/codelab-art-direction/>

Aspect ratio	% of images
1:1	32.92%
4:3	3.99%
3:2	2.74%
2:1	1.66%
16:9	1.62%
3:4	1.02%
2:3	0.72%
5:3	0.54%
6:5	0.48%
8:5	0.47%

Figure 6.7. A ranked list of the top ten image aspect ratios (mobile).

Images' aspect ratios were clustered around "standard" values, such as 4:3, 3:2 and, in particular, 1:1 (square). In fact, 40% of all images had one of those three aspect ratios, and the top 10 aspect ratios accounted for nearly half of all ``s.

Image color spaces

Images are made of pixels and each pixel has a color. The range of colors that are possible within a given image is determined by that image's color space¹⁹⁰.

The default color space on the web is sRGB¹⁹¹. CSS colors are specified in sRGB by default and—unless they're marked otherwise—browsers assume that the colors in images are sRGB, too¹⁹².

This made sense in a world where approximately all display and capture hardware dealt in sRGB—or something close to it. But the times, they are a-changin'. In 2022, most phone cameras capture in wider-than-sRGB gamuts. Also, display hardware capable of richer, outside-of-sRGB colors is now quite common.

Every modern browser that's painting to a wide-gamut display will happily paint vibrant,

190. https://en.wikipedia.org/wiki/Color_space

191. <https://en.wikipedia.org/wiki/sRGB>

192. <https://imageoptim.com/color-profiles.html>

outside-of-sRGB colors, if we encode our images using wider than sRGB gamuts. But are we?

In short: No.

In order to tell a browser that an image uses a non-sRGB color space, authors must generally attach an ICC profile¹⁹³ to it that describes the image's color space. Those ICC profiles have names. We found a little more than 25,000 unique ICC profile names in use on the web. Here are the top 20:

^{193.} https://en.wikipedia.org/wiki/ICC_profile

ICC profile description	sRGB-ish	Wide-gamut	% of images
Untagged	✓		90.17%
sRGB IEC61966-2.1	✓		3.23%
c2ci	✓		2.40%
sRGB	✓		0.88%
Adobe RGB (1998)		✓	0.76%
uRGB	✓		0.54%
Display P3		✓	0.35%
c2	✓		0.33%
Display			0.30%
sRGB built-in	✓		0.24%
GIMP built-in sRGB	✓		0.22%
sRGB IEC61966-2-1 black scaled	✓		0.19%
Generic RGB Profile			0.06%
U.S. Web Coated (SWOP) v2			0.04%
sRGB MozJPEG	✓		0.02%
Artifex Software sRGB ICC Profile	✓		0.02%
Dot Gain 20%			0.02%
Coated FOGRA39 (ISO 12647-2:2004)			0.01%
Apple Wide Color Sharing Profile		✓	0.01%
sRGB v1.31 (Canon)	✓		0.01%
HD 709-A	✓		0.01%

Figure 6.8. A ranked list of the top twenty ICC color space descriptions (mobile).

Nine out of ten images on the web are untagged, meaning that if they contain RGB data, it will be interpreted as sRGB. Most of the remaining 10% are explicitly tagged with sRGB or something similar to it: “c2ci”, “uRGB”, and “c2” are all sRGB variants designed to be minimal and

“lightweight”¹⁹⁴. Just a little more than 1% of all of the web’s images have been tagged with a wider-than-sRGB gamut. More succinctly, wide-gamut images are currently about as popular on the web as grayscale images—which account for 1.16% of the web’s images¹⁹⁵.

One caveat: AVIF and PNG allow tagging images with wide-gamut color spaces using format-specific shorthands, without using ICC profiles. We started down the path of trying to detect wide-gamut AVIFs and PNGs that don’t use ICC profiles, but accounting for the various ways they are encoded—and the ways our tooling reported on them—proved a bit too complex to tackle this year. Maybe next year!

Encoding

Now that we’ve gleaned a bit about the web’s image content, what can we say about how that content is encoded for delivery?

Format adoption

GIF, JPEG, and PNG have been the standard bitmap image file formats on the web for decades. That started to change when Chrome shipped support for WebP in 2014. Over the past couple of years that change has accelerated. Safari and Firefox have now shipped WebP support, and all three major browsers have shipped at least experimental support for AVIF.

By format, here’s every image resource the crawler saw:

194. <https://github.com/saucecontrol/Compact-ICC-Profiles>

195. <https://docs.google.com/spreadsheets/d/1T5oVAVmch3sM6R-VwH4ksr2jFtPhuLx3-iXxAbB3E/edit?pli=1#gid=560546690&range=P5>

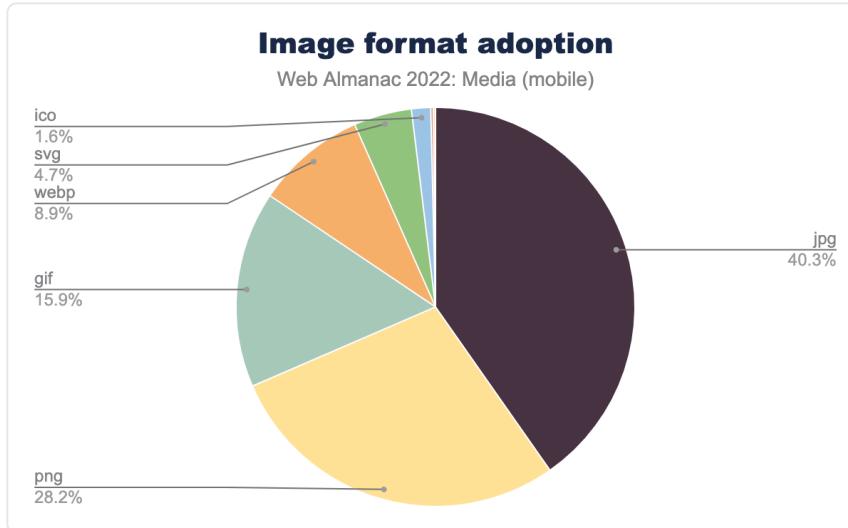


Figure 6.9. Image format adoption.

At 0.22%, AVIF's slice of that pie is so small it's not even labeled on the chart. And while 0.22% may not sound like a lot, compared to last year, it represents quite a bit of progress.

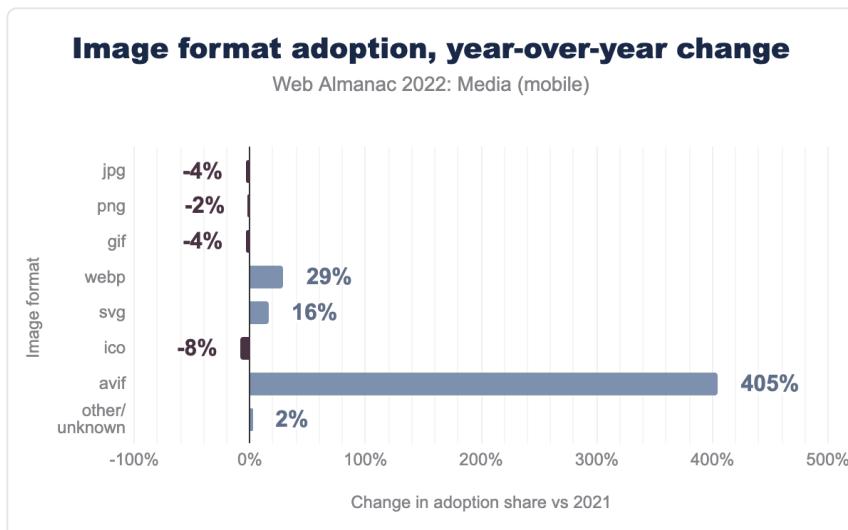


Figure 6.10. Image format adoption, year-over-year change.

Slowly, the old formats are making way for the new ones. As they should! The new formats

outperform the old ones by a significant margin. We'll get a sense of that shortly.

Bytesizes

How heavy is the typical image on the web?

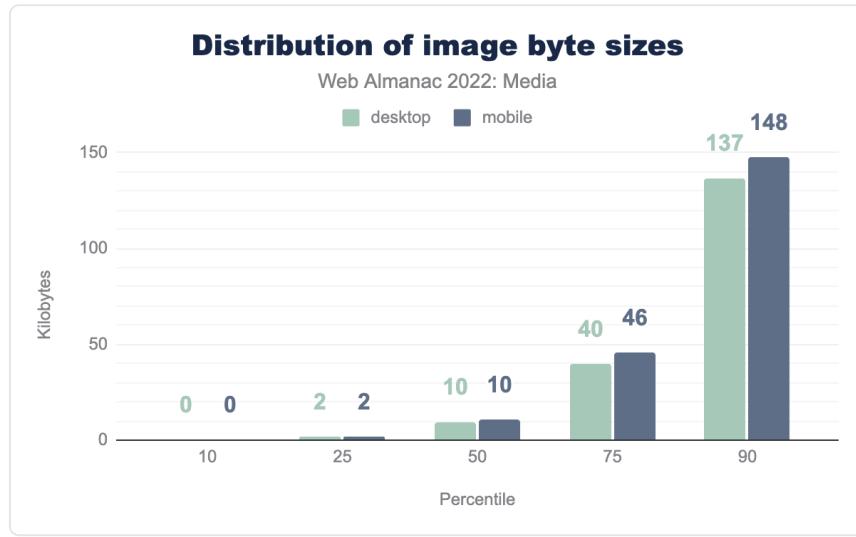


Figure 6.11. Distribution of image byte sizes.

A median of 10 KB might lead one to think, "Eh, not that heavy!" But, just as when we looked at pixel counts, while there are many small images, most pages have at least one large one:

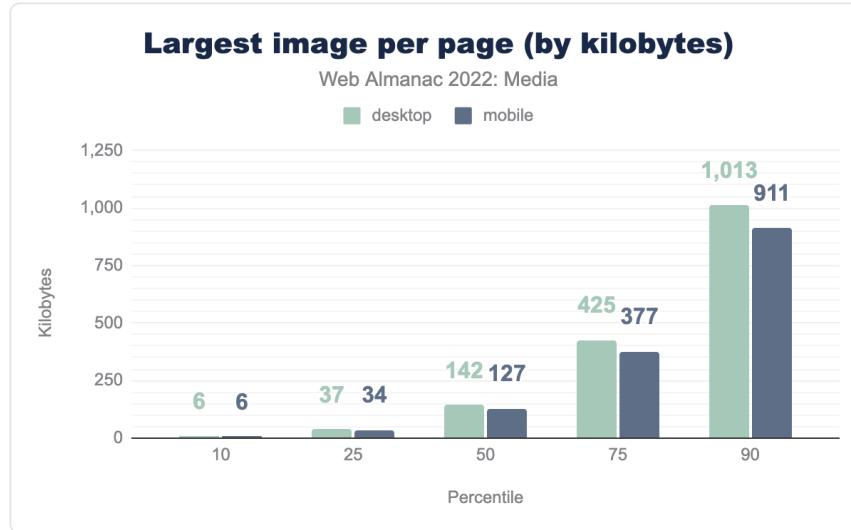


Figure 6.12. Largest image per page (by kilobytes).

Most pages have at least one image over 100 KB, and the top 10% of pages have at least one image that weighs almost 1 MB or more.

Bits per pixel

Bytes and pixel counts are interesting on their own, but to get a sense of how compressed the web's image data is, we need to put bytes and pixels together to calculate bits per pixel. Doing this allows us to make apples-to-apples comparisons of the information density of images, even if those images have different resolutions.

In general, bitmaps on the web decode to eight bits of information per channel, per pixel. So, if we have an RGB image with no transparency, we can expect a decoded, uncompressed image to weigh in at 24 bits per pixel¹⁹⁶. A good rule of thumb for lossless compression is that it should reduce file sizes by a 2:1 ratio (which would work out to 12 bits per pixel for our 8-bit RGB image). The rule of thumb for 1990s-era lossy compression schemes—JPEG and MP3—was a 10:1 ratio (2.4 bits per pixel). It should be noted that, depending on image content and encoding settings, these ratios vary widely, and modern JPEG encoders like MozJPEG¹⁹⁷ typically outperform this 10:1 target at their default settings. To summarize:

196. [https://en.wikipedia.org/wiki/Color_depth#True_color_\(24-bit\)](https://en.wikipedia.org/wiki/Color_depth#True_color_(24-bit))

197. <https://github.com/mozilla/mozjpeg>

Type of bitmap data	Expected compression ratio	Bits per pixel
Uncompressed RGB	1:1	24 bits/pixel
Losslessly compressed RGB	~2:1	12 bits/pixel
1990s-era lossy RGB	~10:1	2.4 bits/pixel

Figure 6.13. Typical compression ratios and resulting bits/pixel numbers for bitmap RGB data.

So, with all of that as context, here's how the web's images stack up:

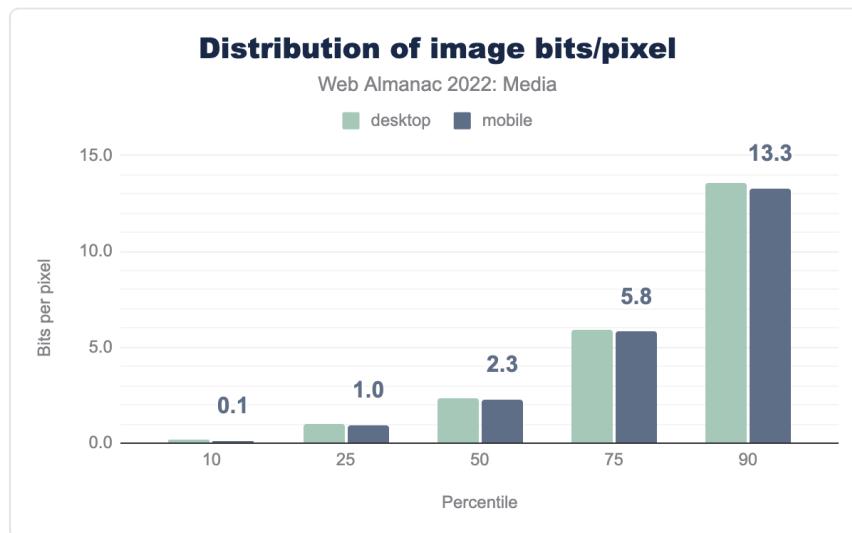


Figure 6.14. Distribution of image bits/pixel.

At 2.3 bits per pixel, the median `` on mobile almost hits that 10:1 compression ratio target on the nose. However, around that median, there is a tremendous spread. Let's break things down by format in order to learn a bit more.

Bits per pixel, by format

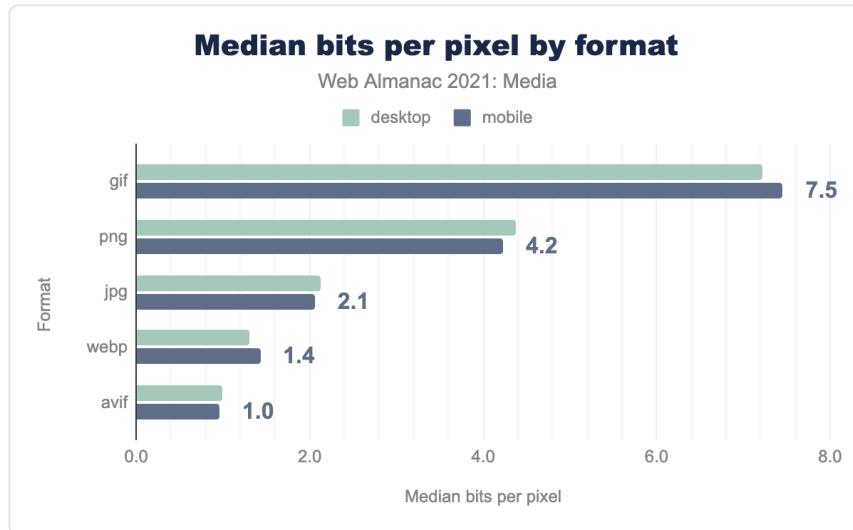


Figure 6.15. Median bits per pixel by format.

Most of these numbers are essentially unchanged from last year¹⁹⁸.

We can again see that while PNG technically employs “lossless” compression techniques (which would lead us to expect 12-16 bits per pixel, depending on whether or not we’re dealing with an alpha channel), its encoders are generally lossy. They reduce color palettes and introduce dithering patterns before “losslessly” compressing images in order to boost compression ratios.

And we again see that the typical WebP is one-third lighter, per pixel, than the typical JPEG. This is about what we would expect: Formal studies¹⁹⁹, which, vitally, use matched qualities²⁰⁰, have estimated that WebP outperforms JPEG by about that same margin.

The only big mover, when compared to last year, is AVIF. The format dropped from 1.5 bits per pixel last year—less compressed than WebP already!—all the way down to 1.0. This is a huge reduction, but it wasn’t entirely unexpected. AVIF is a very young format whose encoders have been quickly iterating, and whose adoption is significantly broadening. I expect next year the median AVIF will be even more compressed.

Without looking at the quality side of the lossy-compression/quality tradeoff, it’s not possible to conclude from these results alone that AVIF offers the “best” compression of all of the web-

198. <https://almanac.httparchive.org/en/2021/media#bits-per-pixel-by-format>

199. https://developers.google.com/speed/webp/docs/webp_study

200. <https://kornel.ski/en/faircomparison>

compatible formats. But this year we can conclude that in real-world usage, it exhibits the most compression. Pair that conclusion with in-the-lab results²⁰¹, which suggest it also does a good job of preserving quality, and the picture starts looking pretty good—pun intended.

AVIF's browser support²⁰² also took a huge leap this year. All of this is to say, if you're sending bitmap images across the web—as you may recall, 99.9% of pages do—you should at least consider sending AVIFs.

GIFs, animated and not

At the other end of the compression chart is our old friend GIF. It comes out looking particularly bad, but it's not all the format's fault. One of the reasons this 35-year-old format is still in common use is its ability to do animation, and we have not accounted for the number of frames when calculating the number of pixels. This raises a few interesting questions. First, how many GIFs are animated?



Figure 6.16. Percentage of GIFs that were animated on mobile.

I found this surprisingly low. Ever since PNG achieved universal support in 2006²⁰³, there hasn't been a good reason to ship a non-animated GIF²⁰⁴. The word "GIF" has become synonymous with its only justifiable use case: Being a portable, universal format for short, silent, autoplaying, looping animation. One wonders whether all of these non-animated GIFs are legacy content, or whether there are a significant number of new, non-animated GIFs being created and published to the web—I hope not!

Now that we've separated out the animated GIFs from the non-animated ones, we can also ask: What are the compression characteristics of non-animated vs animated GIFs?

201. <https://netflixtechblog.com/avif-for-next-generation-image-coding-b1d75675fe4>

202. <https://caniuse.com/avif>

203. <https://caniuse.com/?search=png>

204. https://en.wikipedia.org/wiki/Portable_Network_Graphics#Compared_to_GIF

GIF bits per pixel: animated vs. non-animated

Web Almanac 2022: Media

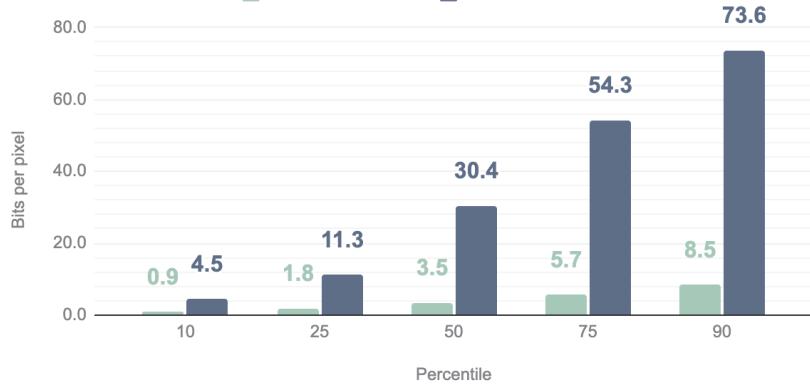



Figure 6.17. GIF bits per pixel: animated vs. non-animated.

Once we remove animated GIFs from the equation, the format looks much better. At a median of 3.5 bits per pixel, GIFs are smaller, pixel-for-pixel, than PNGs. This likely reflects the kinds of content that each format is asked to compress: GIFs, by design, can only contain 256 colors and binary transparency. PNGs can contain 16.7 million colors plus a full alpha channel.

Before we move on from GIFs, I do have one more question about them: How many frames do animated GIFs typically have?

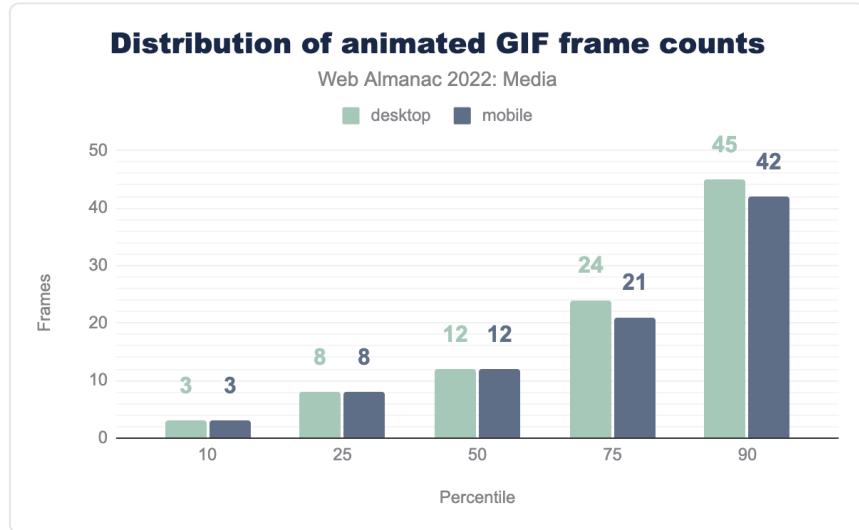


Figure 6.18. Distribution of animated GIF frame counts.

A majority of animated GIFs come in at a dozen frames or less. Incidentally, the most frames we found in a GIF was 15,341. At 30 FPS, that would work out to an eight-and-a-half-minute GIF. The mind reels.

Embedding

Now that we have a sense of how the web's image resources have been encoded, what can we say about how they are embedded on web pages?

Lazy-loading

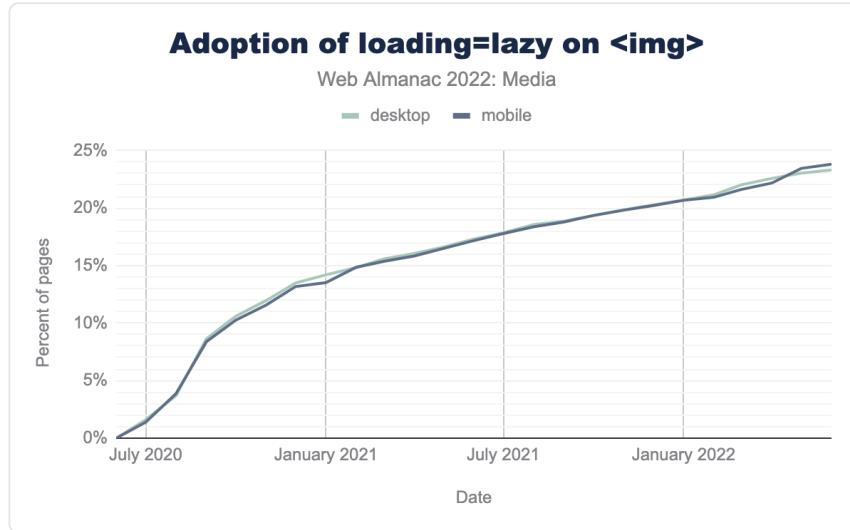


Figure 6.19. Adoption of `loading=lazy` on ``.

The biggest story last year was the rapid adoption of lazy-loading. While the pace of adoption has slowed, it's still proceeding at a remarkable rate. Last June, 17% of pages were using lazy-loading. This year, we saw a 1.4x increase. Now 24% of pages are using lazy-loading.

Given the vast amount of legacy content on the web, going from zero adoption to just about one-quarter of crawled pages within two years is a remarkable feat, and shows just how much demand there was for native lazy-loading.

9.8%

Figure 6.20. Percentage of LCP ``s that use native lazy-loading on mobile.

And indeed, just like last year, it seems pages are using lazy-loading a bit too much.

Lazy-loading LCP elements makes LCP scores much worse. It's an anti-pattern that makes pages slower. Seeing that one-in-ten LCP ``s are lazy-loaded is disheartening. Seeing that this anti-pattern has gotten slightly more common since last year is even more so.

alt text

Images embedded with `` elements are supposed to be contentful. That is to say: They're not just decorative, and they should contain something meaningful. According to both WCAG requirements²⁰⁵ and the HTML spec²⁰⁶, all contentful images must have alternative text, and that alternative text should usually be supplied by the `alt` attribute.

54%

Figure 6.21. Percentage of images that had a non-blank `alt` attribute.

This result means that almost half of all ``s are obviously inaccessible. If the in-depth analysis from this year's accessibility chapter is any indication, a large chunk of the ``s that do have non-blank `alt` attributes aren't all that accessible, either.

We can and must do better.

srcset

Prior to lazy-loading, the biggest thing to happen to ``s on the web was a suite of features for "responsive images," which allowed images to tailor themselves to fit within responsive designs. First shipped in 2014, the `srcset` attribute, `sizes` attribute, and the `<picture>` element have allowed authors to mark up adaptable resources for almost a decade now. How much and how well are we using these features?

Let's start with the `srcset` attribute, which allows authors to give the browser a menu of resources to choose from, depending on context.

34%

Figure 6.22. Percentage of pages using the `srcset` attribute.

One-third of pages use `srcset`, but two-thirds don't. Given the prevalence of fluid grids within responsive designs in 2022, I suspect there are a lot of pages that aren't using `srcset` that should be.

205. <https://www.w3.org/WAI/WCAG22/Understanding/non-text-content>

206. <https://html.spec.whatwg.org/multipage/images.html#alt>

The `srcset` attribute allows authors to describe resources using one of two descriptors: `x` descriptors that specify which screen density a resource is appropriate for, and `w` descriptors, which instead give the browser the resource's width in pixels. Used in conjunction with the `sizes` attribute, `w` descriptors allow browsers to select a resource appropriate for both fluid layout widths and variable screen densities.

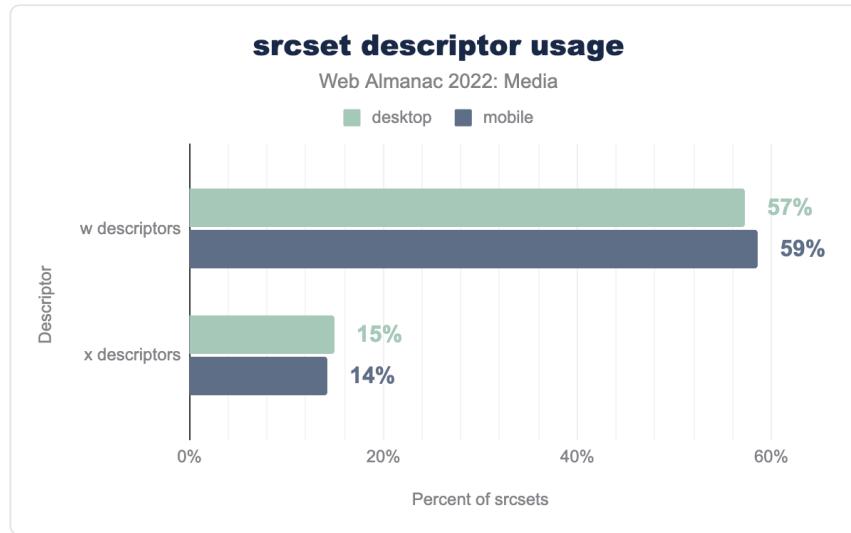


Figure 6.23. `srcset` descriptor usage.

The `x` descriptor came first, and is simpler to reason about. For years it enjoyed more popularity than the more powerful `w` descriptor. It warms my heart that nearly a decade in, the world has come around to `w` descriptors.

sizes

I mentioned earlier that `w` descriptors should be used in conjunction with `sizes` attributes. How well are we using `sizes`? In two words: Not very.

The `sizes` attribute is supposed to be a hint to the browser about the eventual layout size of the image, usually relative to the viewport width. There are many variables that can affect an image's layout width. The `sizes` attribute is explicitly supposed to be a hint, and so a little inaccuracy is OK and even expected. But if the `sizes` attribute is more-than-a-little inaccurate, it can affect resource selection, causing the browser to load an image to fit the `sizes` width when the actual layout width of the image is significantly different.

So how accurate are our `sizes`?

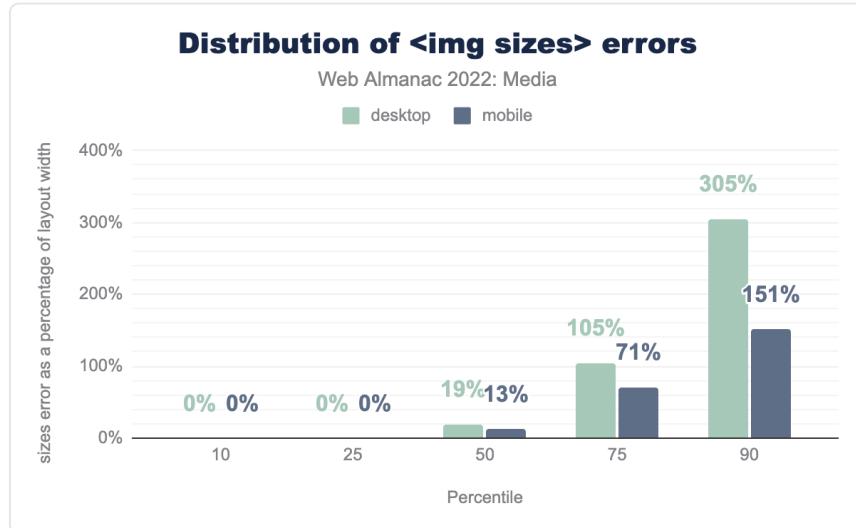


Figure 6.24. Distribution of `` errors.

While many `sizes` attributes are entirely accurate, the median `sizes` attribute is 13% too-large on mobile and 19% too-large on desktop. That might be OK, given the hint-like nature of the feature, but as you can see, the p75 and p90 numbers aren't pretty and lead to bad outcomes.

19%

Figure 6.25. `sizes` attributes that were inaccurate enough to affect `srcset` selection on desktop. On mobile, it's 14%.

On desktop, where the difference between the default `sizes` value (`100vw`) and the actual layout width of the image is likely to be larger than on mobile, one-in-five `sizes` attributes is inaccurate enough to cause browsers to pick a suboptimal resource from the `srcset`. These errors add up.

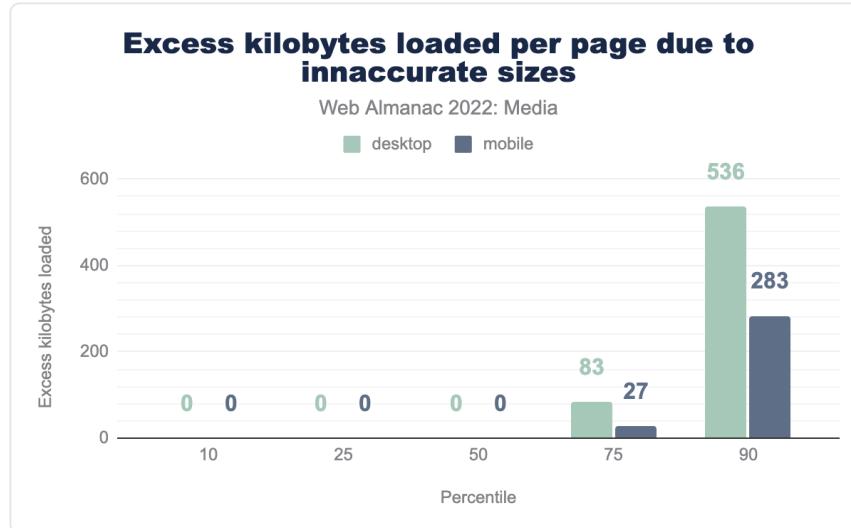


Figure 6.26. Excess kilobytes loaded per page due to inaccurate `sizes`.

We estimate that one-quarter of desktop pages are loading more than 83 KB of extra image data, based purely on bad `sizes` attributes. That is to say: A better, smaller resource is there for the picking in the `srcset`, but because the `sizes` attribute is so erroneous, the browser doesn't pick it. Additionally, 10% of desktop pages that use `sizes` load more than a half-megabyte of excess image data because of bad `sizes` attributes!

Note: Our crawlers didn't actually load the correct resources, so the numbers here are estimates, based in part on the byte sizes of the incorrect resources, which the crawlers actually did load.

In the short term, individual developers can and should use `ResplImageLint`²⁰⁷ to audit and fix their badly broken `sizes` attributes and prevent this kind of waste.

In the medium term, where possible, the web platform needs to provide better tooling. For many developers, authoring—and maintaining!—accurate `sizes` attributes has proven to be too hard. A proposal that would allow automatic sizes for lazy-loaded images²⁰⁸ is on the table. Let's hope it progresses in 2023.

The `lazysizes.js` library²⁰⁹ has already proven the appetite for this sort of solution: 10% of `sizes` attributes currently have the value “auto” before JavaScript runs and are later rewritten to perfectly accurate values by `lazysizes.js` before it lazy-loads the image. Note that, because it relies on lazy-loading, this pattern is not appropriate for LCP images or any ``

207. <https://ausi.github.io/respimagelint/>

208. <https://github.com/whatwg/html/pull/8008>

209. <https://github.com/aFarkas/lazysizes>

elements that are above the fold. For these images, the only way forward for performant responsive loading is a well-authored `sizes` attribute.

<picture>

The last responsive image feature to land in 2014 was the `<picture>` element. While `srcset` hands browsers a menu of resources to choose from, the `<picture>` element allows authors to take charge, giving browsers an explicit set of instructions about which child `<source>` element to load a resource from.

The `<picture>` element is used far less than `srcset`.



Figure 6.27. Percentage of mobile pages that use the `<picture>` element.

This is up a couple ticks from last year, but the fact that there are almost five pages that use `srcset` for every one page that uses `<picture>` suggests that either `<picture>`'s use cases are more niche, or it's more difficult to deploy—or both.

What are people using `<picture>` for?

The `<picture>` element gives authors two ways to switch between resources. Type-switching allows authors to provide cutting-edge image formats to browsers that support them and fallback formats for everyone else. Media-switching facilitates art direction²¹⁰, allowing authors to switch between various `<source>`s based on media conditions.

210. <https://www.w3.org/TR/resping-usecases/#art-direction>

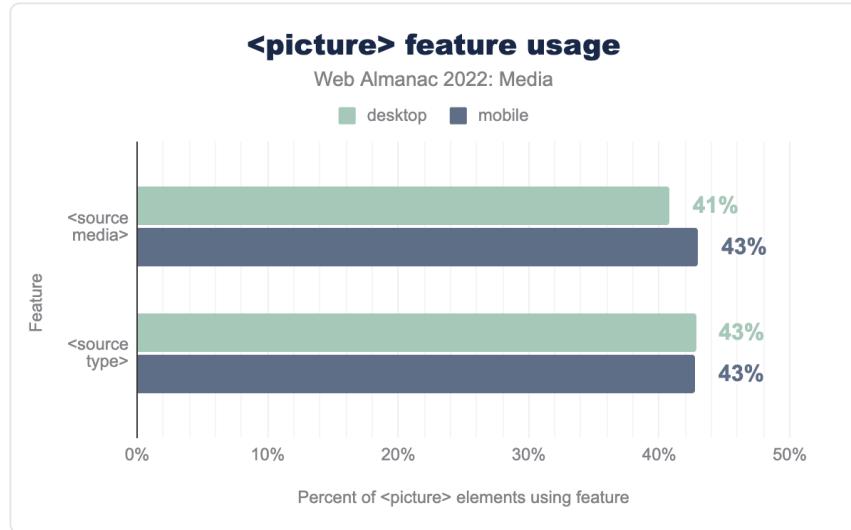


Figure 6.28. `<picture>` feature usage.

Usage is split reasonably evenly. Interestingly, type-switching has closed the gap since last year²¹¹. This may be related to the increasing popularity of next-generation image formats like AVIF and WebP.

Layout

Part of what makes responsive images difficult is that it asks us to think about how ``s will be laid out, while writing HTML. Which leads us to a basic question: How are ``s laid out?

We already saw how the web's image resources size up. But before they can be shown to a user, embedded images must be placed within a layout and potentially squished or stretched to fit it.

Throughout this analysis it will be useful to keep in mind the crawlers' viewports: The desktop crawler was 1376px-wide, with a DPR of 1x; the mobile crawler was 360px-wide, with a DPR of 3x.

Layout widths

The simplest question here might be: How wide do the web's images end up when painted to the page?

²¹¹ <https://almanac.httparchive.org/en/2021/media#fig-23>

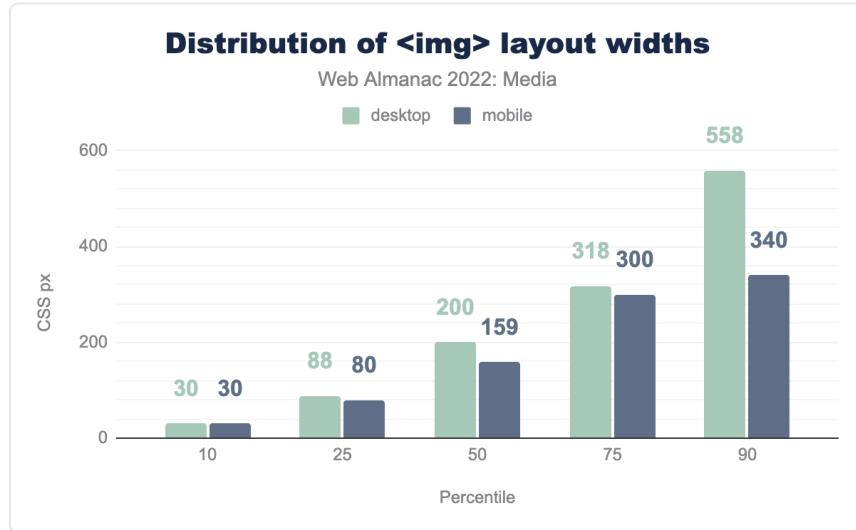


Figure 6.29. Distribution of layout widths.

Just like the resources that they embed, most of the web's images end up pretty small within layouts. Similarly, most pages have at least one fairly large image.

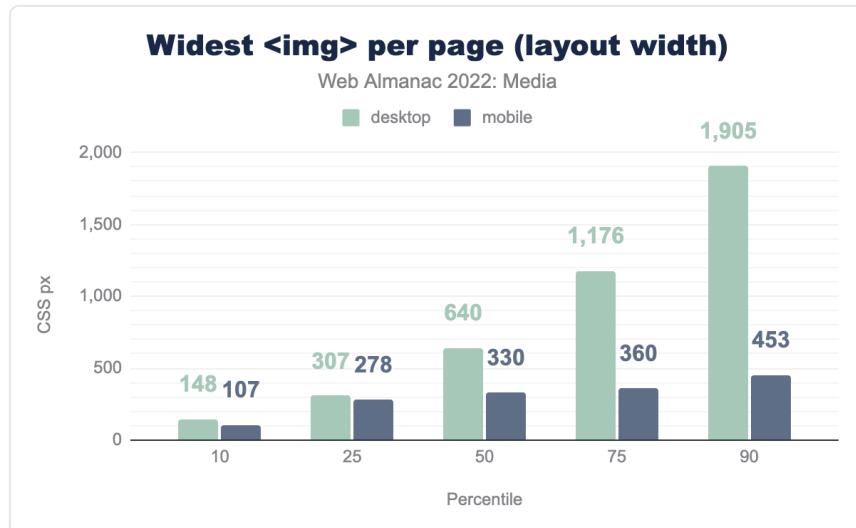


Figure 6.30. Widest per page (layout width).

More than 75% of mobile pages have at least one image that takes up more than 75vw worth of

viewport. From there, things more or less top out, rising slowly until a significant number (somewhere between 10-25%) of pages have an image that ends up wider than the viewport. That's likely because the author has not included a viewport meta tag²¹² and the desktop-sized page is being scaled down to fit within the mobile screen.

It's interesting to contrast this with the desktop layout widths, which don't top out at all. They just keep growing. I find it surprising that more than 10% of pages on desktop included an image that was wider than the crawler's 1360px viewport, presumably triggering horizontal scrollbars.

Intrinsic vs extrinsic sizing

Why do the web's images end up at these layout sizes? There are many ways to scale an image with CSS. But how many images are being scaled with any CSS at all?

Images, like all "replaced elements"²¹³, have an intrinsic size²¹⁴. By default—in the absence of a `srcset` controlling their density or any CSS rules controlling their layout width—images on the web display at a density of 1x. Plop a 640×480 image into an `` and, by default, that `` will be laid out with a width of 640 CSS pixels.

Authors may apply extrinsic sizing to an image's height, width, or both. If an image has been extrinsically sized in one dimension (e.g., with a `width: 100%` rule), but left to its intrinsic size in the other (`height: auto;` or no rule at all), it will scale proportionally, using its intrinsic aspect ratio.

Complicating things further, some CSS rules allow ``s to appear at their intrinsic dimensions, unless they violate some constraint. For instance, an `` element with a `max-width: 100%;` rule will be intrinsically sized, unless that intrinsic size is larger than the size of the `` element's container, in which case it will be extrinsically scaled down to fit.

With all of that explanation out of the way, here's how the web's `` elements are sized for layout:

212. https://developer.mozilla.org/docs/Web/HTML/Viewport_meta_tag

213. https://developer.mozilla.org/docs/Web/CSS/Replaced_element

214. https://developer.mozilla.org/docs/Glossary/Intrinsic_Size

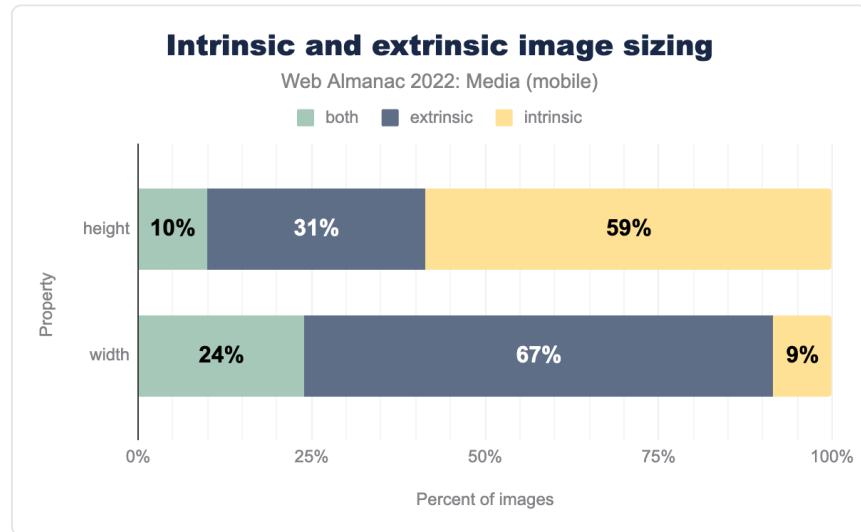


Figure 6.31. Intrinsic and extrinsic image sizing.

The majority of images have extrinsic widths; the majority of images have intrinsic heights. The “both” category for width—representing images with either a `max-width` or `min-width` sizing constraint—is also fairly popular. Leaving images to their intrinsic widths is far less popular—and slightly less popular than it was last year²¹⁵.

`height`, `width`, and Cumulative Layout Shifts

Any `` whose layout size is dependent on its intrinsic width risks triggering a Cumulative Layout Shift²¹⁶. In essence, such images risk being laid out twice: Once when the page’s DOM and CSS have been processed, and then a second time when they finally finish loading and their intrinsic dimensions are known.

As we’ve just seen, extrinsically scaling images to fit a certain width while leaving the height (and aspect ratio) intrinsic is very common. To fight the resulting plague of layout shifts, a couple of years ago browsers decided to change the way that the `height` and `width` attributes on `` work. These days, consistently setting the `height` and `width` attributes to reflect the aspect ratio of the resource is a universally recommended best practice, which allows authors to tell the browser the intrinsic dimensions of an image resource before it loads.

215. <https://almanac.httparchive.org/en/2021/media#intrinsic-vs-extrinsic-sizing>

216. <https://web.dev/cls/>

28%

Figure 6.32. Percentage of `` elements on mobile that have both `height` and `width` attributes set.

Unfortunately, we have a long way to go before we get to universal adoption.

Delivery

Finally, let's take a look at how images are delivered over the network.

Cross-domain image hosts

How many images are being delivered from a different domain than the document they're embedded on? A majority of them, including 3.6 percentage points more than last year²¹⁷.

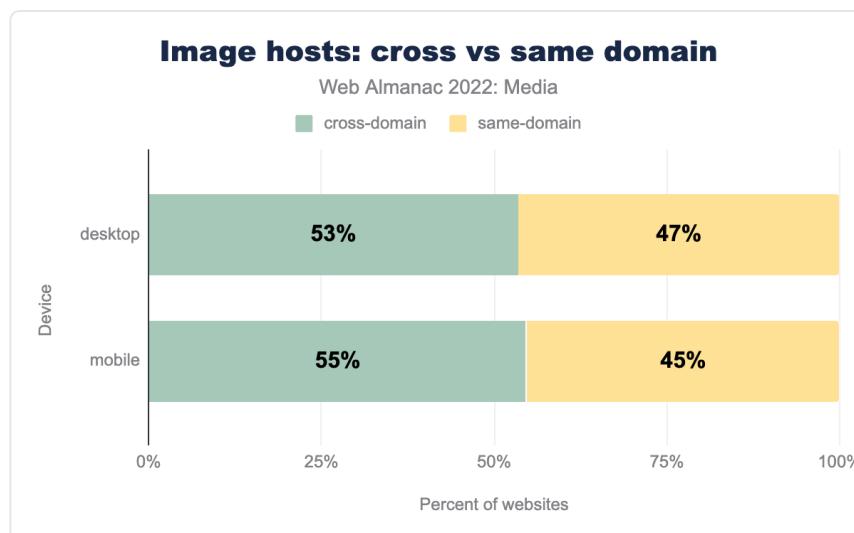


Figure 6.33. Image hosts: cross vs same domain.

The fact that a growing majority of images are being delivered across domains underscores

217. <https://almanac.httparchive.org/en/2021/media#cross-origin-image-hosts>

how hard images are to get right²¹⁸, and the benefits of enlisting an image CDN²¹⁹ to handle your media for you.

And now let's turn our attention to ``'s younger and more dynamic sibling: `<video>`.

Video

The `<video>` element shipped in 2010, and has been the best and—since the demise of plugins like Flash and Silverlight—the only way to embed video content on websites.

Over the last few years, I have had the sense that web content is shifting. Whereas still images (Flickr, Instagram) once ruled, I'm increasingly seeing moving ones (TikTok) dominate. Is this sense borne out in the Web Almanac's dataset? How are we using `<video>` on the web?

Video adoption

Usage of the `<video>` element continues to rise:

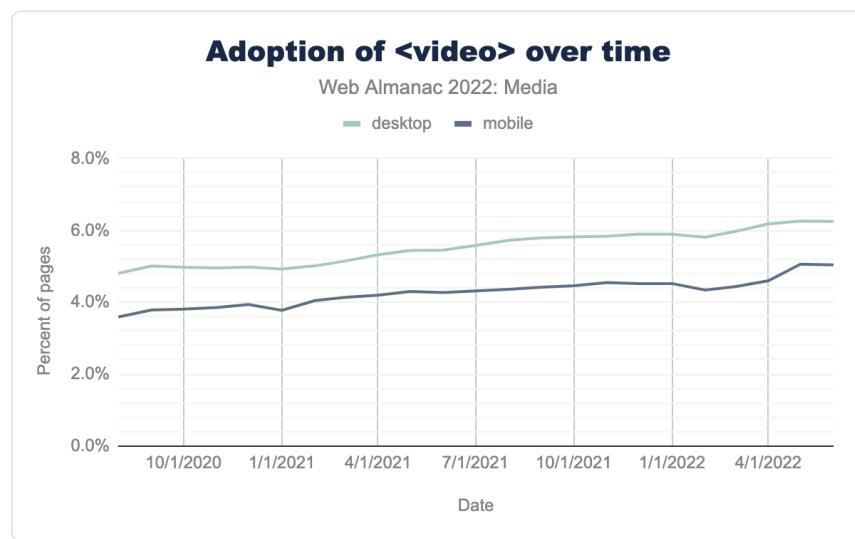


Figure 6.34. Adoption of `<video>` over time.

On mobile, `<video>` usage has risen from 4.3% of pages in June 2021 to 5% of pages in June

218. <https://css-tricks.com/images-are-hard/>

219. <https://web.dev/image-cdns/>

2022. One in 20 pages now include a `<video>` element, representing an increase of 18% year-over-year. I don't expect the web to contain as many `<video>`s as ``s anytime soon, but there are an increasing number of `<video>`s every year!

Video durations

How long are those videos? Not very!

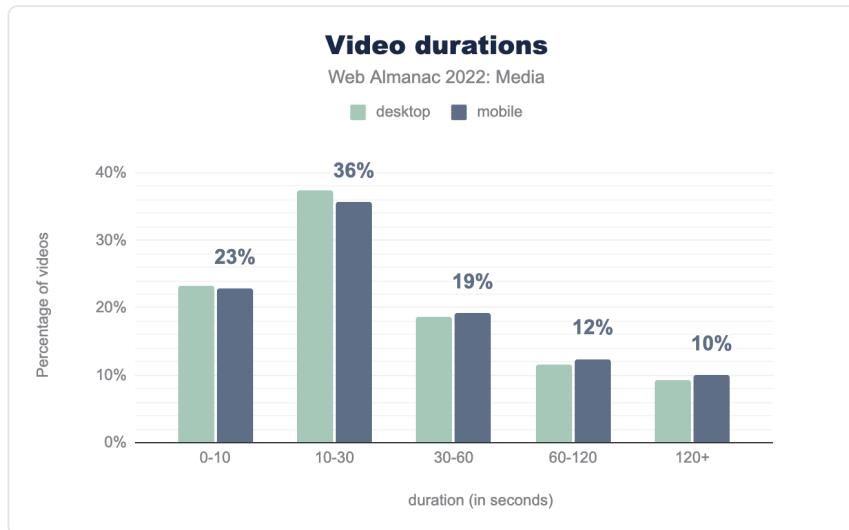


Figure 6.35. Video durations.

Nine out of ten videos are less than two minutes long. And more than half are under 30 seconds. Almost a quarter of videos are under 10 seconds. Perhaps these are GIFs in `<video>` clothing?

Format adoption

What formats are sites delivering in 2022? MP4, with its universal support story²²⁰, is king:

²²⁰. <https://caniuse.com/mpeg4>

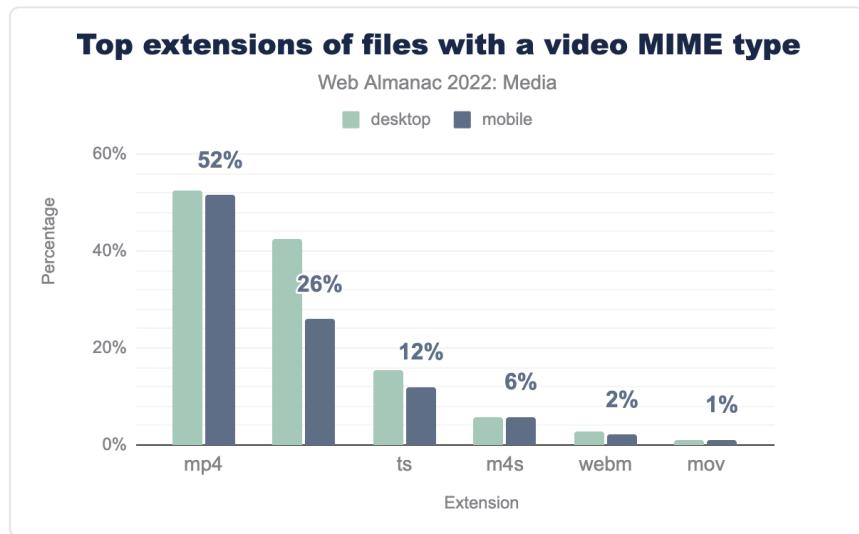


Figure 6.36. Top extensions of files with a video MIME type.

But MP4's numbers are a couple ticks down from last year²²¹, and we continue to see files with blank extensions, `.ts` files, and `.m4s` files gaining ground. These files are delivered when a `<video>` employs adaptive bitrate streaming using either HLS²²² or MPEG-DASH²²³.

It's encouraging to see responsive video delivery using adaptive streaming on the rise. At the same time, we look forward to the web platform offering a simple, declarative solution to adaptive videos²²⁴ that doesn't rely on JavaScript.

Embedding

The `<video>` element offers a number of attributes that allow authors to control how the video will be loaded and presented on the page. Here they are, ranked by usage:

221. <https://almanac.httparchive.org/en/2021/media#fig-29>

222. https://en.wikipedia.org/wiki/HTTP_Live_Streaming

223. https://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP

224. <https://github.com/whatwg/html/issues/6363>

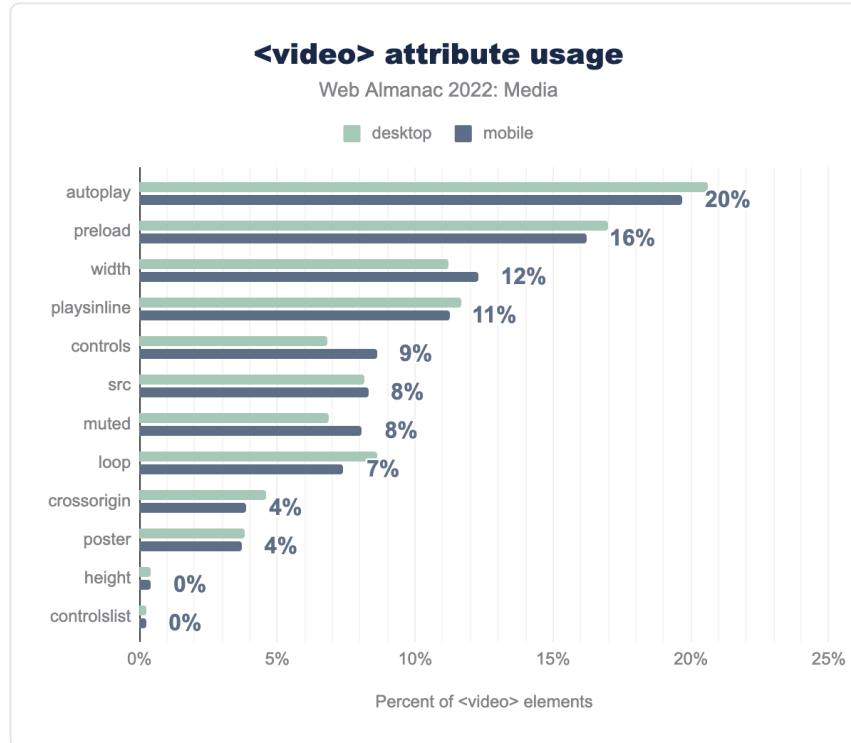


Figure 6.37. `<video>` attribute usage.

There are a number of things to unpack here.

First, `autoplay` overtook `preload` to become the most popular attribute this year. We also see `playsinline`, `muted`, and `loop` increasing in popularity. Perhaps an increasing number of people are using the `<video>` element to replace animated GIFs? If so, good!

The fact that only 12% of `<video>`s have `width` attributes and just 0.4% (!) have `height` attributes means that most `<video>` elements are susceptible to the same kinds of CLS²²⁵ problems we saw with `` elements which lack these attributes. Help the browser help you and add these attributes!

Additionally, the fact that fewer than one-in-ten `<video>` elements has a `controls` attribute suggests a significant number of people are using players that provide their own user interface for interacting with the video.

²²⁵. <https://web.dev/cls/>

Usage of `preload` deserves some more investigation.

`preload`

The `preload` attribute has seen declining usage over the past couple of years.

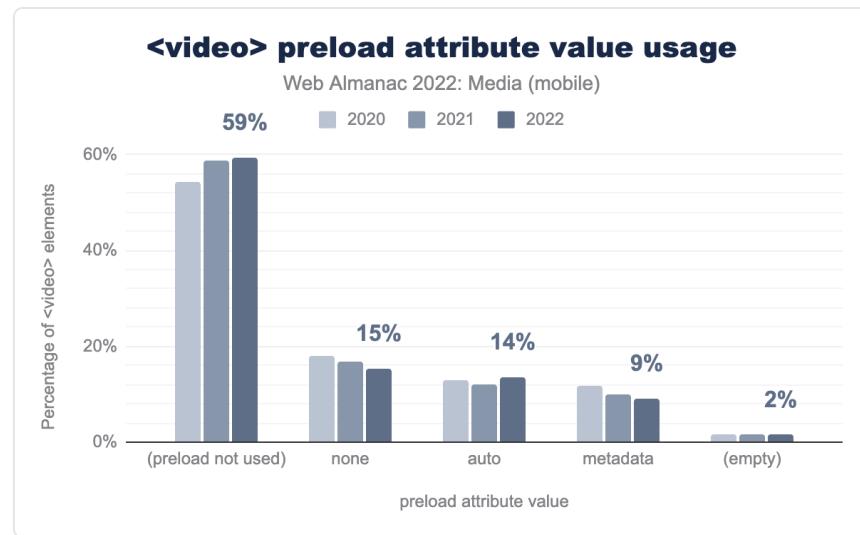


Figure 6.38. `<video preload>` attribute value usage.

Why? I like to think it is authors getting out of the browser's way.

Different browsers do different things when it comes to deciding when to load video data. The `preload` attribute is a way for authors to step in and have more control over that process. That could include explicitly asking the browser not to preload anything with `none`; asking the browser to preload just the `metadata`; or asking the browser to preload, using either the `auto` or empty values. It's interesting, and perhaps heartening, to see authors exert less control over video loading during the past three years. Browsers know the most about their users' contexts; not including the `preload` attribute at all lets them do what they think is best.

`src` and `<source>`

The `src` attribute is only present on 8-9% of `<video>` elements. Many of the rest use multiple `<source>` children, allowing authors to instead supply multiple alternate video resources in alternate formats.

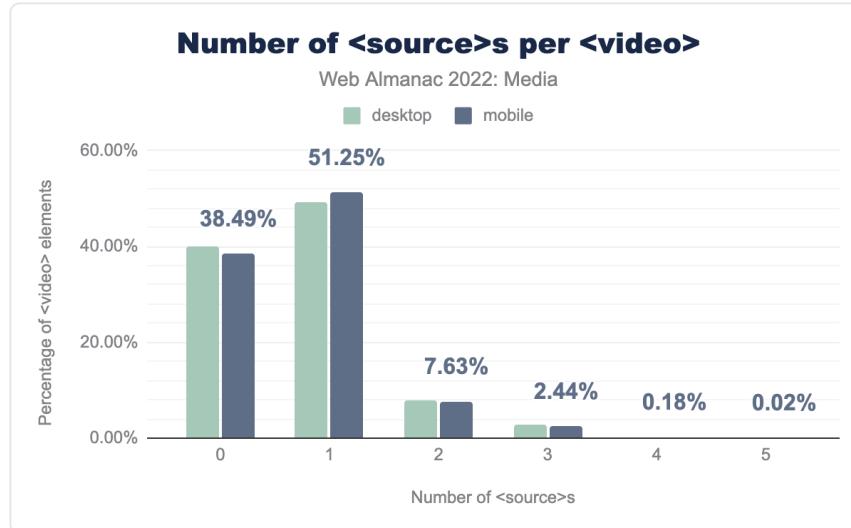


Figure 6.39. Number of <source> s per <video> .

How many <source> children do <video> elements have? Most have just one, and very few use multiple.

Conclusion

So there you have it, a snapshot of the state of media on the web in 2022. We've seen just how pervasive images and – increasingly – videos are on the web, and have gained some insight into how the web's images and videos are encoded and embedded. The most exciting developments this year are the accelerating adoption of AVIF and the ever-increasing adoption of both lazy-loading and adaptive bitrate streaming.

There were, however, some frustrating aspects, including the almost complete lack of wide-gamut color spaces; the undying zombie format that is GIF (in both its animated and, more surprisingly, non-animated forms); and the way that both the `sizes` attribute and lazy-loading – two features designed for performance – are (through improper use) hurting performance on a significant number of pages.

Here's to more effective visual communication on the web in 2023!

Authors



Eric Portis

👤 eeps 🌐 <https://ericportis.com>

Eric Portis is a Web Platform Advocate at Cloudinary²²⁶.



Akshay Ranganath

🐦 @rakshay 🌐 akshay-ranganath 🌐 <https://akshayranganath.github.io/>

Akshay Ranganath is a Sr. Solution Architect at Cloudinary²²⁷ and likes to work on CDN/WebPerf challenges.

226. <https://cloudinary.com/>
227. <https://cloudinary.com/>



Part I Chapter 7

WebAssembly



Written by Colin Eberhardt

Reviewed by Ben Smith and Ingvar Stepanyan

Analyzed by Jamie Macdonald

Edited by Barry Pollard

Introduction

WebAssembly—or Wasm—is a relative newcomer to the family of web technologies (JavaScript, HTML, CSS), becoming an officially recognized W3C standard in December 2019.

WebAssembly introduces a new runtime into the browser, one which works alongside, and in close collaboration, with the JavaScript runtime. It is relatively lightweight in comparison, with a small instruction set and a strict isolation model (WebAssembly has no I/O by default). One of the primary motivators for developing WebAssembly was to provide a compilation target for a wide range of programming languages (C++, Rust, Go etc.), allowing developers to write new web applications, or port existing applications, with a wider toolset.

High-profile examples of WebAssembly include its use within Google Earth²²⁸, where the C++ desktop application is now available within the browser, Figma²²⁹, a browser-based design tool

228. <https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html>

229. <https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/>

that enjoyed significant performance improvements using this technology, and most recently Photoshop²³⁰ which uses WebAssembly for similar reasons.

Methodology

WebAssembly is a compilation target, distributed as binary modules. For this reason, we face a number of challenges when analyzing its usage on the web. The 2021 Web Almanac, which is the first edition that included WebAssembly, includes a detailed section on the methodology used²³¹, and related caveats. The findings here, in the 2022 edition, followed the same methodology. The only enhancement added is a mechanism for determining the language used to author WebAssembly modules. The accuracy of that analysis is covered in more detail in the respective section.

How widely is WebAssembly being used?

We found 3,204 confirmed WebAssembly requests on desktop and 2,777 on mobile. Those modules are used across 2,524 domains on desktop and 2,216 domains on mobile, which represents 0.06% and 0.04% of all domains on desktop and mobile correspondingly.

This represents a modest drop in the number of modules we discovered in the crawl, a reduction of 16% for desktop and 12% mobile requests. This doesn't necessarily mean WebAssembly is in decline, when interpreting this change it is worth noting the following:

- While you can use WebAssembly to create all sorts of web-based content, its main benefit is found in more complex line-of-business applications with large codebases, that are often many years old (e.g. Google Earth, Photoshop, AutoCAD). These web 'apps' are not as numerous as the websites, and are not always available to the Almanac crawl, which is primarily based on home pages where WebAssembly may be less prevalent.
- As we shall see in a later section, much of the WebAssembly usage we see comes from a relatively small number of third-party libraries. As a result, a small change in any one of those libraries will have a significant impact on the number of modules we find.

We found slightly fewer (-13%) WebAssembly modules served to mobile browsers. This isn't a reflection on the WebAssembly capabilities of mobile browsers, which generally have excellent

230. <https://web.dev/ws-on-the-web/>

231. <https://almanac.httparchive.org/en/2021/webassembly#methodology>

support. Rather, it is likely due to the standard practice of progressive enhancement²³², where in these cases the more advanced features that require WebAssembly are not supported for mobile users.

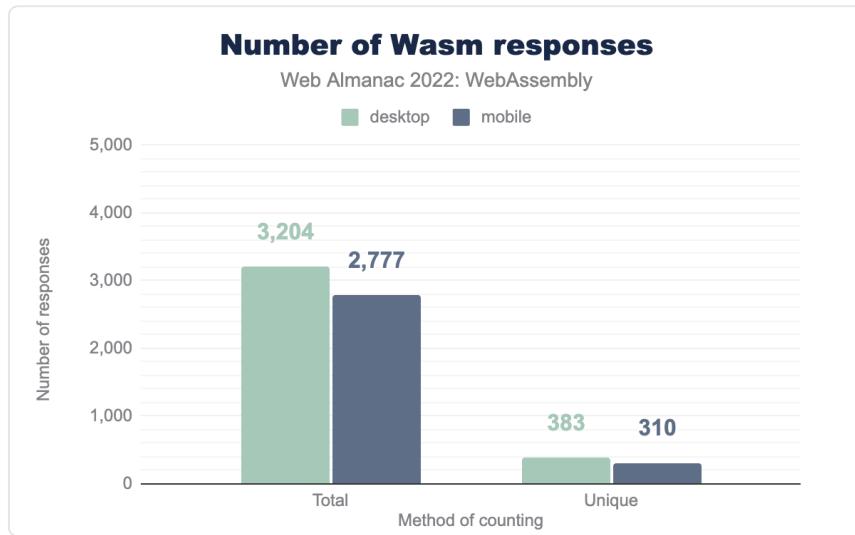


Figure 7.1. Number of Wasm responses.

By hashing the WebAssembly modules we can determine how many of these 3,204 modules—on desktop—are unique. By de-duplicating modules, the total number reduces by roughly a factor of 10, with 383 unique modules served to desktop browsers, and 310 to mobile. This indicates a significant amount of re-use—different websites making use of the same WebAssembly code, most likely through shared modules.

A significant proportion of wasm requests are cross-origin, further reinforcing the notion that they are re-used. Notably this has increased significantly from last year (67.2% vs 55.2%).

232. https://developer.mozilla.org/docs/Glossary/Progressive_Enhancement

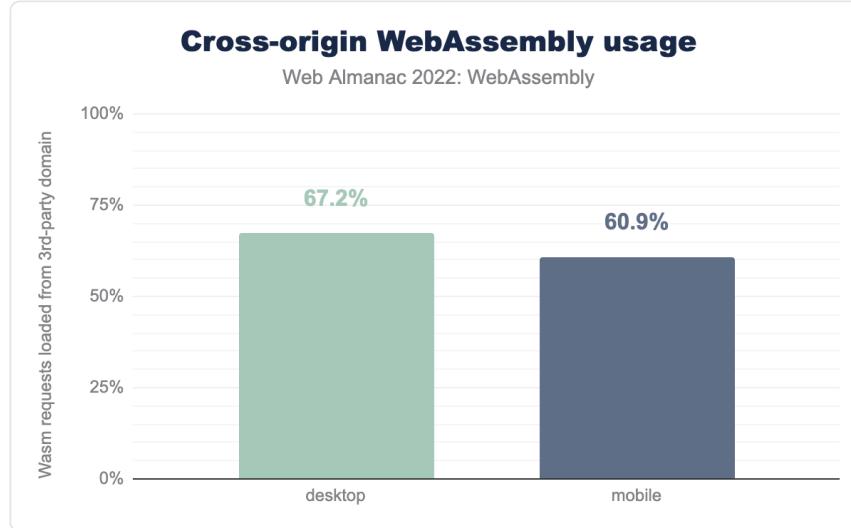


Figure 7.2. Cross-origin WebAssembly usage.

These WebAssembly modules differ considerably in size, with the smallest being just a few kilobytes, and the largest weighing in at 7.3 megabytes. Looking in more detail, at the uncompressed size, we see that the median (50th percentile) size is 835KBytes.

The smallest of WebAssembly modules are likely being used for quite specific functionality, for example polyfilling browser capabilities, or simple encryption tasks. The larger modules are likely entire applications that are compiled to WebAssembly.

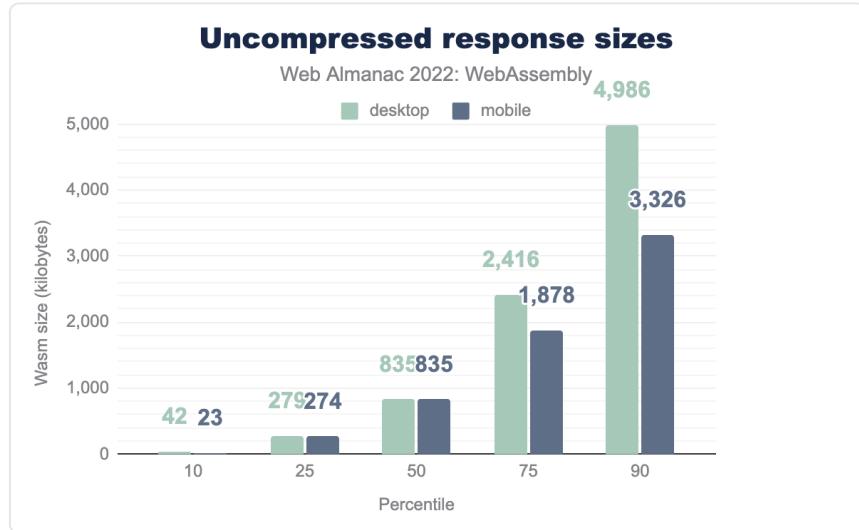


Figure 7.3. Uncompressed response sizes.

Clearly WebAssembly isn't widely used, and rather than seeing a growth in usage, we are seeing a modest contraction.

What is WebAssembly being used for?

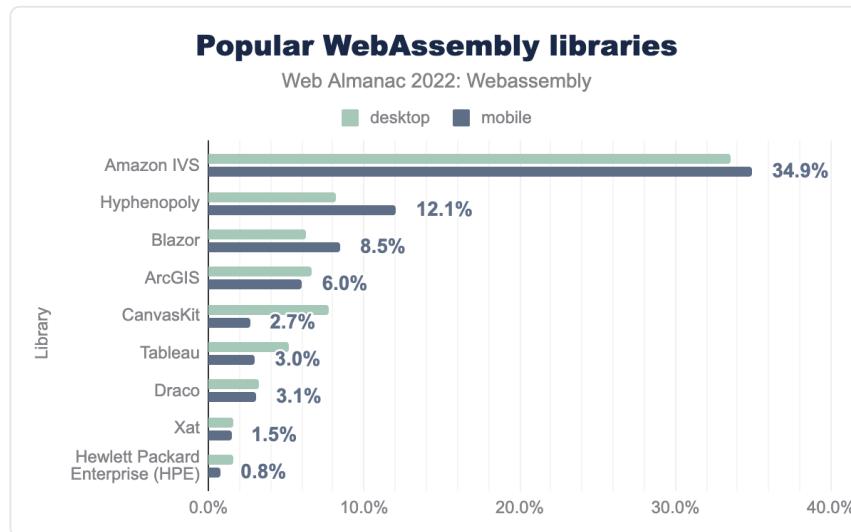


Figure 7.4. Popular WebAssembly libraries.

- Amazon IVS (Amazon Interactive Video Service)²³³ - Here WebAssembly is likely being used as a video codec, allowing consistent video decoding independent of the codec support of the user's browser
- Hyphenopoly²³⁴ - This is an npm module that provides a polyfill for CSS hyphenation. The core algorithm is shipped as a WebAssembly module, giving a small footprint and consistent performance
- Blazor²³⁵ - Microsoft Blazor is a platform-runtime and UI library—that supports the development of web applications using the .NET platform and C#.
- ArcGIS²³⁶ - A comprehensive suite of tools for building interactive mapping applications. Performance is a primary concern for the ArcGIS team, and they employ various technologies such as WebGL to achieve this. Specifically, WebAssembly is used to enable fast client-side projections.
- CanvasKit²³⁷ - This library provides more advanced capabilities than the standard Canvas2D API. It is implemented via Skia, a graphics library written in C++, which is

233. <https://aws.amazon.com/ivs/>

234. <https://mnoter.github.io/Hyphenopoly/>

235. <https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor>

236. <https://developers.arcgis.com/javascript/latest/>

237. <https://skia.org/docs/user/modules/canvaskit/>

compiled to WebAssembly allowing execution in the browser.

- Tableau²³⁸ - A popular tool for building interactive visualizations. It is not clear whether WebAssembly is used as part of their core product, or whether it is just being used for the specific dashboards that were found as part of the crawl.
- Draco²³⁹ - A library for compressing and decompressing 3D geometric meshes and point clouds. It is written in C++, with the WebAssembly building allowing its use within the browser.
- Xat²⁴⁰ - A social media site. It is unclear what they are using WebAssembly for.
- Hewlett Packard Enterprise²⁴¹ - It is unclear what they are using WebAssembly for.

From looking at the popular WebAssembly libraries we can see that its usage is quite targeted, often being used for specific number-crunching tasks, or leveraging large and mature C++ codebases, bringing their capabilities to the web without the need to port to JavaScript.

What languages are people using?

WebAssembly is a binary format, and as a result, much of the information in the source—programming language, application structure, variable names—is obfuscated or entirely lost in the compilation process.

However, modules often have exports and imports, which name functions within the hosting environment—the JavaScript runtime within the browser—that describe the module interface. Most WebAssembly toolchains create a small amount of JavaScript code, for the purposes of ‘binding’, making it easier to integrate modules into JavaScript applications. These bindings often have recognizable function names which are present in the modules exports or imports, giving a reliable mechanism for identifying the language that was used to author the module.

We enhanced the `wasm-stats`²⁴² project, which provides WebAssembly-specific analysis to the crawler, adding code which inspects exports / imports to identify common patterns that provide an indication of the language used to author a given module. As an example, if a module imports a module named `wbindgen`, this is a reference to code generated by `wasm-bindgen`²⁴³ and a clear indicator that the module was written in Rust.

In some cases, the export / import names are minified, making it harder to identify the source

238. <https://www.tableau.com/>
 239. <https://github.io/draco/>
 240. <https://xat.com/>
 241. <https://www.hpe.com/us/en/home.html>
 242. <https://github.com/HTTPArchive/wasm-stats>
 243. <https://crates.io/crates/wasm-bindgen>

language. However, Emscripten (a C++ toolchain), has a distinctive convention for minified names, meaning that we can be relatively confident that modules exhibiting this pattern were generated using Emscripten.

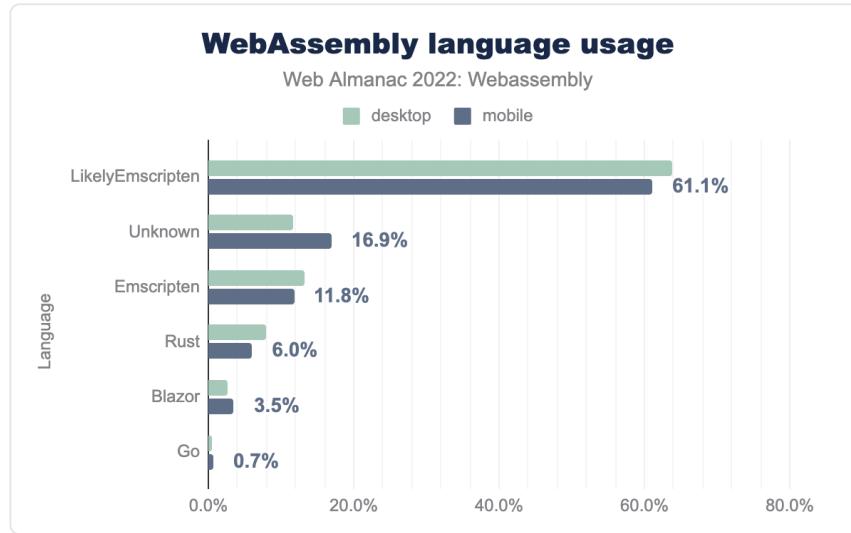


Figure 7.5. WebAssembly language usage.

Looking at the results, we found that, on desktop, 72.8% of modules were very likely created using Emscripten, and as a result are most likely written in C++. Next most popular is Rust at 6.0%, then Blazor (C#) at 3.5%. We also found a small number of modules written in Go.

Notably, 16.9% of modules didn't have an identifiable language. AssemblyScript²⁴⁴ is a popular WebAssembly-specific language which doesn't provide any obvious clues in the modules it produces. We know that Hypehnopoly—which represents 8.2% of all modules—uses AssemblyScript, and it accounts for almost half of these 'unidentified' modules.

It is interesting to contrast these results with the State of WebAssembly 2022 survey²⁴⁵, where Rust was the most frequently used language. However, a significant number of respondents to that survey were using WebAssembly for non-browser based applications.

What features are being used?

The initial release of WebAssembly was considered an MVP. In common with other web

244. <https://www.assemblyscript.org/>
 245. <https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html>

standards, it is continually evolving under the governance of the World Wide Web Consortium (W3C). This year saw the announcement of the WebAssembly v2 draft²⁴⁶, adding a number of new features.

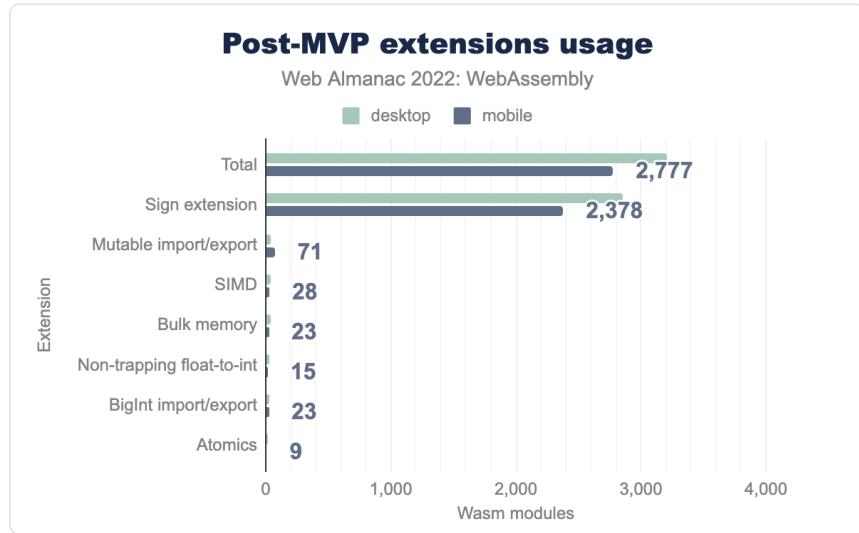


Figure 7.6. Post-MVP extensions usage.

We looked at the Post-MVP features that are being used, finding that *sign extension* (a relatively simple enhancement adding operators that allow you to extend integer values to a greater bit-depth), was by far the most frequently used. This doesn't represent a significant difference to the result found in last year's analysis²⁴⁷.

Notably, while web developers are faced with the choice of which HTML / JavaScript / CSS features to use, with WebAssembly this is often a decision made by the toolchain developers. As a result, we will likely see Post-MVP feature adoption jump when a given toolchain determines that it is now a viable option.

Conclusions

WebAssembly is undeniably a niche technology when it comes to the web, and there is a very good chance that it always will be. While WebAssembly has brought a wide range of languages to the web—C++, Rust, Go, AssemblyScript, C# and more—these cannot yet be used interchangeably with JavaScript. For the vast majority of websites, where the content is

246. <https://www.w3.org/TR/wasm-core-2/>

247. <https://almanac.httparchive.org/en/2021/webassembly#whats-the-usage-of-post-mvp-extensions>

relatively static (rendered in HTML with CSS) with a modest amount of interactivity (via JavaScript) there simply isn't a compelling reason to use WebAssembly at the moment.

There are some significant proposals which could change this in the future. Initially WebIDL, which was superseded by Interface Types, which has once again been superseded by the Component Model specification. These may result in a future where it is possible to easily interchange JavaScript for any other programming language, but for now, this simply isn't the case.

Despite being a niche technology, WebAssembly is already adding value to the web. There are a number of web applications that benefit greatly from this technology. However, web applications are often not visible to the 'crawl' which forms the basis of this study.

Finally, the core features of the WebAssembly runtime—multi-language, lightweight, secure—are making it a popular choice for a wider range of non-browser applications. The State of WebAssembly 2022 survey²⁴⁸ saw a significant increase in the number of people using this technology for serverless, containerization and plug-in applications. The future of WebAssembly could be as a niche web technology, but as an entirely mainstream runtime on a wide range of other platforms.

Author



Colin Eberhardt

@ColinEberhardt ColinEberhardt <https://blog.scottlogic.com/ceberhardt/>

Colin is CTO at Scott Logic²⁴⁹ and is a prolific technical author, blogger and speaker on a range of technologies. He is a board member of FINOS²⁵⁰, which is encouraging open source collaboration in the financial sector. He is also very active on GitHub, contributing to a number of different projects.

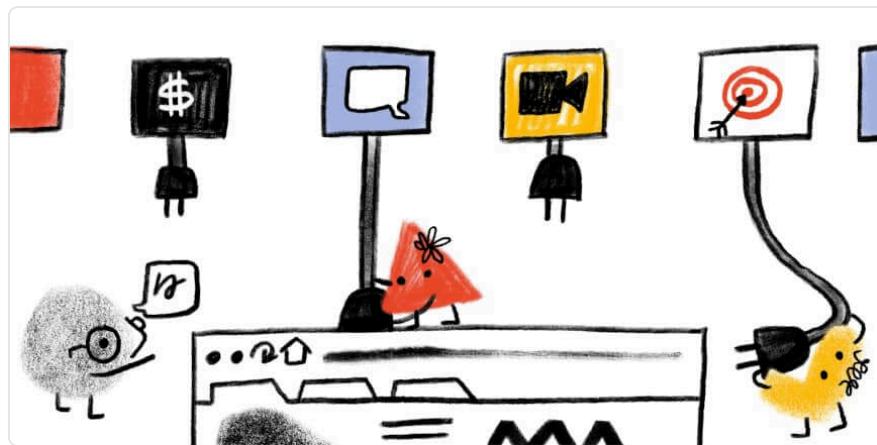
248. <https://blog.scottlogic.com/2022/06/20/state-of-wasm-2022.html>

249. <https://www.scottlogic.com/>

250. <https://www.fin-os.org/>

Part I Chapter 8

Third Parties



Written by Eugenia Zsigisova

Reviewed by Barry Pollard, Kevin Farrugia, and Alex N. Jose

Analyzed by Kevin Farrugia

Edited by Shaina Hantsis

Introduction

Third parties are an integral part of most websites. This chapter shows that nearly all websites use at least one third party, and nearly half of all requests are third-party requests.

Website owners use third parties to delegate some complex functionality such as analytics, advertising, live chats, consent management, and others. Although website developers may not directly control third-party code, they still can influence third parties' impact on the websites. Taking into account how widely third parties are used, they have a crucial impact on web performance. It is quite common that they block page rendering, especially on mobile devices. For instance, the average median blocking time for the top 10 most popular third parties is 1.4 seconds. Because of this, third parties can have a direct effect on Core Web Vitals²⁵¹ and other important performance metrics like First Contentful Paint²⁵².

251. <https://web.dev/vitals/>

252. <https://web.dev/fcp>

Many recommendations can help to eliminate the negative impact. It could be simple techniques like minifying resources or more complex ones, for example, evaluating and choosing third-party scripts that don't serve legacy JavaScript or loading and executing third-party scripts using web workers.

This chapter focuses on the topic of how website owners and third-party developers can reduce negative third-party impact and if the best practices are followed. We start with a review of third-party prevalence and some web performance-related metrics: render-blocking time and impact on the main thread. The second half is an analysis of the best practices regarding minifying and compressing resources, third-party facades, `async` and `defer` script attributes, legacy JavaScript, and other optimization opportunities.

Definitions

A third party is an entity outside the primary site-user relationship. It involves the aspects of the site not directly within the control of the site owner but with their approval. Third-party resources are:

- Hosted on a shared and public origin
- Widely used by a variety of sites
- Uninfluenced by an individual site owner

Some examples of third parties include Google Fonts, the jQuery library served over public origin, and embedded YouTube videos.

To match the definition, only third parties originating from a domain whose resources can be found on at least 50 unique pages in the HTTP Archive dataset were included.

In the case where third-party content is served from a first-party domain, it is counted as first-party content. For example, self-hosting Google Fonts or bootstrap.css is counted as *first-party content*. Similarly, first-party content served from a third-party domain is counted as third-party content—assuming it passes the “more than 50 pages criteria”.

Third-party categories

We are relying on the third-party-web²⁵³ repository from Patrick Hulce²⁵⁴ to help us identify and categorize third parties. This repository breaks down third parties by the following categories:

253. <https://github.com/patrickhulce/third-party-web/#third-parties-by-category>

254. <https://twitter.com/patrickhulce>

- **Ad** - These scripts are part of advertising networks, either serving or measuring.
- **Analytics** -These scripts measure or track users and their actions. There's a wide range in impact here depending on what's being tracked.
- **CDN** - These are a mixture of publicly hosted open source libraries (e.g. jQuery) served over different public CDNs and private CDN usage.
- **Content** - These scripts are from content providers or publishing-specific affiliate tracking.
- **Customer Success** - These scripts are from customer support/marketing providers that offer chat and contact solutions. These scripts are generally heavier in weight.
- **Hosting*** - These scripts are from web hosting platforms (WordPress, Wix, Squarespace, etc.).
- **Marketing** - These scripts are from marketing tools that add popups/newsletters/etc.
- **Social** - These scripts enable social features.
- **Tag Manager** - These scripts tend to load lots of other scripts and initiate many tasks.
- **Utility** - These scripts are developer utilities (API clients, site monitoring, fraud detection, etc.).
- **Video** - These scripts enable video player and streaming functionality.
- **Consent provider** - These scripts allow sites to manage the user consent (eg. for the General Data Protection Regulation²⁵⁵ compliance). They are also known as the 'Cookie Consent' popups and are usually loaded on the critical path.
- **Other** - These are miscellaneous scripts delivered via a shared origin with no precise category or attribution.

Note: The CDN category here includes providers that provide resources on public CDN domains (e.g. bootstrapcdn.com, cdnjs.cloudflare.com, etc.) and does not include resources that are simply served over a CDN. For example, putting Cloudflare in front of a page would not influence its first-party designation according to our criteria.

*The same as in the previous year, the Hosting category is removed from our analysis. For example, if you happen to use WordPress.com for your blog, or Shopify for your e-commerce

255. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation

platform, then we're going to ignore other requests for those domains by that site as not truly "third-party" as they are, in many ways, part of hosting on those platforms.

Caveats

- All data presented here is based on a non-interactive, cold load. These values could start to look quite different after user interaction.
- The pages are tested from servers in the U.S. with no cookies set, so third parties requested after opt-in are not included. This will especially affect pages hosted and predominantly served to countries in scope of the General Data Protection Regulation²⁵⁶, or other similar legislation.
- Only the home pages are tested. Other pages may have different third-party requirements.
- Some of the lesser-used third-party domains are grouped into the *unknown* category.
- We are leveraging different Lighthouse audits²⁵⁷. Some of them have limited coverage. Namely, it is not feasible to cover all existing facade implementations in the third-party facades audit²⁵⁸.

Learn more about our Methodology.

Prevalence



94%

Figure 8.1. Percent of mobile pages that use at least one third-party

The prevalence of third parties remained at the same high levels as the previous year: 94% of websites use at least one third party.

256. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation
 257. <https://github.com/GoogleChrome/lighthouse/tree/master/core/audits>
 258. <https://github.com/GoogleChrome/lighthouse/blob/master/core/audits/third-party-facades.js>

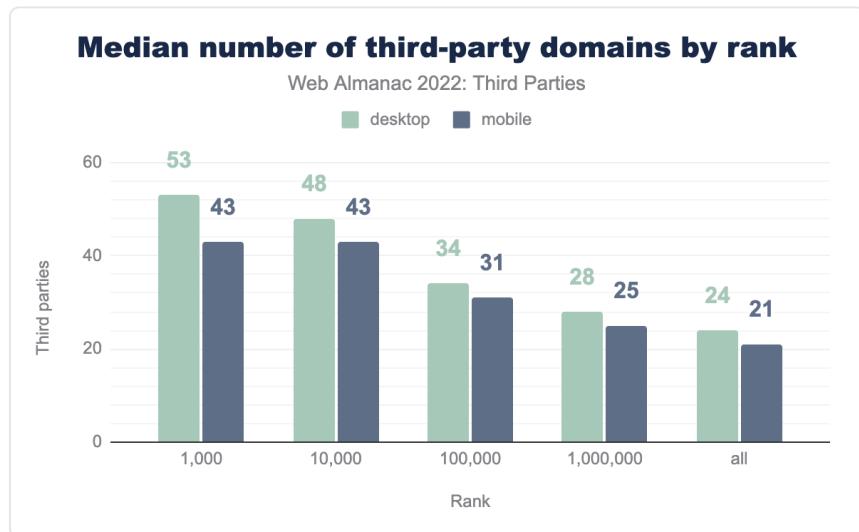


Figure 8.2. Median number of third-party domains per page by rank.

The figure above shows the number of third-party domains for the most-used websites. For example, on average the top 1,000 popular websites use 43 third-party domains on mobile and 53 on desktop devices. More popular websites seem to have a larger number of third-party domains, i.e. the top 1,000 sites have twice more third parties than the median number of third parties for all websites. This large number is explained by the fact that some third-party providers might have content hosted on multiple domains, for example, Yahoo serves their content from `mempf.yahoo.co.jp`, `yjtag.yahoo.co.jp`, etc. Though the exact number of third-party providers is still a subject for further research, the current data about third-party domains gives an overview of how much they might affect time spent on network requests. As every request to a new domain takes time for DNS lookup and establishing an initial connection, the more third party domains are used the more it might affect page loading speed.

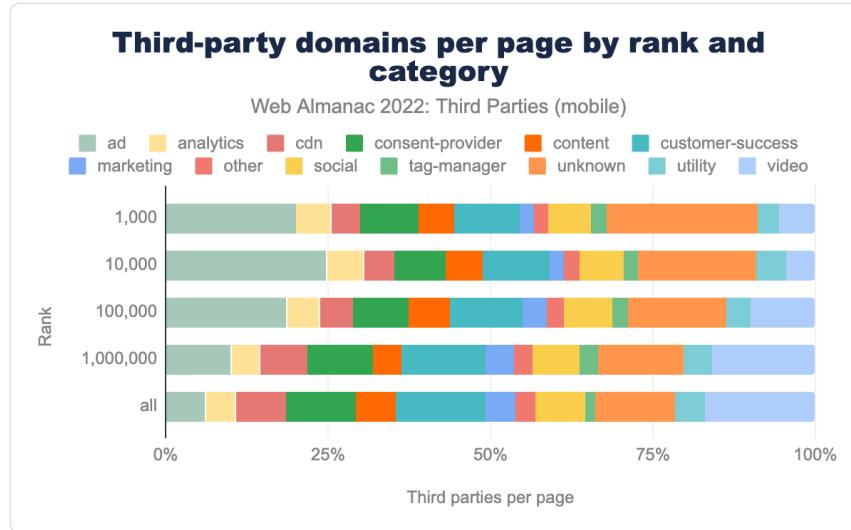


Figure 8.3. Median number of third-party domains per page by category and rank.

When looking at the distribution of third parties by category and by rank, it becomes clear that the increase in the number of third parties on more popular websites is mostly made up of the ad and unknown (i.e. unclassified) third-party categories. That means the third parties—especially ads—have a crucial impact on the users because they are more used on websites with a larger number of users.

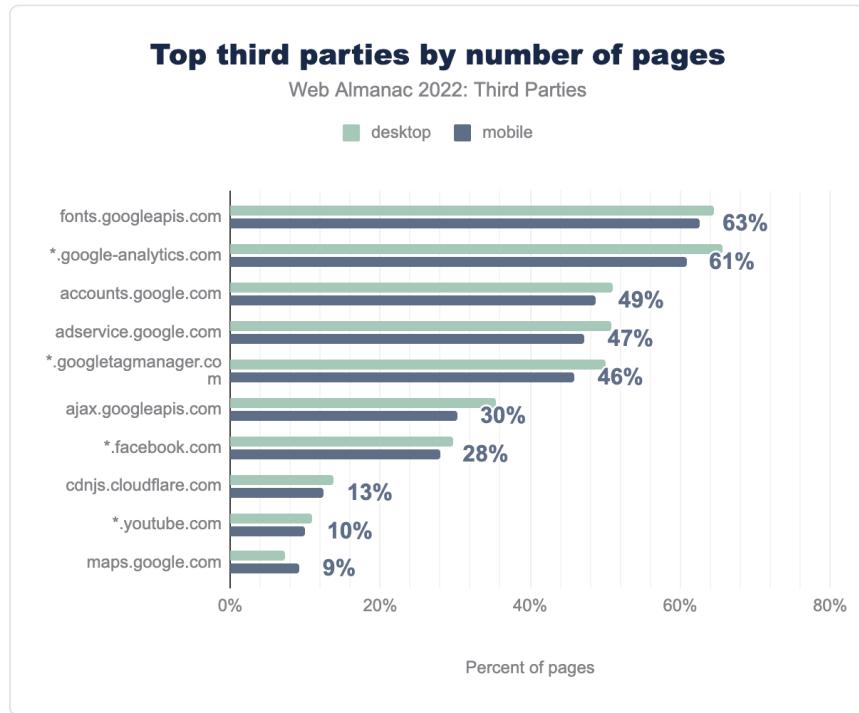


Figure 8.4. Top 10 third parties by number of pages they are used on

Google services: fonts, analytics, account management, advertising, and tag management, are the most popular third parties across the entire web. 63% of all pages use Google Fonts which is over 4.9 million pages out of the 7.9 million mobile pages in our dataset!

34%

Figure 8.5. Percentage of scripts of all third-party requests

Third-party requests account for 45% of all requests made by websites, and of those third-party requests, approximately one-third are scripts. This suggests it is worthwhile analyzing scripts in more detail in the best practices section.

Performance impact

Some third parties might inevitably block page rendering and negatively affect the web page loading experience. Lighthouse has a render-blocking resources audit²⁵⁹, that provides data about render-blocking time.

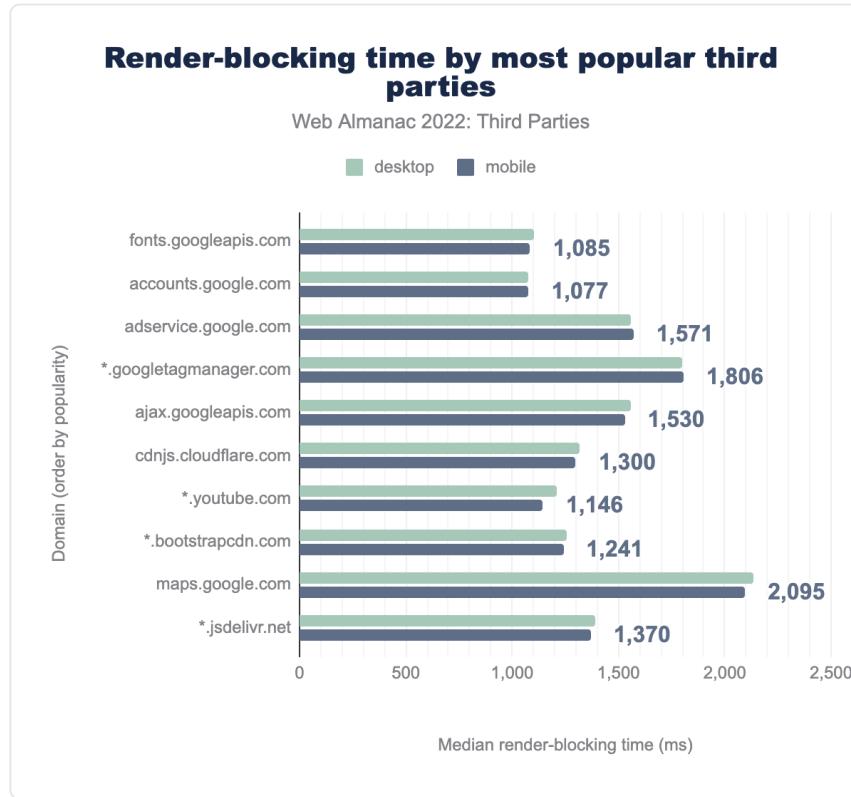


Figure 8.6. Median render-blocking time for top 10 most popular third parties.

The figure above shows the median render-blocking time for the top 10 most popular third parties. Google maps have the most significant impact on rendering time. It accounts for more than 2 seconds for the median website. That is a significant impact taking into account that the recommended time for First Contentful Paint²⁶⁰—a page load speed metric—is 1.8 seconds.

A website can save a lot of loading time by eliminating render-blocking resources. There are

259. <https://github.com/GoogleChrome/lighthouse/blob/master/core/audits/byte-efficiency/render-blocking-resources.js>

260. <https://web.dev/fcp>

many methods to embed third parties in a non-blocking way²⁶¹. For example, in the case of Google Maps, the Maps Static API²⁶² could be used to implement a third-party facade. Also, the third-party iframes can be lazy-loaded.

Additionally, third-party scripts compete for the main thread resources with the website's first-party code. If a third-party has a long-running JavaScript task that runs on the main thread for more than 50 ms, it is considered to be "blocking the main thread". It can significantly influence user experience when interacting with a page as the main thread is responsible for processing user events, as well as rendering the page. When it is blocked, a user suffers from a non-responsive page.

We inspected the third party summary audit²⁶³ to emulate main-thread blocking time caused by third parties.

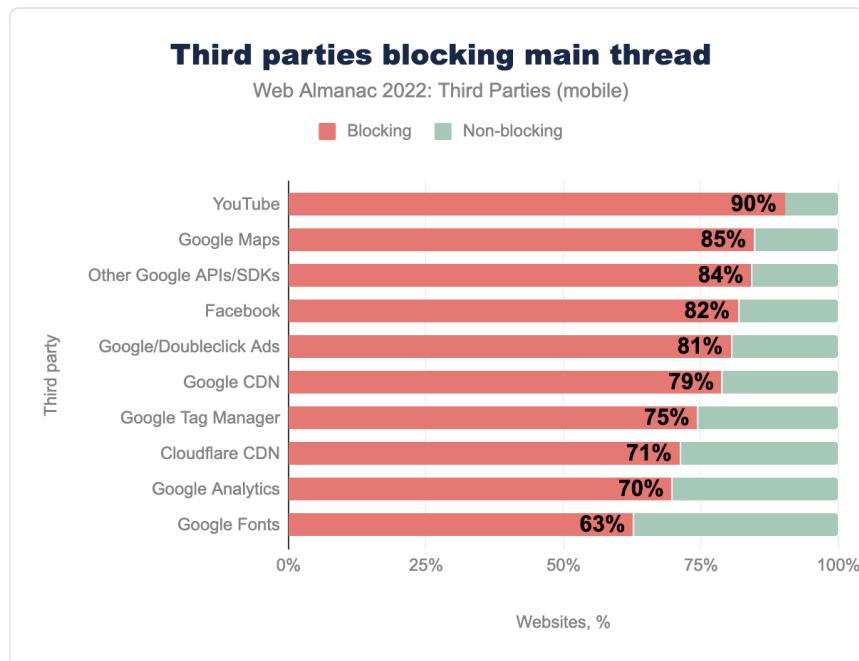


Figure 8.7. Top 10 third parties blocking the main thread

The figure above presents the top 10 most-used third parties and their impact on the main thread on mobile devices. Note that for desktop devices the impact is much lower. For example, YouTube blocks the main thread for 90% of the mobile websites while blocking only 26% of

261. <https://web.dev/embed-best-practices/>

262. <https://developers.google.com/maps/documentation/maps-static/overview>

263. <https://github.com/GoogleChrome/lighthouse/blob/master/core/audits/third-party-summary.js>

desktop websites it is embedded on. This is logical taking into account that desktop devices are much more powerful. However, nowadays mobile user experience is very important and—according to the Mobile Web chapter—58% of website visits are coming from mobile devices.

To examine in more detail how the website users could be affected by the main thread blocking third-party, we can review the median main thread blocking time.

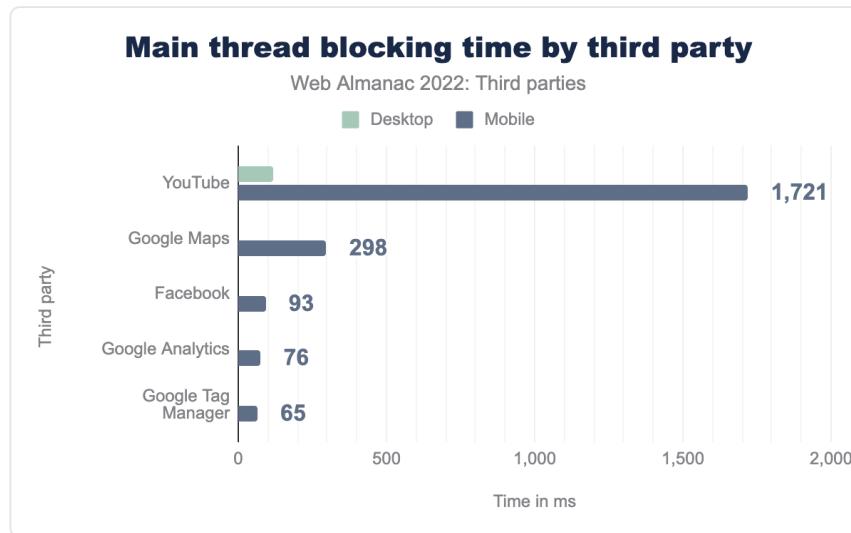


Figure 8.8. Median main thread blocking time in ms for top 5 most popular third parties.

YouTube is the most blocking third party among the top 5 most-used third parties. It blocks the main thread for more than 1.7 seconds for the median website that loads YouTube videos—based on the mobile device we emulate as part of our crawl. There is a notable gap between desktop and mobile websites as the desktop websites are almost completely unaffected.

Note that Lighthouse might mark some third parties as potentially render-blocking while their size is so small that they don't have any tangible effect on the render-blocking time. This is the case for third parties like Google Fonts or Google/Doubleclick Ads whose median render-blocking time is 0 milliseconds.

A blocked main thread has a big impact on the First Input Delay (FID)²⁶⁴ and Interaction to Next Paint (INP)²⁶⁵ performance metrics. To provide good web page responsiveness, FID should be

264. <https://web.dev/fid/>

265. <https://web.dev/inp/>

100ms or less and INP below 200ms. There is research by Annie Sullivan²⁶⁶ about the correlation between Total Blocking Time and Interaction to Next Paint on mobile devices²⁶⁷. It shows that the smaller the main thread blocking time, the more likely the sites meet good INP and FID thresholds. That leads to the conclusion that it becomes harder to achieve good core web performance metrics if third parties are blocking the main thread for such a long time, as in the YouTube example. Moreover, other third-party and first-party assets might also contribute to the render-blocking effect. Despite this, there are many ways to minimize the render-blocking effect of third parties. This will be further explored in the next section.

Web performance best practices

The previous section confirmed that the third parties are potentially causing a huge performance impact that can affect the entire website experience. However, website and third-party developers can follow many best practices to minimize third-party performance impact from minifying resources to using third-party facades. We looked at different third party usage trends to understand how the best practices are followed.

Minifying resources

Minifying JavaScript and CSS files is one of the first recommendations when speaking about web performance. To check how third-party resources are minified we are making use of the following Lighthouse audits: Unminified JavaScript²⁶⁸ and Unminified CSS²⁶⁹.

Minifying scripts should have a large positive impact as they are the most popular third-party content type. Moreover, compared to other content types like CSS, scripts tend to be a lot more verbose, with comments and large variable names that affect the file size.

266. [@anniesullivan](https://twitter.com/anniesullivan)

267. https://github.com/GoogleChromeLabs/chrome-http-archive-analysis/blob/main/notebooks/HTTP_Archive_TBT_and_INP.ipynb

268. <https://web.dev/unminified-javascript/>

269. <https://web.dev/unminified-css/>

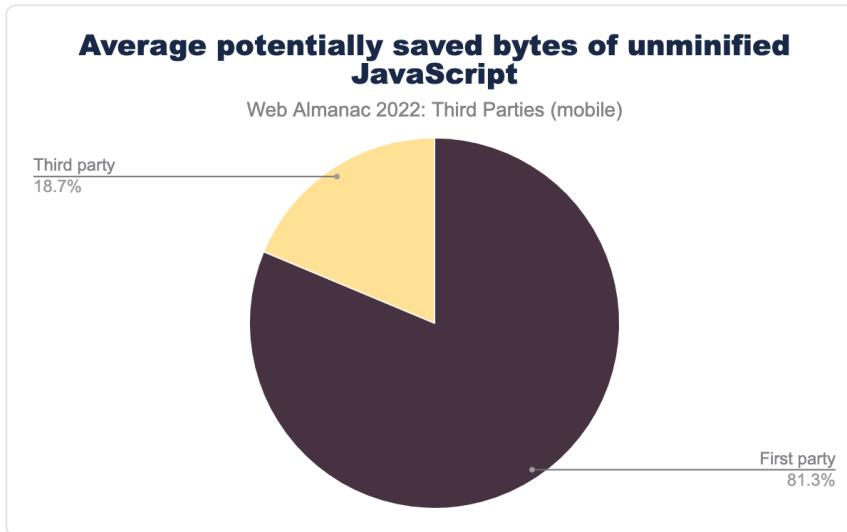


Figure 8.9. Percentage of the average potentially saved bytes of unminified JavaScript by first and third party.

Although some JavaScript bytes could be saved by minifying third-party resources, the first-party scripts are still responsible for the largest amount of unminified JavaScript on the websites, i.e. 81% of the average total potentially saved bytes. The distribution of unminified JavaScript shows that for 75% of websites total potential savings could achieve 34.5 KB of savings while savings from third parties are only 2.8 KB.

The next figure displays the size of potential savings by a third-party. It is important to note that the methodology used only includes third parties that come from external domains and does not count third-party content hosted by a first party, for example libraries imported as node modules.

Potential savings of unminified JavaScript by third party

Web Almanac 2022: Third parties

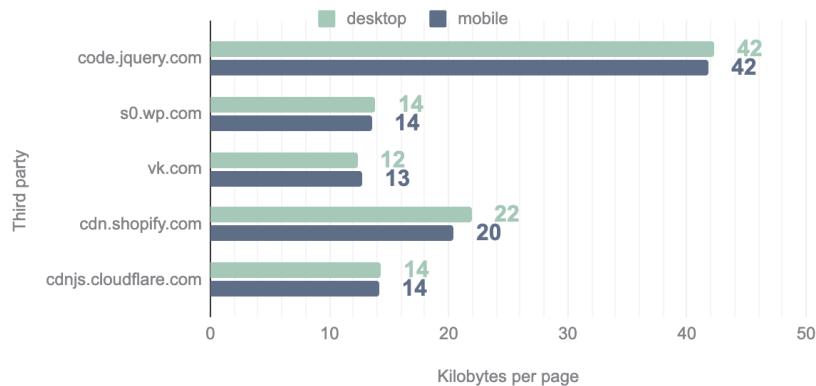


Figure 8.10. Average potentially saved kilobytes of unminified JavaScript for top 5 third-party script providers.

The `code.jquery.com` CDN library for jQuery is the most popular JavaScript third-party library being used on 6% of all websites on desktop (note that jQuery is used on far more websites, but not all uses are served from this CDN). On average 43 KB of data per page that has unminified jQuery could be saved by using the minified version of its resources, which are available on this CDN.

17%

Figure 8.11. Percent of desktop pages with unminified jQuery from all pages using jQuery third-party

17% of the websites that use jQuery hosted on a third-party domain fail the Lighthouse audit for unminified JavaScript²⁷⁰. Digging deeper into how the library is imported shows that many websites are using the unminified versions of jQuery that should only be used for development purposes. A similar tendency can be found in the usage of some other less popular third-party scripts.

This should serve as a reminder for web developers to check if the third-party scripts imported on their websites are optimized for production environments.

270. <https://web.dev/unminified-javascript/>

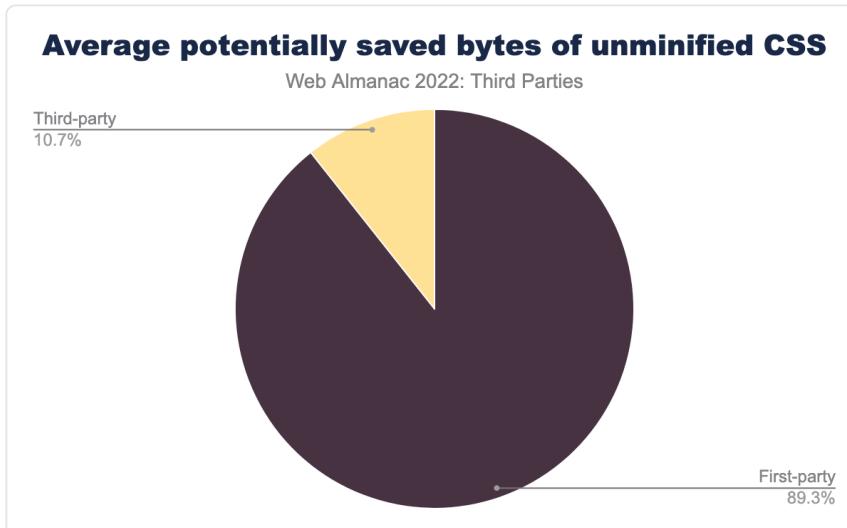


Figure 8.12. Percentage of average potentially saved bytes of unminified CSS by first and third party

The unminified CSS audit shows that third-party assets have a much smaller impact, especially when compared with the first-party average of potentially saved bytes of CSS code, which is 89% of the total average potentially saved bytes. This data demonstrates that third-party CSS content is well minified and has a much lower impact than first-party CSS.

Google fonts are the most-popular third party on mobile devices being used by 62.6% of all websites. The CSS they provide is not minified. The data shows the average page which has Google Fonts could save 13.3 KB from minifying it. This would seem like an opportunity for improvement. However, their CSS contains a number of very similar `font-face` definitions that are almost identically repeated in the case of many fonts, so HTTP-level compression will work really well here and significantly reduce the file size, even without minification. This makes the benefits of minifying very low, compared to the code readability for those wanting to see the CSS to potentially replicate locally. Other third parties with more complicated and larger CSS, may benefit from minification considerably more.

Compressing resources

Compressing files is another quick win that third-party developers can do to have a positive impact on web performance. Most heavy content types like scripts and CSS have a good compression coverage. Only 12% of scripts and 4.5% of CSS files of total third-party requests are not compressed.

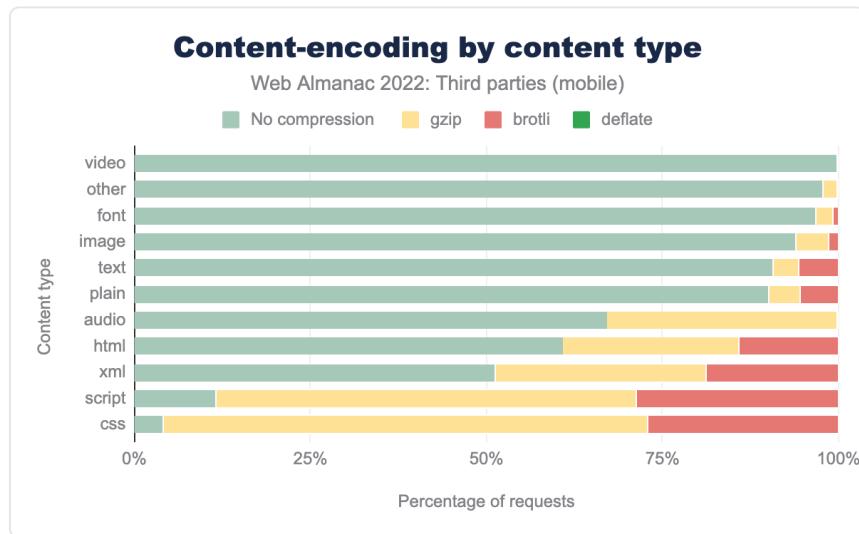


Figure 8.13. Content-encoding by third-party content type on mobile pages.

Website content-encoding data displayed in the above figure revealed an interesting fact about image compression. Even though image formats like JPEG, PNG, WebP, AVIF, and others provide compression under the hood as part of their formats, 5.2% of image content is compressed again using Gzip or Brotli compression. Adding additional layers of compression on top of the standard image compression formats is usually unnecessary and may lead to increased file size and add extra load on the CPU when uncompressing the image.

Gzip is still the most popular compression type, i.e. 59% of scripts and 68% of CSS are compressed with Gzip. However, Brotli compression is more effective. The trends among first and third parties show that usage of Brotli compression has grown in the past 3 years, while no compression and Gzip have fallen.

First-party content-encoding by year

Web Almanac 2022: Third Parties (mobile)

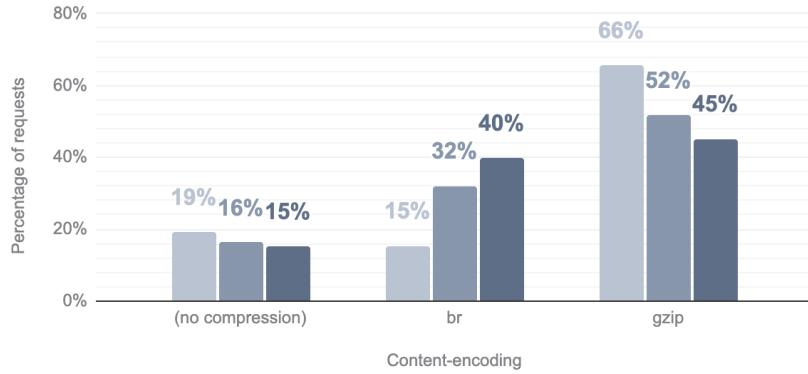


Figure 8.14. Percentage of first-party script requests by content-encoding type and by year on mobile websites.

The rate of first-party scripts compressed via brotli almost tripled, increasing from 15% to 40%.

Third-party content-encoding by year

Web Almanac 2022: Third Parties (mobile)

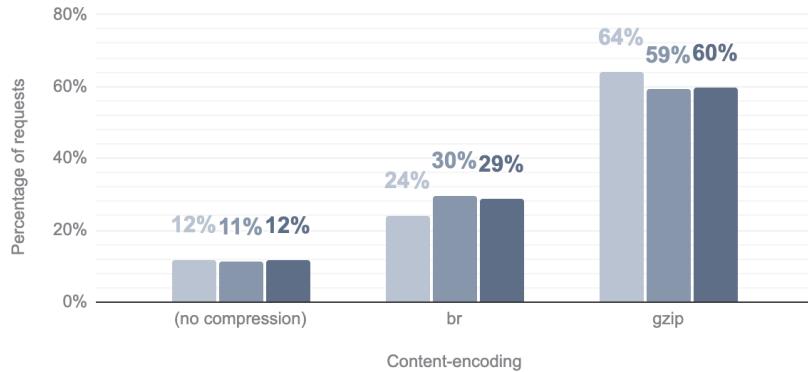


Figure 8.15. Percentage of third-party script requests by content-encoding type and by year

However, Brotli adoption among third parties stayed more or less at the same level, changing from 24% to 29%. Despite the slight positive tendency, there is still room for improvement of

Brotli adoption for third parties.

Usage of third-party facades

There are multiple techniques to eliminate render-blocking resources. One of them is third-party facades²⁷¹ that are useful for visual content like YouTube videos or interactive widgets like a live chat. They help to exclude third parties from the critical loading sequence and lazy load them. Lighthouse has introduced an audit Lazy load third-party resources with facades²⁷². There are multiple third-party facade solutions, for example, lite-youtube-embed²⁷³, lite-youtube²⁷⁴, or some custom approaches, and only a small number of them are in the list of third parties²⁷⁵ checked during the audit. This limitation makes it complicated to assess third-party facade usage across the web at this time.

Usage of `async` and `defer`

Using `async` and `defer` is another technique that could be used by website developers to optimize the loading of render-blocking third-party scripts.

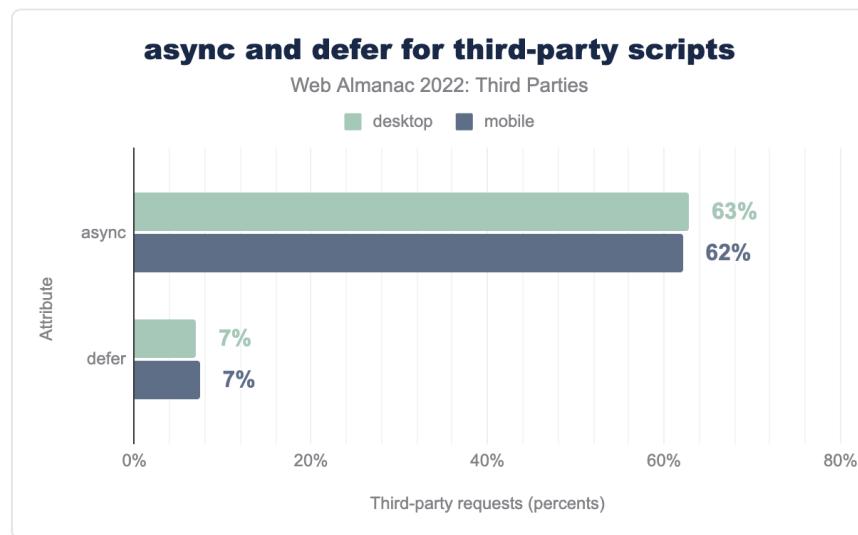


Figure 8.16. Percentage of all third-party script requests by `async` and `defer` attributes.

271. <https://web.dev/third-party-facades/>

272. <https://github.com/GoogleChrome/lighthouse/blob/master/core/audits/third-party-facades.js>

273. <https://github.com/paulirish/lite-youtube-embed>

274. <https://github.com/justinribeiro/lite-youtube>

275. <https://github.com/patrickkhulce/third-party-web/blob/master/data/entities.js>

The `async` attribute is considerably more popular than `defer`. It is used for 62% of total third-party scripts on mobile devices. Usage of the `async` attribute can still result in a render-blocking script as its execution can start during HTML parsing. The `async` attribute is useful for critical resources that are important during the page loading and can interrupt rendering.

The fact that `async` is more used demonstrates that third-party scripts are mostly treated as critical resources. Although this is true for some scripts, many third parties, for example, a video player is less critical. Deferred scripts potentially have a better impact on page rendering time which is reflected in core web performance metrics like Largest Contentful Paint²⁷⁶. Website developers should consider using `defer` for third-party assets that are not important for the critical rendering path.

Which resources are critical and which could be deferred might be a tricky question to consider, especially when considering Consent Management third parties that enable other third parties to be used. For example, analytics scripts are usually considered important for site owners but can't be used without the user's consent in countries with GDPR²⁷⁷ or similar legislation, making user consent third-party critical. Loading consent third-party resources in a critical path may result in a bad user experience causing Cumulative Layout Shifts and First Input Delay. Therefore, developers should strive for a balance between the way third parties are loaded and a good user experience.

Legacy JavaScript

Despite JavaScript's rapid involvement, the prevalence of legacy code is still significant. We are using one of the Lighthouse audits to check how many third parties are serving legacy JavaScript to modern browsers²⁷⁸.



Figure 8.17. Percent of legacy JavaScript Lighthouse audit failure caused by third-party

In general, third parties account for 59% of Lighthouse legacy JavaScript audit failures. A closer look into the audit results highlights the top 5 third-party script providers that include legacy JavaScript.

276. <https://web.dev/lcp/>
 277. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation
 278. <https://github.com/GoogleChrome/lighthouse/blob/master/core/audits/byte-efficiency/legacy-javascript.js>

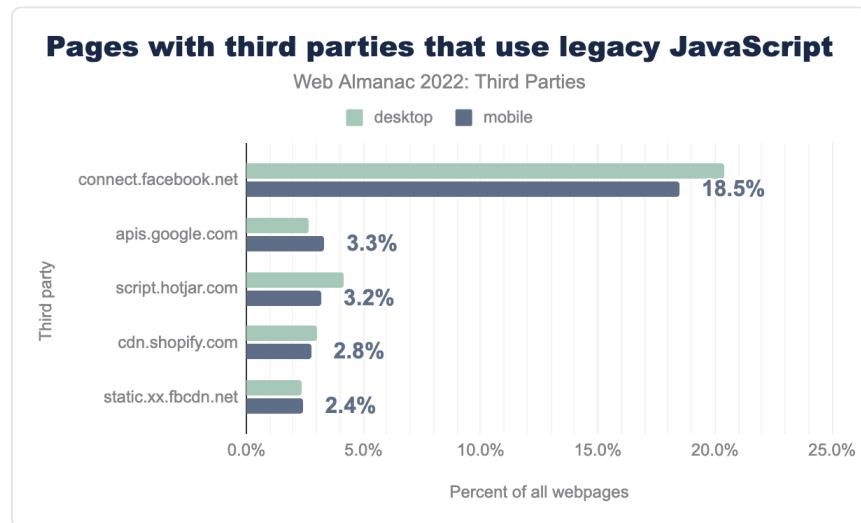


Figure 8.18. Percentage of websites using third parties that have legacy JavaScript

Facebook is a third-party with legacy JavaScript that affects the most pages. It introduces legacy code to around 20% of the total number of web pages on mobile and desktop devices correspondingly when looking at both `facebook.net` and `fbcdn.net` from the above graph. Nowadays, when old browsers like Internet Explorer may no longer need to be supported, the necessity to keep legacy JavaScript becomes lower. Despite this fact, the trends of using Facebook resources that have legacy JavaScript in the past 3 years reveal that the numbers actually increased—from around 14% in 2020 to 18% in 2022 for `facebook.net` alone. This is due to the increasing number of websites that embed this third party.

Serving legacy JavaScript to modern browsers results in a larger amount of redundant and slower code. We can look into this more by analyzing the size of unused JavaScript.

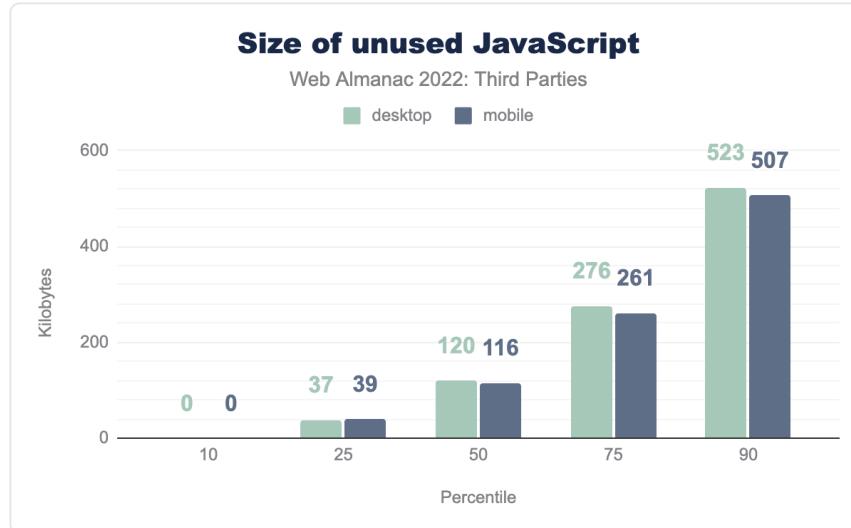


Figure 8.19. Size of unused third-party JavaScript.

The median amount of unused third-party JavaScript is approximately 120 KB. For 25% of websites that use third-party scripts, it is more than 261 KB.

Unfortunately, website owners do not always have the possibility to change the way third-party JavaScript is bundled. However, if the third-party dependencies are self-hosted they can be optimized during development by adopting modern script bundling approaches²⁷⁹ that could help to reduce the amount of unused code.

Other optimization technologies

One of the third-party resource management problems is that sometimes it can skip the development team and be added using tag management tools without proper web performance evaluation. As a result, third-party scripts can uncontrollably affect the page loading and responsiveness experience.

Some modern third-party loading and execution solutions have appeared in recent years. For example, Partytown²⁸⁰ is a library that relocates third-party scripts into the web worker to free up the main thread for first-party code. Currently, the library is in the early adoption stage and its usage is very low. Only 70 websites from the whole dataset are using it in 2022. However, the Next.js framework has started to introduce this solution²⁸¹ that could increase Partytown's

279. <https://web.dev/publish-modern-javascript>

280. <https://partytown.builder.io/>

281. <https://nextjs.org/docs/basic-features/script#off-loading-scripts-to-a-web-worker-experimental>

popularity.

The previous sections showed that the responsibility for third-party negative impact is split between first and third-party developers. However, browsers are also showing interest in optimizing the loading of third-party resources²⁸². The proposals include better real user monitoring and developer tooling providing more data about the impact of third parties on their websites.

25%

Figure 8.20. Percent of third-party requests with Timing-Allow-Origin header

That might be challenging to achieve given only 25% of total third-party requests provide the Timing-Allow-Origin (TAO) header²⁸³ that is important for third-party web performance data transparency.

Taking into account that the TAO header prevalence has not improved in comparison to the previous years²⁸⁴, we would encourage third-party providers to use it more actively, to allow first parties to get more accurate insights into the performance of these resources.

Conclusion

Third parties are often associated with a negative impact on website performance. Indeed some account for notable rendering and main thread blocking time, especially on mobile devices, which are increasingly the more popularly used devices. However, the main goal of this chapter is to show that the responsibility for third-party impact on web performance is shared between third-party providers and website owners. There are lots of opportunities for website developers to lower the third-party impact on their websites. In the future, browsers also might look to automatically apply third-party resource optimizations.

We have analyzed data related to different web performance recommendations, including compressed and minified resources, legacy APIs, unused JavaScript, etc. Based on the findings, we have conducted the following action points that could be helpful for website and third-party developers to improve the user experience:

- Load third-party resources suitable for production environments where the assets

282. <https://developer.chrome.com/blog/third-party-scripts/#proposed-approach>

283. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Timing-Allow-Origin>

284. <https://almanac.httparchive.org/en/2021/third-parties#timing-allow-origin-header-prevalence>

are minified and compressed.

- Leverage different third-party facade techniques, especially for “heavy” content like videos, maps, and live chats, that can block render and have a crucial impact on First Contentful Paint.
- While evaluating third-party candidates, ensure that they are not serving legacy APIs except where necessary.
- Consider how critical third-party content is for the page and load the non-critical resources using the `defer` attribute when it is not render-blocking.
- Explore modern third-party load and execution strategies.
- Choose Brotli compression over gzip.

There are many more optimization opportunities out there! We encourage web developers to take them so that functionality provided by third parties would serve websites without harming the user experience.

Author



Eugenia Zegisova

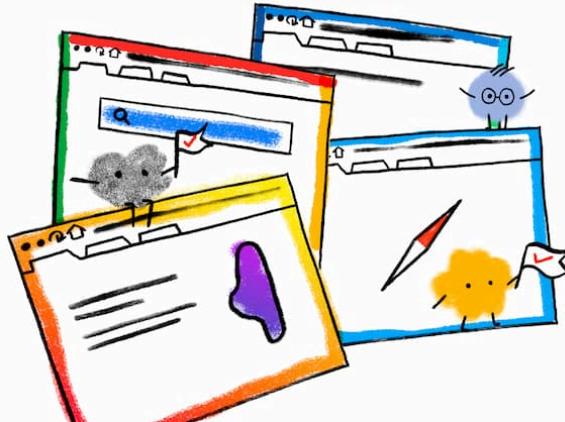
 @jevgeniazi  imeugenia  <https://github.com/imeugenia/speaking/blob/main/README.md>

Eugenia is a frontend engineer and tech event speaker who is passionate about web performance and state machines. She has experience working in fast-growing Berlin-based startups like N26 and Gorillas, and now she has joined RapidAPI²⁸⁵. She ran a Google Developer Group in Latvia for several years.

^{285.} <https://rapidapi.com/>

Part I Chapter 9

Interoperability



Written by Brian Kardell

Reviewed by Eric A. Meyer and Philip Jägenstedt

Analyzed by Rick Viscomi and Kevin Farrugia

Edited by Barry Pollard

Introduction

In 2019, the Mozilla Developer Network's (MDN) Product Advisory Board put together a significant survey of over 28,000 developers and designers from 173 countries. Findings from this were published as the first Web Developer Needs Assessment²⁸⁶ (Web DNA). This study identified—among other things—that some of the key frustrations and pain points most often involved differences between browsers. In 2020 this led to a followup known as the MDN Browser Compatibility Report²⁸⁷.

Historically, implementer priorities and focus are independently managed. However, given this new data, browser manufacturers came together for another first-of-its-kind effort called Compat 2021, which identified 5 specific areas of joint focus toward alignment across thousands of Web Platform Tests. At the beginning of Compat 2021, all engines scored only 65-70% compatibility in the five areas in stable, shipping browsers. Today, all of them are over

²⁸⁶. <https://insights.developer.mozilla.org/reports/pdf/MDN-Web-DNA-Report-2019.pdf>

²⁸⁷. <https://insights.developer.mozilla.org/reports/mdn-browser-compatibility-report-2020.html>

90%. In 2022, this effort was expanded—and renamed—to *Interop 2022*.

Both of these efforts offer some different things for this chapter to look at. It's been nearly a year since most improvements from Compat 2021 shipped, and while many things in Interop 2022 are already deployed in shipping browsers, there is more to come before the end of the year.

An interesting question in these efforts is “how do we know that we did well (or didn’t)?” Seeing significant score improvements is useful, but insufficient without developer adoption. So, this year for the first time, the Web Almanac will also include a new Interoperability chapter to begin wrestling with these questions and provide some central information to developers about what’s changed, and what’s worth another look.

This chapter will summarize the work done in Compat 2021 and measure what we can, as well as look into what’s happening in Interop 2022 and consider whether there are also potentially valuable metrics we can track over time. Both of these efforts contain a wide mix of cases from stable, already useful features with varying degrees of incompatibility or frustration to brand new things we tried to set off right from the start.

Compat 2021

Compat 2021 had 5 major focus areas

- Grid
- Flexbox
- Sticky Position
- Transforms
- Aspect Ratio

In January 2021, all stable/shipping browsers scored 65-70% compatibility in these areas, and it wasn't necessarily the same 30-35% of tests that were failing in each browser.

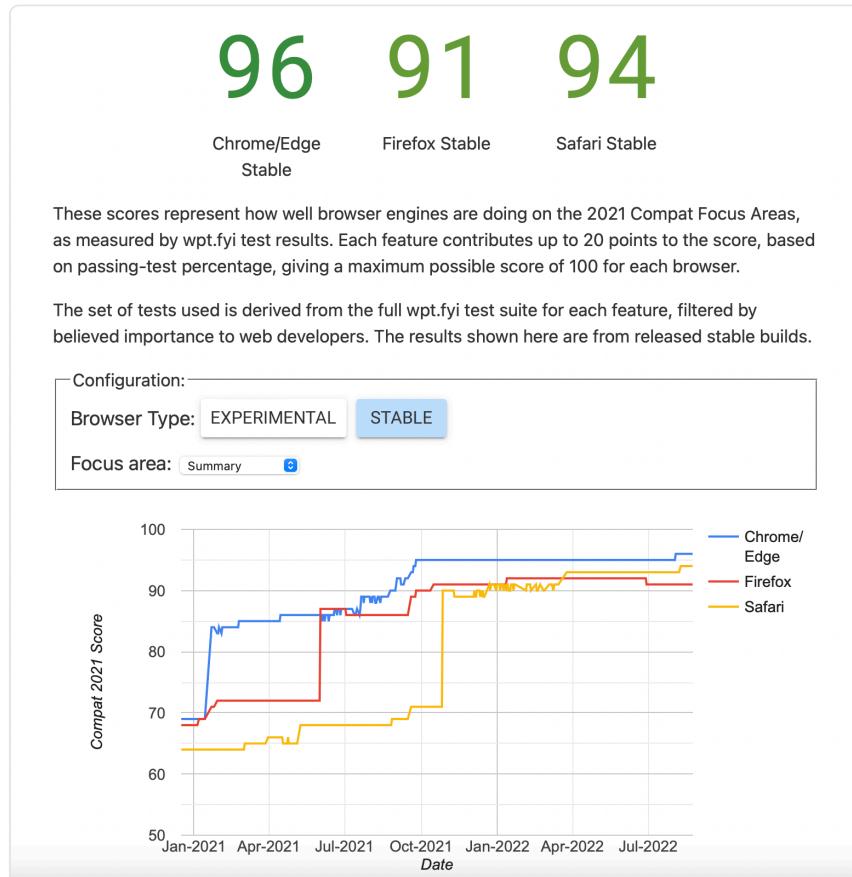


Figure 9.1. Compat 2021 dashboard.
(Source: Web Platform Tests²⁸⁸)

Today, you can see that significant levels of improvement have been made. Chrome and Edge are at 96%, Firefox at 91%, and Safari at 94%.

Grid

CSS Grid is one of the most popular features in many years. The HTTP Archive data shows year over year doubling of adoption since its arrival, with a slight slowdown this year—only increasing half-again instead of doubling. Grid already had quite a high degree of interoperability, but there were still a number of minor differences in support. Work was done

²⁸⁸. <https://wpt.fyi/compat2021>

throughout 2021 and 2022 to improve alignment of the over 900 tests in Web Platform Tests that test features of Grid. If you've had past headaches trying to do something in Grid, give it another try—the situation may have changed for the better.

A good example of this is the ability to animate grid tracks—grid rows and columns—which as of mid-2022 was only supported by Firefox. However, as this chapter was being written, grid-track animation was added to both WebKit²⁸⁹ and Chromium²⁹⁰, meaning all three major engines should be animating grid tracks by the time you read this.

Flexbox

Flexbox is even older and more widely used. This year its use has grown again, now appearing on 75% of mobile pages and 76% of desktop pages. It has a similar number of tests to Grid and despite very wide adoption started in much worse shape. Entering 2021, we had a combination of ragged bugs and sub-features that remained under-implemented. For example, positional-alignment keyword values²⁹¹ (which can be applied to justify-content and align-content and also to justify-self and align-self) had ragged support and several interoperability issues. For absolute positioned flex items this was even worse. These issues have been resolved.

A large, bold, blue sans-serif font displays the number "112,323". The comma is positioned between the first two digits of the hundreds place, indicating a count of 112 thousand, 323 units.

Figure 9.2. Desktop pages using `flex-basis: content` in their stylesheets.

Another bit of focus was toward `flex-basis: content`, which is used to automatically size based on the flex item's content. This was initially implemented in Firefox, but implementations in WebKit and Chromium were underway in 2021. Today these tests pass uniformly in all browsers and `flex-basis: content` appears on 112,323 pages on desktop and 75,565 mobile, roughly 1% of pages. That's not a bad start for a feature in its first year of universal support and about double what it was last year. We'll keep an eye on this metric in the future.

289. <https://webkit.org/blog/13152/webkit-features-in-safari-16-0/>

290. <https://groups.google.com/g/chromium.org/g/blink-dev/c/L17brOgiMk8/m/I4WNHdatBQAJ>

291. https://developer.mozilla.org/docs/Web/CSS/CSS_Box_Alignment#positional_alignment_keyword_values

Sticky positioning

5.5%

Figure 9.3. Desktop pages using `position: sticky` in their stylesheets.

Sticky positioning has been around for a while. In fact, it's worth noting that it is the most popular feature query in used by a large margin²⁹², accounting for over 50% of feature queries. It had several interoperability issues; for example, the inability to stick headers in tables in Chrome. `position: sticky` is actively used in around 5% of desktop pages and 4% of mobile pages in 2022. We'll keep an eye on this metric for some time to come to see how addressing those interoperability issues affects adoption over time.

CSS transforms

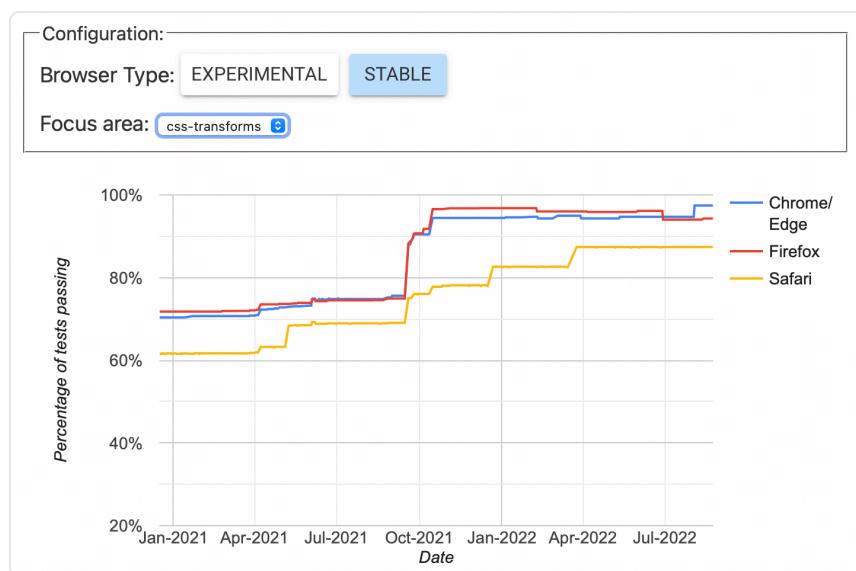


Figure 9.4. CSS Transforms Web Page Tests dashboard (stable).
(Source: Web Platform Tests²⁹³)

CSS Transforms are popular and have been around for a long time. However, there were many

292. <https://almanac.httparchive.org/en/2022/css#feature-queries>

293. <https://wpt.fyi/compat2021?feature=css-transforms&stable>

interoperability issues at the start, particularly around `perspective:none` and `transform-style: preserve-3d`. This meant that many animations were annoyingly inconsistent²⁹⁴.

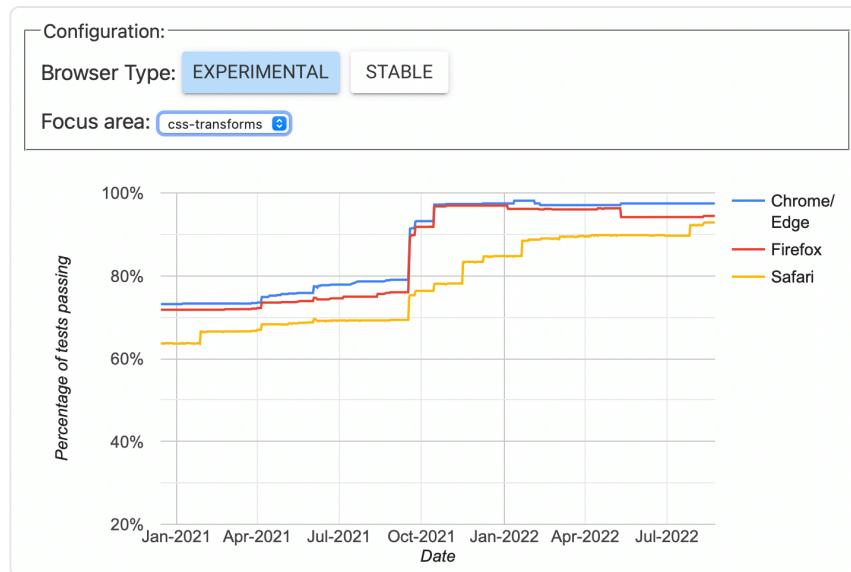


Figure 9.5. CSS Transforms Web Page Tests dashboard (experimental).
(Source: Web Platform Tests²⁹⁵)

A recent compat 2021 graph showing the same CSS transforms in experimental browsers as above shows all browsers are scoring 90% or better in their experimental versions, which show future versions of the browsers. This was one of the areas with big, visible improvements in stable browsers, which continue to improve, as part of Interop 2022 involves continuing Compat 2021 work.

aspect-ratio

`aspect-ratio` was a new feature developed in 2021. Given its potential widespread usefulness, we chose to aim for high interoperability from the start.

294. <https://web.dev/compat2021/#css-transforms>
295. <https://wpt.fyi/compat2021?feature=css-transforms>

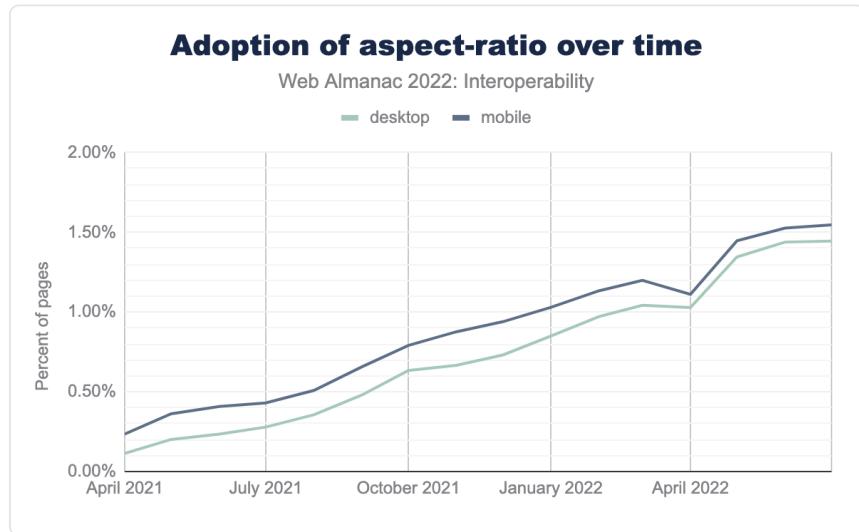


Figure 9.6. Aspect-ratio usage over time.
(Source: Chrome Status²⁹⁶)

In 2022, `aspect-ratio` is already appearing in the CSS of 2% of URLs in the archive crawl. Note that doesn't mean that 2% of these pages are using `aspect-ratio` themselves: rules may be loaded for use in other pages on the site. Which rules are applied in those pages is a different question, and it shows a more modest 1.55% of page views on desktop and 1.44% on mobile. Still, the growth chart shows steady and increasing adoption. This will be an interesting metric to track as we move forward.

Interop 2022

Like the *Compat* effort before it, the renamed *Interop* effort focuses on a mix of things from collections of bugs to landing good, final implementations to relatively new but quickly shipping features we'd like to set off in good standing. Let's start with the bugs...

Bugs

In many cases, we have otherwise mature features with ragged bugs reported in different browsers. Ragged bugs mean that the authoring experience is potentially a lot worse than individual pass rates might imply. For example, if all browsers report a pass rate on a series of

²⁹⁶. <https://chromestatus.com/metrics/css/timeline/popularity/657>

tests of 70%, but all browsers fail on a different 30%, interoperability in practice would be quite low. A significant portion of our focus in Interop 2022 is around aligning implementations and closing bugs on features like these.

Forms

For most of the web's history, forms have played a pretty important role. In 2022, over 69% of desktop pages include a `<form>` element. They've had a lot of investment, but despite that, they're still the source of a lot of browser bugs as developers find cases where things differ from the specs, or differ from other implementations in sometimes subtle ways. We identified a set of 200 tests²⁹⁷ in which the pass rate was very ragged. Individual scores ranged from ~62% (Safari) to ~91% (Chrome), but again, each browser had different gaps in support.

We have made some pretty radical progress toward closing these gaps in experimental releases, and we hope that as we close out the year these will land in stable browsers. There is probably little that can be tracked here using the HTTP Archive data in terms of use, or adoption, but hopefully developers experience less pain and frustration and require fewer workarounds for individual browsers.

297. <https://wpt.fyi/results/?label=master&label=experimental&product=chrome&product=firefox&product=safari&aligned&view=interop&q=label%3Ainterop-2022-forms>



Figure 9.7. Forms WPT dashboard (experimental).
(Source: Web Platform Tests²⁹⁸)

Scrolling

Over the years we've added new patterns and developed new abilities around scroll experiences like `scroll-snap`, `scroll-behavior`, and `overscroll-behavior`. The desire for these sorts of powers are clear—in 2022, the number of CSS stylesheets including some of these key properties looked like this:

²⁹⁸. <https://wpt.fyi/interop-2022?feature=interop-2022-forms&stable>

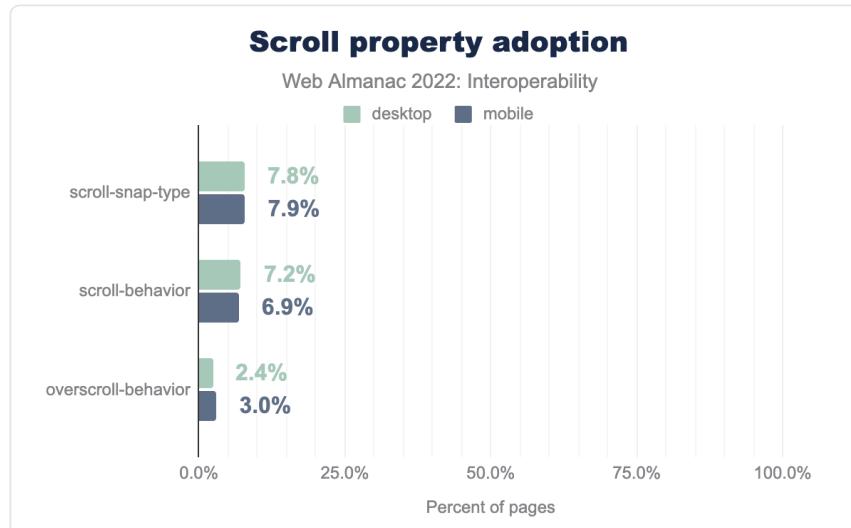


Figure 9.8. Scroll property adoption.

Unfortunately, this is an area where a number of incompatibilities remain, and dealing with incompatibilities in scrolling causes developers a lot of pain. We identified 106 Web Platform Tests²⁹⁹ around scrolling. At the beginning of the process, stable-release scores ranged from ~70% (Firefox and Safari) to about 88% (Chrome). Again, keep in mind that these are overall scores—because the gaps differed, the real “interoperability” intersection was lower than any of these.

²⁹⁹. <https://wpt.fyi/results/css?label=master&label=experimental&product=chrome&product=firefox&product=safari&aligned&view=interop&q=label%3Ainterop-2022-scrolling>

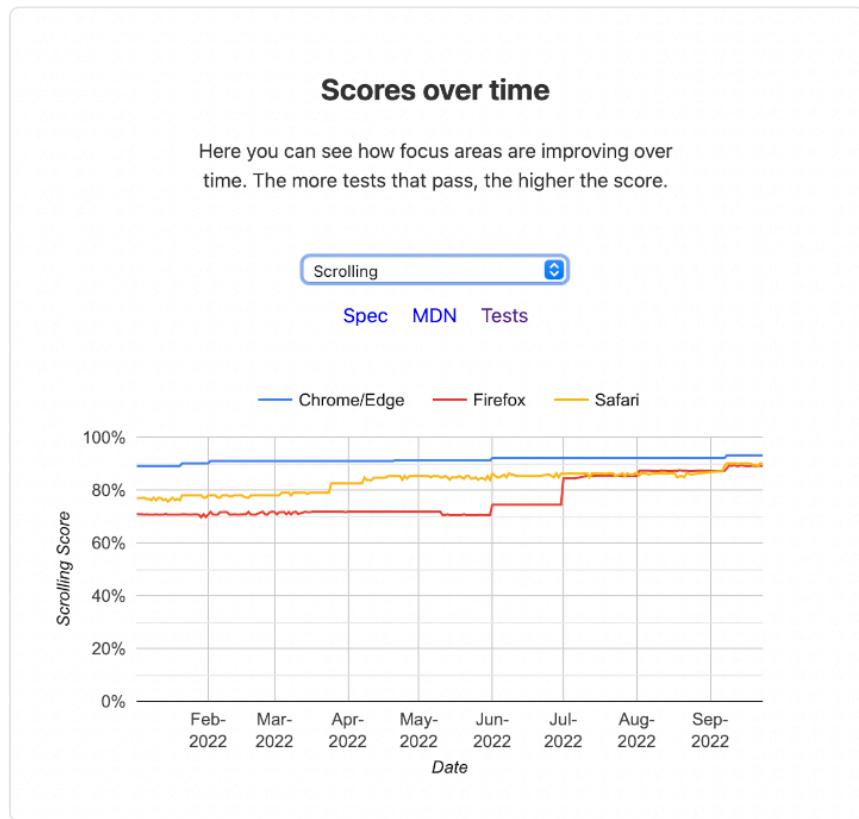


Figure 9.9. Scrolling WPT dashboard.
(Source: Web Platform Tests³⁰⁰)

It is very difficult to estimate what effect these improvements will have on adoption over time, but we'll keep an eye on these metrics. In the meantime, if you've experienced some interoperability pains with scrolling features, you might give them another look. We hope that as these improvements continue and reach stable browser releases, the experience will get a lot better.

Typography and Encodings

Rendering of text is sort of the web's forte. Like forms, many basic ideas have been around forever, but there remain a number of gaps and inconsistencies around support for typography and encodings.

^{300.} <https://wpt.fyi/interop-2022?feature=interop-2022-scrolling&stable>

Interop 2022 took up a general bag of issues around `font-variant-alternates`, `font-variant-position`, the `ic` unit, and CJK text encodings. We identified 114 tests in Web Platform Tests³⁰¹ representing different sorts of gaps.

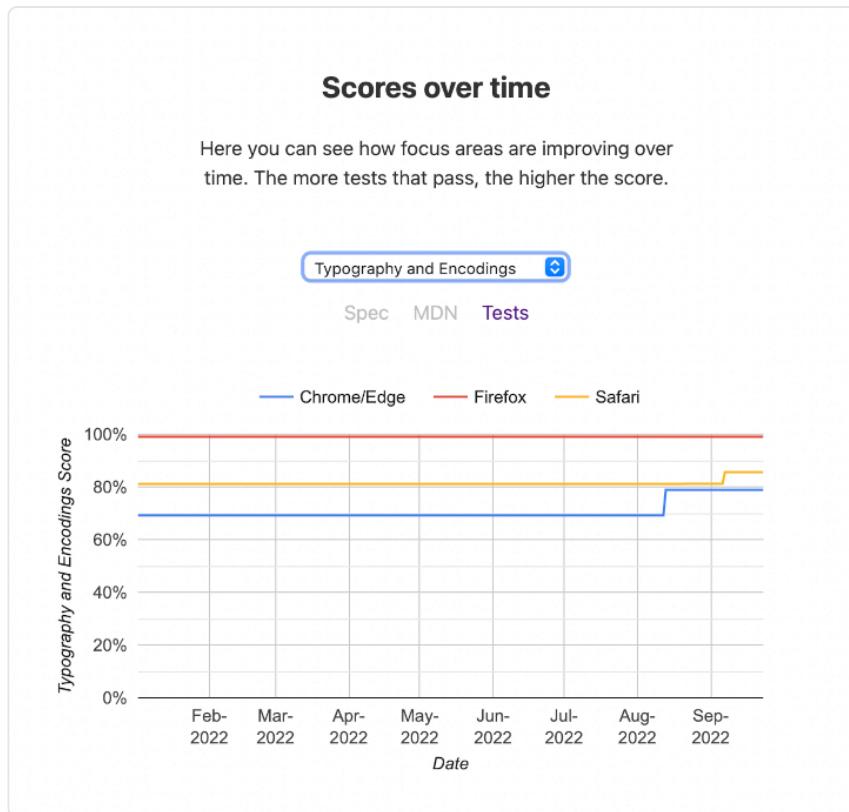


Figure 9.10. Typography and encodings WPT dashboard.
(Source: Web Platform Tests³⁰²)

Chrome has recently begun to close gaps with Safari, but both Safari and WebKit still require some attention to catch up to the completeness of Firefox in this area.

Completing Implementations

Aligning implementations is particularly difficult. There is a delicate balance between the need for experimentation and initial implementation experience and having enough agreement to

301. <https://wpt.fyi/results/?label=master&label=experimental&product=chrome&product=firefox&product=safari&aligned&view=interop&q=label%3Ainterop-2022-text>

302. <https://wpt.fyi/interop-2022?feature=interop-2022-text&stable>

ensure that the work is well understood and very likely to reach the status of shipping implementation in all browsers. Sometimes this alignment can take years. This year we've focused on three items which had an implementation and at least some agreement that it's ready: The `<dialog>` element, CSS Containment, and Subgrid. Let's look at each.

`<dialog>`

A dialog element was first shipped in Chrome 37 in August 2014. Introducing a dialog requires introducing and defining a number of supporting concepts like "top-layer" and "inertness." It also requires answering many new accessibility and UX questions.

A number of things caused work on dialogs to stall for a long time, but over the years it's picked back up. It landed in Firefox Nightly 53 behind a flag in April 2017. Since then, a lot of work has gone into answering all of those questions. Final details were sorted out and work was prioritized as part of Interop 2022 to ensure good interoperability to start with. We identified 88 Tests. It was shipped by default in stable browsers in both Firefox 98³⁰³ and Safari 15.4³⁰⁴ in March 2022, with all browsers scoring ~93% or better.

303. <https://developer.mozilla.org/docs/Mozilla/Firefox/Releases/98>
304. https://developer.apple.com/documentation/safari-release-notes/safari-15_4-release-notes

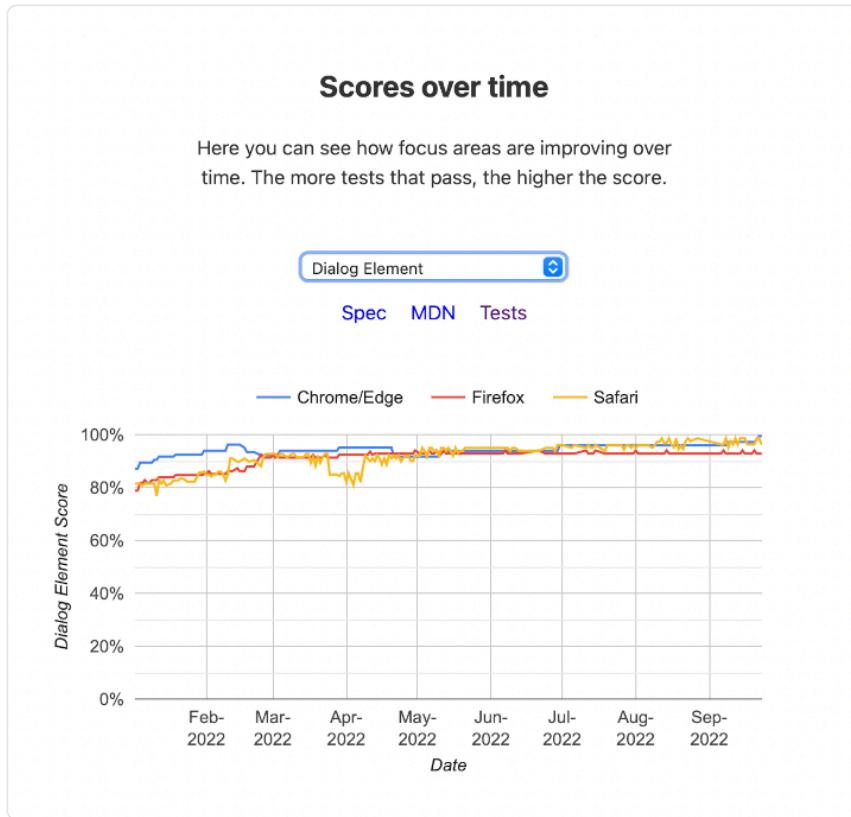


Figure 9.11. Dialog element WPT dashboard.
(Source: Web Platform Tests³⁰⁵)

It's hard to predict how many of the pages the archive crawls will require a `<dialog>`, but tracking its growth will be informative and interesting. Last year, only one shipping browser supported `<dialog>`, and it appeared on ~0.101% of pages in the mobile data set. At the time of the crawl we used for this chapter, it had been shipping universally for about 5 months and we found it appearing in ~0.148%. Still small numbers, but that's ~146% of what it was this time last year. We will continue to track this metric next year. In the meantime, if you have a need for a `<dialog>` there's good news: It's now universally available for use!

CSS containment

CSS containment introduces a concept for isolating a subtree of the page from the rest of the

305. <https://wpt.fyi/interop-2022?feature=interop-2022-dialog&stable>

306. https://docs.google.com/spreadsheets/d/1grkd2_1xSV3jvNK6ucRQ0OL1HmGTsSchuwA8GZuRLHU/edit#gid=2057119066

page in terms of how CSS should process and render it. It was introduced as a primitive which could be used to improve performance, and to open the door for figuring out Container Queries³⁰⁷.

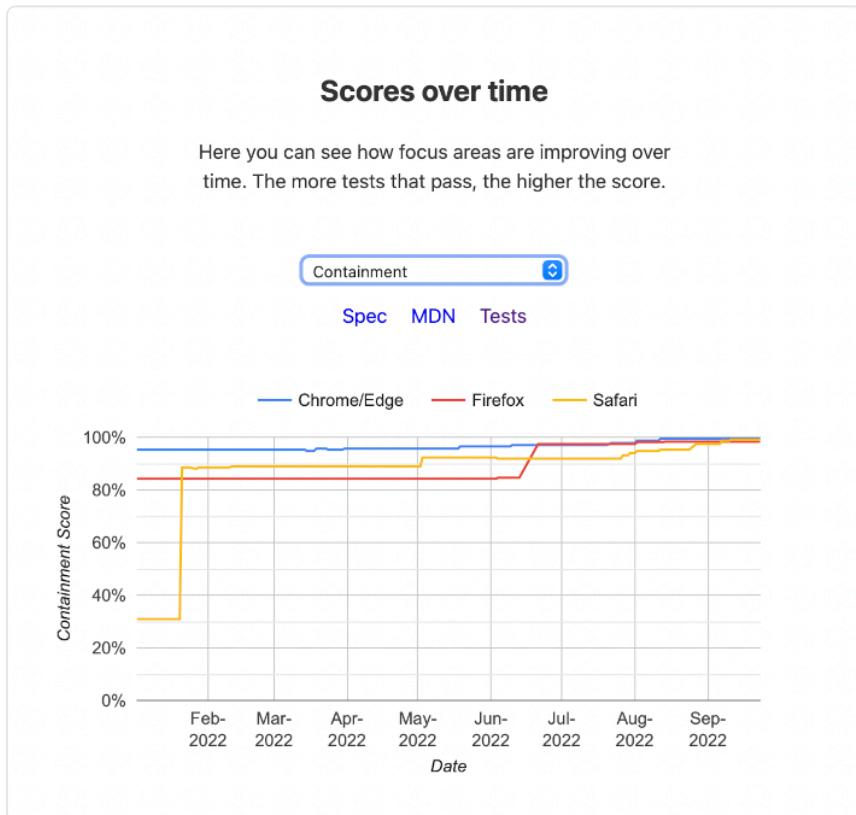


Figure 9.12. Containment WPT dashboard.

(Source: Web Platform Tests³⁰⁸)

It first shipped in Chrome stable in July 2016. Firefox shipped the second implementation in September 2019. This year it was taken up by Interop 2022 to align and ensure that as it becomes universally available, we start out in good shape. We identified 235 tests³⁰⁹. Safari shipped containment support in stable release 15.4³¹⁰ in March 2022. Throughout the year, each browser improved support, and it is now universally available.

307. https://developer.mozilla.org/docs/Web/CSS/CSS_Container_Queries

308. <https://wpt.fyi/interop-2022?feature=interop-2022-contain&stable>

309. <https://wpt.fyi/results/css/css-containslabel=master&label=experimental&product=chrome&product=firefox&product=safari&aligned&view=interop&q=label%3Ainterop-2022-contain>

310. https://developer.apple.com/documentation/safari-release-notes/safari-15_4-release-notes

3.7%

Figure 9.13. Number of mobile pages using containment in their stylesheets.

In the 2022 data, containment appears in stylesheets on 3.7% of pages on mobile and 3.1% of pages on desktop.

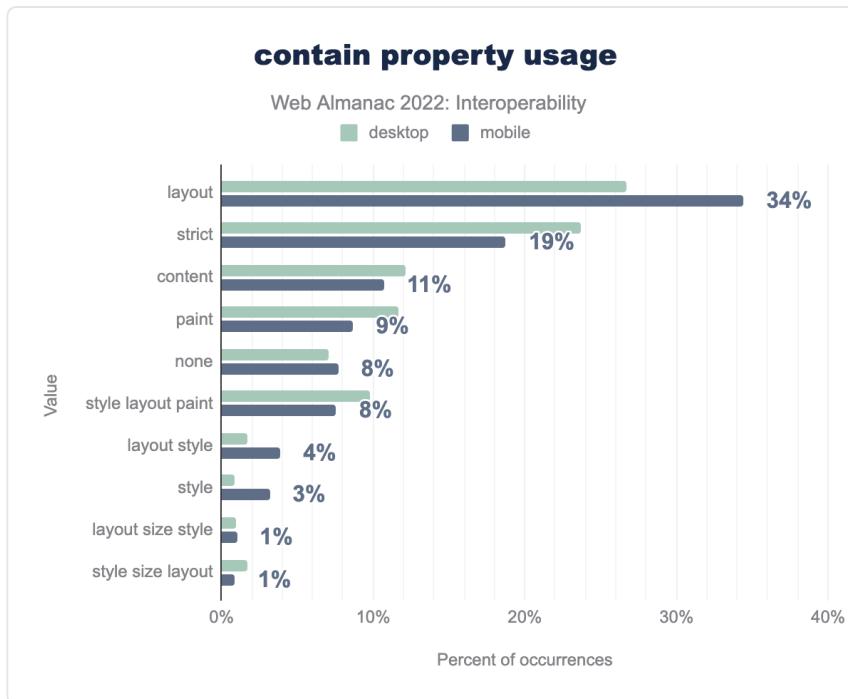


Figure 9.14. `contain` property usage.

The figure above shows the relative appearance of values in those pages—`layout` containment being far and away the most popular usage, accounting for 34% of `contain` values.

While it is useful on its own, additional levels of containment support are a prerequisite to supporting container queries, so this will be an interesting metric to continue to track over time as container queries is the #1 most requested CSS feature³¹¹ for many years. Now that

³¹¹ https://2021.stateofcss.com/en-US/opinions/#currently_missing_from_css_wins

containment is universally available, it's a great time for you to have a look and familiarize yourself with the basic concepts.

Note that some degree of container queries support is already available in Chrome and Safari and polyfills are available, so we also decided to look at how many stylesheets already contain a `@container` ruleset, wondering how much this would account for the use we saw above.

0.002%

Figure 9.15. Percentage of mobile pages containing a `@container` ruleset.

Not much, yet it would seem! A mere 238 pages, out of the nearly 8 million pages we crawled in our mobile data set use container queries. Given that it is brand new and not yet completely shipping, this isn't surprising. It does give us a nice baseline to start tracking adoption from in the future though.

Subgrid

While CSS grid layout has allowed a container to express layout of its children in terms of rows, columns and tracks—there has always been something of a limit here. There is frequently a need for descendants that are not children to participate in the same grid layout. Subgrid³¹² is the solution for problems like this. It was first supported in a stable release in Firefox in December 2019, but other implementations didn't immediately follow.

Coordinating work on this long awaited feature and ensuring good interoperability was another goal in Interop 2022. We marked 51 Tests in Web Platform Tests³¹³.

312. https://developer.mozilla.org/docs/Web/CSS/CSS_Grid_Layout/Subgrid
313. <https://wpt.fyi/results/css/css-grid/subgrid?label=master&label=experimental&product=chrome&product=firefox&product=safari&aligned&view=interop&q=label%3Ainterop-2022-subgrid>

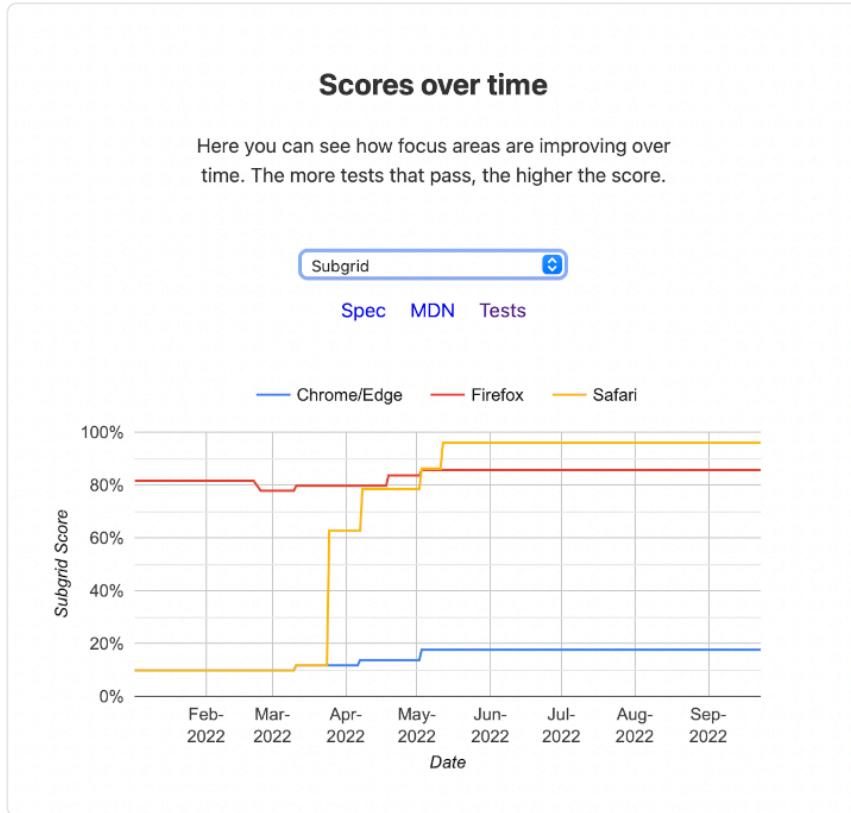


Figure 9.16. Subgrid WPT dashboard.
(Source: Web Platform Tests³¹⁴)

As of the time of this writing, there has been very good progress (Safari is now the most complete), and there are at least 2 implementations (Safari and Firefox) in stable shipping browsers. We hope to see rapid improvement in Chrome before the end of the year.

0.002%

Figure 9.17. Percentage of mobile pages containing a use of `subgrid` in their stylesheet.

While this isn't fully available in all stable browsers yet the dataset did include some small amount of use already.

³¹⁴ <https://wpt.fyi/interop-2022?feature=interop-2022-subgrid&stable>

New Features

This year, all of the new features that fall under the category of CSS, and most of the data about them, will be covered in the CSS chapter. Here, we'll mainly focus on some highlights.

Color Spaces and Functions

Color on the web has always been full of fascinating challenges. Over the years, we've given authors many ways to express what are—in the end—the same sRGB³¹⁵ colors. That is, one can write as a color name (`red`). Simple enough.

However, we could also use its hex equivalent (`#FF0000`). Humans don't generally think in hexadecimal, so we added the `rgb()` color function (`rgb(255, 0, 0)`). Note that both of those are just using two different, but equivalent, numbering systems. They are also about expressing things in terms of mixing the intensity of individual lights that were common in cathode ray tube displays.

The RGB method of constructing colors can be very hard for humans to visualize, so we developed other coordinate systems for expressing sRGB colors in a (perhaps?) easier to understand, like `hsl(0, 100%, 50%)` or `hwb(0, 0%, 0%)`. However, again, these are sRGB coordinate systems.

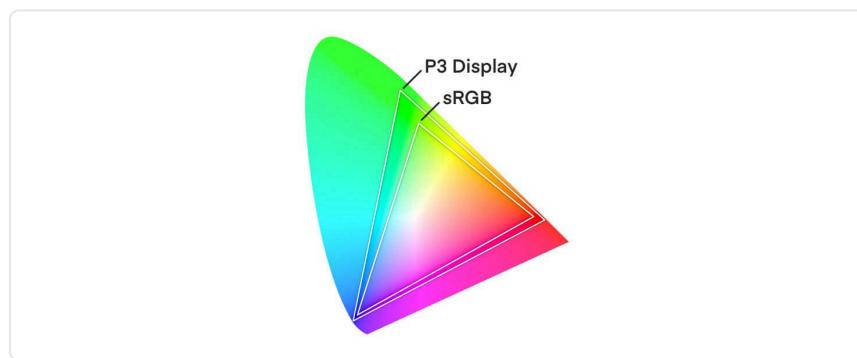


Figure 9.18. p3 color space compared to sRGB.

So, what happens when our display abilities exceed their limits? This is, in fact, the case today, as can be seen with wide gamut displays.

In Safari 10, released in 2017, Apple added support for P3 color images. The new `lab()` and `lch()` coordinate systems were added to CSS in order to support the full available gamut

315. <https://en.wikipedia.org/wiki/SRGB>

space, based on the CIELAB model³¹⁶. They are perceptually uniform, allowing us to express colors we could not previously (and defining what to do if support is lacking). Support for these first shipped in Safari 15 in September 2021.

The fuller gamut space and better perceptual uniformity of `lab()` and `lch()` also allow us to more easily focus on new color functions like `color-mix()`, which takes two colors and returns the result of mixing them in a specified color space by a specified amount.

Interop 2022 took up 189 tests³¹⁷ around these items with a goal of prioritizing good interoperability. Safari began pretty well out ahead and has only improved—both Firefox and Chrome have made steady improvements, but they’re still quite a bit behind in this area. One challenge, inevitably, is that much lower-level support—throughout the underlying graphics library, rendering pipeline, etc.³¹⁸—is also built to deal in sRGB, so adding support is not easy.

316. https://en.wikipedia.org/wiki/CIELAB_color_space#Cylindrical_model

317. <https://wpt.fyi/results/css/color/labeled=master&labeled=experimental&product=chrome&product=firefox&product=safari&aligned&view=interop&q=label%3Ainterop-2022-color>

318. <https://youtu.be/eHZVuHKWdd8?t=906>

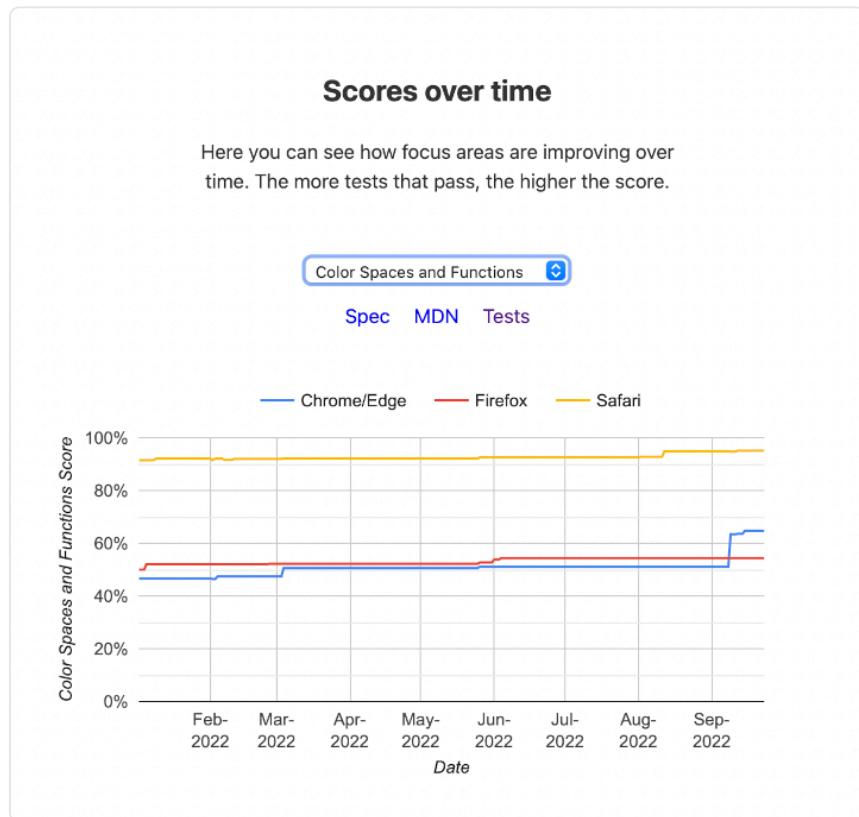


Figure 9.19. Color spaces and functions WPT dashboard.
(Source: Web Platform Tests³¹⁹)

Viewport Units

In the 2020 MDN Browser Compatibility Report, the ability to work with the reported size of the viewport with existing vh/vw units was a common theme³²⁰. As browsers experiment with different interface choices and websites have different design needs, the CSS Working Group defined several new classes of viewport units³²¹ for measuring the largest (`lv*` units), smallest (`sv*` units) and dynamic (`dv*` units) viewport measures. All viewport related measures includes similar units:

- 1% of the width (`vw`, `lvw`, `svw`, `dvw`)

319. <https://wpt.fyi/interop-2022?feature=interop-2022-color&stable>

320. <https://insights.mozilla.org/reports/mdn-browser-compatibility-report-2020.html#findings-viewport>

321. <https://drafts.csswg.org/css-values-4/#viewport-variants>

- 1% of the height (`vh`, `lvh`, `svh`, `dvh`)
- 1% of the size in the inline axis (`vi`, `lvi`, `svi`, `dvi`)
- 1% of the size of the initial containing block (`vb`, `lvb`, `svb`, `dvb`)
- The smaller of two dimensions (`vmin`, `lvmin`, `svmin`, `dvmin`)
- The larger of two dimensions (`vmax`, `lvmax`, `svmax`, `dvmax`)

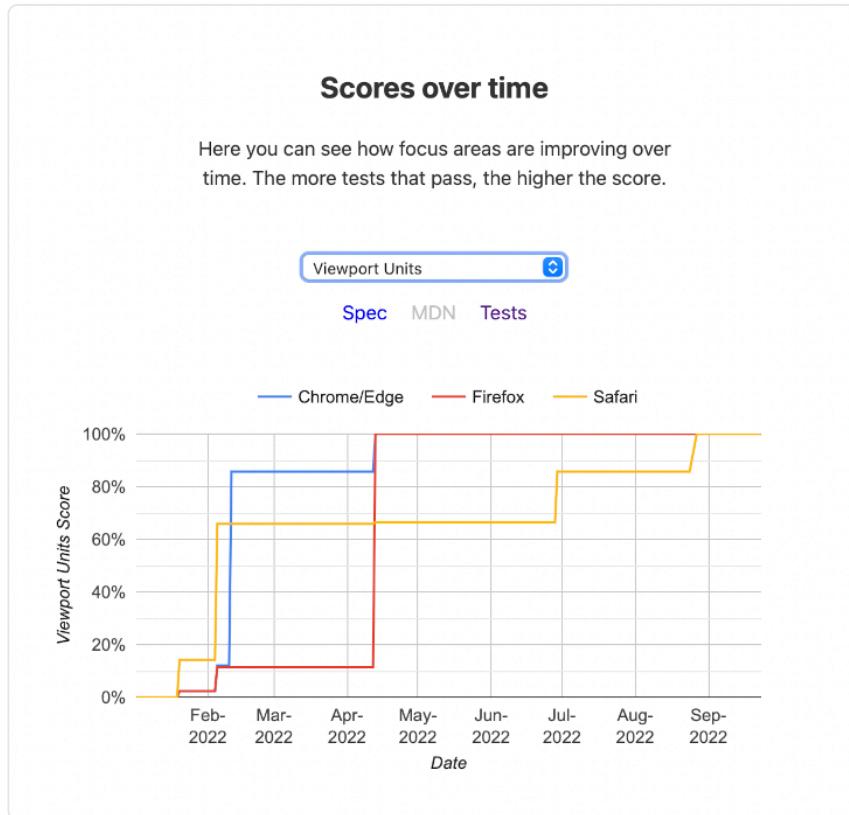


Figure 9.20. Viewport units WPT dashboard (experimental).
(Source: Web Platform Tests³²²)

Interop 2022 identified 7 tests to verify various aspects of those units. Safari shipped the first support for these units in March 2022, followed by Firefox at the end of May. As of the time of this writing it is supported in experimental builds of Chromium.

³²² <https://wpt.fyi/interop-2022?feature=interop-2022-viewport>

As of the time of this writing, the HTTP Archive hasn't turned up any use of these units in the wild yet, but it's very new. We'll continue to track adoption on this going forward.

Cascade Layers

Cascade Layers are an interesting new feature of CSS, built on a fundamental idea that has existed in CSS all along. As authors, our primary means of expressing the importance of rules has been specificity. This works well for a lot of things but it can quickly get unwieldy as we try to incorporate ideas for design systems or components. Browsers also use CSS internally in what is called the UA stylesheet. However, you might note that you don't typically have specificity related battles with the UA stylesheet. That's because there are "layers" of rules built right into how CSS works. Cascade Layers provides a way for authors to plug into that same mechanism and manage their CSS and specificity challenges more effectively. Miriam Suzanne³²³ wrote a fuller explanation and guide³²⁴.

323. <https://twitter.com/TerribleMia>
324. <https://css-tricks.com/css-cascade-layers/>

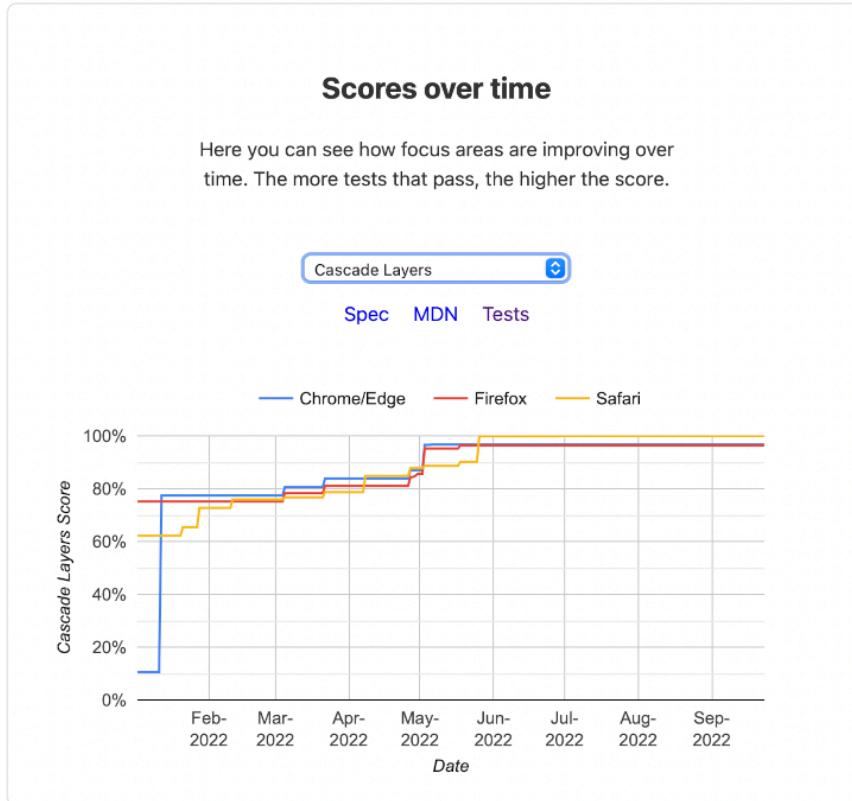


Figure 9.21. Cascade layers WPT dashboard (experimental)..

(Source: Web Platform Tests³²⁵)

To set this off well, Interop 2022 defined 31 tests in Web Platform Tests³²⁶. Support in stable browsers at the beginning of the year was non-existent, but since April it is now universally implemented among stable releases in the 3 engines. Here's what development looked like.

0.003%

Figure 9.22. Percentage of mobile pages containing a `@layer` ruleset.

As of the time of the dataset for this year, layers occurred on a very small number of sites in the

325. <https://wpt.fyi/interop-2022?feature=interop-2022-cascade>

326. <https://wpt.fyi/results/css/css-cascade?label=experimental&label=master&product=chrome&product=firefox&product=safari&aligned&view=interop&q=label%3Ainterop-2022-cascade>

wild.

The largest number of layers defined was 6. Future editions of the Web Almanac will continue to track adoption and trends on Cascade Layers. Hopefully aligned work, close releases and early focus on good interoperability help it reach its potential and reduce any frustrations.

Given that it's shipping everywhere, now would be a great time to learn more about how Cascade Layers can help you wrangle control of your CSS.

Conclusion

Interoperability is the goal of standards, and ultimately key to large scale adoption. However, in practice, reaching interoperability is the culmination of complex independent work, focus, budgeting and priorities. Historically this has occasionally been challenging with gaps of sometimes many years between implementations landing and then various incompatibilities. Browser vendors have heard this feedback and begun to put efforts toward increased focus on coordinated efforts to close existing gaps and to tighten the timeframe that it takes for new implementations to arrive with a very high degree of interoperability.

We hope that this review of the work that has been done this year serves to inform developers and prompt adoption of and attention to these features. We will continue to track those metrics that we can and look toward how we can use data to inform our sense of how we're doing and influence where and how we're focusing.

Author



Brian Kardell

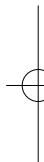
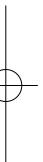
Twitter: [@briankardell](https://twitter.com/briankardell) GitHub: [bkardell](https://github.com/bkardell) Website: <https://bkardell.com>

Brian Kardell is a developer advocate and W3C Advisory Committee Representative at Igalia³²⁷, a standards contributor, blogger³²⁸. He was a founder of the Extensible Web Community Group and co-author of The Extensible Web Manifesto³²⁹.

327. <https://igalia.com>

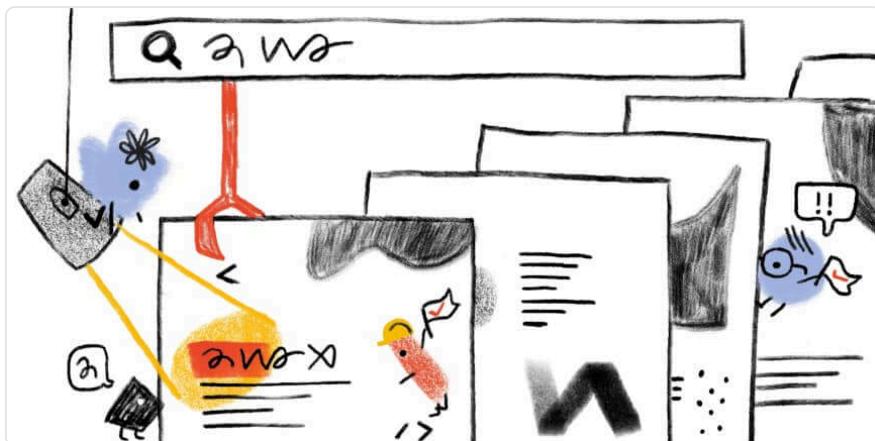
328. <https://bkardell.com>

329. <https://extensiblewebmanifesto.org>



Part II Chapter 10

SEO



Written by Sophie Brannon, Itamar Blauer, and Mordy Oberstein

Reviewed by Patrick Stox, Tushar Pol, Mobeen Ali, Dave Smart, and John Murch

Analyzed by Colt Sliva, JR Oakes, and Derek Perkins

Edited by Michael Lewittes

Introduction

Search engine optimization (SEO) is a digital technique used to improve a website or page's visibility so that it organically ranks higher in search engine results. It often combines technical configuration, content creation, and link acquisition, with the goal of improving relevance for a searcher's query and intent. SEO has continued to grow in popularity and become one of the most popular digital marketing channels.

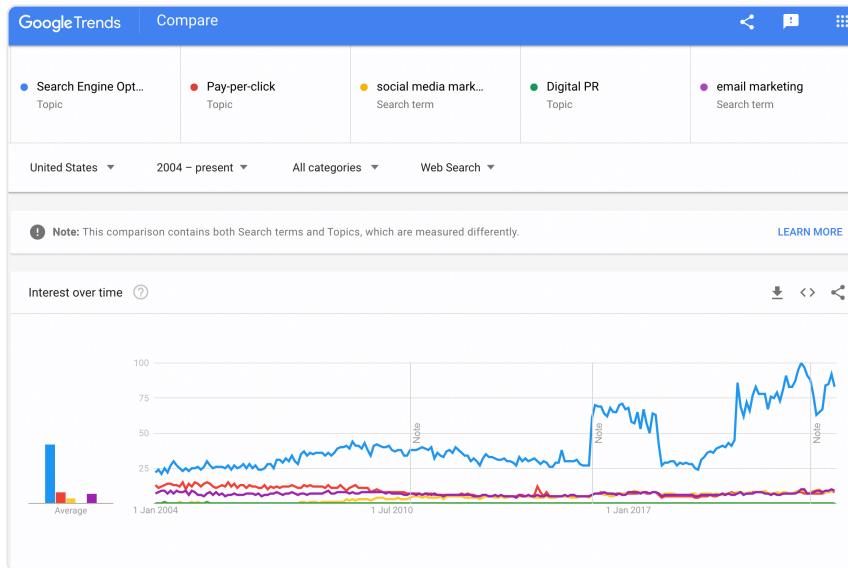


Figure 10.1. Google Trends comparing directional search popularity of topics of SEO versus pay-per-click, social media marketing, and email marketing.

With custom metrics that expose some new, never-before-seen insights, we have analyzed more than eight million homepages across the web, comparing our findings to those from 2021³³⁰ and, in some instances, from 2020³³¹. Note: Our data, particularly from Lighthouse and the HTTP Archive, is limited to just websites' homepages, not site wide crawls. Learn more about these limitations in our Methodology.

Read on for more about how search engine-friendly the web is.

Crawlability and indexability

Crawling and Indexing are the backbone of what Google and other search engines ultimately display on their search results pages. Without them, ranking simply cannot happen.

The first step in the process is discovering web pages via crawling. While numerous pages are crawled, fewer of them are actually indexed, which is essentially stored and categorized in a search engine's database. Based on a searcher's query, matching indexed pages are then served.

This section deals with the state of the web, as it pertains to bots crawling and indexing

330. <https://almanac.httparchive.org/en/2021/seo>

331. <https://almanac.httparchive.org/en/2020/seo>

websites. What directives are sites giving search engines bots? What are sites doing to ensure Google serves the right page and not a near duplicate in search results?

Let's explore the web and some of its facets that impact crawlability and indexability.

Robots.txt

The robots.txt file instructs bots, including search engine crawlers, as to where they can and cannot go, meaning what they can or cannot crawl.

Robots.txt status codes

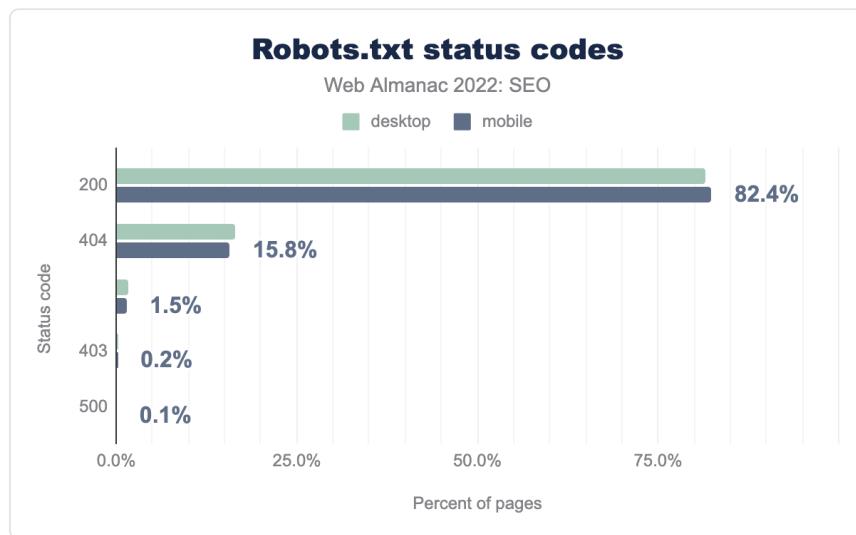


Figure 10.2. Robots.txt status codes.

There has been a nominal increase in the percentage of sites whose robots.txt files return a 200 status code in 2022 compared to 2021. In 2022, 81.5% of robots.txt files for desktop sites returned a 200 status code while 82.4% of mobile sites returned the same. This stands in comparison to 81% and 81.9% of robots.txt files on desktop and mobile sites, respectively, returning a 200 status code in 2021.

Concurrently, there was just a small reduction in the percentage of robots.txt files returning a 404 status code in 2022 compared to 2021. Last year, 17.3% of robots.txt files for desktop sites returned a 404 while 16.5% of mobile sites' robots.txt files returned that status code. In 2022, it's just 16.5% for desktop and 15.8% for mobile sites' robots.txt files that are returning a 404

status code.

Like in 2021, the remaining status codes are associated with a minimal number of robots.txt files.

Note: The above data does not indicate how well optimized a robots.txt file is. Even a file returning a 200 status code can contain directives that are perhaps not in the best interest of a site's overall health.

Robots.txt size

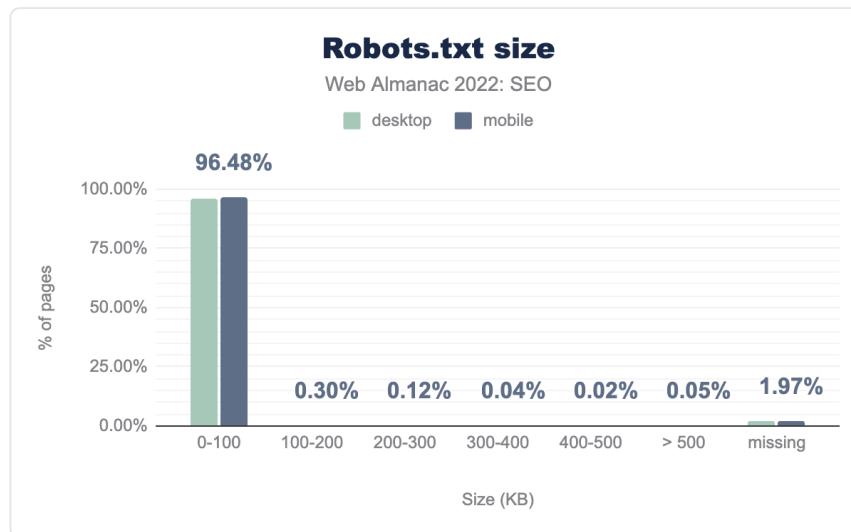


Figure 10.3. Robots.txt size codes.

As expected, the overwhelming majority of robots.txt files were quite small, weighing between 0-100 KB.

Google's max limit for a robots.txt file is 500 KiB. Any directives found after the file reaches that limit are ignored by the search engine. A very small number of robots.txt files fall into that category. Specifically, just .005% of both desktop and mobile sites contain a robots.txt file that is above Google's max limit (which is consistent with 2021's data). In cases where the file size exceeds limits, Google recommends³³² consolidating directives.

³³². https://developers.google.com/search/docs/advanced/robots/robots_txt

Robots.txt user-agent usage

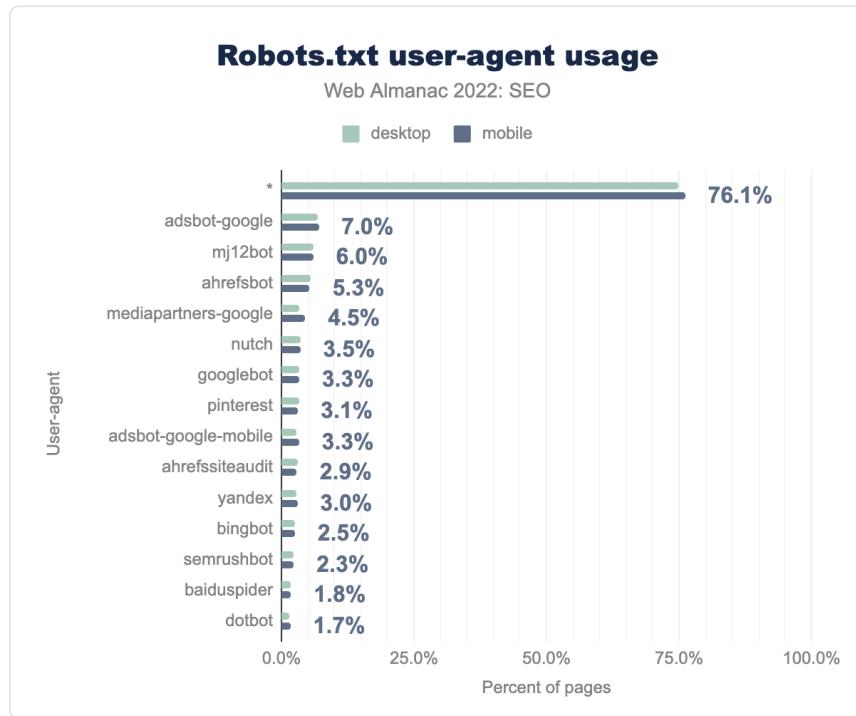


Figure 10.4. Robots.txt user-agent usage.

Most websites today (74.9% desktop and 76.1% mobile) do not indicate a specific user-agent within the robots.txt file, meaning the directives in the file apply to all user-agents. This is consistent with the data from 2020 when 74% of desktop robots.txt files and 75.2% of mobile robots.txt files did not specify a particular user-agent.

Interestingly, Bingbot did not fall into the top 10 most specified user-agents. As for SEO tools, much like in 2021, both Majestic's and Ahrefs' bots were in the top 5 most specified user-agents, while Semrush's bot rounded out the top 15 most specific user-agents.

In terms of search engines, Googlebot leads the pack with 3.3% of robots.txt files specifying the user-agent while Bingbot comes in at 2.5%. Interestingly, there was nearly a full percentage point difference in 2021 between mobile site robots.txt files and desktop files specifying Bingbot. Such is not the case in 2022 where the data is essentially uniform.

Of note, Yandexbot was specified in just 0.5% of robots.txt files in 2021. By 2022, there was a six-fold increase, with 3% of files specifying Yandexbot.

IndexIfEmbedded tag

In January 2022, Google introduced a new robots tag called `indexifembedded`. The tag offers control over indexation when content is embedded in an iframe on a page, even when a `noindex` tag has been applied.

Let's start by determining the percentage of pages for which the new tag is possibly applicable.

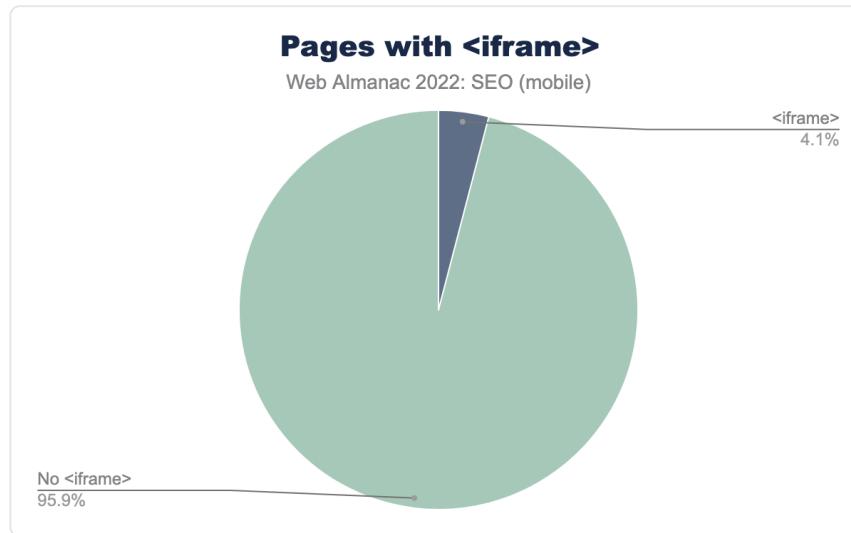


Figure 10.5. Pages with `<iframe>`.

A little more than 4% of pages contain an `<iframe>` element. Of the 4.1% of pages containing that element, 76% of them had the iframe noindexed, making them a potential use case for the new `indexifembedded` tag.

However, a minuscule percentage of sites have adopted the `indexifembedded` robots tag. The tag can be found on just 0.015% of pages surveyed.

Of the pages that have adopted the `indexifembedded` tag, 98.3% of them implemented it in the header while 66.3% are using the HTML.

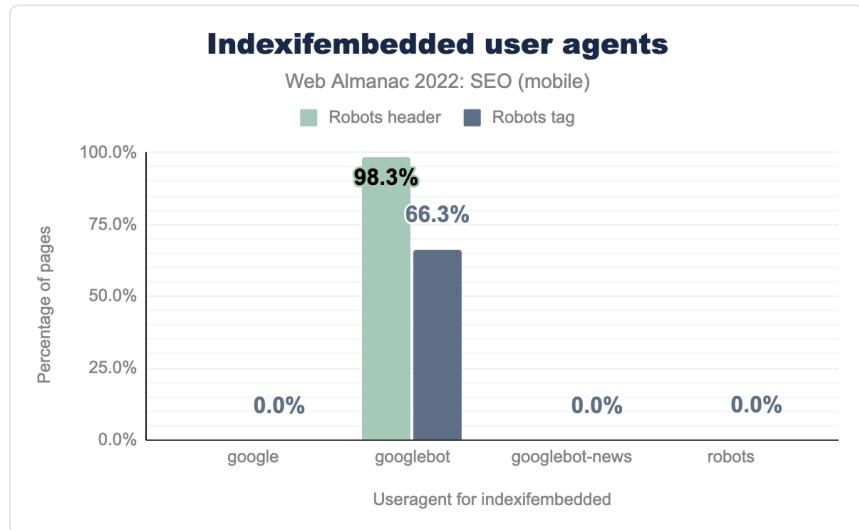


Figure 10.6. *Indexifembedded* user agents.

Invalid head elements

The `<head>` element serves as the container for a page's metadata. From an SEO point of view, a page's title tag and meta description reside within the `<head>` element, as do robots meta tags.

Not all elements, however, belong in the `<head>`. Should Google come across an invalid element in the page's `<head>`, it assumes that it has reached the end of the `<head>` and will not discover the rest of its contents³³³.

Our data from 2022 shows 12.7% of desktop pages and 12.6% of mobile pages contain an invalid element in the `<head>`.

³³³ <https://developers.google.com/search/docs/advanced/guidelines/valid-html>

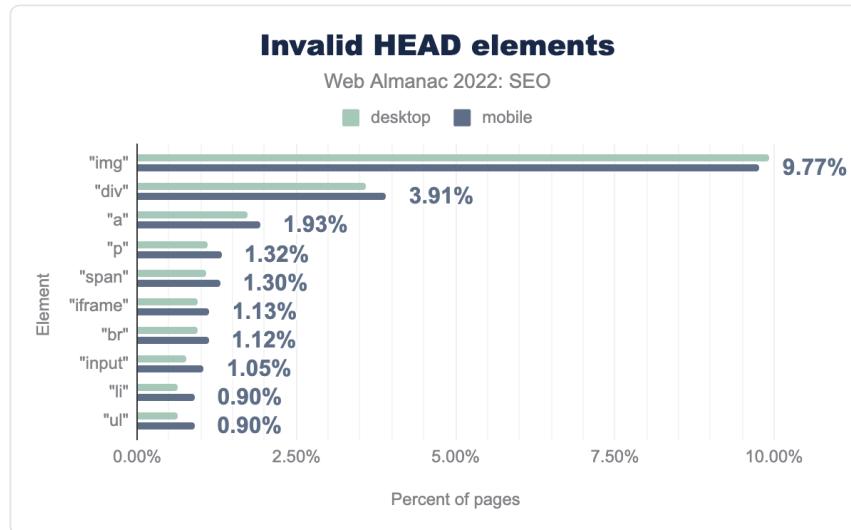


Figure 10.7. Invalid `<head>` elements.

The most misapplied element to the `<head>` by far is the `` element. It is incorrectly placed within the `<head>` on 9.7% of mobile pages and 9.9% of desktop pages.

The `<div>` element is the only other misapplied element to appear within the `<head>` on more than 3% of the pages within the 2022 dataset. It is incorrectly applied to the `<head>` on 3.5% of desktop pages and 3.9% of mobile pages.

Canonical tags

Canonical tags are traditionally used when defining duplicate content pages and to help search engines prioritize. They are a snippet of HTML code (`rel="canonical"`) that allows webmasters to define to the search engine which page is the “preferred” version. They are not directives, and instead act as a “hint.” Therefore, search engines such as Google determine their own canonical version of the page, based on how useful they believe the page is for the user. Canonical tags can also be used to consolidate other signals such as links, as well as to simplify tracking metrics and better manage syndicated content.

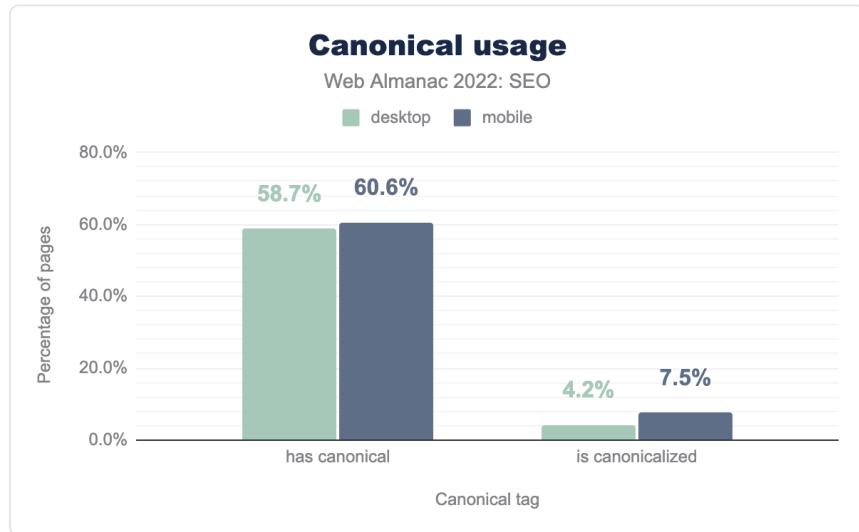


Figure 10.8. Canonical usage.

We see from the data that canonical tags usage has increased over the years. In 2019, 48.3% of mobile pages used canonicals. In 2020, this grew to 53.6%. In 2021, this grew even further to 58.5%. And in 2022, it's increased to 60.6%.

Mobile has a higher percentage of canonical attribution than desktop (60.6% vs. 58.7%), which is likely a direct result of single use URLs on mobile. Since the data set in this chapter is limited to home pages, it's fair to assume that this is the reason for the higher canonical attribution on mobile. According to Google's guidelines³³⁴, having a separate mobile site is not recommended.

HTML vs. HTTP canonical usage

There are two ways of implementing canonical tags:

1. Within the HTML `<head>`
2. Within the HTTP headers (`Link` HTTP header)

³³⁴. <https://developers.google.com/search/mobile-sites/mobile-seo/separate-urls>

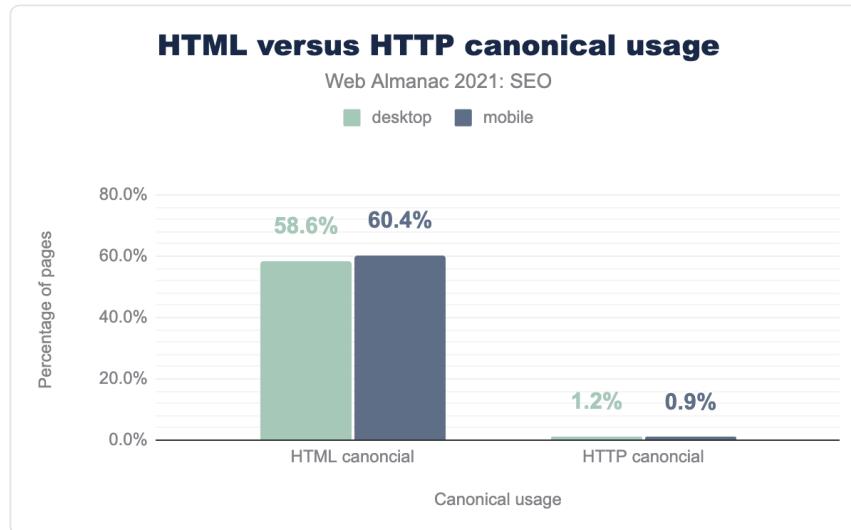


Figure 10.9. HTML versus HTTP Canonical usage.

The most common usage across both desktop and mobile is through HTML at 58.6% and 60.4%, respectively. This is likely due to the ease of implementation. While one requires basic HTML knowledge, the other method (through HTTP headers) requires a more technical skillset.

Raw vs. rendered usage

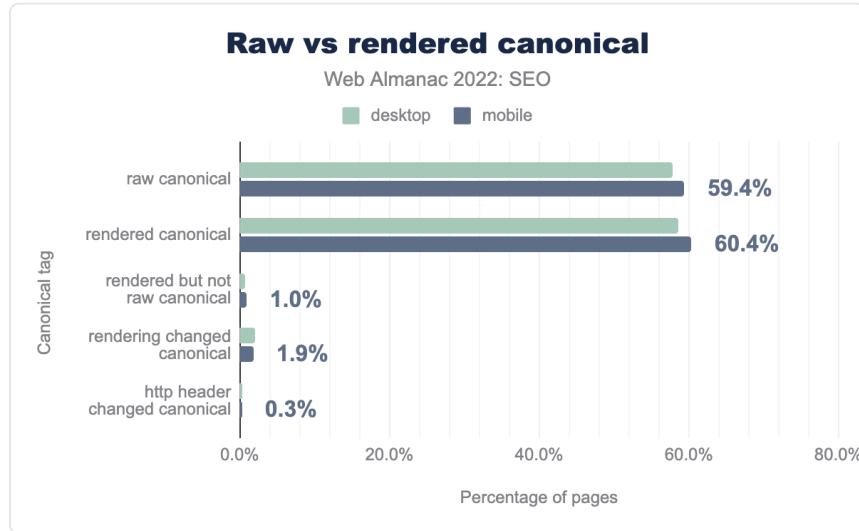


Figure 10.10. Raw vs rendered canonical.

Compared to 2021, where raw canonical usage was 57.7% and rendered canonical usage was 58.4%, in 2022 there was some growth, with raw canonical usage reaching 59.4% and rendered canonical usage rising to 60.4%. This correlates with the growth in overall canonical use.

Page experience

In this section of the chapter, we're looking at different elements of page experience and how this has evolved since the 2021 Web Almanac.

HTTPS

In 2021, there was an increased focus on site speed and page experience following Google's introduction of the Core Web Vitals update, which had been publicized and pushed throughout 2020. While evidence of HTTPS being a ranking factor dates back to 2014³³⁵, the overall focus on page experience since the Core Web Vitals announcement likely had an impact on the adoption of HTTPS across the web.

³³⁵. <https://developers.google.com/search/blog/2014/08/https-as-ranking-signal>

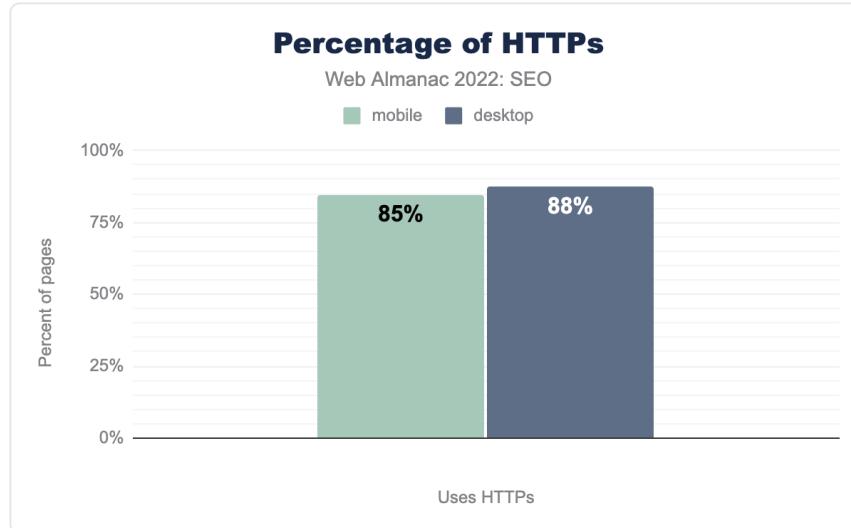


Figure 10.11. Percentage of websites supporting HTTPS.

We see from the data how more sites are using a secure certificate (HTTPS) at the time of the crawl (taking into account expirations of these certificates). In 2021, 84.3% of desktop pages used HTTPS, and it went up to 87.71% in 2022. On mobile, this increased from 81.2% in 2021 to 84.75% in 2022. Since the announcement of the Core Web Vitals update in 2020 to the present there's been an increase of nearly 11% on mobile and 10% on desktop.

Mobile friendliness

Mobile-friendliness can be determined by looking at responsive design implementation vs. dynamic serving. To identify this, we looked at the use of the viewport meta tag which is commonly used in responsive design vs. the vary: User-Agent header to determine if a website is using dynamic serving.

Viewport meta tag

92%

Figure 10.12. Sites supporting a viewport meta tag.

We have seen the use of the viewport meta tag grow from 91.1% of mobile pages using

viewport meta tag in 2021 to now 92%. In 2020, it was at 89.2%.

Vary header usage

The vary header is a HTTP header that enables different content to be served to different users on different devices. This is known as dynamic serving, and is the opposite to responsive design, which serves the exact same content, but to different devices.

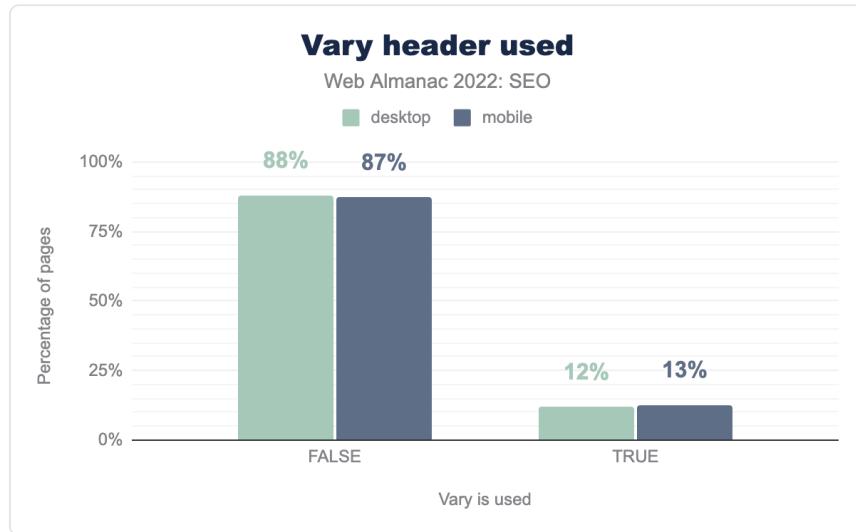


Figure 10.13. Vary header used.

Vary header usage has remained relatively unchanged for the past few years. In 2021, 12.6% of desktop and 13.4% of mobile pages used this footprint. In 2022, the data is nearly identical, with 12% for desktop and 13% for mobile.

Legible font sizes

In 2021, 13.5% of mobile pages were not using a legible font size. Thanks to Google's focus on user experience across all devices, more pages than ever now use a legible font size. Only 11% of mobile pages are still not using a legible font size.



Figure 10.14. Sites not using a legible font size.

Core Web Vitals (CWV)

Core Web Vitals was a hot topic in SEO throughout 2021 following Google announcing the roll out of its Page Experience update that June. We have seen a continued interest this year, with more sites paying attention to their CWV performance.

Core Web Vitals are a series of standardized metrics that can help developers and SEOs to better understand how a user is experiencing a page. The main metrics are:

- Largest Contentful Paint (LCP) measures how quickly a web page's main content loads
- First Input Delay (FID) measures how long it takes from when a user interacts with a web page (i.e. clicks a button) to when the browser is able to respond
- Cumulative Layout Shift (CLS) measures the visual stability and whether a page shifts within the viewport

All three of these metrics are critical to user experience and the stability of a web page.

The data for Core Web Vitals is sourced from the Chrome User Experience Report (CrUX). The report comes from a public dataset of real (opted-in) users, and is sourced from millions of websites (as opposed to lab data, which is simulated).

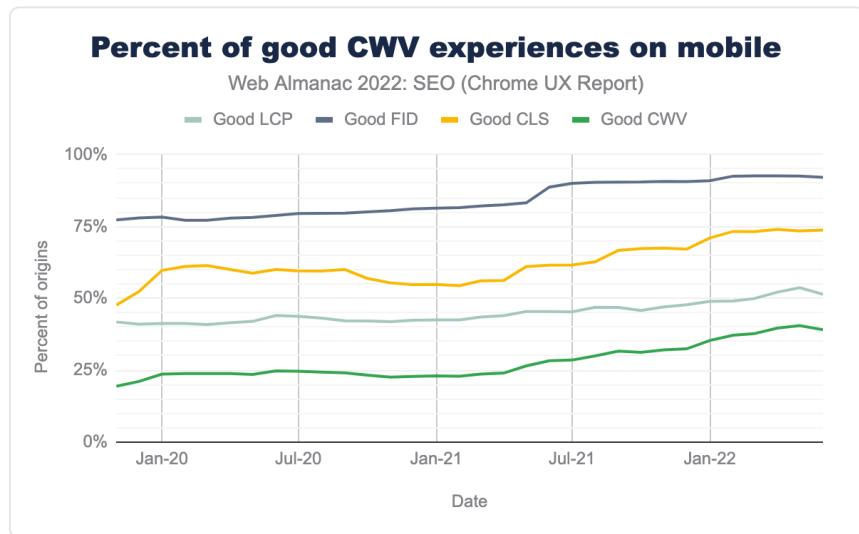


Figure 10.15. Percent of good CWV experiences on mobile.

On mobile, 39% of sites now pass CWV, which is up from 29% in 2021 and just 20% in 2020. And while 92% of sites currently pass FID, most site owners are struggling with LCP, which has a pass rate of 51%.

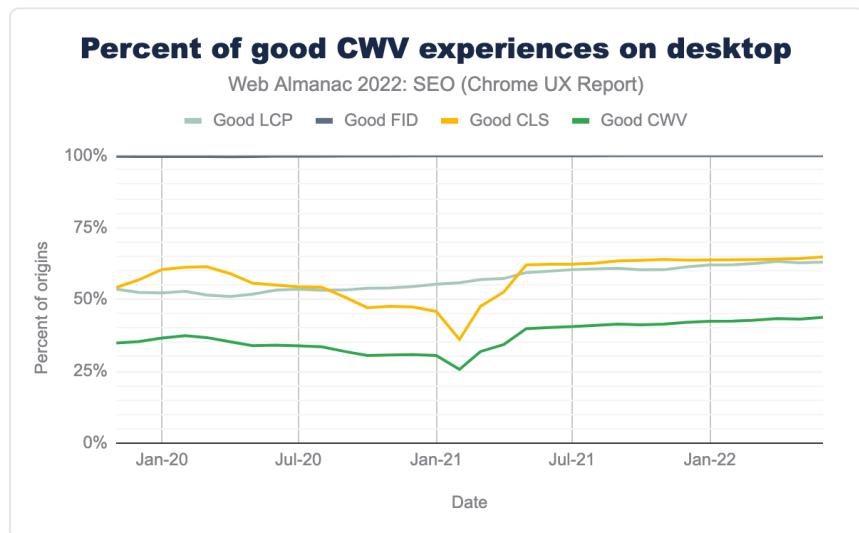


Figure 10.16. Percent of good CWV experiences on desktop.

On desktop, we see an astounding 100% of sites passing FID, though similarly struggling to pass LCP and CLS. Noteworthy, more sites are passing CWV on desktop (43%) than on mobile (39%).

lazy loading vs. **eager** loading iframes

Lazy loading is a technique that defers the loading of non-critical elements on a web page until the point in which they are needed. This can help with the reduction of page weight, as well as conserve bandwidth and system resources. Eager loading is when related entities are simultaneously loaded and fetched all at once.

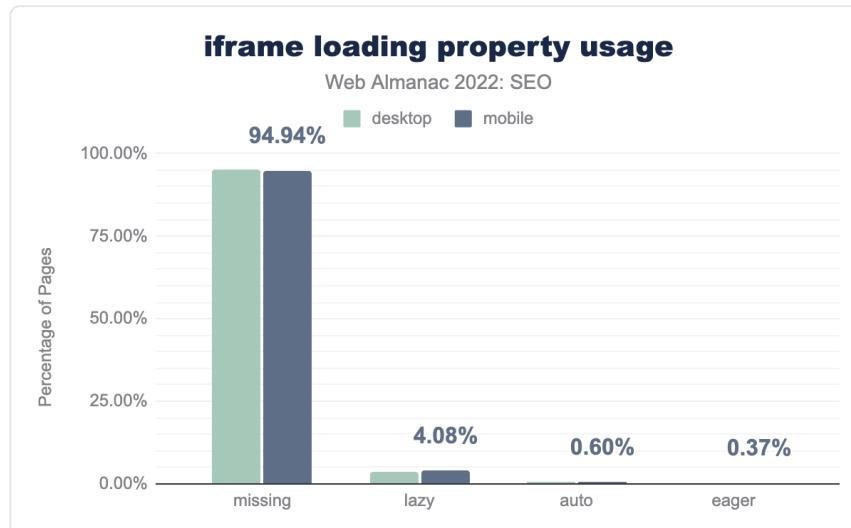


Figure 10.17. *iframe* loading property usage.

When looking solely at iframes, we see lazy loading is preferred far more to eager loading, with 4.08% of iframes being lazy loaded versus 0.37% of iframes being eager loaded.

This is particularly interesting since browser-level lazy loading for iframes has become standardized in Chrome³³⁶. The standardization of the `loading` attribute, without specifying lazy or eager, is likely why data shows 94.4% of attributes do not contain lazy or eager.

336. <https://web.dev/iframe-lazy-loading/>

On page

When looking for relevancy signals, search engines look at the content on a web page. There are various on-page SEO elements that can affect rankings and/or appearance on the SERPs (Search Engine Results Pages).

Meta data

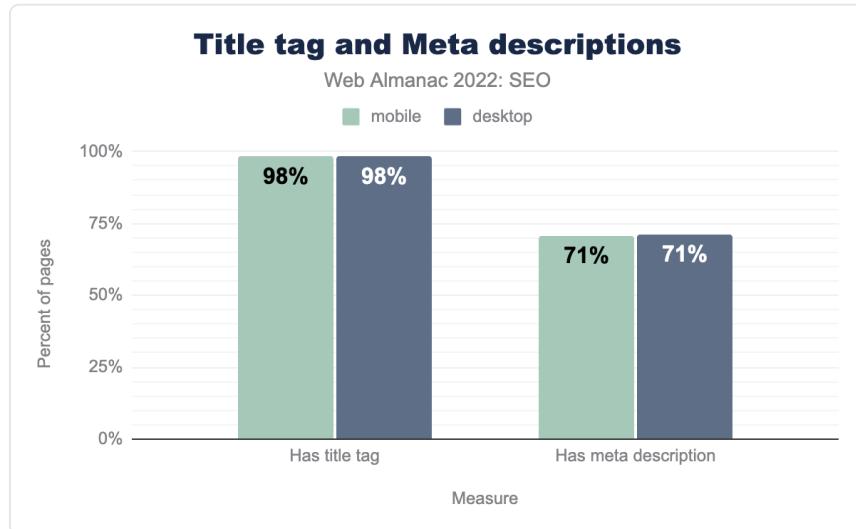


Figure 10.18. Title tag and Meta descriptions.

For the second year in a row, 98.8% of desktop and mobile pages had `<title>` elements. Also in 2022, 71% of desktop and mobile homepages had `<meta name="description">` tags, a 0.1% decrease from last year.

`<title>` element

The `<title>` element is an on-page ranking factor that provides a strong hint regarding page relevance and may appear on the SERP. In August 2021, Google started rewriting more websites' titles in their search results³³⁷.

³³⁷. <https://developers.google.com/search/blog/2021/08/update-to-generating-page-titles>

Title words by percentile

Web Almanac 2022: SEO

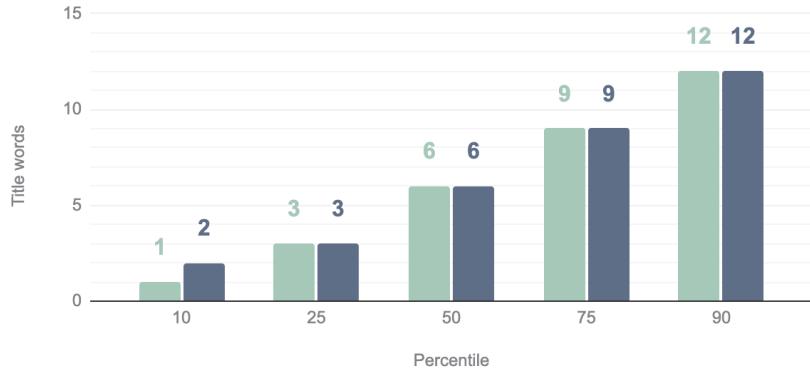
desktop
 mobile


Figure 10.19. Title words by percentile.

Title characters by percentile

Web Almanac 2022: SEO

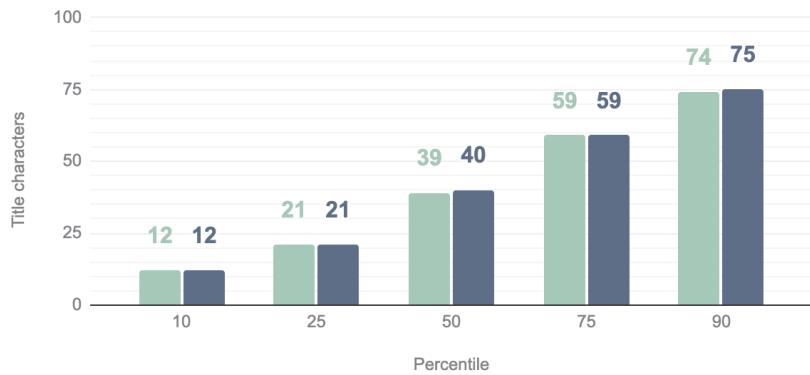
desktop
 mobile


Figure 10.20. Title characters by percentile.

In 2022:

- The median page <title> contained 6 words.
- The median page <title> contained 39 and 40 characters on desktop and mobile,

respectively.

- 10% of pages had `<title>` elements containing 12 words.
- 10% of desktop and mobile pages had `<title>` elements containing 74 and 75 characters, respectively.

These stats remain unchanged from last year. Note: These titles on homepages tend to be shorter than those used on deeper pages.

Meta description tag

The `<meta name="description">` tag does not directly impact rankings. However, it may appear as the page description on the SERP, and it can influence click-through rate.

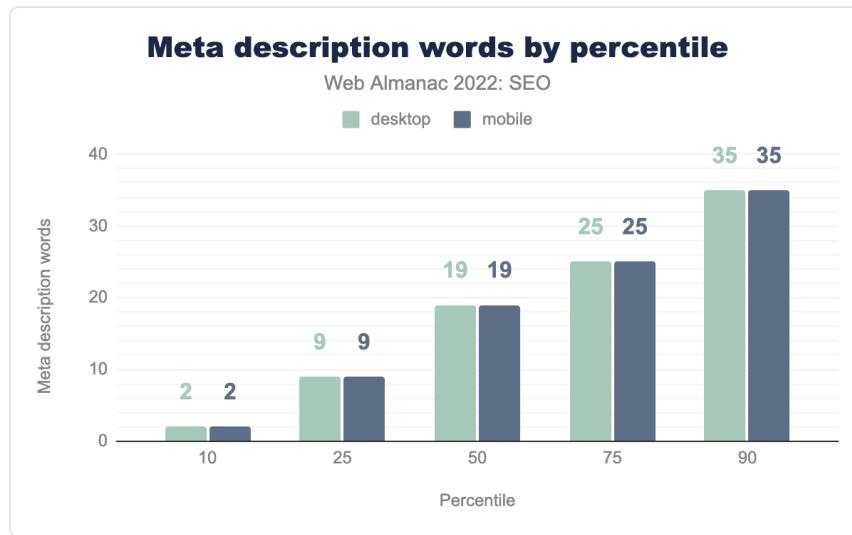


Figure 10.21. Meta description words by percentile.

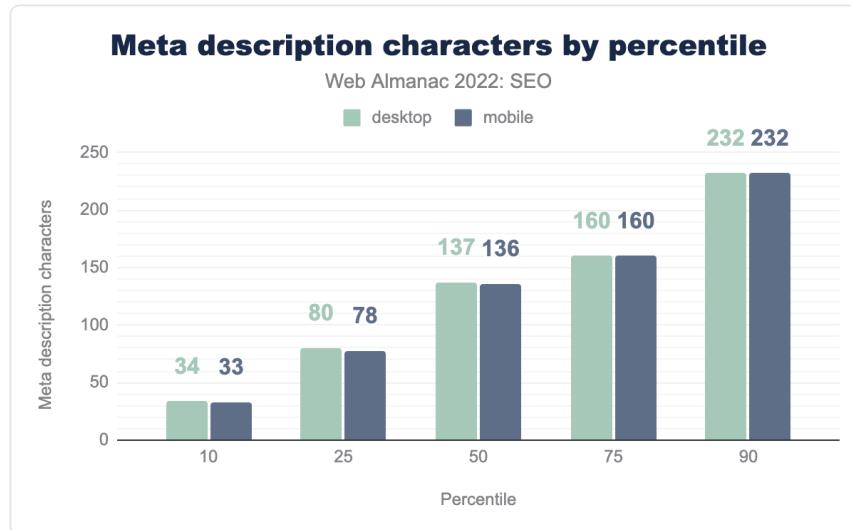


Figure 10.22. Meta description characters by percentile.

In 2022:

- The median desktop and mobile page `<meta name="description">` tag contained 19 words.
- The median desktop and mobile page `<meta name="description">` tag contained 137 and 136 characters, respectively.
- 10% of desktop and mobile pages had `<meta name="description">` tags containing 35 words.
- 10% of desktop and mobile pages had `<meta name="description">` tags containing 232 characters.

For the most part, these stats are relatively unchanged from last year.

Header tags

Heading elements (`<h1>`, `<h2>` ...) are important parts of a page's structure since they help organize the content on the page. Heading elements are not a direct ranking factor, but they can help Google better understand the content found on the page.

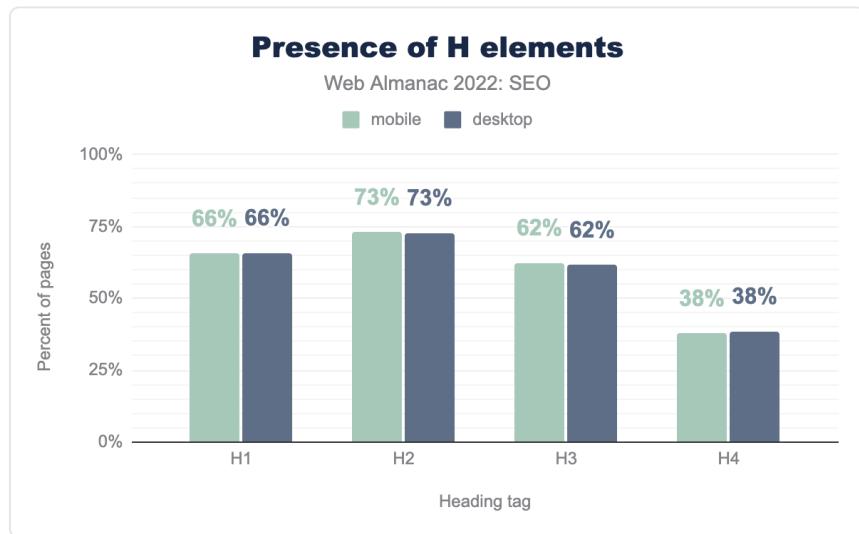


Figure 10.23. Presence of H elements.

The trends around implementation of headings by type in 2022 closely match those from 2021, with just a few small differences. For example, 71.9% of mobile pages utilized an h2 in 2021 while 73.02% did in 2022.

Another trend that has carried over is the discrepancy in usage between the h1 and h2. While 72.7% of desktop pages implement an h2, only 65.8% use an h1 (with similar numbers reflected on mobile).

Although there is no definitive explanation for this, one possible reason is that the h1 is often placed above any content. It's not essential for the natural flow of the content. However, without the h2, there could be a long flow of unstructured content.

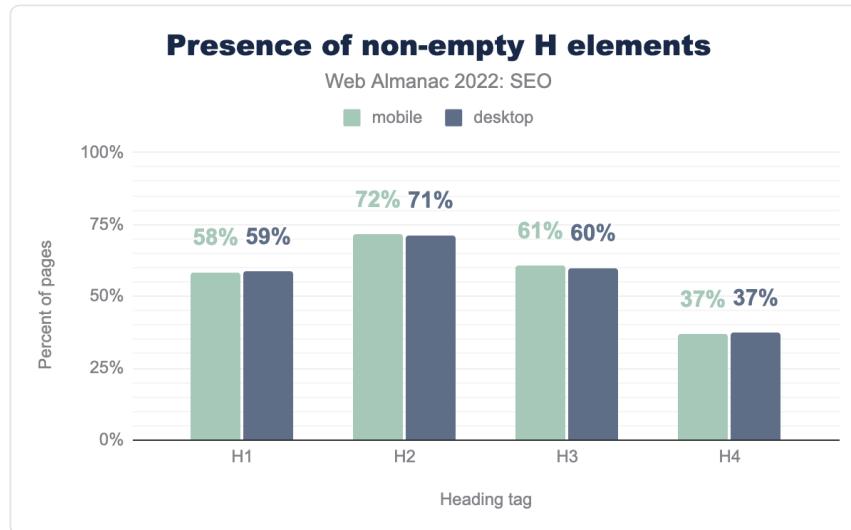


Figure 10.24. Presence of non-empty H elements.

Overall, much like 2021's stats, there are relatively few empty H elements found on pages. Additionally, there is little discrepancy between the desktop and mobile data.

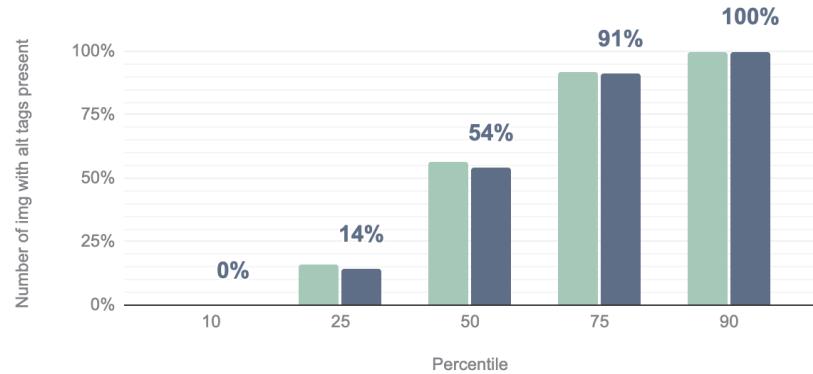
There is divergence, however, with the h1. While 65.8% of pages contained an h1 element, 58.5% contained a non-empty h1 element. That's a 7.3 percentage point difference. Contrast that with the h2, which has just a 1.5 percentage point difference. As noted in the 2021 Web Almanac, this may be a result of the many websites that wrap logo-images in the h1 element on homepages.

Image attributes

The primary purpose of the alt attribute on the `` element is accessibility. Alt attributes also assist search engines rank specific assets in image search.

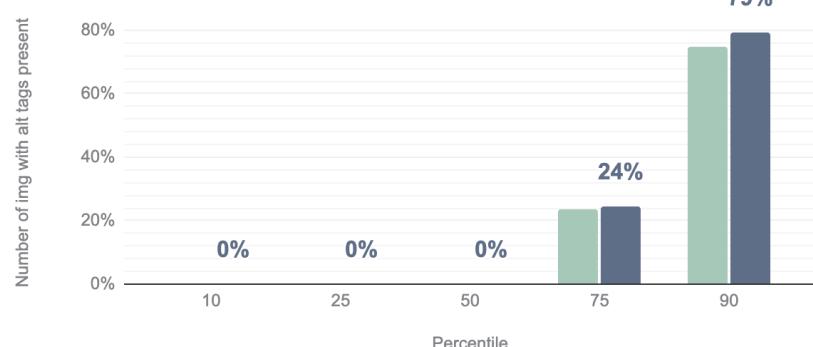
Percentage of img alt attributes present

Web Almanac 2022: SEO

desktop mobileFigure 10.25. Percentage of `img alt` attributes present.

Percentage of img with blank alt

Web Almanac 2022: SEO

desktop mobileFigure 10.26. Percentage of `img alt` with blank `alt`.

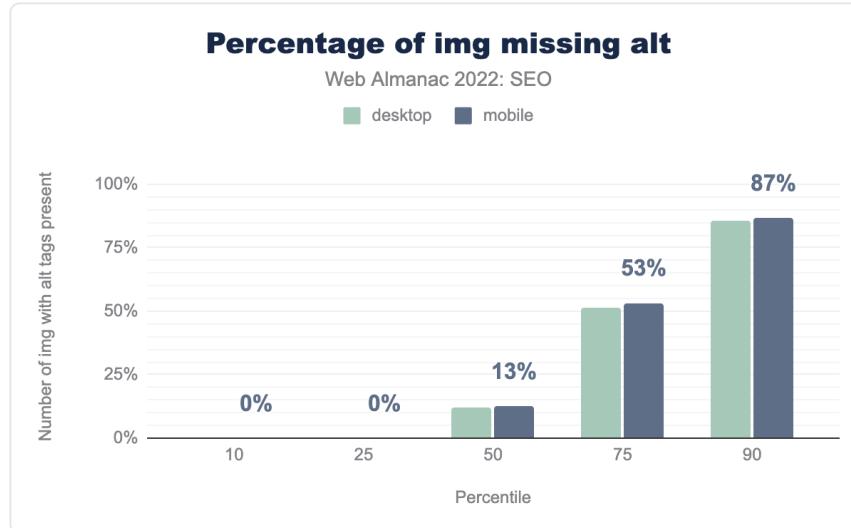


Figure 10.27. Percentage of `img` missing `alt`.

What we found:

- On the median desktop page, 56.25% of `` tags have the alt attribute. This is a negligible decrease of just a quarter of a percentage point from 2021's 56.5%.
- On the median mobile page, 54.9% of `` tags have the alt attribute. This is a marginal increase from the 54.6% of tags with the alt attribute in 2021.
- There is a noticeable change in the median desktop and mobile pages containing `` tags that have blank alt attributes compared to 2021. Last year, the median desktop and mobile pages, respectively had 10.5% and 11.8% `` tags with blank `alt` attributes. In 2022, this figure rose to 12.1% and 12.5% on desktop and mobile, respectively.
- The trend towards 0% of median desktop and mobile pages containing `` tags missing the alt attribute continues. On the median desktop page in 2021, 1.4% of the `` tags had blank attributes. It fell to 0% in 2022.

Image loading property usage

How user agents prioritize the rendering and displaying of images is affected by the loading attribute applied to `` elements. This implementation can impact user experience and

performance time, with possible effects on both SEO success and conversions.

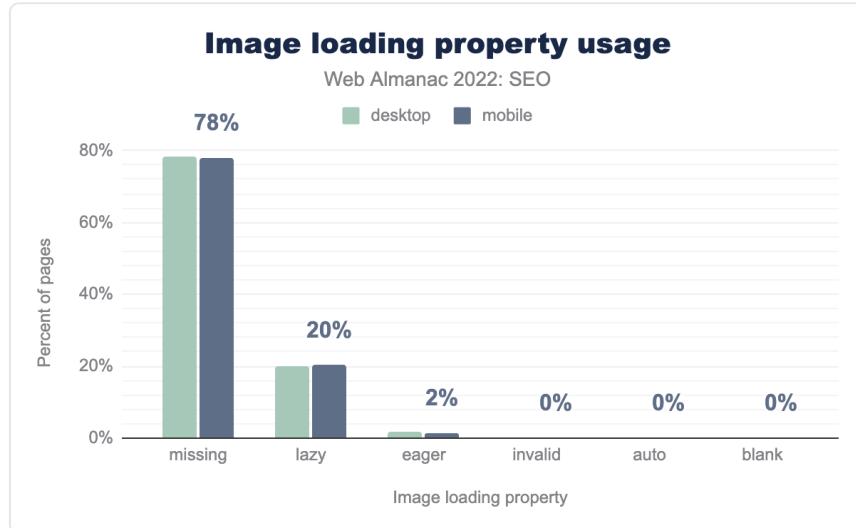


Figure 10.28. Image `loading` property usage.

What we found:

- There has been a significant reduction in the number of pages that do not use any image loading property. In 2021, 83.3% of desktop pages and 83.5% of mobile pages didn't utilize any image loading property at all. It's now 78.3% of desktop pages and 77.9% of mobile pages in 2022.
- Conversely, the implementation of `loading="lazy"` has increased. In 2021, both 15.6% of desktop and mobile pages implemented `loading="lazy"`. This has increased to 19.8% (desktop) and 20.3% (mobile) in 2022.
- The number of pages defaulting to the browser's loading method has fallen in 2022. On desktop, .07% of pages use `loading="auto"` and .08% on mobile. In 2021, .01% of pages utilized `loading="auto"`.

Word count

While content length is not a ranking factor, it is still valuable to assess how many words a page contains on average.

Rendereed word count

Let's begin with the number of words found on the page once it has been rendered.

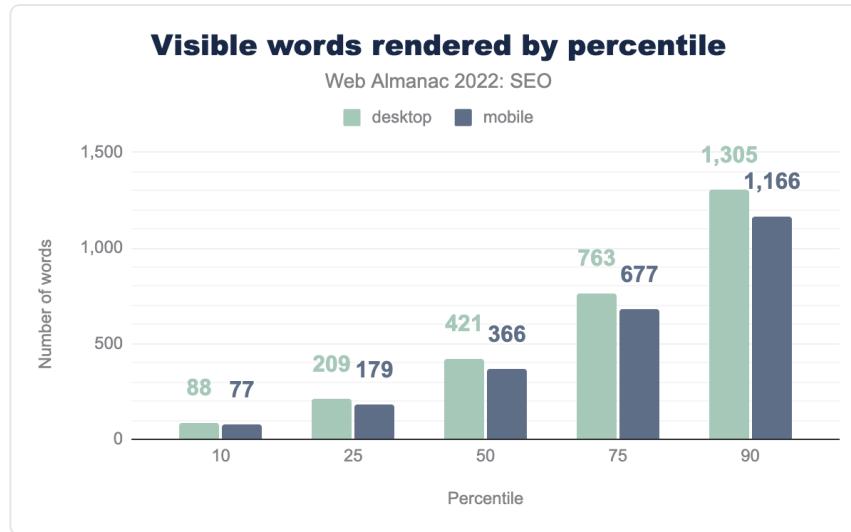


Figure 10.29. Visible words rendered by percentile.

The median desktop page in 2022 contains 421 words. This is quite close to the 425 words found in 2021. However, this is still a big leap percentage-wise from what we found back in 2020 when 402 words were found on the median desktop page. Whatever the cause was in 2021 for the uptick in rendered word count, it appears to have remained through 2022.

Similarly, the median number of rendered words on mobile in 2022 contains 366 words, which is also similar percentage-wise to the data in 2021. For context, desktop pages contain more words than mobile pages. The median desktop page contains 15% more words than mobile pages within the 50th percentile. This is significant since Google some years ago adopted a mobile-first index, and content not found on the mobile version of a page runs the risk of not being indexed by the search engine.

Raw word count

Let's now examine the number of words contained in a page's source code prior to the browser executing any JavaScript code or other modifications in the DOM or CSSOM.

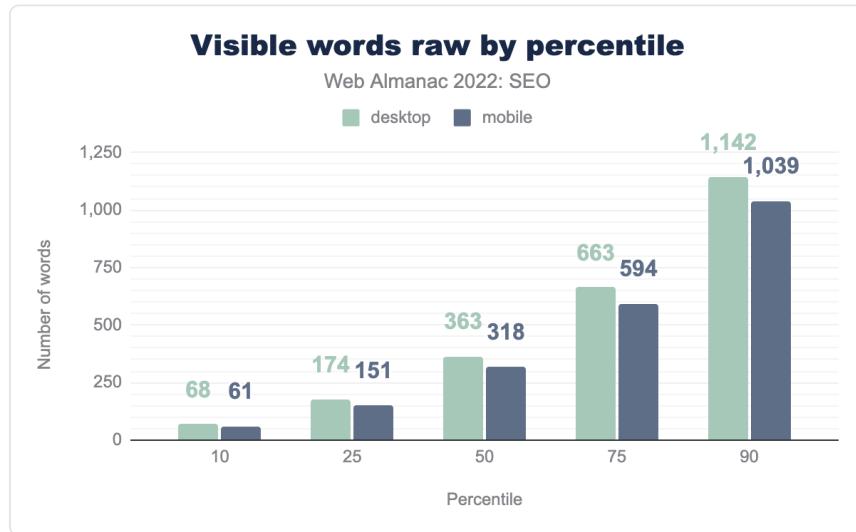


Figure 10.30. Visible words raw by percentile.

Much like the rendered word count, there is a minimal difference between the data in 2022 versus what was found in 2021. For example, the median desktop page's raw word count was 369 in 2021 compared to 363 in 2022 and median mobile page's raw word count was 318 which is slightly less than 2021 which saw 321 words as the median.

Here, too, mobile pages contain fewer words than desktop pages across the board. The median mobile page contains a raw word count that is 12.39% less than desktop. As noted above, this is significant because of Google's mobile-first indexing.

Structured Data

Implementing Structured Data has come into increased focus as rich results on the Google SERP have become more prominent.

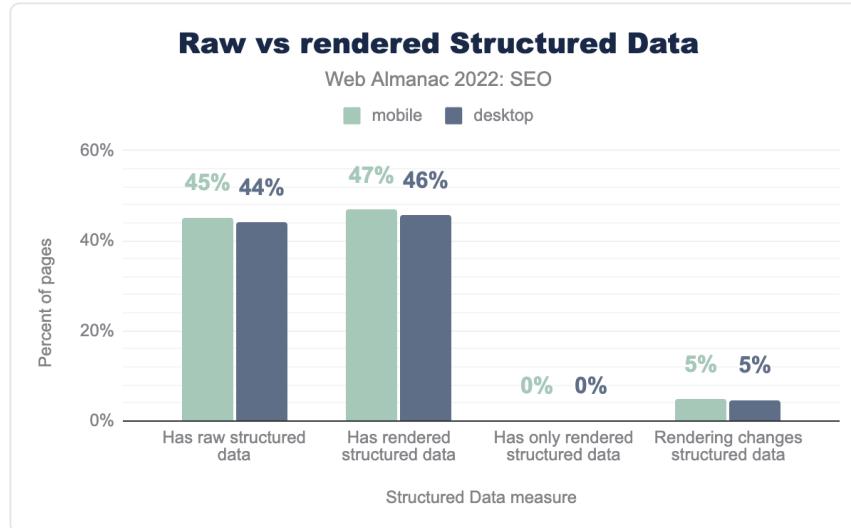


Figure 10.31. Raw versus rendered structured data.

The implementation of structured data in the HTML of a page has continually increased. In 2021, 42% of desktop pages and 43% of mobile pages used structured data. In 2022, it's risen to 44% of desktop pages and 45% of mobile pages that have structured data within their HTML.

This reflects 2 percentage point increases on both desktop and mobile pages. Two possible explanations for greater adoption could be that a number of Content Management Systems have added automatic structured data markup to their pages, as well as the aforementioned prominence that structured data has played in Google SERPs.

There has also been a great reduction in both mobile and desktop pages that have structured data added via JavaScript where it was not contained within the initial HTML response. In 2021, 1.7% of mobile pages and 1.4% of desktop pages had structured data added via JavaScript where it was not contained within the initial HTML response. It's now just .15% on desktop and .13% on mobile.

Most popular Structured Data formats

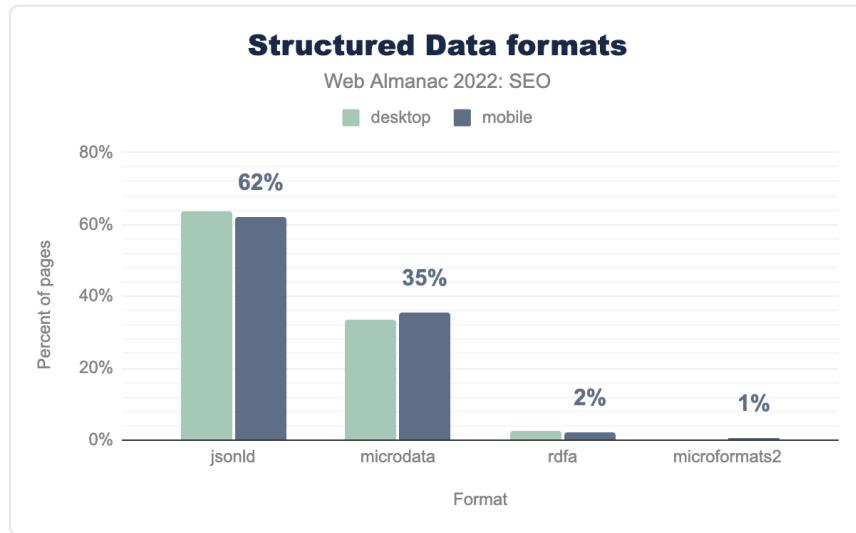


Figure 10.32. Structured Data formats.

Structured data can be implemented through various ways on a given page. However, JSON-LD, which aligns with Google's own recommendation for implementation, is by far the most popular format.

Compared to 2021's figures, 2022's data shows a nominal increase in implementation via JSON-LD and a slight decrease when implementing structured data with microdata. These numbers bear out in particular on mobile. In 2021, 60.5% of mobile pages used JSON-LD to implement structured data. The number of mobile pages in 2022 using JSON-LD for adding structured data is up 2.3% to 61.9%. Conversely, 36.9% of mobile pages in 2021 utilized structured data with microdata. That number fell 4.3% in 2022 to 35.3%.

Most popular schema types

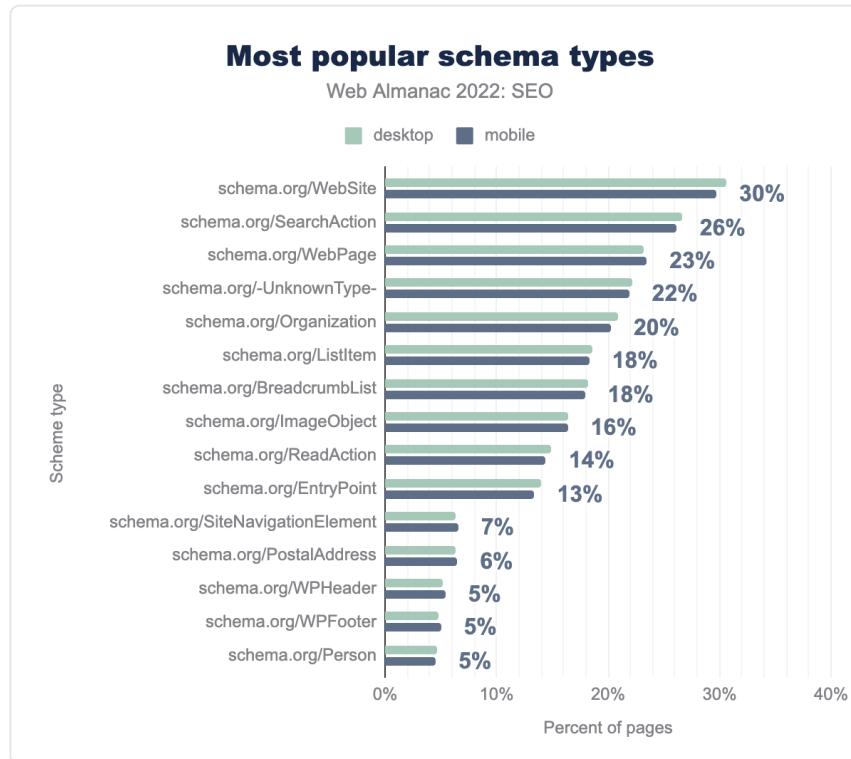


Figure 10.33. Most popular schema types.

There is strong correlation between the most popular types of schema found on homepages in 2021 and 2022.

As noted in previous editions of the Web Almanac, `WebSite`, `SearchAction`, `WebPage`, `SearchAction` is what powers the Sitelinks Search Box³³⁸ [see chart above].

When comparing 2021 to 2022, there has been a significant increase in the adoption of the most popular schemas across the board. In fact, every noted schema type has experienced an increase in adoption in 2022. Among the most notable are the schema for `BreadcrumbsList`, which has risen 22.8% since 2021 and `ImageObject`, which is up 12.3%.

In terms of implementing the most popular schemas, there are relatively tiny differences between the percentages of mobile versus desktop pages.

³³⁸. <https://developers.google.com/search/docs/advanced/structured-data/sitelinks-searchbox>

You can read more about structured data in our dedicated chapter.

Links

Search engines utilize links to discover new pages and to pass PageRank, which helps determine the importance of pages. Links also act as a reference from one page to another (presumably relevant) page.

Non-descriptive link text

Anchor text, which is the clickable text used in a link, helps search engines to understand the content of the linked page. Lighthouse has a test to check if the anchor text used is useful and/or contextual, or if it's generic and/or non-descriptive such as "learn more" or "click here." In 2022, 15% and 17% of the tested links on mobile and desktop, respectively, did not have descriptive anchor text, a missed opportunity from an SEO perspective and bad for accessibility.

Outgoing links

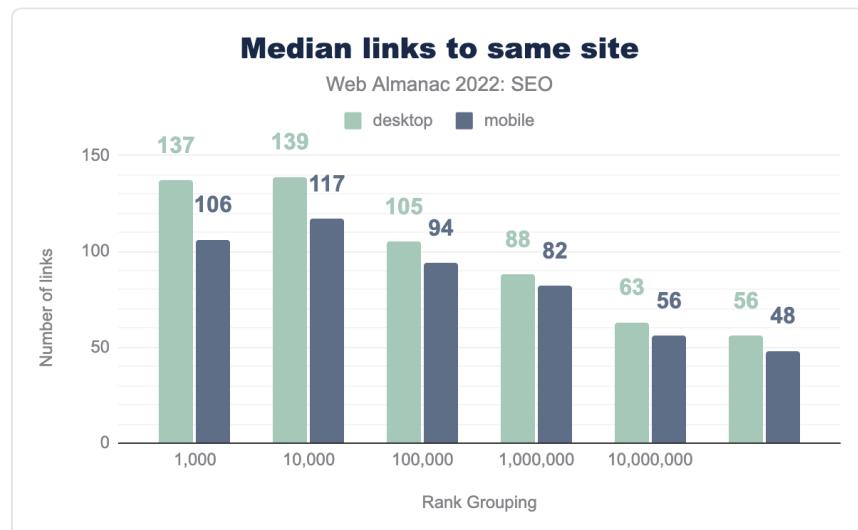


Figure 10.34. Median links to same site.

Internal links are links to other pages on the same website. Much like last year, 2022's figures

suggest pages had fewer links on their mobile versions compared to their desktop counterparts.

The median number of internal links is now 16% higher on desktop than mobile at 56% and 48%, respectively. It's likely a result of developers minimizing the navigation menus and footers on mobile for ease of use on smaller screens.

According to CrUX data, the 1,000 most popular websites have more outgoing internal links than less popular sites, a total of 137 links on desktop versus 106 on mobile. That's more than two times higher than the median. This may be attributed to the use of mega-menus on larger sites that generally have more pages.

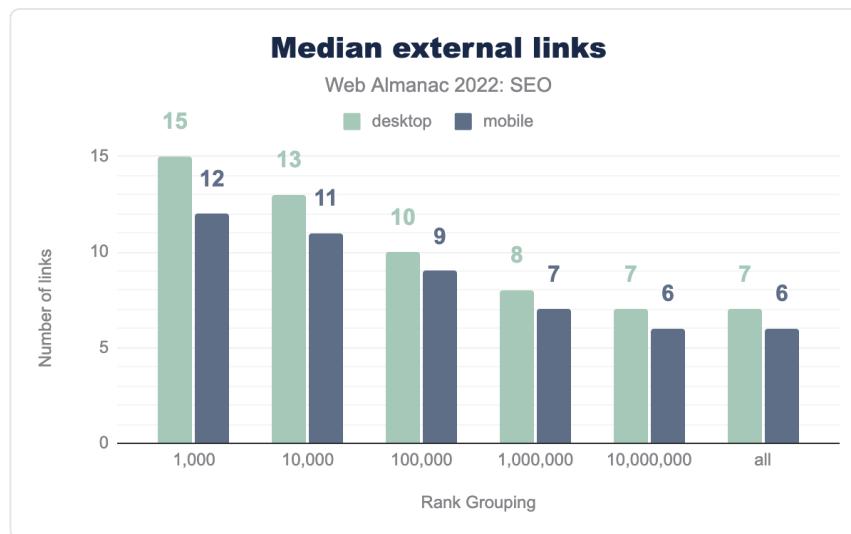


Figure 10.35. Median external links.

External links are links to other pages on a different website. The data, which has been consistent for the past few years, points to there being fewer external links on the mobile versions of pages compared to the desktop versions. Despite Google rolling out mobile-first indexing a few years ago, websites have not brought their mobile versions to parity with their desktop counterparts.

Anchor rel attribute use

In September of 2019, Google introduced attributes³³⁹ that allow publishers to classify links as

339. <https://webmasters.googleblog.com/2019/09/evolving-nofollow-new-ways-to-identify.html>

being sponsored or user-generated content. These attributes are in addition to `rel=nofollow`, which was previously introduced in 2005³⁴⁰. The newer attributes, `rel=ugc` and `rel=sponsored`, add additional information to the links.

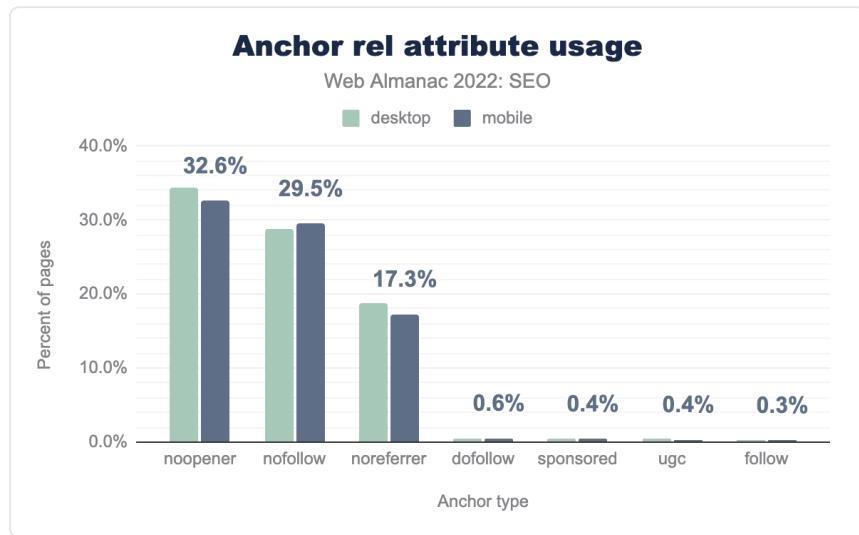


Figure 10.36. Anchor `rel` attribute usage.

Not much has changed in terms of the adoption of the newer attributes, with `rel=ugc` appearing on 0.4% of desktop and mobile pages, and `rel=sponsored` appearing on 0.5% of desktop and 0.4% of mobile pages in 2022.

`rel="dofollow"` once again appeared on more pages than `rel="ugc"` and `rel="sponsored"`. While this is technically not a problem, Google ignores `rel="follow"` and `rel="dofollow"` because, despite their inclusion, they are not actually official attributes.

`rel="nofollow"`, which is a real attribute, was found in 2022 on 29.5% of mobile pages, which is 1.2% less than last year. Google treats `nofollow` as a hint, meaning the search engine can choose whether or not they respect the attribute.

AMP

AMP has been a controversial topic since its launch in 2015, with SEOs debating whether or not

340. <https://googleblog.blogspot.com/2005/01/preventing-comment-spam.html>

it had a direct impact on rankings. Google later released this statement (below) in its documentation for additional clarification:

While AMP itself isn't a ranking factor, speed is a ranking factor for Google Search. Google Search applies the same standard to all pages, regardless of the technology used to build the page.

– Google Search Central³⁴¹

The future of AMP appears to be changing ever since the launch of Core Web Vitals. A main reason for previously implementing AMP, aside from improving page speed, was that it was necessary for inclusion in Top Carousels. In 2021, Google updated its requirements and outlined that any page is now eligible to appear in Top Carousels with or without AMP.

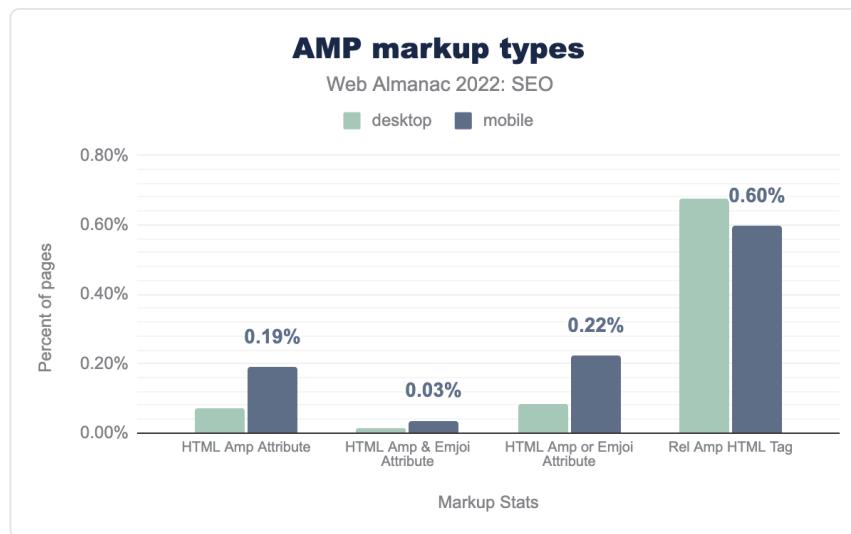


Figure 10.37. AMP markup types.

Desktop usage has dipped in 2022 from 0.09% to 0.07% compared to 2021 while mobile usage is down from 0.22% to 0.19% over the same time period.

341. <https://developers.google.com/search/docs/advanced/experience/about-amp>

Internationalization

Internationalization in SEO is the process of optimizing a website in line with best practices when targeting multiple countries and multiple languages, to ensure that it can be properly crawled and indexed by search engines.

Hreflang usage

Hreflang tags help Google and other search engines, such as Bing and Yandex, understand what the main language is on a given page. It is primarily used in international SEO campaigns when several different languages are used across different versions of a website.

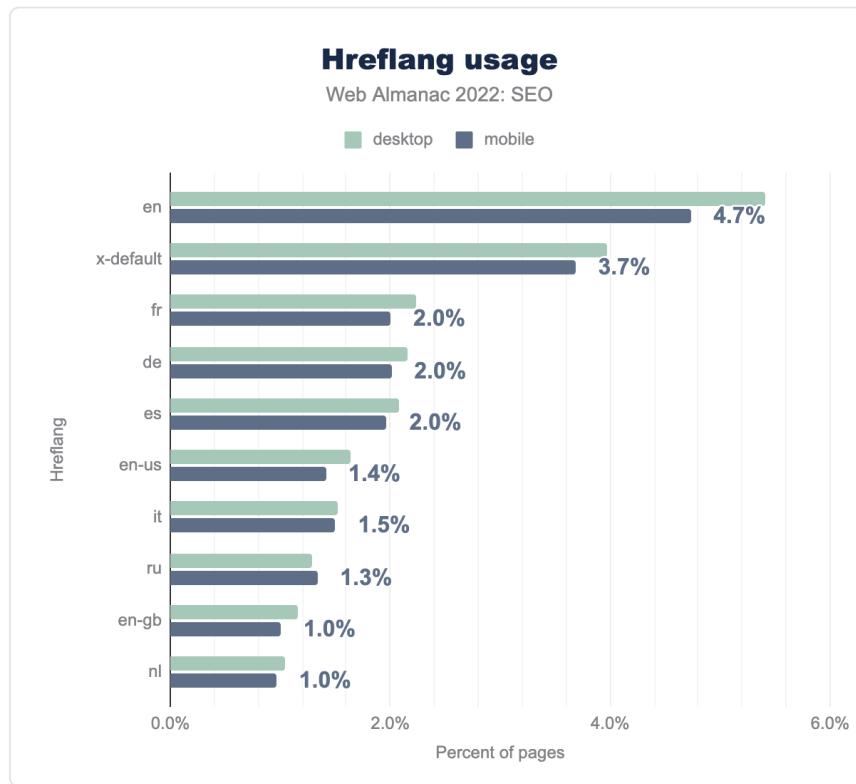


Figure 10.38. Hreflang usage.

Currently, 9.6% of sites use hreflang tags on desktop while 8.9% use them on mobile. This is a slight increase from 2021 when 9.0% of sites used hreflangs tags on desktop and 8.4% implemented them on mobile.

The most popular hreflang tag in 2022 is “en” [English], which accounts for 5.4% usage on desktop and 4.7% on mobile. Those percentages are approximately the same as the year before.

After x-default, which is the “fallback” version (and the second most common to be adopted), the hreflang tags for French, German and Spanish are the next most frequently used.

The three different ways to implement hreflang tags are via the `<head>`, link headers, or XML sitemaps. Note: As this data is looking solely at homepages, XML sitemaps are not included.

Content language usage

While Google tends to use hreflang tags, other search engines such as Bing prefer the content-language attribute³⁴². This can be implemented using two methods:

1. HTML
2. HTTP Header

³⁴² <https://developer.mozilla.org/docs/Web/HTTP/Headers/Content-Language>

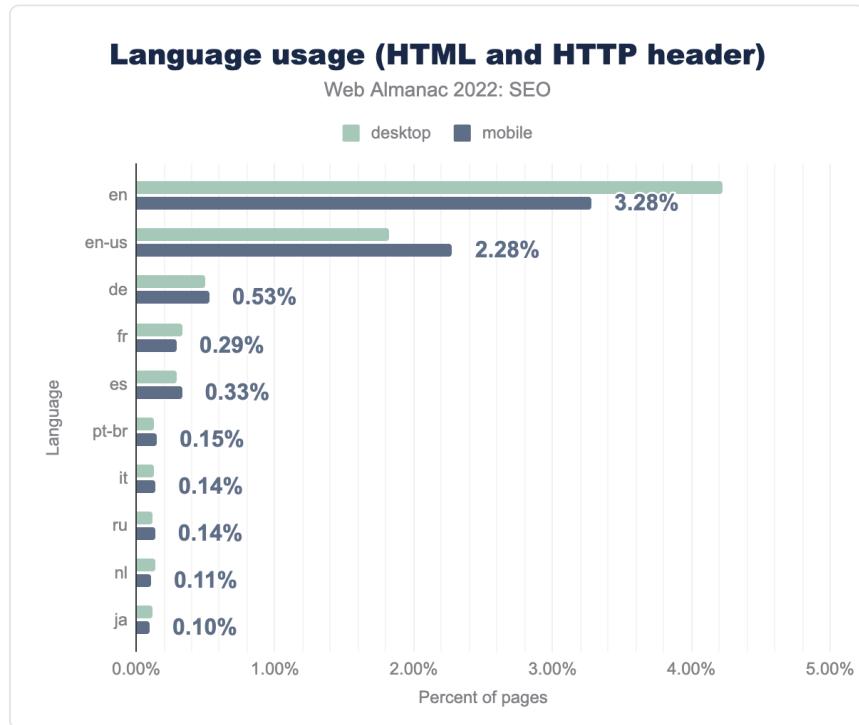


Figure 10.39. Language usage (HTML and HTTP header).

In 2022, HTTP server response is the most popular implementation method of content-language, with 8.27% of mobile sites using this and 8.82% of desktop sites. However this has seen a decline in adoption on mobile compared to 2021 when 9.3% of mobile sites used it. Conversely, desktop has seen a slight increase compared to 2021 when 8.7% of sites used it.

HTML, on the other hand, has 2.98% adoption on desktop in 2022 and 3.01% adoption on mobile. But again there's a decline in mobile usage compared to 2021 when 3.3% of mobile sites used the HTML tag.

Conclusion

Much like patterns in our data from 2021³⁴³, 2020³⁴⁴, and 2019³⁴⁵, the majority of sites analyzed are showing small, yet consistent, improvement when it comes to various fundamentals of SEO,

343. <https://almanac.httparchive.org/en/2021/seo>

344. <https://almanac.httparchive.org/en/2020/seo>

345. <https://almanac.httparchive.org/en/2019/seo>

including having indexable and crawlable pages.

We have also seen an increasing focus on performance elements such as Core Web Vitals, with 39% of sites now having passing scores compared to just 20% in 2020 when the update was first announced. This seems to indicate sites are now taking Google's guidance more to heart. Still, more work needs to be done across the web.

Newer introductions, such as the `indexifembedded` tag, are seeing slow pick-up. This underscores the continuous need for adoption of best practices and how much opportunity for growth there is in SEO, search engine friendliness, and the state of the web in general.

Authors



Sophie Brannon

@SophieBrannon SophieBrannon

Sophie is the Client Services Director at UK-based agency Absolute Digital Media and specializes in SEO strategy and content marketing in highly competitive industries such as health and finance. Sophie is a conference speaker and industry blogger, and has proven experience in strategizing and delivering award-winning campaigns on a local, national and international scale.



Itamar Blauer

@ItamarBlauer itamarblauer <https://www.itamarblauer.com/>

Itamar Blauer is an SEO expert based in London. He has a proven track-record of increasing rankings with SEO that is UX-focused, data-backed, and creative.



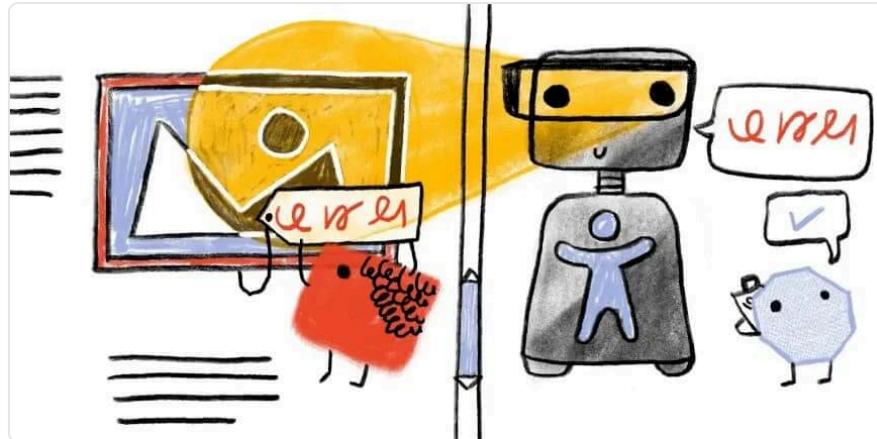
Mordy Oberstein

mordy-oberstein

Mordy Oberstein is the Head of SEO Branding at Wix. He also serves as a consultant for SEMrush and sits behind the mic of multiple SEO podcasts, including the SERP's Up podcast.

Part II Chapter 11

Accessibility



Written by Saptak Sengupta, Thibaud Colas, and Scott Davis

Reviewed by Shaina Hantsis

Analyzed by Thibaud Colas

Edited by Kirsty Simmonds

Introduction

27% of the global online population is using voice search on mobile³⁴⁶. 85% of Facebook videos are watched with closed captions on and sound off³⁴⁷. When you ask voice assistants like Siri, Alexa, and Cortana a question, they typically read their answer from a web page using screen reader technology that has been around for as long as personal computers have existed³⁴⁸.

When does a software feature cease being an “accessibility feature” and simply become a “feature” that we all use? Ask yourself that the next time you put your smartphone in silent/vibrate mode – especially if you’re not a member of the Deaf/Hard of Hearing community.

Good accessibility benefits everyone, not just those with disabilities. This is one of the core principles of Universal Design³⁴⁹. Tim Berners-Lee said, “The power of the Web is in its

346. <https://www.gwi.com/hubfs/Downloads/Voice-Search-report.pdf>

347. <https://idearocketanimation.com/18761-facebook-video-captions/>

348. <https://www.theverge.com/23203911/screen-readers-history-blind-henter-curran-teh-nvda>

349. https://en.wikipedia.org/wiki/Universal_design

universality. Access by everyone regardless of disability is an essential aspect." After the COVID-19 pandemic started, more and more people have been reliant on the internet. Likewise, accessibility needs to improve as well, or we risk alienating a lot of people.

The median overall site score for all Lighthouse Accessibility audit data rose from 80% in 2020 to 82% in 2021, then 83% in 2022. We hope that this increase represents a shift in the right direction.

Although the state of web accessibility still leaves a lot to be desired, we did see an overall improvement in sites' accessibility this year. The median overall site score for all Lighthouse Accessibility audit data rose from 80% in 2020 to 82% in 2021, then 83% in 2022. Looking at Lighthouse results audit-by-audit gives us a sense of what specific improvements have been made.

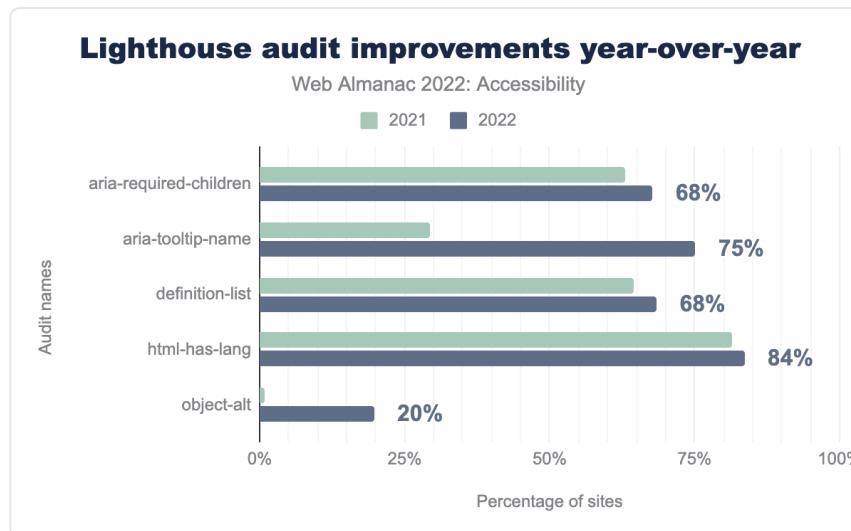


Figure 11.1. Lighthouse audit improvements year-over-year.

Looking at Lighthouse audits reporting results, out of 41 automated checks, 35 passed successfully on more sites in 2022 compared to 2021. 11 audits show improvements greater than 1%, with `aria-required-children`, `aria-tooltip-name`, `definition-list`, `html-has-lang`, and `object-alt` showing the most noteworthy increases. We hope that this increase represents a shift in the right direction.

In the hope of improvement towards accessibility in the web, we have tried to write the chapter with some actionable links and solutions that people can follow. For consistency, we chose to use the person-first term "people with disabilities" throughout this chapter, though we acknowledge that the identity-first term "disabled people" is also used. Our choice in

terminology is in no way prescriptive of which term is most appropriate.

Ease of reading

Readability of information and content on the web is crucial. There are a number of factors in a website that contribute to the content's readability. These factors ensure that everyone on the internet can not only consume content, but also are not harmed by any aspect of the content.

Color contrast

Color contrast refers to how easily the foreground—which can include text, diagrams, iconography or other pieces of information—stands out from the background of the section. A higher color contrast usually means it's easier for people to distinguish the content.

The minimum contrast requirement defined by the Web Content Accessibility Guidelines³⁵⁰ (WCAG) for normal sized text (up to 24px) is 4.5:1 for AA conformance and 7:1 for AAA conformance. However, for larger font sizes, the contrast requirement is only 3:1 as larger text has increased legibility even at a lower contrast.

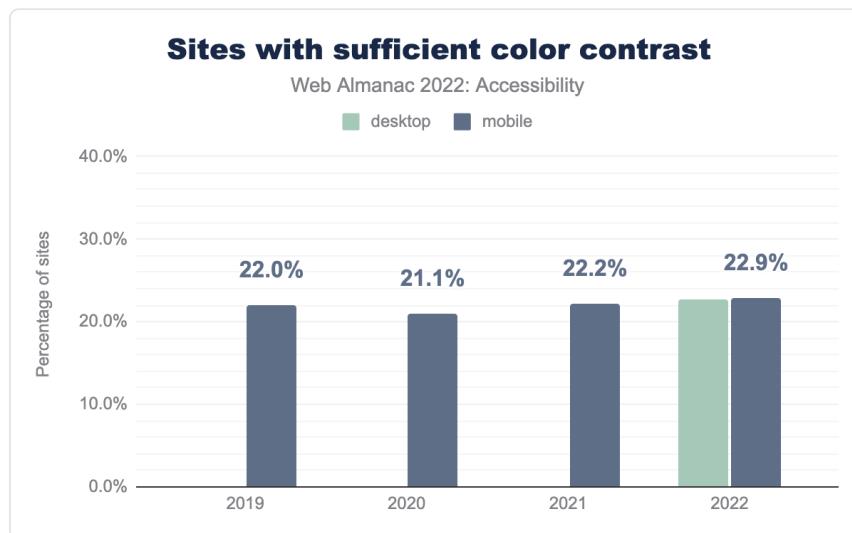


Figure 11.2. Sites with sufficient color contrast.

We found that 22.9% of mobile sites have sufficient text color contrast, which is less than a 1%

350. <https://www.w3.org/WAI/standards-guidelines/wcag/>

increase from last year. In 2022, we also have data for desktop sites, with 22.7% passing automated text contrast checks. The color contrast issue—at least for the text-based color contrasts that we tested—is pretty straightforward to validate even before you start building the website. There are multiple tools that can help developers and designers to check color contrast of text and graphical elements such as:

- Web Color Contrast Checker (by WebAIM)³⁵¹
- Figma Plugin (by Stark)³⁵²

It's a good idea to select a color scheme that passes color contrast requirements at the beginning of a project or while addressing the issues and use it throughout the website. You can also provide other color modes such as dark mode, light mode, high contrast modes to let the user choose.

Zooming and scaling

Zooming is another feature that users with low vision often use to view the text in a website better. There are system settings in the browser, as well as some magnifying tools that allow a user to zoom and scale a website. Adrian Roselli³⁵³ talks in detail about the different reasons you should not disable zoom³⁵⁴.

351. <https://webaim.org/resources/contrastchecker/>
352. <https://www.figma.com/community/plugin/733159460536249875>
353. <https://twitter.com/aardrian>
354. <https://adrianroselli.com/2015/10/dont-disable-zoom.html>

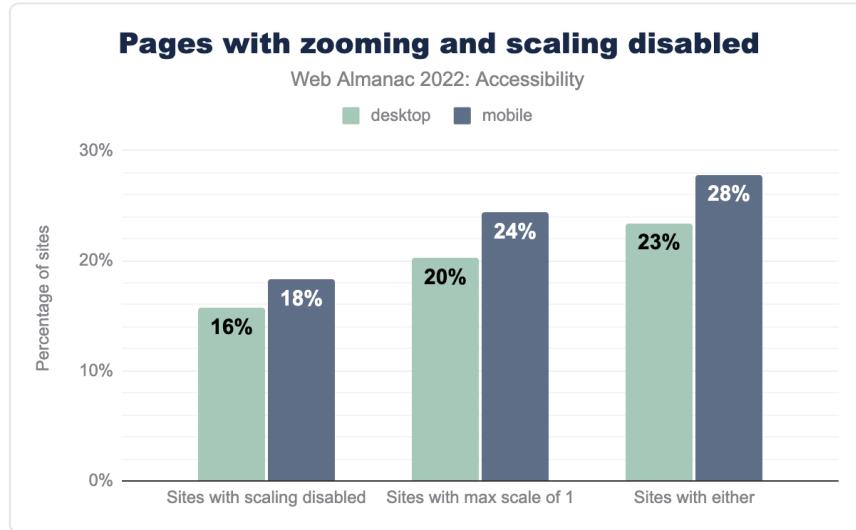


Figure 11.3. Pages with zooming and scaling disabled.

WCAG requires that text in a website can be resized up to at least 200%. We have found that 23% of desktop homepages and 28% of mobile homepages attempt to disable zoom.

The method by which a developer disabled zoom is by adding a `<meta name="viewport" >` tag with a value like `maximum-scale`, `minimum-scale`, `user-scalable=no`, or `user-scalable=0` in the `content` attribute. So if you have a website that has one of these values, please delete those particular values from the `content` attribute to enable zoom.

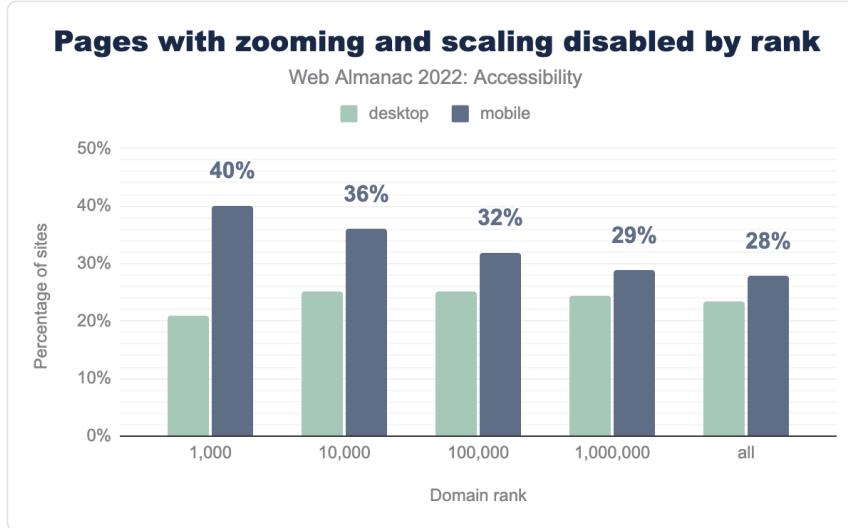


Figure 11.4. Pages with zooming and scaling disabled by rank.

Of the top 1,000 most visited sites, 21% of desktop sites and 40% of mobile sites are built using code that attempts to disable user zooming or scaling. This means that the percentage of sites with zooming disabled is almost double on mobile compared to desktop. It's really important to not disable zooming on any device. To combat this, browsers have begun to override developers' attempts to disable zoom. Manuel Matuzovic³⁵⁵ wrote an article talking about the concerns with disabling zoom and user settings in browsers³⁵⁶.

355. <https://twitter.com/mmatuzo>
 356. <https://www.matuzo.at/blog/2022/please-stop-disabling-zoom/>

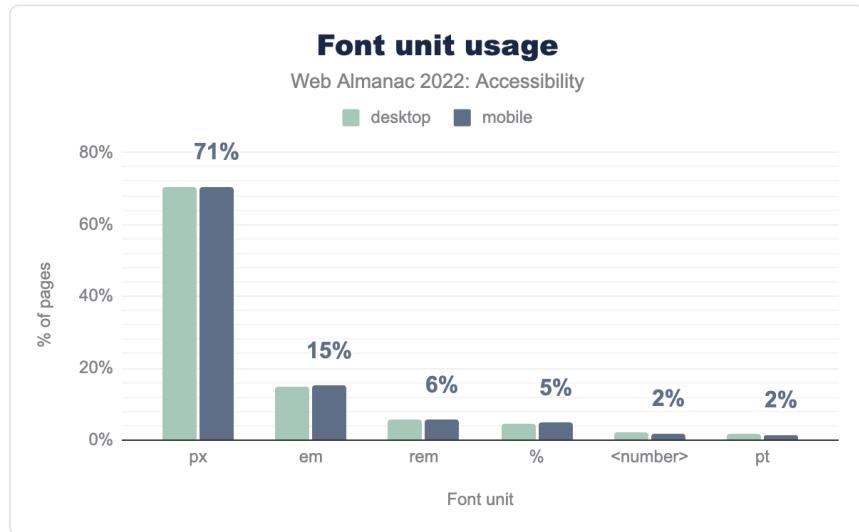


Figure 11.5. Font unit usage.

Another thing to keep in mind is the unit you choose for font size. We found that 71% of pages in desktop use `px`, and only 15% and 6% use `em` and `rem` respectively. So the percentage of `px` usage in desktop has increased by 2% compared to last year, while `em` usage has decreased by 2%. It's considered wise to use relative units such as `em` or `rem` when it comes to `font-size` because if you use `px` it will not scale³⁵⁷ if the user explicitly chooses a bigger or smaller default font size in the browser settings.

Language identification

Language identification using the `lang` attribute is important for providing better screen reader support, and also helps for automatic browser translations. This is another good example of a feature that helps everyone, including people with disabilities. Without the `lang` attribute, the automatic browser translation in Chrome can often translate the text incorrectly. Manuel Matuzovic gives one such example of an auto-translate mishap³⁵⁸ due to the lack of a `lang` attribute.

³⁵⁷. <https://adrianroselli.com/2019/12/responsive-type-and-zoom.html#Update02>

³⁵⁸. <https://www.matuzo.at/blog/lang-attribute/>

83%

Figure 11.6. Mobile sites have a valid `lang` attribute.

It's encouraging to see that 83% of mobile websites do have a `lang` attribute, and within that group, over 99% have a valid value. There's still room for improvement given this is a Level A conformance issue under WCAG 2.1. To meet this success criteria, one can put the `lang` attribute in the `<html>` tag with a known primary language tag³⁵⁹. The `lang` attribute is a global attribute and can be set on other tags as well in case the web page has content in more than one language. It's important to define the correct language for a website. In cases where people copy a template to create a website, there is sometimes a discrepancy between the language used in the website's content and the `lang="en"` attribute used in the code.

User preference

There are certain User Preference Media Queries from the CSS Media Queries Level 5 specification³⁶⁰ that can be used for various accessibility features. These range from choosing a color scheme or contrast mode that works better for the user to reducing animations on the page, which is helpful to people with vestibular disorders.

359. <https://www.w3.org/WAI/standards-guidelines/act/rules/bf051a/#known-primary-language-tag>
360. <https://www.w3.org/TR/mediaqueries-5>

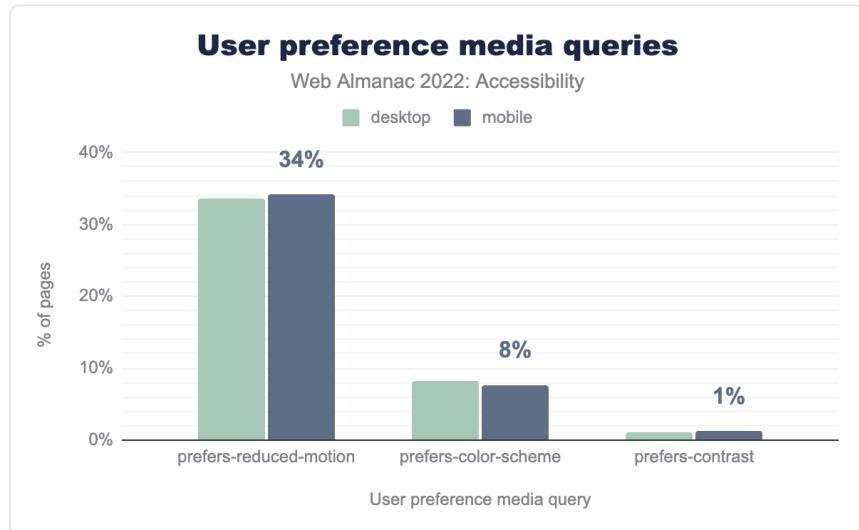


Figure 11.7. User preference media queries.

We found that 34% of mobile websites use `prefers-reduced-motion`. Websites that rely on motion can cause issues for people with vestibular disorders, so it is important to adapt or remove those animations with the `prefers-reduced-motion` media query. There are many great resources³⁶¹ related to designing accessible animation³⁶².

8% of desktop and mobile websites used the `prefers-color-scheme` media query, while 1% of desktop and mobile sites used `prefers-contrast`. Both of these media queries improve content readability by adjusting the display mode³⁶³ based on the user's preference. `prefers-color-scheme` allows the browser to detect the user's system color. Using this information, the web developer can then provide a light or dark mode accordingly. `prefer-contrast` is useful for users with low vision or photosensitivity who may benefit from high contrast modes.

Forced colors mode

Forced colors mode is an accessibility feature intended to increase the readability of text through color contrast. In forced colors mode, the user's operating system takes over control of most color-related styles. Common patterns such as background images are completely disabled, so text-to-background contrast is more predictable. Its best-known implementation is the *High Contrast Mode* in Windows, renamed *Contrast Themes* in Windows 11. Those themes provide alternative low and high contrast color palettes, as well as the ability to customize any

361. <https://alistapart.com/article/designing-safer-web-animation-for-motion-sensitivity/>

362. <https://www.a11yproject.com/posts/design-accessible-animation/>

363. <https://www.a11yproject.com/posts/operating-system-and-browser-accessibility-display-modes/>

of the available system colors³⁶⁴.

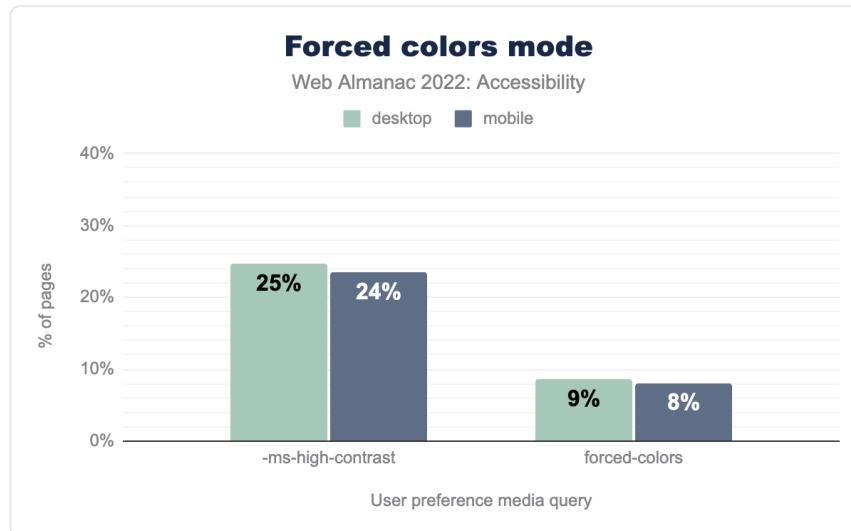


Figure 11.8. Forced colors mode.

Like other user preference media queries, we see a lot of websites making adjustments based on forced colors mode. 8% of mobile sites and 9% of desktop sites use the `forced-colors` media query to alter their styles, while usage of the legacy IE11-only `-ms-high-contrast` media query is above 20% for both mobile and desktop. This doesn't tell us to what extent sites do support forced colors mode, but the data is encouraging nonetheless considering the `forced-colors` media query has only been supported in major browsers since 2020³⁶⁵, and support for emulating `forced-colors` mode on devices other than Windows is only available since February 2022.

Navigation

When talking about navigating through websites, one thing that is important to remember—and be cautious of—is that users may use a variety of methods and input devices. Some people use a mouse to scroll through a page, others use their keyboard or a switch control device, and some may use a screen reader to browse through the different heading levels. When making a website, it's important to ensure that the website works for everyone, irrespective of the device or assistive technology that a person chooses to use.

364. https://developer.mozilla.org/docs/Web/CSS/color_value/system_color_keywords

365. https://caniuse.com/mdn-css_at-rules_media_forced-colors

Focus indication

Focus indication is really important for people who primarily rely on keyboard navigation or switch control devices. These tools are often used by people with limited motor capacity. There are many different switch control devices, from a single switch³⁶⁶ to a sip-and-puff device³⁶⁷. Visible focus styles and proper focus orders are essential for such users to get a visual indication of where they are on a page.

Focus styles

The WCAG requires a visible focus indicator for all interactive content to help people know which element has the keyboard focus as they traverse through a page. In fact for WCAG 2.2³⁶⁸ (which is expected to be published in December 2022), it has been promoted from AA to Level A³⁶⁹.

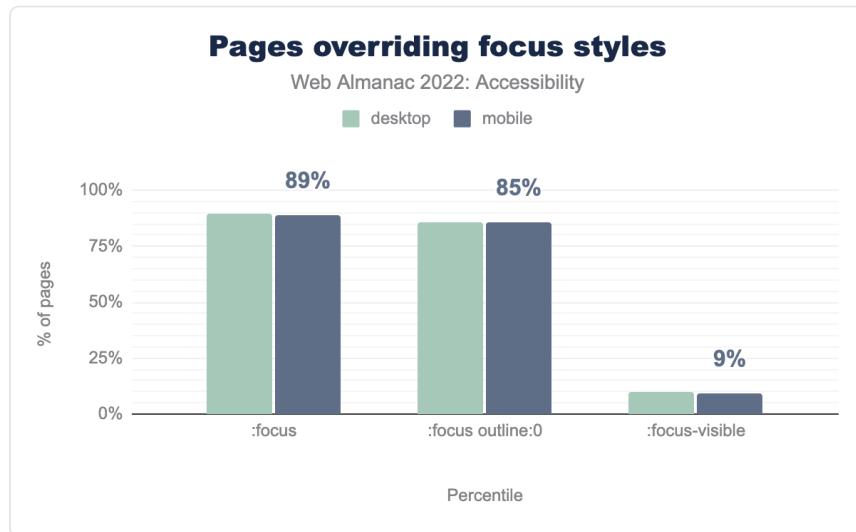


Figure 11.9. Pages overriding focus styles.

We found that 86% of websites add `:focus {outline: 0}`. This removes the default outline that browsers use for the focused interactive element. In some cases, they are overridden using some custom styling, but not always. This makes it impossible for users to determine which element has focus which in turn hinders navigation. Sara Souedan³⁷⁰ has a

366. <https://www.24a11y.com/2018/i-used-a-switch-control-for-a-day/>

367. <https://accessibleweb.com/assistive-technologies/assistive-technology-focus-sip-and-puff-devices/>

368. <https://w3c.github.io/wcag/guidelines/22/>

369. <https://w3c.github.io/wcag/guidelines/22/#focus-visible>

370. <https://twitter.com/SaraSouedan>

great article on how to design WCAG-compliant focus indicators³⁷¹. However, it's exciting to see that 9% of websites have `:focus-visible` compared to only 0.6% last year. This is definitely a step in the right direction.

`tabindex`

`tabindex` is an attribute that can be added to elements to control whether they can receive focus. Depending on its value, the element can also be organized within the keyboard focus or "tab" order.

We found that 60% of mobile websites and 62% of desktop websites use `tabindex`. The `tabindex` attribute can be used for a few different purposes, which may or may not cause accessibility issues:

- Adding `tabindex="0"` adds an element in the sequential keyboard focus order. Custom elements and widgets that are intended to be interactive need an explicitly assigned `tabindex="0"`.
- `tabindex="-1"` means that the element is not in the keyboard focus order, but can be programmatically focused using JavaScript.
- A positive value for `tabindex` is used to override the keyboard focus order and most of the time leads to a WCAG 2.4.3 - Focus Order³⁷² failure

It's important to remember that placing non-interactive elements in the keyboard focus order can be confusing for low-vision users and should hence be avoided.

³⁷¹ <https://www.sarasoueidan.com/blog/focus-indicators/>
³⁷² <https://www.w3.org/TR/UNDERSTANDING-WCAG20/navigation-mechanisms-focus-order.html>

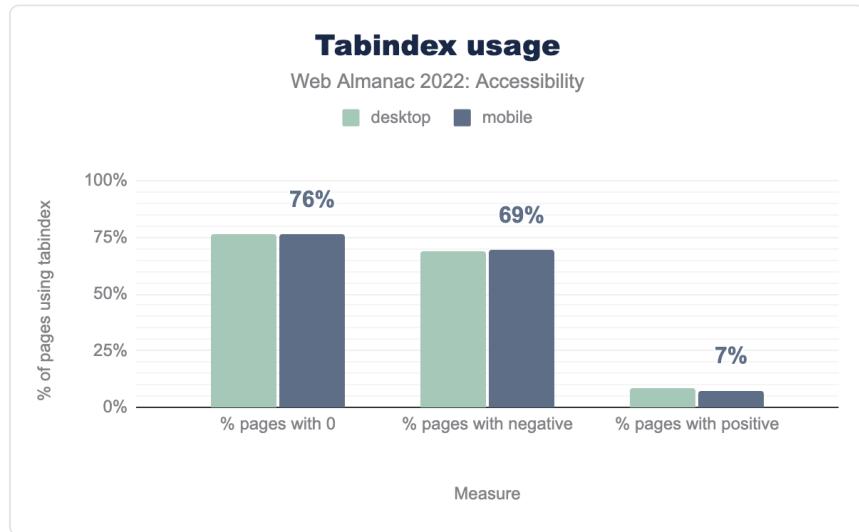


Figure 11.10. `tabindex` usage.

Out of all websites with `tabindex` attribute, 7% have `tabindex` with a positive value. Using positive values for `tabindex` is generally bad practice since it disrupts the normal navigation. Karl Groves³⁷³ has a great article³⁷⁴ that explains this concept further.

Landmarks

Landmarks help divide a web page into thematic regions that makes it easier for users of assistive technologies to understand the page structure and navigate the website. For example, a rotor menu³⁷⁵ can be used to navigate between different page landmarks, while skip links³⁷⁶ can be used to target landmarks, including `<main>`. Landmarks can be created using various HTML5 elements or explicitly adding ARIA landmark roles³⁷⁷. However, following the first rule of ARIA, one should give preference to native HTML5 elements whenever possible.

³⁷³. <https://twitter.com/karlgroves>

³⁷⁴. <https://karlgroves.com/2018/11/13/why-using-tabindex-values-greater-than-0-is-bad>

³⁷⁵. <https://www.w3.org/WAI/ER/techniques/skipnav/>

³⁷⁶. <https://www.w3.org/WAI/ER/techniques/skipnav/>

³⁷⁷. <https://www.w3.org/TR/WCAG20-TECHS/ARIA11.html>

HTML5 element	ARIA role equivalent	Pages with element	Pages with role	Pages with element or role
<main>	role="main"	31%	17%	38%
<header>	role="banner"	63%	13%	65%
<nav>	role="navigation"	63%	22%	67%
<footer>	role="contentinfo"	65%	11%	66%

Figure 11.11. Landmark element and role usage (desktop).

The most commonly expected landmarks that the majority of web pages should have are

<main>, <header>, <nav> and <footer>. We found that only 31% of desktop and mobile pages have a native HTML <main> element, while 17% of desktop pages have an element with a role="main", and 38% of pages have either <main> or role="main". It's good to see the use of native elements increase. Scott O'Hara³⁷⁸'s article on landmarks³⁷⁹ covers all the details that one should keep in mind to ensure better accessibility.

Heading hierarchy

Headings help all users, including those using assistive technologies, to navigate through the website. Users with assistive technologies can navigate to the exact sections that they are interested in. As mentioned in Marcy Sutton³⁸⁰'s article on headings and semantic structure³⁸¹, headings can be thought of as a table of contents that one can navigate through to go to a particular content area.

58%

Figure 11.12. Mobile sites passing the Lighthouse audit for properly ordered headings.

58% of websites pass the test for properly ordered headings that do not skip levels, which is the same as last year. Hopefully this number will increase next year since the document outline example in WHATWG standards have been updated³⁸². A very important thing to remember is

378. <https://twitter.com/scottohara>

379. <https://www.scottohara.me/blog/2018/03/03/landmarks.html>

380. <https://twitter.com/marcysutton>

381. <https://marcysutton.com/how-i-audit-a-website-for-accessibility#Headings-and-Semantic-Structure>

382. <https://github.com/whatwg/html/pull/7829>

that heading levels don't have to represent the actual style (or importance) of a particular element. Headings are to be used primarily for hierarchy purposes, while CSS can be used for the styling of the element. A very good article on how to structure headings in your page is by Steve Faulkner³⁸³ titled, "How to mark up subheadings, subtitles, alternative titles and taglines"³⁸⁴.

Secondary navigation

WCAG requires websites to have multiple ways to navigate between the different pages apart from the primary navigation menu in the header—see Success Criterion 2.4.5: Multiple Ways³⁸⁵. For example many people, including those with cognitive limitations, prefer to use search features to find a page when there are a substantial number of pages in a website.

There are 23% of websites on mobile that have a search input, and 24% on desktop. Another recommended method for secondary navigation is to include a sitemap for a website. Although we do not have any data about the presence of site maps, this technique guide from the W3C³⁸⁶ explains what they are in detail and how to implement one effectively.

Skip links

Skip links allow keyboard or switch control device users to skip through different sections of pages without having to pass every focusable item. One of the most common sections to skip is the primary navigation to go to the `<main>` section, especially if a website has a really large number of interactive items in their primary navigation.



21%

Figure 11.13. Mobile and desktop pages which likely have a skip link

We found 21% of desktop and mobile pages likely have a skip link, allowing users to bypass part of the page content. This figure could be higher in practice, as our detection only checks for the presence of skip links early in the page (for example to skip navigation). Skip links can also be used to skip parts of the page.

383. <https://twitter.com/stevefaulkner>
 384. <https://stevefaulkner.github.io/Articles/How%20to%20mark%20up%20subheadings,%20subtitles,%20alternative%20titles%20and%20taglines.html>
 385. <https://www.w3.org/WAI/WCAG21/Understanding/multiple-ways.html>
 386. <https://www.w3.org/WAI/WCAG21/Techniques/general/G63>

Document titles

Descriptive page titles are useful when navigating between pages, tabs and windows, as the new page has its title read by assistive technologies to help users keep track of where they are.

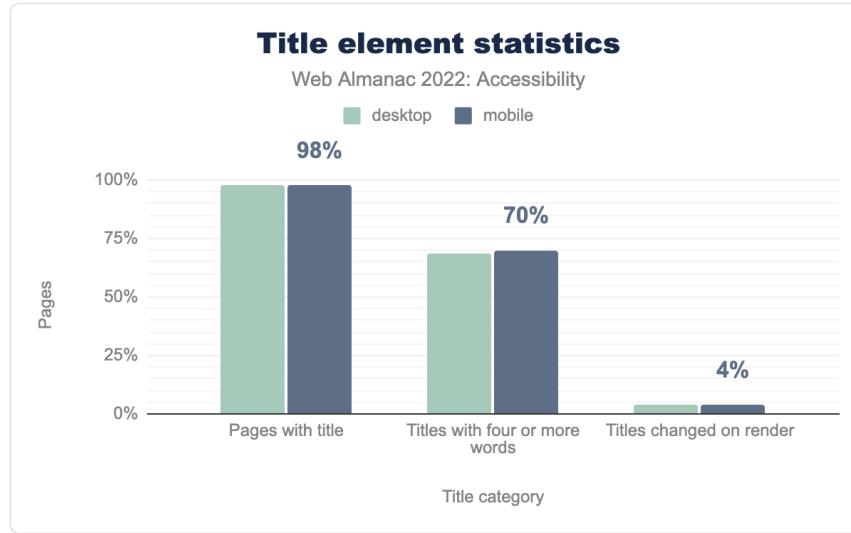


Figure 11.14. Title element statistics.

Though there are 98% of mobile websites which have a document title, only 70% have a title that is longer than four words. Since we only scan homepages of websites, it's not possible for us to determine if the inner pages of the website use a more detailed text in the `<title>` tag that describes the page. A title should ideally have both the title of the website as well as a title giving context about the page in the website for better navigation.

Tables

Tables help in representing data and the relationships between the data using two axes. Tables should have a well-formatted structure with the appropriate elements and markups that helps assistive technology users to easily comprehend the data represented in the table, as well as navigate through the table. Table caption, appropriate headers and appropriate header cells for every row are as such important elements that help users of assistive technology to make sense of the data.

	Table sites		All sites	
	Desktop	Mobile	Desktop	Mobile
Captioned tables	5.4%	4.7%	1.3%	1.2%
Presentational table	1.2%	0.9%	0.6%	0.5%

Figure 11.15. Accessible table usage.

When providing a caption for the table, the `<caption>` element is the correct semantic choice to provide the most context to a screen reader user—though there are alternative ways of labelling a table³⁸⁷. Table captions act as a heading summarizing the information of the table. 1.3% of desktop and mobile sites with table elements present used a `<caption>`.

Tables are also sometimes used for laying out pages, though with the arrival of Flexbox and Grid properties in CSS, one should definitely avoid tables for any visual formatting. However, if there is no other option, tables can set a `role="presentation"`. We observe 1% of tables using this workaround.

Forms

Forms are one of the most common ways that users submit information to and interact with websites. Whether it be logging into a site, creating a post on social media, or making a purchase at an ecommerce site, all of those user journeys will at some stage require a form. Without proper form accessibility, people with disabilities can't interact with them properly which in turn stops them from completing their tasks and achieving full information and feature parity with non-disabled users.

There are specific things that one should keep in mind when it comes to accessibility in forms.

`<label>` element

The `<label>` element is the most effective way of providing accessible names to input fields (or form controls³⁸⁸) in a form. One can link a `<Label>` to a form control programmatically using the `for` attribute. The `for` attribute should contain the value of the `id` attribute of the form control element that you want to link it with. For example:

³⁸⁷ <https://www.w3.org/WAI/tutorials/tables/caption-summary/>
³⁸⁸ https://developer.mozilla.org/docs/Learn/Forms/Basic_native_form_controls

```
<label for="emailaddress">Email</label>
<input type="email" id="emailaddress">
```

The `for` attribute is important because without it, the `<label>` won't be programmatically linked to a corresponding form control. This affects the usability of the form as it's likely that a field does not have a semantically linked label unless another method is used.

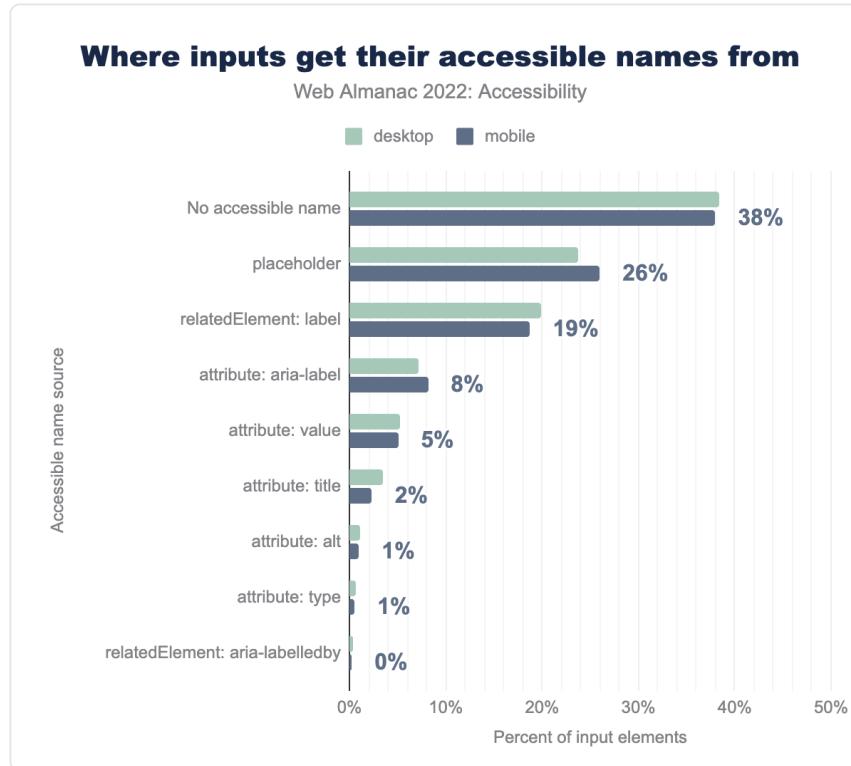


Figure 11.16. Where inputs get their accessible names from.

38% of inputs have no accessible names, while only 19% use `<label>`. Without a proper accessible name, a screen reader user or a voice-to-text user won't be able to identify what data an input is trying to collect. Often there are inputs on websites that don't have any visible labels, which causes issues for all users, or the input's purpose isn't clearly defined both visually and programmatically. In select cases where a label can be visually excluded (such as a search field), one must still add a screen reader-only `<label>` to provide the accessible name.

placeholder attribute

The purpose of a `placeholder` attribute in a form control is to provide an example of the data or format that the form control accepts. For example, `<input type="text" id="credit-card" placeholder="1234-5678-9999-0000">` lets the user know that a card number should be entered with dashes in between every 4 digits.

However, unlike `<label>` elements, the `placeholder` attribute disappears the moment someone starts typing or entering data. This can cause users with cognitive disabilities to get disoriented about the data they were trying to input. Also, not all screen readers support the `placeholder` attribute for accessible names which is also problematic. Hence, using the `placeholder` attribute for accessible names can create many accessibility issues³⁸⁹ and should be avoided.

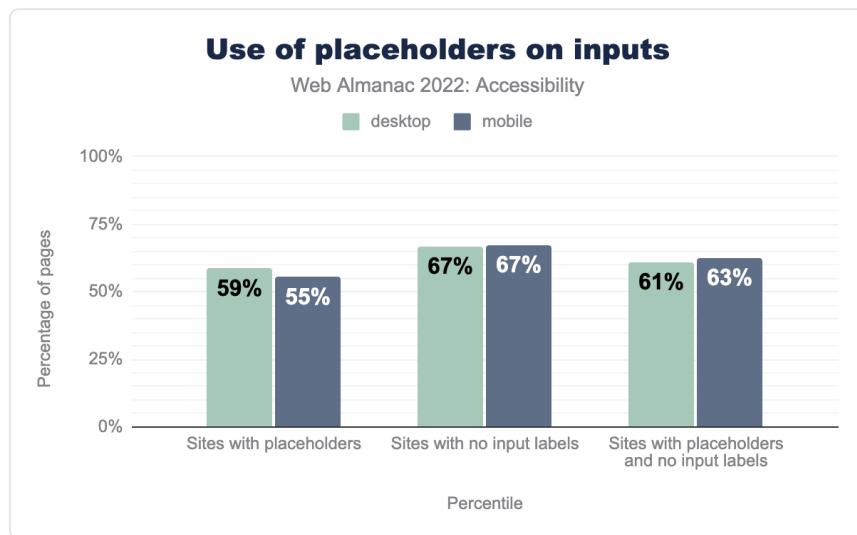


Figure 11.17. Use of placeholders on inputs.

62.7% of the websites surveyed have inputs with only a `placeholder` attribute and no `<label>` element linked to it, which is extremely problematic. The HTML5 specification³⁹⁰ clearly states “The placeholder attribute should not be used as an alternative to a label.” It’s important to provide a `<label>` to improve accessibility for all.

^{389.} [https://www.smashingmagazine.com/2018/06 placeholder-attribute/](https://www.smashingmagazine.com/2018/06	placeholder-attribute/)

^{390.} <https://html.spec.whatwg.org/#the-placeholder-attribute>

Use of the placeholder attribute as a replacement for a label can reduce the accessibility and usability of the control for a range of users including older users and users with cognitive, mobility, fine motor skill or vision impairments.

— The W3C's Placeholder Research³⁹¹

Requiring information

When websites gather input from their users, they need a clear way to indicate what information is optional, and what information is required to submit. For example, in a form an email address might be a required field, but a middle name can be an optional field. Before HTML5 introduced the required attribute for `<input>` fields in 2014, a common convention was to put an asterisk (*) in the label for required input fields. However, just using an asterisk is only a visual indicator, and it provides no validation or sufficient information to assistive technologies that a field is required.

³⁹¹ https://www.w3.org/WAI/GL/low-vision-a11y-tf/wiki/Placeholder_Research

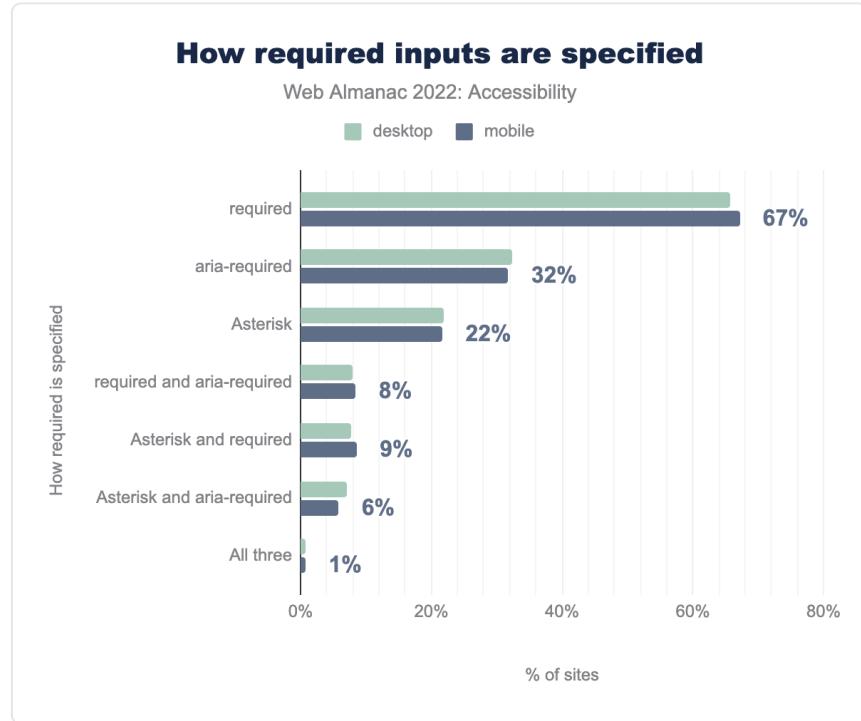


Figure 11.18. How required inputs are specified.

The `required` and `aria-required` attributes are two ways of telling an assistive technology that an input field is not optional. The `required` attribute also prevents form submission without an input, while `aria-required` only conveys the information to assistive technology and doesn't validate the input. We found that 67% of the sites use the `required` attribute and 32% use `aria-required`. However, there are still 22% websites which only use an asterisk (*) to indicate a field is required. That should definitely be avoided unless it is also accompanied with `required` and `aria-required`.

Captchas

Websites often want to verify that the visitor is a human and not a bot, which is a program that crawls through websites for many different purposes. For example, The Web Almanac is created each year by sending out a similar kind of web crawler to gather information from websites. These types of “human-only” tests are called a CAPTCHA – “Completely Automated Public Turing Test, to Tell Computers and Humans Apart”.

19%

Figure 11.19. Mobile sites using a CAPTCHA

19% of mobile websites have one of two captcha implementations which we can detect. This type of test can be difficult to solve for everyone (see CAPTCHAs Have an 8% Failure Rate³⁹²) but would likely be more difficult for people with low vision and other vision or reading-related disabilities. Also, such tests might fail under WCAG 3.3.7 Accessible Authentication³⁹³ once WCAG 2.2 is released. W3C actually has a paper on alternatives to visual turing tests³⁹⁴ that is definitely recommended.

Media on the web

Accessibility considerations become very crucial when it comes to media consumption on the web. A lot of media is often designed in ways that people with disabilities can't consume unless alternative methods are provided. For example, a blind person or a person with a vision impairment needs an audio description for an image or a video so that they can understand the media. A screen reader can only create an audio description if an alternate text describing the image or the video is present. Similarly, for people with hearing disabilities, captions on videos or text tracks for audio are essential to accessing the material.

Images

59%

Figure 11.20. Mobile pages passing the Lighthouse image-alt audit for images with alt text

Images on the web can have an `alt` attribute which provides an alternate text description of the image. A screen reader can then use this information to create an audio description of the image for people with a visual impairment. We found that 59% of sites pass the test for images with alt text, which is a small (1%) increase from 2021.

392. <https://baymard.com/blog/captchas-in-checkout>
 393. <https://w3c.github.io/wcag/understanding/accessible-authentication.html>
 394. <https://www.w3.org/TR/turingtest/>

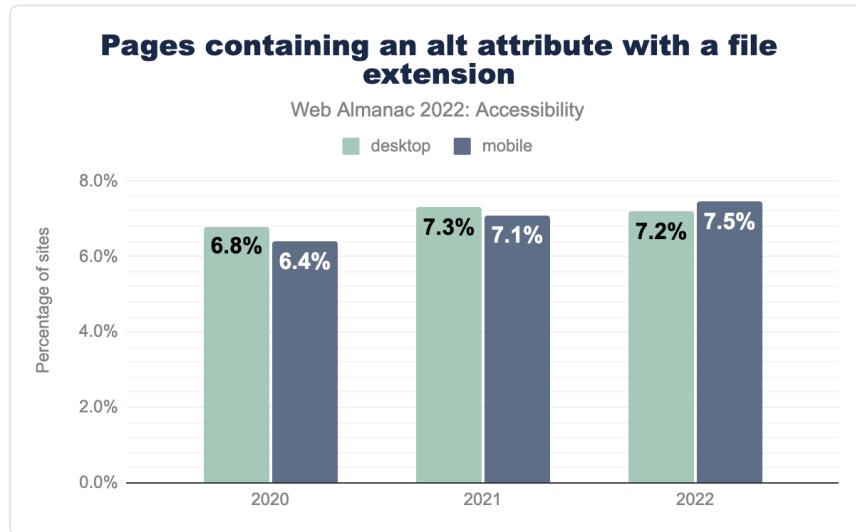


Figure 11.21. Pages containing an `alt` attribute with a file extension.

The text in the `alt` attribute depends on the context. If the image is decorative and doesn't provide any meaningful information, then `alt=""` is ideal. However, if the image is crucial to the context of the page, then a proper text description is essential. If the image is a child of a link, then ideally the `alt` attribute should be used to label the link so the user knows where the link takes them. We found that 7.5% of mobile web pages and 7.2% of desktop pages with an `alt` attribute have a file extension assigned to that `alt` attribute. This probably means that the `alt` attribute just contains the image filename, which is likely not helpful at all, and should be avoided in every case.

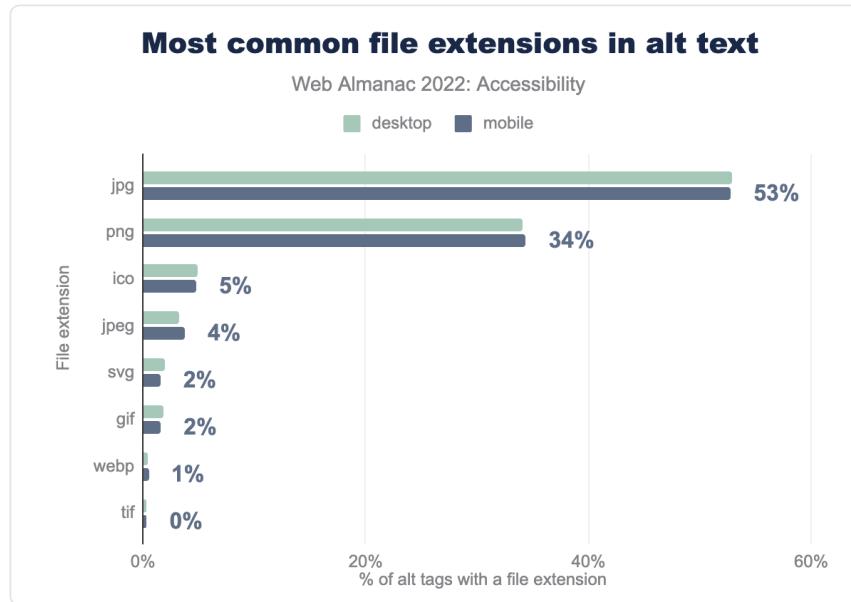


Figure 11.22. Most common file extensions in `alt` text.

The top five file extensions explicitly included in the alt text value (for sites with images that have non-empty alt values) are jpg, png, ico, jpeg and svg. This likely reflects the use of CMS or other content management methods which auto-generate alternative text for images or ask the content editors for image descriptions compulsorily. However, if the CMS just puts the image filename in the `alt` attribute, this often provides no value to the users, so it's important that meaningful text descriptions are provided.

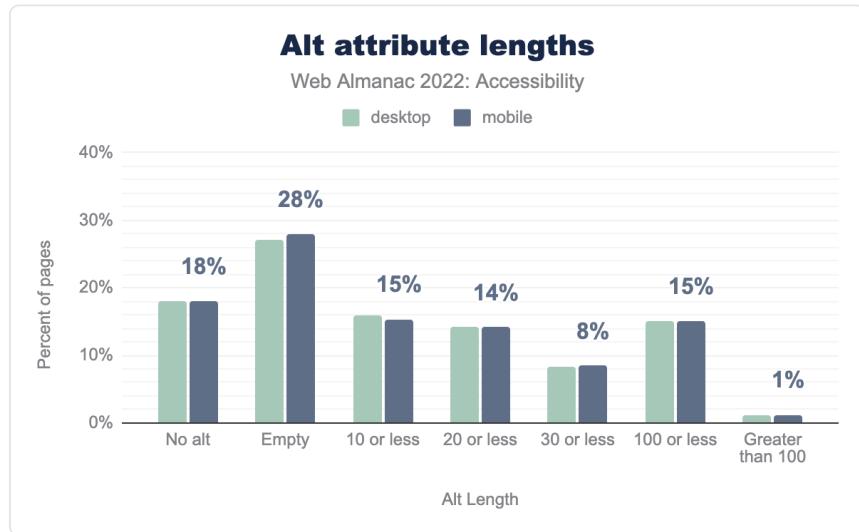


Figure 11.23. `alt` attribute lengths.

We found that 27% of alt text attributes in desktop and mobile websites were empty. An empty `alt` attribute is supposed to be used only when the image is presentational and should not be described by screen readers or other assistive technologies. However, most images on the web do add value to the content in the web page³⁹⁵ and hence should have a proper text description. We found that 15.3% have 10 or fewer characters, which would be a strangely short description for most images, indicating that information parity has not been achieved. Though it is possible that some of them might be used to provide labeling for a link, in which case it's okay.

Audio and video

`<track>` allows providing timed textual content for `<audio>` and `<video>` elements. This can be for subtitles, captions, descriptions, or chapters. Captions allow people with permanent or temporary hearing loss to be able to consume the audio content. Descriptions allow blind screen reader users to understand what is happening in the video.

³⁹⁵. <https://www.smashingmagazine.com/2021/06/img-alt-attribute-alternate-description-decorative/>

0.06%

Figure 11.24. Desktop websites with an `<audio>` element have at least one accompanying `<track>` element

`<track>` loads one or more WebVTT files, which allows text content to be synchronized with the audio it is describing. When looking at only sites with a detectable `<audio>` element, we found that only 0.06% of all pages on desktop and 0.09% of all pages on mobile with had at least one accompanying `<track>` element. Looking at all `<audio>` elements, we see only 0.03% and 0.05% respectively include a `<track>`.

0.71%

Figure 11.25. Desktop `<video>` elements with an accompanying `<track>` element

The `<track>` element was included with a corresponding `<video>` element less than 1% of the time – 0.71% for desktop sites, and 0.65% for mobile sites. These data points do not include audio or video embedded via an `<iframe>` element, which is common for content like podcasts or YouTube videos. It should also be noted that most popular third-party audio and video embedding services include the ability to add synchronized text equivalents.

Assistive technology with ARIA

Accessible Rich Internet Applications, or ARIA³⁹⁶ defines a set of attributes for HTML5 elements that can be used to make web content more accessible for people with disabilities. However, overuse of ARIA can cause more issues than improvements to accessibility. It is always recommended to use ARIA attributes only when using HTML5 is not sufficient to create a fully accessible experience. It should not be used as a replacement of native HTML5 elements, or overused unnecessarily.

ARIA roles

When an assistive technology encounters an element, the element's role communicates information about how someone might interact with its content.

³⁹⁶. <https://www.w3.org/TR/using-aria/>

For example, Tabbed interfaces³⁹⁷ are one of the most commonly used UI elements that need various ARIA roles to be defined explicitly to convey the structure of the UI properly. A common implementation for an accessible tabbed interface is mentioned in the WAI-ARIA Authoring Practices Design Patterns³⁹⁸. When creating a tablist widget, a `tablist` role can be assigned to the container element since there is no native HTML equivalent.

HTML5 introduced many new native elements which have implicit semantics, including roles. For example, the `<nav>` element has an implicit `role="navigation"` and does not need to have this role added explicitly via ARIA.

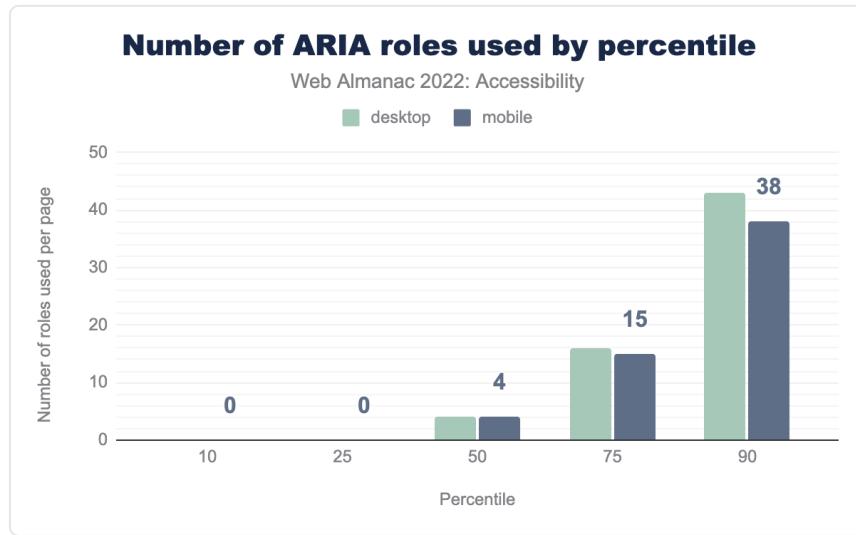


Figure 11.26. Number of ARIA roles used by percentile.

Currently 72% of desktop pages have at least one instance of an ARIA role attribute. The median site has four instances of the `role` attribute.

397. <https://inclusive-components.design/tabbed-interfaces/>

398. <https://www.w3.org/TR/wai-aria-practices-1.1/#tabpanel>

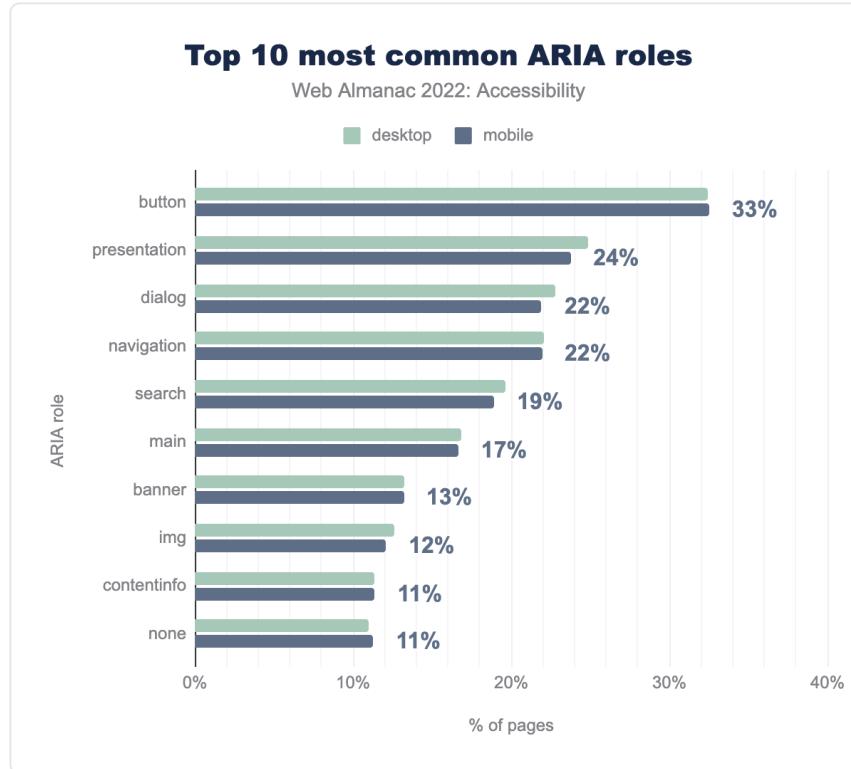


Figure 11.27. Top 10 most common ARIA roles.

We found that 33% (up from 29% in 2021, and 25% in 2020) of desktop and mobile sites had homepages with at least one element with an explicitly assigned `role="button"`. This increase in percentage is likely not good since this indicates that websites are either creating custom buttons using `<div>` or ``, or are adding a redundant role to `<button>` elements. Both of these are bad practices and go against the first rule of ARIA³⁹⁹. Following this rule, one should always use a native HTML element—in this case, `<button>`—when possible.

21%

Figure 11.28. Desktop websites have at least one link with a `button` role

Adding an ARIA role tells assistive technology what the element is, but doesn't provide any

³⁹⁹. <https://www.w3.org/TR/using-aria/#rule1>

other functionality to make elements behave like their native counterpart. For example, we found that 21% of websites had at least one link with a `role="button"`. This kind of pattern can cause issues with keyboard navigation, as links and buttons have different interactions. Though both links and buttons are interactive, links are not activated with the space key, whereas buttons are.

Using the presentation role

When an element has `role="presentation"` declared on it, its semantics are stripped away, as well as the semantics of any of its child elements if those child elements are required child elements (such as `li` nested under a `ul` element, or rows and cells in a table). For example, declaring `role="presentation"` on a parent table or list element will cascade the role to their required child elements and none of them will have table or list semantics.



24%

Figure 11.29. Mobile websites have at least one element with `role=presentation`

Removing an element's semantics causes an element to lose its behavior. It becomes only visually present, and assistive technologies fail to understand the structure of the element and cannot convey that message to the user. For example, a list with a `role="presentation"` will no longer communicate any information to a screen reader about the list structure. We found that 25% of desktop pages and 24% of mobile pages have at least one element with `role="presentation"`.



11%

Figure 11.30. Mobile websites have at least one element with `role=none`

The same effect of semantic removal takes place with `role="none"`. We found that this year, the percentage of `role="none"` has also increased to 11% and is one of the top 10 most common ARIA roles. There are very few use cases where this is particularly helpful for assistive technology users, for example if there is a `<table>` element being used only for layout. But it can otherwise be harmful and should be used sparingly.

Most browsers ignore `role="presentation"` and `role="none"` on focusable elements, including links and inputs, or anything with a `tabindex` attribute set. Browsers also ignore the

inclusion of the role if any of the element contains any global ARIA states and properties, such as `aria-describedby`.

Labeling elements with ARIA

Parallel to the DOM there is a similar browser structure called the accessibility tree. It contains information about HTML elements including accessible names, descriptions, roles and states. This information is conveyed to assistive technologies through accessibility APIs.

The accessible name can be derived from an element's content (such as button text), an attribute (such as an image `alt` attribute value), or an associated element (such as a programmatically associated label for a form control). There is a specificity ranking that is used to determine where the element gets its accessible name from if there are multiple potential sources. Léonie Watson⁴⁰⁰'s article, What is an accessible name?⁴⁰¹ is a great source to learn more about accessible names.

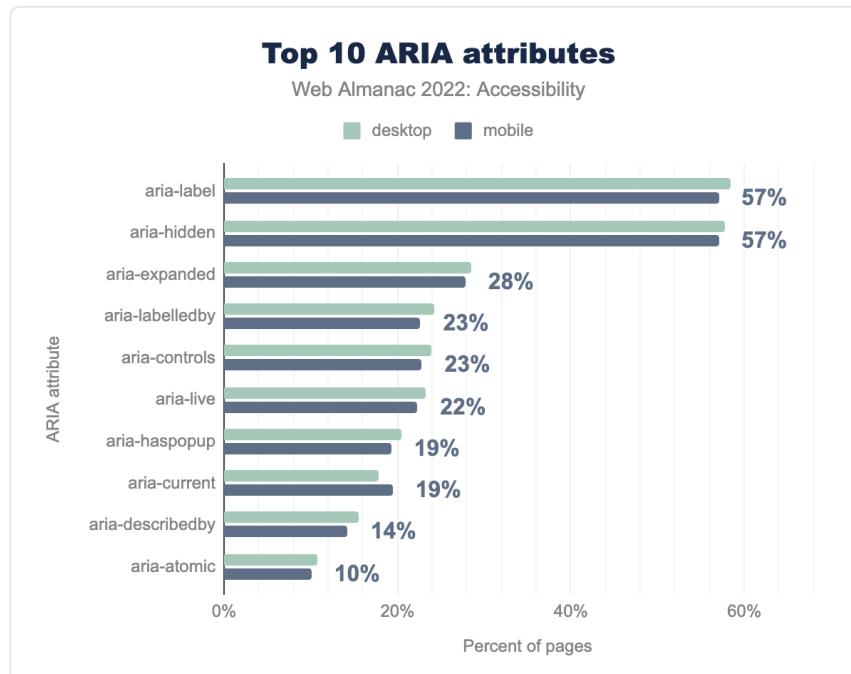


Figure 11.31. Top 10 ARIA attributes.

400. <https://twitter.com/LeonieWatson>

401. <https://developer.paciellgroup.com/blog/2017/04/what-is-an-accessible-name/>

There are two ARIA attributes that help in providing elements with an accessible name: `aria-label` and `aria-labelledby`. These attributes will be preferred over the native derived accessible name so they should be used very carefully and only when necessary. It's always a good idea to test the accessible names obtained with a screen reader, and involve people with disabilities to confirm that it is actually helpful and doesn't make the content more inaccessible.

We found that 58% of desktop pages and 57% of mobile home pages had at least one element with the `aria-label` attribute, making it the most popular ARIA attribute for providing accessible names. We found that 24% of desktop pages and 23% of mobile pages had at least one element with the `aria-labelledby` attribute. This could mean that more elements now have accessible names but it may also be indicative that more elements now lack a visual label which can be problematic for people with cognitive disabilities and voice to text users (see `<label>` element).

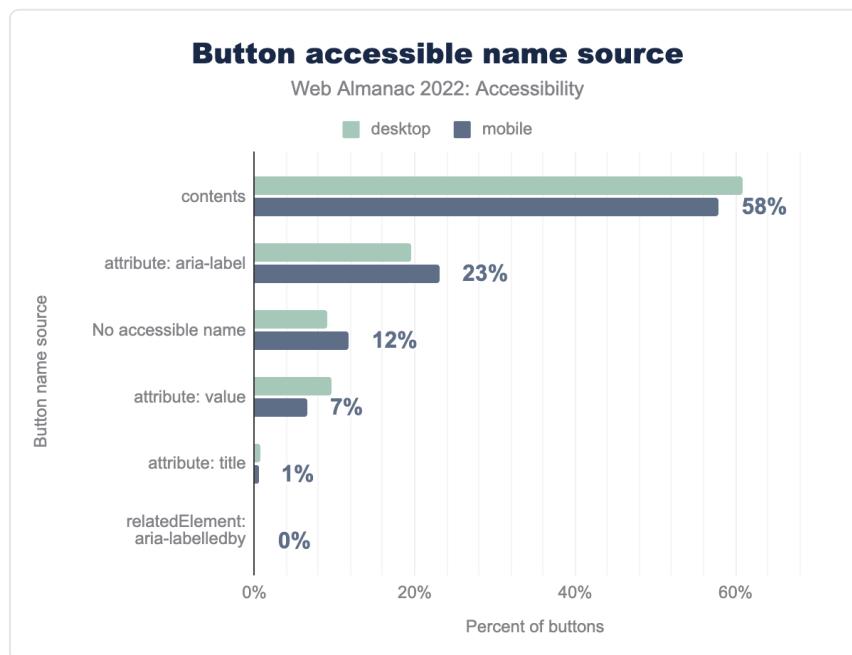


Figure 11.32. Button accessible name source.

Buttons typically get their accessible names from their content or an ARIA attribute. Per the first rule of ARIA, if an element can derive its accessible name without the use of ARIA, this is preferable. Therefore a `<button>` should get its accessible name from its text content rather than an ARIA attribute if possible.

We found that 61% of buttons on desktop and 58% in mobile sites get their accessible name

from content which is good. We also found that 20% of buttons on desktop sites and 23% of buttons on mobile sites get their accessible names from the `aria-label` attribute.

There are a few cases where an `aria-label` can help. For example, if the page has multiple buttons with the same content, but different purposes, then an `aria-label` can be used to provide a better accessible name. Sometimes developers also use `aria-label` when the button has only an image or icon, which might be a reason why more mobile sites use the `aria-label`, rather than the content, to define accessible names.

Hiding content

Sometimes the visual interface can contain some redundant elements that are unhelpful for users of assistive technologies. In such cases `aria-hidden="true"` can be used to hide the element from screen readers. However, they should never be used if omitting such an element would provide a screen reader user with less information than the visual interface. Hiding content from assistive technologies should never be used to skip over content that is challenging to make accessible.

A large, bold, dark blue percentage sign, likely representing the 58% mentioned in the figure caption.

Figure 11.33. Desktop websites having at least one instance of the `aria-hidden` attribute

We found that 58% of desktop pages and 57% of mobile pages had at least one instance of an element with the `aria-hidden` attribute. Hiding and showing content is a prevalent pattern in modern interfaces, and it can be helpful to declutter the UI for everyone.

It's important to use the proper aria attribute to convey the correct message. For example, disclosure widgets should be making use of the `aria-expanded` attribute to indicate to assistive technology that something is revealed by expanding when the control is activated and hidden when activated again. We found that 29% of desktop pages and 28% of mobile pages had at least one element with the `aria-expanded` attribute.

Screen reader-only text

A common technique that developers employ to supply additional information for screen reader users is to use CSS to visually hide a passage of text but make it discoverable by a screen reader. This CSS code⁴⁰² is used such that it's present in the accessibility tree but hidden visually.

402. <https://css-tricks.com/inclusively-hidden/>

15%

Figure 11.34. Desktop websites with a `sr-only` or `visually-hidden` class

`sr-only` and `visually-hidden` are the most common class names used by developers and by UI frameworks to achieve screen reader-only text. For example, Bootstrap and Tailwind use `sr-only` classes for such elements. We found that 15% of desktop pages and 14% of mobile pages had one or both of these CSS class names. It is important to keep in mind that not all screen reader users are fully visually impaired, and thus one should avoid making too much use of screen reader-only solutions.

Dynamically-rendered content

The presence of new or updated content in the DOM sometimes needs to be communicated to screen readers. For example, form validation errors need to be conveyed whereas a lazy-loaded image may not. Updates to the DOM also need to be done in a way that is not disruptive.

23%

Figure 11.35. Desktop pages with live regions using `aria-live`

ARIA live regions allow us to listen for changes in the DOM, such that the updated content can be announced by a screen reader. We found that 23% of desktop pages (up from 21% in 2021, 17% in 2020) and 22% of mobile pages (up from 20% in 2021, 16% in 2020) have live regions using `aria-live`. In addition, pages also use live region ARIA roles⁴⁰³ with implicit `aria-live` values:

403. https://www.w3.org/TR/wai-aria-1.1/#live_region_roles

Role	Implicit aria-live value	Desktop	Mobile
<i>status</i>	<i>polite</i>	5.6%	5.1%
<i>alert</i>	<i>assertive</i>	3.7%	3.4%
<i>timer</i>	<i>off</i>	0.6%	0.6%
<i>log</i>	<i>polite</i>	0.4%	0.4%
<i>marquee</i>	<i>off</i>	0.0%	0.0%

Figure 11.36. Pages with live region ARIA roles, and their implicit `aria-live` value

The `<output>` element also deserves an honorable mention, as the only HTML element with an implicit live region role that will announce its contents to end users. We see it used 16,144 times across our dataset for mobile sites and 12,120 times on desktop, or around 0.0002% of elements usage.

For more information about live region variants and usage check out the MDN live region documentation⁴⁰⁴ or play with this live demo by Deque⁴⁰⁵.

Accessibility apps and overlays

Accessibility overlays are tools claiming to automatically fix websites' accessibility issues.

The Overlay Fact Sheet⁴⁰⁶ defines them as “a broad term for technologies that aim to improve the accessibility of a website. They apply third-party source code (typically JavaScript) to automate improvements to the front-end code of the website.”

Their vendors generally promise a quick and easy solution to online accessibility: integrate one JavaScript snippet onto the site to make it compliant. Web accessibility is simply not possible to achieve with an out of the box solution like this. Automated tools can only detect 30 to 50% of accessibility issues⁴⁰⁷ to start with, and even for issues that can be detected, automated remediation via an overlay cannot always reliably fix those issues.

404. https://developer.mozilla.org/docs/Web/Accessibility/ARIA/ARIA_Live_Regions

405. <https://dequeuniversity.com/library/aria/liveregion-playground>

406. <https://overlayfactsheet.com/#what-is-a-web-accessibility-overlay>

407. <https://alphagov.github.io/accessibility-tool-audit/>

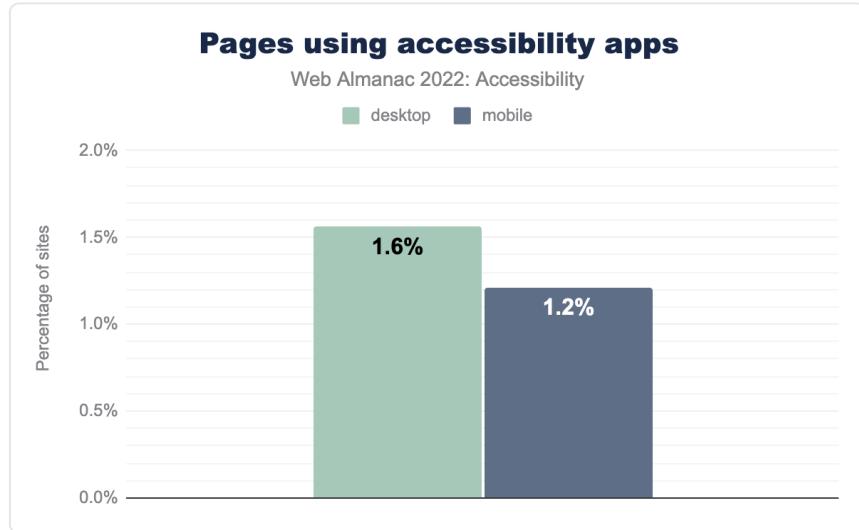


Figure 11.37. Pages using accessibility apps.

We found that 1.6% of desktop websites use one of 22 specific accessibility apps we could detect in 2022. This is a clear rise compared just under 1% in 2021.

Not all of those products are accessibility overlays, however the specific overlays we can detect show a similar rise.

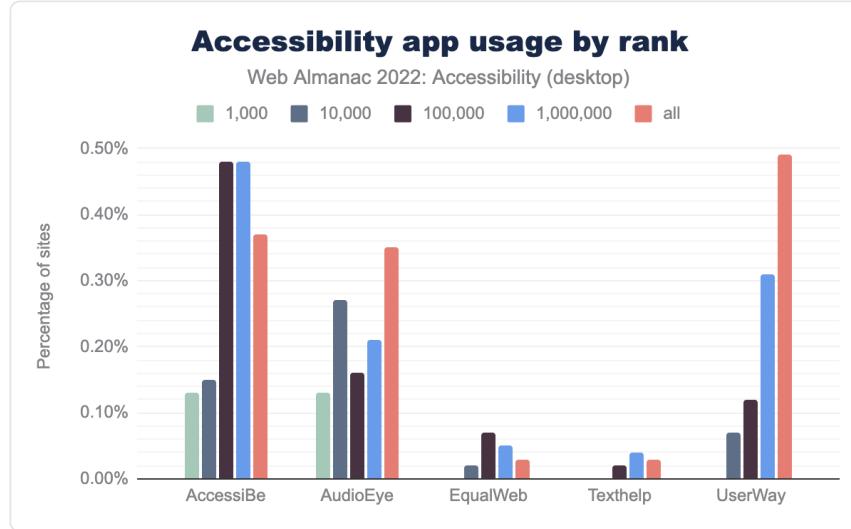


Figure 11.38. Accessibility app usage by rank.

UserWay is the most popular overlay in our dataset, in use by 0.49% of desktop websites and 0.39% on mobile, compared to 0.39% and 0.33% respectively in 2021.

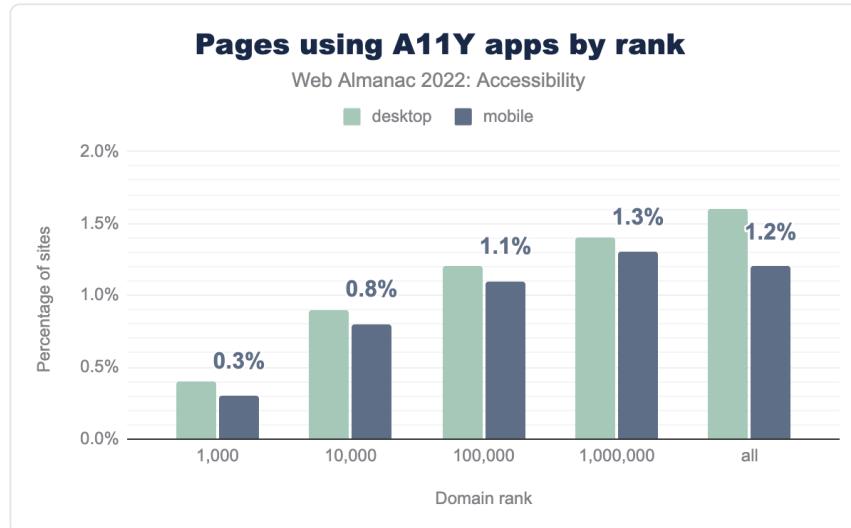


Figure 11.39. Pages using accessibility apps by rank.

Usage of overlays, and accessibility apps generally, is more rare for high-traffic websites. For

sites ranked in the top 1,000 by visits, only 0.3% – or 3 websites – use an overlay.

Concerns with overlays

There is a lot of pushback against overlays, from accessibility advocates⁴⁰⁸ and users⁴⁰⁹ alike. The National Federation of the Blind denounce the practices of accessiBe⁴¹⁰ in particular, a company known for their deceptive marketing, including fake reviews⁴¹¹:

[...] the Board believes that accessiBe currently engages in behavior that is harmful to the advancement of blind people in society. [...] It seems that accessiBe fails to acknowledge that blind experts and regular screen reader users know what is accessible and what is not. The nation's blind will not be placated, bullied, or bought off.

– National Federation for the Blind⁴¹²

accessiBe raised an additional \$30M in 2022⁴¹³, and is one of the overlays showing a clear rise in usage year-to-year, from 0.27% of desktop sites and 0.21% of mobile sites in 2021, to 0.37% and 0.25% in 2022.

Adrian Roselli's #accessiBe Will Get You Sued⁴¹⁴ is an excellent resource on the practical implications of using such an overlay, including legal risks and privacy issues.

Conclusion

Our analysis shows that there hasn't been much of a substantial change in the accessibility issues seen across websites. Though there have been some improvements—for example, adoption of `:focus-visible` has increased by almost 9% this year. Our analysis shows that there are still a lot of straightforward fixes, such as color contrast and image `alt` attributes, that could cause high impacts if addressed.

We see that there are often a lot of misuse of features that may give an illusion of things being more accessible but in reality it often degrades the experience. For example, we see 20% of websites have an anchor tag with `role=button`. Also, we see that 2.2% of `alt` attributes across websites have a file extension in them, which almost certainly doesn't help in conveying the meaning of the image.

408. <https://overlayfactsheet.com/>

409. <https://www.vvce.com/en/article/m7az74/people-with-disabilities-say-this-a11y-tool-is-making-the-web-worse-for-them>

410. <https://nfb.org/about-us/press-room/national-convention-sponsorship-statement-regarding-accessible>

411. <https://www.jedolson.com/2021/02/accessibe-the-fake-wordpress-plugin-in-reviews/>

412. <https://nfb.org/about-us/press-room/national-convention-sponsorship-statement-regarding-accessible>

413. <https://www.geektime.com/accessibe-raised-30m/>

414. <https://adrianroselli.com/2020/06/accessibe-will-get-you-sued.html>

A lot of the accessibility issues that we see in our analysis can be avoided if designers and developers start thinking about web accessibility from the very beginning and not as an enhancement at the end. Like Anna E. Cook once said⁴¹⁵, there's "no MVP without accessibility". The web community needs to realize that a website only has a great User Experience when that User Experience works for everyone, irrespective of the device and assistive technology used. We have tried to focus on key metrics that can be easily addressed in the hope that in 2023 we see the numbers improve.

Authors



Saptak Sengupta

Twitter: @Saptak013 GitHub: SaptakS Website: <https://saptaks.website>

Saptak S is a human rights centered web developer, focusing on usability, security, privacy and accessibility topics in web development. He is a contributor and maintainer of various different open source projects like The A11Y Project⁴¹⁶, OnionShare⁴¹⁷ and Wagtail⁴¹⁸. You can find him blogging at saptaks.blog⁴¹⁹.



Thibaud Colas

Twitter: @thibaud_colas GitHub: thibaudcolas Website: <https://thib.me/>

Thibaud Colas is a web developer and open source contributor focusing on accessibility topics. He is a core contributor to the Wagtail⁴²⁰ CMS, and a member of the accessibility team for Django⁴²¹.



Scott Davis

Twitter: @scottdavis99 GitHub: scottdavis99 Website: <http://thirstyhead.com>

Scott Davis is an author and Digital Accessibility Advocate with Thoughtworks⁴²², where he focuses on leading-edge / innovative / emerging / non-traditional aspects of web development. "Digital Accessibility is so much more than a compliance checkbox; Accessibility is a springboard for innovation."

415. <https://twitter.com/annaecook/status/1404615552883060737>

416. <https://www.a11yproject.com>

417. <https://onionshare.org/>

418. <https://wagtail.org/>

419. <https://saptaks.blog>

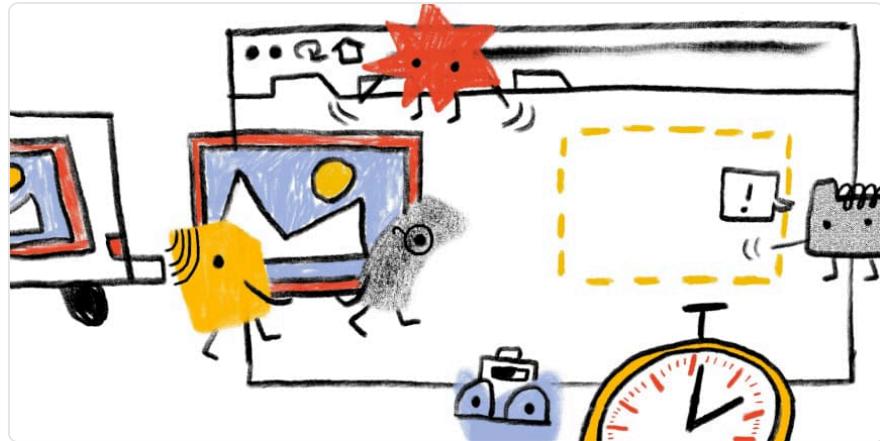
420. <https://wagtail.org/>

421. <https://www.djangoproject.com/>

422. <https://www.thoughtworks.com/>

Part II Chapter 12

Performance



Written by Melissa Ada and Rick Viscomi

Reviewed by Barry Pollard, Patrick Meenan, Prathamesh Rasam, Estelle Weyl, and Kanmi Obasa

Analyzed by Rick Viscomi, Prathamesh Rasam, Sia Karamalegos, and Kanmi Obasa

Edited by Barry Pollard

Introduction

Web performance is crucial to user experience. We've all bounced from a site due to slow load times, or worse, have not been able to access important information. Additionally, numerous case studies⁴²³ have demonstrated that an improvement in web performance results in an improvement in conversion and engagement for businesses. Surprisingly, the industry spotlight is quite elusive for web performance—why is this? Some may say web performance is tough to define and even more challenging to measure.

How do we measure something that is hard to define in the first place? As Sergey Chernyshev⁴²⁴, creator of UX Capture⁴²⁵, says, “*The best way to measure performance is to be embedded into the user's brain to understand exactly what they're thinking as they use the site*”. We can't—and shouldn't—in case that was unclear—do this, so what are our options?

423. <https://wpostats.com/>

424. <https://twitter.com/sergeyche>

425. <https://github.com/ux-capture/ux-capture>

Thankfully, there's a way to measure some aspects of performance automatically! We know the browser is in charge of loading a page, and it goes through a checklist of steps each time.

Depending on which step the browser is on, we can tell how far along the site is in the page load process. Conveniently, a number of performance timeline APIs⁴²⁶ are used to fire off timestamps when the browser gets to certain page load steps.

It's important to note that these metrics are only our best guess at how to gauge user experience. For example, just because the browser fired an event that an element has been painted onto the screen, does that always mean it was visible to the user at that time? Additionally, as the industry grew, more and more metrics showed up while some became deprecated. It can be complicated to know where to start and understand what performance metrics are telling us about our users, especially for folks newer to the field.

This chapter focuses on Google's solution to the problem: Core Web Vitals⁴²⁷ (CWV), web performance metrics introduced in 2020 and made a signal in search ranking⁴²⁸ during 2021. Each of the three metrics covers an important area of user experience: loading, interactivity, and visual stability. The public Chrome UX Report⁴²⁹ (CrUX) dataset is Chrome's view of how websites are performing on CWV. There's zero setup on the developer's part; Chrome automatically collects and publishes data from eligible websites⁴³⁰, for users who have opted in. Using this dataset, we're able to get insights into the web's performance over time.

Although the spotlight of this chapter, it's important to note that CWV are relatively new to the field and not the only way to measure web performance. We chose to focus on these metrics because the search ranking influence was effective almost exactly one year ago, and this year's data gives us insights on how the web is adjusting to this major shift in the industry and where room for opportunity might still exist. CWV are a common baseline that allows performance to be loosely comparable across sites, but it's up to site owners to determine which metrics and strategies are best for their sites. As much as we wish otherwise, there's no way to fit the entire history of the industry or all the different ways to evaluate performance in one chapter.

The CWV program suggests a clearly defined approach to measuring how users actually experience performance—a first for the industry. Are CWV the answer to helping the web become more performant? This chapter examines where the web is currently with CWV and takes a look into the future.

Disclosure: This chapter is coauthored by an employee of Google, which created the Core Web Vitals program. This chapter and its underlying analysis were reviewed and approved by others not affiliated with Google.

426. <https://developer.mozilla.org/docs/Web/API/Performance>

427. <https://web.dev/vitals/>

428. <https://developers.google.com/search/blog/2020/11/timing-for-page-experience>

429. <https://developer.chrome.com/docs/crux/>

430. <https://developer.chrome.com/docs/crux/methodology/#eligibility>

Core Web Vitals

Now that it's been a year since CWV were added as a ranking signal in Google Search, let's see how the program may have influenced user experiences on the web.

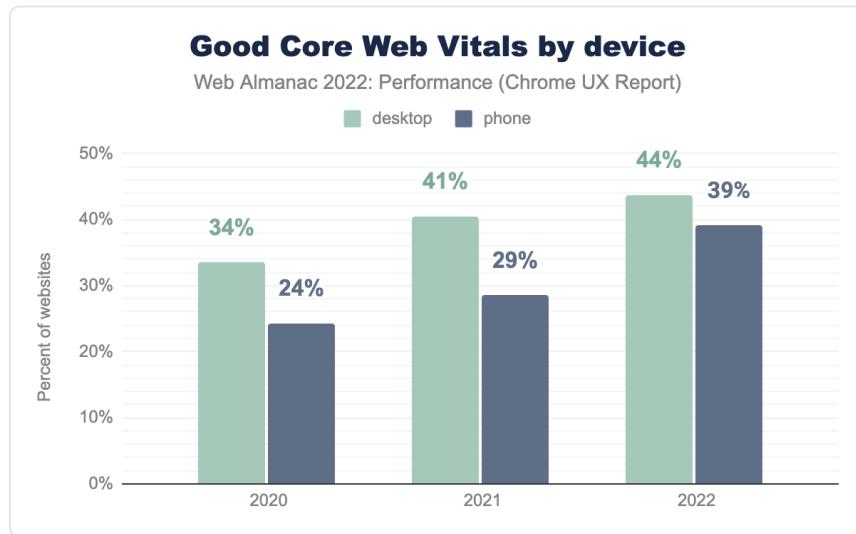


Figure 12.1. The percent of websites having good CWV, segmented by device and year.

In 2021, 29% of websites were assessed as having good CWV for mobile users. This was a significant step up from 2020, representing a 5 percentage point increase. However, the progress in 2022 was an even bigger leap forward, now with 39% of websites having good CWV on mobile—representing a further 10 point increase!

44% of websites have good CWV on desktop. While this is better than mobile, the rate of improvement for desktop experiences is not as rapid as mobile, so the gap is closing.

There are a few possible explanations for why mobile experiences tend to be worse than desktop. While the portability of a pocket-sized computer is a great convenience, it may have adverse effects on the user experience. As described in the Mobile Web chapter, the smaller form factor has an impact on the amount of processing power that can be packed in, which is further constrained by the high cost to own more powerful phones. Devices with poorer processing capabilities take longer to perform the computations needed to render a web page. The portability of these devices also means that they can be taken into areas with poor connectivity, which hinders websites' loading speeds. One final consideration is the way that developers decide how to build websites. Rather than creating a mobile-friendly version of the page, some websites may be serving desktop-sized images or unnecessary amounts of scripting

functionality. All of these things put mobile users at a disadvantage compared to desktop users and may help to explain why their CWV performance is lower.

Many more websites were assessed as having good CWV in 2022 relative to 2021. But how evenly distributed was that improvement across the web?

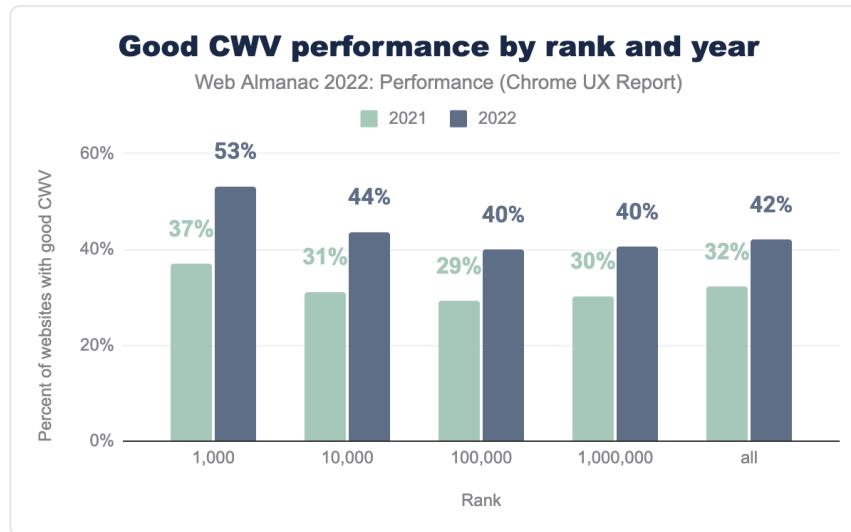


Figure 12.2. The percent of websites having good CWV, segmented by rank and year.

We segmented sites by their relative popularity (rank) and year, without distinguishing between desktop and mobile. What's interesting is that it seems websites across the board generally got more performant this year, regardless of their popularity. The top 1,000 most popular websites improved most significantly, a 16 percentage point increase to 53%, with all ranks improving by 10 points or more. The most popular websites also tend to have the best CWV experience, which is not too surprising if we assume that they have bigger engineering teams and budgets.

To better understand why mobile experiences have gotten so much better this year, let's dive deeper into each individual CWV metric.

Largest Contentful Paint (LCP)

Largest Contentful Paint⁴³¹ (LCP) is the time from the start of the navigation until the largest

431. <https://web.dev/lcp/>

block of content is visible in the viewport. This metric represents how quickly users are able to see what is likely the most meaningful content.

We say that a website has good LCP if at least 75 percent of all page views are faster than 2,500 ms. Of the three CWV metrics, LCP pass rates are the lowest, often making it the bottleneck to achieving good CWV assessments.

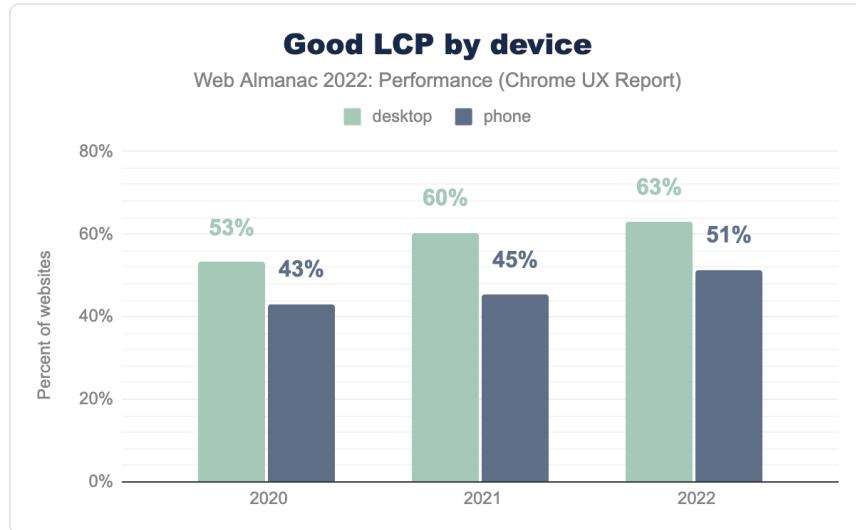


Figure 12.3. The percent of websites having good LCP, segmented by device and year.

This year, 51% of websites have good LCP experiences on mobile and 63% on desktop. LCP appears to be one of the major reasons why websites' mobile experiences have gotten so much better in 2022, having a 6 percentage point improvement this year.

Why did LCP improve so much this year? To help answer that, let's explore a couple of loading performance diagnostic metrics: TTFB and FCP.

Time to First Byte (TTFB)

Time to First Byte⁴³² (TTFB) is the time from the start of the navigation to the first byte of data returned to the client. It's our first step in the web performance checklist, representing the backend component of LCP performance, particularly network connection speeds and server response times.

432. <https://web.dev/ttfb/>

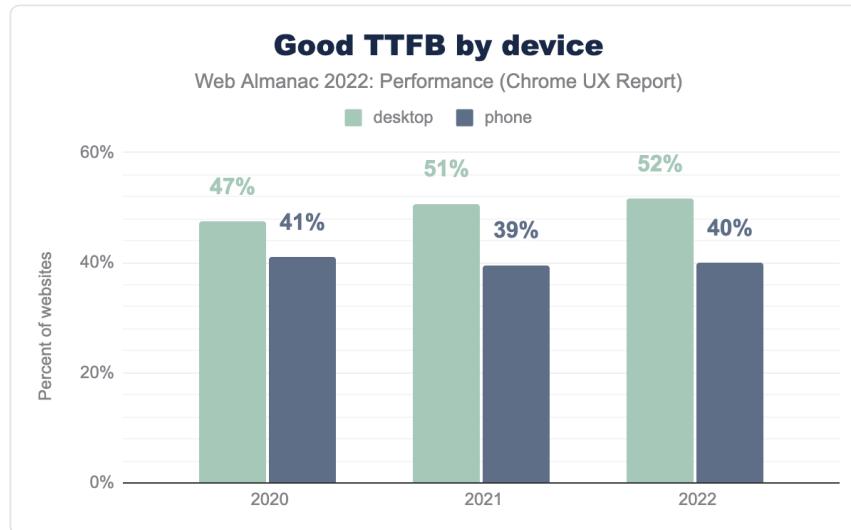


Figure 12.4. The percent of websites having good TTFB, segmented by device and year.

As an aside, note that earlier this year Chrome changed⁴³³ the threshold for “good” TTFB from 500 ms to 800 ms. In the chart above, all historical data is using this new threshold for comparison purposes.

With that in mind, the percentage of websites having good TTFB has not actually improved very much. In the past year, websites’ desktop and mobile experiences have gotten one percentage point better, which is nice but doesn’t account for the gains observed to LCP. While this doesn’t rule out improvements to the “needs improvement” and “poor” ends of the TTFB distribution, the “good” end is what matters most.

Another complication is that TTFB is still considered to be an experimental⁴³⁴ metric in CrUX. According to the CrUX documentation, TTFB does not factor in more advanced navigation types like pre-rendered and back/forward navigations. This is somewhat of a blind spot, so if there were improvements in these areas, they wouldn’t necessarily affect the TTFB results.

First Contentful Paint (FCP)

First Contentful Paint⁴³⁵ (FCP) is the time from the start of the request to the first meaningful content painted to the screen. In addition to TTFB, this metric can be affected by render-blocking content. The threshold for “good” FCP is 1,800 ms.

433. <https://developer.chrome.com/docs/crux/release-notes/#202204>

434. <https://developer.chrome.com/docs/crux/methodology/#experimental-metrics>

435. <https://web.dev/fcp/>

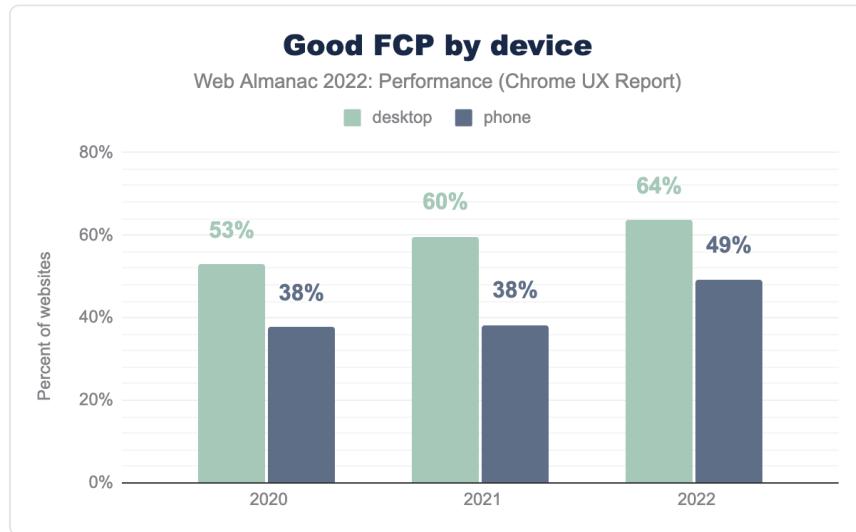


Figure 12.5. The percent of websites having good FCP, segmented by device and year.

FCP improved dramatically this year, with 49% of websites having good mobile experiences and 64% for desktop. This represents an 11 and 4 percentage point increase for mobile and desktop, respectively.

In the absence of TTFB data to the contrary, this indicates that there were major improvements to frontend optimizations, like eliminating render-blocking resources or better resource prioritization. However, as we'll see in the following sections, it seems like there may have been something else entirely to thank for the LCP improvements this year.

LCP metadata and best practices

These performance improvements may not actually be due to changes to the websites themselves. Changes to network infrastructure, operating systems, or browsers could also impact LCP performance at web-scale like this, so let's dig into some heuristics.

Render-blocking resources

A page is considered to have render-blocking resources if resources hold up the initial paint (or render) of the page. This is particularly likely for critical scripts and styles that are loaded over the network. Lighthouse includes an audit⁴³⁶ that checks for these resources, which we've run on

⁴³⁶. <https://web.dev/render-blocking-resources/>

the home page of each website in CrUX. You can learn more about how we test these pages in our Methodology.

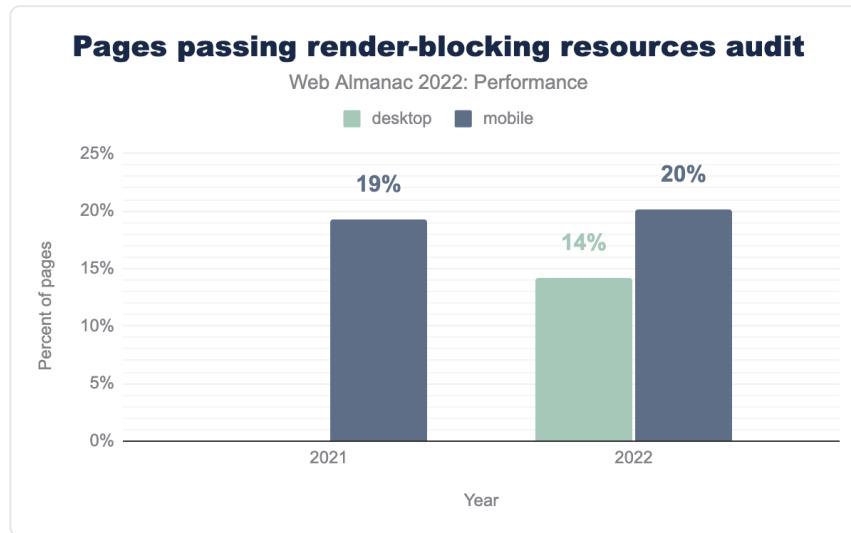


Figure 12.6. The percent of pages that pass the render-blocking Lighthouse audit , segmented by device and year.

Surprisingly, there was no dramatic improvement in the percent of pages that have render-blocking resources. Only 20% of mobile pages pass the audit, which is a mere 1 percentage point increase over last year.

2022 is the first year in which we have Lighthouse data for desktop. So while we're unable to compare it against previous years, it's still interesting to see that many fewer desktop pages pass the audit relative to mobile, in spite of the trend of desktop pages tending to have better LCP and FCP performance.

LCP content types

The LCP element can be a number of different types of content, like an image, a heading, or a paragraph of text.

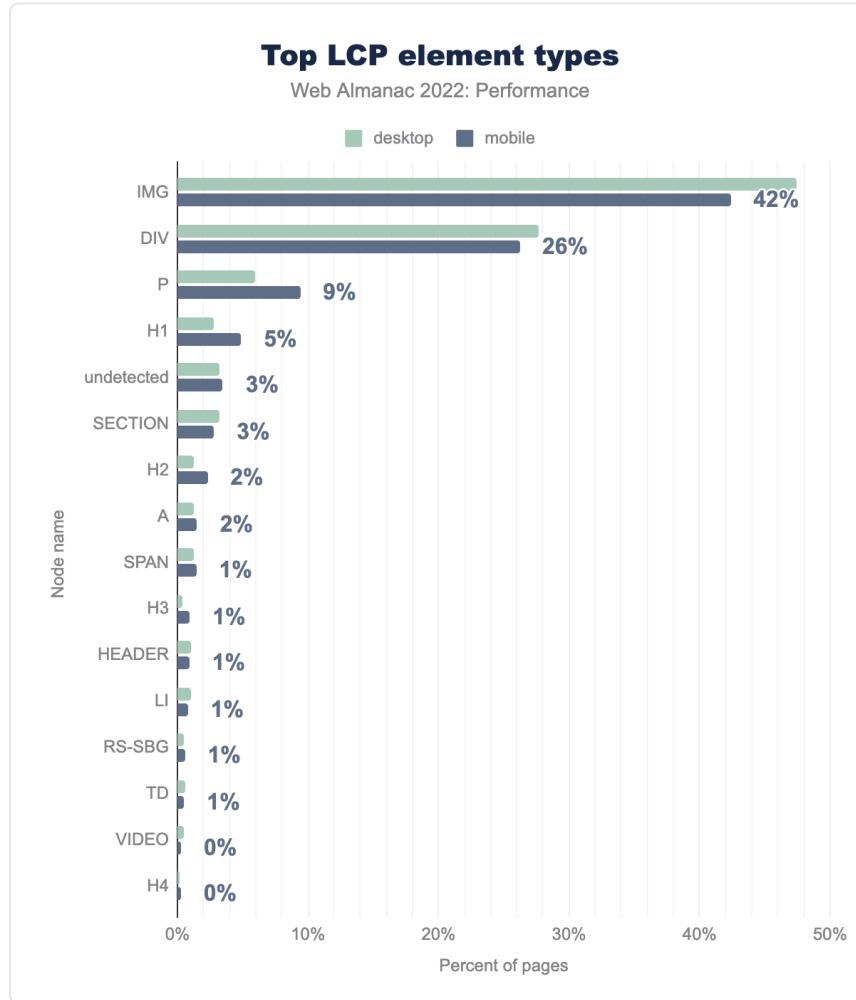


Figure 12.7. The percent of pages that have a given element as its LCP.

It's clear that images are the most common type of LCP content, with the `img` element representing the LCP on 42% of mobile pages. Mobile pages are slightly more likely to have heading and paragraph elements be the LCP than desktop pages, while desktop pages are more likely to have image elements as the LCP. One possible explanation is the way that mobile layouts—especially in portrait orientation—make images that are *not* responsive appear smaller, giving way to large blocks of text like headings and paragraphs to become the LCP elements.

The second most popular LCP element type is `div`. This is a generic HTML container that

could be used for text or styling background images. To help disambiguate how often these elements contain images or text, we can evaluate the `url` property of the LCP API⁴³⁷. According to the specification⁴³⁸, when this property is set, the LCP content must be an image.

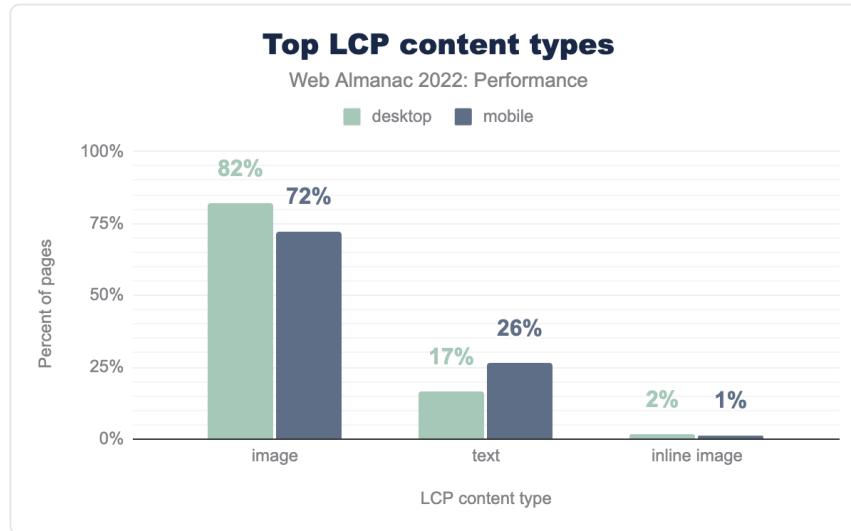


Figure 12.8. The percent of pages that use each type of LCP content.

We see that 72% of mobile pages and 82% of desktop pages have images as their LCP. For example, these images may be traditional `img` elements or CSS background images. This suggests that the vast majority of the `div` elements seen in the previous figure are images as well. 26% of mobile pages and 17% of desktop pages have text-based content as their LCP.

1% of pages actually use inline images as their LCP content. This is almost always a bad idea for a number of reasons, mostly around caching and complexity.

LCP prioritization

After the HTML document is loaded, there are two major factors that affect how quickly the LCP resource itself can be loaded: discoverability and prioritization. We'll explore LCP discoverability later, but first let's look at how LCP images are prioritized.

Images are not loaded at high priority by default, but thanks to the new Priority Hints⁴³⁹ API, developers can explicitly set their LCP images to load at high priority to take precedence over

437. <https://developer.mozilla.org/docs/Web/API/LargestContentfulPaint>

438. <https://www.w3.org/TR/largest-contentful-paint/#dom-largestcontentfulpaint-url>

non-essential resources.

0.03%

Figure 12.9. The percent of pages that use `fetchpriority=high` on their LCP element.

0.03% of pages use `fetchpriority=high` on their LCP elements. Counterproductively, a handful of pages actually *lower* the priority over their LCP images: 77 pages on mobile and 104 on desktop.

`fetchpriority` is still very new and not supported everywhere, but there's little to no reason why it shouldn't be in every developer's toolbox. Patrick Meenan⁴⁴⁰, who helped develop the API, describes it⁴⁴¹ as a "cheat code" given how easy it is to implement relative to the potential improvements.

LCP static discoverability

Ensuring an LCP image is discovered early is key to the browser loading it as soon as possible. Even the prioritization improvements that we discussed above, cannot help if the browser does not know it needs to load the resource until later.

An LCP image is considered to be *statically discoverable* if its source URL can be parsed directly from the markup sent by the server. This definition includes sources defined within `picture` or `img` elements as well as sources that are explicitly preloaded.

One caveat is that text-based LCP content is always statically discoverable based on this definition. However, text-based content may sometimes depend on client-side rendering or web fonts, so consider these results as lower bounds.

```

```

Custom lazy-loading techniques like the example above are one way that images are prevented from being statically discoverable, since they rely on JavaScript to update the `src` attribute. Client-side rendering may also obscure the LCP content.

440. <https://twitter.com/patmeenan>
 441. <https://twitter.com/patmeenan/status/1460276602479251457>

39%

Figure 12.10. The percent of mobile pages on which the LCP element was not statically discoverable.

39% of mobile pages have LCP elements that are not statically discoverable. This figure is even worse on desktop at 44% of pages, potentially as a consequence of the previous section's finding that LCP content is more likely to be an image on desktop pages.

This is the first year that we're looking at this metric, so we don't have historical data for comparison, but these results hint at a big opportunity to improve the load delay of LCP resources.

LCP preloading

When the LCP image is not statically discoverable, preloading⁴⁴² can be an effective way to minimize the load delay. Of course, it would be better if the resource was statically discoverable to begin with, but addressing that may require a complex rearchitecture of the way the page loads. Preloading is somewhat of a quick fix by comparison, as it can be implemented with a single HTTP header or `meta` tag.

0.56%

Figure 12.11. The percent of mobile pages that preload their LCP images.

Only about 1 in 200 mobile pages preload their LCP images. This figure falls to about 1 in 400 (0.25%) when we only consider pages whose LCP images are not statically discoverable.

Preloading statically discoverable images might be considered overkill, as the browser should already know about the image thanks to its preload scanner⁴⁴³. However, it can help load critical images earlier above other statically discoverable images that may be earlier in the HTML—for example header images or mega menu images. This is especially true for browsers that do not support `fetchpriority`.

These results show that the overwhelming majority of the web could benefit from making their LCP images more discoverable. Loading LCP images sooner, either by making them statically

442. <https://web.dev/preload-critical-assets/>

443. <https://web.dev/preload-scanner/>

discoverable or preloading them, can go a long way to improving LCP performance. But as with all things related to performance, always experiment to understand what's best for your specific site.

LCP initiator

When an LCP resource is not statically discoverable, there must be some other, more convoluted process by which it gets discovered.

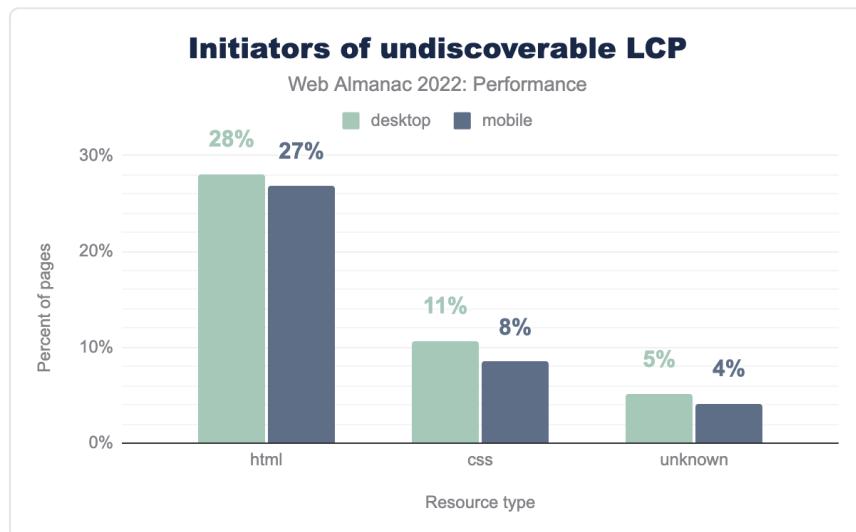


Figure 12.12. The percent of pages whose LCP is not statically discoverable and initiated from a given resource.

27% of mobile pages have LCP images that are discovered in the HTML after the preload scanner has already run, typically due to script-based lazy-loading or client-side rendering.

8% of mobile pages depend on an external stylesheet for their LCP resource, for example using the `background-image` property. This adds a link in the resource's critical request chain and may further complicate LCP performance if the stylesheet is loaded cross-origin.

4% of mobile pages have an undiscoverable LCP initiator whose type we're unable to detect. These may be a combination of HTML and CSS initiators.

Both script- and style-based discoverability issues are bad for performance, but their effects can be mitigated with preloading.

LCP lazy-loading

Lazy-loading is an effective performance technique to delay when non-critical resources start loading, usually until they're in or near the viewport. With precious bandwidth and rendering power freed up, the browser can load critical content earlier in the page load. A problem arises when lazy-loading is applied to the LCP image itself, because that prevents the browser from loading it until much later.

A large, bold, blue percentage value of "9.8%" centered on the page.

Figure 12.13. The percent of mobile pages having image-based LCP that set `loading=lazy` on it.

Nearly 1 in 10 of pages with `img`-based LCP are using the native `loading=lazy` attribute. Technically, these images are statically discoverable, but the browser will need to wait to start loading them until it's laid out the page to know whether they will be in the viewport. LCP images are always in the viewport, by definition, so in reality none of these images should have been lazy-loaded. For pages whose LCP varies by viewport size or initial scroll position from deep-linked navigations, it's worth testing whether eagerly loading the LCP candidate results in better overall performance.

Note that the percentages in this section are out of only those pages in which the `img` element is the LCP, not all pages. For reference, recall that this accounts for 42% of pages.

A large, bold, blue percentage value of "8.8%" centered on the page.

Figure 12.14. The percent of mobile pages having image-based LCP that use custom lazy-loading on it.

As we showed earlier, one way that sites might polyfill the native lazy-loading behavior is to assign the image source to a `data-src` attribute and include an identifier like `lazyload` in the class list. Then, a script will watch the positions of images with this class name relative to the viewport, and swap the `data-src` value for the native `src` value to trigger the image to start loading.

Nearly as many pages are using this kind of custom lazy-loading behavior as native lazy-loading, at 8.8% of pages with `img`-based LCP.

Beyond the adverse performance effects of lazy-loading LCP images, native image lazy-loaded

is supported⁴⁴⁴ by all major browsers, so custom solutions may be adding unnecessary overhead. In our opinion, while some custom solutions may provide more granular control over when images load, developers should remove these extraneous polyfills and defer to the user agent's native lazy-loading heuristics.

Another benefit to using native lazy-loading is that browsers like Chrome are experimenting with using heuristics to ignore the attribute on probable LCP candidates⁴⁴⁵. This is only possible with native lazy-loading, so custom solutions would not benefit from any improvements in this case.

18%

Figure 12.15. The percent of mobile pages having image-based LCP that use native or custom lazy-loading on it.

Looking at pages that use either technique, 18% of pages with `img`-based LCP are unnecessarily delaying the load of their most important images.

Lazy-loading is a good thing when used correctly, but these stats strongly suggest that there's a major opportunity to improve performance by removing this functionality from LCP images in particular.

WordPress was one of the pioneers of native lazy-loading adoption, and between versions 5.5 and 5.9, it didn't actually omit the attribute from LCP candidates. So let's explore the extent to which WordPress is still contributing to this anti-pattern.

72%

Figure 12.16. The percent of mobile pages using native lazy-loading on their LCP image that also use WordPress.

According to the CMS chapter, WordPress is used by 35% of pages. So it's surprising to see that 72% of pages that use native lazy-loading on their LCP image are using WordPress, given that a fix has been available since January 2022 in version 5.9. One theory that needs more investigation is that plugins may be circumventing the safeguards built into WordPress core by injecting LCP images onto the page with the lazy-loading behavior.

444. <https://caniuse.com/loading-lazy-attr>

445. <https://bugs.chromium.org/p/chromium/issues/detail?id=996963>

54%

Figure 12.17. The percent of mobile pages using custom lazy-loading on their LCP image that also use WordPress.

Similarly, a disproportionately high percentage of pages that use custom lazy-loading are built with WordPress at 54%. This hints at a wider issue in the WordPress ecosystem about lazy-loading overuse. Rather than being a fixable bug localized to WordPress core, there may be hundreds or thousands of separate themes and plugins contributing to this anti-pattern.

LCP size

A major factor in the time it takes to load the LCP resource is its size over the wire. Larger resources will naturally take longer to load. So for image-based LCP resources, how large are they?

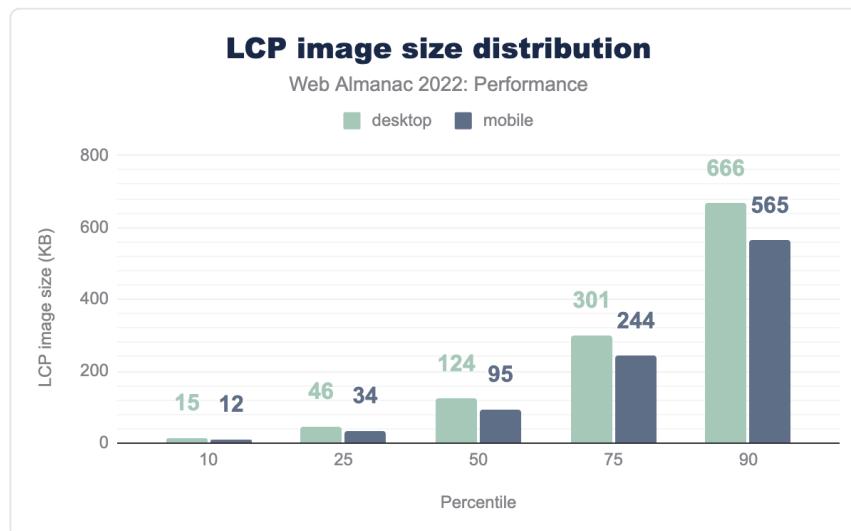


Figure 12.18. Distribution of the size of image-based LCP resources.

The median LCP image on mobile is 95 KB. To be honest, we expected much worse!

Desktop pages tend to have larger LCP images across the distribution, with a median size of 124 KB.

114,285 KB

Figure 12.19. The size of the largest LCP image.

We also looked at the largest LCP image sizes and found a 68,607 KB image on desktop and 114,285 KB image on mobile. While it can be fun to look at how obscenely large these outliers are, let's keep in mind the unfortunate reality that these are active websites visited by real users. Data isn't always free, and performance problems like these start to become accessibility problems for users on metered mobile data plans. These are also sustainability problems considering how much energy is wasted loading blatantly oversized images like these.

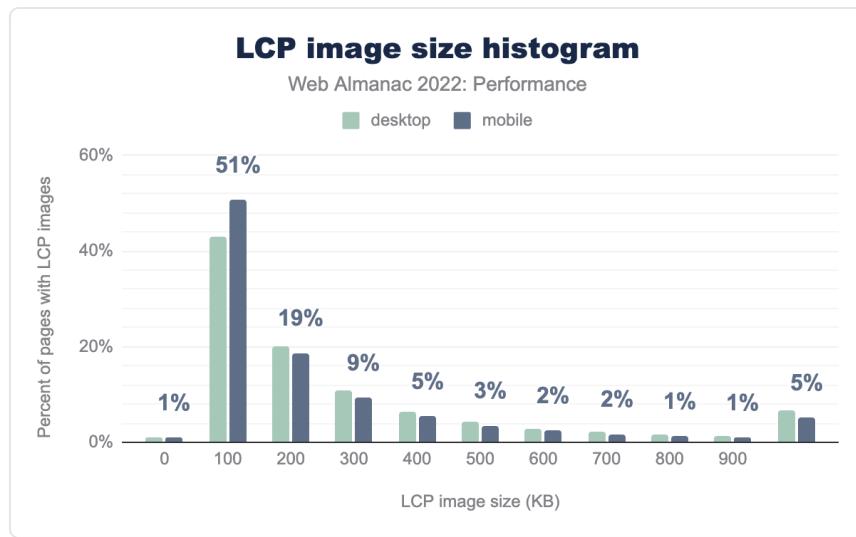


Figure 12.20. Histogram of image-based LCP sizes.

Looking at it a different way, the figure above shows the distribution as a histogram in 100 KB increments. This view makes it clearer to see how LCP image sizes fall predominantly in the sub-200 KB range. We also see that 5% of LCP images on mobile are greater than 1,000 KB in size.

How large an LCP image should *optimally* be depends on many factors. But the fact that 1 in 20 websites are serving megabyte-sized images to our 360px-wide mobile viewports clearly highlights the need for site owners to embrace responsive images⁴⁴⁶. For more analysis on this

446. https://developer.mozilla.org/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images

topic, refer to the Media and Mobile Web chapters.

LCP format

Choice of LCP image format can have significant effects on its byte size and ultimately its loading performance. WebP and AVIF are two relatively newer formats that are found to be more efficient than traditional formats like JPG (or JPEG) and PNG.

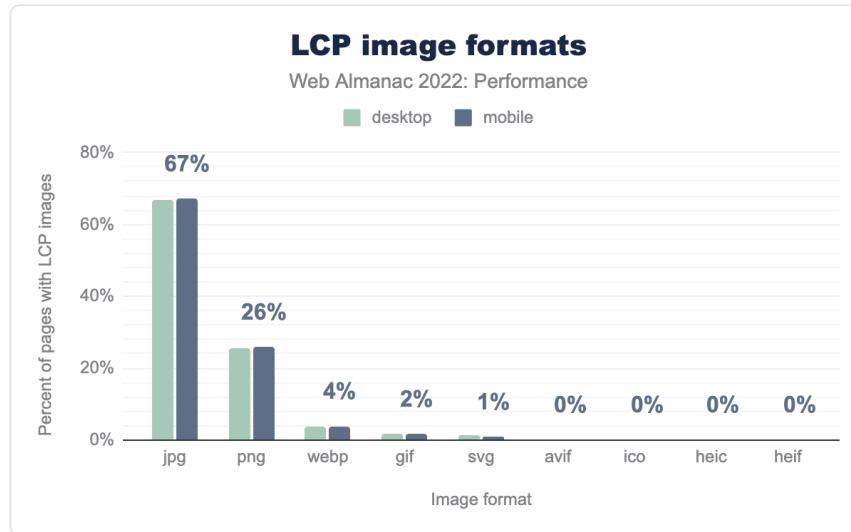


Figure 12.21. The percent of pages that use a given format for their LCP images.

According to the Media chapter, the JPG format makes up about 40% of all images loaded on mobile pages. However, JPG makes up 67% of all LCP images on mobile, which is 2.5x more common than PNG at 26%. These results may hint at a tendency for pages to use photographic-quality images as their LCP resource rather than digital artwork, as photographs tend to compress better as JPG compared to PNG, but this is just speculation.

4% of pages with image-based LCP use WebP. This is good news for image efficiency, however less than 1% are using AVIF. While AVIF may compress even better than WebP, it's not supported in all modern browsers, which explains its low adoption. On the other hand, WebP is supported in all modern browsers, so its low adoption represents a major opportunity to optimize LCP images and their performance.

LCP image optimization

The previous section looked at the popularity of various image formats used by LCP resources. Another way that developers can make their LCP resources smaller and load more quickly is to utilize efficient compression settings. The JPG format can be lossily compressed to eke out unnecessary bytes without losing too much image quality. However, some JPG images may not be compressed enough.

Lighthouse includes an audit⁴⁴⁷ that will measure the byte savings from setting JPGs to compression level 85. If the image is more than 4 KB smaller as a result, the audit fails and it's considered an opportunity for optimization.

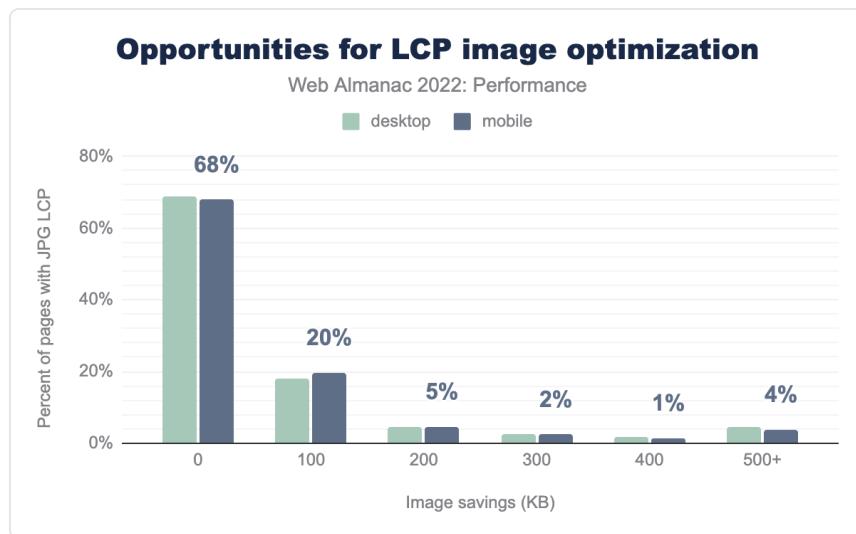


Figure 12.22. Histogram of byte savings for JPG-based LCP images.

Of the pages whose LCP images are JPG-based and flagged by Lighthouse, 68% of them do not have opportunities to improve the LCP image via lossy compression alone. These results are somewhat surprising and suggest that the majority of "hero" JPG images use appropriate quality settings. That said, 20% of these pages could save as much as 100 KB and 4% can save 500 KB or more. Recall that the majority of LCP images are under 200 KB, so this is some serious savings!

⁴⁴⁷ <https://web.dev/uses-optimized-images/>

LCP host

In addition to the size and efficiency of the LCP image itself, the server from which it loads can also have an impact on its performance. Loading LCP images from the same origin as the HTML document tends to be faster because the open connection can be reused.

However, LCP images may be loaded from other origins, like asset domains and image CDNs. When this happens, setting up the additional connection can take valuable time away from the LCP allowance.

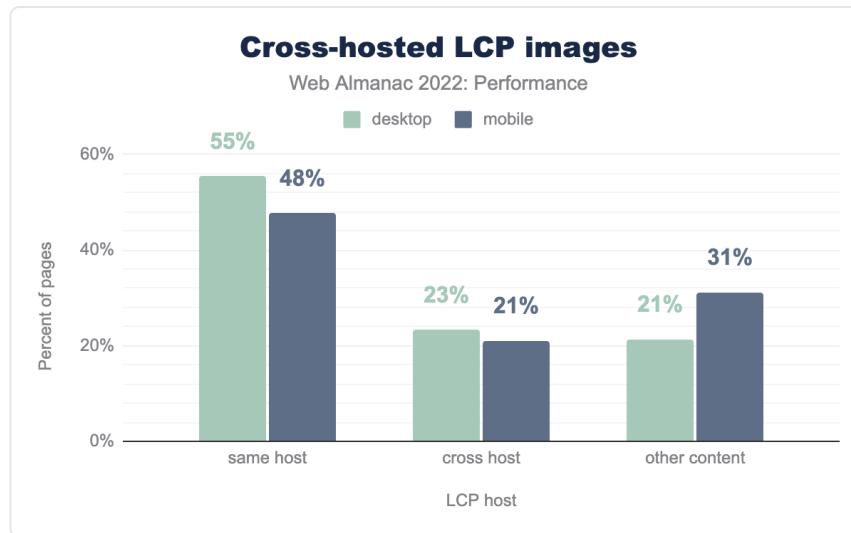


Figure 12.23. Cross-hosted LCP images

One in five mobile pages cross-host their LCP images. The time to set up the connection to these third-party hosts could add unnecessary delays to the LCP time. It's best practice to self-host LCP images on the same origin as the document, whenever possible. Resource hints⁴⁴⁸ could be used to preconnect to the LCP origin—or better yet, preload the image itself—but these techniques are not very widely adopted.

LCP conclusions

LCP performance has improved significantly this year, especially for mobile users. While we don't have a definitive answer for why that happened, the data presented above does give us a few clues.

⁴⁴⁸. <https://almanac.httparchive.org/en/2021/resource-hints>

What we've seen so far is that render-blocking resources are still quite prevalent, very few sites are using advanced prioritization techniques, and more than a third of LCP images are not statically discoverable. Without concrete data to suggest that site owners or large publishing platforms are conceretedly optimizing these aspects of their LCP performance, other places to look are optimizations at the OS or browser level.

According to a Chromium blog post⁴⁴⁹ in March 2022, loading performance on Android improved by 15%. The post doesn't go into too much detail, but it credits the improvement to "*prioritizing critical navigation moments on the browser user interface thread.*" This may help explain why mobile performance outpaced desktop performance in 2022.

The six percentage point improvement to LCP this year can only happen when hundreds of thousands of websites' performance improves. Putting aside the tantalizing question of how that happened, let's take a moment to appreciate that user experiences on the web *are* getting better. It's hard work, but improvements like these make the ecosystem healthier and are worth celebrating.

Cumulative Layout Shift (CLS)

Cumulative Layout Shift⁴⁵⁰ (CLS) is a layout stability metric that represents the amount that content unexpectedly moves around on the screen. We say that a website has good CLS if at least 75% of all navigations across the site have a score of 0.1 or less.

449. <https://blog.chromium.org/2022/03/a-new-speed-milestone-for-chrome.html#:~:text=Chrome%20continues%20to%20get%20>

450. chrome.html#:~:text=Chrome%20continues%20to%20get%20faster%20on%20Android%20as%20well.%20Loading%20a%20page%20now%20takes%2015%25%20less%20time%2C%20thank

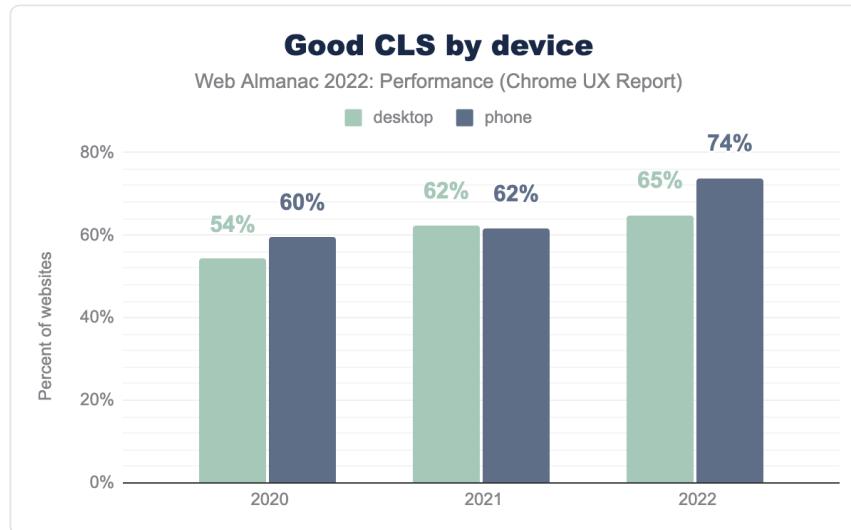


Figure 12.24. Good CLS by device

This year, the percentage of websites with “good” CLS improved significantly on mobile devices, going from 62% to 74%. CLS on desktop improved by 3 percentage points to 65%.

While LCP is the bottleneck for most sites to be assessed as having good CWV overall, there’s no doubt that the major improvements to mobile CLS this year have had a positive effect on the CWV pass rates.

What happened to improve mobile CLS by such a significant margin? One likely explanation is Chrome’s new and improved back/forward cache⁴⁵¹ (bfcache), which was released in version 96 in mid-November 2021. This change enabled eligible pages to be pristinely restored from memory during back and forward navigations, rather than having to “start over” by fetching resources from the HTTP cache—or worse, over the network—and reconstructing the page.

⁴⁵¹. <https://web.dev/bfcache/>

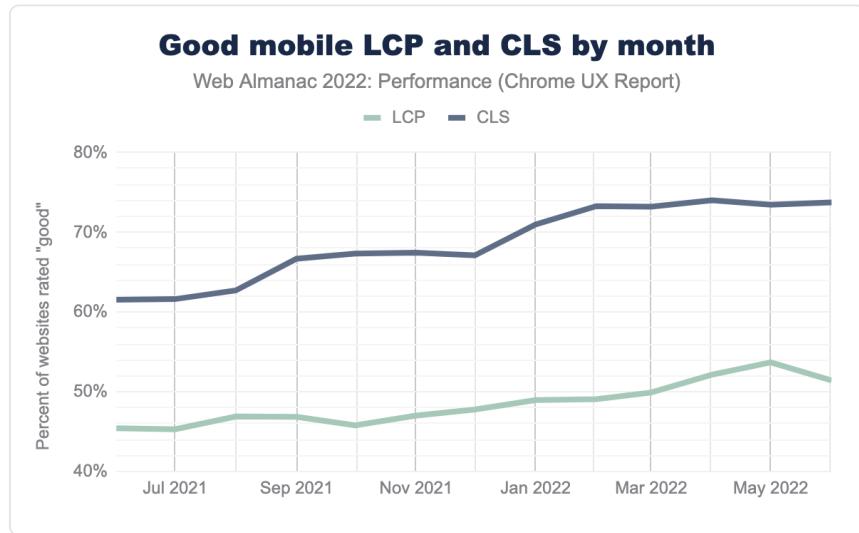


Figure 12.25. Monthly timeseries of the percent of websites having good mobile LCP and CLS.

This chart shows LCP and CLS performance over time at monthly granularity. Over a two month period starting in January 2022, after Chrome released the bfcache update, the percent of websites having good CLS started to climb much more quickly than before.

But how did bfcache improve CLS so much? Due to the way Chrome instantly restores the page from memory, its layout is settled and unaffected by any of the initial instability that typically occurs during loading.

One theory why LCP experiences didn't improve as dramatically is that back/forward navigations were already pretty fast thanks to standard HTTP caching. Remember, the threshold for "good" LCP is 2.5 seconds, which is pretty generous assuming any critical resources would already be in cache, and there are no bonus points for making "good" experiences even faster.

CLS metadata and best practices

Let's explore how much of the web is adhering to CLS best practices.

Explicit dimensions

The most straightforward way to avoid layout shifts is to reserve space for content by setting dimensions, for example using `height` and `width` attributes on images.

72%

Figure 12.26. The percent of mobile pages that fail to set explicit dimensions on at least one image.

72% of mobile pages have unsized images. This stat alone doesn't give the full picture, because unsized images don't always result in user-perceived layout shifts, for example if they load outside of the viewport. Still, it's a sign that site owners may not be closely adhering to CLS best practices.

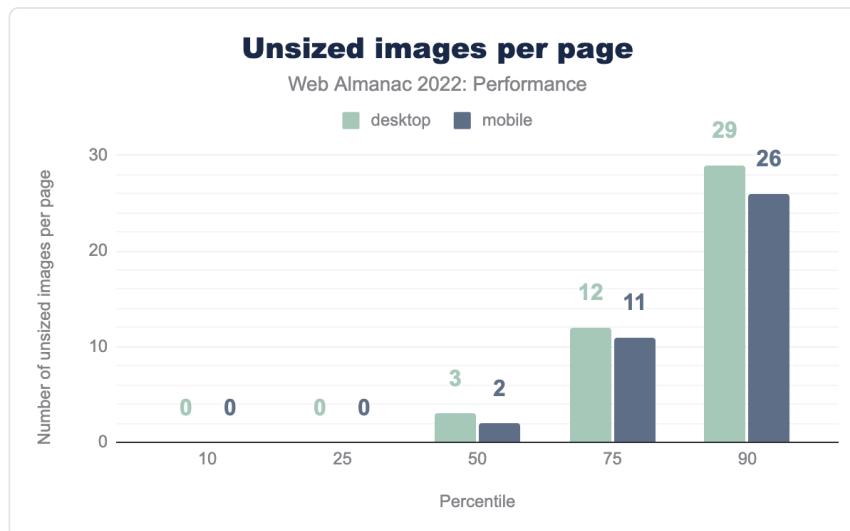


Figure 12.27. Distribution of the number of unsized images per page.

The median web page has 2 unsized images and 10% of mobile pages have at least 26 unsized images.

Having any unsized images on the page can be a liability for CLS, but perhaps a more important factor is the size of the image. Large images contribute to bigger layout shifts, which make CLS worse.

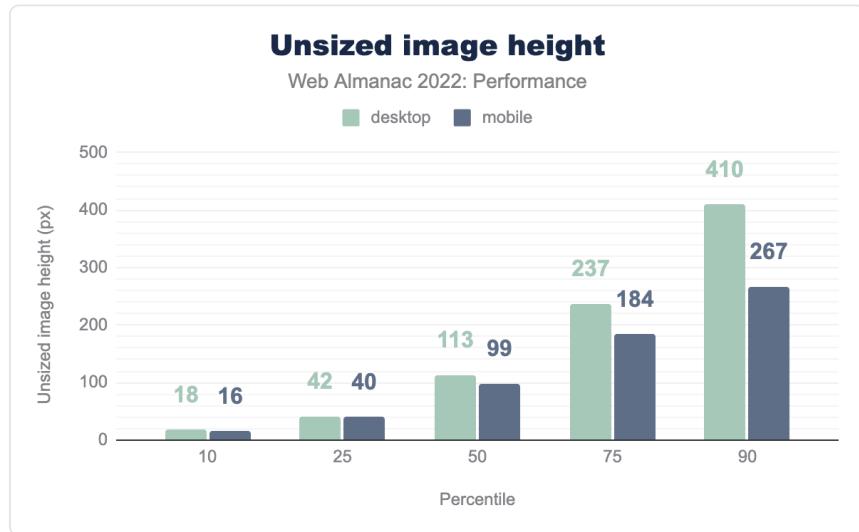


Figure 12.28. Distribution of the heights of unsized images.

The median unsized image on mobile has a height of 99px. Given that our test devices have a mobile viewport height of 512px, that's about 20% of the viewport area that would shift down, assuming full-width content. Depending on where that image is in the viewport when it loads, it could cause a layout shift score⁴⁵² of at most 0.2, which more than exceeds the 0.1 threshold for "good" CLS.

Desktop pages tend to have larger unsized images. The median unsized image on desktop is 113px tall and the 90th percentile has a height of 410px.

4,048,234,137,947,990,000px

Figure 12.29. The height of the largest unsized image.

In what we can only hope is either a measurement error or a seriously mistaken web developer, the largest unsized image that we found is an incredible 4 quintillion pixels tall. That image is so big it could stretch from the Earth to the moon... three million times. Even if that is some kind of one-off mistake, the next biggest unsized image is still 33,554,432 pixels tall. Either way, that's a big layout shift.

452. <https://web.dev/cls/#layout-shift-score>

Animations

Some non-composited⁴⁵³ CSS animations can affect the layout of the page and contribute to CLS. The best practice⁴⁵⁴ is to use `transform` animations instead.

38%

Figure 12.30. The percent of mobile pages that have non-composited animations.

38% of mobile pages use these layout-altering CSS animations and risk making their CLS worse. Similar to the unused images issue, what matters most for CLS is the degree to which the animations affect the layout relative to the viewport.

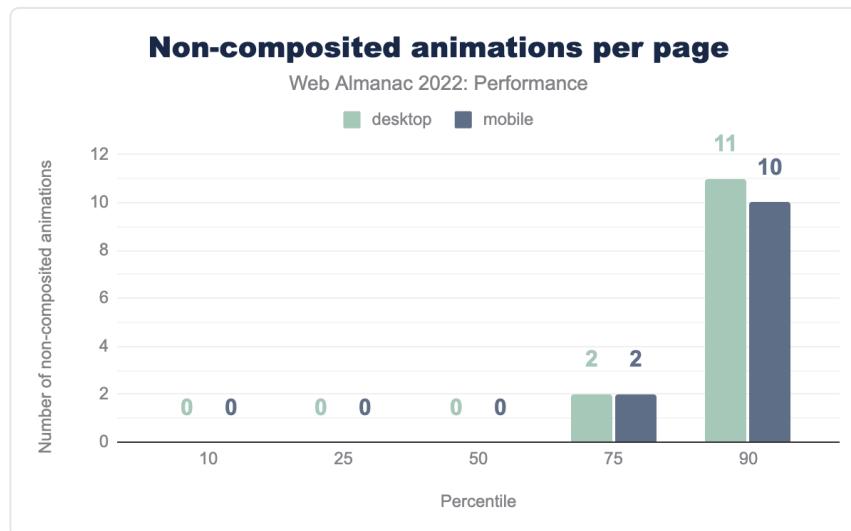


Figure 12.31. Distribution of the number of non-composited animations per page.

The distribution above shows that most pages don't use these types of animations, and the ones that do only use it sparingly. At the 75th percentile, pages use them twice.

453. <https://web.dev/non-composited-animations/>

454. <https://web.dev/optimize-cls/#animations-%F0%9F%8F%83%E2%80%8D%E2%99%80%EF%B8%8F>

Fonts

In the page load process, it can take some time for the browser to discover, request, download, and apply a web font. While this is all happening, it's possible that text has already been rendered on a page. If the web font isn't yet available, the browser can default to rendering text in a system font. Layout shifts happen when the web font becomes available and existing text, rendered in a system font, switches to the web font. The amount of layout shift caused depends on how different the fonts are from each other.

82%

Figure 12.32. The percent of mobile pages that use web fonts.

82% of pages use web fonts, so this section is highly relevant to most site owners.

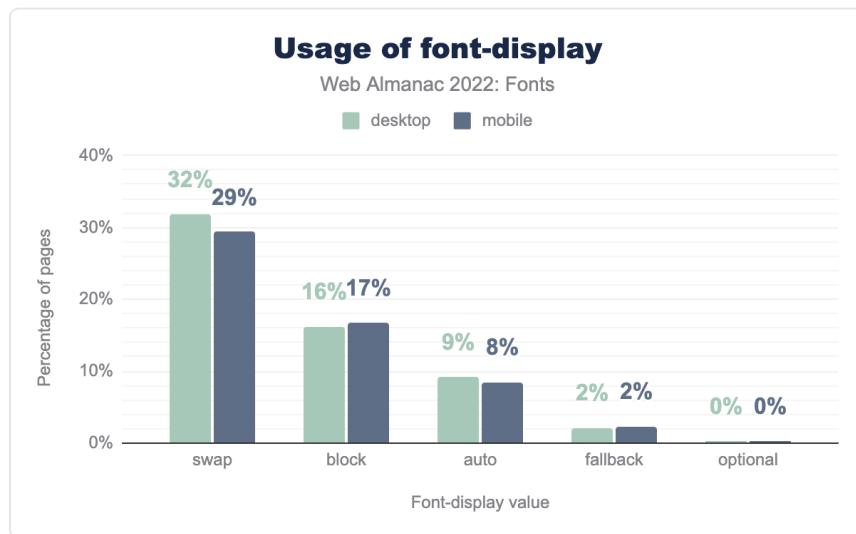


Figure 12.33. Adoption of `font-display` values.

One way to avoid font-induced layout shifts is to use `font-display: optional`, which will never swap in a web font after the system text has already been shown. However, as noted by the Fonts chapter, less than 1% of pages are taking advantage of this directive.

Even though `optional` is good for CLS, there are UX tradeoffs. Site owners might be willing to have some layout instability or a noticeable flash of unstyled text (FOUT) if it means that their

preferred font can be displayed to users.

Rather than hiding the web fonts, another strategy to mitigate CLS is to load them as quickly as possible. Doing so would, if all goes well, display the web font before the system text is rendered.

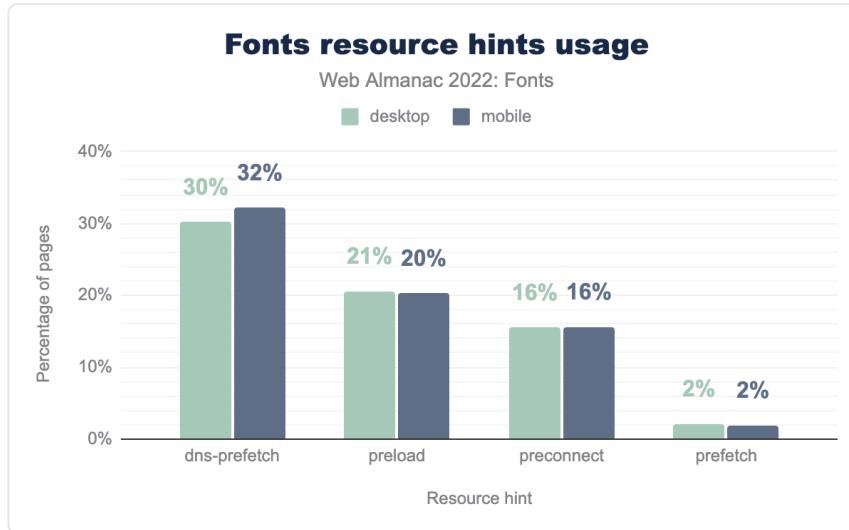


Figure 12.34. Adoption of resource hints for font resources.

According to the Fonts chapter, 20% of mobile pages are preloading their web fonts. One challenge with preloading the font is that the exact URL may not be known upfront, for example if using a service like Google Fonts. Preconnecting to the font host is the next best option for performance, but only 16% of pages are using that, which is half as many pages that use the less-performant option to prefetch the DNS.

bfcache eligibility

We've shown how impactful bfcache can be for CLS, so it's worth considering eligibility as a somewhat indirect best practice.

The best way to tell if a given page is eligible for bfcache is to test it in DevTools⁴⁵⁵. Unfortunately, there are over 100 eligibility criteria⁴⁵⁶, many of which are hard or impossible to measure in the lab. So rather than looking at bfcache eligibility as a whole, let's look at a few criteria that are more easily measurable to get a sense for the lower bound of eligibility.

455. <https://web.dev/bfcache/#test-to-ensure-your-pages-are-cacheable>

456. https://docs.google.com/spreadsheets/d/1liOpo_ETjAlybpaSX5rW_IUN62upQhY0tH4pR5UPt60/edit?usp=sharing

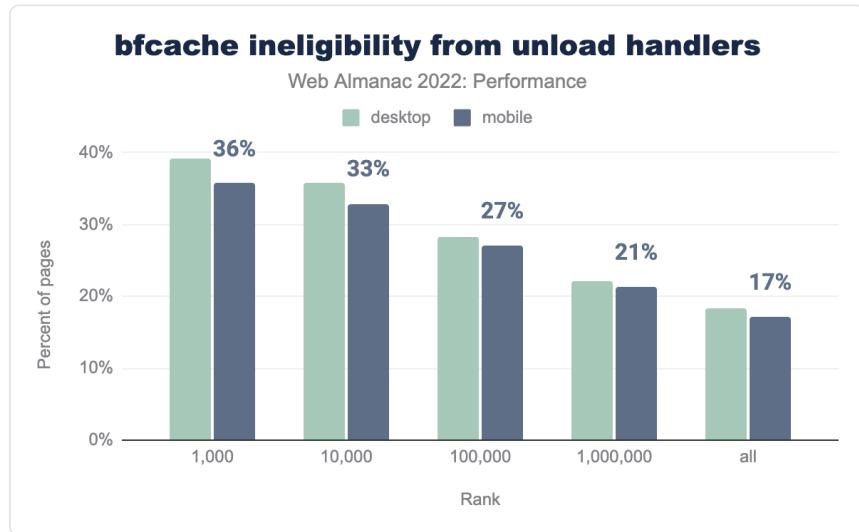


Figure 12.35. Usage of `unload` by site rank.

The `unload` event is a discouraged way to do work when the page is in the process of going away (unloading). Besides there being better ways⁴⁵⁷ to do that, it's also one way to make your page ineligible for bfcache.

17% of all mobile pages set this event handler, however the situation worsens the more popular the website is. In the top 1k, 36% of mobile pages are ineligible for bfcache for this reason.

457. <https://web.dev/bfcache/#never-use-the-unload-event>

bfcache ineligibility from Cache-Control: no-store

Web Almanac 2022: Performance

desktop mobile

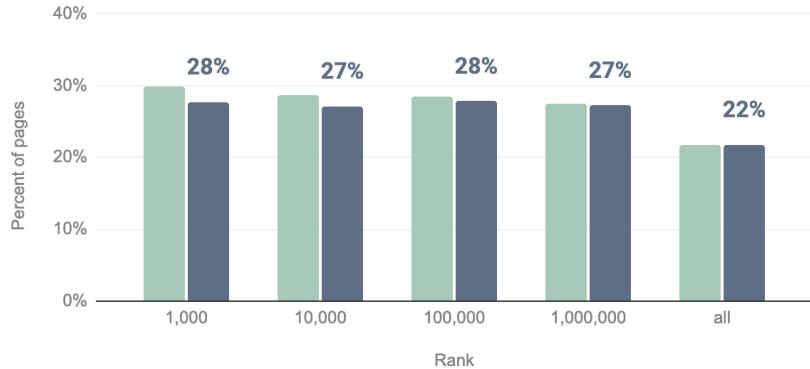


Figure 12.36. Usage of `Cache-Control: no-store` by site rank.

The `Cache-Control: no-store` header tells user agents never to cache a given resource. When set on the main HTML document, this makes the entire page ineligible for bfcache.

22% of all mobile pages set this header, and 28% of mobile pages in the top 1k. This and the `unload` criteria are not mutually exclusive, so combined we can only say that at least 22% of mobile pages are ineligible for bfcache.

To reiterate, it's not that these things cause CLS issues. However, fixing them may make pages eligible for bfcache, which we've been shown to be an indirect yet powerful tool for improving layout stability.

CLS conclusions

CLS is the CWV metric that improved the most in 2022 and it appears to have had a significant impact on the number of websites that have "good" overall CWV.

The cause of this improvement seems to come down to Chrome's launch of bfcache, which is reflected in the January 2022 CrUX dataset. However, at least a fifth of sites are ineligible for this feature due to aggressive `no-store` caching policies or discouraged use of the `unload` event listener. Correcting these anti-patterns is CLS's "one weird trick" to improve performance.

There are other, more direct ways site owners can improve their CLS. Setting `height` and

`width` attributes on images is the most straightforward one. Optimizing how animations are styled and how fonts load are two other—admittedly more complex—approaches to consider.

First Input Delay (FID)

First Input Delay⁴⁵⁸ (FID) measures the time from the first user interaction like a click or tap to the time at which the browser begins processing the corresponding event handlers. A website has “good” FID if at least 75 percent of all navigations across the site are faster than 100 ms.

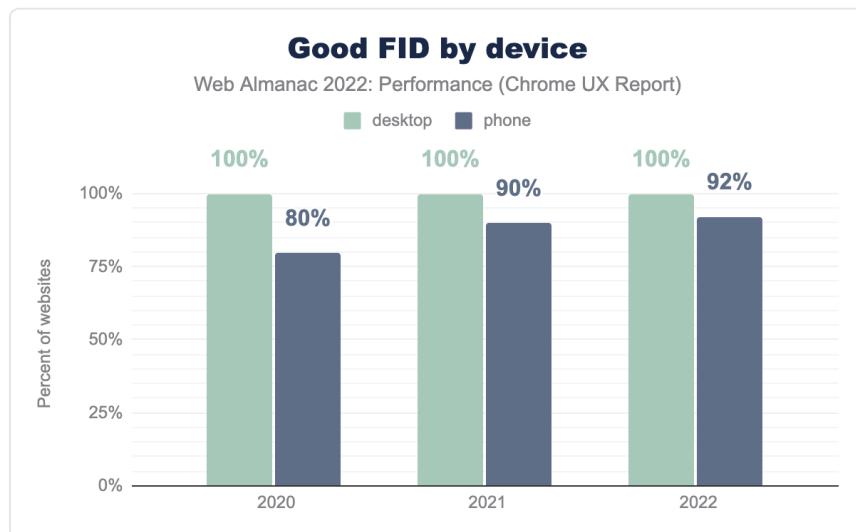


Figure 12.37. Good FID by device

Effectively all websites have “good” FID for desktop users, and this trend has held firm over the years. Mobile FID performance is also exceptionally fast, with 92% of websites having “good” FID, a slight improvement over last year.

While it’s great that so many websites have good FID experiences, developers need to be careful not to become too complacent. Google has been experimenting with a new responsiveness metric⁴⁵⁹ that could end up replacing FID, which is especially important because sites tend to perform much worse on this new metric than FID.

458. <https://web.dev/fid/>

459. <https://web.dev/better-responsiveness-metric/>

FID metadata and best practices

Let's dig deeper into the ways that responsiveness can be improved across the web.

Disabling double-tap to zoom

Some mobile browsers, including Chrome, wait at least 250 ms before handling tap inputs to make sure users aren't attempting to double-tap to zoom⁴⁶⁰. Given that the threshold for "good" FID is 100 ms, this behavior makes it impossible to pass the assessment.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

There's an easy fix, though. Including a `meta` viewport tag in the document head like the one above will prompt the browser to render the page as wide at the device width, which makes text content more legible and eliminates the need for double-tap to zoom.

This is one of the quickest, easiest, and least intrusive ways to meaningfully improve responsiveness and all mobile pages should be setting it.



Figure 12.38. The percent of mobile pages that do not set a viewport `meta` tag.

7.3% of mobile pages fail to set the `meta` viewport directive. Recall that about 8% of mobile websites fail to meet the threshold for "good" FID. This is a significant proportion of the web that is needlessly slowing down their sites' responsiveness. Correcting this may very well mean the difference between failing and passing the FID assessment.

Total Blocking Time (TBT)

Total Blocking Time⁴⁶¹ (TBT) is the time between the First Contentful Paint⁴⁶² (FCP) and Time to Interactive⁴⁶³ (TTI), representing the total amount of time that the main thread was blocked and unable to respond to user inputs.

460. <https://developer.chrome.com/blog/300ms-tap-delay-gone-away/>

461. <https://web.dev/tbt/>

462. <https://web.dev/fcp/>

463. <https://web.dev/tti/>

TBT is often used as a lab-based proxy for FID, due to the challenges of realistically simulating user interactions in synthetic tests.

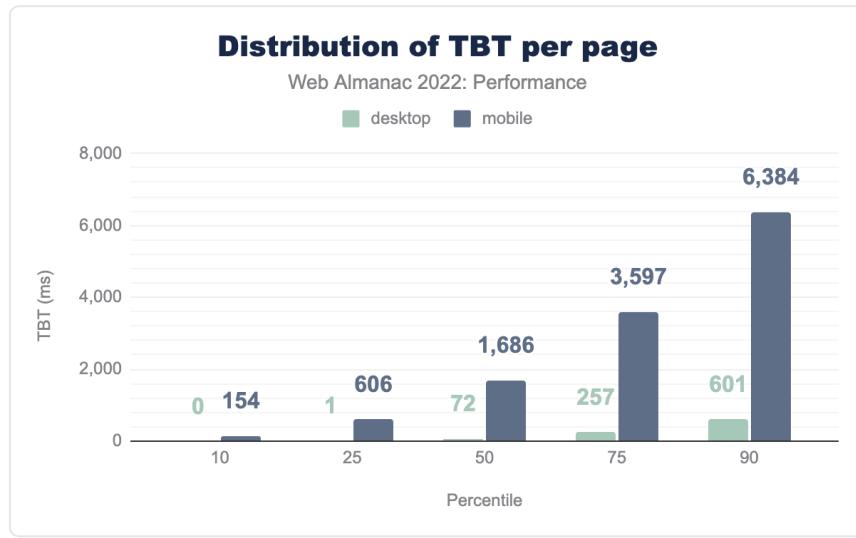


Figure 12.39. Distribution of lab-based TBT per page.

Note that these results are sourced from the lab-based TBT performance of pages in the HTTP Archive dataset. This is an important distinction because for the most part we've been looking at real-user performance data from the CrUX dataset.

With that in mind, we see that mobile pages have significantly worse TBT than desktop pages. This isn't surprising given that our Lighthouse mobile environment is intentionally configured to run with a throttled CPU in a way that emulates a low-end mobile device. Nevertheless, the results show that the median mobile page has a TBT of 1.7 seconds, meaning that if this were a real-user experience, no taps within 1.7 seconds of FCP would be responsive. At the 90th percentile, a user would have to wait 6.3 seconds before the page became responsive.

Despite the fact that these results come from synthetic testing, they're based on the actual JavaScript served by real websites. If a real user on similar hardware tried to access one of these sites, their TBT might not be too different. That said, the key difference between TBT and FID is that the latter relies on the user actually interacting with the page, which they can do at any time before, during, or after the TBT window, all leading to vastly different FID values.

Long tasks

Long tasks⁴⁶⁴ are periods of script-induced CPU activity at least 50 ms long that prevent the main thread from responding to input. Any long task is liable to cause responsiveness issues if a user attempts to interact with the page at that time.

Note that, like the TBT analysis above, this section draws from lab-based data. As a result, we're only able to measure long tasks during the page load observation window, which starts when the page is requested and ends after 60 seconds or 3 seconds of network inactivity, whichever comes first. A real user may experience long tasks throughout the entire lifetime of the page.

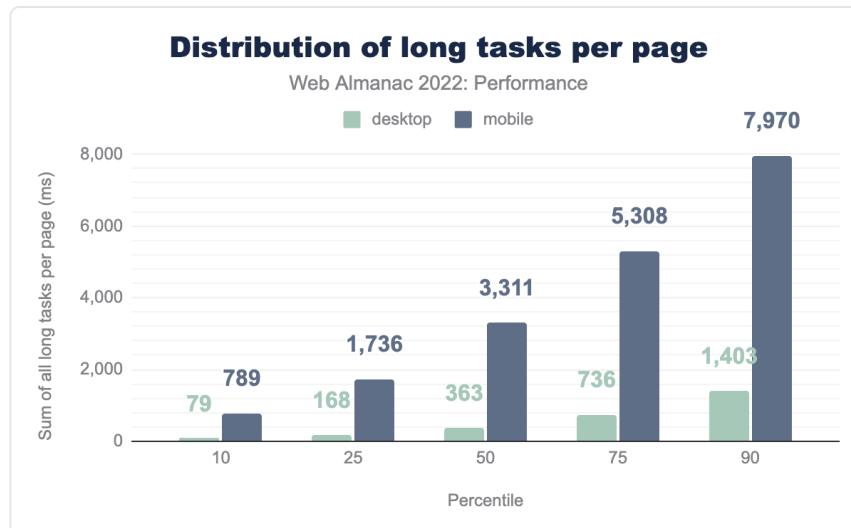


Figure 12.40. Distribution of lab-based long tasks per page.

The median mobile web page has 3.3 seconds-worth of long tasks, compared to only 0.4 seconds for desktop pages. Again, this shows the outsized effects of CPU speed on responsiveness heuristics. At the 90th percentile, mobile pages have at least 8.0 seconds of long tasks.

It's also worth noting that these results are significantly higher than the distribution of TBT times. Remember that TBT is bounded by FCP and TTI and FID is dependent on both how busy the CPU is and when the user attempts to interact with the page. These post-TTI long tasks can also create frustrating responsiveness experiences, but unless they occur during the first interaction, they wouldn't be represented by FID. This is one reason why we need a field metric that more comprehensively represents users' experiences throughout the entire page lifetime.

464. <https://web.dev/long-tasks-devtools/>

Interaction to Next Paint (INP)

Interaction to Next Paint⁴⁶⁵ (INP) measures the amount of time it takes for the browser to complete the next paint in response to a user interaction. This metric was created after Google requested feedback⁴⁶⁶ on a proposal to improve how we measure responsiveness. Many readers may be hearing about this metric for the first time, so it's worth going into a bit more detail about how it works.

An *interaction* in this context refers to the user experience of providing an input to a web application and waiting for the next frame of visual feedback to be painted on the screen. The only inputs that are considered for INP are clicks, taps, and key presses. The INP value itself is taken from one of the worst interaction latencies on the page. Refer to the INP documentation⁴⁶⁷ for more info on how it's calculated.

Unlike FID, INP is a measure of all interactions on the page, not just the first one. It also measures the entire time until the next frame is painted, unlike FID which only measures the time until the event handler starts processing. In these ways, INP is a much more representative metric of the holistic user experience on the page.

A website has “good” INP if 75% of its INP experiences are faster than 200 ms. A website has “poor” INP if the 75th percentile is greater than or equal to 500 ms. Otherwise, it’s INP is assessed as “needs improvement”.

465. <https://web.dev/inp/>
466. <https://web.dev/responsiveness/>
467. <https://web.dev/inp/#what-is-inp>

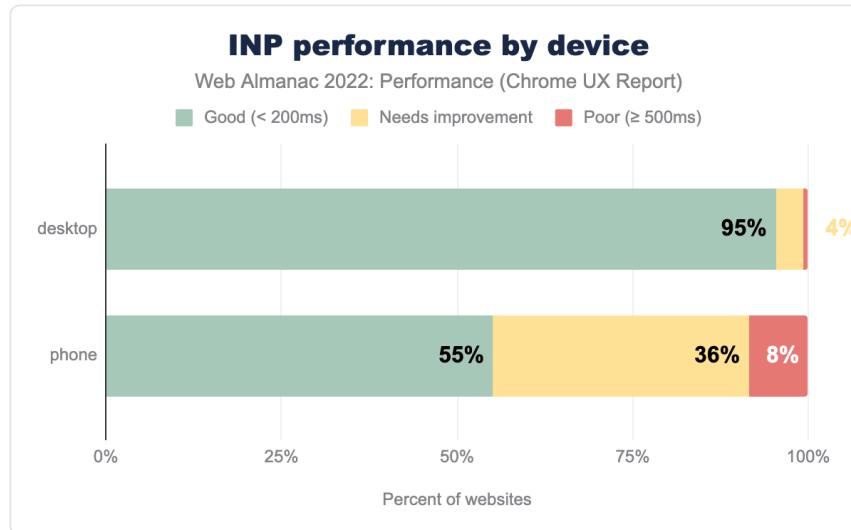


Figure 12.41. Distribution of INP performance by device.

55% of websites have “good” INP on mobile, 36% are rated “needs improvement”, and 8% have “poor” INP. The desktop story of INP is more similar to FID in that 95% of websites are rated “good”, 4% are rated “needs improvement”, and 1% are “poor”.

The enormous disparity between desktop and mobile users’ INP experiences is much wider than with FID. This illustrates the extent to which mobile devices are struggling to keep up with the overwhelming amount of work websites do, and all signs point to the increasing reliance on JavaScript⁴⁶⁸ as a major factor.

INP by rank

To see how unevenly distributed INP performance is across the web, it’s useful to segment websites by their popularity ranking.

⁴⁶⁸. <https://httparchive.org/reports/state-of-javascript?start=earliest&end=latest&view=grid#bytesJs>

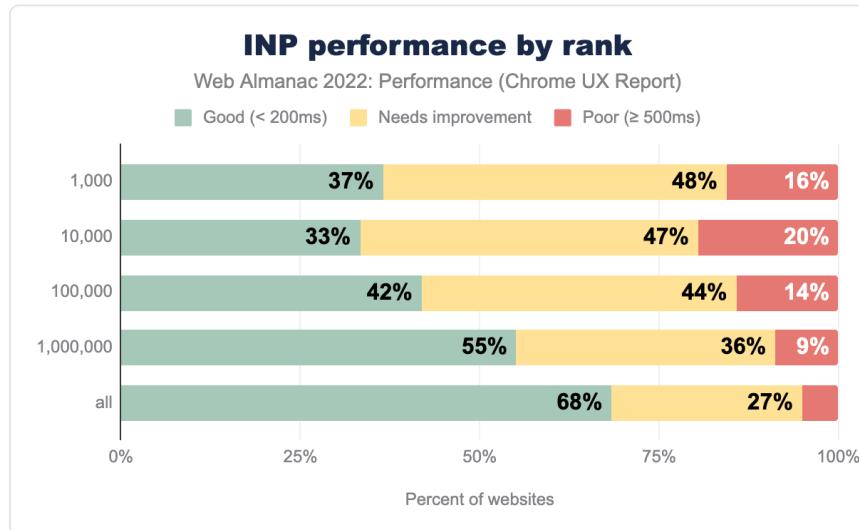


Figure 12.42. INP performance by rank.

27% of the top 1k most popular websites have good mobile INP. As the site popularity decreases, the percent having good mobile INP does something funny; it worsens at bit at the top 10k rank to 25%, then it improves to 31% at the top 100k, 41% at the top million, and it ultimately lands at 55% for all websites. Except for the top 1k, it seems that INP performance is inversely proportional to site popularity.

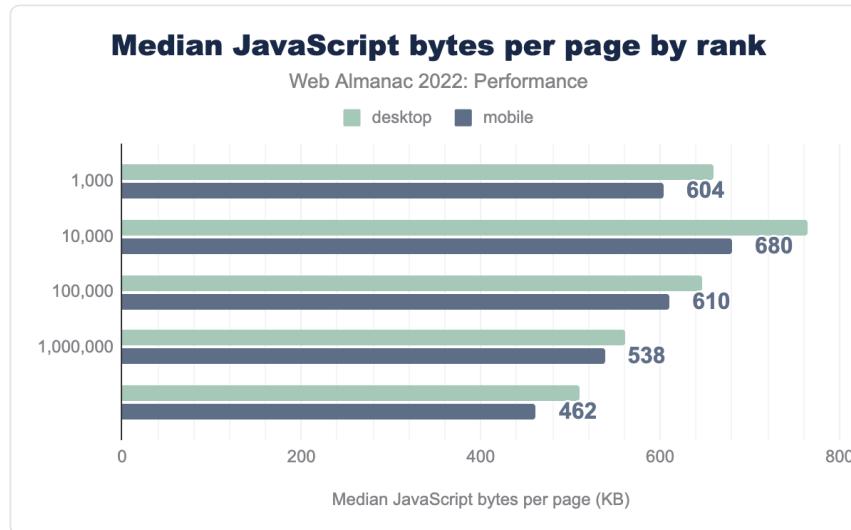


Figure 12.43. Median amount of JavaScript loaded per page, by rank.

When we look at the amount of JavaScript that the median mobile page loads for each of these ranks, it follows the same funny pattern! The median mobile page in the top 1k loads 604 KB of JavaScript, then it increases to 680 KB for the top 10k before dropping all the way down to 462 KB over all websites. These results don't prove that loading—and using—more JavaScript necessarily causes poor INP, but it definitely suggests a correlation exists.

INP as a hypothetical CWV metric

INP is not an official CWV metric, but Annie Sullivan⁴⁶⁹, who is the Tech Lead for the CWV program at Google, has commented⁴⁷⁰ about its intended future, saying "*INP is still experimental! Not a Core Web Vital yet, but we hope it can replace FID.*"

This raises an interesting question: hypothetically, if INP were to be a CWV metric today, how different would the pass rates be?

⁴⁶⁹. <https://twitter.com/anniesullivan>

⁴⁷⁰. <https://twitter.com/anniesullivan/status/1535208365374185474>

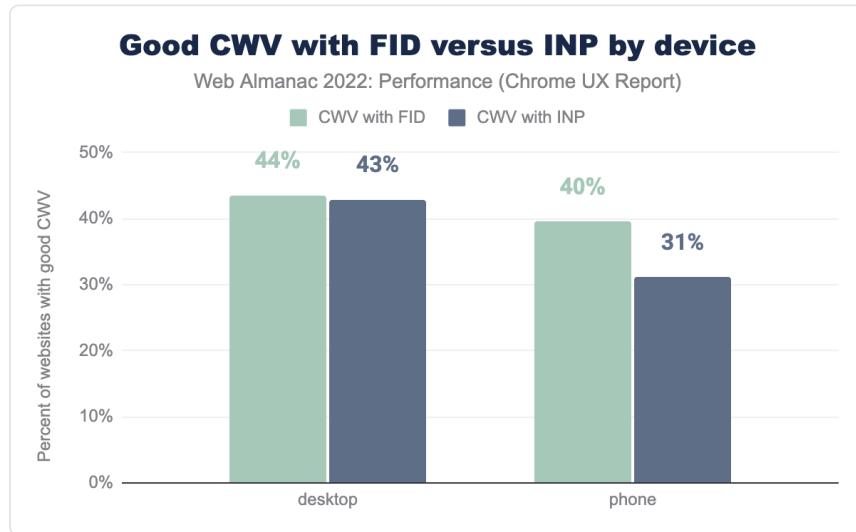


Figure 12.44. Comparison of the percent of websites having good CWV with FID and INP, by device.

For desktop experiences, the situation wouldn't change much. 43% of websites would have good CWV with INP, compared to 44% with FID.

However, the disparity is much more dramatic among websites' mobile experiences, which would fall to 31% having good CWV with INP, from 40% with FID.

Good mobile CWV with FID versus INP by rank

Web Almanac 2022: Performance (Chrome UX Report)

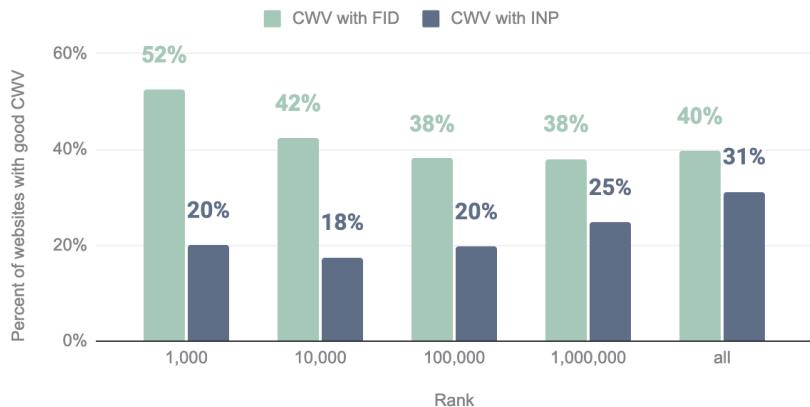


Figure 12.45. Comparison of the percent of mobile websites having good mobile CWV with FID and INP, by rank.

The situation gets even starker when we look at mobile experiences by site rank. Rather than 52% of the top 1k websites having good CWV with FID, only 20% of them would have good CWV with INP, a decrease of 32 percentage points. So even though the most popular websites overperform with FID compared to all websites (52% versus 40%), they actually *underperform* with INP (20% versus 31%).

The story is similar for the top 10k websites, which would decrease by 24 percentage points with INP as a CWV. Websites in this rank would have the lowest rate of good CWV with INP. As we saw in the previous section, this is also the rank with the highest usage of JavaScript. The rate of good CWV converges with FID and INP as the ranks become less popular, with the difference falling to 18, 13, and 9 percentage points respectively.

These results show that the most popular websites have the most work to do to get their INP into shape.

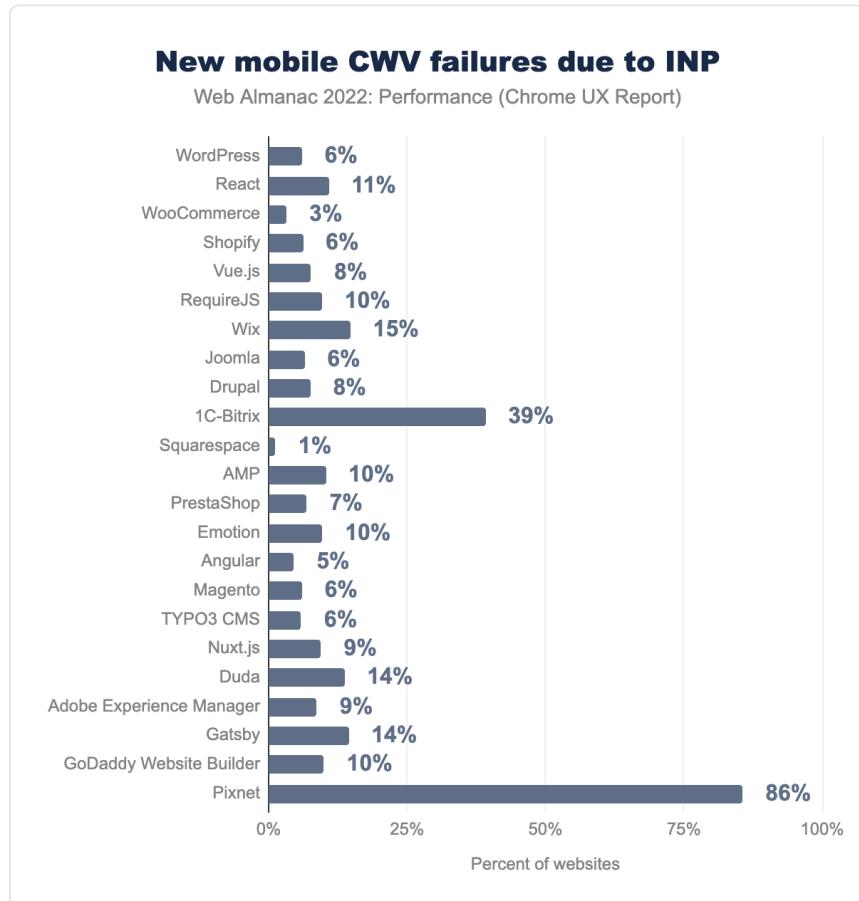


Figure 12.46. Percent change of websites having good CWV from FID to INP, by technology.

In this chart we're looking at the percent of a given technology's websites that would no longer be considered as having "good" CWV should FID be replaced with INP.

Two things jump out in this chart: 1C-Bitrix and Pixnet, which are CMSs and would have an enormous proportion of their websites ceasing to pass the CWV assessment with INP. Pixnet stands to lose 86% of its websites, down from 98% to 13%! The passing rate for 1C-Bitrix would fall from 79% to 40%, a difference of 39%.

11% of websites using the React framework would no longer pass CWV. Wix, which is now the second most popular CMS, uses React. 15% of its websites would not have "good" CWV. Proportionally though, there would still be more Wix websites passing CWV than React websites overall, at 24% and 19% respectively, but INP would narrow that gap.

WordPress is the most popular technology in the list and 6% of its 2.3 million websites would no longer have “good” CWV. Its passing rate would fall from 30% to 24%.

Squarespace would have the least amount of movement due to this hypothetical change, only losing 1% of its websites’ “good” CWV. This suggests that Squarespace websites not only have a fast first interaction but also consistently fast interactions throughout the page experience. Indeed, the CWV Technology Report⁴⁷¹ shows Squarespace significantly outperforming other CMSs at INP, having over 80% of their websites passing the INP threshold.

INP and TBT

What actually makes INP a better responsiveness metric than FID? One way to answer that question is to look at the correlation between field-based INP and FID performance and lab-based TBT performance. TBT is a direct measure of how unresponsive a page *can* be, but it can never be a CWV itself because it doesn’t actually measure the user-perceived experience⁴⁷².

This section draws from Annie Sullivan’s research⁴⁷³ using the May 2022 dataset.

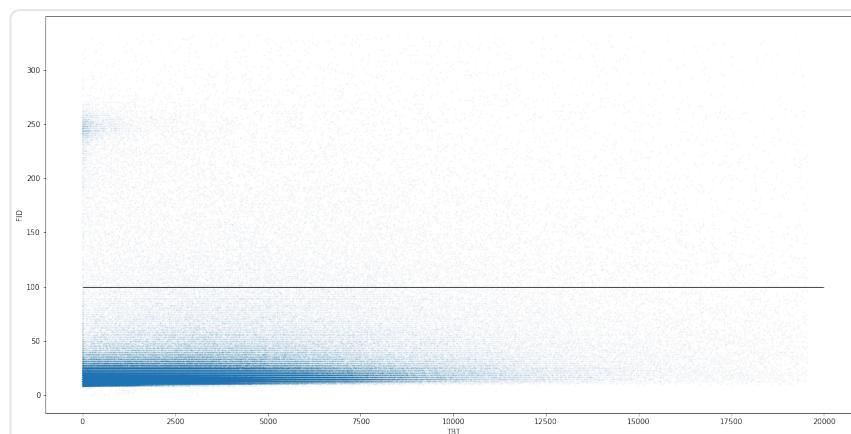


Figure 12.47. Scatterplot visualizing the correlation between FID and TBT. (Source⁴⁷⁴)

This chart shows the relationship between pages’ FID and TBT responsiveness. The solid horizontal line at 100 ms represents the threshold for “good” FID, and most pages fall comfortably under this threshold.

The most notable attribute of this chart is the dense area in the bottom left corner, which

471. <https://datastudio.google.com/s/M9D7EUjxU8>

472. <https://web.dev/user-centric-performance-metrics/>

473. <https://colab.sandbox.google.com/drive/12UlmAA8gyVjaUbmWvrbzj9BkkTxw6ay2>

appears to be smeared out across the TBT axis. The length of this smear represents pages having high TBT and low FID, which illustrates the low degree of correlation between FID and TBT.

There's also a patch of pages that have low TBT and a FID of about 250 ms. This area represents pages that have tap delay⁴⁷⁵ issues due to missing a `<meta name=viewport>` tag. These are outliers that can be safely ignored for this analysis's purposes.

The Kendall⁴⁷⁶ and Spearman⁴⁷⁷ coefficients of correlation for this distribution are 0.29 and 0.40, respectively.

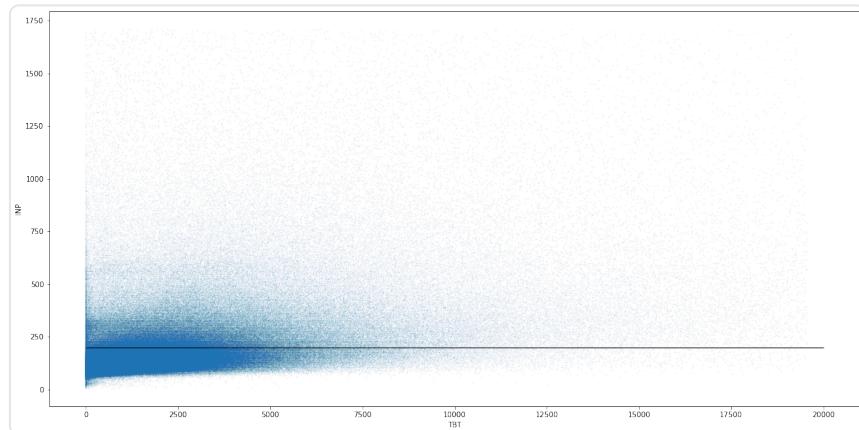


Figure 12.48. Scatterplot visualizing the correlation between INP and TBT. (Source⁴⁷⁸)

This is the same chart, but with INP instead of FID. The solid horizontal line here represents the 200 ms threshold for “good” INP. Compared to FID, there are many more pages above this line and not assessed as “good”.

Pages in this chart are more densely packed in the bottom left corner, which signifies the higher degree of correlation between FID and TBT. There's still a smear, but it's not as pronounced.

The Kendall and Spearman coefficients of correlation for this distribution are 0.34 and 0.45, respectively.

475. <https://developer.chrome.com/blog/300ms-tap-delay-gone-away/>
 476. https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient
 477. https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient
 478. <https://colab.sandbox.google.com/drive/12JmAAByVjaUbmWvrbzj9BkkTxw6ay2>

*First, is INP correlated with TBT? Is it more correlated with TBT than FID?
Yes and yes!*

But they are both correlated with TBT; is INP catching more problems with main thread blocking JavaScript? We can break down the percent of sites meeting the “good” threshold: yes it is!

—Annie Sullivan on Twitter⁴⁷⁹

As Annie notes, both metrics are correlated with TBT, but she concludes that INP is more strongly correlated, making it a better responsiveness metric.

FID conclusions

These results show that sites absolutely do have responsiveness issues, despite the rosy picture painted by FID. Regardless of whether INP becomes a CWV metric, your users will thank you if you start optimizing it now.

Nearly one in ten mobile sites are leaving free performance on the table by failing to disable double-tap to zoom. This is something all sites should be doing; it's only one line of HTML and it benefits both FID and INP. Run Lighthouse on your page and look for the viewport⁴⁸⁰ audit to be sure.

By taking a hypothetical look at INP as a CWV, we can see just how much work there is to be done just to get back to FID-like levels. The most popular mobile websites would be especially affected by such a change as a consequence of their (over)use of JavaScript. Some CMSs and JavaScript frameworks would be hit harder than others, and it'll take an ecosystem-wide effort to collectively rein in the amount of client-side work that they do.

Conclusion

As the industry continues to learn more about CWV, we're seeing steady improvement both in terms of implementation and across all top-level metric values themselves. The most visible performance optimization strides are at the platform level, like Android and bfcache improvements, given that their impact can be felt across many sites at once. But let's look at the most elusive piece of the performance puzzle: individual site owners.

479. <https://twitter.com/anniesullivan/status/1525161893450727425>

480. <https://web.dev/viewport/>

Google's decision to make CWV part of search ranking catapulted performance to the top of many companies' roadmaps, especially in the SEO industry. Individual site owners are certainly working hard to improve their performance and played a major role in the CWV improvements over the last year, even if those individual efforts are much harder to spot at this scale.

That said, there's still more work to be done. Our research shows opportunities to improve LCP resources' prioritization and static discoverability. Many sites are still failing to disable double-tap to zoom to avoid artificial interactivity delays. New research into INP has uncovered responsiveness problems that were easy to overlook with FID. Regardless of whether INP becomes a CWV, we should always strive to deliver fast and responsive experiences, and the data shows that we can be doing better.

At the end of the day, there will always be more work to do, which is why the most impactful thing we can do is to continue making web performance more approachable. In the years to come, let's emphasize getting web performance knowledge the "last mile" to site owners.

Authors



Melissa Ada

🐦 @mel_melificent ⚡ mel-ada 💬 mel-ada

Mel Ada is a software engineer on the Web Performance team at Etsy. Her current involvement in the community includes co-organizing the NY Web Performance Meetup and speaking about recent works.

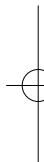
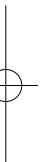


Rick Viscomi

🐦 @rick_viscomi ⚡ rviscomi

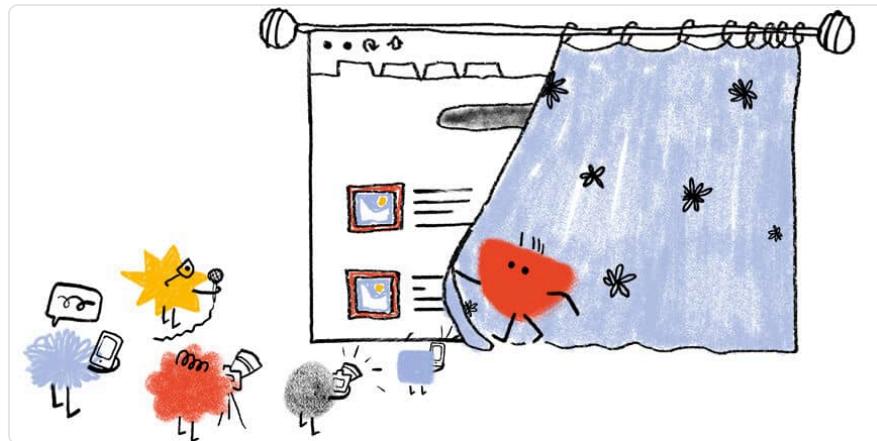
Rick Viscomi is a DevRel engineer at Google, focusing on web performance. He is the co-author of *Using WebPageTest*⁴⁸¹, a book about web performance testing. He also co-maintains HTTP Archive and is the Editor-in-Chief of the Web Almanac.

481. <https://usingwpt.com/>



Part II Chapter 13

Privacy



Written by Tom Van Goethem and Nurullah Demir

Reviewed by Iskander Sanchez-Rola

Analyzed by Max Ostapenko and Yana Dimova

Edited by Abel Mathew

Introduction

Whether it is to keep up-to-date with the latest news, stay in touch with friends via online social media, or look for a nice dress or sweater to buy, many of us rely on the web to provide us with these services and information with just a couple of clicks. A side-effect of spending almost 7 hours per day on the internet on average⁴⁸², is that a lot of our browsing activities, and thus indirectly our personal interests and data, is captured or shared with a plethora of online web services and companies.

As advertisers try to provide users with ads that are most relevant to them (as these are the ones they are most likely to interact with), they often resort to third-party tracking to infer the user's interest. In essence, a user's online activities are tracked step by step, providing trackers, in particular those that are most prevalent on the web, with a heap of information, most of which is probably not even relevant to infer the user's interests. On top of that, users are

482. <https://datareportal.com/reports/digital-2022-global-overview-report>

generally not given an adequate choice to opt-out of this.

In this chapter, we explore the current state of the web in terms of privacy. We report on the ubiquitousness of third-party tracking, the different services that make up this ecosystem, and how certain parties are trying to circumvent the protective measures that users are employing to protect their privacy (for example, blocklist-based anti-trackers). Furthermore, we also look into how websites are trying to enhance the privacy of their visitors, either by adopting features that limit the information shared with other parties, or by being compliant with privacy regulations such as GDPR⁴⁸³ and CCPA⁴⁸⁴.

Online tracking



Figure 13.1. The percentage of websites that include at least one third-party tracker on desktop.

Tracking is one of the most pervasive web technologies on the web—we find that 82% of desktop websites (80% for mobile) include at least one third-party tracker. By following users' behavior online, these tracking companies can create profiles of them, which can be used for personalized advertising, give insights to website owners on who visits their websites, or use this information to distinguish legitimate users from (unwanted) bots. In this section, we explore the different techniques that are used to track the activities of users online and look at how trackers aim to circumvent the various privacy features that aim to protect users from being tracked.

Third-party tracking

One of the most common forms of online tracking is through third-party services, where a website owner typically includes a third-party, cross-site, script that provides site analytics or shows advertisements to visitors. This script can then set a third-party cookie, and log which website the user visited. Whenever the user visits another website that includes the same third-party service, the cookie will be sent along to the tracker, allowing them to re-identify the user and link both website visits to the same profile.

The types of third-party services that are included—and by doing so are implicitly given the capabilities of tracking website visitors—somewhat vary. The two most common categories (as

483. <https://gdpr.eu/>

484. <https://oag.ca.gov/privacy/ccpa>

defined by WhoTracks.me⁴⁸⁵) of such trackers are site analytics scripts (68% on mobile, 73% on desktop) and advertising (66% on mobile, 68% on desktop). These two are followed by a few other categories, some of which might not have a clear link to tracking: customer interaction (services that allow customers to easily send messages to the website owner), audio/video players (for example, YouTube embedded videos), and social (for example, Facebook “like” buttons).

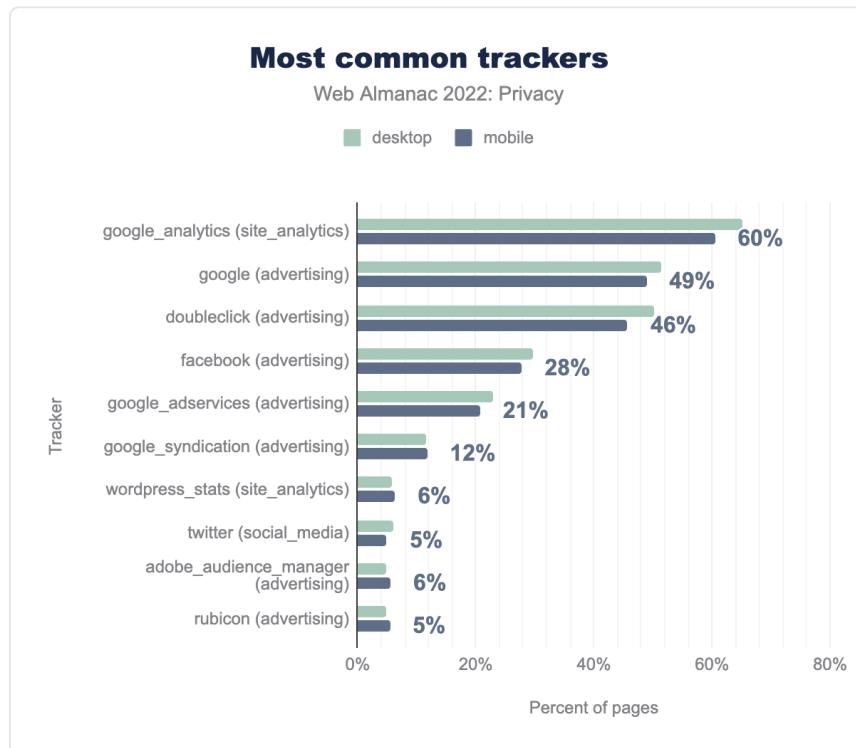


Figure 13.2. Most common trackers.

For a tracker to successfully profile a user, they need to be included in a large fraction of websites in order to be able to track a significant fraction of users’ online activities. When we look at the most common trackers, these are mostly the “usual suspects”. Of the top 10 most common trackers five are affiliated with Google. Also included in this list are popular social networks such as Facebook and Twitter.

485. https://whotracks.me/blog/tracker_categories.html

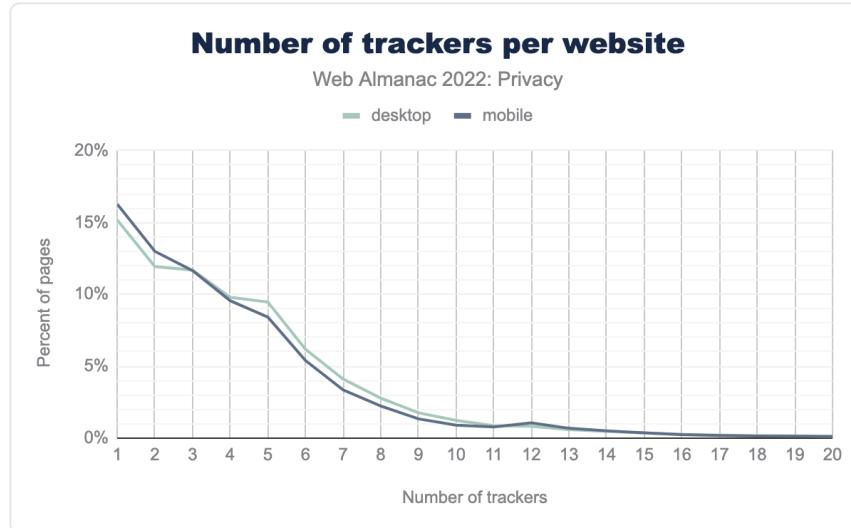


Figure 13.3. Number of trackers per website.

Websites might want to make use of multiple third-party services, and thus may include multiple trackers in their website (be sure to check out the Third Parties chapter for a deep dive into which third parties are included on the web!). We find that approximately 15% of desktop sites and 16% of mobile sites include “just” one tracker. Unfortunately, this means that it is in fact more common for websites to include multiple trackers. We even found one website that included 126 different trackers!

(Re)targeting

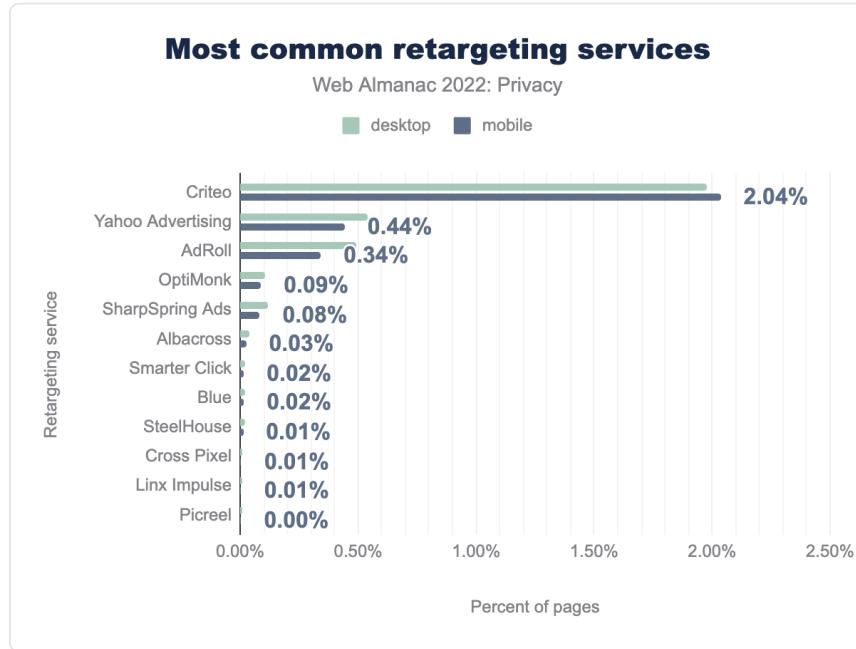


Figure 13.4. Most common retargeting services.

When browsing the web, we often encounter advertisements for products that we recently looked up. The reason for that is ad retargeting. When a website detects that a user might be interested in a specific product, they report this to the tracker and/or advertiser, who will later on, when the user is visiting other, unrelated websites, show advertisements for the product that the user is supposedly interested in, in an attempt to nudge them into purchasing it.

The tracker offering most of the purely retargeting services is Criteo, with a prevalence of 1.98% on desktop and 2.04% on mobile. It is followed by Yahoo Advertising and AdRoll, which collectively make up less than half of Criteo's market share. The most widely used retargeting service of last year⁴⁸⁶, Google Tag Manager, does not show in these results as it is now classified under the "tag managers" Wappalyzer category. Although this service is used for retargeting, it does so indirectly, by the inclusion of retargeting tags which are detected separately.

486. <https://almanac.httparchive.org/en/2021/privacy>

Third-party cookies

As mentioned before, the most established way to track users across different websites is by means of third-party cookies. With recent changes in browser policies, cookies will no longer be included in cross-site requests by default. In technical terms this means that most browsers set the `SameSite` attribute of cookies to the default value `Lax`. Websites can override this by explicitly setting the value themselves. This has been happening on a large scale: of the third-party cookies that set the `SameSite` cookies, 98% of them set it to the value `None`, allowing them to be included in cross-site requests. Furthermore, the expiration time of the cookie also determines how long it remains valid; we find that the median lifetime of a cookie is 365 days. For a deeper dive into cookies and cookie attributes, please refer to the Security chapter.

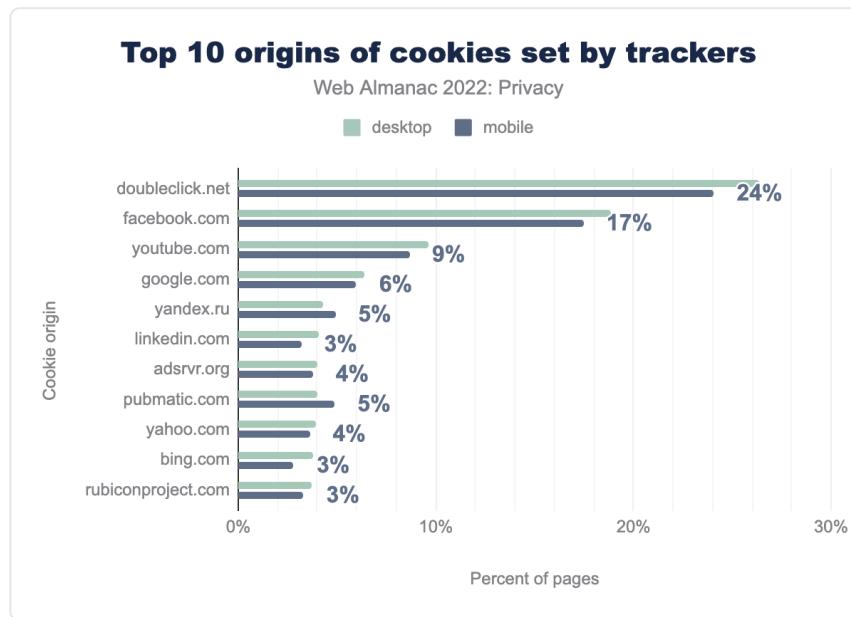


Figure 13.5. Top 10 origins of cookies set by trackers.

For a large part, the third-party trackers that set cookies largely coincide with the third parties that are included on websites. However, the most popular third-party tracker, Google Analytics, is not as prevalent here. This can be attributed to the fact that Google Analytics sets a first-party cookie (`_ga`), which according to their definition⁴⁸⁷ “is unique to the specific property, so it cannot be used to track a given user or browser across unrelated websites”. Nevertheless, the most common tracking domain that sets third-party cookies, `doubleclick.net`, is still Google affiliated. The other domains on the list are associated with social media and

487. <https://policies.google.com/technologies/cookies?hl=en-US>

advertising.

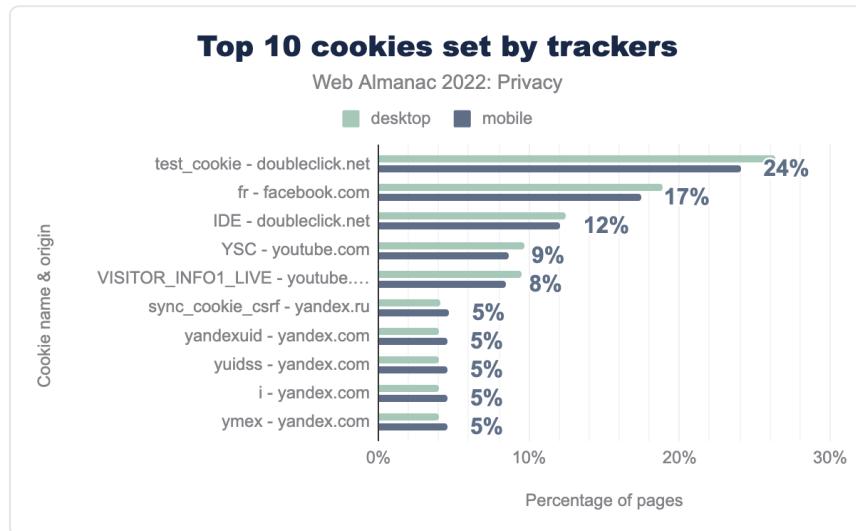


Figure 13.6. Top 10 cookies set by trackers.

When looking at the most common third-party cookies, we again see several tracking domains, lead by the `test_cookie` from `doubleclick.net` —a cookie with a lifespan of 15 minutes that is used for functionality purposes according to its description⁴⁸⁸. This cookie is followed by the `fr` cookie set by `facebook.com` —a cookie “used to deliver, measure and improve the relevancy of ads, with a lifespan of 90 days” according to its definition⁴⁸⁹. The rest of the 10 most prevalent third-party cookies are set by YouTube and Yandex.

^{488.} <https://business.safety.google/adscookies/>

^{489.} <https://www.facebook.com/policy/cookies/>

Evasion technique: fingerprinting

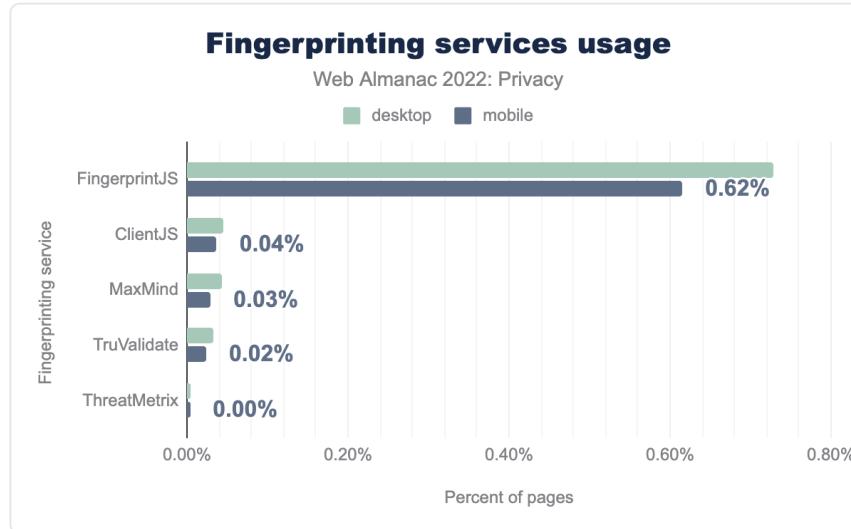


Figure 13.7. Fingerprinting services usage.

As more and more browsers develop countermeasures for cookie-based tracking, and giving users more control to block third-party cookies, some trackers aim to circumvent these protections. One such technique is fingerprinting, where browser-specific features (for example, installed browser extensions), OS-specific features (for example, installed fonts) and hardware-specific features (for example, differences in rendering complex composition based on which GPU is used) are used to create a unique fingerprint of the user. This fingerprint then allows the tracker to re-identify the same user across different, unrelated websites.

In our analysis, we looked for five different, known fingerprinting libraries and we find that the most prevalent library used on the web to perform fingerprinting is FingerprintJS⁴⁹⁰, which we find on 0.62% of all websites. Most likely this is because the library is open source, and has a free version. Compared to our measurements last year⁴⁹¹, we find that the use of fingerprinting has approximately stayed the same.

Evasion technique: CNAME tracking

As most of the tracking countermeasures focus on blocking or disabling third-party cookies, another way to circumvent these protections is to use first-party cookies instead. Here, the

490. <https://github.com/fingerprintjs/fingerprintjs>

491. <https://almanac.httparchive.org/en/2021/privacy>

tracker is cloaked using a CNAME record on a subdomain of the website it is embedded in. When the tracker then sets a cookie, it will be considered a first-party cookie. A limitation of CNAME-based tracking is that it can only be used to track a user's activities within a specific website, although the tracker could still rely on cookie syncing⁴⁹² to match visits across multiple sites together.

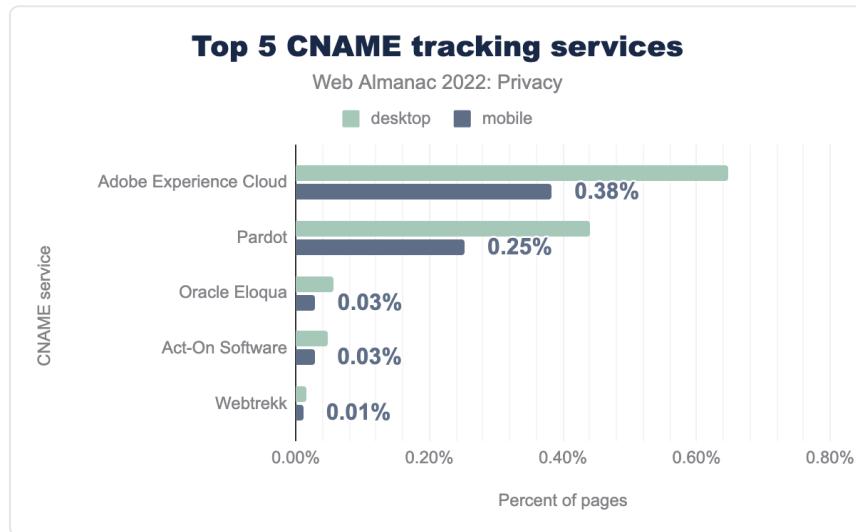


Figure 13.8. Top 5 CNAME tracking services.

By analyzing the various CNAME trackers, we find that the market share is mainly concentrated around two main services: Adobe Experience Cloud (0.65% on desktop and 0.38% on mobile) and Pardot (0.25% on desktop and 0.44% on mobile). Interestingly, the adoption of CNAME tracking is significantly higher on websites visited with a desktop browser compared to those visited on mobile. Presumably this is because there are fewer privacy-preserving mechanisms on mobile browsers—for example, most of the popular browsers on mobile do not support extensions.

^{492.} <https://adtechexplained.com/cookie-syncing-explained/>

CNAME tracking usage per website rank group

Web Almanac 2022: Privacy

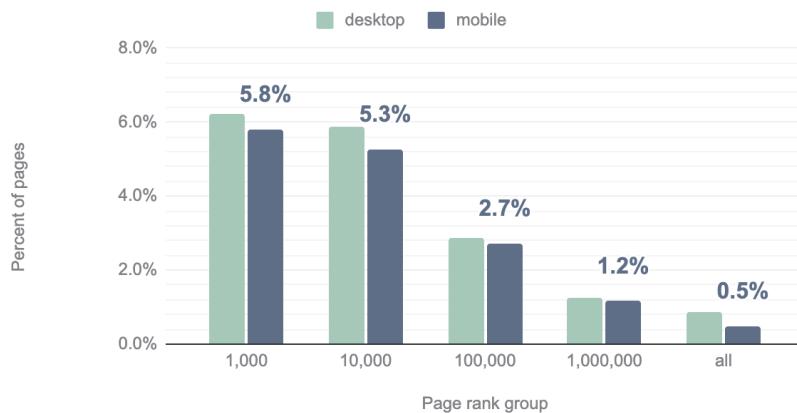


Figure 13.9. CNAME tracking usage per website rank group.

Although the overall prevalence of CNAME-based tracking might not seem very high (0.9% on desktop websites, 0.5% on mobile sites), its adoption is mainly concentrated on highly popular websites. Within the top 1,000 most visited websites 6.2% of desktop sites and 5.8% of mobile sites embed a CNAME tracker. This means that users are quite likely to encounter such trackers when browsing the web.

Access to (sensitive) data from the browser

Browsers have an abundant number of APIs, which provide developers with useful mechanisms to interact with different components in whichever way they want. Several of these APIs can also be used to extract information from sensors or other peripherals connected to the user's device. While most APIs provide a limited amount of information (such as the orientation of the screen), others provide very detailed information (for example, the accelerometer and gyroscope), which could be used for device fingerprinting, or even inferring which password a user types based on the movements they make with their mobile device.

Sensor events

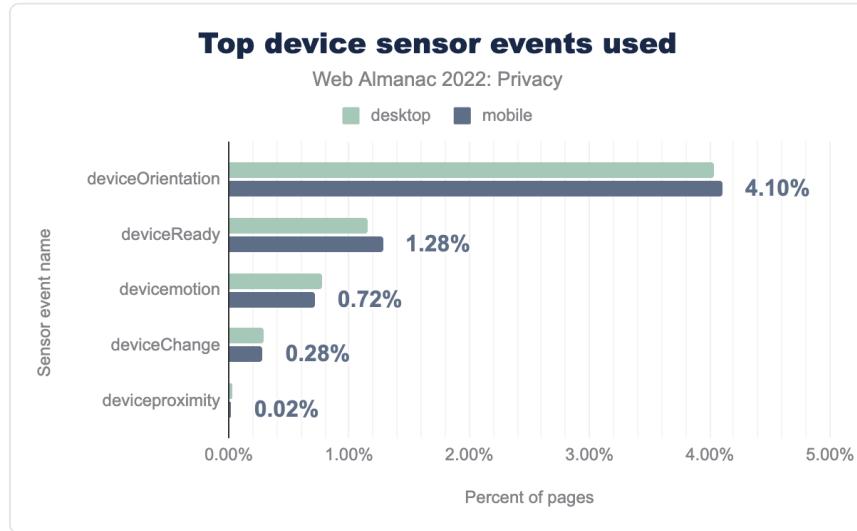


Figure 13.10. Top device sensor events used.

We find that the most prominent sensor event that websites listen for is the `deviceOrientation` event, which fires when the device changes from portrait to landscape mode or vice versa. It is used on 4.0% of desktop websites and 4.1% of mobile websites. The usage is likely this high (relatively) because websites might want to update elements of the layout when the orientation of the device changes.

Media devices

0.59%

Figure 13.11. The percentage of desktop pages that enumerate media devices.

Using the MediaDevices API⁴⁹³, web developers can use the `enumerateDevices()` method to get a list of all media devices connected to the user's device. While this feature is useful to determine whether a user has a camera or microphone connected to initiate a video call, it can also be used to gather information about the system's environment for fingerprinting purposes.

493. <https://developer.mozilla.org/docs/Web/API/MediaDevices>

We find that 0.59% of desktop websites and 0.48% of mobile sites try to access the list of connected media devices—note that our crawler does not interact with the site, nor click on any buttons. Interestingly, the usage of this API has significantly reduced since last year⁴⁹⁴, when the prevalence of sites accessing the list of media devices was 12 times higher. Most likely this is due to a popular library that no longer calls the API.

Geolocation

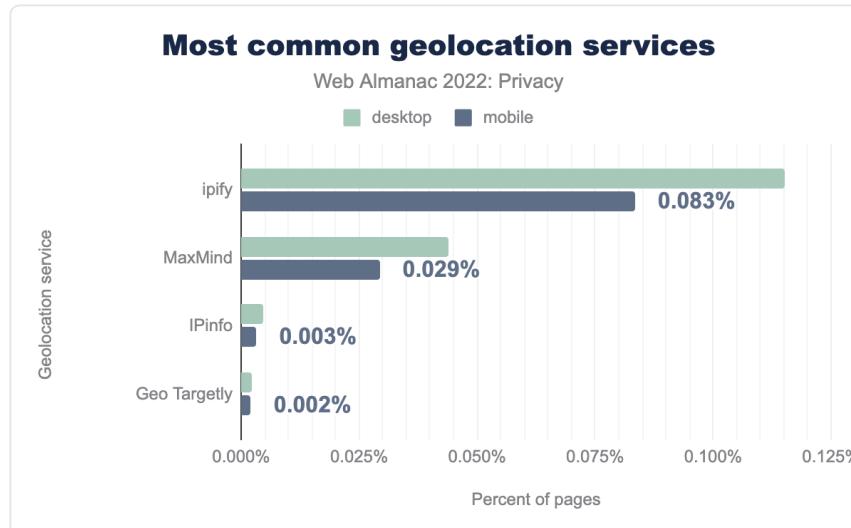


Figure 13.12. Most common geolocation services.

A lot of the content that is served to us is localized based on the location that we're visiting websites from. For web developers to determine where a visitor is from, they can use third-party geolocation service. These will determine a user's location based on their IP address. Although this geolocation is typically used on the back-end, we do find some usage also in the front-end: 0.115% of desktop sites and 0.083% of mobile sites contact ipify to determine the user's IP location.

0.65%

Figure 13.13. The percentage of desktop pages that try to access the browser's geolocation.

494. <https://almanac.httparchive.org/en/2021/privacy#media-devices>

As the IP-based geolocation service can be quite inaccurate, especially when users rely on a VPN to hide their original IP address, websites might request a more granular location through the Geolocation API⁴⁹⁵. Of course, access to this (privacy-intrusive) API is still guarded by a permission that users manually need to provide. Yet, we find that 0.65% of desktop sites and 0.61% of mobile sites try to access the user's current location upon a visit to the home page, without any user interaction. Interestingly, we still find 574 desktop sites—down from 900 last year—that try to access the feature while the page was loaded over an insecure connection. Due to the sensitive nature of the data that this feature provides, most browsers restrict its use to secure origins.

Established controls to improve visitor's privacy

As websites include a lot of content (scripts, plugins... etc.) from third parties that they might not entirely trust, they might want to protect their users' privacy from these third parties. Next, we explore the various controls that can be used to restrict the features or data that third parties have access to, or that make it explicitly clear which information a website wants to obtain from a user.

Permissions Policy

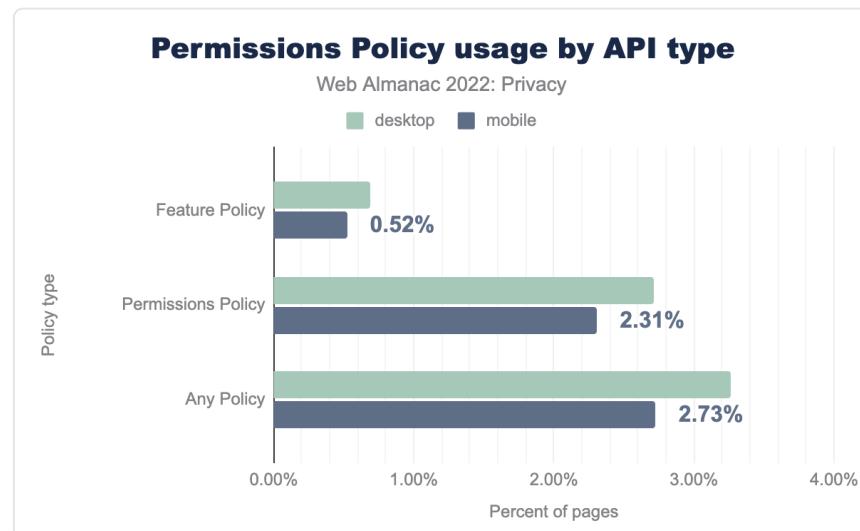


Figure 13.14. Permissions Policy usage by API type.

495. https://developer.mozilla.org/docs/Web/API/Geolocation_API

By default, any third-party script can access the same browser features as the website they're embedded in. In order to limit the features that will be enabled for the website, the website can make use of the Permissions Policy⁴⁹⁶. Through an HTTP response header the website can indicate which features it wants to allow. For instance, if the `microphone` feature is not included in this list, none of the scripts embedded in the web page can use it. Although the policy is fairly new, we are seeing an adoption of 2.71% on desktop sites and 2.31% on mobile sites.

The Permissions Policy supersedes the Feature Policy⁴⁹⁷, which can still be found on 0.69% of desktop sites and 0.52% of mobile sites. By default most of the features regulated by the Permissions Policy are disabled in cross-origin iframes, they can be explicitly enabled through the `allow` attribute. We find that 15.18% of desktop sites and 14.32% of mobile sites make use of this feature. For a more detailed analysis on the use of the `allow` attribute on iframes, please refer to the Security chapter.

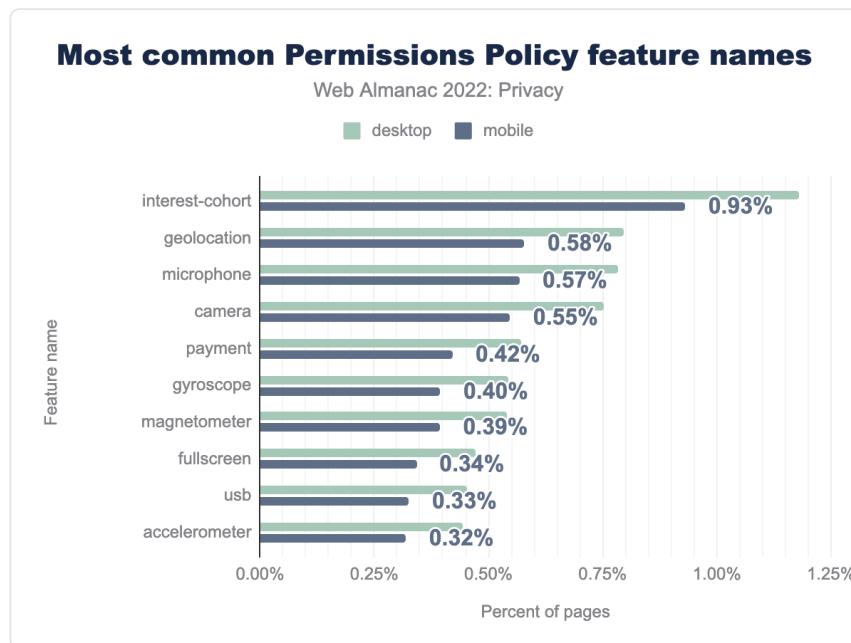


Figure 13.15. Most common Permissions Policy feature names.

When we look at the directives that are used in the Permissions Policy, we see a similar usage compared to last year⁴⁹⁸, with the exception of the one that's most widely used in 2022, namely

496. <https://developer.chrome.com/en/docs/privacy-sandbox/permissions-policy/>

497. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Feature-Policy>

`interest-cohort`. This directive can be used to limit the access to the now-defunct FLoC API. Presumably, this increase can be attributed to the various shortcomings of FLoC (increases fingerprinting surface, reveals potentially sensitive information about users... etc.) where website owners, providers and libraries took an active step in trying to protect the privacy of their users.

Referrer Policy



Figure 13.16. The percentage of desktop sites that sets a document-wide Referrer Policy.

By default, most user agents will include a `Referer` header. In short, this reveals to third parties from which web site—or even page—a request was initiated. This is the case for any resource that was embedded in the web page, as well as for the request that was initiated after a user clicked on a link. Of course, this has the undesirable side-effect that these third parties learn which website, or even which web page a specific user was visiting. By making use of the Referrer Policy⁴⁹⁹, websites can limit the instances in which the `Referrer` header is included in requests and thus improve user privacy. We find that 12% of the desktop sites and 10%- of the mobile sites set such a document-wide policy, mostly via an HTTP response header.

499. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Referrer-Policy>

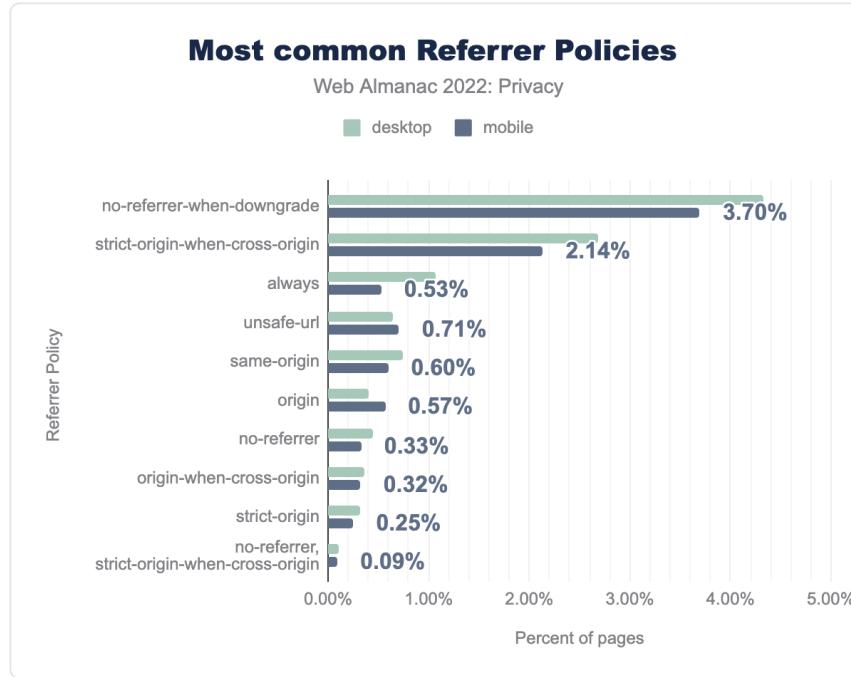


Figure 13.17. Most common Referrer Policies.

We find that the most common usage of the Referrer Policy is to not include the `Referer` header on downgrade requests, that is, HTTP requests initiated on an HTTPS-enabled page. Unfortunately, this still leaks the page that the user is visiting in most scenarios—in HTTPS-enabled requests. We do see that 2.7% of desktop sites and 2.1% of mobile sites aim to hide the specific web page that a user is visiting through the `strict-origin-when-cross-origin` policy, which is now most browsers default when a policy is not specified.

User-Agent Client Hints

In an effort to reduce the information that is revealed about the browser environment, and more specifically the `User-Agent` string, the User-Agent Client Hints⁵⁰⁰ mechanism was introduced. Through this feature, websites that want to access certain information about the user's browsing environment (browser version, operating system... etc.) now have to set a header (`Accept-CH`) in the first response, upon which the browser will send the requested data in subsequent requests. Among other benefits, this feature reduces the fingerprinting surface and allows browsers to intervene in sending certain data, for example, via the Privacy

⁵⁰⁰. <https://wicg.github.io/ua-client-hints/>

Budget⁵⁰¹ proposal.

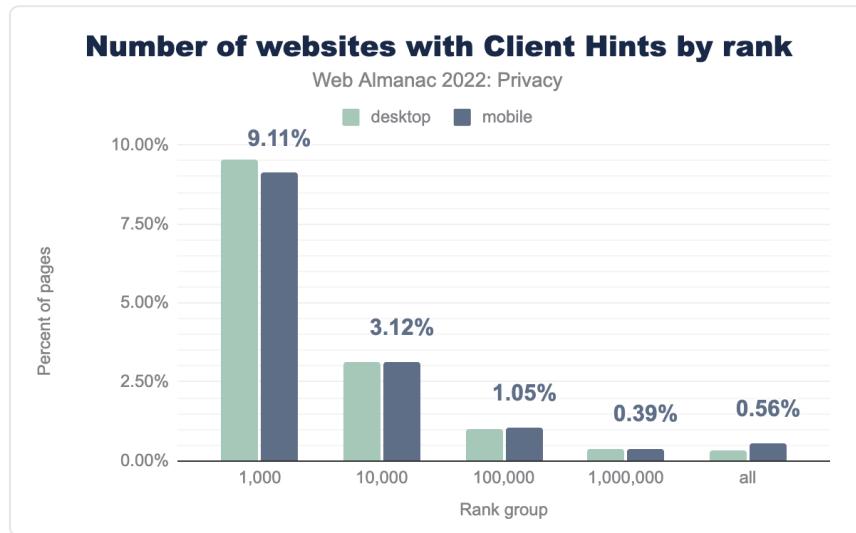


Figure 13.18. Number of websites with Client Hints by rank group.

When we look at the adoption of sites that respond with the `Accept - CH` header in comparison with the results from last year⁵⁰² (top 1k: 3.56%, top 10k: 1.44%), we see a significant increase in adoption, almost 3x for the most popular sites. Presumably, this increase in adoption is related to the fact Chromium has been reducing the information that is shared in the `User-Agent` string (through the User-Agent Reduction plan⁵⁰³).

We find that the sites that make use of User-Agent Client hints, generally request access to a relatively large number of properties, limiting the benefit of what browsers aim to achieve through efforts such as User-Agent Reduction. It will be interesting to see in the near future how/whether browsers will limit the practices of acquiring a lot of information about the user's browsing environment.

New efforts to improve privacy by the browser

Over the last few years, the average web user has become increasingly conscious about their online privacy. On the one hand, the many data breaches, which just seem to keep on happening and getting bigger and bigger⁵⁰⁴, have left very few unaffected. On the other hand, the fact of the

501. <https://github.com/mikewest/privacy-budget>

502. <https://almanac.httparchive.org/en/2021/privacy/>

503. <https://www.chromium.org/updates/ua-reduction/>

504. <https://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/>

ubiquitous tracking of users through third-party cookies is becoming increasingly well known within the general population. As a result, more and more users are starting to expect their browser to protect their privacy, and give them more control over the tracking of online behaviors. Browser vendors, online publishers and ad-tech companies have heard this demand for improved privacy, and have proposed the Privacy Sandbox—an initiative led by Google Chrome.

Privacy Sandbox Origin Trial

At the time of publishing this year's Web Almanac, Privacy Sandbox features are not yet available for general use. Websites and web services—such as ads, which are typically shown in iframes—can however participate in early testing of the Privacy Sandbox features, by making use of the Origin Trial⁵⁰⁵. Note that this is only for users whose browser supports the feature—Privacy Sandbox features are only implemented in Chrome, and are still disabled by default at the time of this writing. This gives the web services access to three Privacy Sandbox-related APIs: Topics⁵⁰⁶, FLEDGE⁵⁰⁷, and Attribution Reporting⁵⁰⁸.

Origin requesting feature	Desktop	Mobile
https://www.googletagmanager.com	12.53%	10.99%
https://googletagservices.com	11.05%	10.52%
https://doubleclick.net	11.04%	10.51%
https://googlesyndication.com	11.04%	10.51%
https://googleadservices.com	2.50%	2.29%
https://s.pinimg.com	1.49%	1.21%
https://criteo.net	0.64%	0.41%
https://criteo.com	0.59%	0.37%
https://imasdk.googleapis.com	0.10%	0.07%
https://teads.tv	0.04%	0.03%

Figure 13.19. Prevalence of origins requesting access to the Privacy Sandbox API Origin Trial.

The most prevalent services on the web that will test during the Origin Trial of Privacy Sandbox

505. <https://developer.chrome.com/en/blog/privacy-sandbox-unified-origin-trial/>

506. <https://developer.chrome.com/docs/privacy-sandbox/topics/>

507. <https://developer.chrome.com/docs/privacy-sandbox/fledge/>

508. <https://developer.chrome.com/docs/privacy-sandbox/attribution-reporting/>

are: Google Tag Manager, Doubleclick, Google Syndication and Google Ad Services make up the top five on both desktop and mobile sites. These are followed by the social media site Pinterest, and other trackers and advertisers: Criteo, Google Ads SDK, and Teads.

Privacy Sandbox experiments

The Privacy Sandbox initiative consists of many different features that each touch upon different aspects, and aim to still support the current common actions that users perform on the web when third-party cookies are phased out. As most features are still under active development, websites have not adopted them yet (with the exception of services opting-in to the `PrivacySandboxAdsAPIs` Origin Trial).

For some time the Origin Trial for various Privacy Sandbox features was divided into separate trials, one for each feature. Although these trials do not have any effect in modern browsing environments, some web services did opt-in to them and forgot to remove the `Origin-Trial` response header.

For example, we find that on 34,128 sites a web service opts-in to the `ConversionMeasurement` Origin Trial, which at one point gave them access to the Attribution Reporting API⁵⁰⁹ (previously called the Conversion Measurement API). This API is used to track the conversion of a user clicking an ad to a purchase, for example.

For the TrustTokens⁵¹⁰ Origin Trial, which has also expired, we are still seeing 6,005 sites where a web service opts-in to it. This mechanism aims to allow websites to combat fraud by enabling one browsing context (for example, site) to convey a limited amount of information to another.

Interestingly, on more than 30,000 websites a web service is still opting-in to the `InterestCohort` origin trial, which would give them access to the interest group of the user of FLoC. However, due to privacy concerns with the API, it was no longer pursued and development was discontinued. It is superseded by the FLEDGE API⁵¹¹, which aims to provide “on-device ad auctions to serve remarketing and custom audiences” and Topics API⁵¹², which aims to allow advertisers to serve ads based on the interests of the user without the need of cross-site tracking.

Compliance with privacy regulations

The data privacy regulatory space continues to expand as the newest frontier of legislation.

509. <https://developer.chrome.com/en/docs/privacy-sandbox/attribution-reporting/>

510. <https://developer.chrome.com/en/docs/privacy-sandbox/trust-tokens/>

511. <https://developer.chrome.com/docs/privacy-sandbox/fledge/>

512. <https://developer.chrome.com/docs/privacy-sandbox/topics/>

These regulations require organizations to be more transparent regarding their users' data processing to protect their data. Following the advent of key data privacy regulations like General Data Protection Regulation (GDPR)⁵¹³ and IAB Transparency and Consent Framework (TCF) v2.0⁵¹⁴, website providers took action to inform the users about processed data during the visit and take consent from these users to process their data also for non-functional purposes—for example, tracking and ads. This has led to us seeing cookie banners on websites more often because website providers notify their users or ask for consent mainly through (cookie) consent banners.

In most cases, users can interact with such consent banners and set which data should be processed. However, managing such tasks is not easy on our modern, sophisticated web, which is also getting more complicated. For this reason, website operators try to hand over this task to third parties—so-called Consent Management Platform (CMP). CMPs ensure that the cookies are used on the respective websites by the law. In the following, we discuss the use of CMPs and notification of privacy policy.

Consent Management Platforms

As we have already discussed, using the consent management platform should ensure that the website, in particular the behavior with cookies, should run in a legally compliant manner.

At this point, we would also like to note that the integration of CMP services does not always ensure that the websites remain legally compliant, as the studies in this field show (for example, Santos et al.⁵¹⁵ and Fouad et al.⁵¹⁶).

513. <https://data.consilium.europa.eu/doc/document/ST-9565-2015-INIT/en/pdf>

514. <https://www.iabeurope.eu/>

515. <https://arxiv.org/abs/2104.06861>

516. <https://ieeexplore.ieee.org/document/9229842>

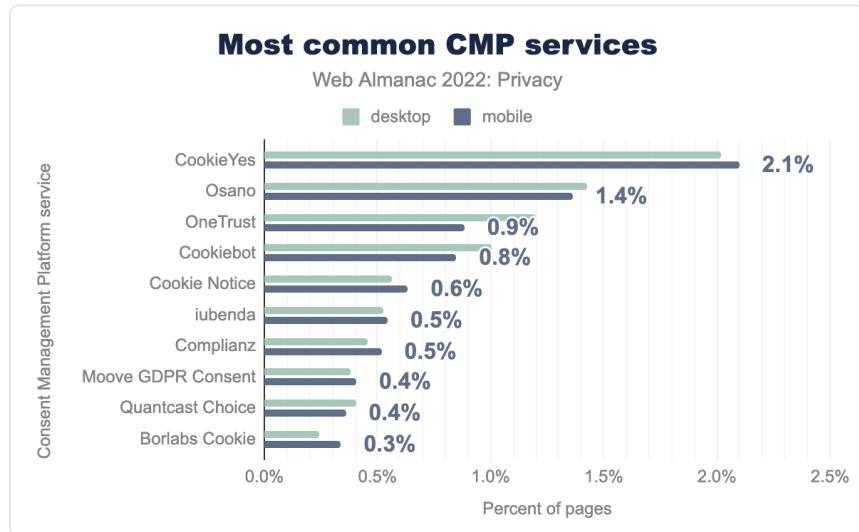


Figure 13.20. Most common Consent Management Platform (CMP) services.

Our analysis shows that CMP usage has increased from 7% to 11% since last year. So we recorded an increase of almost 60%. Also, this year we see that mobile is less involved than desktop—although the difference is minimal. We also see that the providers CookieYes (18%), OneTrust (64%), and Cookiebot (56%) have increased their market share since last year.

IAB consent frameworks

Compared to GDPR, the IAB Europe Transparency and Consent Framework (TCF)⁵¹⁷ is an industry-standard where global vendors⁵¹⁸ are involved. The goal is to establish communication between user consent and advertisers. TCF ensures that the websites in Europe are GDPR-compliant. IAB Tech Lab US developed the U.S. Privacy Technical Specifications (USP)⁵¹⁹ was designed for the United States using the same concept of TCF.

⁵¹⁷ <https://iabeurope.eu>

⁵¹⁸ <https://iabeurope.eu/vendor-list/>

⁵¹⁹ <https://iabtechlab.com/standards/ccpa/>

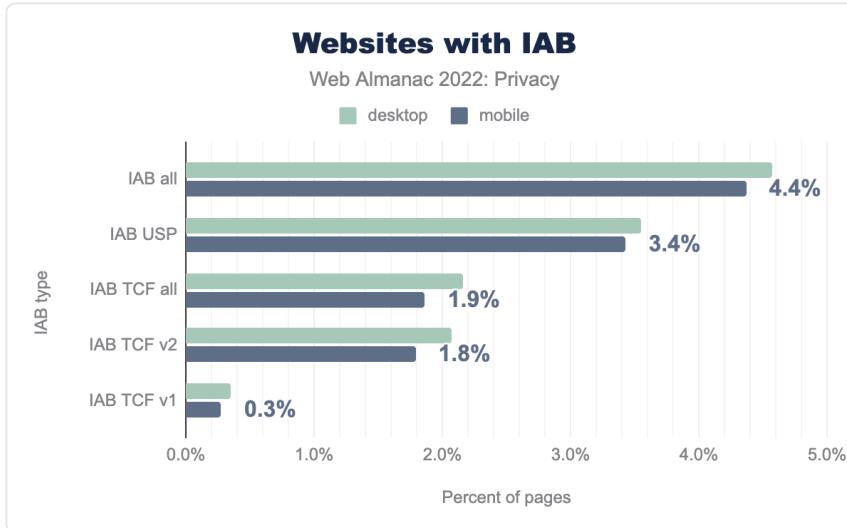


Figure 13.21. Websites with IAB.

We record that 4.6% of desktop websites use any IAB, with 3.5% using USP and 2.2% using IAB. Thus, we have recorded an increase for both specifications since last year. We would like to note here that our measurement is USA-based, so according to TCF, no consent banner is required for non-EU visits. So this can be the reason why we identify more websites with USP.

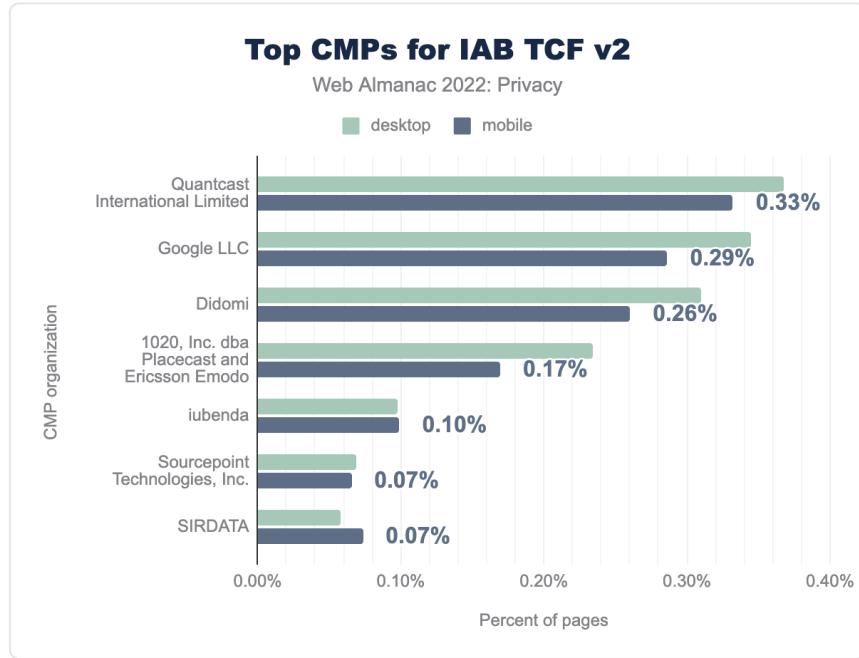


Figure 13.22. Top CMPs for IAB TCF v2.

We see that Quantcast International Limited (0.37%), Google LLC (0.34%) and Didomi (0.31%) are popular CMP providers for IAB TCF v2.

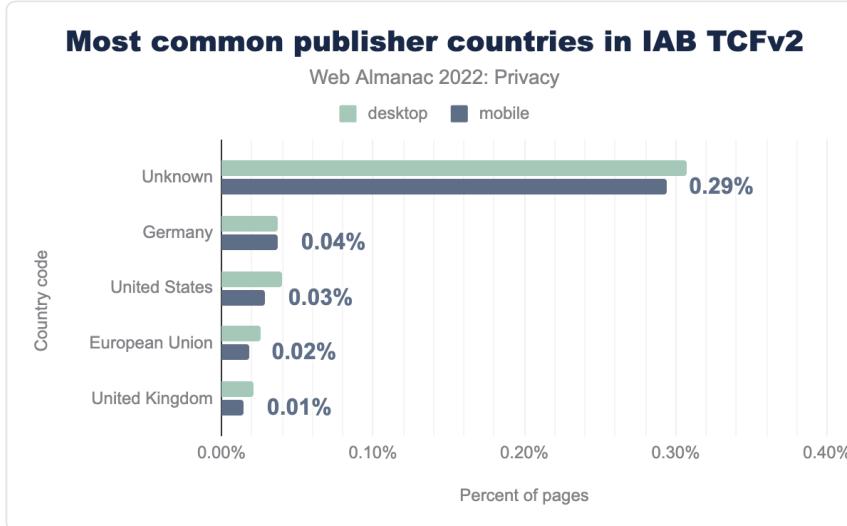


Figure 13.23. Most common publisher countries in IAB TCF v2.

Our analysis shows that the most common publishers we identified are from Germany, the US, and the EU.

Privacy policy

Notifications regarding data processing do not always take place via a consent banner. They are also usually described in more detail on separate pages compared to such banners. On such pages, you will find information on integrated third parties, which data is processed for which purpose, etc. To identify such sites, we used the privacy-relevant signatures from a study⁵²⁰. Using this method, we could determine that 45% of desktop websites (41% on mobile) contained a link on their homepage to a privacy-related page. The figure below shows the distribution of the top privacy link keywords.

520. https://github.com/RUB-SysSec/we-value-your-privacy/blob/master/privacy_wording.json

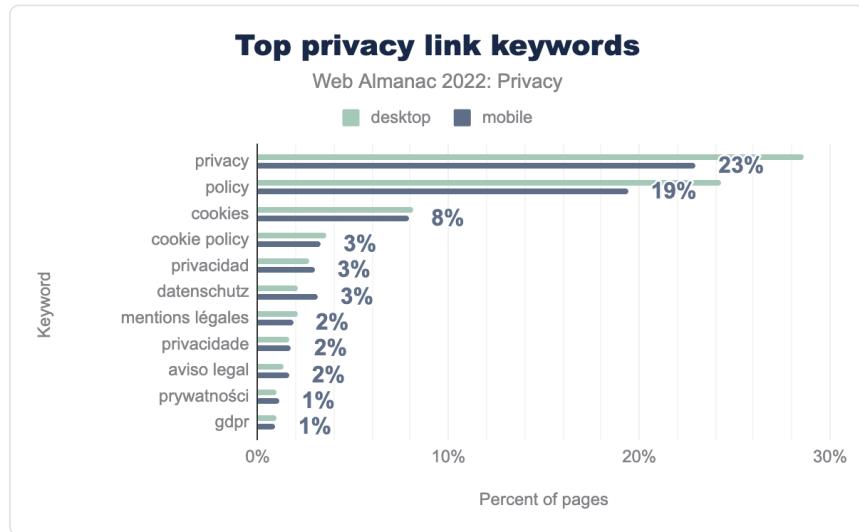


Figure 13.24. Top privacy link keywords.

We see that privacy (29%), policy (24%), and cookies (8%) are the top keywords for such links.

Conclusion

In this chapter, we explored many different aspects related to our online privacy on the web. It is clear that in the past year a considerable amount of things have changed that affect our privacy, and this progress can be expected to continue in the following years. In short, there are some exciting times ahead of us. On the one hand, we found some unfortunate evolutions, which hopefully one day we will be able to refer to as the web's legacy. Third-party tracking, mainly fueled by third-party cookies, is still ubiquitous with over 82% of websites containing at least a single tracker. Furthermore, there still is a non-negligible number of websites or web services that employ evasive techniques to circumvent anti-tracking measures.

On a more positive, privacy-preserving, track, we find that fewer sites are trying to access potentially sensitive information from browser APIs. Hopefully, this remains the case with the new APIs that are introduced in browsers on a regular basis.

Generally, it seems that websites are starting to hear the call of users to respect their privacy—a call that is getting louder and louder. More and more sites are switching to employing browser features that restrict the information that is sent to third parties. Furthermore, mainly motivated by privacy regulations such as GDPR and CCPA, we are seeing a clear increase—almost 60%—in the adoption of consent management platforms (CMPs),

giving users more control over which information they want to share.

Finally, on the side of the browsers, we are also seeing a strong evolution towards providing users with more control of their online privacy. Next to the features that several privacy-focused browsers offer as a built-in solution, there is also the Privacy Sandbox initiative that aims to continue providing the current functionalities on the web—such as targeted advertising, anti-fraud, attribution of purchases... etc.—without the nefarious side-effects of cross-site tracking. Although the development is still in fairly early stages, we see that web services on a substantial number of websites are already opting-in to the Origin Trial. As such, the features are extensively being tested, and are likely to become a persistent part of the web.

While it may still take a couple of years to finally get there, we are transitioning towards a web that gives users more control over what they want to share with which parties. We can see this convergence on both sides of the spectrum: on the one hand initiated by the website, and on the other hand enforced by the browser. We can be hopeful that in the not-so-distant future the data we share, is the data that we intend to share, and the journey on the web that we take on a day-to-day basis no longer needs to be collected, shared, and analyzed by the numerous trackers that we currently encounter—in the hope of respectfully tomorrow for all.

Authors



Tom Van Goethem

[@tomvangoethem](https://twitter.com/tomvangoethem) [tomvangoethem](https://github.com/tomvangoethem) <https://tom.vg/>

Tom Van Goethem recently joined the Chrome Privacy team at Google. Before, Tom was in PhD program with the DistriNet group of the University of Leuven, Belgium. His research interests cover a broad spectrum of topics in the field of web security and privacy, with a primary focus on side-channel attacks. By uncovering threats and proposing mitigations, Tom aims to make the web a nicer place, a tiny bit at a time.



Nurullah Demir

[@nrllah](https://twitter.com/nrllah) [nrllh](https://github.com/nrllh) <https://www.internet-sicherheit.de/team/demir-nurullah.html>

Nurullah Demir is a cyber security researcher and PhD student at Institute for Internet Security⁵²¹ and Intelligent System Security, KASTEL Security Research Labs⁵²². His research focuses on web security & privacy, and web measurements.

⁵²¹. <https://www.internet-sicherheit.de/en/>

⁵²². <https://intellisec.de>

Part II Chapter 14

Security



Written by Saptak Sengupta, Liran Tal, and Brian Clark

Reviewed by Kushal Das and Barry Pollard

Analyzed by Victor Le Pochat, Vik Vanderlinden, and Gertjan Franken

Edited by Barry Pollard

Introduction

As people's personal details continue to become more digital, security and privacy are becoming extremely crucial across the internet. It's the website owner's responsibility that they can secure the data they are taking from the user. Hence, it is essential for them to adopt all the security best practices to ensure protection of the user against vulnerabilities that malwares can exploit to get sensitive information.

Like previous years⁵²³, we have analyzed the adoption and usage of security methods and best practices by the web community. We have analyzed metrics related to the bare essential security measures that every website should adopt such as transport security and proper cookie management. We have also discussed the data related to the adoption of different security headers and how they help in content inclusion and preventing various malicious attacks.

523. <https://almanac.httparchive.org/en/2021/security>

We looked at correlations for adoption of security measures with the location, technological stack and website popularity. We hope that such correlations encourage all technological stacks to aim for better security measures by default. We also discuss some well-known URIs that help towards vulnerability disclosure and other security related settings based on Web Application Security Working Group's standards and drafts.

Transport security

Transport Layer Security ensures secure communication of data and resources between the user and the websites. HTTPS⁵²⁴ uses TLS⁵²⁵ to encrypt all communication between the client and the server.

A large, bold, blue percentage value '94%' is displayed, likely representing the percentage of requests made over HTTPS on desktop.

Figure 14.1. Requests that use HTTPS on desktop.

94% of total requests in desktop and 93% of total requests in mobile are made over HTTPS. All major browsers now have an HTTPS-only mode⁵²⁶ to show warning if a website uses HTTP instead of HTTPS.

524. <https://developer.mozilla.org/docs/Glossary/https>
525. <https://www.cloudflare.com/en-gb/learning/ssl/transport-layer-security-tls/>
526. <https://support.mozilla.org/en-US/kb/https-only-prefs>

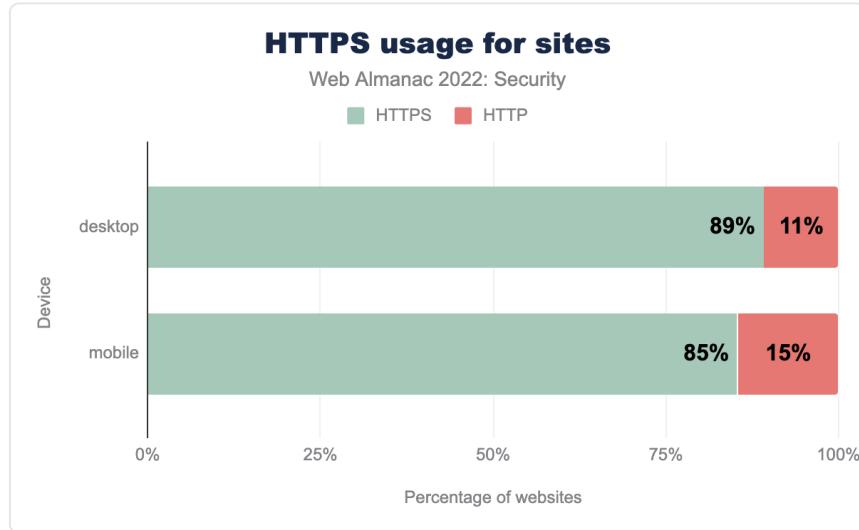


Figure 14.2. HTTPS usage for sites.

The percentage of homepages that are served over HTTPS continues to be lower compared to the total requests because a lot of the requests to a website are dominated by third-party services like fonts, CDN, etc. which have a higher HTTPS adoption. We do see a slight increase from last year in the percentage. 89.3% of homepages are now served over HTTPS on desktop compared to 84.3% last year. Similarly, in our mobile dataset, 85.5% of homepages are served over HTTPS compared to 81.2% last year.

Protocol versions

It's important, not only to use HTTPS, but also to use an up-to-date TLS version. The TLS working group⁵²⁷ has deprecated TLS v1.0 and v1.1 since they had multiple weaknesses. Since the last year's chapter, Firefox has now updated its UI⁵²⁸ and the option to enable TLS 1.0 and 1.1 has been removed from the error page in Firefox version 97. Chrome has also stopped allowing bypassing⁵²⁹ the error page which is shown for TLS 1.0 and 1.1 since version 98.

TLS v1.3 is the latest and was released in August 2018 by IETF. It's much faster and is more secure⁵³⁰ than TLS v1.2. Many of the major vulnerabilities in TLS v1.2 had to do with older cryptographic algorithms which TLS v1.3 removes.

527. <https://datatracker.ietf.org/doc/rfc8996/>

528. https://support.mozilla.org/en-US/kb/secure-connection-failed-firefox-did-not-connect#w_tls-version-unsupported

529. <https://chromestatus.com/feature/5759116003770368>

530. <https://www.cloudflare.com/en-in/learning/ssl/why-use-tls-1.3/>

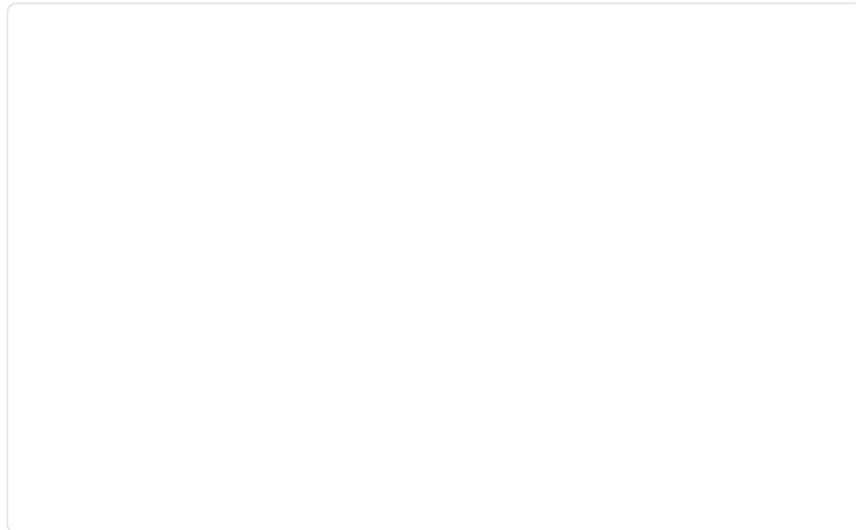


Figure 14.3. TLS versions usage for sites.

In the above graph, we see that 70% of homepages in mobile and 67% of homepages in desktop are served over TLSv1.3 which is approximately 7% more than last year. So, we are seeing some constant shift from use of TLS v1.2 to TLS v1.3

Cipher suites

A cipher suite⁵³¹ is a set of cryptographic algorithms that the client and server must agree on before they can begin communicating securely using TLS.

531. <https://learn.microsoft.com/en-au/windows/win32/secauthn/cipher-suites-in-schannel>

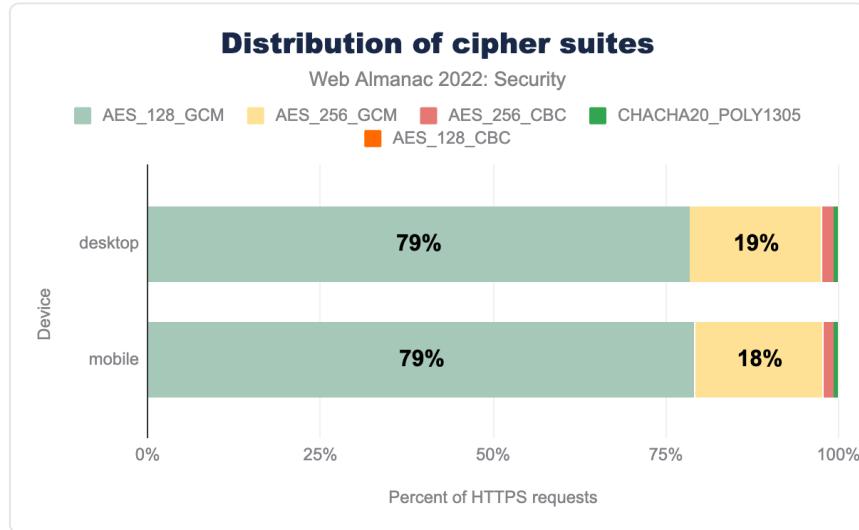


Figure 14.4. Distribution of cipher suites.

Modern Galois/Counter Mode (GCM)⁵³² cipher modes are considered to be much more secure since they are not vulnerable to padding attacks⁵³³. TLS v1.3 only supports GCM and other modern block cipher modes⁵³⁴ making it more secure. It also removes the trouble of cipher suite ordering⁵³⁵. Another factor that determines the usage of a cipher suite is the key size for the encryption and decryption. We still see 128-bit key size being widely used. Hence, we don't see much difference from last year's graph with `AES_128_GCM` continuing to be the most used cipher suite with 79% usage.

532. https://en.wikipedia.org/wiki/Galois/Counter_Mode

533. <https://blog.qualys.com/product-tech/2019/04/22/zombie-poodle-and-goldendoodle-vulnerabilities>

534. <https://datatracker.ietf.org/doc/html/rfc8446#page-133>

535. <https://go.dev/blog/tls-cipher-suites>

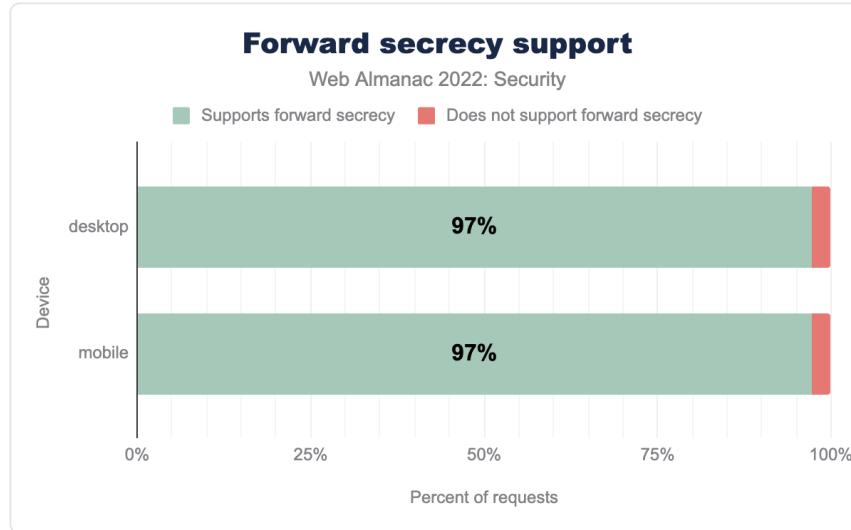


Figure 14.5. Forward secrecy usage.

TLS v1.3 also makes forward secrecy⁵³⁶ compulsory. We see 97% of requests in both mobile and desktop using forward secrecy.

Certificate Authority

A Certificate Authority⁵³⁷ or CA is a company or organization that issues the TLS certificate to the websites that can be recognized by browsers and then establish a secure communication channel with the website.

536. https://en.wikipedia.org/wiki/Forward_secrecy
 537. <https://www.ssl.com/faqs/what-is-a-certificate-authority/>

Issuer	Desktop	Mobile
R3	48%	52%
Cloudflare Inc ECC CA-3	13%	12%
Sectigo RSA Domain Validation Secure Server CA	7%	8%
cPanel, Inc. Certification Authority	5%	5%
Amazon	3%	3%
Go Daddy Secure Certificate Authority - G2	3%	2%
DigiCert SHA2 Secure Server CA	2%	1%
RapidSSL TLS DV RSA Mixed SHA256 2020 CA-1	1%	1%
E1	1%	1%

Figure 14.6. Top 10 certificate issuers for websites.

Let's Encrypt⁵³⁸ (or R3) continues to lead the charts with 48% of websites in desktop and 52% of websites in mobile using certificates issued by them. Let's Encrypt being a non-profit organization has played an important role in the adoption of HTTPS and they continue to issue an increasing number of certificates⁵³⁹. We would also like to take a moment to recognize one of its founders, Peter Eckersley⁵⁴⁰, who unfortunately passed away recently.

Cloudflare⁵⁴¹ continues to be in second position with its similarly free certificates for its customers. Cloudflare CDNs increase the usage of Elliptic Curve Cryptography (ECC)⁵⁴² certificates which are smaller and more efficient than RSA certificates but are often difficult to deploy, due to the need to also continue to serve non-ECC certificates to older clients. We see as Let's Encrypt and Cloudflare's percentage continues to increase, the percentage for usage of other CAs are decreasing a little.

HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS)⁵⁴³ is a response header that informs the browser to automatically convert all attempts to access a site using HTTP to HTTPS requests instead.

538. <https://letsencrypt.org/>

539. <https://letsencrypt.org/stats/#daily-issuance>

540. <https://community.letsencrypt.org/t/peter-eckersley-may-his-memory-be-a-blessing/183854>

541. <https://developers.cloudflare.com/ssl/ssl-tls/certificateAuthorities/>

542. <https://www.digicert.com/faq/ecc.htm>

543. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Strict-Transport-Security>

25%

Figure 14.7. Mobile requests that have an HSTS header.

25% of the mobile responses and 28% of desktop responses have an HSTS header.

HSTS is set using the `Strict-Transport-Security` header that can have three different directives: `max-age`, `includeSubDomains`, and `preload`. `max-age` helps denote the time, in seconds, that the browser should remember to access the site only using HTTPS. `max-age` is a compulsory directive for the header.

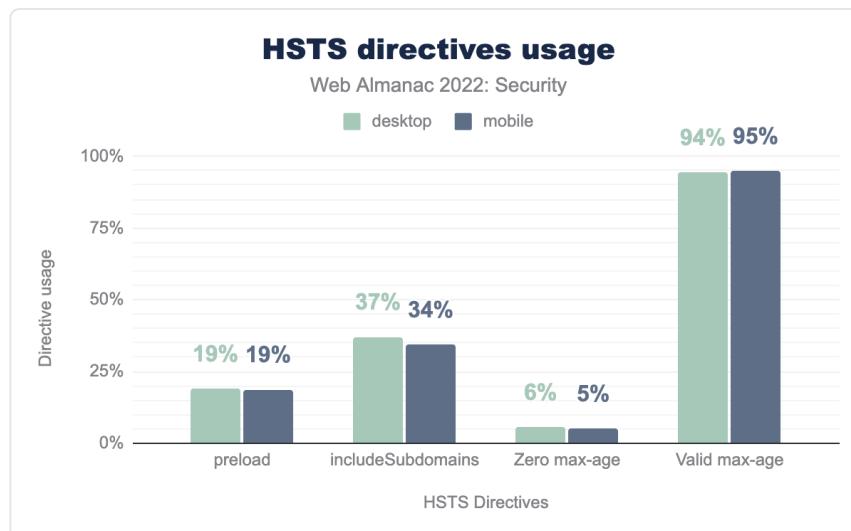


Figure 14.8. Usage of different HSTS directives.

We see 94% of sites in desktop and 95% of sites in mobile have a non-zero, non-empty `max-age`.

34% of request responses for mobile, and 37% for desktop include `includeSubdomain` in the HSTS settings. The number of responses with the `preload` directive, which is not part of the HSTS specification⁵⁴⁴ is lower. It needs a minimum `max-age` of 31,536,000 seconds (or 1 year) and also the `includeSubdomain` directive to be present.

544. https://developer.mozilla.org/docs/Web/HTTP/Headers/Strict-Transport-Security#preloading_strict_transport_security

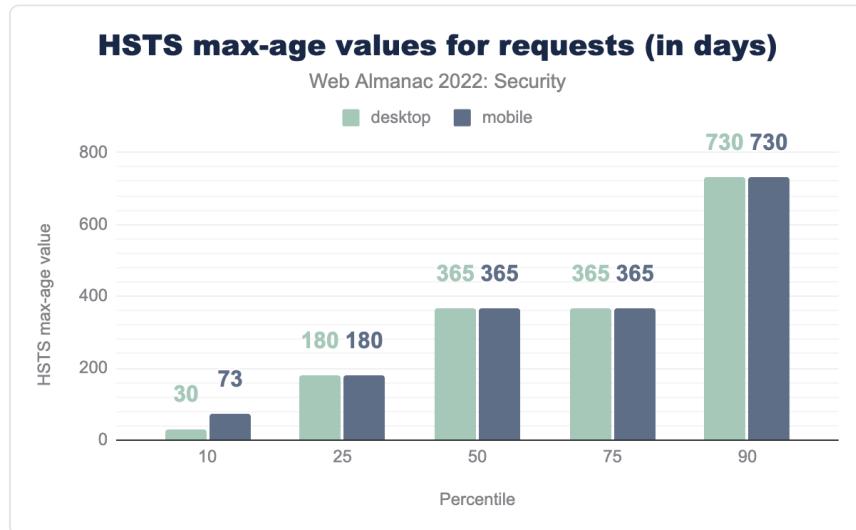


Figure 14.9. HSTS `max-age` values for all requests (in days).

The median value for `max-age` attribute in HSTS headers over all requests is 365 days in both mobile and desktop. <https://hstspreload.org/> recommends a `max-age` of 2 years once the HSTS header is set up properly and verified to not cause any issues.

Cookies

An HTTP cookie⁵⁴⁵ is a set of data about the user that the server sends to the browser. A cookie can be used for things like session management, personalization, tracking and other stateful information related to the user over different requests.

If a cookie is not set properly, it can be susceptible to many different forms of attacks such as session hijacking⁵⁴⁶, Cross-Site Request Forgery (CSRF)⁵⁴⁷, Cross-Site Script Inclusion (XSSI)⁵⁴⁸ and various other Cross-Site Leak⁵⁴⁹ vulnerabilities.

Cookie attributes

To defend against the above mentioned threats, developers can use 3 different attributes in a

545. <https://developer.mozilla.org/docs/Web/HTTP/Cookies>

546. https://owasp.org/www-community/attacks/Session_hijacking_attack

547. <https://owasp.org/www-community/attacks/csrf>

548. https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/11-Client_Side_Testing/13-Testing_for_Cross_Site_Script_Inclusion

549. <https://xsleaks.dev/>

cookie: `HttpOnly`, `Secure` and `SameSite`. The `Secure` attribute is similar to the `HSTS` header as it also ensures that the cookies are always sent over HTTPS, preventing Manipulator in the Middle attacks⁵⁵⁰. `HttpOnly` ensures that a cookie is not accessible from any JavaScript code, preventing Cross-Site Scripting⁵⁵¹ Attacks.

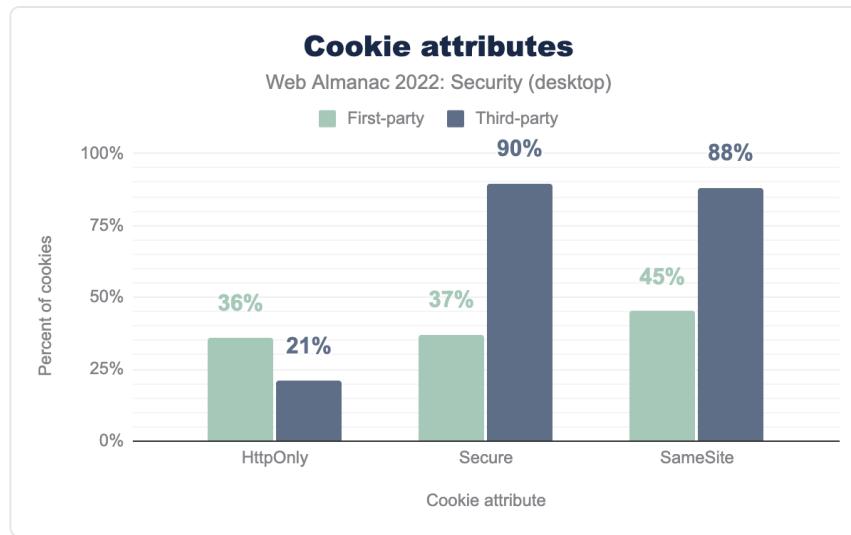


Figure 14.10. Cookie attributes (desktop).

There are 2 different kinds of cookies: first-party and third-party. First-party cookies are usually set by the direct server that you are visiting. Third-party cookies are created by third-party services and are often used for tracking and ad-serving. 37% of the first-party cookies on the desktop have `Secure` and 36% of them have `HttpOnly`. However, in the third party cookies we see that 90% of cookies have `Secure` and 21% of cookies have `HttpOnly`. The percentage of `HttpOnly` is less in third party cookies, because they mostly do want to allow access to them by JavaScript code.

550. https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack
 551. <https://owasp.org/www-community/attacks/xss/>

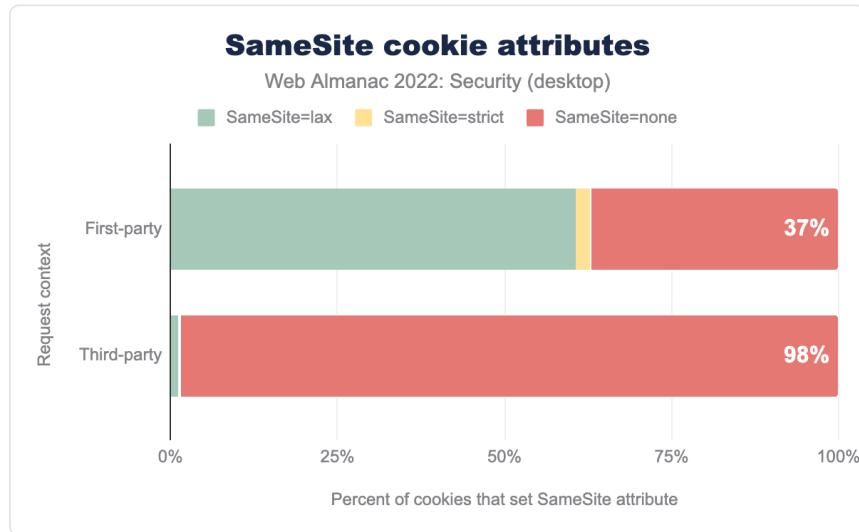


Figure 14.11. Same site cookie attributes.

The `SameSite` attribute can be used to prevent CSRF attacks by telling the browser whether to send the cookie to cross-site requests. `Strict` value allows the cookie to be sent only to the site where it originated while `Lax` value allows cookies to be sent to cross-site requests only if a user is navigating to the origin site by following a link. For `None` value, cookies are sent to both originating and cross-site requests. If `SameSite=None` is set, the cookie's `Secure` attribute must also be set (or the cookie will be blocked). We see that 61% of first-party cookies with the `SameSite` attribute have the value `Lax`. Most browsers default to `SameSite=Lax` if no `SameSite` attribute is present for the cookie hence we see that it continues to dominate the charts. In third party cookies, `SameSite=None` still continues to be super high with 98% cookies in desktop, because third-party cookies do want to be sent across cross-site requests.

Cookie age

There are two different ways to set the time when a cookie is deleted: `Max-Age` and `Expires`. `Expires` uses a specific date (relative to the client) to determine when the cookie is deleted whereas `Max-Age` uses a duration in seconds.

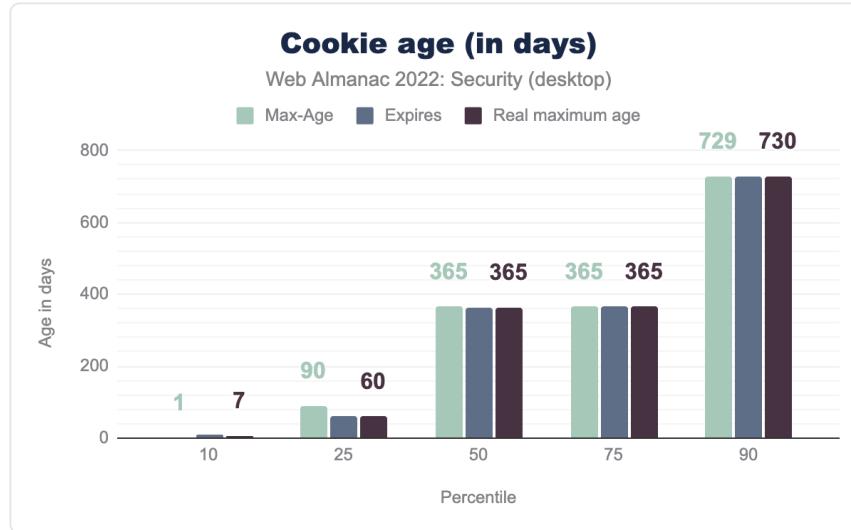


Figure 14.12. Cookie age usage in days (desktop).

Unlike last year, where we saw that the median for `Max-Age` was 365 days but the median for `Expires` was 180 days, this year it's around 365 days for both. Hence the median for real maximum age has gone up from 180 days to 365 days this year. Even though the `Max-Age` is 729 days and `Expires` is 730 days in the 90th percentile, Chrome has been planning to put a cap of 400 days⁵⁵² for both `Max-Age` and `Expires`.

%	<code>Expires</code>
1.8%	"Thu, 01-Jan-1970 00:00:00 GMT"
1.2%	"Fri, 01-Aug-2008 22:45:55 GMT"
0.7%	"Mon, 01-Mar-2004 00:00:00 GMT"
0.7%	"Thu, 01-Jan-1970 00:00:01 GMT"
0.3%	"Thu, 01 Jan 1970 00:00:00 GMT"

Figure 14.13. Most common cookie expiry values on desktop.

⁵⁵² <https://chromestatus.com/feature/4887741241229312>

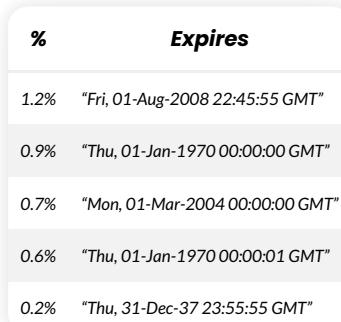


Figure 14.14. Most common cookie expiry values on mobile.

The most prevalent `Expires` has some interesting values. We see that the most used `Expires` value in Desktop is `January 1, 1970 00:00:00 GMT`. When cookies `Expires` value is set to a past date, they are deleted from the browser. January 1, 1970 00:00:00 GMT is the Unix epoch time and hence it's often commonly used to expire or delete a cookie.

Content inclusion

A website's content often takes many shapes and requires resources such as CSS, JavaScript, or other media assets like fonts and images. These are frequently loaded from external service providers of the likes of remote storage services of cloud-native infrastructure, or from content delivery networks (CDNs) with the aim of reducing worldwide networking round-trips just to serve the content.

However, ensuring that the content we include on the website hasn't been tampered with is of high stakes, and the impact of which can be devastating. Content inclusion is of even higher importance these days given the recent rise of awareness to supply chain security, and growing incidents of Magecart attacks⁵⁵³ that target website content systems to inject persistent malware through means of cross-site scripting (XSS) vulnerabilities and others.

Content Security Policy

One effective measure you can deploy to mitigate security risks around content inclusion is by employing a Content Security Policy (CSP). It is a security standard that adds a defense-in-depth layer in order to mitigate attacks such as code injection via cross-site scripting, or clickjacking, to name a few.

^{553.} <https://www.imperva.com/learn/application-security/magecart/>

It works by ensuring that a predefined trusted set of content rules is upheld and any attempts to bypass or include restricted content are rejected. For example, a content security policy that would allow JavaScript code to run in the browser only from the same origin it was served, and from that of Google Analytics would be defined as `script-src 'self' www.google-analytics.com;`. Any attempts of cross-site scripting injections that add inline JavaScript code such as `` would be rejected by the browser enforcing the set policy.

+14%

Figure 14.15. Relative increase in adoption for Content-Security-Policy header from 2021.

We're seeing a 14% relative increase in adoption for `Content-Security-Policy` header from 2021's data of 12.8% to 2022's data of 14.6% which demonstrates a growing trend of adoption across developers and the web security community. This is positive, though it's still a minority of sites using this more advanced feature.

CSP is most useful, when served on the HTML response itself and here we're seeing consistent growing adoption in mobile requests serving a CSP header with 7.2% two years ago, 9.3% last year, and this year a total of 11.2% of mobile homepages.

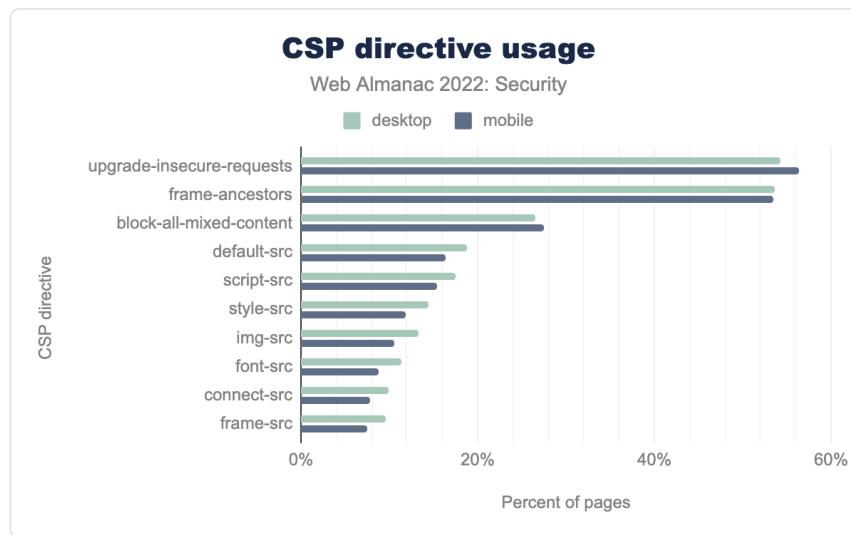


Figure 14.16. Most common directives used in CSP.

The top three CSP directives that we're seeing serving more than a quarter of the HTTP requests for both desktop and mobile are `upgrade-insecure-requests` at a 54%, `frame-ancestors` at 54%, and the `block-all-mixed-content` policy at 27%. Trailing policies are `default-src`, `script-src`, `style-src`, and `img-src` to name a few.

The `upgrade-insecure-requests` policy's high adoption rate could perhaps be attributed to the high adoption of TLS requests as a de-facto standard. However, despite `block-all-mixed-content` being considered deprecated as of this date, it's showing a high adoption rate. This perhaps speaks to the fast rate at which the CSP specification is advancing and users having a hard time keeping up to date.

More to do with mitigating cross-site scripting attacks is Google's security initiative for Trusted Types⁵⁵⁴, which requires native browser API support to employ a technique which helps prevent DOM-injection class of vulnerabilities. It is actively advocated by Google engineers yet is still in draft proposal mode⁵⁵⁵ for the W3C. Nonetheless, we record its CSP related security headers `require-trusted-types-for` and `trusted-types` at 0.1% of requests which is not a lot, but perhaps speaks to a growing trend of adoption.

To assess whether a CSP violation from the pre-defined set of rules is occurring, websites can set the `report-uri` directive that the browser will send JSON formatted data as an HTTP POST request. Although `report-uri` requests account for 4.3% of all desktop traffic with a CSP header, it is to date a deprecated directive and has been replaced with `report-to` which accounts for 1.8% of desktop requests.

One of the biggest contributors to the challenge of implementing a tight content security policy is the existence of inline JavaScript code that's commonly set as event handlers or at other parts of the DOM. To allow teams a progressive adoption of the CSP security standard, a policy may set `unsafe-inline` or `unsafe-eval` as keyword values for its `script-src` directive. Doing so, fails to prevent some cross-site scripting attack vectors and is counter-productive to the preventative measure of a policy.

Teams can utilize a more secure posture of inline JavaScript code by signing them with a nonce or a SHA256 hash. This would look like something along the lines of:

```
Content-Security-Policy: script-src 'nonce-4891cc7b20c'
```

And then referencing that in the HTML:

⁵⁵⁴. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Content-Security-Policy/trusted-types>

⁵⁵⁵. <https://w3c.github.io/trusted-types/dist/spec/>

```
<script nonce="nonce-4891cc7b20c">
...
</script>
```

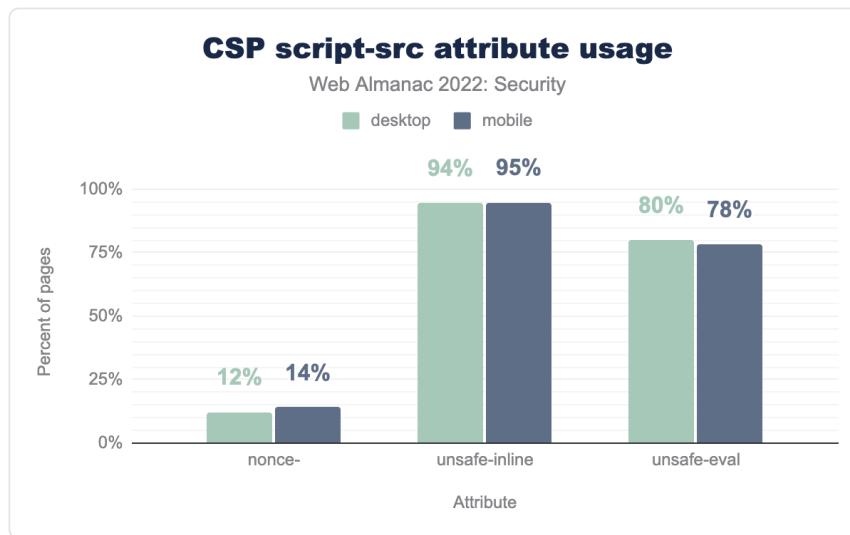


Figure 14.17. CSP `script-src` attribute usage.

The statistics collected for all desktop HTTP requests show that `unsafe-inline` is present on for 94%, and `unsafe-eval` on 80% of all `script-src` values. This demonstrates the real challenges in locking down a website's application code to avoid inline JavaScript code. Furthermore, Only 14% of all above described requests employ a `nonce-` directive which assists in securing the use of unsafe inline JavaScript code.

Perhaps speaking to the high complexity of defining a content security policy is the statistics we observe for CSP header length.

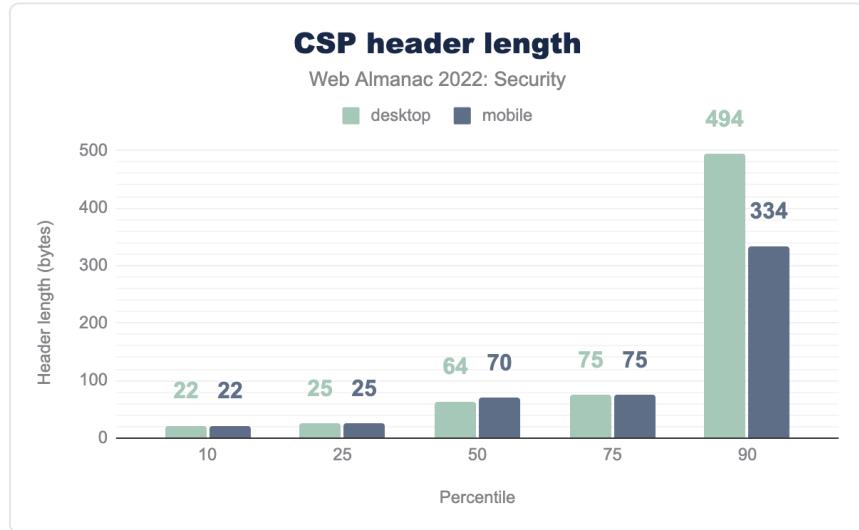


Figure 14.18. CSP header length.

At a median ranking, 50% of requests are only up to 70 bytes in size for desktop requests. This is a slight drop from last year's report which showed both desktop and mobile requests at 75 bytes in size. The 90th percentile of requests and above has grown from last year's 389 bytes for desktop requests, to 494 bytes this year. This demonstrates a slight progress towards more complex and complete security policies.

Observing the complete definitions for a content security policy, we can see that single directives still make up a large proportion of all requests. 19% of all desktop requests are set only to `upgrade-insecure-requests`. 8% of all desktop requests are set to `frame-ancestors 'self'` and 23% of all desktop requests are set to the value of `block-all-mixed-content; frame-ancestors 'none'; upgrade-insecure-requests;` which mixes together the top 3 most common CSP directives.

The content security policy often has to allow content from other origins than its own in order to support loading of media such as fonts, ad related scripts, and general content delivery network usage. As such, the top 10 origins across requests are as follows:

Origin	Desktop	Mobile
https://www.google-analytics.com	0.39%	0.26%
https://www.googletagmanager.com	0.37%	0.25%
https://fonts.gstatic.com	0.27%	0.19%
https://fonts.googleapis.com	0.27%	0.18%
https://www.google.com	0.24%	0.17%
https://www.youtube.com	0.21%	0.15%
https://stats.g.doubleclick.net	0.19%	0.13%
https://connect.facebook.net	0.18%	0.13%
https://www.gstatic.com	0.17%	0.12%
https://cdnjs.cloudflare.com	0.16%	0.11%

Figure 14.19. Most frequently allowed hosts in CSP policies.

The above hosts account for roughly the same positioning in rank as was reported last year, but the usage is up slightly.

The CSP security standard is widely supported both by web browsers, as well as content delivery networks and content management systems and is a highly recommended tool for websites and web applications in defense of web security vulnerabilities.

Subresource Integrity

Another defense-in-depth tool is Subresource Integrity which provides a web security defensive layer against content tampering. Whereas a Content Security Policy defines which types and source of content are allowed, a Subresource Integrity mechanism ensures that said content hasn't been modified for malicious intents.

A reference use-case for using Subresource Integrity is when loading JavaScript content from third-party package managers which also act as a CDN. Some examples of these are unpkg.com or cdnjs.com, both of which serve the content source for JavaScript libraries.

If a third-party library could be compromised due to a hosting issue by the CDN provider, or by one of the project's contributors or maintainers then you are effectively loading someone else's code into your website.

Similar to CSP's use of a `nonce-`, Subresource Integrity (also known as SRI) allows browsers to validate the content that is served matches a cryptographically signed hash and prevents tampering with the content, whether over the wire or at its source.

20%

Figure 14.20. Desktop sites using SRI.

Just about one of every fifth website (20%) adopts a subresource integrity in one of its web page elements on desktop. Out of these, 83% were specifically used in `<script>` type elements on desktop, and 17% were used in `<link>` type elements in desktop requests.

At a per page coverage, adoption rate for the SRI security feature is still considerably low. Last year, the median percentage for both mobile and desktop was 3.3% and this year it decreased by 2% to 3.23%.

Subresource integrity is specified as a base64 string of a computed hash of one of SHA256, SHA384 or SHA512 cryptographic functions. As a use-case reference⁵⁵⁶, developers can implement them as follows:

```
<script src="https://example.com/example-framework.js"
integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/
uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"
crossorigin="anonymous"></script>
```

⁵⁵⁶. https://developer.mozilla.org/docs/Web/Security/Subresource_Integrity

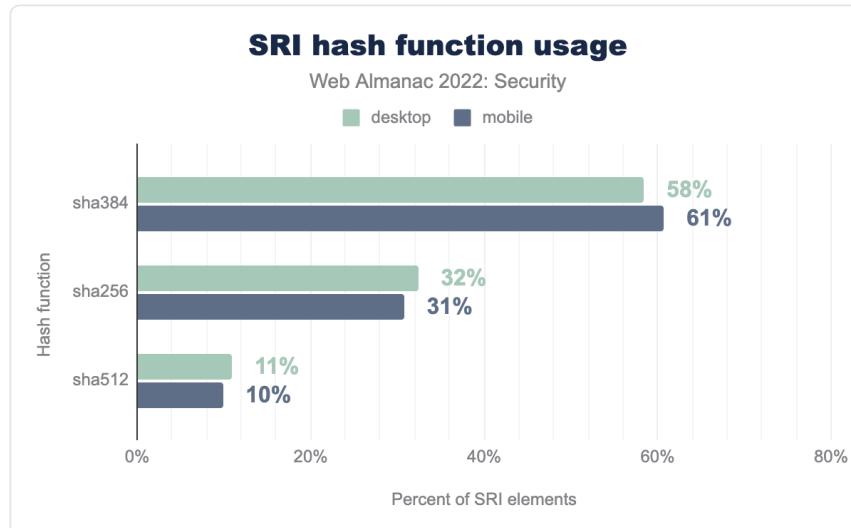


Figure 14.21. SRI hash functions.

Consistent with last year's report, SHA384 continues to demonstrate the majority of SRI hash types observed between all available hash functions.

CDNs are no strangers to Subresource Integrity and provide secure defaults to their consumers by including a Subresource Integrity value as part of their URL references for content to be served on the page.

Host	Desktop	Mobile
www.gstatic.com	39%	40%
cdn.shopify.com	22%	23%
cdnjs.cloudflare.com	8%	7%
code.jquery.com	7%	7%
static.cloudflareinsights.com	5%	4%
cdn.jsdelivr.net	3%	3%
t1.daumcdn.net	3%	1%
stackpath.bootstrapcdn.com	2.7%	2.7%
maxcdn.bootstrapcdn.com	2.2%	2.3%

Figure 14.22. Most common hosts from which SRI-protected scripts are included.

The above list shows the top 10 most common hosts for which a subresource integrity value has been observed. Notable changes from last year are the Cloudflare hosts jumping from position 4 to position 3, and jsDelivr jumping from position 7 to position 6 in ranking, surpassing Bootstrap's hosts rankings.

Permissions Policy

Browsers are becoming more and more powerful with time, adding more native APIs to access and control different sorts of hardware and feature sets that are made available to websites. These also introduce potential security risks to users through misuse of said features, such as malicious scripts turning on a microphone and collecting data, or fingerprinting geolocation of a device to collect location information.

Previously known as `Feature-Policy`, and now named `Permissions-Policy`, this is an experimental browser API that enables control to an allowlist and a denylist of a wide array of capabilities a browser is able to access.

We've noticed a high correlation of usage for the `Permissions-Policy` with HTTPS-enabled connections (97%), with `X-Content-Type-Options` (82%), and `X-Frame-Options` (78%). All correlations are across desktop requests. Another high correlation is within the specific technology intersection, observed for Google My Business mobile pages (99%), and the next closest is Acquia's Cloud Platform (67%). All correlations are across mobile requests.

+85%

Figure 14.23. Relative increase in adoption of `Permissions-Policy` from 2021.

We're seeing an 85% relative increase in adoption for `Permissions-Policy` from 2021's data (1.3%) to 2022's data (2.4%) for mobile requests and similar trend for desktop requests too. The deprecated `Feature-Policy` shows a minuscule increase of 1 percentage point between last year's data and this year's which demonstrates that users are keeping pace with web browsers' specification changes.

Besides being used as an HTTP header, this feature can be used within `<iframe>` elements as follows:

```
<iframe src="https://example.com" allow="geolocation 'src'  
https://example.com'; camera *"></iframe>
```

18.9% of 11.5 million frames in mobile contained the `allow` attribute to enable permission or feature policies.

The following is a list of the top 10 `allow` directives that were detected in frames:

Directive	Desktop	Mobile
<code>encrypted-media</code>	75%	75%
<code>autoplay</code>	48%	49%
<code>picture-in-picture</code>	31%	31%
<code>accelerometer</code>	26%	27%
<code>gyroscope</code>	26%	27%
<code>clipboard-write</code>	21%	21%
<code>microphone</code>	9%	9%
<code>fullscreen</code>	8%	7%
<code>camera</code>	6%	7%
<code>geolocation</code>	5%	6%

Figure 14.24. Prevalence of `allow` directives on iframes.

Interesting to point out are places 11th, 12th and 13th allow directives for mobile that didn't make it into the above list and they are `vr` with 6%, `payment` with 2%, and `web-share` with 1%. They perhaps speak of growing trends we're seeing in the industry around virtual reality (and the metaverse, if you will), online payments and the fintech industry. Lastly, it seems to indicate better support for web-based sharing which is presumably due to the rise of work-from-home habits in the last couple of years.

Iframe Sandbox

Using iframe elements in websites has been a long-time practice for developers in order to easily embed third-party content such as rich media, cross-application components, or even ads. Some may even assume that iframe elements form a security boundary between the website embedding them to the sourced website, however that's not exactly the case.

HTML `<iframe>` elements by default have access to top-level page capabilities such as pop-ups or direct interaction with the top-page browser navigation. For example, the following code could be embedded in the source of an iframe element which makes use of active user gestures and results in the hosting website of the iframe to navigate to a new URL at <https://example.com>:

```
function clickToGo() {
  window.open('https://example.com')
}
```

This is largely known as a clickjacking attack and is one of many other security risks that are embedded within iframes (pun intended).

To mitigate these concerns the HTML specification (version 5) introduced the `sandbox` attribute that may be applied to iframe elements. It acts as an allowlist and if kept empty it essentially does not enable any capabilities within the iframe element. This results in no access to page interactivity like pop-ups, no permissions to run JavaScript code, and no access to cookies.

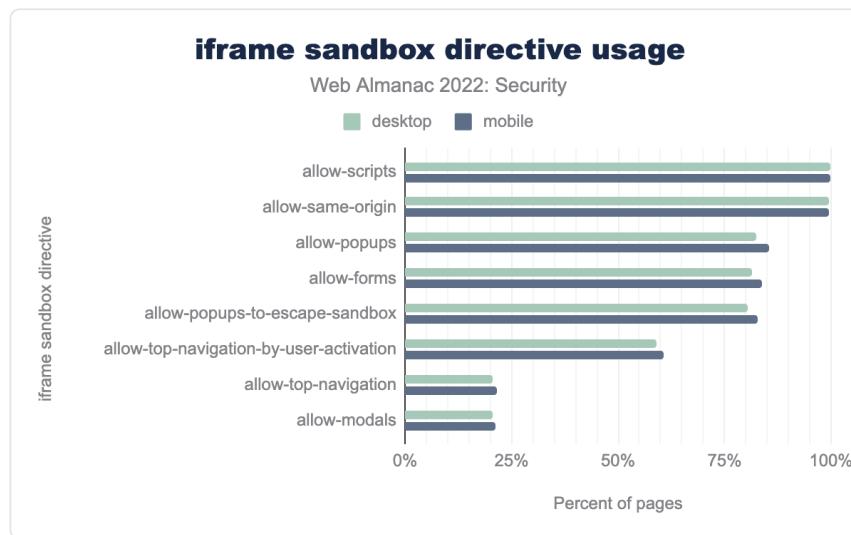


Figure 14.25. Prevalence of sandbox directives on frames.

The above chart of the 2022 data shows that more than 99% of websites with a `sandbox` attribute enable the `allow-scripts` and `allow-same-origin` permissions.

Of desktop websites that embed an iframe, 35.2% also include the `sandbox` attribute.

We find that `Content-Security-Policy` headers which include a `sandbox` directive are at a mere 0.3% usage for mobile (desktop is similar at 0.4%) which may speak to the fact that this attribute is only applied on a per-case basis for the practice of embedding iframe content within pages, rather than ahead-of-time planning through a content security policy definition.

Attack preventions

There are many different attacks that can exploit a website, and it's almost never possible to fully secure your website. However, there are many steps that a web developer can take to prevent most of these attacks, or to limit the effects of them on the users.

Security header adoptions

Security headers are one of the most common ways of preventing attacks by restricting the kind of traffic and data flow. But most of these security headers have to be set manually by website developers. Thus, the presence of security headers are often a good indication of the security hygiene that the developers of the website follow.

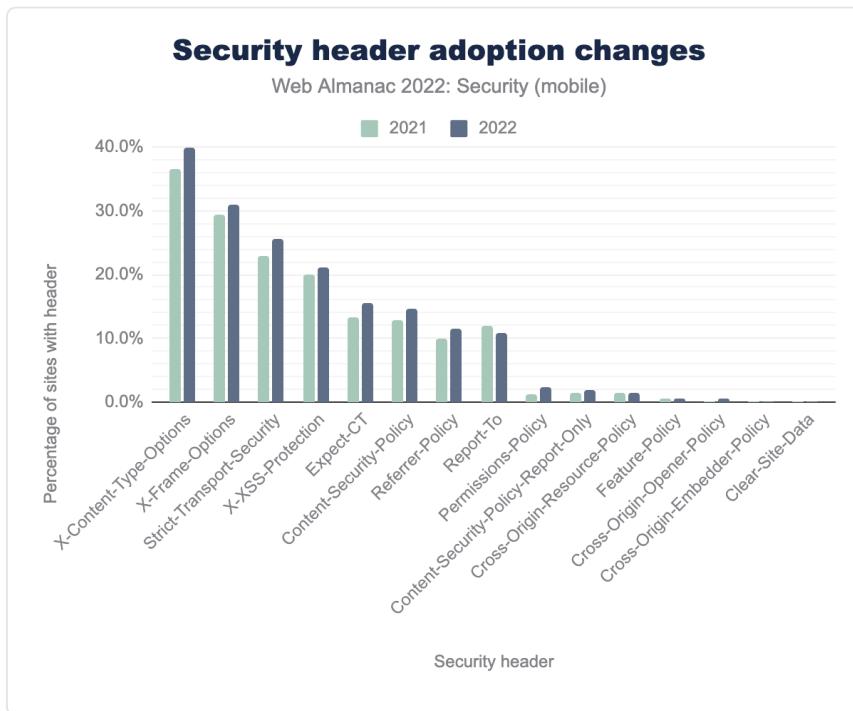


Figure 14.26. Adoption of security headers for site requests in mobile pages.

The most widely used security mechanism is still the X-Content-Type-Options header, which is used on 40% of the websites we crawled on mobile, to protect against MIME-sniffing attacks. This header is followed by the X-Frame-Options header, which is enabled on 30% of all sites. We don't see much difference from last year's data with a gradual increase in adoption of all the

security headers but the ranking of security headers by percentage usage is the same.

Preventing attacks using CSP

The main use of Content Security Policy (CSP) is to determine the trusted sources from which content can be loaded safely. This makes CSP a really useful security header in preventing various kinds of attacks such as clickjacking, cross-site scripting attacks, mixed-content inclusion and many more.

53%

Figure 14.27. The percentage of website on mobile with CSP that have frame-ancestors directive.

One of the common ways to prevent clickjacking attacks is to prevent the browser from loading the website in a frame. One can use the `frame-ancestors` directive in a CSP Header to restrict other domains from including the page content in a frame. We found 53% of the websites in mobile that have CSP have a `frame-ancestor` directive. It's the second most used CSP directive, which is good for prevention of clickjacking attacks. Setting the value of `frame-ancestors` directive to `none` or `self` is the safest. `none` doesn't allow any domain to frame the content, while `self` allows only the origin domain to frame the contents. We found that 8% of websites in mobile which have a CSP header have only `frame-ancestors 'self'` and is the third most common value of CSP header.

Keyword	Desktop	Mobile
<code>strict-dynamic</code>	6%	5%
<code>nonce-</code>	12%	14%
<code>unsafe-inline</code>	94%	95%
<code>unsafe-eval</code>	80%	78%

Figure 14.28. Prevalence of CSP keywords based on policies that define a `default-src` or `script-src` directive.

One of the mechanisms to defend against XSS attacks is by setting a restrictive `script-src` directive for the website. This ensures that JavaScript is loaded only from a trusted source and the attacker cannot inject some malicious code. `strict-dynamic` is gradually gaining more adoption across websites with 6% websites in desktop using it compared to 5% of websites last

year. `strict-dynamic` is helpful if you have a root script in your HTML that dynamically loads or injects other script files. It makes use of nonce or hash on the root script and ignores other allowlists like `unsafe-inline` or individual links. It's supported in all modern browsers apart from IE. Also, we see that `unsafe-inline` and `unsafe-eval` usage has decreased by approximately 2% from last year. This is a step in the right direction.

Preventing attacks using Cross-Origin policies

Cross Origin policies are one of the main mechanisms used to defend against micro-architectural attacks like Cross Site leaks. XS-Leaks are kind of similar to Cross Site Request Forgery, however they infer small pieces of information about the user which are exposed during interactions between websites.

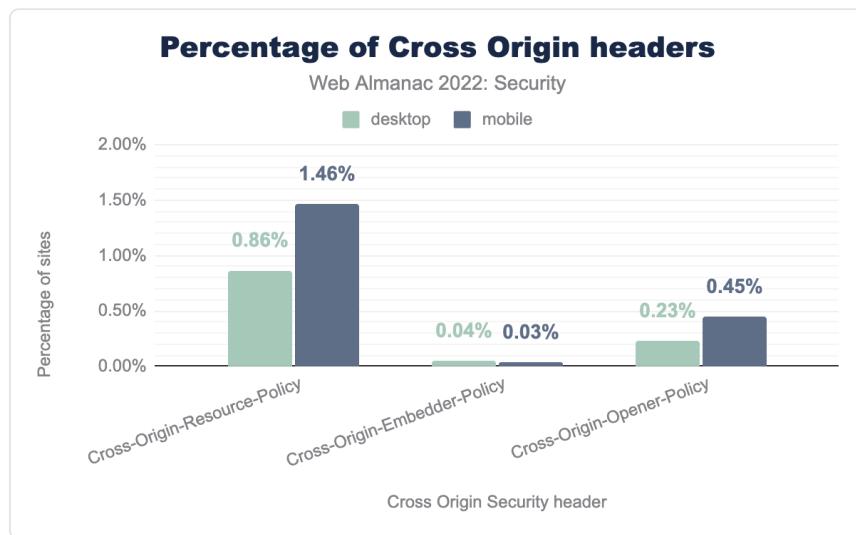


Figure 14.29. Percentage of Cross Origin headers.

`Cross-Origin-Resource-Policy` is present on 114,111(1.46%) websites in mobile and is the most used cross origin policy. It is used to indicate to the browser whether a resource should be included from cross-origin or not. `Cross-Origin-Embedder-Policy` is now present in 2,559 websites compared to 911 websites last year. We see a similar growth in the adoption of `Cross-Origin-Opener-Policy` as well with 34,968 websites in mobile now having the header compared to 15,727 sites last year. So there is a steady growth in the adoption of all the cross-origin policies, which is great because they can be really helpful in preventing XS-Leak attacks.

Preventing attacks using Clear-Site-Data

Clear-Site-Data⁵⁵⁷ provides web developers more control over clearing of user data related to their website. For example, a web developer can now make decisions such that when a user signs out of their web site, then all related cookie, cache, storage information about the user can be removed. This helps in limiting the consequences of an attack by having a restricted amount of data stored in the browser when not needed. This is a comparatively new header which is restricted only for sites served over HTTPS and only some of the features are supported by browsers⁵⁵⁸. There were only 75 sites in mobile which had `Clear-Site-Data` header in 2021 and it has increased to 428 this year. It is worth noting that often times websites use this header only in their logout pages, which are not tracked in the web almanac data.

CSD Header	Desktop	Mobile
<code>cache</code>	65%	63%
<code>*</code>	9%	8%
<code>"cache"</code>	7%	7%
<code>cookies</code>	3%	6%
<code>"storage"</code>	2%	1%
<code>cache,cookies,storage</code>	1%	1%
<code>"cache", "storage"</code>	1%	1%
<code>"*"</code>	1%	2%
<code>"cache", "cookies"</code>	1%	0%
<code>"cookies"</code>	1%	1%

Figure 14.30. Prevalence of `Clear-Site-Data` headers.

`cache` is the most prevalent directive (63% websites in mobile) for Clear-Site-Data which could mean that many developers are using this security header more for clearing cache to probably load newer static files, than for privacy and security of the user. But directives are supposed to follow quoted-string grammar⁵⁵⁹ and hence this directive is invalid. It is great to see that 9% of mobile websites using this header use `"*"` which means that they indicate the

557. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Clear-Site-Data>

558. https://developer.mozilla.org/docs/Web/HTTP/Headers/Clear-Site-Data#browser_compatibility

559. <https://datatracker.ietf.org/doc/html/rfc7230#section-3.2.6>

browser to clear all user data stored. Third most used directive is again "cache", but used properly this time.

Preventing attacks using `<meta>`

A Content-Security-Policy and Referrer-Policy can be set using a `<meta>` tag in the HTML code itself for a website. For example, one can set Content-Security-Policy using the code: `<meta http-equiv="Content-Security-Policy" content="default-src 'self'">`. We found that 0.47% and 2.60% of the websites in mobile enabled CSP and Referrer-Policy this way.

Meta tag values	Desktop	Mobile
include Referrer-Policy	3.11%	2.60%
include CSP	0.55%	0.47%
include note-allowed headers	0.08%	0.06%

Figure 14.31. Security headers used in `<meta>` tags.

The issue with preventing attacks using `<meta>` tag is if you set any other security headers using it, then the browser will ignore that security header. For example, 2,815 sites had X-Frame-Options in the `<meta>` tag. So the developer might be expecting the website to be secure against certain attacks since they added the `<meta>` tag when in reality, that security header never gets added. However, this number has gone down from 3,410 sites last year, so maybe websites are fixing this misuse of the `<meta>` tag.

Web Cryptography API

Web Cryptography API⁵⁶⁰ is a JavaScript API for performing basic cryptographic operations on a website such as random number generation, hashing, signing, encryption and decryption.

560. <https://www.w3.org/TR/WebCryptoAPI/>

Feature	Desktop	Mobile
<i>CryptoGetRandomValues</i>	69.6%	65.5%
<i>SubtleCryptoDigest</i>	0.6%	0.6%
<i>CryptoAlgorithmSha256</i>	0.5%	0.3%
<i>SubtleCryptoImportKey</i>	0.2%	0.1%
<i>SubtleCryptoGenerateKey</i>	0.2%	0.2%
<i>SubtleCryptoEncrypt</i>	0.2%	0.1%
<i>SubtleCryptoExportKey</i>	0.2%	0.1%
<i>CryptoAlgorithmAesGcm</i>	0.1%	0.1%
<i>SubtleCryptoSign</i>	0.2%	0.2%
<i>CryptoAlgorithmAesCtr</i>	0.1%	< 0.1%

Figure 14.32. Top used cryptography APIs.

There is not much change in the data from last year. `CryptoGetRandomValues` continues to be the most adopted feature (even though its usage has dropped by approximately 1% since last year) because of its use in generating strong pseudo-random numbers. Its high usage is attributable to being used in common services such as Google Analytics.

Bot protection services

The internet today is filled with bots, and hence there is a constant rise in bad bot attacks. According to 2022 Bad Bot Report⁵⁶¹ by Imperva, 27.7% of all internet traffic was by bad bots. Bad bots are the ones which try to scrape data and exploit it. According to the report, the end of 2021 saw a surge in bad bot attacks probably because of the log4j vulnerability which is exploitable by bots.

⁵⁶¹. <https://www.imperva.com/resources/reports/2022-imperva-Bad-Bot-Report.pdf>

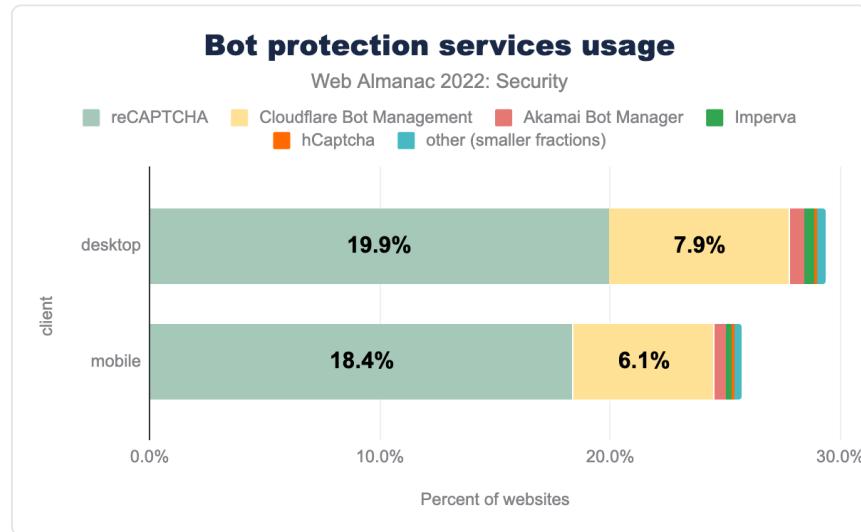


Figure 14.33. Usage of bot protection services by provider.

Our analysis shows that 29% of desktop websites and 26% of mobile websites use a mechanism to fight malicious bots which is a significant increase from last year (11% and 10% respectively). This increase could be because of Cloudflare Bot Management⁵⁶² which is a captcha-free solution for protection against bad bots. Last year's data crawl didn't track this, but identifying this was added this year and we see 6% of websites on mobile using it. Usage of reCaptcha has also increased from last year on both desktop and mobile by approximately 9%.

Drivers of security mechanism adoption

There are multiple driving factors for a website to adopt more security practices. The three primary ones are:

- Societal: more security-oriented education in certain countries, or laws that take more punitive measures in case of a data breach
- Technological: it might be easier to adopt security features in certain technology stacks, or certain vendors might enable security features by default
- Popularity: widely popular websites may face more targeted attacks than a website that is little known

562. <https://www.cloudflare.com/en-gb/products/bot-management/>

Location of website

Location of the website, where the website developers are based or where the website is hosted can often have impacts on adoption of security features. This can be because of the awareness among developers being different, but can also be because of the laws of the country mandating adoption of certain security practices.

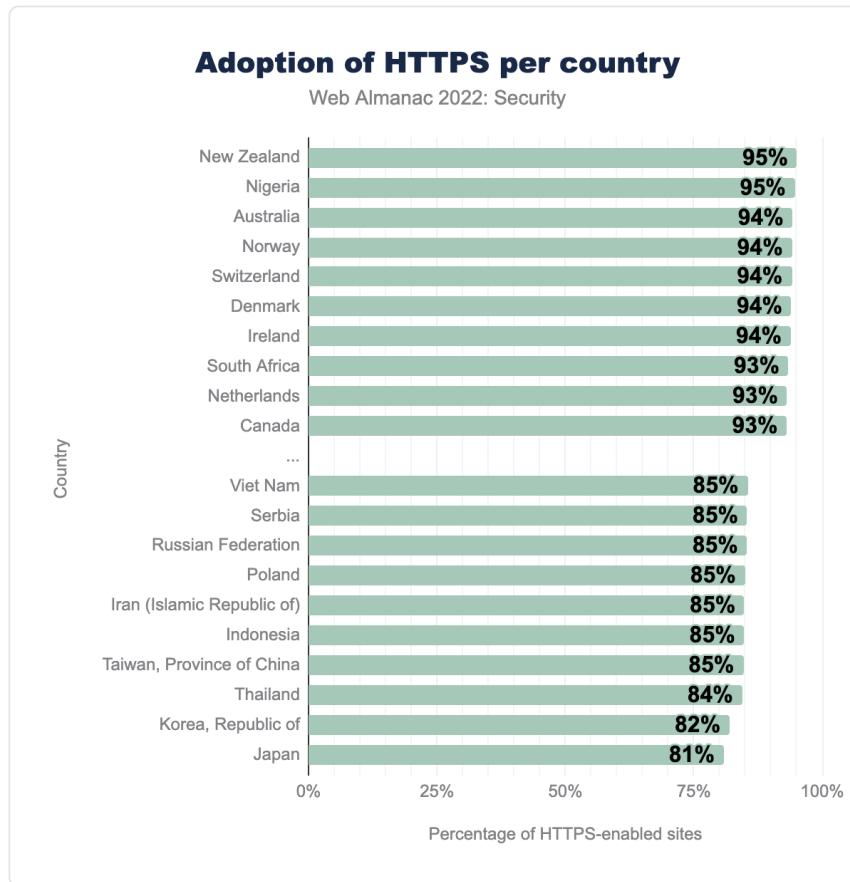


Figure 14.34. Adoption of HTTPS per country.

We see a lot of new countries like Nigeria, Norway and Denmark quickly rise to the top in the adoption of HTTPS. It's a good sign to see new countries also adopting widespread security practices because that can be an indication of rising awareness everywhere. Also, the difference between the least adoption and most adoption of HTTPS is reducing, which shows that almost all countries at least strive to have HTTPS by default on their websites.

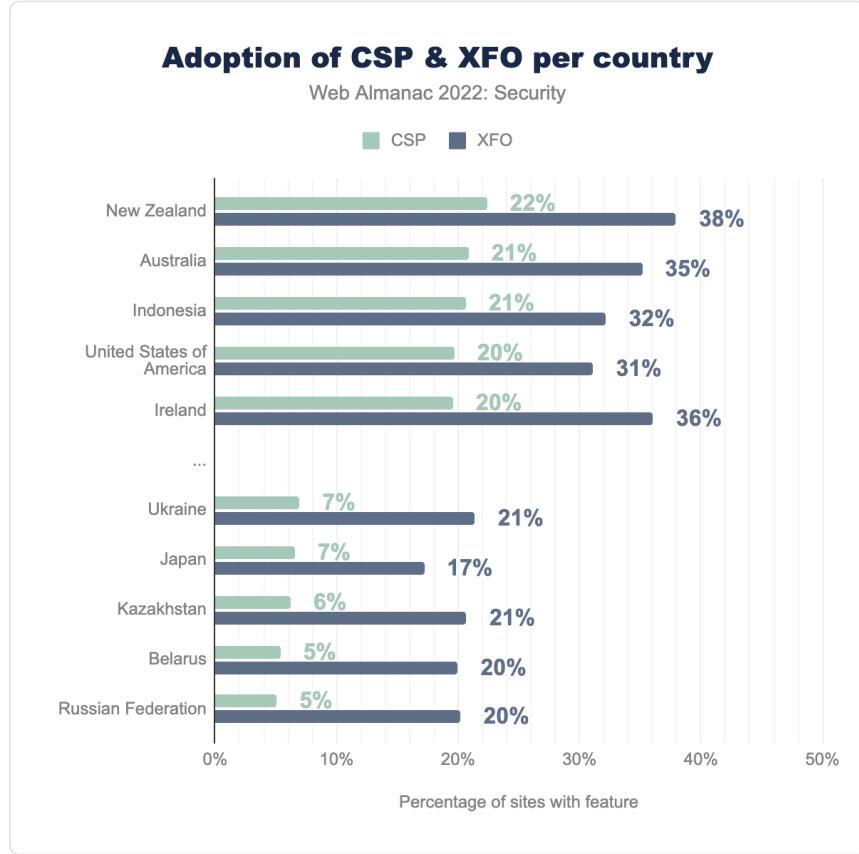


Figure 14.35. Adoption of CSP and XFO per country.

The adoption of CSP and [X-Frame-Options](#) (XFO) is very similar to last year. Surprisingly, we see websites in Indonesia have started adopting CSP a lot, even though their adoption of HTTPS continues to be low. The adoption of CSP still seems to be very varied across countries but the gap between adoption of XFO is gradually decreasing. More countries need to increase the adoption of CSP since it is a very important security feature that protects against a varied number of attacks.

Technology stack

Another factor that can strongly influence the adoption of certain security mechanisms is the technology stack that's being used to build a website. In some cases, security features may be enabled by default, for example, in content management systems.

Technology	Security features
Blogger	<i>Content-Security-Policy (99%), X-Content-Type-Options (99%), X-Frame-Options (99%), X-XSS-Protection (99%)</i>
Wix	<i>Strict-Transport-Security (99%), X-Content-Type-Options (99%)</i>
Drupal	<i>X-Content-Type-Options (77%), X-Frame-Options (77%)</i>
Squarespace	<i>Strict-Transport-Security (91%), X-Content-Type-Options (98%)</i>
Google Sites	<i>Content-Security-Policy (96%), Cross-Origin-Opener-Policy (96%), Referrer-Policy (96%), X-Content-Type-Options (97%), X-Frame-Options (97%), X-XSS-Protection (97%)</i>
Plone	<i>X-Frame-Options (60%)</i>
Wagtail	<i>X-Content-Type-Options (51%), X-Frame-Options (72%)</i>
Medium	<i>Content-Security-Policy (75%), Expect-CT (83%), Strict-Transport-Security (84%), X-Content-Type-Options (83%)</i>

Figure 14.36. Security features adoption by various technology.

Above we see some of the common CMS and blogging sites. A common pattern that we see with sites that provide very little customization and focus more on content editing, like Wix, Squarespace and Medium, is they have basic security features by default such as [Strict-Transport-Security](#). Content management systems like Wagtail, Plone and Drupal have very bare minimum security features, since they are often used by developers to set up the website and hence the responsibility to add security features are more on developers. We also see that websites using Google Sites have many security features by default.

Website popularity

Websites that have many visitors may be more prone to targeted attacks given that there are more users with potentially sensitive data to attract attackers. Therefore, it can be expected that widely visited websites invest more in security in order to safeguard their users.

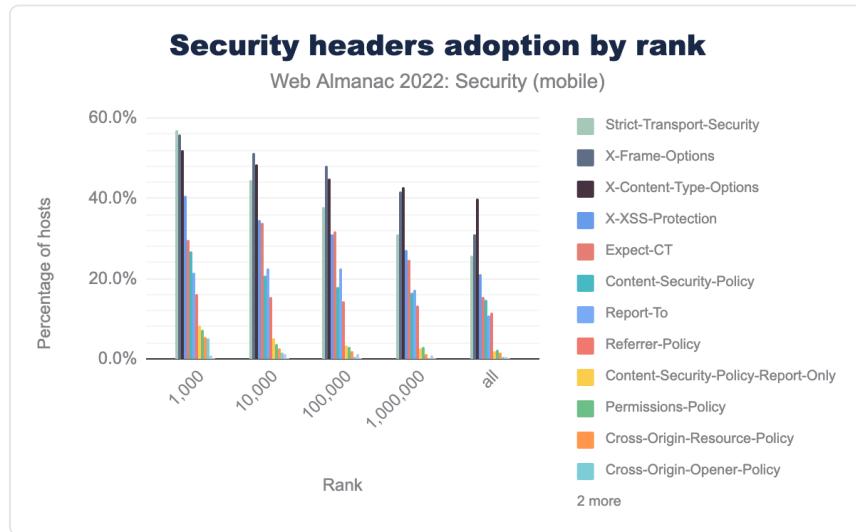


Figure 14.37. Prevalence of security headers set in a first-party context by rank.

We found that `Strict-Transport-Security`, `X-Frame-Options`, and `X-Content-Type-Options` always have more adoption for websites which are more popular. 56.8% of the top 1000 websites in mobile have `Strict-Transport-Security`, which means these websites care more about serving their content and data only via HTTPS. The less popular websites might have HTTPS enabled, but often seem to not add a `Strict-Transport-Security` header to ensure that their website is always served over HTTPS. The numbers this year are pretty consistent with last year's findings.

Malpractices on the web

Cryptocurrencies continued to grow in popularity this year with more types available for various use cases. With that continued growth and existing economic incentive, cybercriminals have consistently leveraged this to their advantage via cryptojacking⁵⁶³. However the use of crypto miners has overall trended downward since last year. What seems to occur is certain

⁵⁶³. <https://en.wikipedia.org/wiki/Cryptojacking>

vulnerability events that enable attackers to inject crypto miners into systems on both desktop and mobile triggers a spike in their usage:

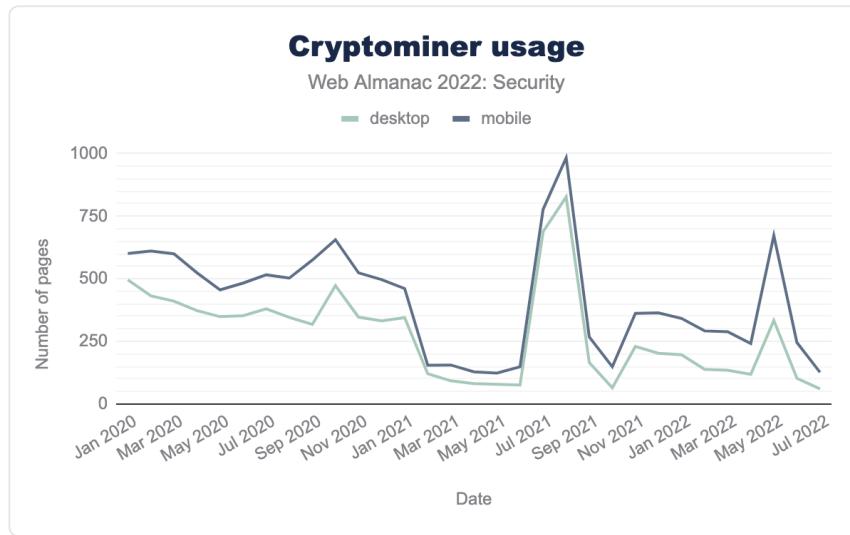


Figure 14.38. Cryptominer usage.

As an example, around July and August of 2021, there were reports of several cryptojacking campaigns and vulnerabilities^{1,2,3} which could be the cause for the spikes in cryptominers found in websites around that time. More recently, in April of 2022 hackers attempted to leverage the SpringShell vulnerability to set up and run crypto miners⁵⁶⁴.

Getting into the specifics of the cryptominers found in use among websites on both desktop and mobile we found that the share among miners has spread from last year. For example, Coinimp's share has shrunk since last year by about 24% while Minero.cc has grown by about 11%.

⁵⁶⁴. <https://arstechnica.com/information-technology/2022/04/hackers-hammer-springshell-vulnerability-in-attempt-to-install-cryptominers/>

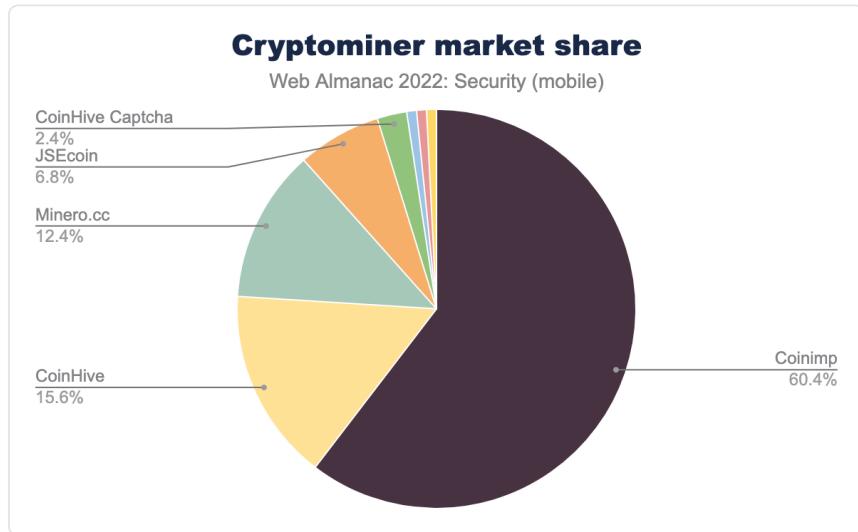


Figure 14.39. Cryptominer market share (mobile).

These results suggest that cryptojacking continues to be a serious attack vector each year with usage spikes based on newly emerged vulnerabilities that enable them. Therefore proper diligence is still required in order to mitigate risks in this space.

Note that not all of these websites are infected. Website operators may also deploy this technique (instead of showing ads) to finance their website. But the use of this technique is also heavily discussed technically, legally, and ethically.

Please also note that our results may not show the actual state of the websites infected with cryptojacking. Since we run our crawler once a month, not all websites that run a cryptominer can be discovered. This is the case, for example, if a website remains infected for only X days and not on the day our crawler ran.

Well-known URIs

Well-known URIs⁵⁶⁵ are used to designate specific locations to data or services related to the overall website. A well-known URI is a URI⁵⁶⁶ whose path component begins with the characters `/well-known/`

⁵⁶⁵. <https://datatracker.ietf.org/doc/html/rfc8615>

⁵⁶⁶. <https://datatracker.ietf.org/doc/html/rfc3986>

security.txt

`security.txt` is a file format for websites to provide a standard for vulnerability reporting. Website providers can provide contact details, PGP key, policy, and other information in this file. White hat hackers and penetration testers can use this information to conduct security analyses on these websites and report a vulnerability.

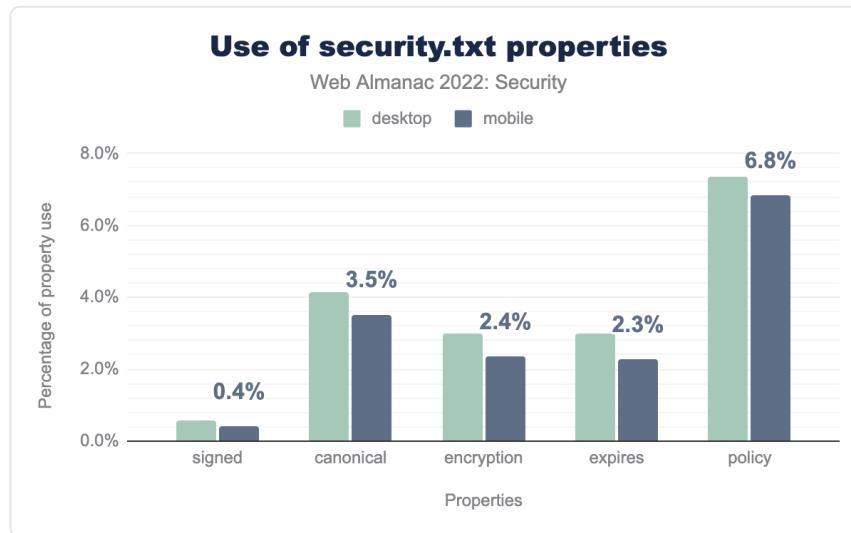


Figure 14.40. Use of security.txt properties.

The percentage of `security.txt` URIs with the `expires` property has increased from 0.7% to 2.3% this year. The `expires` property is a required property based on the standard, so it is good to see more websites adhering to the standard. `policy` continues to be the most popular property in a `security.txt` URI. `policy` is very essential in a `security.txt` URI since it describes the steps to be followed by a security researcher to report a vulnerability.

change-password

The `change-password` well-known URI is a specification under the Web Application Security Working Group of the W3C which is in editor's draft state right now. This specific well-known URI was suggested as a way for users and softwares to easily identify the link to be used for changing passwords.

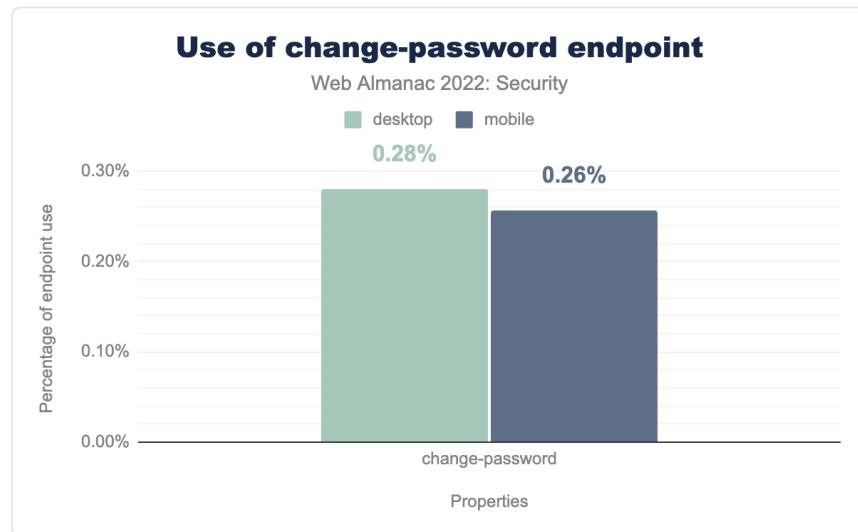


Figure 14.41. Use of change-password endpoint.

The adoption of this well-known URI is still pretty low. The specification is still work-in-progress so it's understandable that not many websites have started adopting it. Also, not all websites will have a change-password form, especially if they don't have a sign-in system for their website.

Detecting Status Code Reliability

This particular well-known URI determines the reliability of a website's HTTP response status code. This URI is also still in editor's draft⁵⁶⁷ state and may change in the future. The idea behind this well-known URI is that it should never exist in any website. So this well-known URI should never respond with an ok-status⁵⁶⁸. If it redirects and returns an "ok-status", that means the website's status codes are not reliable.

⁵⁶⁷ <https://w3c.github.io/webappsec-change-password-url/response-code-reliability.html>

⁵⁶⁸ <https://fetch.spec.whatwg.org/#ok-status>

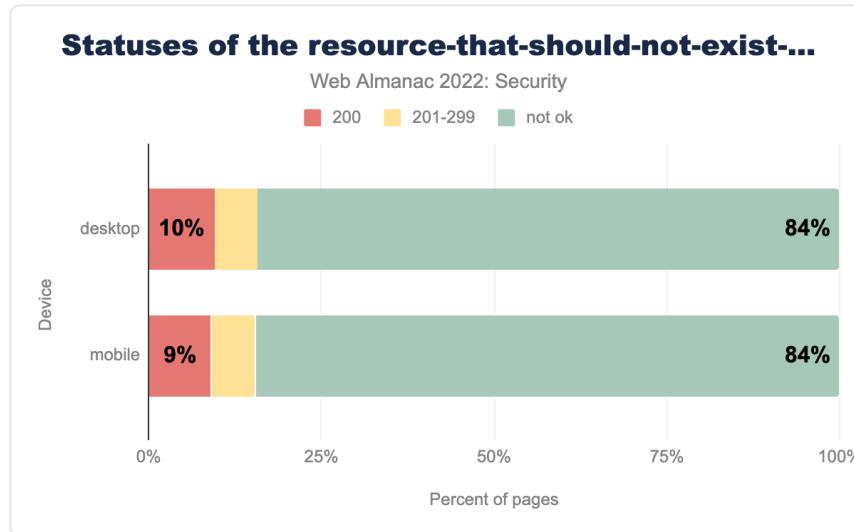


Figure 14.42. Statuses of the resource-that-should-not-exist-whose-status-code-should-not-be-200 endpoint.

We found that 84% of websites in both mobile and desktop respond with a not-ok status for this well-known URI. The good thing about this specification is if websites are correctly configured, this specification should automatically work and won't need website developers to make any specific changes.

Conclusion

Our analysis this year shows that websites are continuing to make improvements in their security features like we have seen over the past years. It's also exciting to see that many countries who were behind on web security adoptions are increasing their usage. This could mean that awareness around web security in general is increasing.

We found that web developers are also slowly adopting new standards and replacing the old ones. This is definitely a step in the right direction. The importance of security and privacy on the internet is growing everyday. The web keeps becoming an integral part of life for many people and hence, web developers should continue to increase the usage of web security features.

There's still a lot of progress that we need to do in setting stricter Content Security Policy.

Cross-site scripting continues to be in OWASP Top 10⁵⁶⁹. There needs to be wider adoption of stricter `script-src` directives to prevent such attacks. Also, more developers can look into taking advantage of Web Cryptography API. Similar efforts need to be made in adopting well-known URIs like `security.txt`. Not only does it provide a way for security professionals to report vulnerabilities in the website, but it also shows that the developers care about the website's security and are open to making improvements.

It's encouraging to observe the continuous progress in usage of web security over the past years, but the web community needs to continue researching and adopting more security features since the web continues to grow and security becomes more crucial.

Authors



Saptak Sengupta

@Saptak013 SaptakS <https://saptaks.website>

Saptak S is a human rights centered web developer, focusing on usability, security, privacy and accessibility topics in web development. He is a contributor and maintainer of various different open source projects like The A11Y Project⁵⁷⁰, OnionShare⁵⁷¹ and Wagtail⁵⁷². You can find him blogging at saptaks.blog⁵⁷³.



Liran Tal

@liran_tal lirantal https://twitter.com/liran_tal

Known for his open source and JavaScript security initiatives, Liran Tal⁵⁷⁴ is an award-winning software developer, security researcher, and open source champion in the JavaScript community. He's an internationally recognized GitHub Star⁵⁷⁵, acknowledged for his open source advocacy, and has received the OpenJS Foundation's Pathfinder for Security⁵⁷⁶ for his work on Node.js security. His contributions to developer security education include leading OWASP projects, building supply chain security tools, participation in CNCF and OpenSSF initiatives, and authoring books such as O'Reilly's Serverless Security. He leads the developer advocacy team at Snyk.io and is on a mission to empower developers with better application security skills.

569. <https://owasp.org/Top10/>

570. <https://www.a11yproject.com>

571. <https://onionshare.org/>

572. <https://wagtail.org/>

573. <https://saptaks.blog>

574. <https://www.lirantal.com>

575. <https://stars.github.com/profiles/lirantal/>

576. <https://openjsf.org/announcement/2022/06/07/first-ever-javascriptlandia-awards-celebrate-community-leaders/>



Brian Clark

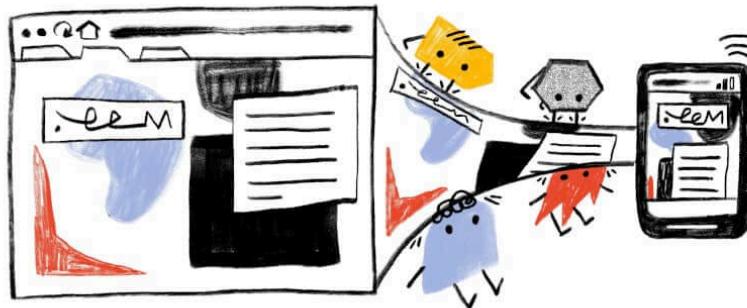
 @_clarkio  clarkio  <https://www.clarkio.com>

Brian is a web developer with in-depth experience in application security. He helps developers build secure web applications through his work as a Developer Advocate at Snyk.io. While he has experience working across full stack projects, his focus is on backend services, API's and developer tools. Brian loves to teach developers what he's learned from the successes and failures he's had throughout his career. You can find him doing just that on his weekly livestreams⁵⁷⁷ or in one of his Pluralsight courses⁵⁷⁸.

577. <https://clarkio.live>
578. <https://www.pluralsight.com/authors/brian-clark>

Part II Chapter 15

Mobile Web



Written by Cindy Krum

Reviewed by Dave Smart, David Fox, and Hemanth HM

Analyzed by Sia Karamalegos and Rick Viscomi

Edited by Rick Viscomi

Introduction

Mobile access to web content is a critical aspect of internet access as a whole. In fact, in many situations and regions, it is the default means of accessing the internet⁵⁷⁹. It is also often the backbone of communication⁵⁸⁰ that happens seamlessly on platforms that use desktop, native app, and web apps to facilitate cross-device behavior and allows consumers to use their preferred method of access, simplifying and further democratizing information and communication online.

This chapter will outline the state of the web in 2022 when it is accessed from a mobile device. In some cases, mobile data is compared to desktop data, which many people are more familiar with. This comparison is important because though many will focus on desktop data, there is now more mobile web traffic around the world than there is desktop, and this has been the case since about 2016 or 2017, depending on the source.

579. <https://www.gsma.com/mobileeconomy/wp-content/uploads/2022/02/280222-The-Mobile-Economy-2022.pdf>

580. <https://www.investopedia.com/is-having-a-smartphone-a-requirement-in-2021-5190186>

Worldwide connectivity

As seems to always be the case, we are living in a more connected world this year than the world has ever experienced. The evolution of mobile technology and mobile web has been fueled not only by more than two years of even more digital-focused commerce caused by the COVID-19 pandemic, but also the growth and evolution of 5G communication networks, cloud and hybrid-cloud computing environments, and the growing adoption of digital and voice assistants, “casting” technology, and IoT.

New generations are getting involved in social media and have earlier access to mobile technology, and it is more readily socially acceptable than ever before. So, the growth of connectivity marches on, with no end in sight, and children, teens, and young adults of today—colloquially referred to as *digital natives* because they were born into the digitally connected world—are sure to push the evolution of mobile technology, mobile web, and connectivity to new highs. All this progress makes the newest technology of today look foundational and fundamental for technologies of the future.

Traffic from mobile versus desktop

In keeping with the Methodology, the primary data source for this report is the HTTP Archive and the Chrome UX Report (CrUX). In cases where tablet data was included as a separate measurement from any data source, it was omitted, since it does not neatly fit in the primary mobile or desktop classifications and can add confusion and complexity when interpreting or contrasting mobile and desktop information that is more neatly segmented out. Refer to the CrUX documentation⁵⁸¹ for more information about eligible mobile platforms.

⁵⁸¹. <https://developer.chrome.com/docs/crux/methodology/#user-eligibility>

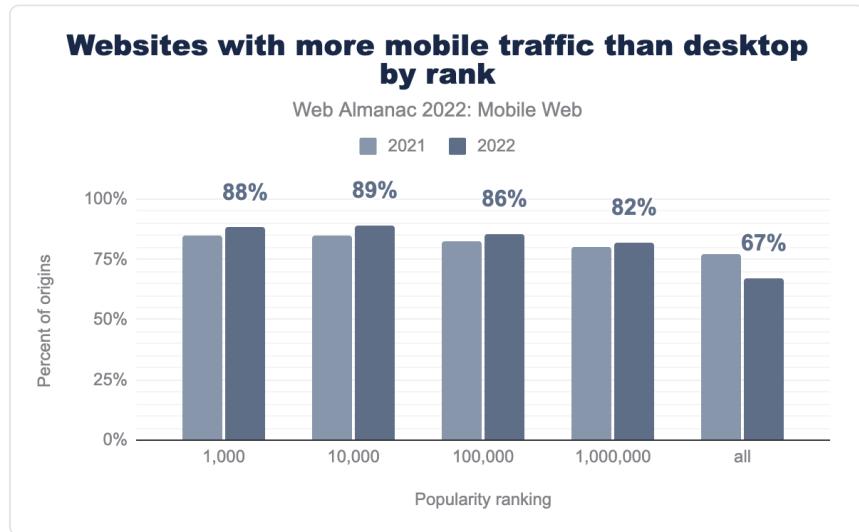


Figure 15.1. Annual comparison of the percent of websites that receive more mobile traffic than desktop, segmented by popularity ranking.

Across the most popular ranks, the percentage of sites with more mobile traffic than desktop traffic has increased relative to last year⁵⁸². In 2022, 88% of the top 1,000 most popular sites receive more traffic from mobile devices than desktop, up from about 85% in 2021. And among the top 10,000 most popular sites, 89% of them receive more traffic from mobile devices and that is up from about 86% in 2021 - so roughly a 3% increase in sites that receive more mobile traffic than desktop in both of those top-popularity ranking groups.

According to Oberlo, about 58%⁵⁸³ of web traffic in 2022 is from mobile devices. The consistent growth and pervasiveness of these statistics is a clear indication of the obvious increasing importance in the overall evaluation of mobile web access and interactions.

Communication from the mobile web

The value of the mobile web is largely in its ability to seamlessly connect people to other people and to information in an ongoing, portable, familiar, and unobtrusive way. Phones have always been primarily for communication, but the addition of the mobile web to the mobile phone fundamentally changed how phones are seen, used, and evaluated. This section of the report will go through how we can perceive the most important aspects of communication from the mobile web, and how potential for communication is reflected in mobile websites.

582. <https://almanac.httparchive.org/en/2021/mobile-web#traffic-use-by-popularity>

583. <https://www.oberlo.com/statistics/mobile-internet-traffic>

Alternative protocol links

Being that mobile devices are so critical in people's daily communication, it can be interesting to evaluate the most common types of link formatting that is present in the mobile web, something called *alternative protocol links*. Rather than linking one page on a website to another page on a website, these alternative protocols link to a type of activity other than web browsing.

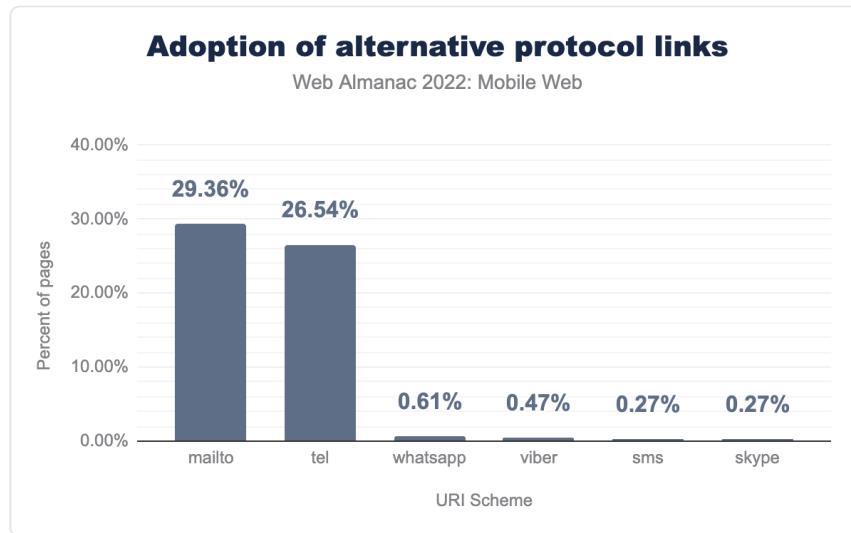


Figure 15.2. Adoption of alternative protocols used on mobile web pages.

The figure above outlines how the different alternative protocol links are being used on the web in 2022. The most popular options are: `mailto` for email, `tel` to make a call, `sms` to send a text message, and other application-specific protocols for different messaging services like WhatsApp, Skype, and Viber.

The `mailto` protocol is found on 29% of mobile pages. Close behind is the `tel` protocol, found on 27% of pages. That these protocols are found more on mobile pages is potentially indicative that websites are being modified with functionality to encourage a more multi-modal experience on mobile, like placing a call, which would be more complex to execute on desktop.

0.27%

Figure 15.3. Percent of mobile pages that use the `sms` link protocol.

Adoption of the `sms` protocol on mobile websites is much lower, coming in at 0.27%. This is interesting because linking someone to a pre-formatted SMS message can be an effective way to encourage people to sign up for an SMS loyalty program that can keep users much more engaged with a brand in the long-term, but it seems likely that designers, developers, or marketers do not consider this when they are building a site.

Messaging application links are used very rarely by comparison, at less than 1% adoption. This could potentially be explained by the growth of *deep linking* on the mobile web, whereby an app can be directly opened and navigated to a deep section within the app.

Input fields

Engagement and interactivity are critical for a good mobile web experience, but for years many developers have overlooked setting specific type declarations for mobile users, to ensure that the correct keyboard appears immediately, when the user finally decides to interact. The correct setting will return a number keyboard when a user is entering a phone number, and a keyboard that includes an `@` symbol when they are entering an email address. Other types of input fields include submit functions, searches, checkboxes, password fields, radio buttons, other buttons, or as hidden elements.

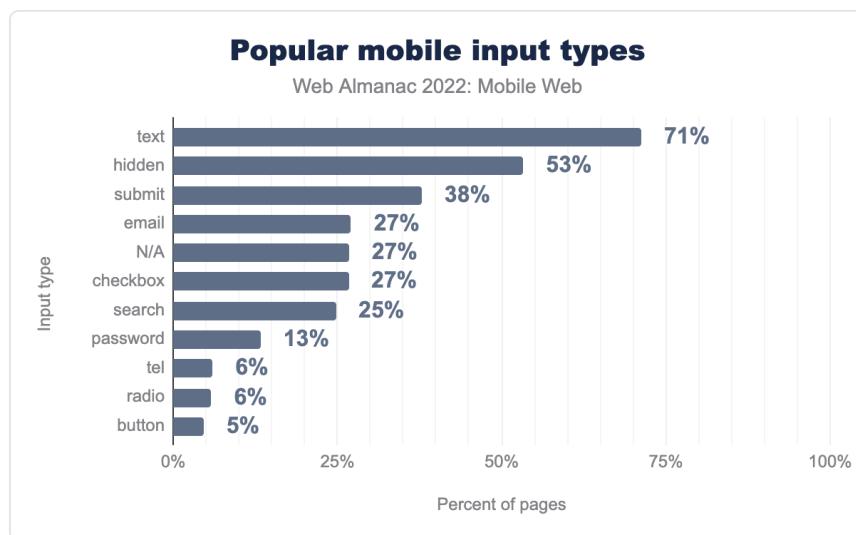


Figure 15.4. Adoption of input types.

Since modulating keyboards with input fields is much more important on mobile than it is on desktop, we track the most popular mobile input types across their archive of websites. In 2022, `text` was still the most popular input type, used on 71% of pages, which is down slightly

from last year⁵⁸⁴ at 73%. `hidden` was the next most common input type at 53% of pages, then `submit` at 38%.

After the most common input types, there is a cluster of input types that occur on around one quarter of the pages. These include `email`, no input fields (`N/A`) and `checkbox` at 27%, followed by `search` at 25%.

Compared to last year, `checkbox`, `email`, and `search` have increased a bit in prevalence. This possibly indicates more attention to the mobile use-case than in previous years, when more input fields may have been less indiscriminately lumped together as simply `text`.

Advanced input types

Advanced input types include `color`, `date`, `datetime-local`, `email`, `month`, `number`, `range`, `reset`, `search`, `tel`, `time`, `url`, `week`, `datalist`. They are considered *advanced* because they were introduced in the HTML 5 specification to alter the input method in the browser, generally changing the presentation of options that a user can interact with on the mobile site.



Figure 15.5. Percent of mobile pages that use advanced input types.

Of pages with at least one input, 47% of them use one or more advanced input types, which is up from 2021, when only 45% of mobile sites used an advanced input type.

In general, the use of specialized input fields and protocols can help make visitor interactions more engaging, useful, and efficient. It is unfortunate that it is still quite common for input fields and related functionality to be so often overlooked on the mobile web, but it is not entirely disheartening. These types of mobile controls have been around for many years, but they may be growing outdated or may now be executed by web app functionality and JavaScript, rather than by the traditional methods of coding that are expected in this type of analysis.

Similarly, with the evolution of deep links, which can open and directly navigate to a deep screen within a native app, specialized input fields might not be as necessary. In some cases, brands may try to push visitors from a website to an app, if they believe that it will provide users with a better experience or have better overall conversion statistics than the website. This is especially true when apps are the main focus of the company, or when the apps are being more

584. <https://almanac.httparchive.org/en/2021/mobile-web#type-declarations>

actively updated and maintained than the website. This is rarely technically necessary, since both websites and apps can basically complete all of the necessary tasks, so it is mostly about the brand's preference, or their own internal success metrics.

The variations on input fields are generally included to make web interaction more profitable for the company and more efficient for the user. They are often associated with digital checkout processes, and thus, can directly impact the bottom line. For some brands, the importance of the functionality may make it more likely that developers push users to a native app to complete a purchase, in order to streamline the processing, leverage saved user loyalty information, or speed up the completion of the transaction. This perception that native apps are better for this kind of interaction seems misguided though. Now, most consumers expect to have a smooth and seamless experience wherever they begin their encounter, rather than being forced to transition from web to app or vice versa. Whenever possible, parity between app and web functionality should be a top priority and advanced input methods can help with this, especially for things that impact the bottom line, like checkout processes.

It is possible that traditional input fields—whether they are link protocols, input types, or advanced input types—may just be handled by deep links to apps, and these are handled differently. The protocols for linking to apps that may be replacing some web functionality is broadly called deep linking, or *Universal Links* on iOS and *Android App Links* on Android. In code, these links look just like a regular web link, and the launching of the app is handled by a web app manifest file hosted in the `/well-known` directory at the root of the website. That said, it is hard to make assumptions about what changes and variations in these numbers mean in a practical interpretation, because we don't know if losses and gains are absolute, or if there is just a transitioning of technology and norms in these aspects of the mobile web.

Looking at this topic in another light, it may also be too high of an expectation that developers specifically code the input types and variations for everything on their site. It may be possible for mobile browsers to do more of the heavy lifting in determining what the right action or keyboard layout is for a particular link or input field, based on obvious clues from the code or previous user interactions. Browsers may even be able to leverage this very research to optimize experiences like that.

For example, if every user that clicks on a particular input field switches to the number keyboard or submits only numbers, maybe the browser could find a way to use that kind of interaction and metadata to heuristically enable the numeric keyboard by default. We hope that browsers will continue the trend of simplifying tasks that have historically been complex and inefficient for site owners to code, relieving developers of the burden. When browsers assume more responsibility for aspects of interactivity, it allows them to live up to their namesake role as the *user agent*, rather than pushing all the responsibility for good user experiences on the site owners. These heuristic defaults for web-only sites, along with deep links to apps when their enhanced potential for functionality are available, seem like the ideal

path forward.

Accessibility on the mobile web

Mobile devices are cheaper, lighter, and more portable than computers, so they house a lot of potential to help populations that have historically been ignored or marginalized by technology, often with only minor tweaks to a site's accessibility. According to Google⁵⁸⁵, true accessibility on the web means that “the site's content is available, and its functionality can be operated by literally anyone.” In a more detailed explanation, Google offers that:

Accessibility, then, refers to the experience of users who might be outside the narrow range of the “typical” user, who might access or interact with things differently than you expect. Specifically, it concerns users who are experiencing some type of impairment or disability - and bear in mind that that experience might be non-physical or temporary.

Web accessibility and mobile web accessibility are evolving in their importance and prevalence in the conversations and general consciousness of the web as a whole. Building more accessible sites will allow the mobile web to reach more users⁵⁸⁶ and potential customers for businesses online. It will also help position information and associated brands as inclusive and in-touch with the needs of *all* users rather than perpetuating insensitivity and further marginalization by only considering the *average user*.

Accessibility concerns can be broken down into three types: situational, temporary, and permanent. In most cases, it is possible to imagine circumstances in which an aspect of web accessibility is critical for someone with a permanent impairment while also being incredibly useful for people with situational or temporary impairments. And while it is not explicitly rewarding aspects of accessibility with rankings, meeting web accessibility standards often has the side benefit of improving organic rankings⁵⁸⁷ in Google search results.

The W3C⁵⁸⁸ summarizes the basic tenets of accessibility in four groups of concerns, which make the website: *perceivable* (something you can see or hear), *operable* (buttons and gestures you can use), *understandable* (layout and presentation concerns), and *robust* (ability to enter and submit forms and information). The elements that we focus on here fall mostly under the *perceivable* and *operable* groups, including: color contrast, tap targets, and zooming and scaling.

It is possible that the increased focus on mobile accessibility will reframe the function and importance of mobile and digital communication, especially if it helps us to reassert the

585. <https://web.dev/accessibility/#what-is-accessibility>

586. <https://moz.com/blog/seo-and-accessibility-introduction>

587. <https://searchengineering.com/seo-accessibility-tips-deaf-disabled-386577>

588. <https://www.w3.org/TR/mobile-accessibility-mapping/>

humanity of digital technology and communication. This is particularly important in cultures where the increased digital connectivity may actually be perpetuating isolation and a lack of real-world connectivity. Making the mobile web more accessible should have a direct and indirect positive impact on mobile technology and society at large.

Color contrast

Color contrast is an important aspect of mobile web accessibility for a large variety of reasons, and it is one of the easiest accessibility needs to spot and update when reviewing a mobile site. On mobile devices, where screens are always small and fonts may be smaller than normal, having good color contrast can make a big difference in the legibility of the content. When any kind of vision impairment is present, including common afflictions like color blindness, glaucoma, or cataracts, viewing web content on a mobile phone can be difficult or impossible.

Even people with perfect vision can sometimes struggle to consume mobile web content on a screen when the screen is dirty, has a lot of glare, or the user is in bright sunlight. A high contrast ratio between the colors of a mobile site can help make it easier to use and appreciate, even in bad conditions and even for people with perfect vision.

That said, many people don't have perfect vision and use glasses or contact lenses to help them see. People with perfect vision can expect their vision to eventually degrade as they age, at least to some degree, so this accessibility standard is one that we can expect to impact everyone, albeit eventually.

23%

Figure 15.6. Percent of mobile pages with sufficient color contrast.

Given the stakes, it is sad that only 23% of mobile sites this year actually have adequate color contrast. This is only one full percentage point up from the 2021 statistic (22%), so it is an improvement, but not a substantial improvement. When we look back further, this statistic has actually been 22% since 2019, so there has been almost no improvement in 4 years - Disappointing for something that is widely considered the most impactful accessibility standard on the web.

According to the US General Services Administration⁵⁸⁹, site owners should "make sure the contrast between the text and background is greater than or equal to 4.5:1 for small text and

⁵⁸⁹. <https://accessibility.digital.gov/visual-design/color-and-contrast/>

3:1 for large text.” The W3C⁵⁹⁰ backs this ratio and also adds an enhanced guideline that calls for “a contrast of at least 7:1 (or 4.5:1 for large-scale text).” Obviously there are more mobile sites that miss the mark for *enhanced accessibility*, making that statistic likely far below the 23% that meet the basic requirements for color contrast on mobile sites.

If enhancing the perceptibility of a site by improving the color contrast and making it actually usable is not enough of an incentive, having a minimum level of contrast between text and background colors has long been an important element for ranking in Google searches as well. It started as a means of preventing spammers from including hidden text on websites, which was often white text on a white background. Later, Google moved it over to be part of the original Mobile Friendly⁵⁹¹ guidelines, and now is simply included as part of their general guidelines for designing websites that are good for users.

Color contrast is something that we would like to encourage webmasters to focus on, especially in light of new color capabilities in browsers, like `dark-mode`. If you are making site updates to improve your sites compatibility in `dark-mode`, you should ideally also take the time to improve overall color contrast as part of that project. This feature is often easier for some people to use, so you can count this combined effort as a double-benefit!

Tap targets

Tap targets are effectively the clickable elements on a page. Since many designers and developers still think about websites as being designed on and for desktop interactions rather than *mobile-first*, it can be common to overlook having large enough spaces for touchscreen interactions, on which navigation and clicking with a finger is so much less precise. On mobile devices, when things are resized and rearranged to fit the smaller screen, it is common for linked elements to be close together. This can become a problem if multiple clickable elements are clustered together, as they might appear in a navigation menu or footer.

Having appropriately sized tap targets makes it less likely that a user will errantly click on the wrong button or link, and have to navigate back to try the click again. It is also not ideal to expect users to zoom in on the mobile screens simply to click the right button. From an accessibility standpoint, having appropriately sized tap targets is also important for users that have any kind of motor impairment⁵⁹², or difficulty with fine motor skills.

590. <https://www.w3.org/TR/mobile-accessibility-mapping/#h-contrast>
591. <https://developers.google.com/search/mobile-sites/get-started>
592. <https://www.w3.org/WAI/WCAG21/Understanding/target-size.html#benefits>

42%

Figure 15.7. Percent of mobile pages with sufficient tap targets.

The minimum size for a tap target is generally considered to be no smaller than 48 pixels by 48 pixels, which is a rough estimation of the size of a finger being used on a touchscreen. Tap targets are also expected to be a minimum of 8 pixels apart from each other in order to pass any of Google's evaluations. In our research, 42% of mobile sites had sufficient tap targets, which is disappointing as less than half of the sites manage to universally have appropriately sized tap targets.

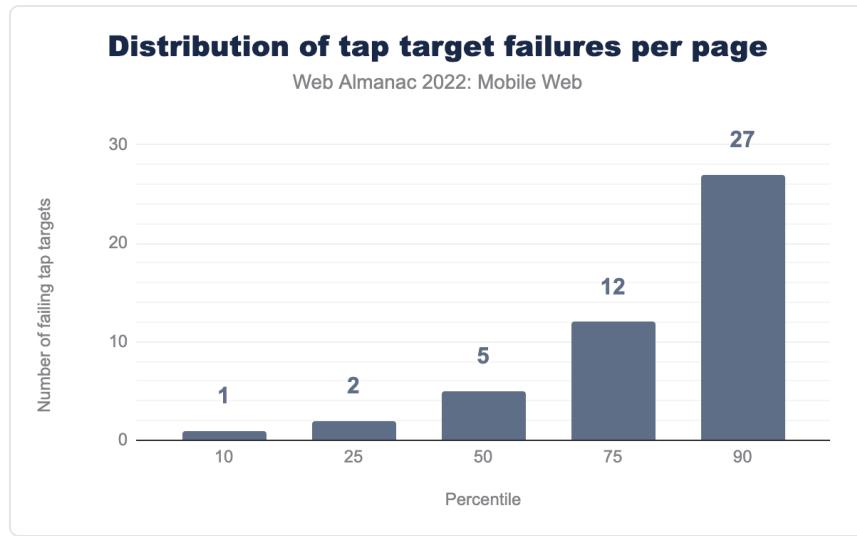


Figure 15.8. Distribution of the number of tap target failures per page.

Pages that fail the audit generally have more than one failing link. The median number of failing tap targets is five, but in some cases, when sites fail, the number of failing tap targets can be quite high. We see that the worst 10 percent of sites have at least 27 failing tap targets.

Zooming and scaling

Mobile devices have become a big part of daily life for most people, and the expectation is that interaction with mobile web content should be quick and easy. How websites handle zooming and scaling can go a long way to improving interactions on mobile. There are different takes on

this, and while most will agree that you need to set a proper initial scale in the viewport for mobile users (`<meta name="viewport" content="width=device-width, initial-scale=1">`), there is not universal agreement about the second part of a viewport setting, which controls if you should or shouldn't disable scaling and zooming (`user-scalable=no` or perhaps `user-scalable=yes`). Most authorities, including the W3C⁵⁹³, suggest that restricting scaling and zooming can create a bad user experience and adversely impact accessibility, so it should be avoided. Settings for `minimum-scale` and `maximum-scale` can also be set, and these are often safer limits, if a developer believes that limits are needed.

Zooming can be a good workaround for a user who is visually impaired, or anyone who just doesn't have their reading glasses handy when they need them. On the other hand, it can be hard to build sites that universally scale well on mobile. There are many different font size settings that would need to be accommodated, and getting it wrong can make the site much harder to interact with. This is why some designers prefer to prevent scaling and zooming, to ensure that the page renders in a highly predictable way that is not impacted by scaling and zooming. While this is true, disabling zooming and scaling impedes the usability of a mobile site, and thus, should be avoided for the sake of accessibility.

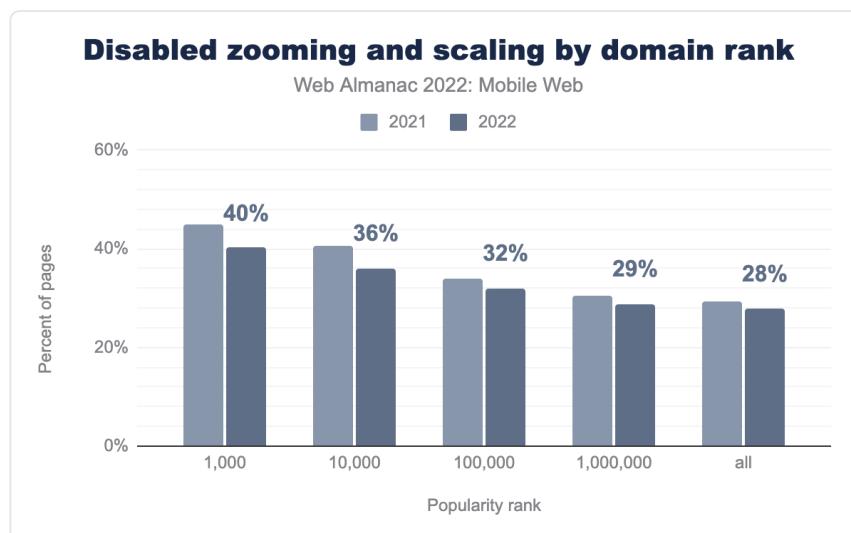


Figure 15.9. Annual comparison of the percent of websites that disable zooming and scaling, segmented by popularity ranking.

Of the mobile websites in the top 1,000 rank, 40% of them had disabled zooming and scaling.

^{593.} <https://www.w3.org/WAI/standards-guidelines/act/rules/b4f0c3/proposed/>

down from about 45% in 2021. When you look at the top 10,000, 36% of mobile sites had disabled scaling and zooming, and this is down from about 41% in 2021. Looking at the other end of the scale, the widest ranking group, which includes all the sites in the data set, 28% of sites prevented zooming and scaling, down from about 29% in 2021. Overall, what we can see here is that the prevalence of this accessibility-limiting setting is on the decline, especially for the most popular sites, which is good news, but the fact that more than half of the sites are still using this type of limiting setting is disappointing.

As a whole, we know that accessibility concerns are not going to go away, and as time progresses, meeting accessibility standards will become a basic expectation—especially on mobile devices. As is often the case, the use cases for mobile interactivity are more broad-ranging compared to desktop, so users' expectations are higher, even though the development constraints for web mobile content make it more difficult to actually achieve. Nevertheless, accessibility is becoming a critical component of good web design, and should be embraced to create a more inclusive mobile web. It is good to see that there is increasing adherence to basic accessibility guidelines, but there is still considerable room for improvement.

For many years, Google has created a positive impact on the web by rewarding websites that meet certain basic requirements with better rankings. They have done it for load time, performance, security and mobile-friendliness, but not yet for accessibility. Google does write and support a lot of advancements for web accessibility in their official communications, but there is an opportunity to do more. While many accessibility updates do naturally have a positive impact on website rankings, it may be time for Google to explicitly incentivize some level of compliance to basic accessibility standards with better rankings—not just because they can enhance the semantic understanding of a website, but also because they simply make the web a better place for everyone.

Mobile performance

One of the most complex problems that site owners have to address on the mobile web is performance, which is experienced as the delay that users incur when they request or interact with websites on smaller devices. Though they have evolved quite a lot, mobile devices still generally have less-powerful processors than larger devices and are often getting data over slower and less reliable internet connections. In addition, the probability that users may incur a cost for mobile data requests, especially if they go over a plan limit, makes efficiency important as a practical concern, beyond just its impact on user experience and speed.

Core Web Vitals

Core Web Vitals⁵⁹⁴ is a collection of performance metrics that Google compiles to evaluate different websites, and specifically, different page groups⁵⁹⁵ on websites to describe how they perform in both mobile and desktop page settings. The elements of Core Web Vitals include loading, interactivity, and layout stability.

All three are aspects of how users perceive the performance of a page that can help or hinder the loading experience for users. This type of evaluation began in May of 2020, and these metrics are taken into account in Google's ranking algorithm specifically as an aspect of the page experience⁵⁹⁶ evaluation, and thus, the metrics are organized around thresholds of performance that are either considered "good", "needs improvement", or "poor". For a site to be considered "good", 75% of visits must meet the prescribed thresholds for each of the Core Web Vitals metrics.

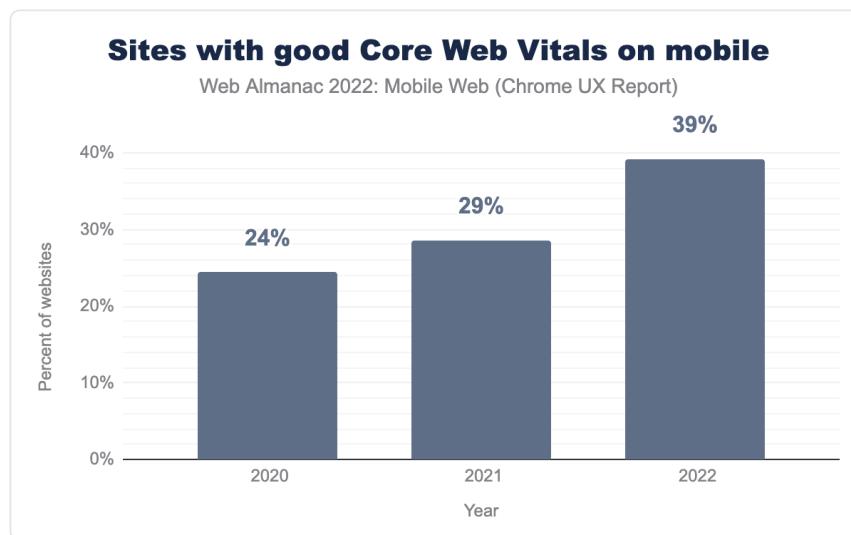


Figure 15.10. Annual comparison of the percent of websites having good Core Web Vitals on mobile.

The figure above shows how the overall performance of the web has changed since Core Web Vitals first launched in 2020. You can see that overall, mobile websites are consistently improving year over year. In 2022, 39% of websites have good Core Web Vitals experiences on mobile devices. See the Performance chapter for a deeper look at what may have caused such a significant change this year.

594. <https://web.dev/vitals/>

595. https://support.google.com/webmasters/answer/9205520#page_groups

596. <https://developers.google.com/search/docs/advanced/experience/page-experience>

Loading performance metrics

Load time on the mobile web is measured in the same way that it is measured in any other context, and despite the more difficult burdens that mobile devices face, that makes it harder for them to load a site quickly. Research⁵⁹⁷ shows that mobile visitors are more impatient, and actually expect and want the experience on a phone to be faster than it is on a larger device.

Time to First Byte (TTFB)

Time to First Byte⁵⁹⁸, which is often abbreviated as TTFB, is the measurement of the amount of time that elapses from the start of the navigation to the first byte of data received in response to the request. TTFB describes the responsiveness of the server and other network resources that are needed to begin building a page.

TTFB is not a Core Web Vitals metric itself, but it has a direct impact on all loading performance metrics, and thus, is often discussed as part of the optimization of all Core Web Vitals elements and general site performance, especially Largest Contentful Paint.

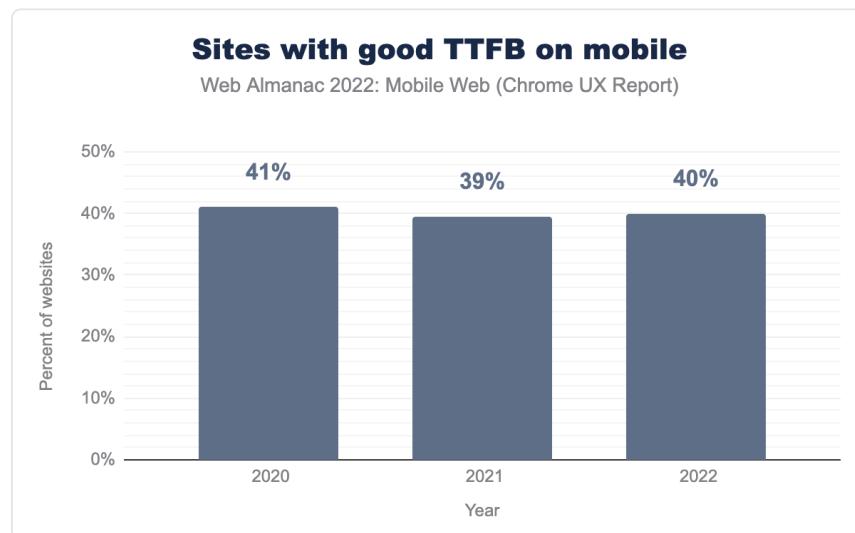


Figure 15.11. Annual comparison of the percent of websites having good TTFB on mobile.

As you can see above, there are only minor fluctuations in the percent of mobile sites that are considered “good” from 2020 to 2022, going from 41% in 2020 down to 39% in 2021, then back up to 40% in 2022.

597. <https://unbounce.com/page-speed-report/#~-text=Young%20folks%20have,an%20a%20cellphone.>

598. <https://web.dev/ttfb/>

Largest Contentful Paint (LCP)

Largest Contentful Paint⁵⁹⁹, or LCP, is a metric that describes how long it takes for a website to load the largest portion of the page content that is first displayed to a user with meaningful content; it is often a function of the size and performance of a header image or design. The LCP is important because it signals to a user when the page is ready to start consuming.

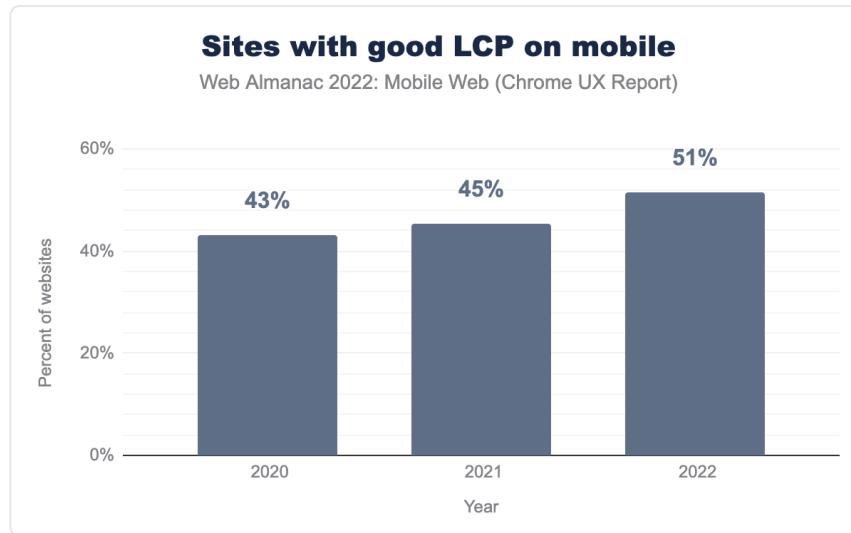


Figure 15.12. Annual comparison of the percent of websites having good LCP on mobile.

LCP performance is improving. In 2020, 43% of mobile sites had a LCP assessed as “good”. In 2021, this number improved to 45% of mobile sites. There was a significant jump in 2022 in which 51% of mobile sites had good LCP performance. The Performance chapter explores some possible explanations for why this may have happened.

Images

Images can contribute a lot to providing a good mobile experience, but they can also contribute a lot to slow performance and bad loading experiences if they are not set up correctly. This section explores how site owners are handling—or not handling—the performance impact of images.

^{599.} <https://web.dev/lcp/>

Appropriately sized images

Using images that are sized properly for a mobile device has long been one of the easiest ways that anyone could improve mobile load time. In the early days of the mobile web, site owners would often simply send the same images to desktop users as they would for mobile users, because ultimately, mobile browsers would scale and resize the images to fit in the mobile rendering of the page. Unfortunately, this didn't work well, because it ended up requiring a lot of extra data to send rich, high-quality images that were better suited for a desktop viewing experience.

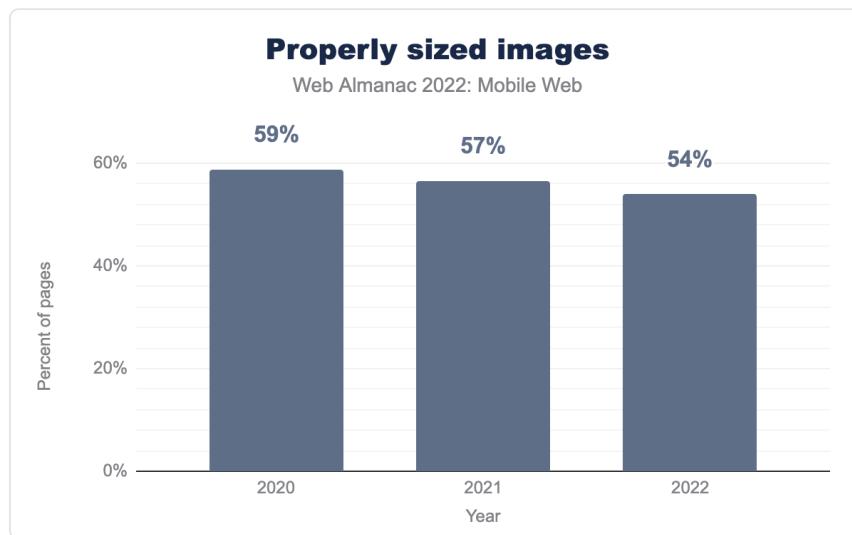


Figure 15.13. Annual comparison of the percent of websites that have appropriately sized images.

Given Google's increased focus on rewarding good performance with their Core Web Vitals program, you would expect that more sites would be optimizing their images. However, it's interesting to see that sites are actually having fewer optimized images over time. The figure above shows that there is a decrease in the percentage of pages with properly sized images since 2020⁶⁰⁰, when 59% of sites had properly sized images. But in 2022, that number is down to only 54%.

Responsive images

Creating images that can be responsive to different screen sizes is a common way to handle mobile image sizing. Using responsive images is a great way for websites to handle even the

600. <https://almanac.httparchive.org/en/2020/mobile-web#images>

most unique presentation scenarios, like viewing a website on a wide-screen TV, viewing a website on a connected digital assistant, or even on a small feature phone or handheld gaming system.

There are two main methods for embedding images on a screen: the `img` element, and the more-expanded `picture` element. The `picture` element offers a few more possibilities to include images based on certain criteria. The `srcset` attribute is available on both elements and enables images to be conditionally included based on things like screen size and display density. Some browsers may also take bandwidth into account when choosing an appropriate image.

The `picture` element further expands on these capabilities to allow for art direction⁶⁰¹, for example specifying a 4:3 ratio image for mobile portrait screens, and a 16:9 for desktop or landscape views. A further use is being able to specify different image formats, for example the browser can load an AVIF image where supported, otherwise it will fall back to a WebP or PNG image. Allowing the browser to make the sensible choice based on the conditions it's operating in usually means better performance and thus a better user experience.

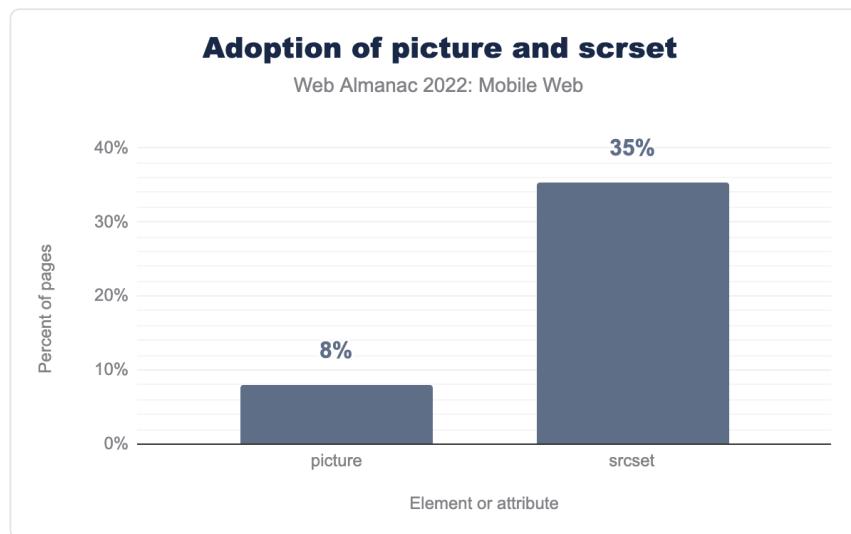


Figure 15.14. Adoption of the `picture` element and `srcset` attribute.

The adoption of the `picture` element is at only 8% of mobile pages (up from 6% last year), but `srcset` is at 35%. In 2021, the use of `srcset` was around 32%, so we have seen a growth of about 3 percentage points.

⁶⁰¹. https://developer.mozilla.org/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images#art_direction

Overall, it is great to see an increase in the use of responsive image techniques. The relatively small uptake for `picture`, despite offering more flexibility in art direction and format fallback support, could be down to the fact that it can require more work to produce, and is less likely to be supported by default in popular CMS systems, like WordPress, which stick to the venerable `img` tag (for now⁶⁰²).

Lazy-loading

Lazy-loading is the process of assigning different loading priority levels to elements of a web page based on where they occur on a page. Without lazy loading, all of the elements and images on a page will eventually be loaded, but lazy loading allows images to be deferred until it is clear that they will be needed, based on where the user has scrolled to on the page. Lazy-loading is especially relevant on mobile devices, because common responsive design patterns will almost always stack content for a mobile rendering. The narrow nature of mobile screens ensures that many stacked elements of the page are pushed far down below the fold, and may not be immediately necessary—especially if all the user wants to do is click a link in the top navigation. Lazy-loading eliminates that unnecessary data transfer, and the load time associated with it.



Figure 15.15. Percent of mobile pages that contained images using `loading="lazy"`.

Native lazy-loading⁶⁰³ has been available since 2019, which allows browsers to do the complex calculations in the most efficient way possible, and only requires that site owners tag images with either `lazy` or `eager`. This simple tagging can be a great boon for page and site performance, and can also save a lot of time and effort associated with maintaining your own lazy-loading code. As long as you don't inadvertently lazy-load your LCP image⁶⁰⁴ at the top of the page, it is an easy win, but we found that only 25% of sites are currently using the `loading=lazy` attribute for their images.

Layout stability

Layout stability is an important part of performance that has recently been pushed to the fore with Google's introduction of Core Web Vitals. Elements of Core Web Vitals are designed to measure and assess the loading experience of a page, and layout stability of a page is an important part of that. If a page is constantly moving and re-painting while it is being loaded,

602. <https://github.com/WordPress/performance/issues/21>

603. <https://web.dev/browser-level-image-lazy-loading/>

604. <https://web.dev/lcp-lazy-loading/>

this makes it seem like the page is taking much longer to load than it otherwise could if the experience were more predictable and once things load, they remain exactly in the place where they have loaded.

Since the loading order of content can be different under different conditions, or in different browsers, building and planning around layout stability is a good way to ensure a smooth loading experience regardless of the circumstances. The most relevant metrics for evaluating layout stability is Cumulative Layout Shift (CLS) but other aspects of performance like image sizing and lazy loading can also impact layout stability.

Cumulative Layout Shift (CLS)

Cumulative Layout Shift⁶⁰⁵, which is often abbreviated as CLS, is a representation of the stability of a page while it is in use.

A low CLS represents a visually stable layout, which makes the experience less frustrating to users. Pages with a high CLS often experience movement when images begin to load and text must be rendered to fit, or wrap around the image. This can also happen when font files load and the page has to be painted to accommodate differences caused by the font, sometimes described as reflow, or when a large header image loads and moves all the content on the page—occurrences that had been colloquially described as *jank*.

The screen size of the device requesting the page can have a significant impact on the way elements are laid out, and how much they move when shift-causing elements are loaded.

CLS is measured as a score, and the highest instance of movement in any session window during the page lifespan is what is measured. This changed⁶⁰⁶ in 2021, when CLS was previously measured as the sum of all individual shift scores on a page. Google considers scores of 0.1 or less as “good” and scores over 0.25 to be “poor”.

605. <https://web.dev/cls/>
606. <https://web.dev/cls-web-tooling/>

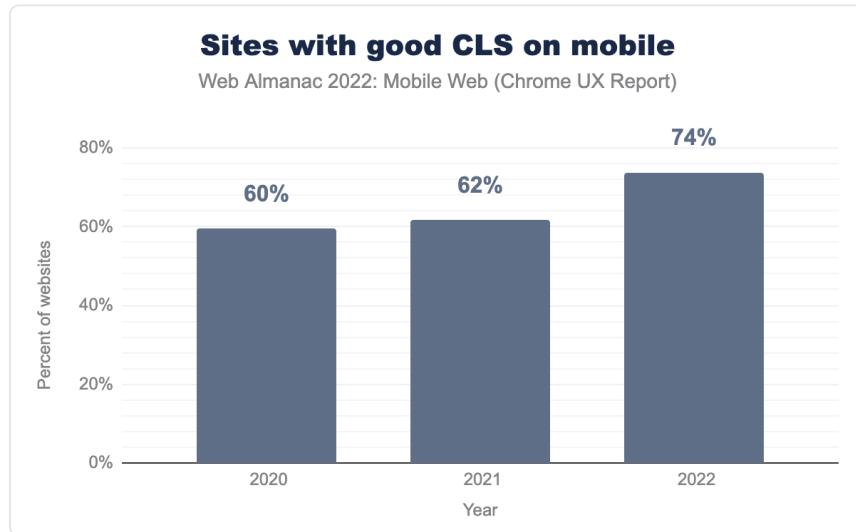


Figure 15.16. Annual comparison of the percent of websites having good CLS on mobile.

The percentage of websites with “good” CLS on mobile has improved significantly to 74%, up from 62% last year.

Responsiveness

Responsiveness is always good in a mobile scenario, but it can have a layered meaning. In general, technology that is responsive is good, and is reacting to cues that it is given in an efficient and meaningful way. When we talk about mobile responsiveness in terms of design, we are describing content that will respond and adjust its layout to accommodate the different screen sizes of the devices that request it. In a more broad sense though, responsive also indicates how quickly and efficiently a page responds to user interactions - so this type of responsiveness is less about layout, and more about quick interaction. Having a site that is very responsive is good because it creates an efficient user interaction, in which users feel immediately acknowledged when they interact with the site. The metrics used to evaluate this kind of responsiveness are First Input Delay (FID) and Interaction to Next Paint (INP), and those will be covered in this section.

First Input Delay (FID)

First Input Delay⁶⁰⁷ (FID) describes the responsiveness of a site, especially related to how long it

607. <https://web.dev/fid/>

takes for a site to respond after a user first clicks on a page element. A low FID is desirable, especially on mobile web interactions, where the responsiveness of a mobile site should ideally rival the responsiveness of a comparable native app, to make the interactions feel equally fluid and satisfying. Google considers a site to have “good” FID if at least 75% of experiences are under 100 ms.

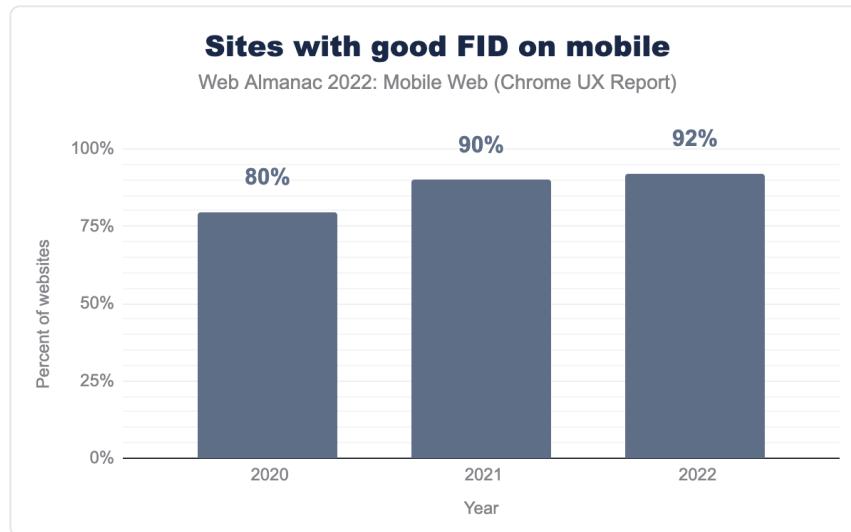


Figure 15.17. Annual comparison of the percent of websites having good FID on mobile.

There has been a consistent growth in the percent of mobile sites that have “good” FID in the past three years, going from only 80% in 2020 to 90% in 2021, and reaching 92% in 2022.

Interaction to Next Paint (INP)

Interaction to Next Paint⁶⁰⁸ (INP) is an experimental metric from Google that is used to measure responsiveness and response time on a page when a user interacts with it. A low INP is desirable because it means that the page was able to respond quickly to user interactions without substantial delays waiting for content to paint after it is requested. A “good” INP is 200 ms or less, and a poor one is anything over 500 ms. Eventually, INP could be added to the official Core Web Vitals metrics, but for now it is still being tested to make sure that it is a reliable and consistent metric that site owners will find useful.

^{608.} <https://web.dev/inp/>



Figure 15.18. Percent of websites that have good INP on mobile.

This is the first year that INP data is available to us, so we don't have any historical context, but what we see is that 55% of websites have good INP on mobile. This is especially interesting because of how much worse the mobile web performs on INP compared to FID. If INP does end up replacing FID as a Core Web Vital, responsiveness will become a much more prevalent issue.

Conclusion

In a comparison of data from 2020 and 2021 to 2022, there has been a lot of evolution in both the use and expectations of the mobile web. Aspects of performance like layout stability used to be considered optional, overly technical, or niche, have now become mainstream concerns for site owners that focus on mobile and desktop alike. The added complexity of providing a good user experience has come into higher relief for mobile devices, where it is confoundingly more expected but also much harder.

The good news for those who care about the mobile web and its user experiences is that much of the hard work of building sites that work well across all different devices is being offloaded from individual site and CMS development teams to browsers. We see this happening with lazy-loading, the `srcset` element for responsive images, and some other aspects of performance, but we hope that this type of initiative will also be taken on by the browsers to help simplify work in other ways as well. There are still many opportunities to improve mobile web accessibility, interactivity across different types of phone functionality, and eventually maybe even interplay with connected devices for things like casting, sharing, and other connected home and IoT scenarios. Google and other search engines can also do more to explicitly incentivize attention to mobile accessibility concerns to create a wide-spread positive impact on the web.

Traffic and popularity statistics in this report are clear that mobile web is basically now just synonymous with web. The expectation of mobile website interaction should be a default in almost every scenario. This means that to move forward, site owners and the web community will need to continue to raise consciousness of this reality with the less technical teams, divisions, and groups that we work with. It is not enough to simply pay lip-service to the concept of mobile-first design and development; these concepts need to continue to be embraced and pushed when necessary. They also need to experience their own growth—outside of the web scenario to larger elements of business planning, marketing,

strategy, and communication.

Author



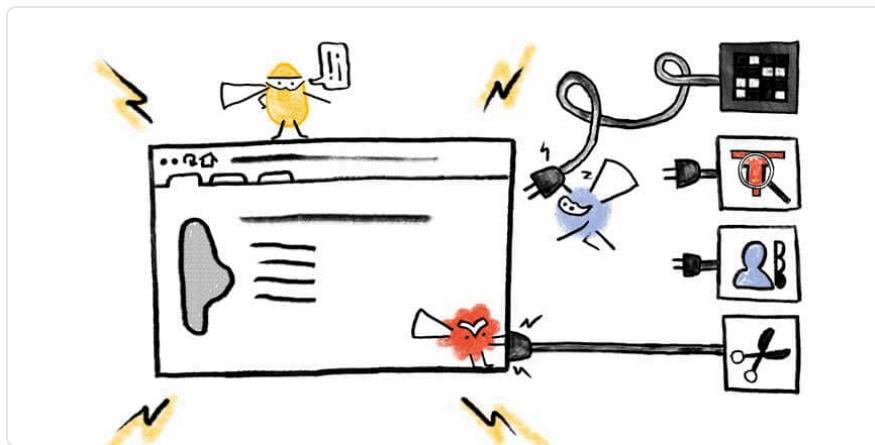
Cindy Krum

 @suzzicks  Suzzicks  <https://mobilemoxie.com/>

Cindy Krum is the CEO and Founder of MobileMoxie. She specializes in mobile SEO, app SEO (ASO), and anticipating and explaining changes in Google before they are announced.

Part II Chapter 16

Capabilities



Written by Michael Solati

Reviewed by Thomas Steiner and Christian Liebel

Analyzed and edited by Barry Pollard

Introduction

Compelling web experiences aren't limited to basic browser capabilities; they can take advantage of their underlying operating system. Web platform APIs expose these capabilities that are the foundation for Progressive Web Apps (PWA)—web applications capable of providing high-quality experiences like platform-specific apps.

In addition, some functionality on the web platform gives access to lower-level features such as access to the file system⁶⁰⁹, geolocation⁶¹⁰, access to the clipboard⁶¹¹, and even the ability to detect gamepads⁶¹².

609. https://developer.mozilla.org/docs/Web/API/File_System_Access_API

610. https://developer.mozilla.org/docs/Web/API/Geolocation_API

611. https://developer.mozilla.org/docs/Web/API/Clipboard_API

612. https://developer.mozilla.org/docs/Web/API/Gamepad_API

Methodology

This chapter used the HTTP Archive's public dataset of millions of pages. These pages were archived as if they were visited on both desktop and mobile, as some sites will serve different content based on what device is requesting the page.

The HTTP Archive's crawler then parsed the source code for all of these pages to determine which APIs were (potentially) used on the pages. For instance, regular expressions, such as `/navigator\.share\s*\(\)/g`, test pages to see if in the concrete case the Web Share API⁶¹³ is found in its source code.

This method does have two significant issues. First, it may underreport some APIs used as it can not detect obfuscated code that may exist due to minification, for example, when `navigator` was minified to `n`. Additionally, it may overreport occurrences of APIs because it does not execute code to see if an API is actually used. Regardless of these limitations, we believe this methodology should provide a sufficiently good overview of what capabilities are used on the web.

Seventy-five total regular expressions for supported capabilities exist; view this source file⁶¹⁴ to see all the expressions used.

The usage data in this chapter is from a crawl in June 2022; you can view the raw data in the Capabilities 2022 Results Sheet⁶¹⁵.

This chapter will also compare API usage to last year's usage; you can view the raw data from the previous year in the Capabilities 2021 Results Sheet⁶¹⁶.

Async Clipboard API

Our first API, the *Async Clipboard API*, allows read/write access to the system's clipboard.

Note that the Async Clipboard API replaces the deprecated `document.execCommand()` API to access the clipboard.

Write access

In order to write data into the clipboard, there are the `writeText()` and `write()` methods. The `writeText()` method takes a String argument and returns a Promise, while `write()`

613. https://developer.mozilla.org/docs/Web/API/Web_Share_API

614. <https://github.com/HTTPArchive/custom-metrics/blob/5d2f74fbdc580e76da5d1dad738fc8381429b9a/dist/fugu-apis.js>

615. <https://docs.google.com/spreadsheets/d/1359FRjBOPRtoMPb94fFh6pPNz3lNS9yvtuoorZYc288/edit#usp=sharing>

616. <https://docs.google.com/spreadsheets/d/1b4moteB9EILYkH1Ln9qfi1tnU-E4N2UQ87uayWytDkw/edit#gid=2077755325>

takes an array of `ClipboardItem` objects and also returns a Promise. `ClipboardItem` objects can hold arbitrary data, such as images.

A list of the mandatory data types a browser must support by the Clipboards API specification exists; see this list by the W3C⁶¹⁷. Unfortunately, not all vendors support the complete list; check browser-specific documentation when possible.

```
await navigator.clipboard.writeText("hello world");

const blob = new Blob(["hello world"], { type: "text/plain" });
await navigator.clipboard.write([
  new ClipboardItem({
    [blob.type]: blob,
  }),
]);
```

Read access

In order to read data from the clipboard, there are the `readText()` and `read()` methods. Both methods return a Promise which will resolve with data from the clipboard. The `readText()` method resolves as a String while `read()` resolves as an array of `ClipboardItem` objects.

```
const item = await navigator.clipboard.readText();
const items = await navigator.clipboard.read();
```

To keep user data safe, the `"clipboard-read"` permission of the Permissions API⁶¹⁸ must be granted to read data from the clipboard.

Both read and write access to the clipboard is available on modern versions of Chrome, Edge, and Safari. Firefox only supports `writeText()`.

⁶¹⁷ <https://www.w3.org/TR/clipboard-apis/#mandatory-data-types-x>

⁶¹⁸ https://developer.mozilla.org/docs/Web/API/Permissions_API

Growth of the Async Clipboard API

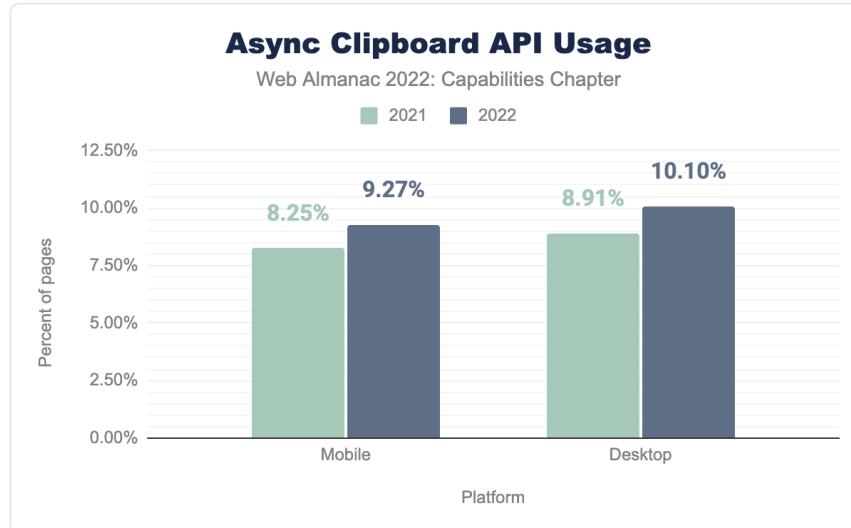


Figure 16.1. Usage of the Async Clipboard API from 2021 to 2022 on desktop and mobile.

The Async Clipboard API saw growth in usage from 8.91% in 2021 to 10.10% in 2022 on desktop. On mobile, there was also growth from 8.25% in 2021 to 9.27% in 2022. As a result, this year, the Async Clipboard API was the most used API on both desktop and mobile, beating the Web Share API (last year's most used API).

Web Share API

The *Web Share API* invokes the platform-specific sharing mechanism of the device, allowing data such as text, a URL, or files from a web application to be shared with any other application, such as mail clients, messaging applications, and more.

The method called to share data is `navigator.share()`. The `navigator.share()` method accepts an object containing the data to share and returns a Promise. Not every file type can be shared, though, and the `navigator.canShare()` method can test a data object to see if the browser can share it. You can see the list of shareable file types⁶¹⁹ on MDN.

After calling `navigator.share()`, the browser will open a platform-specific sheet where users select which application to share the data with.

⁶¹⁹ https://developer.mozilla.org/docs/Web/API/Navigator/share#shareable_file_types

Additionally, the Web Share API can only be triggered by a user's interaction with the page, such as a button click; the Web Share API cannot be called arbitrarily by executed code.

```
const data = {
  url: "https://almanac.httparchive.org/en/2022/capabilities",
};

if (navigator.canShare(data)) {
  try {
    await navigator.share(data);
  catch (err) {
    console.error(err.name, err.message);
  }
}
}
```

The Web Share API is available on modern versions of Chrome, Edge, and Safari. For Chrome, though, it is only supported on Windows and ChromeOS.

Growth of the Web Share API

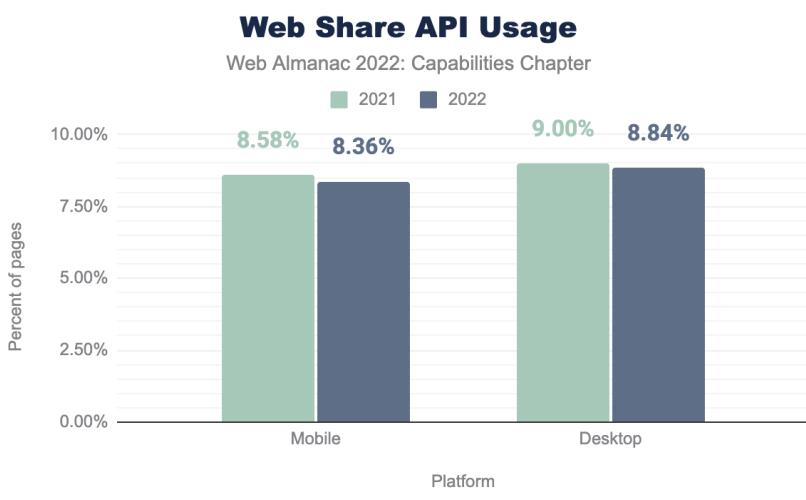


Figure 16.2. Usage of the Web Share API from 2021 to 2022 on desktop and mobile.

The Web Share API shrunk in usage from 9.00% in 2021 to 8.84% in 2022 on desktop. On mobile, usage shrunk from 8.58% in 2021 to 8.36% in 2022. As a result, this year, the Web Share API was the second most used API on both desktop and mobile, falling behind the Async Clipboard API—last year's second most used API.

On many sites, you can find the Web Share API in use. For example, social media platforms, documentation sites, and others use it as a great way to share content. Some examples where you can find the API in use include web.dev⁶²⁰ and twitter.com⁶²¹.

Sharing a Twitter profile using the Web Share API.

Figure 16.3. Sharing a Twitter profile using the Web Share API.

Add to Home Screen

The ability to add a web application to a device's home screen is a feature we didn't look at in last year's Capabilities report. To calculate how many sites have this functionality, pages were tested to see if they had a listener for the `beforeinstallprompt` event.

Note that the `beforeinstallprompt` event is a Chromium-only API and is currently incubating within the WICG⁶²².

The `beforeinstallprompt` event triggers right before a user is about to be prompted to "install" a web app. The usage of an event listener for the `beforeinstallprompt` event is not required for web apps to be added to a device's home screen, so it is safe to assume that the actual usage is much higher. However, this methodology will allow us to get an idea of how popular of a feature it is.

The ability to add an application to the home screen is a crucial feature of PWAs. To use this feature, web applications must meet the following criteria⁶²³:

- The web app must not already be installed.
- The user must have spent at least 30 seconds viewing the page at any time.
- The user must have clicked or tapped at least once on the page at any time.
- The web app must be served over HTTPS.

620. <https://web.dev/>

621. <https://twitter.com/>

622. <https://wicg.github.io/manifest-incubations/index.html#installation-prompts>

623. <https://web.dev/install-criteria/#criteria>

- The web app must include a web app manifest⁶²⁴ with:
 - `short_name` or `name`.
 - `icons` (must include a 192×192px and a 512×512px icon).
 - `start_url`.
 - `display` (must be one of `fullscreen`, `standalone`, or `minimal-ui`).
 - `prefer_related_applications` (must not be present, or `false`).
- The web app must register a service worker with a `fetch` handler.

Installed applications can appear in Start menus, desktops, home screens, the Applications folder, when searching for applications on a device, content sharing sheets, and more.

The ability to add to the home screen is only available on modern versions of Chrome, Edge, and Safari on iOS and iPadOS.

Usage of Add to Home Screen



Figure 16.4. Usage of Add to Home Screen on mobile.

As mentioned, the add to home screen capability was not measured last year. However, for posterity and detailed reporting, the `beforeinstallprompt` event was used on 8.56% of desktop pages and 7.71% of mobile pages, making it the third most used capability on desktop and mobile.

By taking advantage of the `beforeinstallprompt` event, developers can provide a customized experience in how users install their web application. One example is YouTube TV, which invites users to install their application to access it more quickly and easily.

⁶²⁴. <https://developer.mozilla.org/docs/Web/Manifest>

Installing YouTube TV from an in app prompt, powered by the beforeinstallprompt event.

Figure 16.5. Installing YouTube TV from an in app prompt, powered by the beforeinstallprompt event.

Media Session API

The *Media Session API* allows developers to create custom media notifications for audio or video content on the web. The API includes action handlers that browsers can use to access media control on keyboards, headsets, and the software controls on a device's notification area and lock screens. The Media Session API empowers users to know and control what's playing on a web page without needing to be actively viewing said page.

When a page plays audio or video content, users get a media notification that appears in their mobile device's notification tray or on their desktop's media hub. Browsers will try to show a title and an icon, but the Media Session API allows the notification to be customized with rich media metadata, such as the title, artist name, album name, and album artwork.

```
navigator.mediaSession.metadata = new MediaMetadata({
  title: "Creep",
  artist: "Radiohead",
  album: "Pablo Honey",
  artwork: [
    {
      src: "https://via.placeholder.com/96",
      sizes: "96x96",
      type: "image/png",
    },
    {
      src: "https://via.placeholder.com/128",
      sizes: "128x128",
      type: "image/png",
    },
    {
      src: "https://via.placeholder.com/192",
      sizes: "192x192",
    }
  ]
})
```

```
        type: "image/png",
    },
{
    src: "https://via.placeholder.com/256",
    sizes: "256x256",
    type: "image/png",
},
{
    src: "https://via.placeholder.com/384",
    sizes: "384x384",
    type: "image/png",
},
{
    src: "https://via.placeholder.com/512",
    sizes: "512x512",
    type: "image/png",
},
],
});
});
```

The Media Session API is available on modern versions of Chrome, Edge, Firefox, and Safari.

Usage of the Media Session API



Figure 16.6. Usage of Media Session API on mobile.

The Media Session API was not measured last year. In its first year of tracking, the API was used on 8.37% of desktop pages and 7.41% of mobile pages, making it the fourth most used capability on desktop and mobile.

Web applications such as YouTube, YouTube Music, Spotify, and others take advantage of the Media Session API and provide rich controls for the video or audio played.

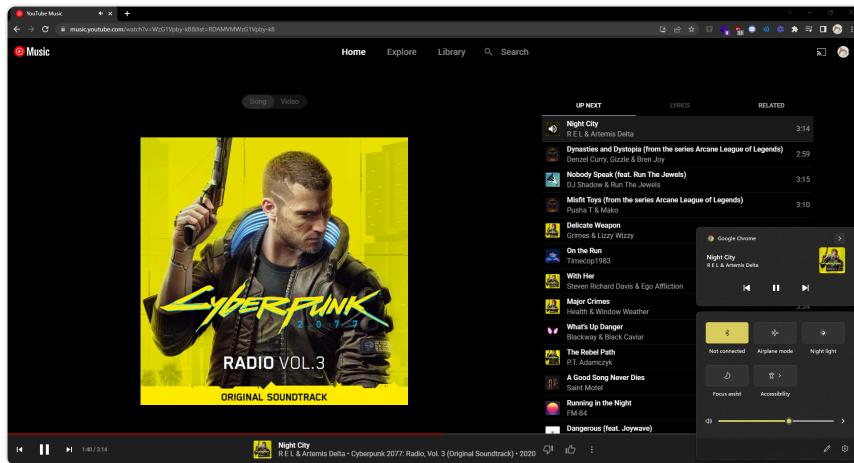


Figure 16.7. Accessing controls and information for YouTube Music via the Window's Taskbar.

For a deeper dive into video usage on the web, check out the Media chapter.

Device Memory API

A device's capabilities depend on a few things, like the network, the CPU core count, and the amount of memory available. The `Device Memory API` provides insight into the memory available by providing the read-only property `deviceMemory` on the `Navigator` interface. The property returns an approximate amount of device memory in gigabytes as a floating point number.

The value returned is imprecise, protecting the user's privacy. It's calculated by rounding down to the nearest power of 2, then dividing that number by 1,024. The number is also clamped within an upper and lower bound. So you can expect the numbers: `0.25`, `0.5`, `1`, `2`, `4`, and `8` (gigabytes).

```
const memory = navigator.deviceMemory;
console.log('This device has at least ', memory, 'GiB of RAM.');
```

The Device Memory API is only available on modern versions of Chrome and Edge.

Usage of the Device Memory API

5.76%

Figure 16.8. Usage of Device Memory API on mobile.

The Device Memory API was not measured last year. In its first year of tracking, the API was used on 6.27% of desktop pages and 5.76% of mobile pages, making it the fifth most used capability on desktop and mobile.

For the release of Facebook's 2019 redesign, FB5, they actively integrated adaptive loading into this new version. They did this by adapting based on users' actual hardware, changing what loaded and what ran based on what users were using. For example, on the desktop, Facebook defined buckets of users based on CPU cores (`navigator.hardwareConcurrency`) and device memory (`navigator.deviceMemory`) available.

Check out this video⁶²⁵ from Chrome Dev Summit 2019, starting at 24:03, where Nate Schloss shares how Facebook handles adaptive loading using features such as the Device Memory API.

Service Worker API

Service workers are one of the core components of Progressive Web Apps. They act as a client-side proxy that puts developers in control of the system's cache and how to respond to resource requests. By pre-caching essential resources, developers can eliminate the dependence on the network, ensuring instant and reliable experiences.

In addition to caching resources, service workers can update assets from the server, allow for push notifications, and allow access to the background and periodic background sync APIs.

While service workers have become widely adopted and supported by major browsers, not all features of service workers are available on all browsers. An example of a currently unsupported feature is that of the Push API on Safari. Safari will support the Push API in the upcoming release of macOS Ventura⁶²⁶ in 2022 and iOS 16⁶²⁷ and iPadOS 16 in 2023.

The Service Worker API is available on modern versions of Chrome, Edge, Firefox, and Safari.

625. <https://www.youtube.com/watch?v=puUPpVrRkc&t=1443s>

626. <https://www.apple.com/macos/macos-ventura-preview/features/>

627. <https://www.apple.com/ios/ios-16/features/>

Growth of the Service Worker API

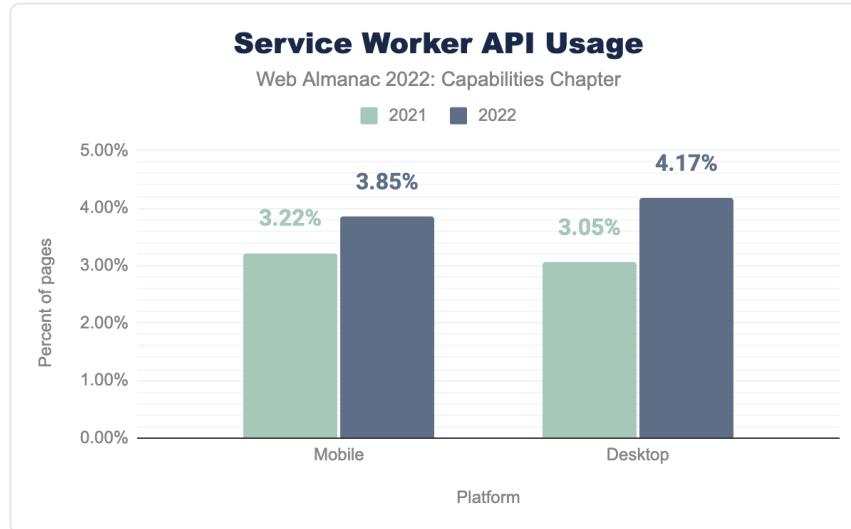


Figure 16.9. Usage of the Service Worker API from 2021 to 2022 on desktop and mobile.

The Service Worker API was not measured in last year's Capabilities chapter. However, using data from the previous year's PWA chapter, the API grew in usage from 3.05% to 4.17% on desktop and 3.22% to 3.85% on mobile pages, making it the sixth most used capability on desktop and the seventh most used mobile.

Note that how the service worker usage in the PWA chapter is measured differs from how the Capabilities chapter measures it. Additionally, a bug in the data pipeline for last year's PWA chapter was found, resulting in an undercounting of service worker usage.

For a deeper dive into service worker usage on the web, check out the PWA chapter of the 2022 Web Almanac.

Gamepad API

The *Gamepad API* is how web applications respond to input from gamepads and other game controllers. This API has three interfaces; one that represents the controller connected to the device, one that represents buttons on the connected controller, and finally, one that is for events fired when a gamepad is connected or disconnected.

```
window.addEventListener("gamepadconnected", (e) => {
  const gp = navigator.getGamepads()[e.gamepad.index];
  console.log(`Controller connected at index ${gp.index}`);
});
```

The Gamepad API is available on modern versions of Chrome, Edge, Firefox, and Safari.

Growth of the Gamepad API

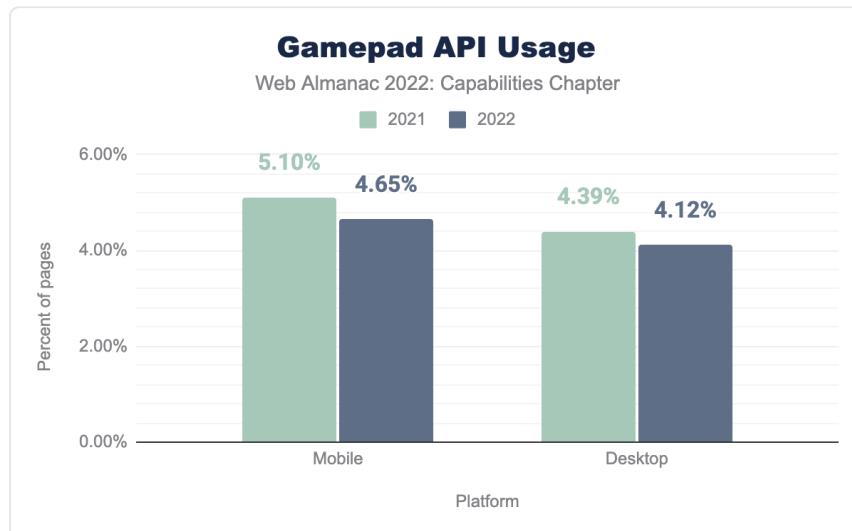


Figure 16.10. Usage of the Gamepad API from 2021 to 2022 on desktop and mobile.

The Gamepad API shrunk in usage from 4.39% in 2021 to 4.12% in 2022 on desktop. On mobile, use shrunk from 5.10% in 2021 to 4.65% in 2022. As a result, this year, the Gamepad API was the seventh most used capability on desktop and the sixth most used mobile.

Web applications such as Google's Stadia, NVIDIA's GeForce Now, and Microsoft's Xbox Cloud Gaming provide gaming experiences that run on the cloud comparable to the experience of running games on local devices or a gaming console. Thanks to the Gamepad API, these web applications allow users to use traditional console game controllers rather than just a keyboard and mouse.

Connecting an Xbox controller to Google Stadia in the Chrome browser.

Figure 16.11. Connecting an Xbox controller to Google Stadia in the Chrome browser.

Push API

The *Push API* allows web applications to receive messages from a server regardless of whether the application was in the foreground. Developers can send asynchronous notifications and updates to users who opt in, giving them meaningful updates and a nudge to reengage with an application.

Web applications must also have a service worker to receive push notifications from a server. From within the service worker, push notifications can be subscribed to using the `PushManager.subscribe()` method.

The Push API is available on modern versions of Chrome, Edge, and Firefox.

Usage of the Push API

1.86%

Figure 16.12. Usage of Push API on mobile.

The Push API was not measured last year. In its first year of tracking, the API was used on 2.03% of desktop pages and 1.86% of mobile pages, making it the eighth most used capability on desktop and mobile.

Project Fugu

Many features users expect to belong to platform-specific applications also exist on the web. However, thanks to the Capabilities Project, known by many as Project Fugu, these features exist on the web. Project Fugu is a cross-company effort to bring feature parity to web applications, considering what iOS, Android, or desktop apps can do. Project Fugu works on exposing platform-specific capabilities to the web while maintaining user security, privacy, trust, and the web's other core tenets.

Project Fugu comprises Microsoft, Intel, Samsung, Google, and many other groups and

individuals.

Check out this post⁶²⁸ on the Chrome Developers blog to learn more about the Capabilities Project.

Conclusion

Capabilities unlock new possibilities and functionality for developers to take advantage of on the web. This chapter shared eight of the most popular web platform APIs currently being used on the web. It also showcased some of these capabilities used in different web applications. The beauty of the web is that it can use these platform-based functionalities without needing to (necessarily) be installed onto a device or additional libraries and plugins.

Some exciting experiences that utilize the web's capabilities include What Web Can Do Today?⁶²⁹ (WWCDT) and Discourse⁶³⁰. WWCDT, which uses 38 of the capabilities we track, showcases many Web APIs with a live demo of each API. Discourse provides communities with web forums and uses 14 of the capabilities we track, such as the Badging API, so users can see the number of unread notifications they have.

The Capabilities Project, Project Fugu, allows applications to migrate to the web, removing some barriers associated with platform-specific applications. No need to write "native" code, no need to worry about users having access to the latest updates, and no need to get users to search for and download from your application in app stores. The web, and its capabilities, open up all new possibilities in building compelling experiences for users.

Author



Michael Solati

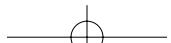
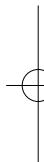
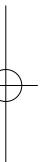
Twitter: [@MichaelSolati](https://twitter.com/@MichaelSolati) GitHub: [@MichaelSolati](https://github.com/MichaelSolati) Website: <https://michaelsolati.com>

Michael is a Developer Advocate at Amplification, focusing on helping developers build APIs and drink IPAs. Additionally, he is a Web GDE and has found his love in creating compelling experiences on the web and the voodoo ways of the web...

628. <https://developer.chrome.com/blog/fugu-status/>

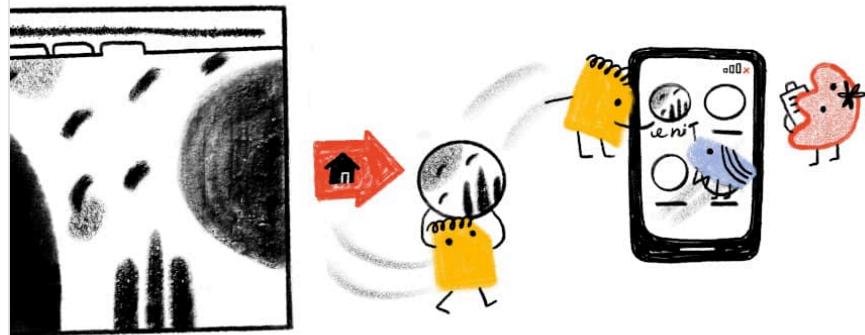
629. <https://whatwebcando.today/>

630. <https://www.discourse.org/>



Part II Chapter 17

PWA



Written by Diego Gonzalez

Reviewed by Aaron Gustafson, Adriana Jara, Maxim Salnikov, Kai Hollberg, and Beth Pan

Analyzed by Beth Pan

Edited by Siwin Lo and Barry Pollard

Introduction

In the early days of Progressive Web Apps, there were two key features that harnessed the promise of an advanced modern web application: offline support and a direct icon on the home screen of the device.

These two concepts were enabled after installing a PWA, a process that generally began by tapping on an “ambient badge” that would appear on the browser’s URL bar. This badge would prompt the user to install the website. Mobile browsers such as Samsung Internet and Mozilla Firefox, were among the first ones to explicitly support this behavior, commonly known as “Add to home screen” (A2HS)⁶³¹.

Five years ago, this was a radical idea. A website would be able to launch directly from the home screen, listed alongside other applications a user had installed on their device. This was the

631. https://developer.mozilla.org/docs/Web/Progressive_web_apps/Add_to_home_screen

start of progress made towards reducing the gap between capabilities of web apps and OS-specific experiences.

The A2HS scenario has evolved into web apps that can be fully installed and deeply integrated into the host OS, in both mobile and desktop contexts. These past 12 months have seen browsers take important steps towards making sure that PWAs have a tight integration with desktop platforms, and many of the new additions to this year's almanac reflect these changes. This is the state of PWAs in 2022.

Note: As a set of web technologies, PWAs are not isolated from the rest of the web platform. While there is a chapter dedicated to Capabilities, this year we have investigated the intersection of some of these new advanced capabilities when used inside a PWA.

Service workers

Service workers⁶³² is one of the core technologies of PWAs and the enabler of offline apps, getting push notifications, and doing background processing. They serve as the base for most of the advanced experiences we expect from applications. They are also being used to define data updates and for upcoming modern functionality like widgets based on PWA technologies⁶³³.

While there isn't parity between major browsers when it comes to service worker feature support, Webkit adding support for push notifications⁶³⁴ was a huge milestone. Earlier this year it was announced that Apple had made changes⁶³⁵ to their desktop platform to support the relevant parts of the Push API⁶³⁶, Notifications API⁶³⁷ and that service workers⁶³⁸ would enable Web Push. They also announced the feature would be coming to their mobile platforms in 2023.

Service worker usage

For comparison reasons, we have run the same queries as last year, which allows us to try to make sense of the evolution of service worker usage. Last year's chapter gave the explanation of why it isn't trivial to find out actual usage of service worker⁶³⁹, and that is just as true this year.

Looking at two of the measures:

632. https://developer.mozilla.org/docs/Web/API/Service_Worker_API
633. <https://github.com/aarongustafson/pwa-widgets#rich-widgets>
634. <https://caniuse.com/push-api>
635. <https://webkit.org/blog/12945/meet-web-push/>
636. https://developer.mozilla.org/docs/Web/API/Push_API
637. https://developer.mozilla.org/docs/Web/API/Notifications_API
638. https://developer.mozilla.org/docs/Web/API/Service_Worker_API
639. <https://almanac.httparchive.org/en/2021/pwa#service-workers-usage>

- Lighthouse detects a 1.6% (mobile) and 1.7% (desktop) of all websites employ a service worker. We expect this is lower than the real-world percentage due to additional checks⁶⁴⁰ that Lighthouse takes into consideration.
- Following the same metrics introduced last year⁶⁴¹, usage of a Service Worker in websites comes up to 1.63% on desktop and 1.81% on mobile.

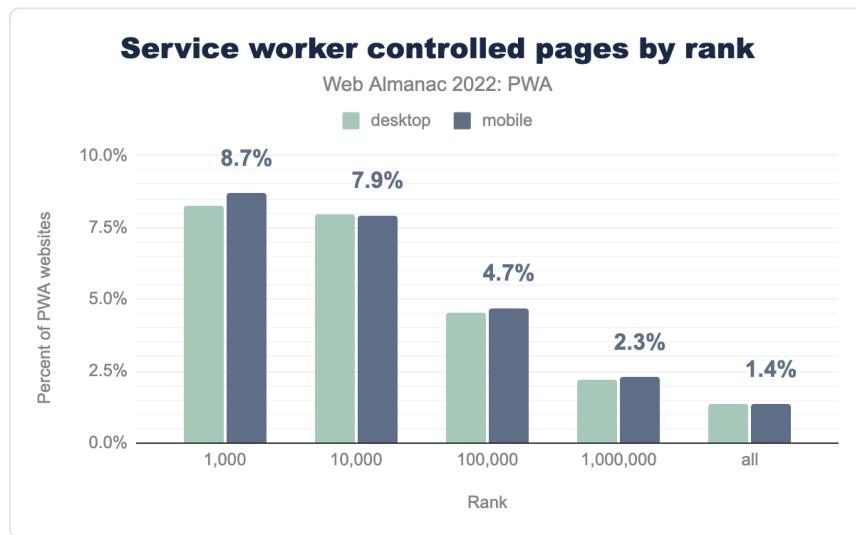


Figure 17.1. Service worker controlled pages by rank.

There hasn't been a noticeable change for service worker controlled pages in the top 1,000 sites as well, with only a slight decrease in desktop and even smaller increase in mobile properties. But there was an increase in all other categories. If we follow the reasoning from last year⁶⁴²—where we postulated that bigger websites adopted the advanced technologies faster—then seeing more growth in other categories makes sense. It would seem smaller companies and developers have learned and adopted the technology shared from case studies and examples from the companies with more traffic.

Service worker events

A service worker acts as a proxy server that sits between the web app, the browser and the network. To install a service worker it must be fetched and registered. If this is successful, the

640. <https://web.dev/service-worker/>

641. https://github.com/HTTPArchive/legacy.httparchive.org/blob/master/custom_metrics/pwajs

642. <https://almanac.httparchive.org/en/2021/pwa#fig-2>

service worker is executed in a special worker container⁶⁴³ that runs off the main thread and has no DOM access. This is when events can be processed.

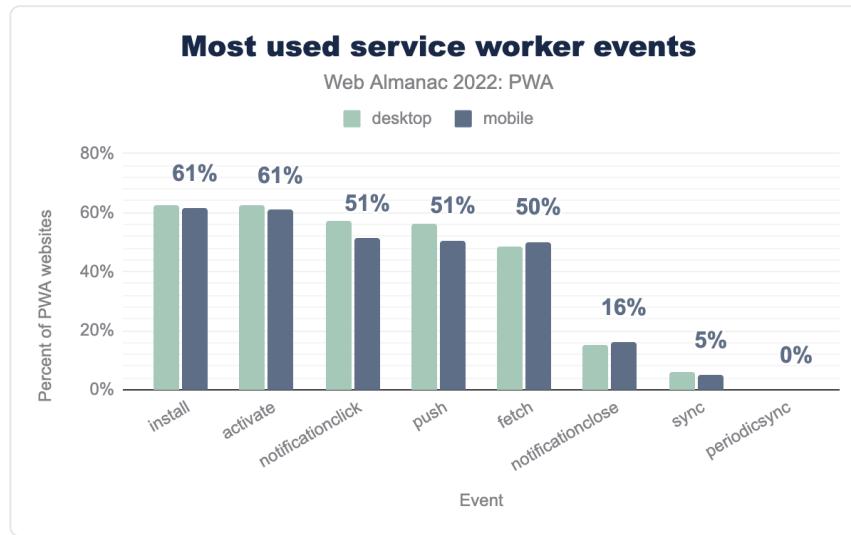


Figure 17.2. Most used service worker events.

The previous chart displays information on the most used service worker events. Each one of these events can be categorized into:

- Lifecycle events
- Notification-related events
- Background processing events

Lifecycle events

`install` and `activate` are lifecycle events. It is common practice to create a cache of assets that will allow running the app offline when installing. Activation is generally used to clean up old caches associated with the previous service worker.

^{643.} <https://developer.mozilla.org/docs/Web/API/ServiceWorkerGlobalScope>

61%

Figure 17.3. Service worker events that are `install` events on mobile.

The service worker needs to be active in order to receive events like fetch and push. This makes them the cornerstone of service workers, and hence the 63% usage on desktop and 61% on mobile for `install`, and the same for `activate`.

The remaining ~40% of sites with service worker might not be actively using these events, as they can be using service worker for notifications or utilizing techniques to cache requests made only when the site is running, also known as runtime caching⁶⁴⁴.

While these are still the most used events, the increase of other types of events being used leads us to speculate that there is an increased number of service workers not (only) being used for pre-caching as their main task.

Notification-related events

57%

Figure 17.4. Service worker events that are `notificationclick` events on desktop.

Push notification events come next in most used service worker methods.

- `notificationclick` comes up to 57% ($\Delta 11\%$ over last year's data) on desktop and 51% ($\Delta 5\%$) on mobile.
- `push` 56% ($\Delta 12\%$) on desktop and 50% ($\Delta 5\%$) on mobile.
- `notificationclose` is now at 15% ($\Delta 9\%$) on desktop and 16% ($\Delta 10\%$) on mobile.

A couple of takeaways here is that momentum continues to grow this year for PWAs on desktop, and push notifications is not an exception. Usage of related events for notifications has gone up around 11%. Many tweaks and fixes have been worked on in different platforms to make sure that these pieces of UX feel completely integrated with the host OS. We expect

⁶⁴⁴. <https://web.dev/runtime-caching-with-workbox/>

these numbers to continue growing, following the newly announced support for Web Push on Webkit⁶⁴⁵. This is a feature that has been requested by many developers for a long time and finally having support on macOS—and hopefully soon iOS devices—can encourage developers to use the API.

Background processing events

A large, bold, blue percentage value '49%' with a thin white border around it, centered on the page.

49%

Figure 17.5. Service worker events that are `fetch` events on desktop.

The remaining events in the chart represent background processing events:

- `fetch`, which occurs when a request is sent to the server, can be used to intercept said request and respond with custom or cached assets, enabling offline support for our PWAs. Fetch usage is 49% on desktop and 50% on mobile.
- `sync`, which fires when the UA believes the user has connectivity, has a usage of 6% on desktop and 5% on mobile.
- `periodicsync`, which allows web applications to periodically synchronize data in the background, is currently at 0.01% on both desktop and mobile platforms. It should be noted that `periodicsync` is limited to a max of once every 12 hours. This can be artificially suppressing usage of the feature.

645. <https://webkit.org/blog/12945/meet-web-push/>

Other popular SW features

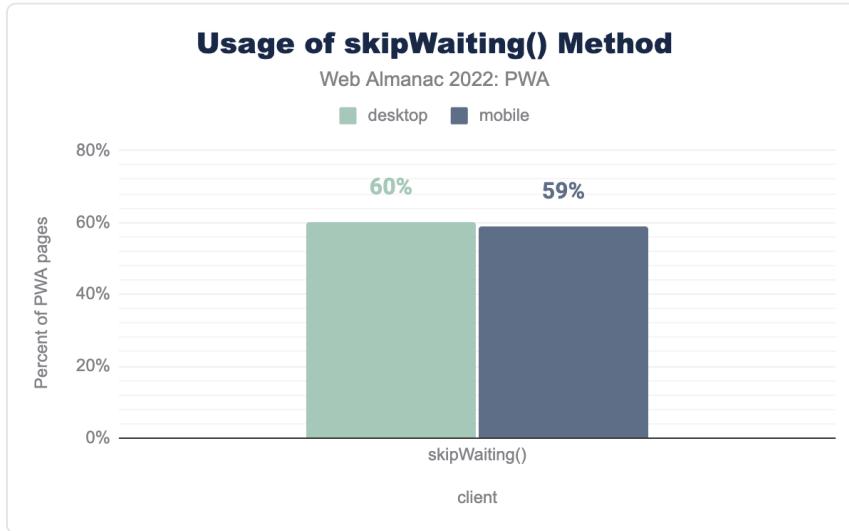


Figure 17.6. Usage of `skipWaiting()` method.

Similar to the stats of last year⁶⁴⁶, the `skipWaiting()` method that is used to immediately activate the service worker is still very popular among developers, being present on 60% of desktop and 59% of mobile web apps.

These are the top most used service worker objects:

646. <https://almanac.httparchive.org/en/2021/pwa#other-popular-service-worker-features>

Most used service worker objects

Web Almanac 2022: PWA

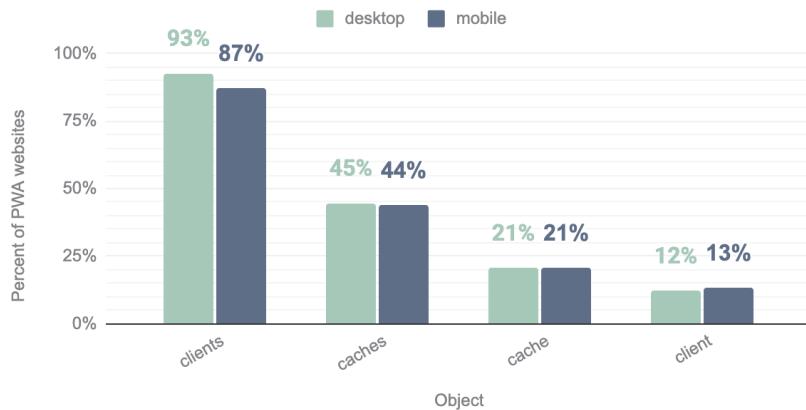


Figure 17.7. Most used service worker objects.

And these are the most used methods:

Most used service worker objects

Web Almanac 2022: PWA

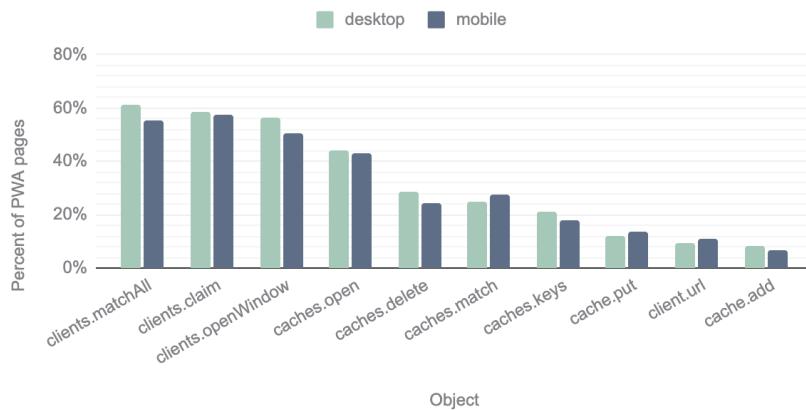


Figure 17.8. Most used service worker object methods.

Web App Manifest

The *Web App Manifest* file is a JSON file that contains information about the application itself. The manifest file is the other main core technology that defines PWAs. Among the data that is present in the key-value pairs is information relevant to display, promote and integrate the app into the host OS.

Keeping the web app's manifest fully authored is essential to take advantage of advanced discoverability through online repositories, submissions to application stores, and more recently, a way to tap into advanced capabilities like share target and file handling for your app. Cutting edge work to enable Widgets based on PWA technology⁶⁴⁷ is also being rooted in the manifest, proving the versatility of the file itself for advanced platform integration even further.



Figure 17.9. Percent of manifest files parsable on desktop.

For most cases—95% in desktop and 94% in mobile—the manifests we found are JSON parsable. This indicates that almost all web apps that use the manifest are correctly formed.

This does not indicate completeness or minimum availability of certain fields that would contribute to the installation of the web app. As a matter of fact, there is currently no required properties for the Manifest file. An empty file technically is a valid Manifest file.

The manifest file plays a key part in signaling to the browser to prompt for installation, though the way the prompt is triggered varies with different browsers⁶⁴⁸, there's always a subset of the manifest file involved.

Below are the usage numbers of manifest file alongside service worker. These two used in conjunction generally imply installability.

647. <https://github.com/aarongustafson/pwa-widgets#how-widgets-are-represented-in-these-apis>

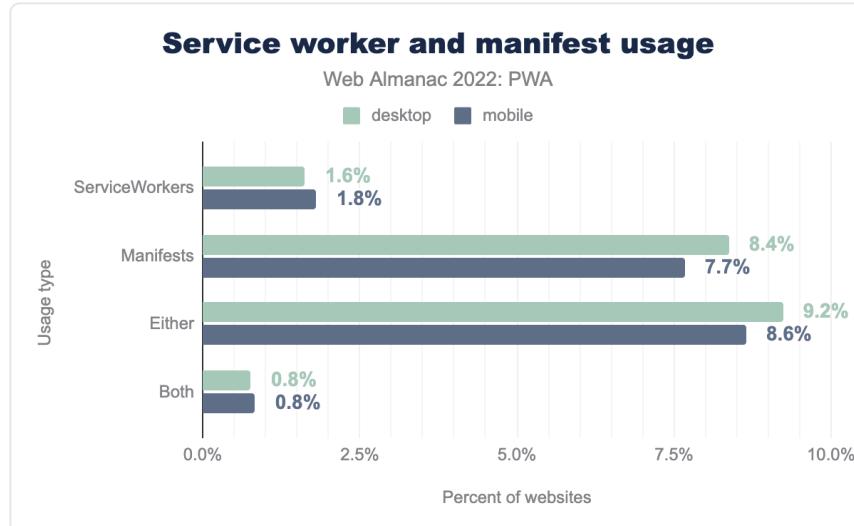


Figure 17.10. Service Worker and Manifest usage.

The data tells us that web applications are around 5 times more likely to have a manifest file than a service worker. A contributing factor is that many platforms, such as Content Management Systems (CMSs), auto-generate manifest files for websites.

Only a small percentage of websites—0.8% on both desktop and mobile—implement both service worker and manifest files, which means less than 1% of websites can be installed on devices like traditional apps.

For this chapter we are mostly interested in sites that have both a service worker and a manifest so—unless otherwise noted—the manifest data present in this chapter are for PWA sites.

Manifest properties

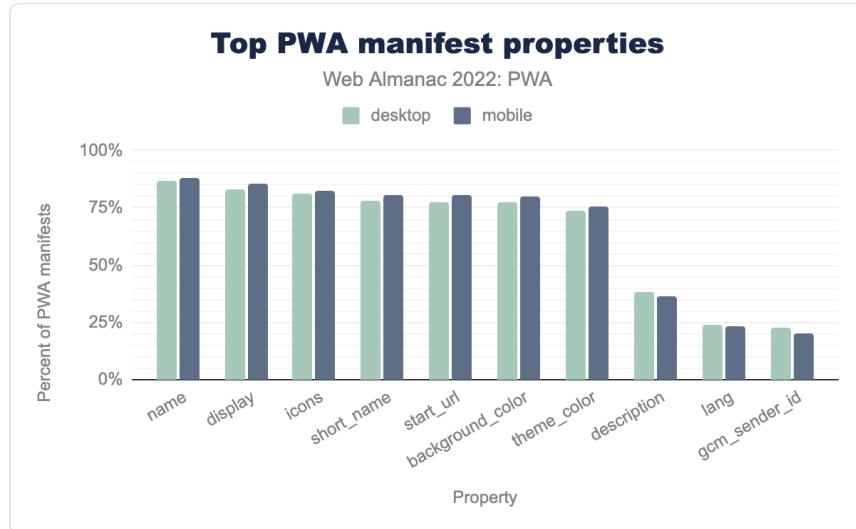


Figure 17.11. Top PWA manifest properties.

Looking at top properties used in manifest files this year as compared to last year, there is no significant change.

Note that the `gcm_sender_id`, is not a standardized property. It is used by the Google Developer Console to identify an app and enabled older versions of Chrome to implement web push, which relied on the GCM service.

Most PWAs, 80% for desktop and 79% for mobile do not define a preferred orientation. When set, the most frequently used value is “portrait,” with 13% on desktop and 15% on mobile web sites defining that value on their manifest.

display property

Digging into the `display` property more, we see the following values:

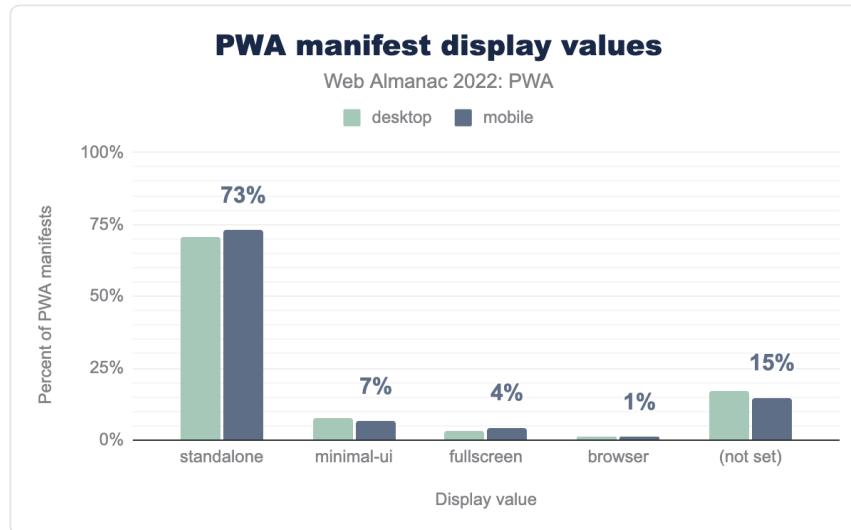


Figure 17.12. PWA manifest display values.

`display: standalone` mode is the most common display mode, used by almost 3/4 of websites that define a display mode. It's also one of the display modes that enables an app to be installable.

icons property



Figure 17.13. Top PWA manifest icon sizes.

PWAs need to generate different icon sizes to accommodate the range of surfaces where the app can be advertised and placed. Many tools exist to generate the plethora of icons needed for different desktop and mobile environments. The 2 most common icon sizes present in manifest files are `192x192` and `512x512`. Both sizes appear in around 70% of the manifest files analyzed.

Installation and discoverability properties

A web app manifest file can contain data that is useful in describing of the application. These properties can be used by stores or other distribution mechanisms to promote the application. An increased growth of rich browser-based installation dialogs⁶⁴⁹ is also utilizing these fields more prominently. Relevant fields, which can be found as part of the Application Information supplement to the Manifest file are listed below:

- `description` : This property exists in 36% of desktop and 34% of mobile web app manifests. The description is important since it explains what the application does. It's commonly used to provide information about the app for a store. Currently around a third of installable PWAs would offer this information.

649. <https://developer.chrome.com/blog/richer-pwa-installation/>

- `screenshots` : This property provides the URLs of one or more screenshots for use in app stores and in-browser install prompts. PWAs with manifests that take advantage of this feature total 1.12% for desktop and 1.19% for mobile devices.
- `categories` : Used as hints for catalog organization.
- `iarc_rating_id` : It's a string that represents the IARC certification code⁶⁵⁰ of the web app. 0.05% of desktop and mobile apps utilize this field to advertise the rating of their app or game.

Manifest categories

Let's dig into the categories little more.

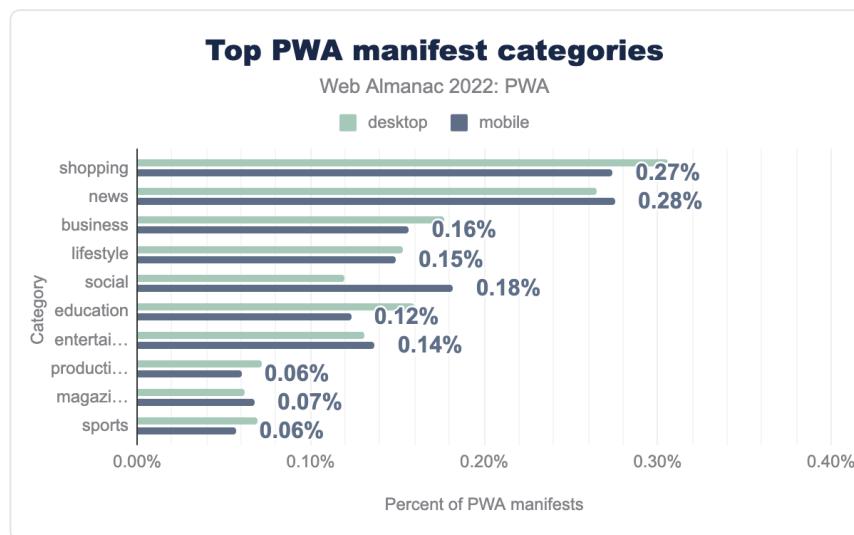


Figure 17.14. Top PWA manifest categories.

And we will also show the same data for all websites, rather than just those with a service worker which we are using as our definition of “PWA sites”:

⁶⁵⁰ <https://www.globalratings.com/how-iarc-works.aspx>

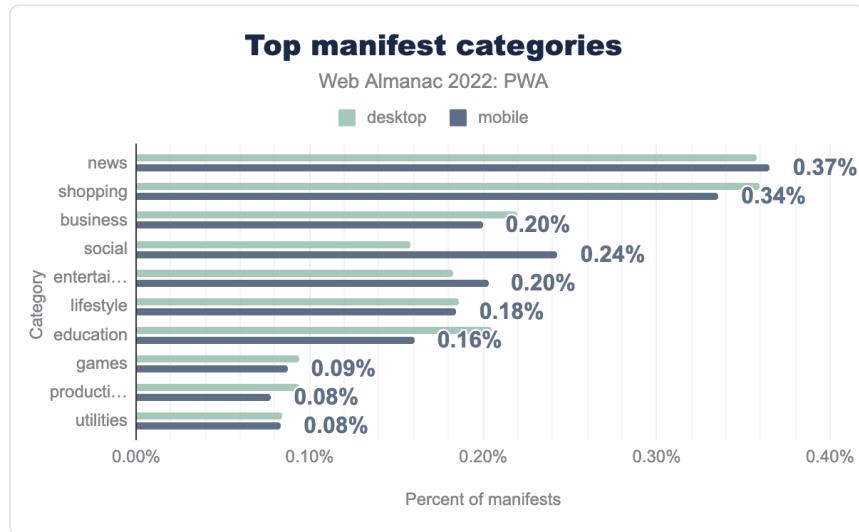


Figure 17.15. Top manifest categories.

The top categories for both websites and PWAs remain the same, yet each is slightly different. Top categories are shopping, news, and business.

Advanced capabilities

The manifest file also allows for the activation of modern platform capabilities. These capabilities can allow for advanced windowing capabilities or registration of behaviors in the host OS. Many of these capabilities have landed very recently into the platform, and therefore, we hope this data register an inception of many of these new APIs.

As these are lesser-use, more advanced, capabilities they do not show on our previous graph of the top manifest properties, but are worth looking at to see their usage too:

- `shortcuts` : 6.2% of desktop and 4.3% of mobile PWAs are using shortcuts to deep link into the app.
- `file_handlers` : allows an installed PWA to register itself as a handler for a specific file extension. Only 0.01% of desktop and 0.02% of mobile are using `file_handlers`.
- `protocol_handlers` : PWAs can register to be handlers for predefined or custom protocols. Current usage stands at 0% on desktop and 0.01% for mobile web sites.

- `share_target` : 5.3% of desktop and 3.1% of mobile PWAs have the ability to register themselves to receive shared data.
- Window Controls Overlay⁶⁵¹: The ability to free the area generally occupied by the title bar is a desktop only feature. This feature has recently launched in Chromium 105 and 0.01% of manifests of desktop PWAs are utilizing it.
- `note_taking` : 0% of desktop and 0.01% of mobile sites are using the ability to have special integration as a convenient way of taking a quick note.

Manifest preferring native

2.0%

Figure 17.16. Manifest files with a `related_applications` field on mobile

There is a property in the manifest that specifies if applications listed in the `related_applications` field should be preferred over the web application. This might make the user agent suggest the installation of the related app instead of the web app. From all the manifest files analyzed, only 2.3% on desktop and 2.0% on mobile manifests set this property.

Fugu APIs

PWAs go hand in hand with advanced web capabilities. These capabilities are generally part of project Fugu which is the codename for a collection of new web platform features incubating within the Chromium project.

8.8%

Figure 17.17. Most used Fugu API (desktop)

From the growing list of features that have been added to the web platform, these are the top APIs being used on the web that are useful for PWAs with web:

^{651.} <https://wicg.github.io/window-controls-overlay/>

<i>Api</i>	Desktop	Mobile
Web Share	8.8%	8.4%
Add to Home Screen	8.6%	7.7%
Service worker	4.2%	3.9%
Push	2.0%	1.9%

Figure 17.18. Most used Fugu APIs.

We won't delve into these much more as we have a separate Capabilities chapter that covers them.

PWA insights from Lighthouse

Lighthouse⁶⁵² is an open-source, automated tool for improving the quality of web pages. It can be used to run a number of audits on a website and has a dedicated category for PWA audits. The available data sheds some light on interesting facts about the state of PWAs these past 12 months.

652. <https://developer.chrome.com/docs/lighthouse/>

Lighthouse audits

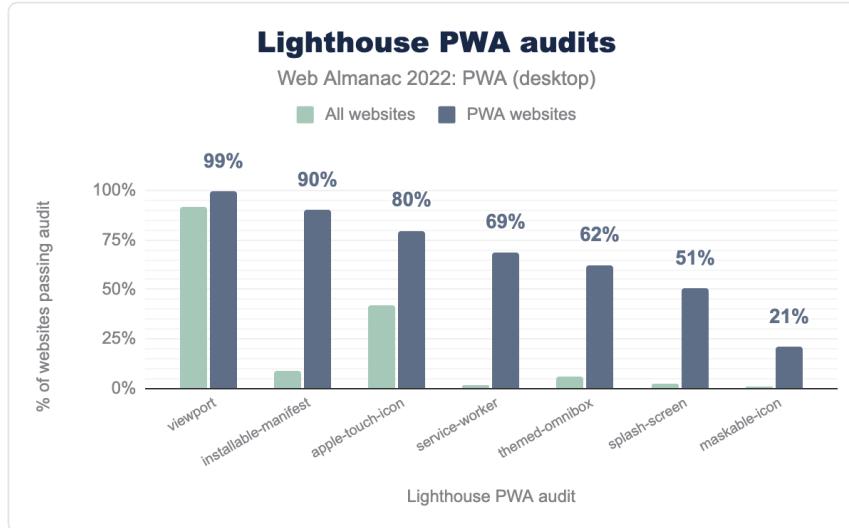


Figure 17.19. Lighthouse PWA Audits for desktop.

It is not surprising to see PWA sites passing the PWA audits much more frequently than the general web, though some audits such as the presence of a viewport meta tag⁶⁵³ and the apple-touch-icon⁶⁵⁴ meta tag are also often applicable—and used—by non-PWA sites.

653. <https://web.dev/viewport/#how-to-add-a-viewport-meta-tag>
 654. <https://web.dev/apple-touch-icon/#how-the-lighthouse-apple-touch-icon-audit-fails>

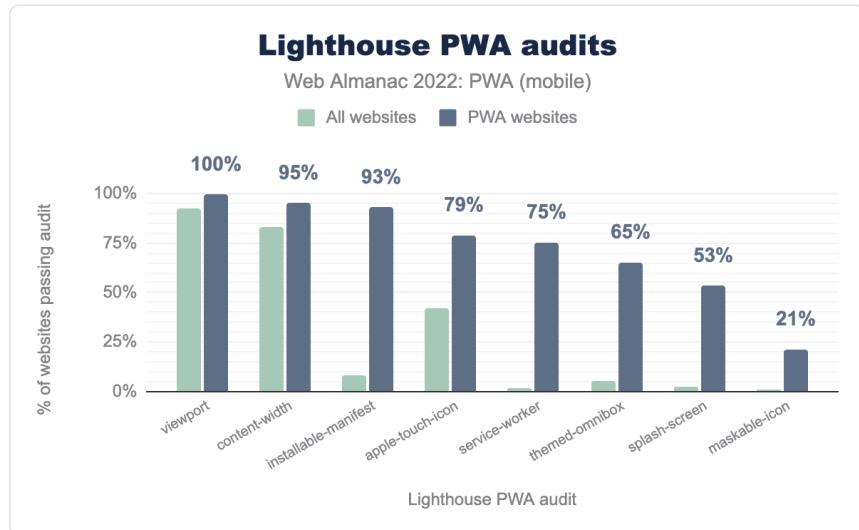


Figure 17.20. Lighthouse PWA Audits for mobile.

Looking at the Lighthouse data on mobile sites we see similar stats, but the mobile-only `content-width`⁶⁵⁵ shows here and is pleasingly passed by both.

The presence of a `viewport` meta tag is relevant because it removes a 300-350ms delay that waits for a double-tap back when that was the way to zoom in. It has the additional benefit on mobile devices of optimizing the app for the device's screen size. It is not surprising that almost all websites, PWA or not, include this.

Installable manifest also appears in both top 3 lists. As expected, this has a very high value for PWA sites, both on desktop (90.2%) and mobile (95.2%), with a very low counterpart for all websites, presumably because developers don't intend for these to be installed.

Finally, `apple-touch-icon` is third on PWA-related Lighthouse audits. Since iOS 1.1.3, Safari for iOS has supported a way for developers to specify an image that will be used to represent the web site or app on the home screen. This is mostly relevant for mobile devices.

Lighthouse scores

To conclude the Lighthouse insights section, we take a look at the overall Lighthouse PWA scores for PWA sites, based on the audits.

655. <https://developer.chrome.com/docs/lighthouse/pwa/content-width/>

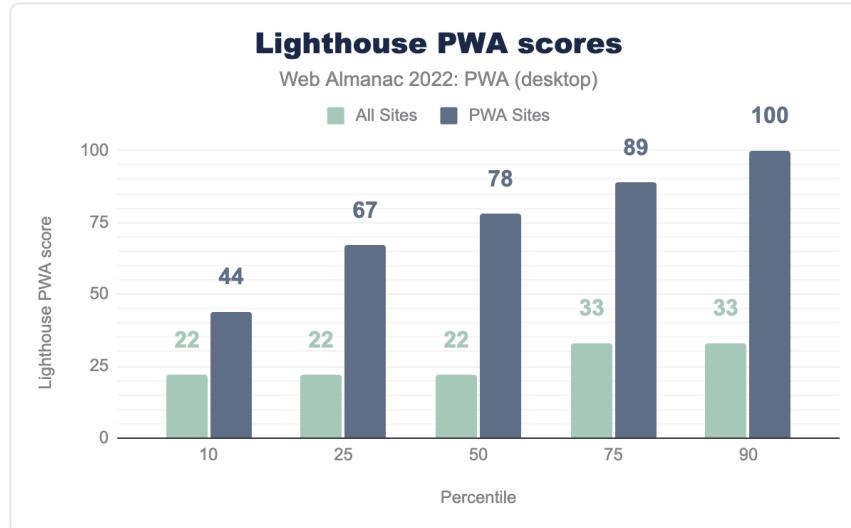


Figure 17.21. Lighthouse scores for desktop.

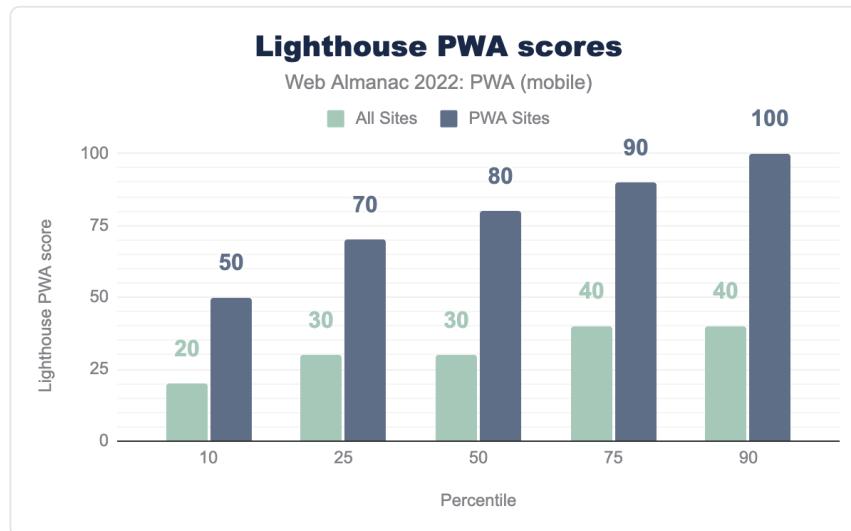


Figure 17.22. Lighthouse scores for mobile.

As expected, PWA sites tend to have considerably higher PWA audit scores. These audits look into speed, reliability, installability and other PWA requirements, as detailed in their documentation⁶⁵⁶.

^{656.} <https://developer.chrome.com/docs/lighthouse/pwa/>

What's also notable is the range of audit scores in PWA sites (50-100) representing the difference in PWAs out there. In contrast the rest of the web has a fairly consistent range of scores (20-40) reflecting the two main audits relevant for most sites discussed previously-viewport and icons.

Service worker libraries

Service workers are really powerful tools, their API allows developers to create app experiences that were impossible before, like creating their own offline experience or caching assets to improve performance, however, creating code that handles the relationship between your web app and the network comes with complexities and caveats. Here is where libraries can make life better for developers by providing higher level abstractions around the Service Worker API.

Workbox usage

Workbox⁶⁵⁷ is a set of libraries that was born to ease the usage of service workers for developers. It includes a set of libraries that go from the basics that are reused in other Workbox libraries with workbox-core⁶⁵⁸ to more specific tasks like caching strategies⁶⁵⁹, background sync⁶⁶⁰, precaching⁶⁶¹ and many more⁶⁶².

657. <https://developer.chrome.com/workbox/>
658. <https://developer.chrome.com/docs/workbox/modules/workbox-core/>
659. <https://developer.chrome.com/docs/workbox/modules/workbox-strategies/>
660. <https://developer.chrome.com/docs/workbox/modules/workbox-background-sync/>
661. <https://developer.chrome.com/docs/workbox/modules/workbox-precaching/>
662. <https://developer.chrome.com/docs/workbox/modules/>

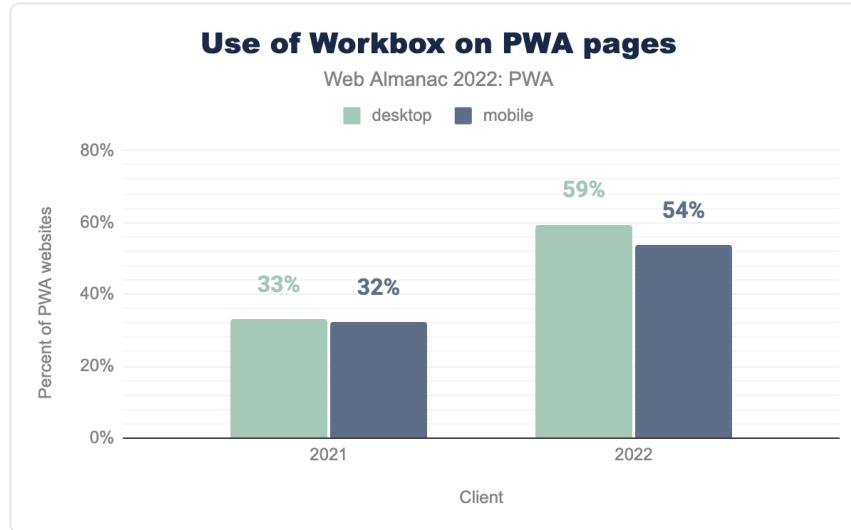


Figure 17.23. Use of Workbox on PWA pages.

Compared with last year we see a big spike in Workbox usage. Last year its usage on mobile was 33% compared to 54% this year, and almost 60% of desktop PWAs use Workbox in some capacity.

Since the growth that we saw in the number of pages controlled by service workers was not in the top 1,000 websites but in more granular categories, and this growth on Workbox usage we can infer that adoption of Workbox is happening inside the companies and websites that might have waited for the technology to be adopted by the top websites, or that might not have the need for a completely custom implementation of service workers and get the most out of Workbox's tested patterns.

Workbox packages

Workbox is structured in a way that developers can choose which parts to add to their projects depending on their site's needs. The usage shown below helps us document which PWA features are developers implementing at the moment.

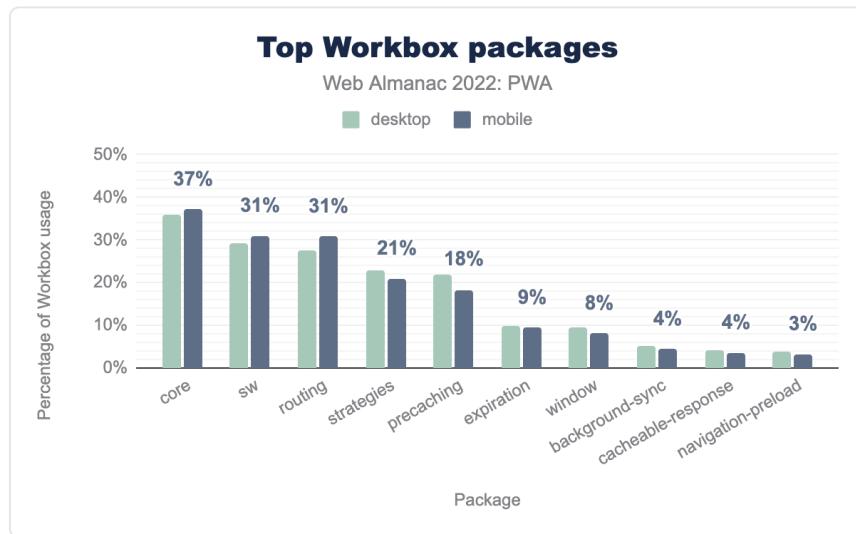


Figure 17.24. Top Workbox packages.

Here we also see an overall increase in usage, `workbox-core`. The base library saw an increase of 14% in usage. `workbox-core`, together with `workbox-routing` and `workbox-strategies`, is used to create a caching strategy that works to serve different artifacts in their web app to improve performance. It makes sense they are all at the top as they enable a core benefit of service workers.

There is also a considerable jump in usage on `workbox-precaching`. Pre-caching can be used to emulate the model that packaged apps use. With `workbox-precaching`, you can choose assets that will be cached at the time of service worker installation to make those assets load faster in subsequent visits.

What is surprising is the rise in `workbox-sw` usage, because starting with Workbox 5⁶⁶³, the Workbox team has encouraged developers to create custom bundles of the Workbox runtime instead of using `importScripts()` to load `workbox-sw` (the runtime). The Workbox team will continue supporting `workbox-sw`, but the new technique is now the recommended approach. In fact, the defaults for the build tools have switched to prefer that method.

It is possible the increase is coming from libraries that use older versions of Workbox like `create-react-app` version 3

⁶⁶³ <https://github.com/GoogleChrome/workbox/releases/tag/v5.0.0>

Web Push Notifications

Notifications are a powerful way to re-engage with users. It is also one characteristic that we expect from platform-specific applications. Notifications are the perfect way to give timely, relevant and precise information, and it is powered by the Web Push API.

Web Push notification acceptance rates

We can acknowledge that the implementation for web notifications has not been the smoothest for developers or users, but it is important to also note how useful of a tool they are. Like calendar notifications, subscription updates, or games, the important thing is that users get to choose when to turn them on.

It bears repeating that for a notification to be useful it has to be timely, precise, and relevant⁶⁶⁴. At the moment of showing the prompt to request permission, the user needs to understand the value of the service. Developers have the chance to onboard the users into notifications before they show the browser permissions dialog by sharing the advantages the users will get your specific notifications.

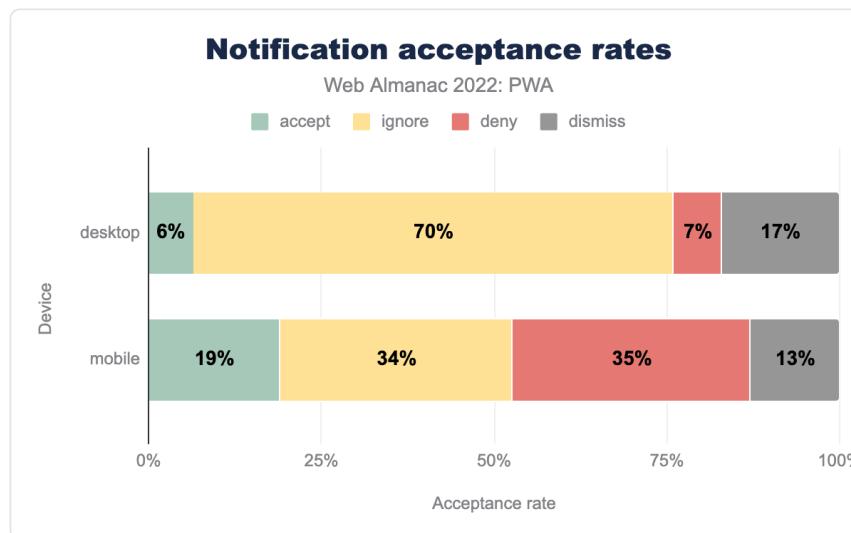


Figure 17.25. Notification acceptance rates.

With the growth of notification support and the UX improvements in different platforms, there hasn't been any major changes in the acceptance of notifications. Since early 2020 they have

⁶⁶⁴ <https://developers.google.com/web/fundamentals/push-notifications>

been around the 6% acceptance rate on desktop and 20% on mobile.

Desktop and mobile notification acceptance rate share a common fashion, and it is the trend to ignore notifications. The sum of “ignore” has gone up from 48% in February 2020 all the way up to 70% in June 2022. For mobile platforms, from 1.88% in February 2020 to a staggering 34% for June this year. Notification fatigue, coupled with increasing number of prompts for security, privacy, and advanced capabilities are partially responsible, and work is being carried out to address this and present better unified UX across different platforms.

Conclusion

2022 has been a stellar year for PWAs. The increasing features that allow integration of installable web applications with desktop platforms has driven adoption of the technology by big names in the industry. This past year advanced capabilities like protocol handlers, window controls overlay, run on OS login, and more have started to position PWAs as a key technology for application development. Whilst encouraging, this is not representative of the totality of the web platform. service worker usage percentage fell to around half, compared to the data from 2021, but the rise of big applications constructed using PWA technology rose.

Manifest files continue to be in a healthy state, with a slight increase over last year to a 95% on desktop. The correctness of these files is superb, but their completeness still leaves much to be desired. Currently, only around 0.8% of all websites qualify as installable. Many advanced capabilities like `shortcuts` and `share_target` are beginning to gain traction, appearing in around 5% of PWAs. Other capabilities like `protocol_handlers` and windows controls overlay are too new to have an impact on the data. This year also provides an initial snapshot for many of these Fugu APIs.

Notification fatigue is, understandably, still a factor, but users also request and appreciate legitimate notification use cases. Browser vendors are experimenting with less intrusive permission requests and web push notifications have the advantage of providing a consistent experience across platforms, giving the users the nudge they requested independently of the device they are using.

We hope this information sheds some light in your PWA journey and helps developers understand the current technology trends in API adoption.

Author



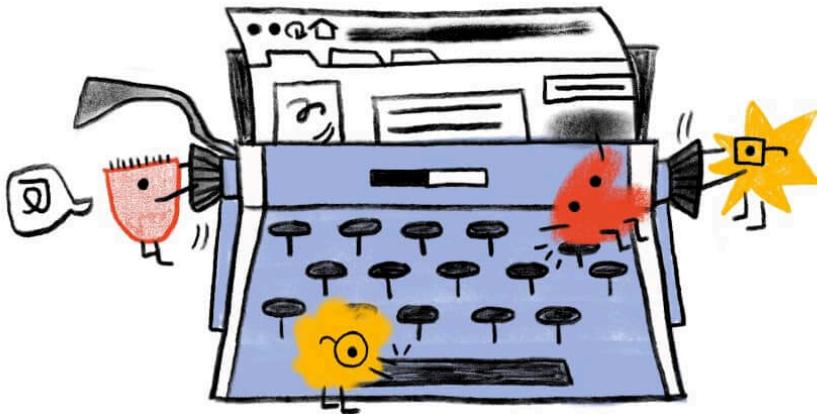
Diego Gonzalez

Twitter: @diekus GitHub: diekus Website: <https://diek.us>

Diego Gonzalez is a computer engineer from Costa Rica working as the PM for PWA platform features for the Microsoft Edge browser.

Part III Chapter 18

CMS



Written by Jonathan Wold

Reviewed by Alex Denning, Alon Kochba, and Dan Knauss

Analyzed by Colt Sliva

Edited by Dan Knauss

Introduction

In this chapter, we work to understand the current state of Content Management System (CMS) ecosystems and the growing role they play in shaping users' perception of how content can be experienced on the web. Our goal is to explore the CMS landscape in general and the characteristics of web pages created by these systems.

We believe that the CMS plays a key role in the success of our collective efforts to build a fast and resilient web. Understanding the current state, asking questions, and posing lines of inquiry for future work is our path to achieving this goal.

As a team, we've approached this year's data with curiosity, and we've combined that curiosity with personal expertise with several of the most popular CMSs. We recommend that you read our analysis in light of the variability between CMSs and types of content on them.

What is a CMS?

The term Content Management System (CMS) refers to systems enabling individuals and organizations to create, manage, and publish content. A CMS for web content, specifically, is a system aimed at creating, managing, and publishing content to be experienced on the web.

Each CMS implements a range of content management capabilities and corresponding mechanisms for users to build websites around their content. CMSs also provide administrative capabilities to facilitate the addition and management of content.

CMSs differ widely in the approaches they offer for building sites. Some provide ready-to-use templates which are supplemented with user content, and others require user involvement for designing and constructing the site structure.

In this chapter of the Web Almanac, we tried to account for all the things that form an ecosystem around a CMS platform, including hosting providers, extension developers, development agencies, site builders, etc. For this reason, when we refer to a CMS, we usually intend both the platform itself and its surrounding ecosystem.

Our dataset, based on Wappalyzer's definition⁶⁶⁵ of a CMS, identified over 270 individual CMSs. Know a CMS that's missing? Contribute to Wappalyzer⁶⁶⁶.

Some CMSs in the dataset are open source (e.g., WordPress and Joomla), and some of them are proprietary (e.g., Wix and Squarespace). Some CMSs can be used on “free” hosted or self-hosted plans, and there are also options for using these platforms on higher-tier plans up to the enterprise level.

The CMS space as a whole is a complex, federated universe of discrete but also interrelated CMS ecosystems.

CMS adoption

Our analysis throughout this work included desktop and mobile websites. The vast majority of URLs we looked at are in both datasets, but some URLs are only accessed by desktop or mobile devices. This can cause divergences in the data, so we considered desktop and mobile results separately.

665. <https://www.wappalyzer.com/technologies/cms>

666. <https://github.com/wappalyzer/wappalyzer/blob/7ac625c34432cb35d01abd683f88d3bfadca4cca/CONTRIBUTING.md>

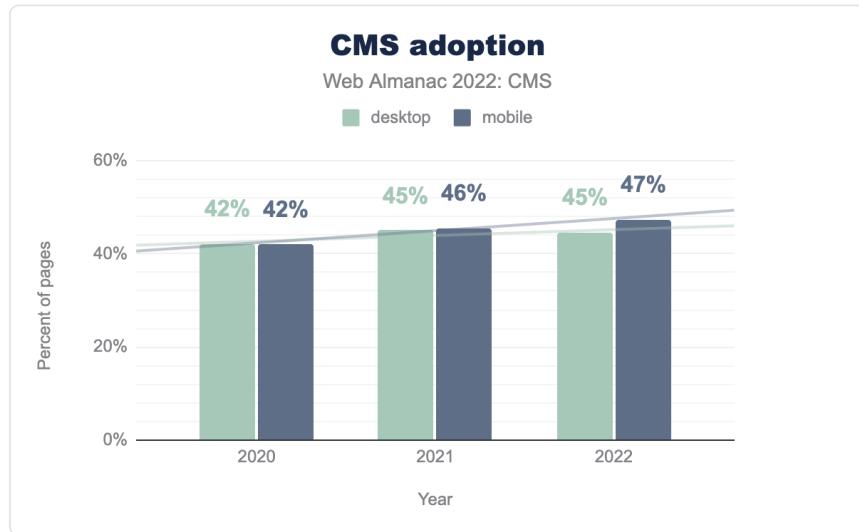


Figure 18.1. CMS adoption.

As of June 2022, 45% of the websites in the Web Almanac's desktop dataset were powered by a CMS, indicating similar usage to 2021⁶⁶⁷. The mobile dataset shows an increase from 46% in 2021⁶⁶⁸ to 47% here in 2022. Looking closer at the desktop raw figures we actually see a slight drop in both absolute and percentage terms, but the drop is more likely an artifact of minor variances in attribution than an indicator of a downward trend in CMS usage. It should be noted that the number of desktop sites tracked by HTTP Archive (and the source CrUX dataset) has fallen considerably from 6.4 million sites to 5.4 million sites, while the number of mobile sites has grown by about 400,000 sites from 7.5 million to 7.9 million sites. We take this increase to reflect continued growth in mobile device usage at the expense of the desktop.

It is instructive to compare these numbers with another commonly used dataset, such as W3Techs⁶⁶⁹. W3Techs reported that as of June 2021, 64.6% of websites are created using a CMS. This is up from 59.2% in June 2020—an increase of over 9%.

The deviation between our analysis and W3Techs' analysis can be explained by differences in research methodologies and definitions of a CMS.

W3Techs' definition is as follows: “Content Management Systems are applications for creating and managing the content of a website. We include all such systems in this category, also systems that are often classified as wikis, blog engines, discussion boards, static site generators, website editors or any type of software that provides website content.”

667. <https://almanac.httparchive.org/en/2021/cms#cms-adoption>

668. <https://almanac.httparchive.org/en/2021/cms#cms-adoption>

669. https://w3techs.com/technologies/history_overview/content_management/all/q

As we mentioned previously, Wappalyzer has a stricter definition of a CMS than we do. Wappalyzer excludes some major CMSs that appear in W3Techs reports. You can read more about our definition of a CMS on the Methodology page.

CMS adoption by geography

CMSs are used around the world, with some variance by country.

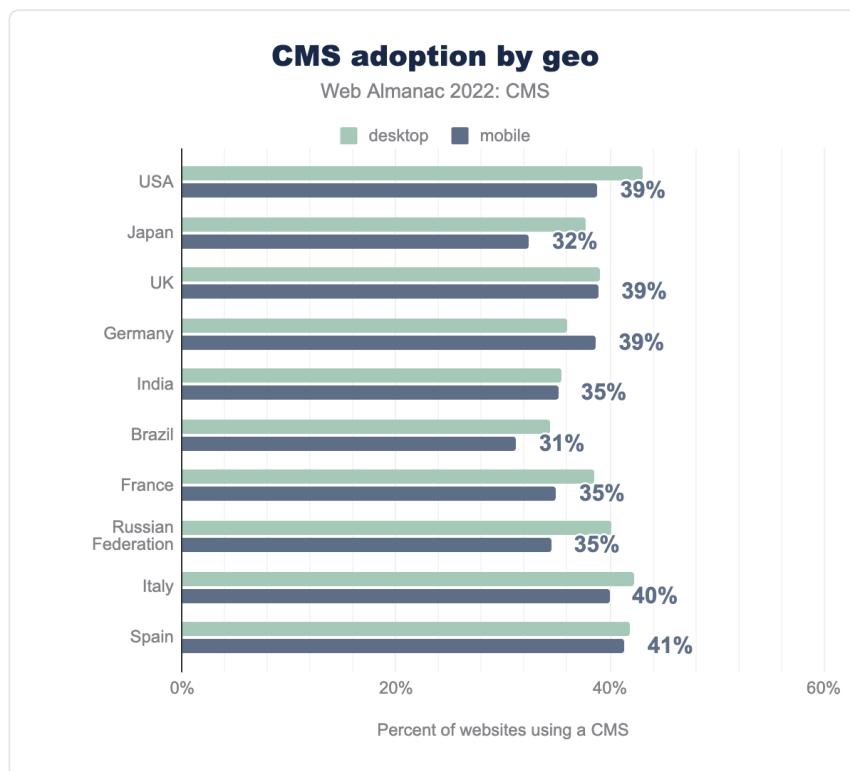


Figure 18.2. CMS adoption by geography.

Among the countries with the highest number of websites, CMS adoption is highest in Italy and Spain where 40%–41% of mobile sites are built with a CMS. Brazil and Japan have the lowest adoption with only 31% and 32% respectively.

Of particular interest is the decrease across the board compared to our 2021 dataset⁶⁷⁰ when individual countries are considered. Comparing year-over-year for mobile results, all countries

⁶⁷⁰. <https://almanac.httparchive.org/en/2021/cms#cms-adoption-by-geography>

except India appear to show a drop, ranging from a 4% decrease for the UK and Germany to an 8% decrease for the US and Italy. Given the consistency of the decreases across geographies, it seems more plausible to be a variance in attribution than a wholesale drop in CMS adoption. We recommend evaluating this further in next year's analysis.

CMS adoption by rank

We examined CMS adoption by the estimated rank of the sites included within the dataset.

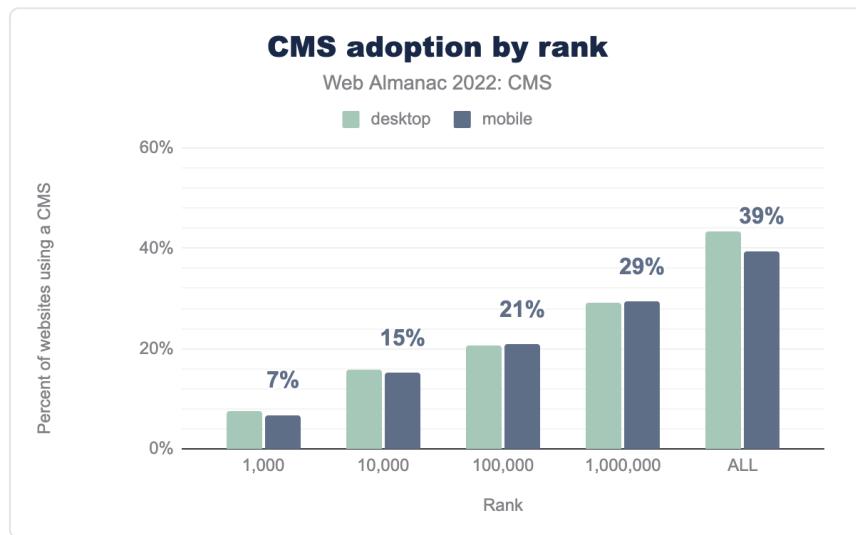


Figure 18.3. CMS adoption by rank.

According to the dataset, CMSs are used by fewer than 7% of the top 1,000 websites for both desktop and mobile even though 47% of all mobile sites in the dataset use a CMS. A possible explanation for this apparent discrepancy, and the one we offered last year, is that smaller businesses with websites tend to use a popular CMS for their ease of use, and those CMSs are easily identified. However, larger businesses with higher ranked websites tend to have custom-built CMS solutions that we can't identify.

Another explanation is that higher ranking sites with more resources allocated to their development are more likely to obfuscate the identity of their CMS for security reasons. It is improbable that more than 90% of the top 1,000 would forgo a CMS entirely and more likely that they just don't show up in our dataset.

A potentially correlated trend is the adoption of "headless" CMSs and the move to separate content—and the CMS that powers it—from the frontend experience offered to end-users.

While our confidence in the overall dataset remains high, we're interested in investigating the adoption-by-rank dataset further in future editions of this report to see if more can be done to detect and identify a greater number of CMSs to improve the overall accuracy of our results.

Most popular CMSs

Among all websites that use an identifiable CMS, WordPress sites account for the majority of the relative market share—with over 35% adoption on mobile—followed by Wix (2%), Joomla (1.8%), Drupal (1.6%), and Squarespace (1.0%).

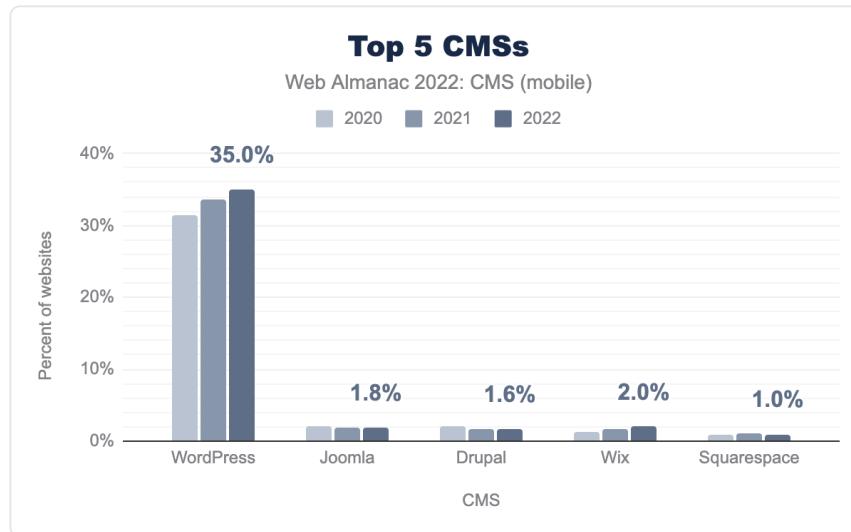


Figure 18.4. Top five CMSs year-over-year.

Comparing year-over-year, Drupal and Joomla continue to decline in market share, while Squarespace remains steady and Wix grows. WordPress continues its ascent, increasing 1.4% over 2021 on mobile, and 0.2% over 2021 on desktop.

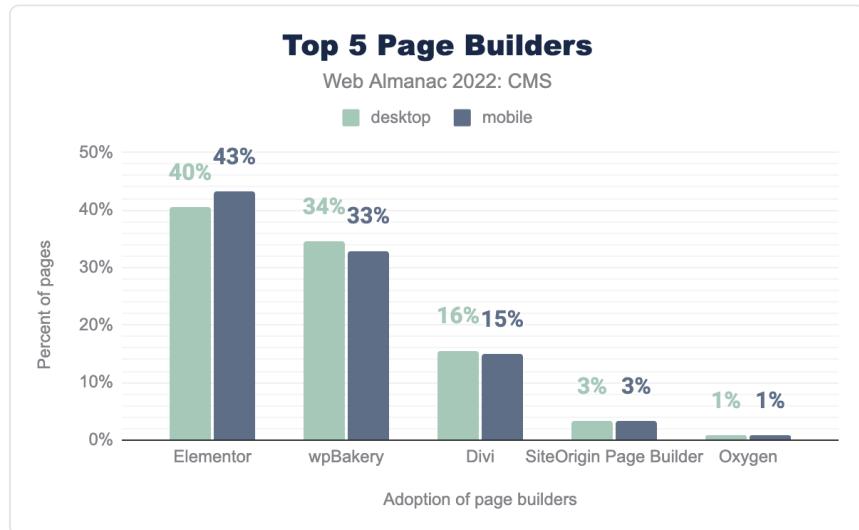


Figure 18.5. WordPress page builder adoption.

Within WordPress, users often make use of a “page builder” that provides an interface for content management. This year, with Wappalyzer’s detection methods improving, we looked at page builder adoption. We discovered that of the WordPress sites attributed to a page builder (approximately 34% of all WordPress sites in our dataset), Elementor and WP Bakery are the clear winners, with Divi, SiteOrigin, and Oxygen trailing behind.

As we see it today, page builders exert significant influence on the performance of a site. Historically, page builders have been anecdotal indicators of poor performance. As one example, our dataset indicates that it’s not uncommon for websites to have multiple page builders installed, adding a significant increase to the resources loaded by a site.

Now that we’re tracking page builder data, we’ll have the opportunity in future editions to evaluate year-over-year changes in page builder adoption and look for correlations in those changes to the overall performance of WordPress as a CMS.

CMS user experience

An important feature of CMSs is the user experience they provide for users visiting sites built on these platforms. We attempt to examine these experiences through Real User Measurements (RUM) via the Chrome User Experience Report⁶⁷¹ (CrUX), and synthetic testing

⁶⁷¹ <https://developers.google.com/web/tools/chrome-user-experience-report>

using Lighthouse.

Core Web Vitals

The Core Web Vitals Technology Report⁶⁷² can be used to drill down into the available data and view the progress of evaluated platforms updated on a monthly basis.

In this section we focus on data from June 2022 to provide a consistent timeframe for data presented across the Web Almanac. We examine three important metrics provided by the Chrome User Experience Report⁶⁷³ which can shed light on our understanding of how users are experiencing CMS-powered web pages in the wild:

- Largest Contentful Paint⁶⁷⁴ (LCP)
- First Input Delay⁶⁷⁵ (FID)
- Cumulative Layout Shift⁶⁷⁶ (CLS)

These metrics aim to cover the technical fundamentals of a great web user experience. The Performance chapter covers these metrics in greater detail, but here we are interested in looking at them specifically in terms of CMSs.

Initially, let's review the 10 CMS platforms with the highest number of origins and examine the percentage of sites on each platform that have a "passing" grade. A passing grade means that each of the above metrics must be in the "good" (green) range for each site: an LCP of 2.5 seconds or less, a FID of 100ms or less, and a CLS of 0.1 or less.

672. <https://httparchive.org/reports/cvv-tech>

673. <https://almanac.httparchive.org/en/2021/methodology#chrome-ux-report>

674. <https://web.dev/lcp/>

675. <https://web.dev/fid/>

676. <https://web.dev/cls/>

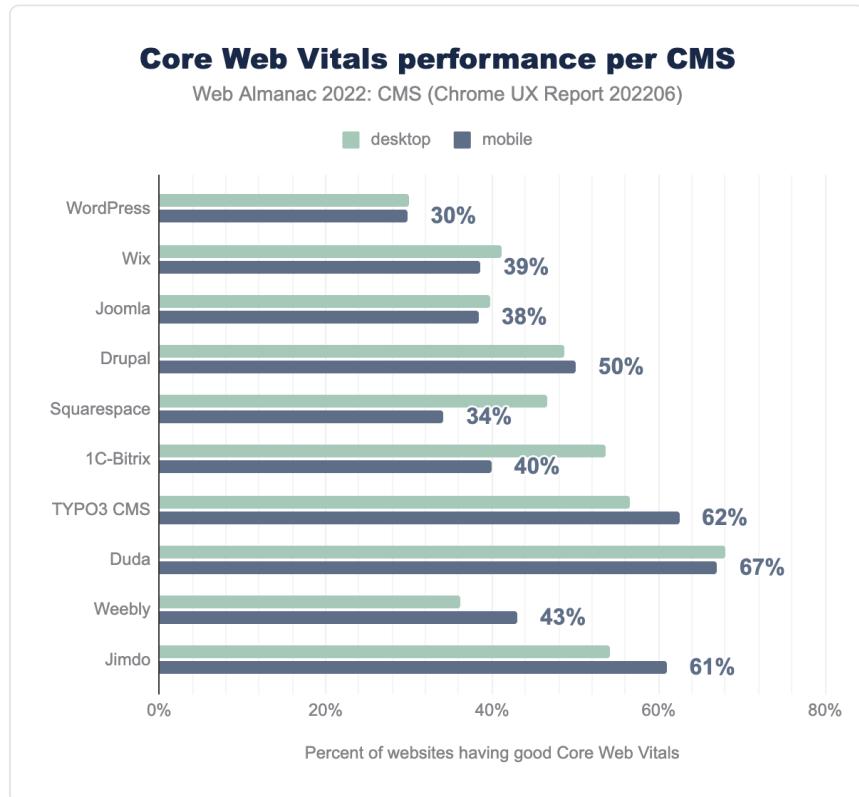


Figure 18.6. Core web vitals performance by CMS.

We can see that desktop visitors generally score better than mobile. This can be explained by resource limitations on mobile devices and poorer connections. The large difference between mobile and desktop performance on some platforms also suggests that very different pages are served to users depending on the device they use.

In June, for mobile devices, Duda had the largest percentage of passing sites, with 67% of mobile sites passing all three CWVs. WordPress trailed farthest behind, with 30% of its sites passing. Nevertheless, this indicates a significant increase over our 2021 data, where only 19% of WordPress sites passed.

Desktop device experience was better for most CMSs. Duda had the largest CWV passing rate at 68%. WordPress had the lowest ratio of passing sites: 30%.

We can also evaluate the progress of these CMS platforms' performance on mobile devices by looking at last year's data:

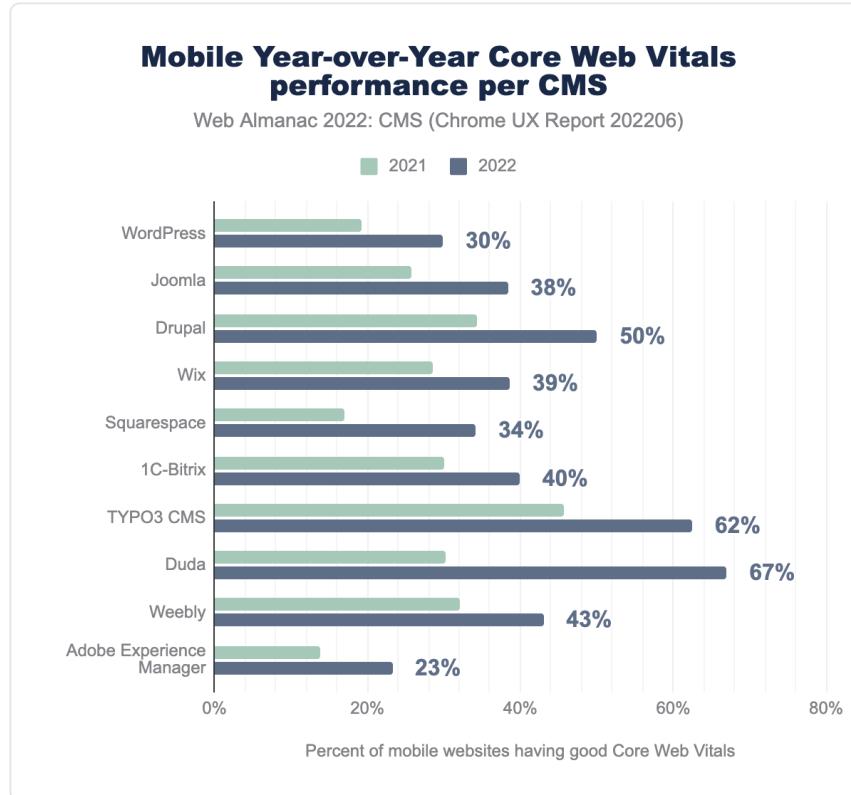


Figure 18.7. Core web vitals mobile year-over-year.

All of these CMSs showed an improvement in the percentage of origins with good CWVs since June 2021.

Let's drill into the three Core Web Vitals to see where each platform has room to improve and which metrics improved the most since last year:

Largest Contentful Paint (LCP)

Largest Contentful Paint (LCP) measures the point in time when the page's main content has likely loaded and thus the page is useful to the user. LCP is assessed by measuring the render time of the largest image or text block visible within the viewport.

A "good" LCP is regarded as being under 2.5 seconds.

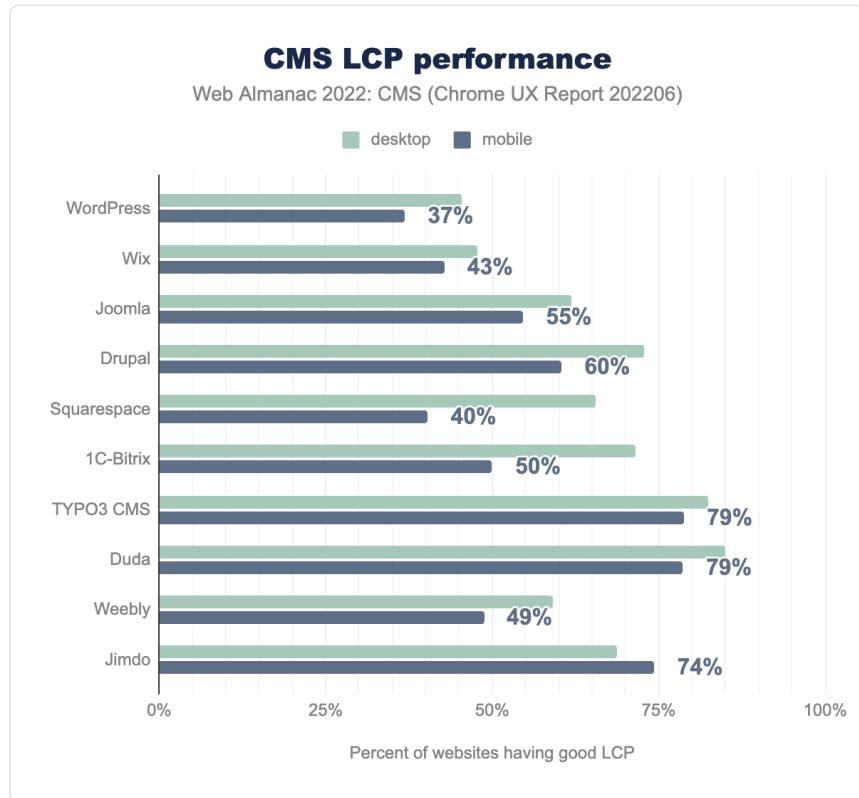


Figure 18.8. Percentage of sites with good LCP by CMS.

TYPO3 and Duda had the best LCP scores with 79% of origins having a “good” LCP experience. WordPress and Squarespace have the worst LCP scores with 37% and 40% of origins having good LCP scores, respectively.

Mobile year-over-year CMS LCP performance

Web Almanac 2022: CMS (Chrome UX Report 202206)

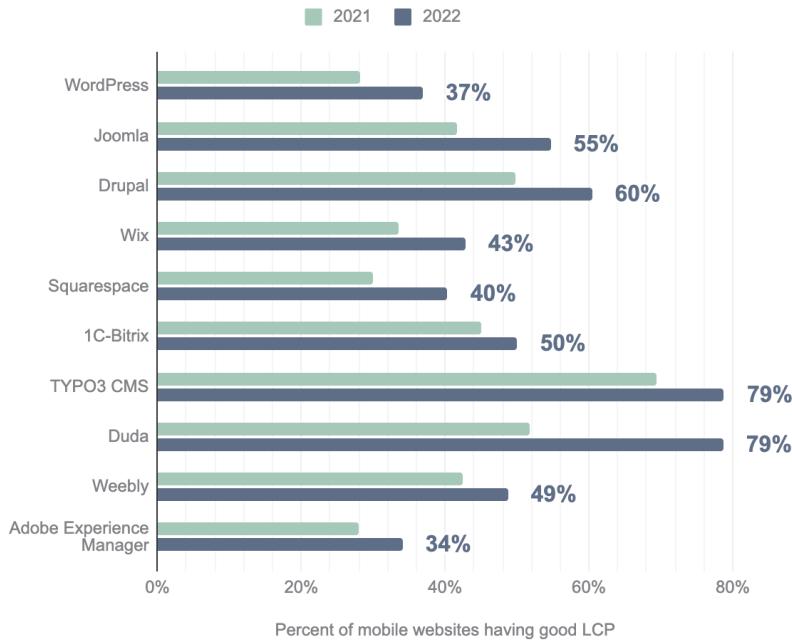


Figure 18.9. LCP mobile year-over-year.

Compared to the 2021 dataset, all CMSs showed improvements in LCP. Joomla improved by 13%. Drupal, Squarespace, and TYPO3 improved by 10%. WordPress improved by 9%.

These improvements are a positive sign even though the numbers still are low for most CMSs. The difficulty in achieving a good LCP score probably relates to the fact that the LCP is dependent on the download of image/font/CSS and then displaying the appropriate HTML elements. Achieving this in under 2.5 seconds for all device types and connection speeds can be challenging. Improving LCP scores usually involves the correct use of caching, pre-loading, resource prioritization, and lazy loading of other competing resources.

First Input Delay (FID)

First Input Delay (FID) measures the time from when a user first interacts with the page (i.e., when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time

when the browser is able to process that interaction. A “fast” FID from a user’s perspective would be almost immediate feedback from their actions rather than a stalled experience.

Any delay is a pain point and could correlate with interference from other parts of the site loading while the user tries to interact with the site. A “good” FID is regarded as being under 100 milliseconds.

In 2021’s report, the fact that almost all platforms manage to deliver a good FID raised questions about the strictness of this metric. The Chrome team published an article⁶⁷⁷ that was updated in May of 2022 to include a reference to a new metric, Interaction to Next Paint (INP)⁶⁷⁸. Given its beta nature at the time of this writing, we’re limiting its inclusion to this reference in anticipation of a possible expansion in next year’s report.

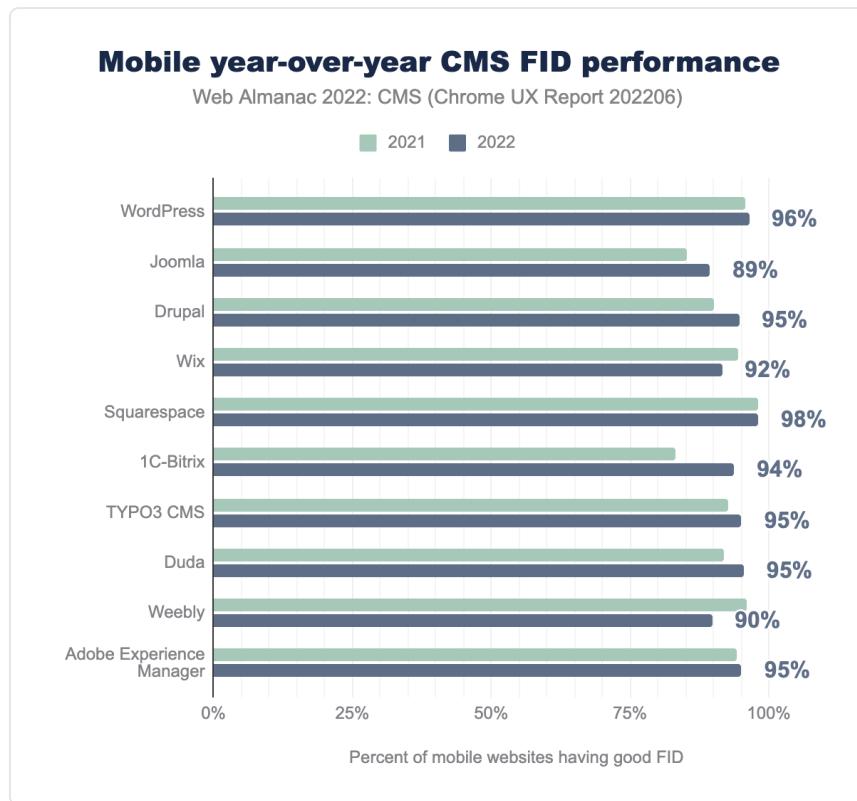


Figure 18.10. FID mobile year-over-year.

677. <https://web.dev/responsiveness/>

678. <https://web.dev/inp/>

Yearly data shows that most CMSs managed to improve their FID over the past year. Wix and Weebly both regressed by several percentage points over the previous year's data.

Cumulative Layout Shift (CLS)

Cumulative Layout Shift (CLS) measures the visual stability of content on a web page, measuring the largest burst of layout shift scores for every unexpected layout shift that occurs during the entire lifespan of a page that was not caused by direct user interactions.

A layout shift occurs any time a visible element changes its position from one rendered frame to the next. The CLS metric evolved in 2021⁶⁷⁹, mainly introducing the concept of Session Windows, to be fairer to long-lived pages and Single Page Apps (SPAs).

A score of 0.1 or below is measured as “good,” over 0.25 as “poor,” and anything in between “needs improvement.”

679. <https://web.dev/evolving-cls/>

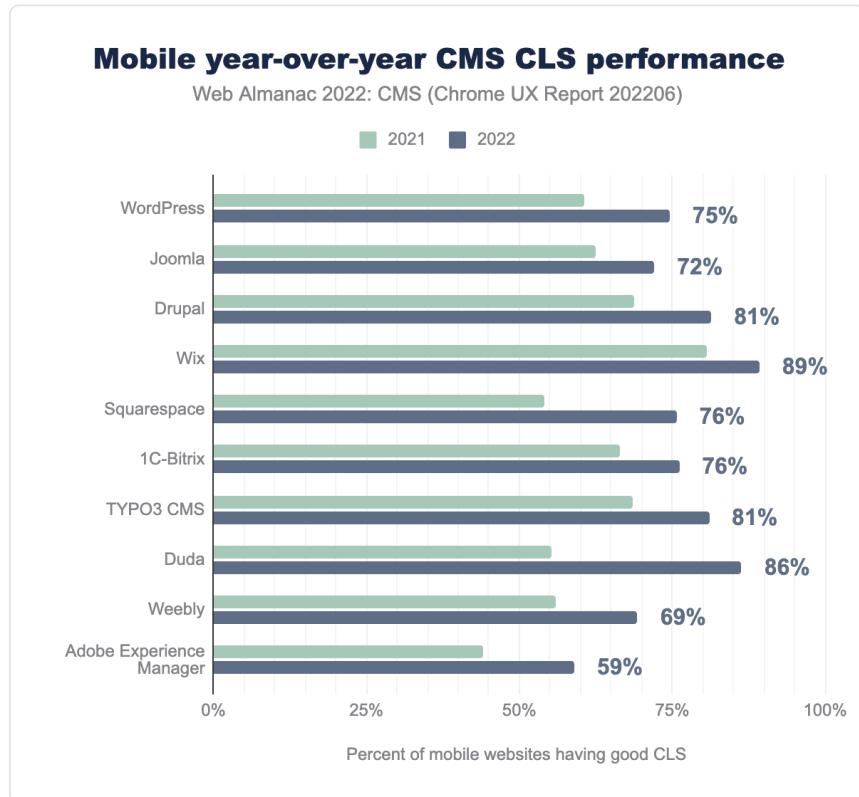


Figure 18.11. CLS mobile year-over-year.

Comparing yearly data, we can see that all CMSs made progress. WordPress, Squarespace, Duda, and Adobe Experience Manager in particular show significant gains.

Lighthouse

Lighthouse⁶⁸⁰ is an open-source, automated tool for improving the quality of web pages. One key aspect of the tool is that it provides a set of audits to assess the status of a website in terms of performance, accessibility, SEO, best practices, and more. Lighthouse reports provide lab data, a way developers can get suggestions on how to improve website performance, but the Lighthouse score has no direct implications on the actual field data collected by CrUX⁶⁸¹. You can read more on Lighthouse and the correlation between its lab scores and field data⁶⁸².

680. <https://developers.google.com/web/tools/lighthouse/>

681. <https://developers.google.com/web/tools/chrome-user-experience-report>

682. <https://web.dev/lab-and-field-data-differences/>

HTTP Archive runs Lighthouse on its mobile web pages, which are also throttled to emulate a slow 4G connection with a CPU slowdown, and also this year they started running on Desktop as well.

We can analyze this data to provide another perspective on CMS performance, using the results of these synthetic tests, which also include metrics that are not tracked in CrUX.

Performance score

The Lighthouse performance score⁶⁸³ is a weighted average of several scored metrics.

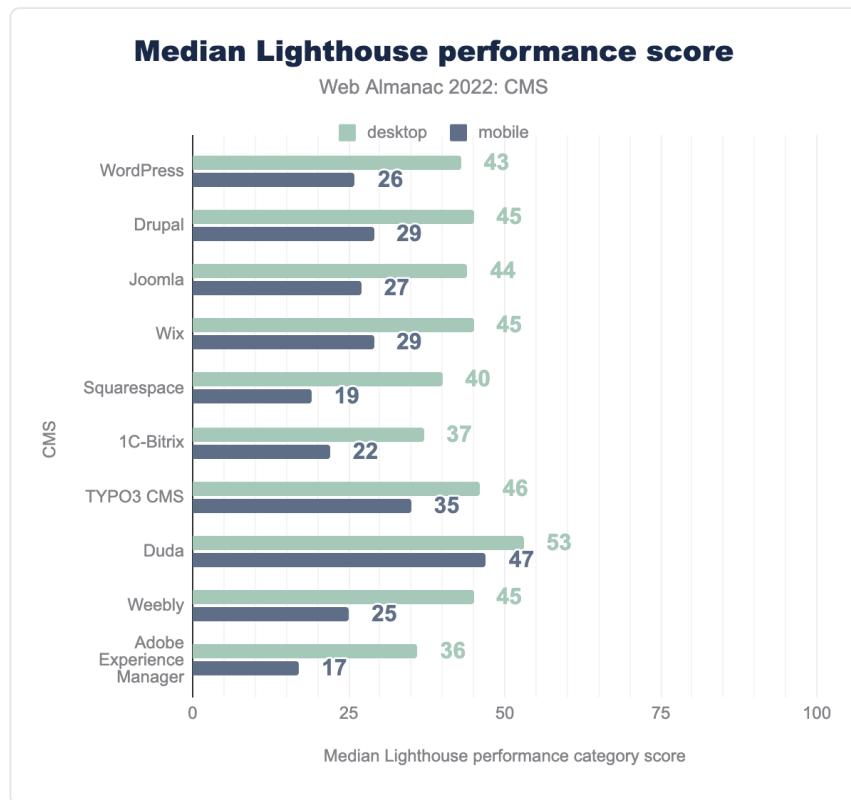


Figure 18.12. Median lighthouse performance scores.

We can see that the median performance scores for most platforms on mobile are low and range from about 19 to 35. Duda at 47 is the exception. As Philip Walton noted in 2021, this

^{683.} <https://web.dev/performance-scoring/>

does not directly imply bad results⁶⁸⁴ in mobile field data, but it does imply that all platforms have room for improvement, especially on low-end devices with network connections similar to those Lighthouse attempts to emulate.

WordPress, Joomla, Drupal, and 1C-Bitrix showed no change from last year's results. Wix dropped from 30% to 29% while the rest showed improvement.

Desktop scores were good across the board with all CMSs seeing 10-20 point improvements. This isn't surprising, given the faster CPUs and networks available to the desktop.

SEO score

Search Engine Optimization (or SEO) is the practice of improving a website to make it more easily found in search engines. This is covered more in-depth in our SEO chapter, but it relates to CMSs as well. A CMS and content on it is generally set up to serve as much information to search engine crawlers as possible to make it as easy as possible for them to index site content appropriately in search engine results. Compared to a custom-built website, one might expect a CMS to provide good SEO capabilities, and the Lighthouse scores in this category are appropriately high.

684. <https://philipwalton.com/articles/my-challenge-to-the-web-performance-community/>

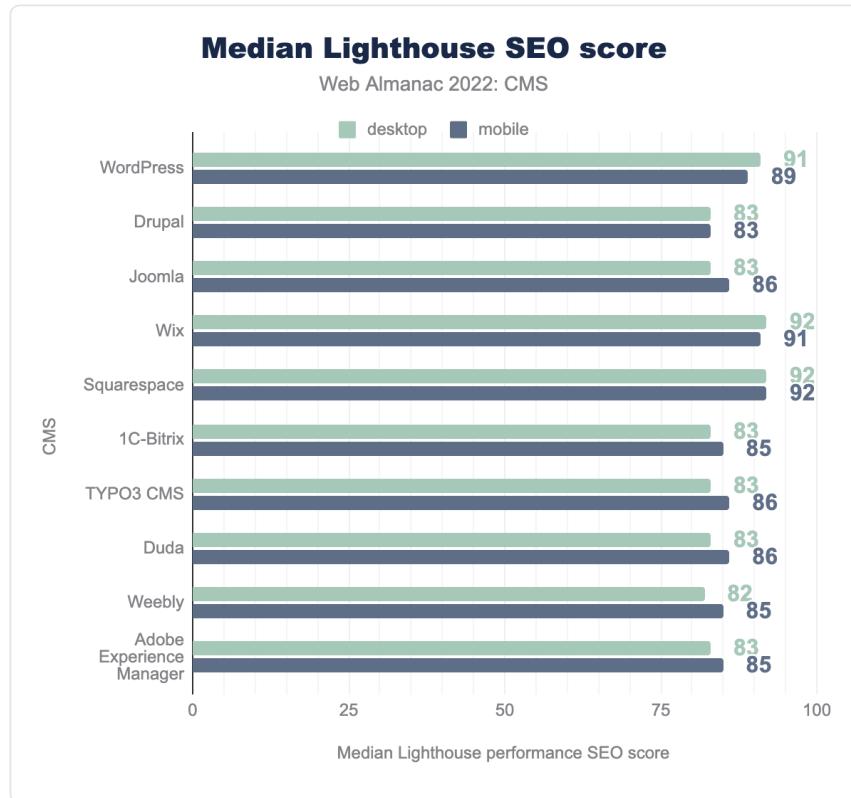


Figure 18.13. Median lighthouse SEO scores.

The median SEO scores in all of the top 10 platforms range from 83-92, a slight reduction from 84-95 in 2021⁶⁸⁵. Desktop scores are similar—slightly better in some cases, slightly worse in others.

Accessibility score

An accessible website is a site designed and developed so that people with disabilities can use them. Web accessibility also benefits people without disabilities, such as those on slow internet connections. Read more in our Accessibility chapter.

Lighthouse provides a set of accessibility audits, and it returns a weighted average of all of them. See Scoring Details⁶⁸⁶ for a full list of how each audit is weighted.

685. <https://almanac.httparchive.org/en/2021/cms#seo-score>

686. <https://web.dev/accessibility-scoring/>

Each accessibility audit is either a pass or a fail, but unlike other Lighthouse audits, a page doesn't get points for partially passing an accessibility audit. For example, if some elements have screen reader-friendly names, but others don't, that page gets a zero for the screen reader-friendly-names audit.

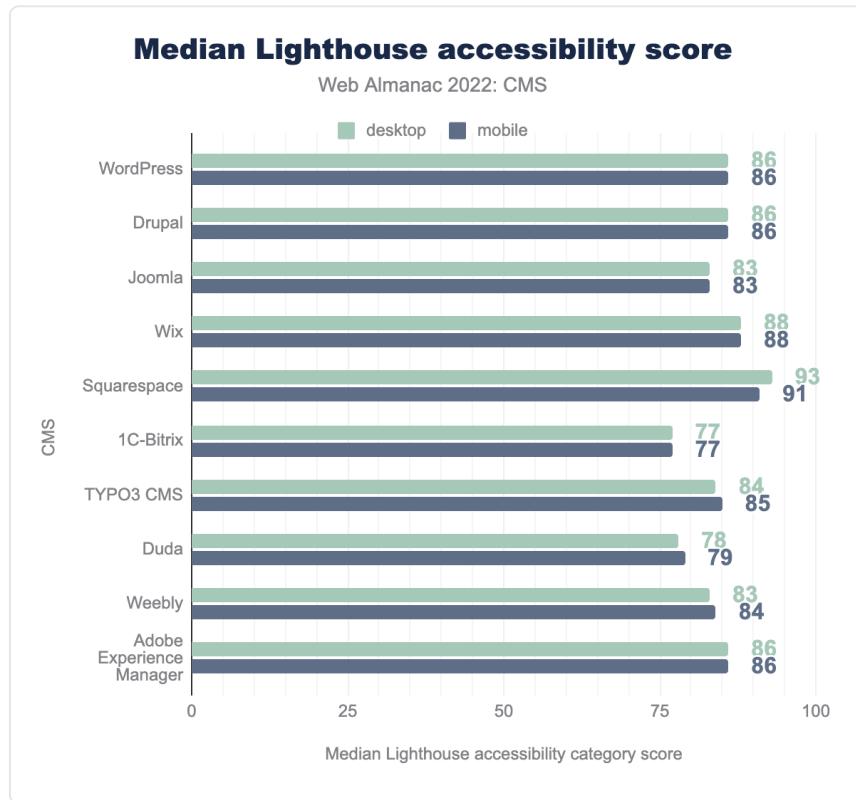


Figure 18.14. Media Lighthouse accessibility scores.

The median Lighthouse accessibility score for the top 10 CMSs ranges between 77 and 91. Squarespace had the highest score of 91, while 1C-Bitrix had the lowest accessibility scores. The desktop scores are almost identical to mobile, perhaps reflecting that the same sites are delivered to both desktop and mobile devices.

Best practices

The Lighthouse best practices⁶⁸⁷ try to ensure that web pages are following best practices for

⁶⁸⁷. <https://web.dev/lighthouse-best-practices/>

the web for a variety of different metrics such as supporting HTTPS, no errors logged in the console, and more.

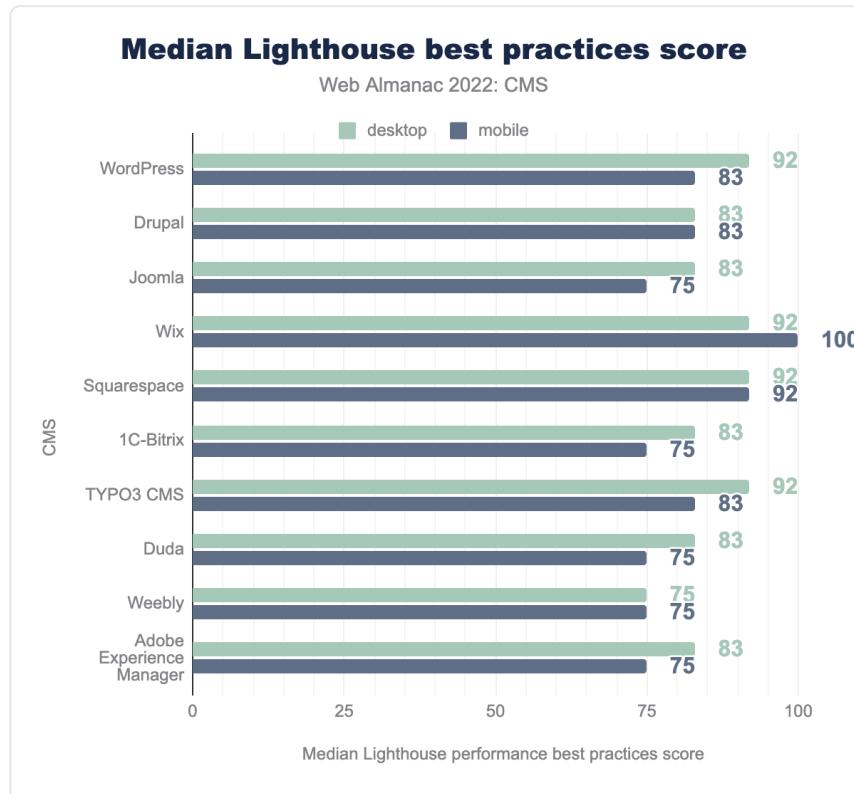


Figure 18.15. Media Lighthouse best practices scores.

Wix had the highest median best practices score of 100, while many of the other top 10 platforms share the lowest score of 75. Again, desktop results are very similar though larger in some cases. This may reflect incorrect image aspect ratios on mobile pages since most of the other audits in this category are platform-based.

Resource weights

We also used HTTP Archive data to analyze the weight of resources used across different platforms. We did this to highlight possible opportunities for performance improvement. Page loading performance does not depend exclusively on the number of downloaded bytes, but fewer bytes necessary to load a page results in reduced costs, fewer carbon emissions, and

potentially faster performance—especially for slower connections.

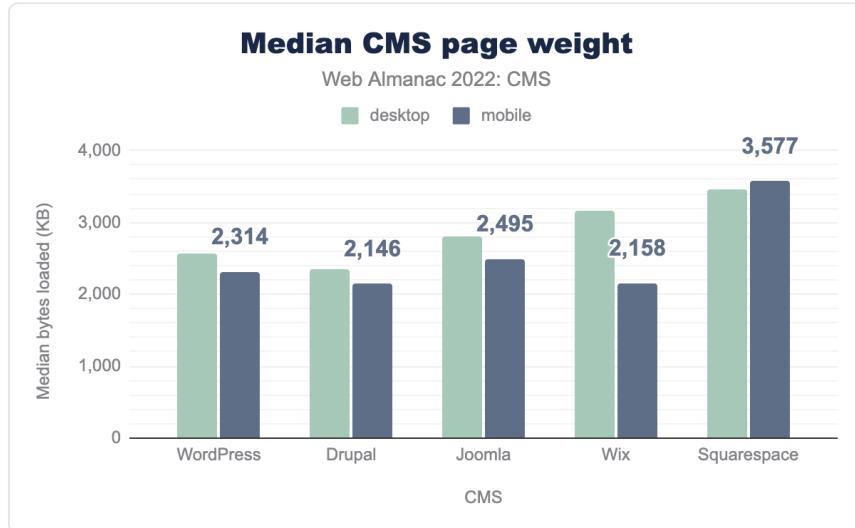


Figure 18.16. Media resource weights by CMS.

Most of the top five CMSs deliver a median page weight of around ~2 MB, except Squarespace which delivers a larger ~3.5 MB. All except Joomla showed increases in page weight over 2021 data⁶⁸⁸.

688. <https://almanac.httparchive.org/en/2021/cms#page-weight-breakdown>

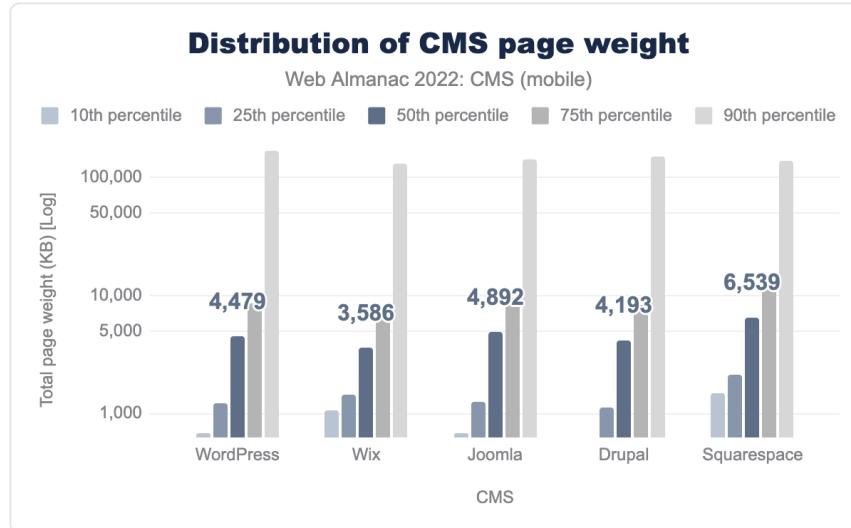


Figure 18.17. Mobile page weight distribution by CMS.

The distribution of page weight in each platform's percentiles is substantial. Page weight is probably related to the differences in user content across web pages, the number of images used, plugins installed, etc. The smallest pages delivered per platform come from WordPress, a marked improvement over last year's data. This year, WordPress only sends 673 KB for their 10th percentile of visits. The largest pages come from Squarespace, with ~11.4 MB delivered for their 90th percentile of visits, a ~2 MB increase over last year's data.

Page weight breakdown

Page weight is the sum in kilobytes of resources used on a page. We can attempt to evaluate these different resource sizes across different CMSs.

Images

Images, which are usually the heaviest resource loaded on a web page, account for a large portion of the resource weight.

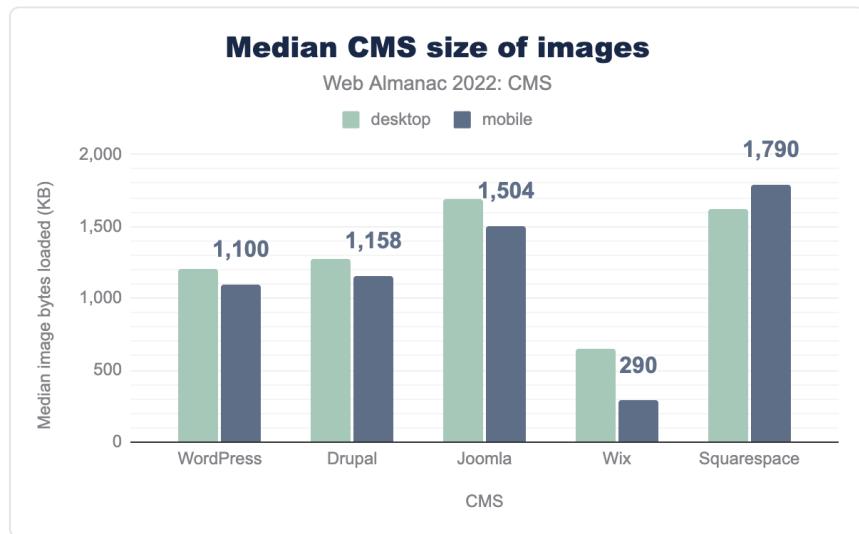


Figure 18.18. Median image size by CMS.

Wix delivers substantially fewer image bytes, with only 290 KB delivered for the median of mobile views. This suggests good use of image compression and lazy image loading. All of the other top five platforms deliver over 1 MB of images, with Squarespace delivering the largest ~1.7 MB.

Advanced image formats provide a considerable improvement in compression, enabling resource savings and faster site loading. WebP is commonly supported in all major browsers today, with over 95% support⁶⁸⁹. In addition, there are several newer image formats gaining popularity and adoption, namely AVIF⁶⁹⁰, and JPEG-XL⁶⁹¹.

We can examine the usage of the different image formats across the top CMSs:

689. <https://caniuse.com/webp>

690. <https://caniuse.com/avif>

691. <https://jpegxl.info>

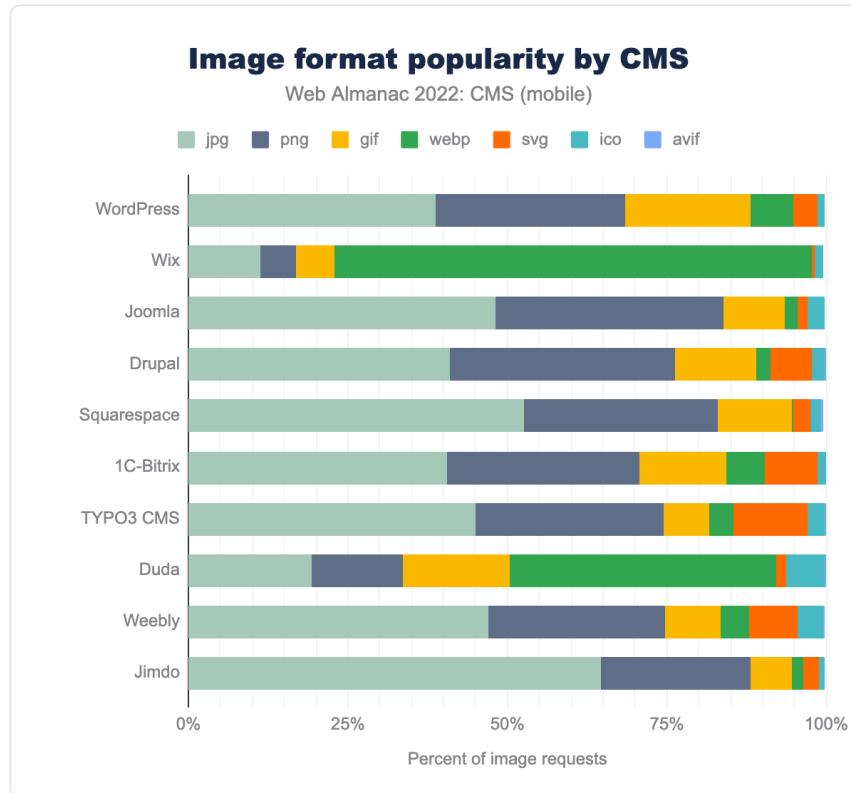


Figure 18.19. Image format popularity by CMS.

Wix and Duda make the most use of WebP, at ~75% and 42% adoption respectively, while the rest show minor increases.

With the growing support of WebP⁶⁹², it seems all platforms have work to do to reduce the usage of the older JPEG and PNG formats without compromising on image quality.

WordPress introduced support for WebP in WordPress 5.8, which was released in June of 2021. WebP support was planned to be included by default⁶⁹³ in WordPress 6.1. However, this decision has been delayed. Eventually, we expect a significant increase in WebP adoption via WordPress which may be apparent in the 2023 results.

692. <https://caniuse.com/webp>

693. <https://make.wordpress.org/core/2022/06/30/plan-for-adding-webp-multiple-mime-support-for-images/>

JavaScript

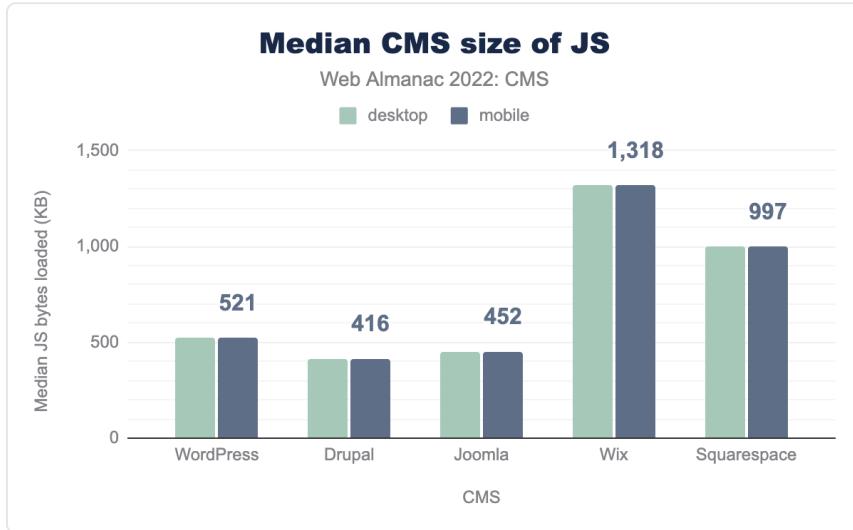


Figure 18.20. Median JavaScript resources by CMS.

The top five CMSs all deliver pages that rely on JavaScript, with Drupal delivering the fewest JavaScript bytes: 416 KB on mobile. Wix delivers the most JavaScript bytes—over 1.3 MB.

HTML document

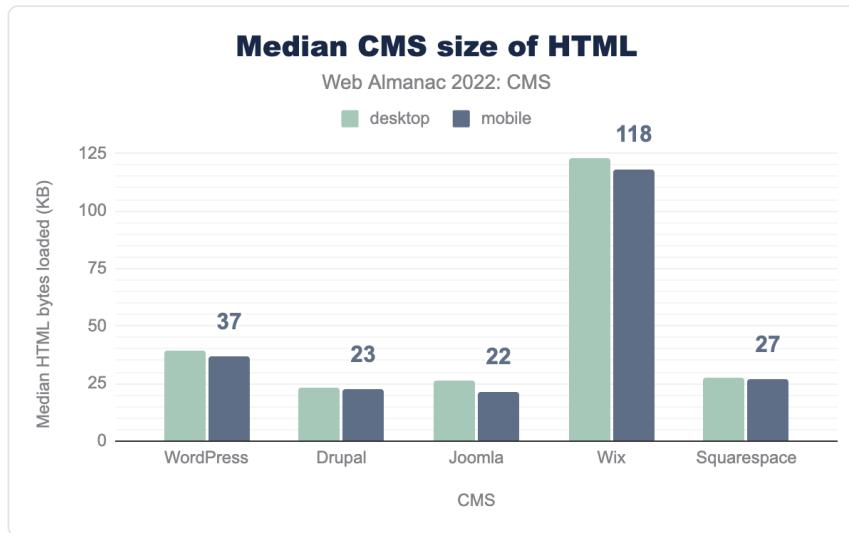


Figure 18.21. Median HTML size by CMS.

Examining the HTML document sizes, we can see that most of the top CMSs deliver a median HTML size of ~22 KB–37 KB. The only exception is Wix which delivers ~118 KB, a minor improvement over 2021's results. This may suggest extensive use of inlined resources and shows an area that can be further improved.

CSS

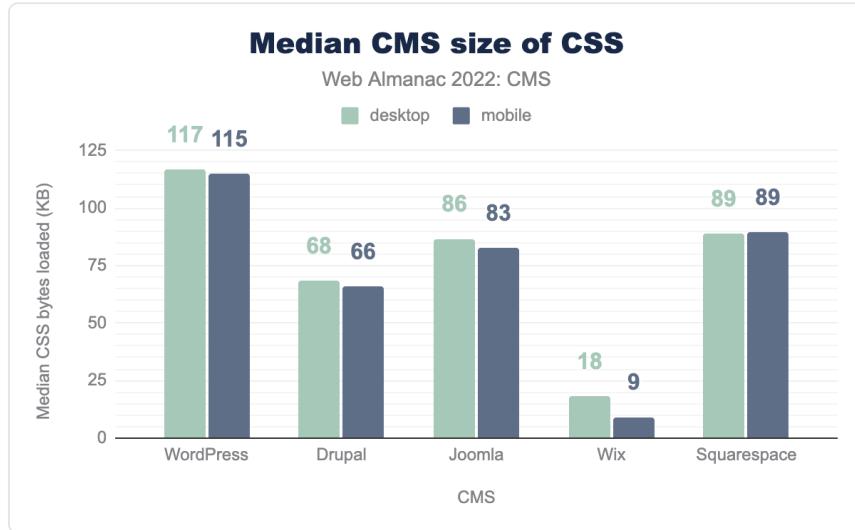


Figure 18.22. Median CSS size by CMS.

Next, we examine the use of explicit CSS resources that are downloaded. Here we can see a different distribution between platforms that strengthens the case for inlining CSS. Wix delivers the fewest CSS resources, with only ~9 KB sent on mobile views. WordPress delivers the most with ~115 KB.

Fonts

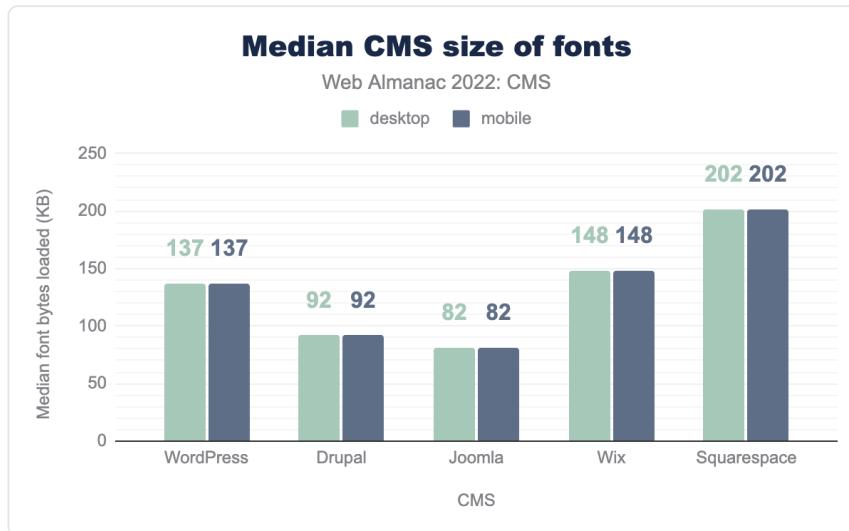


Figure 18.23. Median font size by CMS.

To display text, web developers often choose to use a variety of fonts. Joomla delivers the fewest font bytes, with 82 KB on mobile views. Squarespace delivers the most with 202 KB.

WordPress in 2022

WordPress is the most commonly used CMS today. Almost three out of four sites built with a CMS are using WordPress, which merits further discussion.

WordPress is an open-source project, which has been around since 2003. Many sites built on WordPress use a variety of themes and plugins, sometimes through page builders such as Elementor, WP Bakery, or Divi.

The WordPress ecosystem maintains the CMS and services required for additional functionality through custom services and products (themes and plugins). This community has an outsized impact, with a relatively small number of people maintaining both the CMS itself and providing the additional functionality which makes WordPress sufficiently powerful and flexible that it can serve most kinds of websites. This flexibility is important when explaining the market share, but it also complicates the discussion around WordPress-based site performance.

In 2021, contributors from the WordPress community acknowledged the current state of performance, in this proposal⁶⁹⁴ to create a performance-dedicated core team, which we hope will improve the performance of the average WordPress site.

This year, we compared our results against last year, focusing on adoption by geography and passing Core Web Vitals by geography along with a look at average resource usage.

Adoption by geography

First, we examined WordPress adoption by geography across all sites in our dataset in comparison to our 2021 results.

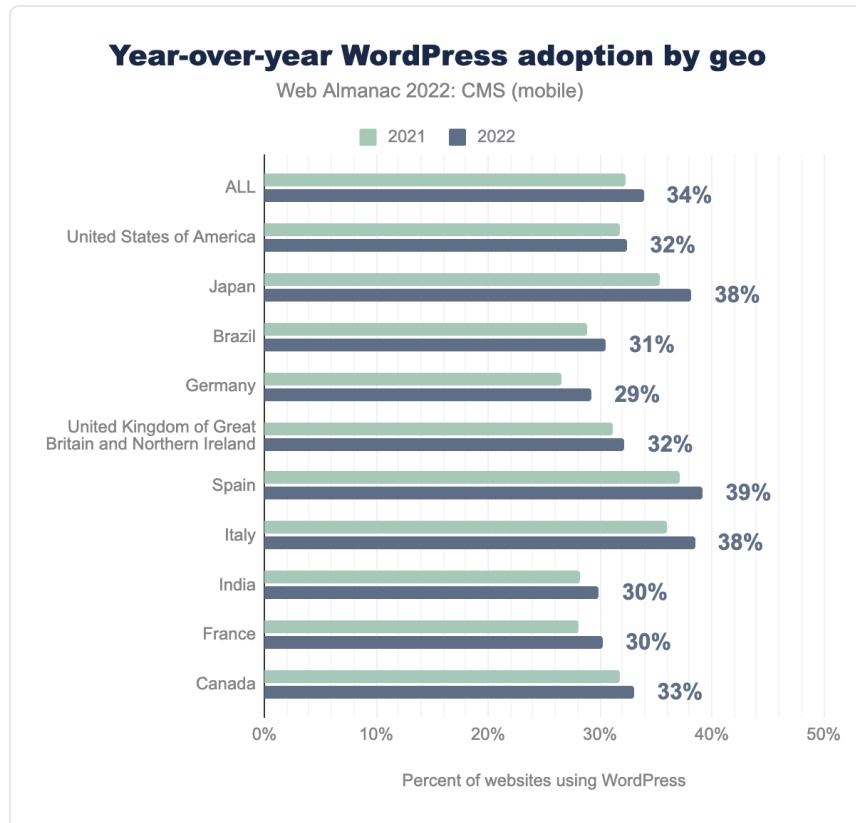


Figure 18.24. WordPress adoption by geography year-over-year on mobile.

⁶⁹⁴ <https://make.wordpress.org/core/2021/10/12/proposal-for-a-performance-team/>

According to our dataset, WordPress adoption is growing significantly in all the top geographies.

Passing CWVs by geography

Next, we looked at the number of WordPress origins with passing scores for Core Web Vitals, but this time we broke them down by geography, mobile device usage, and in comparison with our 2021 results.

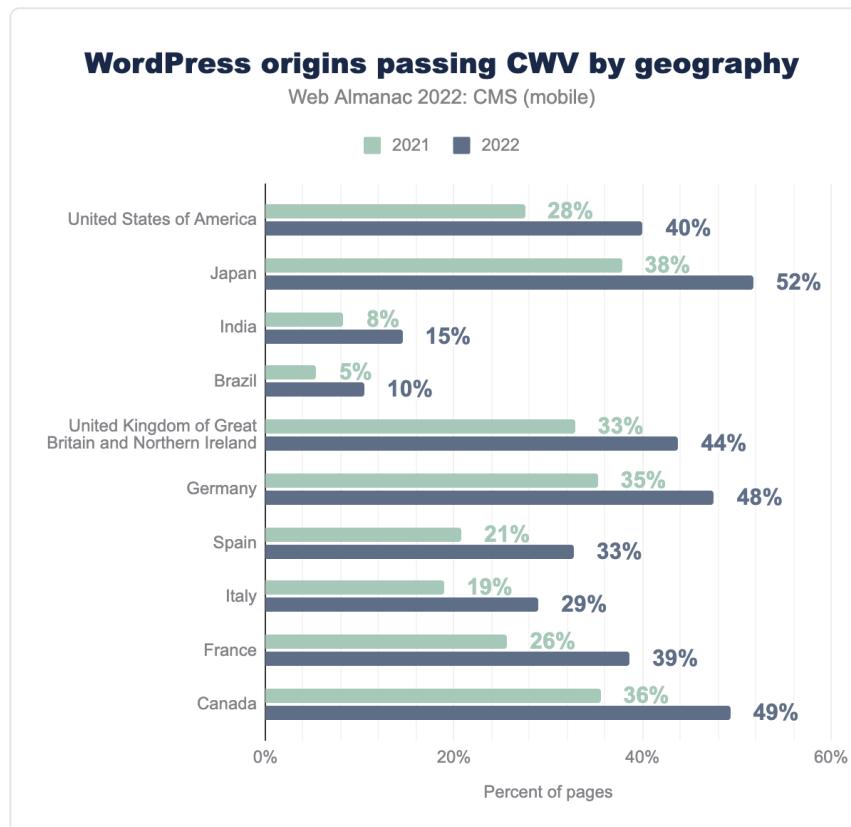


Figure 18.25. WordPress core web vitals by geography year-over-year.

All geographies showed improvements, ranging from a 5% overall gain in Brazil to 14% in Japan. Also worth noting is the large disparity across geographies, with Brazil at 10% total compared to Japan at 52%. Brazil on the low end is growing, though, improving 100% year-over-year. As we evaluate next year's dataset, it may be worth investigating the low end performers further to identify potential causes and opportunities for improvement.

Plugins

We explored how WordPress sites use external resources and separated them into resources that are included in plugins, themes, and shipped in WordPress core (wp-includes) with comparison to our 2021 results.

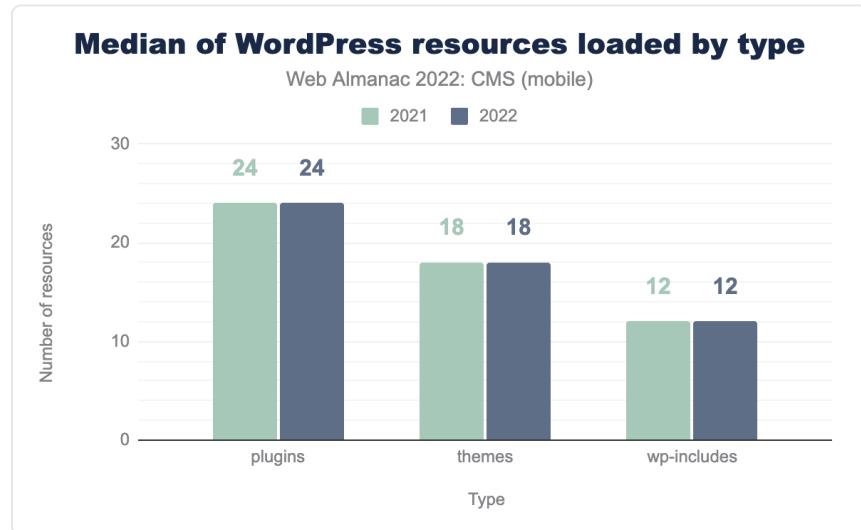


Figure 18.26. WordPress resources year-over-year.

We can see no noticeable change in the number of resources used year on year. We'll revisit again next year, perhaps looking more closely at the implied performance impact of popular themes and plugins.

Conclusion

CMS platforms continue to grow and are becoming more ubiquitous year by year. They are essential for creating and consuming content on the internet, especially as more people and businesses establish an online presence.

The introduction of Core Web Vitals, along with the advancements in performance data visibility, has made web performance a priority everywhere CMSs are used. We hope the insights in this chapter will help us all form a better understanding of the current state of the web, ultimately making the web a better place.

CMSs are doing great work and have opportunity to further improve user experiences on the

web at scale by striving to enhance their infrastructure, experiment and integrate with new standards as they evolve, and follow best practices.

On the other hand, Core Web Vitals as standards still have some evolving to do. We mentioned some ideas for a better responsiveness metric⁶⁹⁵ above. In addition, navigation between pages in a site should be better tracked and take into account the architectural differences between Single-Page Applications (SPAs) and Multi-Page Applications (MPAs)⁶⁹⁶ architectures.

We look forward to next year's results and hope to both expand our datasets and improve our methodologies. In the meantime, onward and upward, let's keep making the web better.

Author



Jonathan Wold

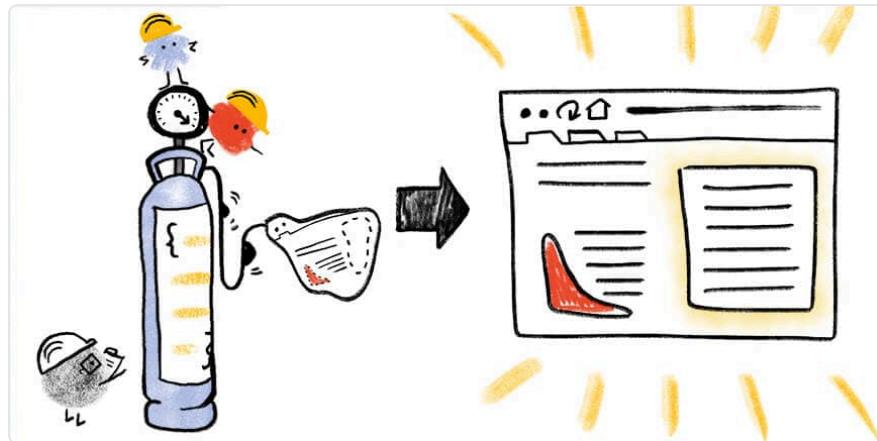
Twitter: @sirjonathan | GitHub: sirjonathan | Website: <https://jonathanwold.com>

Jonathan Wold is an Open Web advocate with more than 17 years focused on the WordPress ecosystem. Beyond his love for WordPress, he enjoys reading widely, playing strategy games, acting, rock climbing, and occasionally writing in third-person.

695. <https://web.dev/responsiveness/>
696. <https://web.dev/vitals-spa-faq>

Part III Chapter 19

Jamstack



Written by Laurie Voss and Salma Alam-Naylor

Reviewed by Barry Pollard

Analyzed by Laurie Voss and Barry Pollard

Edited by Abel Mathew

Introduction

One of the biggest problems in writing about Jamstack is defining what, exactly, the Jamstack is. Here are three different definitions (we have emphasized some words):

1. Fast and secure sites and apps delivered by pre-rendering files and serving them directly from a CDN, removing the requirement to manage or run web servers.
2. Jamstack is an **architecture** designed to make the web **faster**, more secure, and easier to scale. It builds on many of the tools and workflows which developers love, and which bring maximum productivity.
3. Jamstack is an **architectural approach** that **decouples** the web experience layer from data and business logic, improving flexibility, scalability, **performance**, and maintainability.

All three of the above definitions come from Jamstack.org⁶⁹⁷: in 2020⁶⁹⁸, 2021⁶⁹⁹, and 2022⁷⁰⁰ respectively. It's hard to think of a more authoritative source for the definition of Jamstack, so it's fair to say the definition is something of a moving target.

But as the emphasized words demonstrate, there's clearly some continuity: the sites should be fast, they should be pre-rendered, and they should use an architectural approach that decouples "where you get your data" from "how you render your data". Even if a precise dictionary definition is hard to come by, Jamstack developers know what you mean when you say "Jamstack": you've got a site that loads really quickly, renders a lot of its useful content once, at build time, and retrieves additional data (if it needs to) via JavaScript.

Disclosure: the two authors of this report were Netlify employees. Netlify invented the term Jamstack and owns Jamstack.org. This report and the underlying analysis were reviewed and approved by others not affiliated with Netlify.

Quantifying the Jamstack: what counts?

But the problem gets more tricky when you're trying to put together the 2022 Web Almanac. When you're dealing with millions of websites, "I know it when I see it" can't be your definition. How do we quantify the Jamstack? How do we precisely identify it so we can learn about it? We started by asking ourselves a series of questions.

Is every static site a Jamstack site?

That seems like it should be an obvious "yes": if the page is flat HTML that renders all at once then it certainly sounds like it should be Jamstack. But what about all those pages people built in the 90s, before JavaScript was popular and most sites were static? Are they Jamstack? It felt like they weren't, not every static site is a Jamstack site. So we tried to think of why.

We landed on the "CDN" aspect of the early definition of Jamstack: it doesn't have to be any specific CDN provider, but part of the definition is definitely the "pre-rendering" part, which implies: you're rendering something, and then caching it. So a Jamstack site should be cached (though whether you cache it yourself, or use a CDN, doesn't matter).

That produces another edge-case: lots of sites are cached! Even completely dynamic sites often cache things for a few minutes to avoid load spikes, and lots of sites are served by CDNs these days, which are intrinsically a cache even if the site's architecture owes nothing to Jamstack

697. <https://jamstack.org/>

698. <https://web.archive.org/web/20200331214426/https://jamstack.org/>

699. <https://web.archive.org/web/20210924115533/https://jamstack.org/what-is-jamstack/>

700. <https://web.archive.org/web/20220809174445/https://jamstack.org/>

patterns. So what's the difference between a normal cached site and a Jamstack site? Cacheability is one part, but what else?

Does a Jamstack site have to use JavaScript?

We decided a Jamstack site doesn't necessarily use JavaScript. Lots of Jamstack sites do, of course: the "J" in Jamstack used to stand for "JavaScript", after all. But even the earliest definitions of Jamstack made it clear that using JavaScript was optional – a fully static site with no JavaScript has always been Jamstack.

Does using the Jamstack mean a specific framework?

There are definitely some frameworks that people think of when they think about the Jamstack; so much so that the 2020 and 2021 versions of the Web Almanac defined the Jamstack exclusively by the frameworks used⁷⁰¹, focusing on Static Site Generators (SSGs). That's logical enough, but we thought this presented some problems.

First, what about frameworks you can't easily detect? As an example, Eleventy⁷⁰², a growing and popular choice in the Jamstack, leaves no trace in the generated HTML; it's invisible to the end user (by default, though you can add a generator tag⁷⁰³ if you want to). Not counting frameworks that politely get out of the way seems wrong.

Secondly: there are a lot of frameworks! There are dozens of big ones and thousands of smaller ones. Even for the ones that can be detected, we don't always have a good way to detect them. Plus, we agreed that it is definitely possible to build a site that feels "Jamstack-y" without using a framework at all. Plain HTML can definitely be Jamstack.

Thirdly: using a framework that's popular with Jamstack developers by no means guarantees that the site you build will be a Jamstack site. If for architectural reasons it's really slow to render, or dynamically renders every page, it's not going to be a Jamstack site even if you're using the same framework as many Jamstack sites. Not every site has to be Jamstack, and that's okay.

So we decided to ignore frameworks as part of the definition this time around, although as you'll see later, the frameworks you'd expect to find turned up in the results anyway.

701. <https://almanac.httparchive.org/en/2021/jamstack>

702. <https://www.11ty.dev/>

703. <https://www.11ty.dev/docs/data-eleventy-supplied/#eleventy-variable>

Does a Jamstack site have to be performant?

The only feature common to all three definitions of Jamstack was *performance*. But “fast” is kind of a fuzzy word when it comes to websites: if you’ve spent any time reading the Web Almanac, you’ll know there are dozens of metrics you can use to measure the performance of a website, and lots of good arguments for all of them.

So we thought hard about what a Jamstack site feels like. First was that a Jamstack site renders its initial content very quickly. Of all the metrics we could use, we decided the one that most clearly captured that idea was Largest Contentful Paint⁷⁰⁴, or LCP. This is a measure of the time it takes for the largest item on the page to render. You can pull in extra content via APIs, or not, and still be Jamstack, but you have to render something substantial quickly.

Defining the metrics

If you are not interested in the nuts and bolts of how we picked a precise definition of Jamstack that we could represent as queries in the HTTP Archive, you can safely skip the next two sections and head down to the growth of the Jamstack. Understanding our methodology is not critical to you getting actionable insights here.

We knew we wanted to measure: sites that load most of their content very quickly, and can be cached. After a lot of experimentation with different ways of measuring these things, we came up with some specific metrics.

Largest Contentful Paint (LCP): we got the distribution of all LCP times across all pages, picked the median of real-world user data from the Chrome UX Report⁷⁰⁵, and decided that any site equal or less to the median counted as “loaded most content quickly”. This was 2.4 seconds on mobile devices, and 2.0 seconds on desktop devices.

Cumulative Layout Shift (CLS): we wanted to avoid sites that very quickly load a skeleton but then take a long time to load real content. The closest we could get to that is the Cumulative Layout Shift⁷⁰⁶, a measure of how much the page layout jumps around while loading. While there are ways to “game” CLS, we still believe it’s a reasonable proxy for what we’re trying to measure. We liked this measure because we felt that a “jumpy” site also felt less “Jamstack-y”, a word we were going to end up using a lot. Again, we picked the median of Chrome UX Report data.

Chrome UX report data rounds CLS data to the nearest 0.05, which is a shame, because the “real”

704. <https://web.dev/lcp/>
705. <https://developer.chrome.com/docs/crx/>
706. <https://web.dev/cls/>

median seems to be around 0.02-0.03, so on mobile it rounds down to zero and on desktop it rounds up to 0.05. Since 0 excludes huge numbers of pages, we decided to use 0.05 as the best available threshold for both mobile and desktop.

Caching: this was particularly tricky to quantify, since most home pages, even on Jamstack sites, request revalidation even if they are in practice cached for a long time. We went with a combination of HTTP Headers including `Age`, `Cache-Control`, and `Expires` that we found were common in pages that could be cached for a long time.

We initially thought we'd need another measure to exclude "tiny" sites – sites that load very quickly because they are just a "coming soon" or "Hello world" page that nobody would visit in real life – but the HTTP Archive data is defined by their popularity according to Chrome⁷⁰⁷ user visits, and very few of those sites are visited enough to make it into the sample (although example.com is in there!).

A good question is: why not use Core Web Vitals⁷⁰⁸ (CWV) numbers for these metrics? For LCP, our numbers are nearly the same as CWV. For CLS, the CWV team explicitly relaxed the requirements⁷⁰⁹ (their threshold is more than double the median) which we thought was not representative of a Jamstack experience. So we decided it was fairer to pick the median for both. And CWV does not have a metric for "cacheability".

“Jamstack-y”: a disclaimer

We want to be clear: this is a very, very fuzzy definition of "Jamstack". In fact, we started using the word "Jamstack-y" when doing this work, just to be clear.

The biggest source of error is fundamental: the definition of Jamstack is about *architecture*, but architecture is not something you can determine by crawling the generated HTML, except in very broad strokes. There is simply no way to look at a pile of HTML and tell whether the front-end and the back-end were decoupled. So our measurements, while a best effort, are a rough estimate, and we don't want to misrepresent that.

This methodology both under-counts and over-counts Jamstack sites:

- If your site is static but not decoupled (for instance, SquareSpace⁷¹⁰ and Wix⁷¹¹ sites are clearly tightly coupled to their back-ends) but performs well, we'll over-count it.
- If your Jamstack site is decoupled but you update your content very frequently,

707. <https://developer.chrome.com/docs/crxu/methodology/#popularity-eligibility>

708. <https://web.dev/vitals/>

709. <https://web.dev/defining-core-web-vitals-thresholds/#achievability-3>

710. <https://www.squarespace.com/>

711. <https://www.wix.com/>

your caching strategy might be different than what we look for, so we'll under-count it.

- If your Jamstack site renders very slowly, even though it's decoupled, your LCP number will be high and we will under-count it.
- Conversely, if your non-Jamstack dynamic site is really fast, we might mistake it for Jamstack and over-count it.

Despite all these caveats, we think this year's estimate of "Jamstack-y" sites is an improvement over a strictly SSG-focused definition and gives a better sense of where the web really is today, which is after all the goal of the Almanac.

The growth of the Jamstack

Applying our new criteria, we measured what percentage of sites in the HTTP Archive qualify as "Jamstack". Because the measures we used in 2020 and 2021 were very different, we also went back and re-measured those samples using the 2022 definitions.

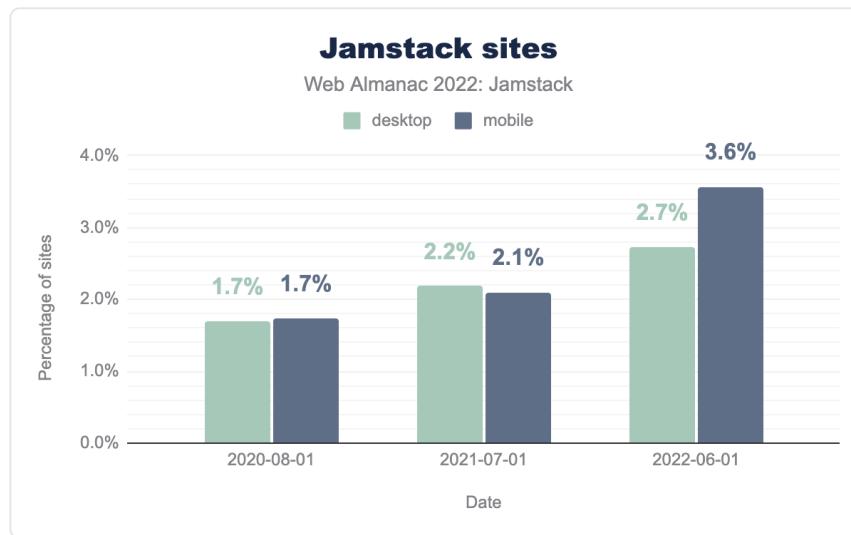


Figure 19.1. Jamstack sites.

Our headline finding is that 3.6% of mobile websites in 2022 seem "Jamstack-y" and that this has grown more than 100% since 2020. On desktop, 2.7% of sites are Jamstack-y and that number is also growing, the difference between the two groups being driven primarily by

different numbers of sites meeting the CLS threshold, which varies a lot by device because of layout differences. Again, see above for the many caveats about how approximate this is.

The historic figures, with this new definition, are considerably higher than measured last year⁷¹² when we based the adoption purely on technologies used. This is perhaps not surprising when we consider the limits of detecting certain technologies, coupled with the inclusion of technologies that were not previously considered as Jamstack.

When we as humans randomly sampled the sites in the set we identified, we were mostly satisfied that we were finding sites that, to the best of our abilities to judge, looked and felt like Jamstack sites are supposed to look and feel.

To judge for yourself, and keep us honest, here are 10 “Jamstack-y” sites from our sample, selected entirely at random without curation of any kind:

- www.cazador-del-sol.de
- [snpbooks.org⁷¹³](http://snpbooks.org/)
- [eikounoayumi.jp⁷¹⁴](http://eikounoayumi.jp)
- [rochesteronline.precollegeprograms.org⁷¹⁵](http://rochesteronline.precollegeprograms.org)
- [surveyforcustomers.com⁷¹⁶](http://surveyforcustomers.com)
- www.shopmate.eu
- [docs.saleor.io⁷¹⁷](http://docs.saleor.io)
- www.wildeyebrewing.ca
- [360insurancegroup.com⁷¹⁸](http://360insurancegroup.com)
- [144onthehill.co.uk⁷¹⁹](http://144onthehill.co.uk)

Whether or not exactly 3.6% (or 2.7% on desktop) of the web is Jamstack – which, because the definition of Jamstack relies on architectural choices we can't verify, we cannot definitively say – what we can be sure of is that Jamstack is growing. The percentage of sites that meet all of our criteria has been getting steadily bigger year on year. The web is getting more “Jamstack-y”.

Of course, since our definition is two performance metrics and a caching metric, one way we

712. <https://almanac.httparchive.org/en/2021/jamstack#adoption-of-ssgs>

713. <https://snpbooks.org/>

714. <https://eikounoayumi.jp/>

715. <https://rochesteronline.precollegeprograms.org/>

716. <https://surveyforcustomers.com/>

717. <https://docs.saleor.io/>

718. <https://360insurancegroup.com/>

719. <https://144onthehill.co.uk/>

could be wrong if the web is just getting more performant in general. To check that, we split the metrics back up (this is mobile data; desktop data was not significantly different):

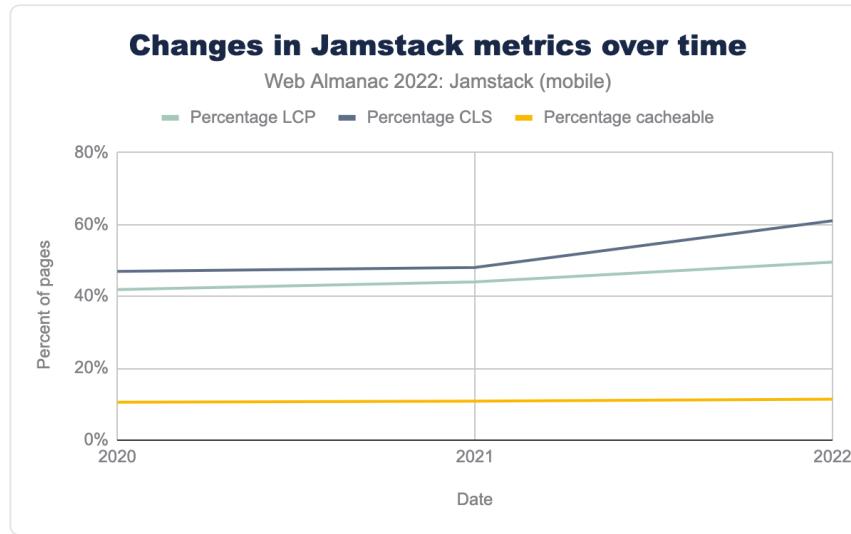


Figure 19.2. Changes in Jamstack metrics over time on mobile.

As you can see, there has been some mild improvement in our metrics from 2020 to 2022. However, even the smallest number here – the percentage of sites that meet our caching criteria – is 11-14% of the web, depending on the year and whether you’re looking at mobile or desktop. Our set of Jamstack sites is the intersection of these groups; the set of sites that meet all 3 of these criteria at the same time is a lot smaller than any of the individual groups.

So we really are looking at a distinct subset of sites, and the set is growing a lot faster than the performance of the web as a whole is improving. We aren’t just measuring “how many fast sites are there”.

Jamstack-y frameworks

Having satisfied ourselves that Jamstack-y sites are real and identifiable, we can now ask some questions about them. This is where it gets fun: because our definition of Jamstack-y doesn’t include a framework, we can look at our sites and see what frameworks show up most often in the Jamstack.

We used framework identifications provided for us by Wappalyzer, which means (as we mentioned earlier) some “invisible” frameworks like Eleventy can’t be counted or analyzed.

Wappalyzer has a somewhat arbitrary distinction between “web frameworks” and “JavaScript frameworks”. Here are the top 10 JavaScript frameworks for the web as a whole:

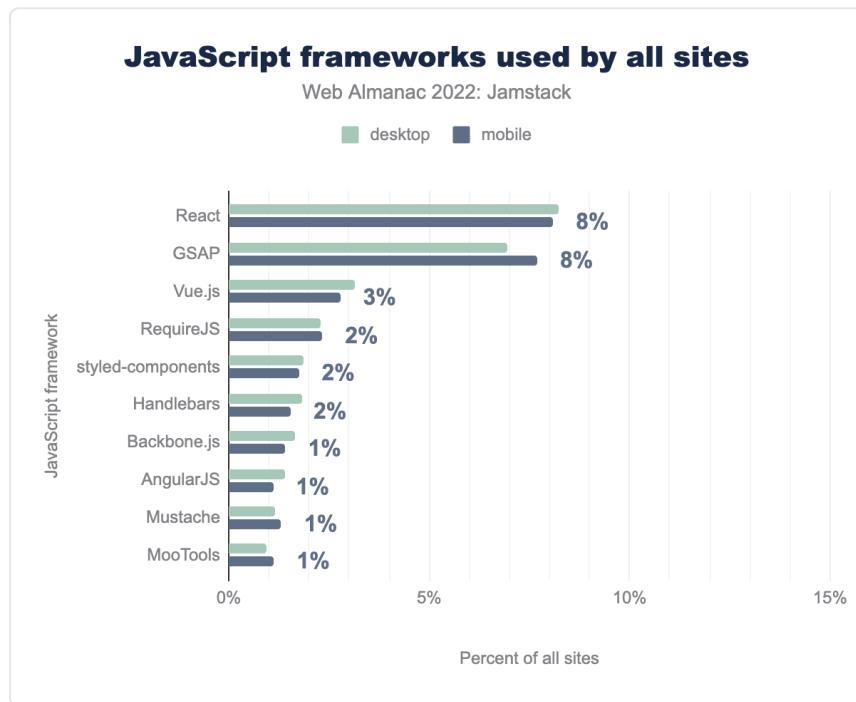


Figure 19.3. JavaScript frameworks used by all sites.

And here's the top 10 in Jamstack sites:

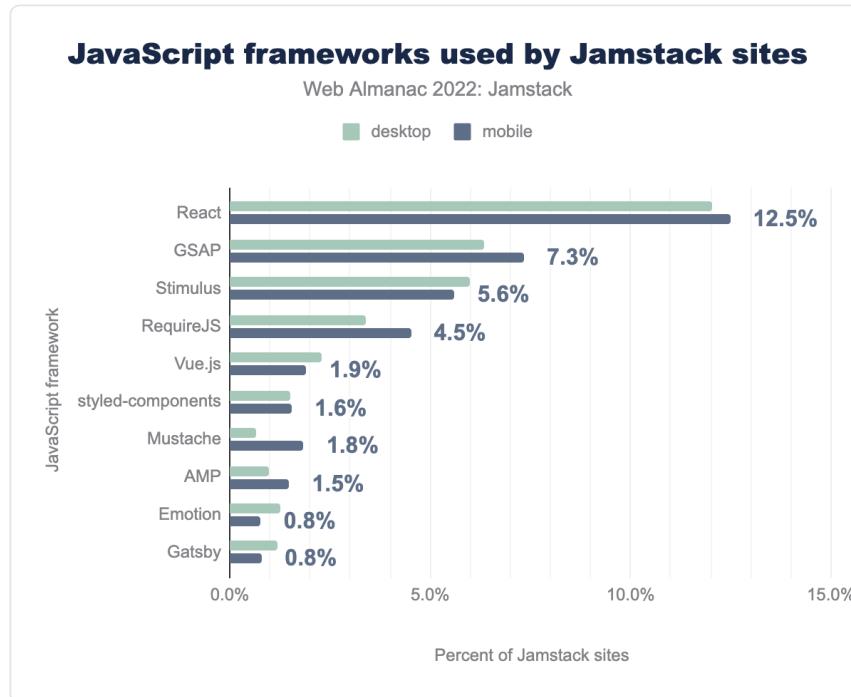


Figure 19.4. JavaScript frameworks used by Jamstack sites.

You can see that React is more popular in the Jamstack than the general web, and so is Gatsby. Now let's look at "web frameworks", again as somewhat arbitrarily defined by Wappalyzer:

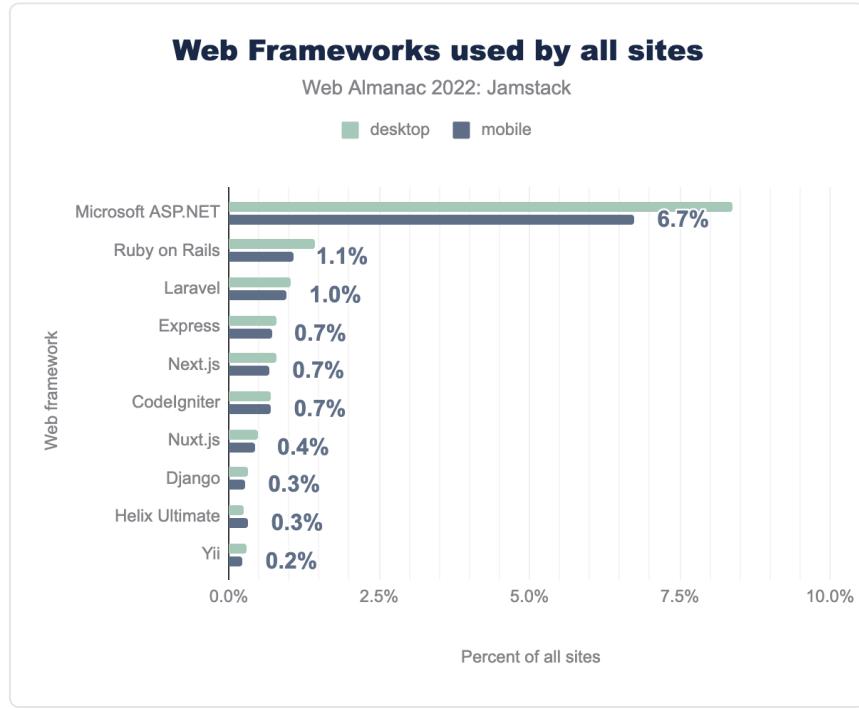


Figure 19.5. Web frameworks used by all sites.

A great question here is: why are Next.js and Nuxt.js considered web frameworks, but Vue.js and React considered JavaScript frameworks? But leaving that aside, you can see that Microsoft's ASP.Net framework is extremely popular across the web as a whole, and so is stalwart Ruby on Rails. What does it look like in the Jamstack?

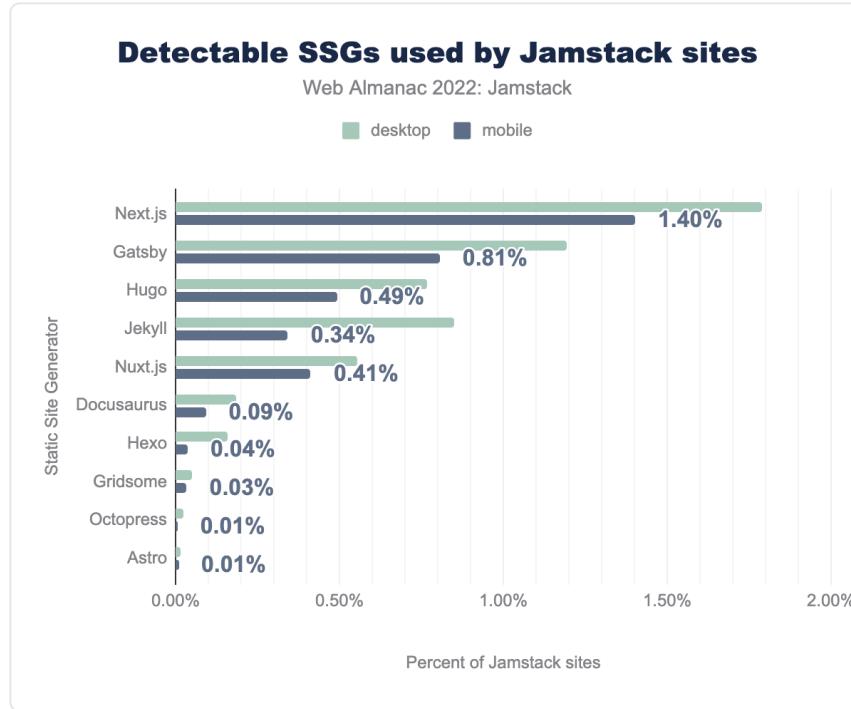


Figure 19.6. Web frameworks used by Jamstack sites.

As you can see, while still the number one web framework, ASP.Net is far less popular in Jamstack, and so is Ruby on Rails. Instead, Jamstack favorite Next.js climbs from fifth to third place, and Nuxt.js from seventh to fifth. A surprising addition is Symfony, which misses the cut for general sites (it's number 11) but climbs all the way to second place in our Jamstack set.

Since Next.js and Nuxt.js are two of the biggest frameworks in the Jamstack community, this is not a huge surprise, but it was again nice to see our framework-agnostic definition correctly identifying “Jamstack-y” sites.

At first glance it might be surprising that ASP.Net is still #1 in the Jamstack-y group, and even more to see PHP-based Symfony hit #2. But there’s no reason you can’t build a performant, modern site using ASP.NET or PHP: Jamstack is an architectural approach, not any specific list of technologies, so we hope those working in less-trendy tech stacks will find this result encouraging.

What about the SSGs? Wappalyzer has them as a separate category; here are their numbers for both Jamstack-y and general sites (note: we added Nuxt.js and Next.js manually to this list; Wappalyzer does not consider them SSGs but both can be used that way, so we thought it was

useful to consider them). Here they are for all sites:

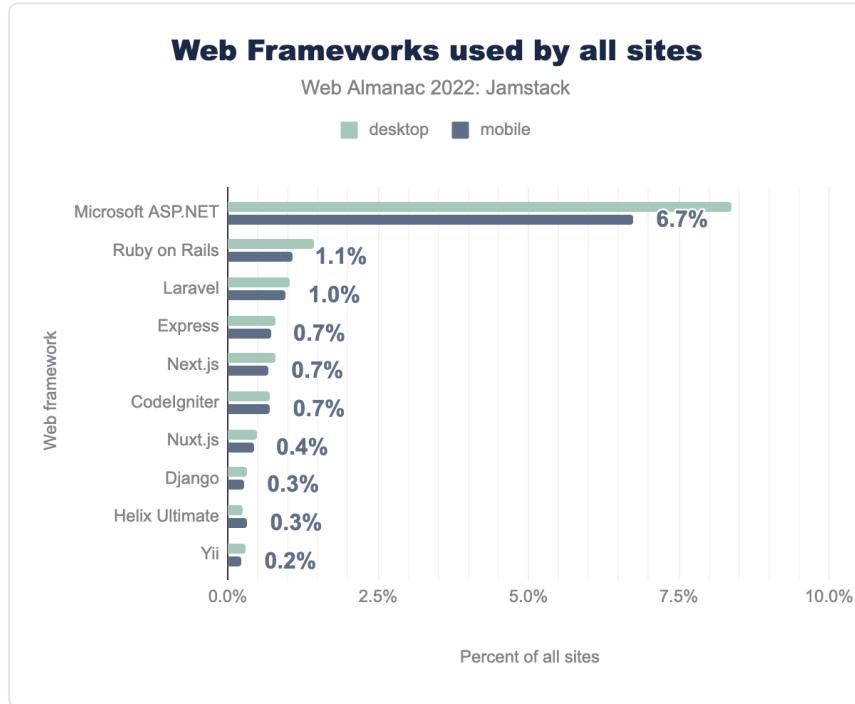


Figure 19.7. Detectable SSGs used by all sites.

And here they are for Jamstack sites:

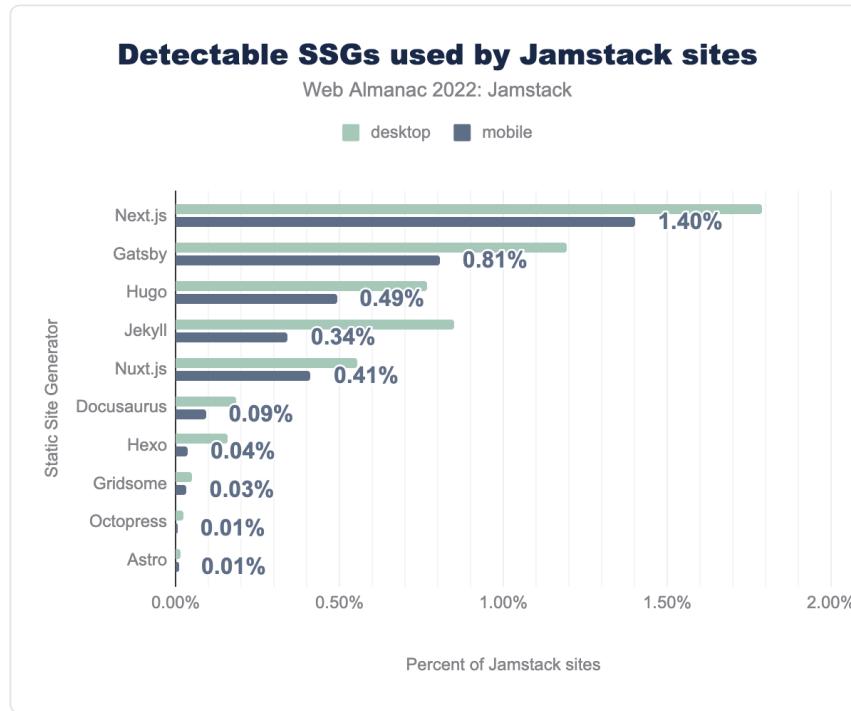


Figure 19.8. Detectable SSGs used by Jamstack sites.

As you can see, it's very much the same list, in almost the same order, although Nuxt drops a few spots. This makes intuitive sense, since you'd expect sites generated by SSGs to qualify as Jamstack-y, although they are clearly not the only way to achieve that architectural goal.

The SSGs also make up a much larger percentage of all Jamstack sites than they do of all sites in general, indicating that an SSG is a pretty good way of getting a Jamstack site. However, using an SSG doesn't guarantee you'll make a Jamstack site. Take a look at the total numbers of some of the frameworks in our sample:

SSG	All sites	Jamstack sites	Jamstack %
Next.js	39,928	2,651	7%
Nuxt.js	24,600	824	3%
Gatsby	12,014	1,765	15%
Hugo	5,071	1,135	22%
Jekyll	3,531	1,259	36%

Figure 19.9. SSGs as a percentage of Jamstack sites (desktop).

For all the SSGs, the percentage of sites that qualified as Jamstack-y by our definition was less than the total number of sites in that framework. Jekyll does best with more than a third of sites in Jekyll also meeting our criteria. Next and Nuxt have particularly low percentages, which is to be expected since even though they can be used as SSGs they are also frequently used to make dynamic sites, and we don't have a way of determining which mode they're in.

Jamstack-y hosting

We were also interested in where people host their Jamstack-y sites. Would there be a pattern? Once again, we used Wappalyzer's data to identify technologies, this time using their Platform as a Service (PaaS) category.

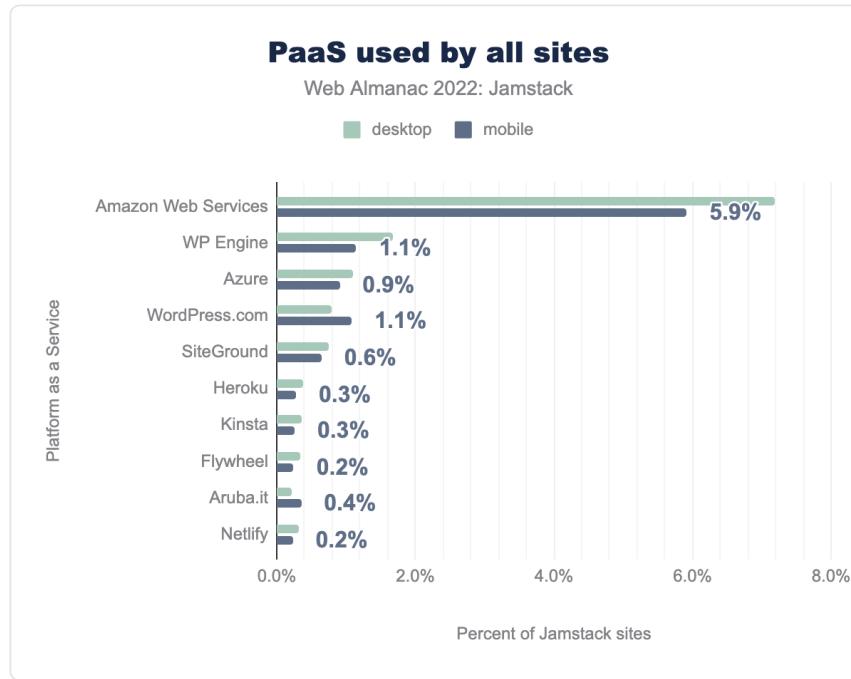


Figure 19.10. PaaS used by all sites.

And here for Jamstack sites:

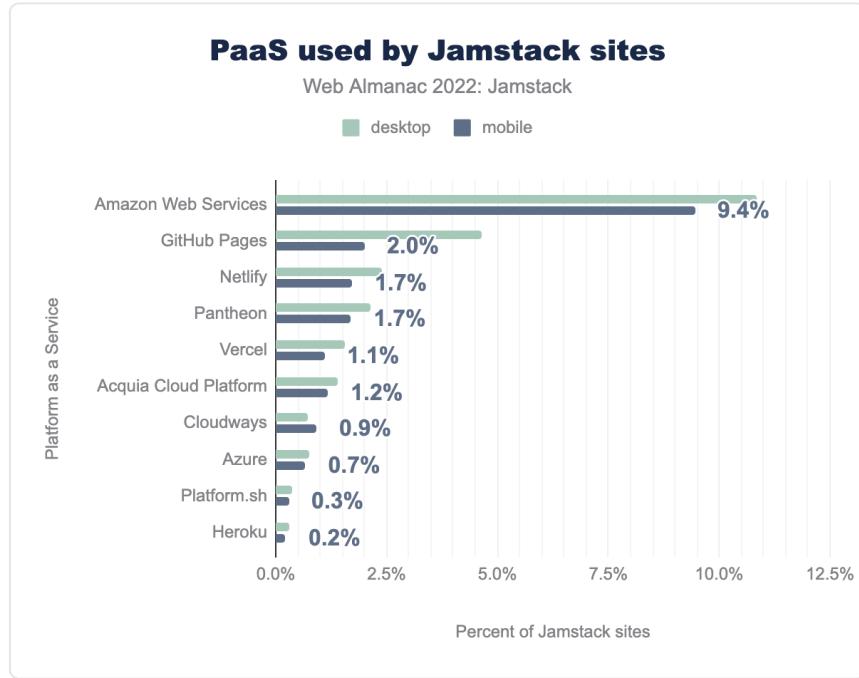


Figure 19.11. PaaS used by Jamstack sites.

Web giant Amazon Web Services is unsurprisingly dominant in both sets, but there are some significant differences between the global preferences and those of Jamstack-y developers.

WP Engine, Azure, and WordPress.com, hugely popular on the web as a whole, drop significantly in popularity in the Jamstack crowd. GitHub pages, which is #11 on the wider web, is #2 in the Jamstack set. Meanwhile Netlify and Vercel, darlings of Jamstack developers, occupy the #3 and #5 spots, while in the larger web Netlify is at #10 and Vercel at #14 (not shown). Pantheon and Acquia Cloud Platform, neither in the top 10 overall, jump significantly from #11 to #4 and from #12 to #6 respectively.

The change in relative popularity of some of these hosts relative to the wider web is perhaps surprising given that they are not all household names, so we thought it was worth looking at how platform preferences changed from 2021 to 2022 in our sets. Using mobile data, here's how the percentage of Jamstack sites using various platforms changed from 2021 to 2022:

PaaS	2021	2022	Change
Amazon Web Services	7.00%	9.45%	2.45%
GitHub Pages	2.62%	1.99%	-0.63%
Pantheon	1.97%	1.70%	-0.27%
Netlify	1.68%	1.72%	0.04%
Acquia Cloud Platform	1.37%	1.18%	-0.20%
Vercel	0.50%	1.10%	0.60%
Cloudways		0.91%	N/A
Azure		0.67%	N/A
Platform.sh	0.27%	0.29%	0.02%
Heroku	0.28%	0.22%	-0.05%

Figure 19.12. SSGs as a percentage of Jamstack sites (desktop).

GitHub Pages, Pantheon, Acquia Cloud Platform and Heroku all appear to be declining in popularity as a Jamstack hosting choice, while AWS, Netlify, Vercel, and Platform.sh are getting more popular. Note that Cloudways or Azure are not in the 2021 PaaS data, so we can't compare them. We can hypothesize that AWS, Netlify and Vercel are growing in popularity because they're not just hosting—they offer a suite of tools for a developer workflow.

Absent from all the platform lists is web giant Cloudflare, which Wappalyzer categorizes as a CDN rather than a PaaS. Although Cloudflare has a PaaS offering that is very Jamstack-y, called Cloudflare Pages, Wappalyzer data does not distinguish between "hosted on a CDN" and "hosts some assets on that CDN" so we could not include it in this analysis. The author believes that Cloudflare is a very popular Jamstack hosting option, but we do not have good data to verify this.

Conclusion

Our most important takeaway from this year's analysis is that the Jamstack is hard to measure just by looking at HTTP Archive data. Nevertheless, our ability to use a measurement approach that was agnostic to both platform and framework and find in the resulting data strong correlations to "known" Jamstack platforms and frameworks was an encouraging sign that we have made progress in reliably identifying Jamstack sites in the Archive.

Although we can't claim to know exactly what percentage of the web is Jamstack, we can say that around 3% of the web is Jamstack-y, and that this group has been growing strongly for the last 3 years—a great sign for the Jamstack community.

We also found some frameworks and hosting platforms are more popular in the Jamstack than they are in the wider web. This might be because they are technically better at achieving our criteria, or it might just be because Jamstack developers have community preferences for specific stacks.

Of course, if the Jamstack community prefers certain platforms and frameworks, that becomes a self-reinforcing trend: there will be more documentation and tutorials on how to achieve Jamstack sites using those tools, which will in turn make it easier to build Jamstack sites using those tools. So while we believe (and the data demonstrates) that you can achieve Jamstack-y results with any tech stack, we hope you find this data useful in identifying tools and platforms that might make it easier to achieve a Jamstack site.

We also believe that the final useful takeaway from this exercise in quantifying “what counts as Jamstack” is that now, as a developer, you have a firmer target to aim for when building a Jamstack site. It doesn’t mean you pick a specific framework and forget about it: it’s about the results. By analyzing your LCP and CLS times you can quantify if your efforts are “Jamstack-y”, which is a useful thing to be able to automate.

Authors



Laurie Voss

[seldo](#) https://seldo.com

Laurie has been a web developer since 1996, with occasional breaks to found companies like `awe.sm`⁷²⁰ (2010) and `npmjs.com`⁷²¹ (2014). He currently works as a Data Evangelist at Netlify⁷²². He loves making the web bigger and better. He thinks one of the best ways to do that is to encourage more people to do web development, by teaching them existing techniques and by building tools and services that make web development easier, so they don't have to learn so much.

720. <https://www.crunchbase.com/organization/snowball-factory>

721. <https://npmjs.com/>

722. <https://netlify.com>



Salma Alam-Naylor

Twitter [@whitep4nth3r](https://twitter.com/whitep4nth3r) GitHub [whitep4nth3r](https://github.com/whitep4nth3r) Website <https://whitep4nth3r.com/>

Salma helps developers build stuff, learn things and love what they do. She works at Netlify as a Staff Developer Experience Engineer, streams live coding, and loves helping people get into tech. After a career as a music teacher and comedian, Salma transitioned to technology in 2014, specializing in front end development and tech leadership for startups, agencies and global e-commerce. Find Salma on Twitch⁷²³ to see what she's currently building.

⁷²³ <https://twitch.tv/whitep4nth3r>

Part III Chapter 20

Sustainability



Written by Laurent Deverney, Gerry McGovern, and Tim Frick

Reviewed by Chris Adams, Caleb Queern, and Edmond de Tournadre

Analyzed by Fershad Irani, Cameron Casher, and Arik Smith

Edited by Barry Pollard

Introduction

Back in 2019, GreenIT.fr estimated that there were 34 billion pieces of equipment and 4.1 billion internet users⁷²⁴. As such, the digital world's contribution to humanity's carbon footprint may represent roughly 4% of primary energy consumption and greenhouse gas emissions, as well as 0.2% of water consumption and 5.5% of electricity consumption.

Another significant indicator is its contribution to the depletion of abiotic resources ("not alive" resources, such as metals). All the devices we use need materials in order to be produced. As such, the manufacture of user equipment is considered the most important source of environmental impact. This is followed by the end of life of equipment as most of them not being recycled at all. It is way more impactful than data centers, the network or even the usage of user equipment. Despite the efforts from some manufacturers, it will likely only get worse in the coming years because of the depletion of some required materials (indium, copper, gold,

724. <https://www.greenit.fr/environmental-footprint-of-the-digital-world/>

etc).

The previously mentioned study from GreenIT.fr states that the overall impact of digital services has been steadily increasing for years and could double or triple between 2010 and 2025. If we want to avoid—or at least mitigate—this, we should reduce the number of connected devices that we own and keep each of them for as long as possible: repairing rather than buying. This might sound tough because some devices, especially smartphones, seem to be aging quickly: the longer websites and applications take to load, the less a battery will last.

What we can do about that is to reduce the impact of digital services and change the way we think about digital services as being immaterial and environmentally friendly by default.

Considering all the data gathered, the Web Almanac sounds like a great place to assess the environmental impacts of websites as a whole. On this journey, we will also see how to reduce them through best practices and how widely these are already adopted.

For this, we will differentiate:

- Moderation: implementing something only when needed. It could be digital as a whole (do you really need connected diapers?), some functionality (are these social media feeds useful on your homepage?) or content (decorative images, videos, etc). Ask yourself if everything on your website is useful, used, usable (and reusable).
- Efficiency: how you reduce the size and/or impact of what remains on your website after considering sobriety. For websites, this is mostly done through technical optimizations such as minification, compression, caching, etc.

Some online activities to get started:

- Are you an eco-responsible Internet user?⁷²⁵
- What is the impact of your internet browsing?⁷²⁶
- Weight comparison of various elements composing a web page⁷²⁷

To guide us on this journey, we can rely on some resources, including:

- Repositories of best practices: 115 bonnes pratiques⁷²⁸, Handbook of sustainable digital services⁷²⁹.

725. https://learninglab.gitlabpages.inria.fr/mooc-impacts-num/mooc-impacts-num-ressources/Partie3/Activites/Capsule_Partie3_4_AgitUtilisateur/story.html

726. https://learninglab.gitlabpages.inria.fr/mooc-impacts-num/mooc-impacts-num-ressources/Partie3/Activites/Capsule_Partie3_3_Mesurer/story.html

727. https://learninglab.gitlabpages.inria.fr/mooc-impacts-num/mooc-impacts-num-ressources/Partie3/Activites/Capsule_Partie3_2_Mesurer/indexEn.html

728. <https://github.com/cnurm/best-practices/>

729. <https://gr491.isit-europe.org/en/>

- Books and websites: Sustainable Web Design⁷³⁰.
- Online courses: INR - Sustainable IT⁷³¹, Environmental impact of digital services⁷³², Principles of Sustainable Software Engineering⁷³³.

Limitations and hypothesis

We won't be covering all best practices and available metrics can't cover all of them. Metrics can't tell us if a given website has unnecessary functionality or if some images are purely decorative. Even if such considerations go beyond the scope of this chapter, there is still a lot that can be done. And with Lighthouse providing more and more types of audits, we can expect new metrics to become available.

Carbon emissions are the only environmental indicator here but others—such as water consumption, land use, abiotic resources consumption—should be considered to avoid pollution transfers. This is exactly the point of LCA (Life Cycle Assessment)⁷³⁴. However, such an operation requires expertise, lots of information and time. As of today, some structures are reaching for a compromise by using less metrics and information, combined with LCI (Life Cycle Inventory). This helps make the evaluation of environmental impacts more affordable and accessible (and repeatable, for example in CI/CD or monitoring) while keeping under control the assumptions you need to make.

We will only use metrics collected on pages but, in order to assess the environmental impacts of some digital services, it might be more accurate to collect metrics on a whole user journey. For example, on an ecommerce website, it would be better to consider a user purchasing an article and paying for it.

Intersectional environmental issues

Sustainability has evolved significantly since its initial definition in 1987 by the Brundtland Commission. It now incorporates a variety of intersectional social and governance issues (the "S" and "G" in "ESG") alongside its core environmental focus. A more responsible and sustainable internet should reflect this.

In other words, digital sustainability cannot focus only on emissions. While climate change is a huge driver, it cannot be used to justify inequitable solutions⁷³⁵ or fuel inequality⁷³⁶ [PDF] in any

730. <https://sustainablewebdesign.org/>

731. <https://www.iisit-academy.org/>

732. <https://learninglab.inria.fr/en/mooc-impacts-environnementaux-du-numerique/>

733. <https://docs.microsoft.com/en-us/learn/modules/sustainable-software-engineering-overview/>

734. <https://learninglab.gitlabpages.inria.fr/mooc-impacts-num/mooc-impacts-num-ressources/en/Partie3/FichesConcept/FC3.3.1-ACVservicesnumeriques-MoocImpactNum.html?lang=en>

735. <https://qz.com/845206/renewable-energy-human-rights-violations/>

736. <https://www.iisd.org/system/files/publications/green-conflict-minerals.pdf>

way. It must be grounded in climate justice.

To this end, when designing digital products and services, keep the following intersectional issues in mind:

- **Accessibility:** By removing barriers to content, your website becomes more usable and accessible. This also improves its environmental impact because users, especially those with disabilities, don't have to find workarounds to accomplish tasks.
- **Privacy:** A less intrusive website is better for users, giving them control over their data and what they choose to share. Privacy-focused websites are also often more environmentally-friendly in that they track, store, and maintain less data.
- **Mis/disinformation:** People turn to the internet to answer questions. Content that includes misinformation (unintentional) and disinformation (intentional) undermines users' ability to do this in an efficient manner.
- **Attention economy:** Avoiding deceptive patterns⁷³⁷ keeps users focused, reducing pointless browsing or diverting them from their initial purpose.
- **Security:** Aiming for sustainability can also help secure your website by reducing its attack surface: less external resources, less functionality, etc

These are all part of a broader organizational approach to corporate digital responsibility⁷³⁸ that aligns with digital sustainability principles.

Understanding the environmental impact of the web

The internet is the greatest, most energy intense, machine that has ever existed. To create and maintain the internet requires massive material input. One server can cause one ton or more of CO₂ during manufacture. A laptop can cause 300 kg of CO₂ to manufacture, and result in the mining of 1,200 kg of raw materials. There is no such thing as sustainable mining.

While most of the energy and waste of the internet is embedded in the devices themselves, the energy required to run the internet is not insignificant. While we are constantly marketed to about how data is essentially free, to store as much as we want, data storage and processing have real and exponentially growing energy demands. In 2015, for example, data centers in Ireland were consuming 5% of electricity, by 2021 that had grown to 14%—more than the demand of all the houses and buildings in rural Ireland.

737. <https://blog.mozilla.org/en/internet-culture/mozilla-explains/deceptive-design-patterns/>

738. <https://www.mightybytes.com/blog/what-is-corporate-digital-responsibility/>

We can design and develop more sustainability for the web by focusing on better managing our devices, and by seeking to put as little stress as possible on the devices that are used to interact with our websites or apps. In relation to our own devices, we must focus on device life and energy consumption. The longer the working life of a computer, the more we can amortize the harm that was caused during its manufacture. The pinnacle of this thinking is going open source and using an operating system like Linux to extend the life of a device. Open source is the original digital sustainability philosophy by focusing on reusing and sharing. Nonetheless, it should not prevent the implementation of sustainability best practices.

The less energy consumed during the design and development process the better. If we can reuse code or content, then that's a great idea. Use the least amount of wattage. A laptop will be much more energy efficient than a desktop. Large screens, for example, should be avoided, as they can consume as much energy themselves as a laptop. Anything that reduces energy consumption is a good thing.

For popular, high demand websites or apps, up to 98% of the energy and waste consequences will occur on the smartphones or laptops of the users. Small savings can make a big difference. Danny van Kooten, developed a Mailchimp plugin for WordPress that is used by two million websites. He made a 20 KB reduction in code and estimated that that resulted in a monthly reduction of 59,000 kgs of CO₂⁷³⁹.

Evaluating the environmental impact of websites

We decided on the methodology already shared by some available tools such as Ecograder, Website Carbon, E coping, CO2.js and others⁷⁴⁰. Thanks to this, we estimate the Greenhouse Gas emissions based solely on the amount of data transferred (for instance here, the page weight).

The community is still struggling on reaching a consensus on this topic⁷⁴¹. Given the metrics available here, this sounded as the best possible compromise. Yet, we are aware that not everybody will agree on this and that this methodology should and will probably evolve in the coming months or years.

So, we will start with an overview of page weight, then proceed with a calculation of carbon emissions.

Page weight

Page weight represents the amount of data transferred to access the web page (based only on

739. <https://raidboxes.io/en/blog/wordpress/wordpress-plugin-co2/>

740. <https://sustainablewebdesign.org/calculating-digital-emissions/>

741. <https://marmelab.com/blog/2022/04/05/greenframe-compare.html>

HTTP requests). As explained before, it is here used as a proxy to calculate Greenhouse Gas Emissions.

It is recommended to keep this metric as low as possible. 1 MB should be a maximum when you get started but 500 kB should be your ultimate threshold⁷⁴².

For more on this, see the Page Weight chapter.

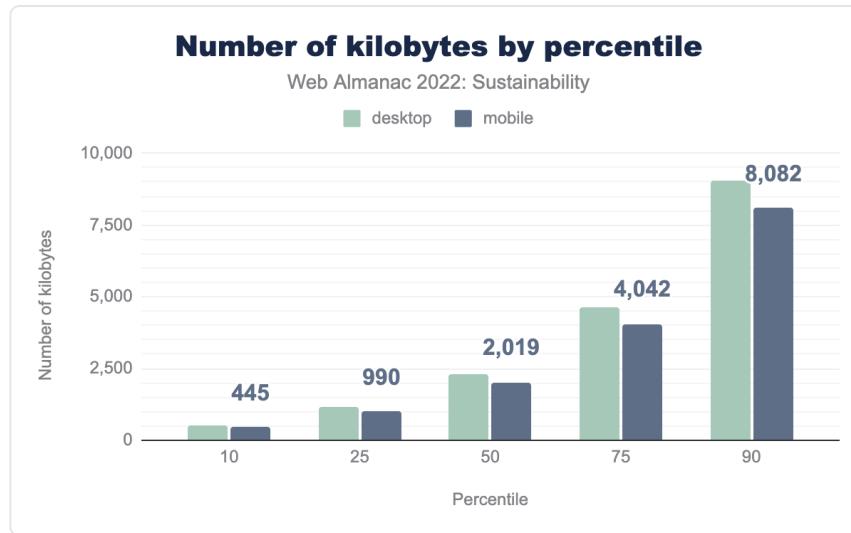


Figure 20.1. Number of kilobytes by percentile

Comparing page weights on mobile and desktop, we notice that the difference between them is small, which seems surprising. Media should be served in an appropriate size and format depending on the size of the screen. This might not be the case here.

At the 90th percentile, desktop pages were over 9 MB and mobile pages over 8 MB. We are far from the recommended threshold of 500 kB. To find pages under this threshold, we have to get to the 10th percentile. If we feel generous and aim for 1 MB, this can be found around the 25th percentile. There is still a long way to go...

Carbon emissions

Note: The notion of “carbon emissions” is a simplification since we are considering Greenhouse Gas Emissions, not only carbon emissions.

⁷⁴² <https://infrequently.org/2021/03/the-performance-inequality-gap/>

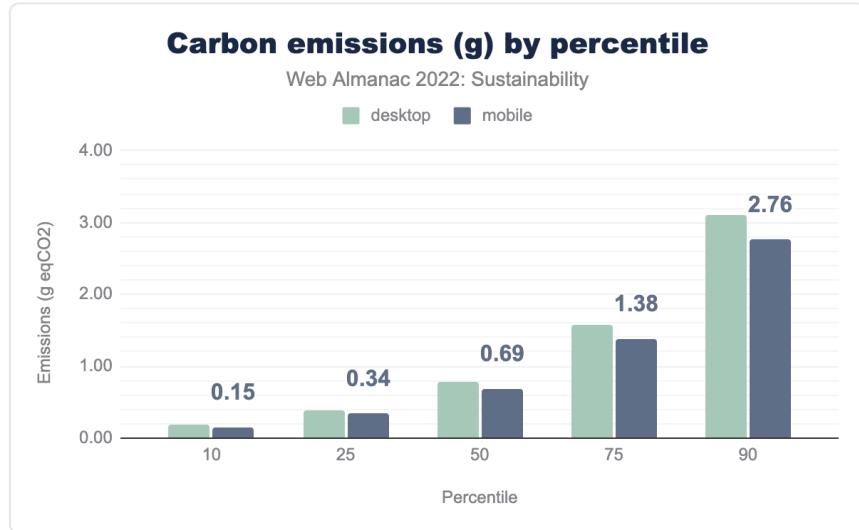


Figure 20.2. Carbon emissions (g) by percentile

The carbon emissions for websites are very close on mobile and desktop. They seem quite low on the 10th percentile (around 0.15 g eqCO₂, which would be equivalent to a little less than 1 meter with a thermal car⁷⁴³). They reach as much as 2.76 g eqCO₂ on the 90th percentile (a little more than 14 meters with a thermal car).

This doesn't seem like much but you should keep in mind that each website gets thousands or even millions of visitors each month (sometimes even more) and what you see in the graph is emissions for a single page visited once. The environmental impact each month for all websites adds up.

Now for an additional graph: emissions per percentile by type of content.

743. <https://datagir.ademe.fr/apps/mon-impact-transport/>

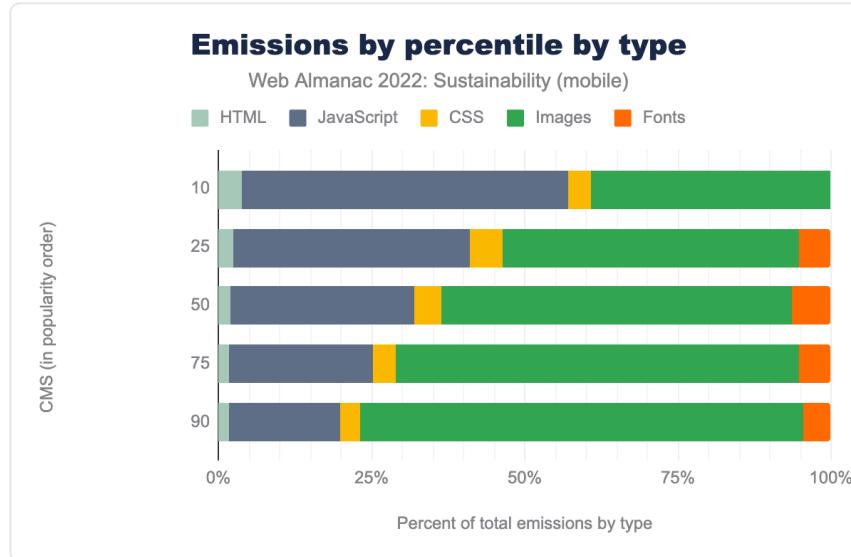


Figure 20.3. Percent of total emissions by percentile by type (mobile)

Images and JavaScript seem to be the more impactful but images get even more impactful as you go to upper percentiles. However, keep in mind that we only take data transfer into account to calculate carbon emissions. Processing JavaScript is usually more impactful than images. Once you have downloaded the JavaScript files, you still need to process them, sometimes leading to reloading your page or fetching other resources. Nonetheless, this graph underlines the necessity to reduce these impacts. It can be quite easy for images, as we will see later in this chapter. It gets more tricky with JavaScript, even though there are some easier technical optimizations such as minifying, compressing or reducing the need for it. More on that later too.

Number of requests

Requests are issued whenever a file is needed to load the page. As such, it helps represent the impact of the page on the network and servers, which is why it is sometimes used to calculate environmental impact. Analyzing the requests helps find possible optimizations, which we'll consider when discussing the various types of assets and external requests.

The number of requests should be kept to a minimum. Keeping an upper limit of no more than 25 is a fairly good start. But trackers and such often make that target difficult to reach.

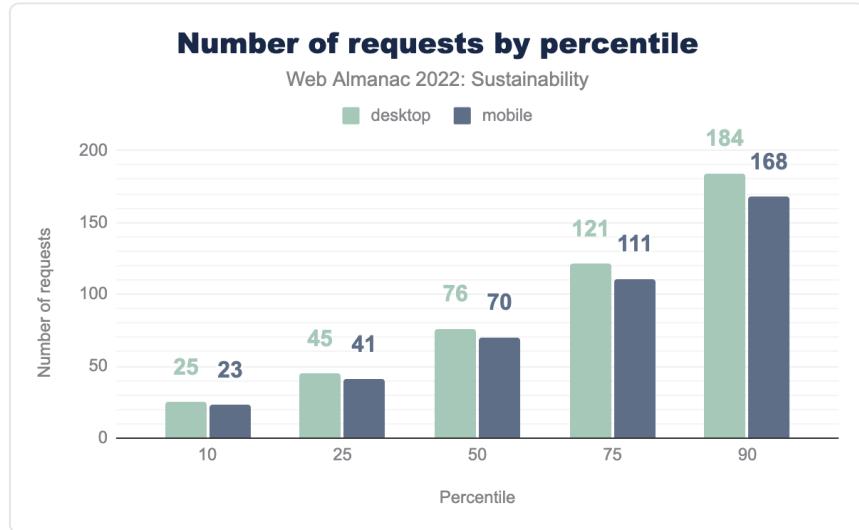


Figure 20.4. Number of requests by percentile

Comparing the number of requests of mobile and desktop, we once again find only a small difference, which shouldn't be the case. To find pages under the threshold of 25 HTTP requests, we need to get to the 10th percentile again.

So, which content type is to blame for this?

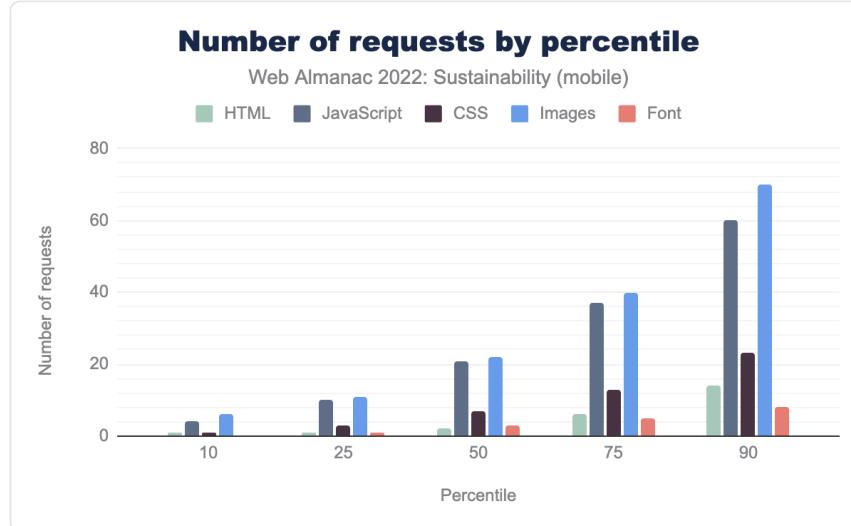


Figure 20.5. Number of requests by percentile by type on mobile

As usual, images are the main offenders but JavaScript is close behind.

There are almost as many HTTP requests for mobile and desktop versions, which shouldn't be the case. As with page weight, mobile pages should be kept as light as possible to take into account aging devices, erratic connectivity and expensive mobile data. Since many individuals still use the web in such suboptimal conditions, mobile web should account for this and do everything possible to be accessible for all.

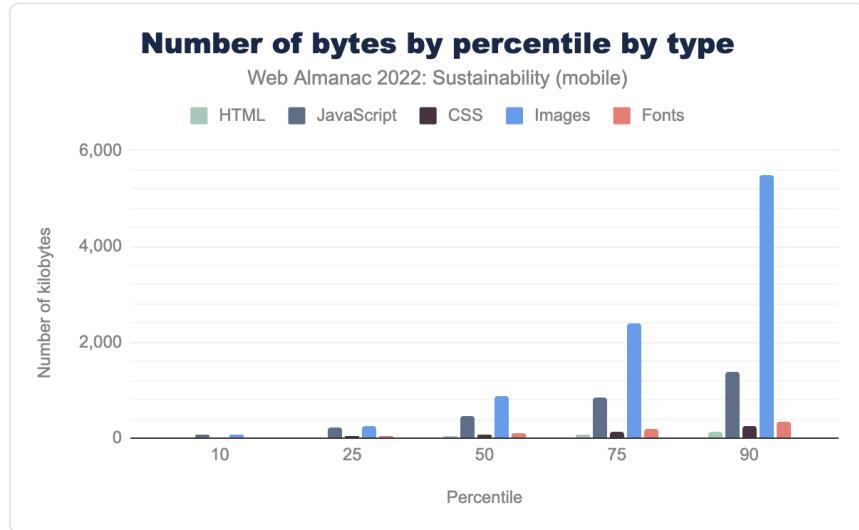


Figure 20.6. Number of bytes by percentile by type on mobile

There are almost as many HTTP requests for images and JavaScript but the overall weight is much higher for images. JavaScript being generally heavier to process than images, this is still bad news. Once again, the results are really close for mobile and desktop, even if it would seem to make sense to offer lighter experiences on mobile.

More sustainable hosting

Note: Here (and elsewhere), you should find mention of “Green Hosting”. This is kind of a shortcut since no hosting will be truly green, carbon neutral or other such things. We will focus here on how to use more sustainable hosting.

For the majority of this chapter, we focus on how changes in the quantity of resources like network, compute and storage affect the environmental impact of digital services—you might think of this as in terms of *consumption* as a lever for sustainability. However there are other levers too. You can’t efficiency your way to zero, and the same code, run on the same kind of server, but running on greener energy will have a lower environmental impact than otherwise. We can think of this lever as *intensity*.

Here there is some good news. Across the world, electricity grids we rely on are getting greener over time, driven by the falling costs of renewables and storage. 38% of our electricity came

from clean sources in 2022 (examples⁷⁴⁴ in the ember climate, and this chart⁷⁴⁵).

However, not every grid, and not every region a provider operates is equally green. Amazon's Web Service's customer carbon footprint tools⁷⁴⁶ show how running services in one region over another can provide a measurable difference in carbon emissions, as does the open source cloud carbon footprint⁷⁴⁷, for a growing number of providers. Elsewhere, the Green Web Foundation⁷⁴⁸ also provides an API for looking up any domain, for an estimate of how much the grid in that region is powered by fossil fuels.

You should however keep in mind that using renewable energy isn't enough to provide truly sustainable hosting. You should also check the PUE (Power Usage Effectiveness), WUE (Water Usage Effectiveness), how equipment is handled, etc. To further investigate this, you could check an article from Wholegrain Digital⁷⁴⁹ and the European Data Centres Code of Conduct⁷⁵⁰. More generally, beware of companies claiming to be carbon neutral (as stated by the french institute ADEME⁷⁵¹), especially since most of them don't include Scope 3 emissions. Also, as stated above, compensating your carbon emissions is not enough, you should reduce them too.

How many of the sites listed in the HTTP Archive run on green hosting?

An increasing number of technology firms are also taking steps to green all the electricity they buy to power their infrastructure. Companies like Microsoft and Salesforce already buy as much green energy as their server farms use on an annual basis, as do many other companies. We used the Green Web Foundation Dataset⁷⁵² to see how many organizations are "green hosts", taking similar steps⁷⁵³, and where they have shared evidence of powering all the energy they use on green energy, each year.

744. <https://ember-climate.org/insights/research/global-electricity-review-2022/>
745. https://public.flourish.studio/story/1176231#utm_source=showcase&utm_campaign=story/1176231
746. <https://aws.amazon.com/aws-cost-management/aws-customer-carbon-footprint-tool/>

747. <https://www.cloudcarbonfootprint.org/>

748. <https://www.thegreenwebfoundation.org/>

749. <https://www.wholegraindigital.com/blog/choose-a-green-web-host/>

750. <https://e3p.jrc.ec.europa.eu/communities/data-centres-code-conduct>

751. <https://presse.ademe.fr/2022/02/lademe-publie-un-avis-dexperts-sur-lutilisation-de-largement-de-neutralite-carbone-dans-les-communications.html>

752. <https://www.thegreenwebfoundation.org/green-web-datasets/>

753. <https://www.thegreenwebfoundation.org/what-we-accept-as-evidence-of-green-power/>

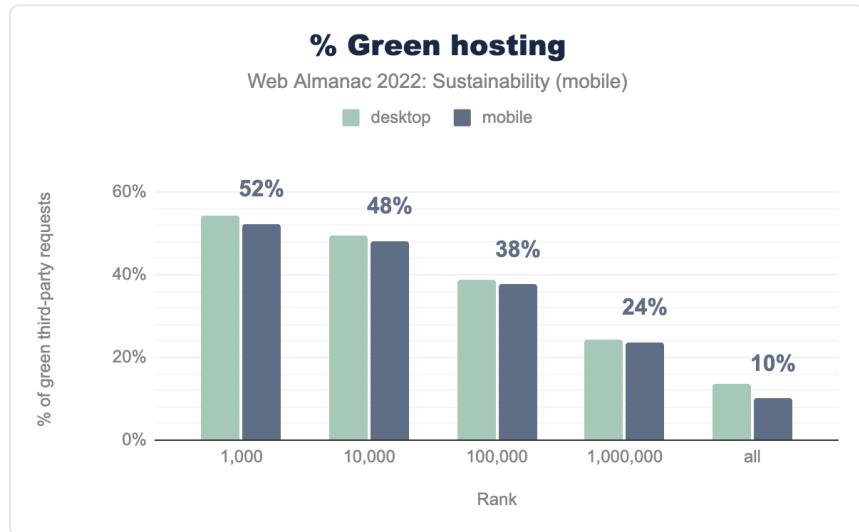


Figure 20.7. % Green hosting

Overall, only 10 percent of the measured websites rely on green hosting. This highlights that a lot could and should be done on both sides: websites opting for green hosting as well as hosting companies aiming for more sustainability.

Note: these figures for green domains are based on the information that is either shared directly with The Green Web Foundation, or placed in the public domain, where it is linked in API responses for their lookup service. See their explainer page⁷⁵⁴ for why a site might show up as “not green” when you think it should.

Reducing the environmental impact of websites

Best practices cannot work without measurements and vice versa. Now that we have a better representation of the environmental impacts of websites, let's see how to mitigate this.

Avoiding waste

One of the most obvious ways to reduce the impact of websites is to avoid all that is unnecessary.

754. <https://www.thegreenwebfoundation.org/support/why-does-my-website-show-up-as-grey-in-the-green-web-checker/>

- Reduce content and code waste: A great many websites and apps have unnecessary content and features. Be the voice that asks whether we need this page or that feature. Cliché stock images add nothing to most pages except weight. Removing images is one of the best ways to reduce page weight. Unused code should go too.
- Reduce processing. Byte for byte, JavaScript has a much bigger impact than HTML, CSS, images or text, because it causes energy-intensive processing to occur on the user's device. Try and choose the least energy intense implementation. Many web pages don't even need JavaScript. Always choose the lightest coding option.
- Choose the lightest communication option. Text is by far the most environmentally friendly way to communicate⁷⁵⁵. Video is by far the most energy-intense and unsustainable. As such, it should be used only if necessary, according to the needs of users. In these cases, video should be integrated as efficiently as possible.
- Design for long life. Design so that those using older machines and older operating systems can still use your website / app. Design so that you support people in holding onto their devices as long as possible. From a digital perspective, there's nothing better you could do for the environment.

Loading unused assets

You should only load assets that are needed to display the page and more particularly the portion of the page that is visible. This could be done through lazy-loading, critical CSS and patterns such as *Import on Interaction* and *Import on Visibility*. It could also involve loading images at the right size for the client device. We will mostly focus here on oversized fonts and unused code.

Fonts

For sustainability, it is recommended to stick to system fonts⁷⁵⁶. If you really need to use some custom fonts, there are some things to consider to avoid waste. Loading a font sometimes involves loading lots of characters and symbols that you might not need. For example, not all websites need Cyrillic characters but some fonts still include them natively. To check this, you can use tools such as wakamaifondue⁷⁵⁷. To reduce the size of your font files, you should aim for a WOFF2 format and using variable fonts⁷⁵⁸. You could also use subsets⁷⁵⁹ or use a tool such as

755. <https://www.google.com/url?q=https://text.npr.org/&sa=D&source=docs&ust=1662467318246688&usg=AOvVaw2K1v83mXXmEePMRoG6edxq>

756. <https://www.smashingmagazine.com/2015/11/using-system-ui-fonts-practical-guide/>

757. <https://wakamaifondue.com/>

758. <https://the-sustainable.dev/reduce-the-weight-of-your-typography-with-variable-fonts/>

759. <https://everythingfonts.com/subsetter>

subfont⁷⁶⁰. The Google Fonts API offers some clever options for all this⁷⁶¹. Regarding Google Fonts, you should still keep GDPR in mind⁷⁶².

For more on this topic, see the Fonts chapter. You can also find some documentation on web.dev⁷⁶³.

Unused CSS

Unused CSS is especially found when using CSS frameworks (Bootstrap and others). When doing so, you should keep in mind to remove unused CSS during your build phase. Chrome Dev Tools offer a Coverage tool to check on this⁷⁶⁴. Be careful: on many websites, all CSS and JavaScript are loaded on the first visit in order to cache them for further visits and exploration of the website. This is not necessarily a bad thing, but unused code is one of the drawbacks that you should keep in mind, especially because it might slow down further code processing.

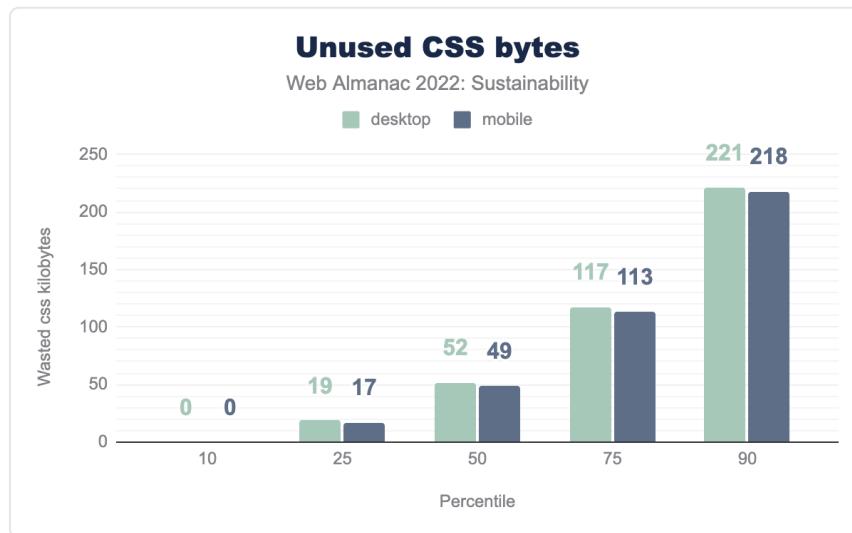


Figure 20.8. Unused CSS bytes

The good news is the 10 percentile websites load no unnecessary CSS. Unfortunately, it rises steadily on this graph, reaching more than 200 kB on the 90th percentile. Whether this for early caching reasons or otherwise, this should be checked. For sustainability, 200 kB of CSS is a big deal.

760. <https://github.com/Munter/subfont>

761. <https://web.dev/api-for-fast-beautiful-web-fonts/>

762. <https://revisit.io/urteil/urteil/lhm-20-01-2022-3-o-1749320/>

763. <https://web.dev/reduce-webfont-size/>

764. <https://developer.chrome.com/docs/devtools/coverage/>

Unused JavaScript

The amount of unused JavaScript can grow quickly when adding dependencies or using libraries such as jQuery. The Coverage tool from Chrome Dev Tools⁷⁶⁵ is a good way to check on this. As for CSS, this is sometimes part of a strategy to cache everything needed for further browsing. This should be balanced by the fact that unused JavaScript tends to result in longer processing. When possible, look for smaller alternatives⁷⁶⁶ with only the functionality that you need instead of loading the whole toolbox, hoping it will one day prove useful. Once upon a time, jQuery was the all-in-one solution that you found on almost every website. As of today, a lot of things can be handled with modern JavaScript⁷⁶⁷. Check your NPM dependencies and how they make your bundle bigger⁷⁶⁸.

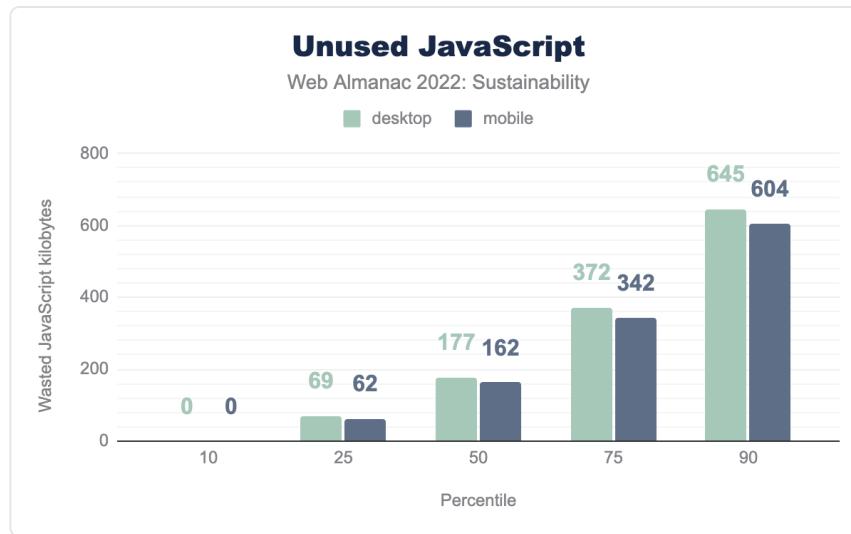


Figure 20.9. Unused JavaScript bytes

One again, the 10th percentile looks great with no unused JavaScript. However, this gets even worse than CSS for the upper percentiles, reaching more than 600 kB on the 90th percentile. This is already more than the ideal total page weight you should aim for.

Sustainable UX

Sustainable choices and optimizations can be made for a website before the development

765. <https://developer.chrome.com/docs/devtools/coverage/>

766. <http://microjs.com>

767. <https://youmightnotneedjquery.com/>

768. <https://bundlephobia.com/>

process during the early stages of design and prototyping. It is possible to design user experiences that prioritize efficient content from the beginning, even while creating experiences that engage users as active participants in sustainability practices. Contrary to some beliefs, all of this can be accomplished while still crafting beautiful, planet-centric web experiences.

While emissions associated with specific user experience tasks are difficult to quantify, some studies have estimated that consumer device use comprises as much as 52% of a product's overall digital footprint⁷⁶⁹. Therefore, it stands to reason that optimizing UX for sustainability can significantly reduce a product's environmental impact.

Designing for stakeholders

The most sustainable products are those that retain a clear picture of who stakeholders are, including the non-human ones. In doing so, we have products that take both a human-centered and planet-centered⁷⁷⁰ approach during the design process.

Engaging in streamlined practices such as stakeholder mapping⁷⁷¹ is helpful in identifying an ecosystem of stakeholders and their needs to set a path towards curating an inclusive experience for everyone involved. You'll be able to use this research to map out opportunities for designing a product that prevents unintended consequences from ignoring the needs of all stakeholders involved (human and non-human). This can be taken even further by leveraging intersecting touchpoints to build planet-centric innovation into your business model⁷⁷² that aligns with users' goals.

Optimizing user journeys

Crafting strategic user journeys that prioritize helping users achieve their goals in the least amount of steps is one method for creating a carbon-friendly web experience for your site. The less time a user spends navigating your product—overcoming obstacles, and completing their tasks—the less energy, data, and resources are used during their visit. Strategies in doing so often include being mindful in the use of images, videos, and visual assets to help drive user engagement and direction.

This also involves a “less is more” approach by engaging in wasteless design practices that only show content that is necessary to a user at a given time and emphasizing asset choices that deliver the same value. These are all things that aid the user in getting what they need faster by

769. <https://www.mightybytes.com/blog/where-do-digital-emissions-come-from/>

770. <https://planetcentricdesign.com/>

771. <https://www.mightybytes.com/blog/stakeholder-mapping/>

772. <https://www.mightybytes.com/blog/how-to-design-an-impact-business-model/>

avoiding the already surging attention economy⁷⁷³ on our devices for each page they visit. Continue to test and gather user feedback through prototyping and other methods to identify potential pain points that ensure you're creating the most optimal experience for your users.

Empowering sustainable behavior

There has been a rising popularity in incorporating a choice architecture into product features to nudge users into making sustainable choices relative to the environmental touchpoint of that product. Examples of this practice range anywhere from providing users more sustainable packaging options at checkout, displaying the most carbon-friendly product options, and even building reward systems or dashboards that visualize and incentivize these choices.

Aiding in this decision-making and offering these types of choice can not only help users interact with your website in more sustainable ways, but also help remove barriers of entry that help optimize user interactions. More recently, popular options can include accessibility features, language choices, device optimization, or the ever-popular dark mode that utilizes low-energy colors while promoting proper contrast.

These types of options help minimize potential pain points of users while enabling a custom experience that saves time, energy, and prevents frustration in a user. The power of choice can grow deeper into popular opt-out features such as the enabling and frequency of notifications—all choices that inevitably save resources when utilized, thus allowing users to customize both their experience and impact per visit.

Designing for circularity and end-of-Life

Analyzing and understanding the entire lifecycle of a digital product or service reveals opportunities to reduce waste and improve environmental impact over time. Defining and tracking clear, measurable success indicators can help guide this process.

- **Circularity:** Designing modular, easily replaceable or updatable components and focusing on continuous improvement can help you reduce technical debt⁷⁷⁴ and prolong the life of a digital product or service. This also saves time and reduces resource use.
- **End-of-Life:** Creating a clear retirement plan for your digital product or service will reduce the energy required to store and serve outdated or unused data. Plus, good data disposal practices also align with emerging data privacy laws that respect users' "right to be forgotten".

773. <https://econreview.berkeley.edu/paying-attention-the-attention-economy/>

774. <https://www.mightybytes.com/blog/technical-debt-agile-and-sustainability/>

Optimizing your content

Let's say you have optimized your website, making sure unneeded content and functionalities were removed. This was the moderation part (and usually the tough part). We can now look into the efficiency part: making sure everything you keep is as sustainable as possible.

In this section, we will look into images, videos and animations. More info on these in the Media chapter.

Image optimization

Images represent a huge part of requests and page weight. Let's see what we can do to mitigate this—in addition to avoiding stock images that bring you no additional information. As already mentioned, you should have already removed unnecessary images.

For a closer look on the relative benefits you can expect from possible technical optimizations, there is a post on the HTTP Archive that compares them⁷⁷⁵. Since you can more and more easily rely on native HTML (and sometimes CSS) for this, you should implement all of them.

Format (WebP/AVIF)

WebP is already widely supported⁷⁷⁶ and one of the best formats you can find for your images. Its compression is impressive and results in less data being transferred and processed. In addition to this, it enjoys wide support. AVIF should be even better but it might be wise to wait until it has reached a wider adoption from browsers⁷⁷⁷. Until then, just make sure you use the WebP format for your images. Your icons should be in optimized SVG⁷⁷⁸ and you could even include them directly in the HTML to avoid additional requests.

⁷⁷⁵. <https://discuss.httparchive.org/t/state-of-the-web-top-image-optimization-strategies/1367>

⁷⁷⁶. <https://caniuse.com/webp>

⁷⁷⁷. <https://caniuse.com/avif>

⁷⁷⁸. <https://jakearchibald.github.io/svgomg/>

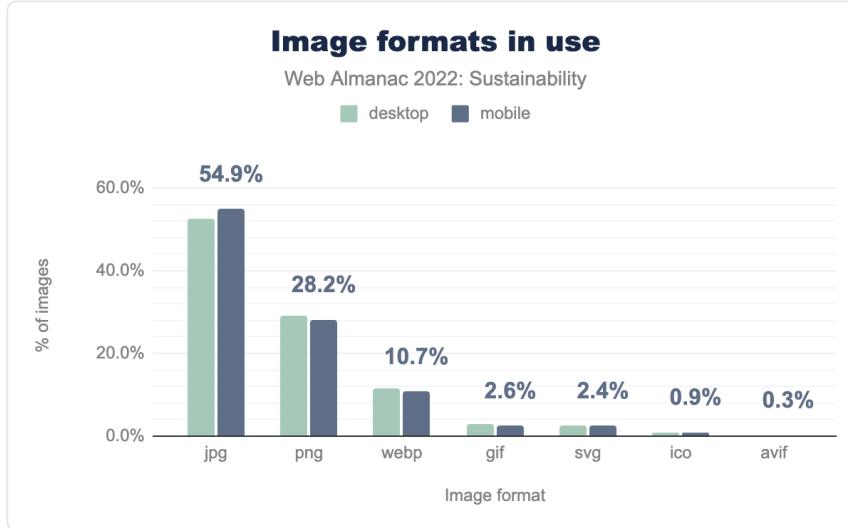


Figure 20.10. Image formats in use

As of today, only 10% of the websites use WebP, which is already better than last year⁷⁷⁹ but far from ideal. This could be a huge opportunity and help reduce the overall weight of images. AVIF is even further behind, only slightly over 0% but we can hope this figure will rise in the coming years.

Responsiveness, size, and quality

As a growing proportion of users browse the web on various devices (mostly smartphones but also game consoles, smart watches, tablets, etc), you should aim to deliver images of the right size—and weight—for each of them. After all, this is one of the major topics of responsive design and developers have lots of tools to automate this.

Also remember that you often don't need a quality of more than 85% since the human eye won't detect a difference above this. Reducing quality to 85% will help reduce the size of images.

⁷⁷⁹. <https://almanac.httparchive.org/en/2021/media#format-adoption>

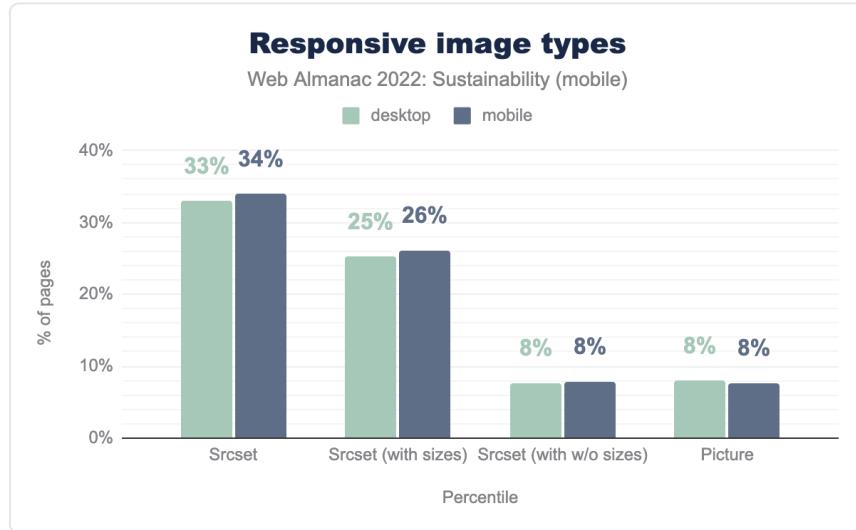


Figure 20.11. Responsive image types

Around 34% of the websites use the `srcset` attribute, which is a great way to integrate responsive images. The `<picture>` element works great too and is already found on 7% of the websites. Being optimistic, we could focus on the fact that responsive images are gaining ground each year, even if it's not used on a majority of websites. However, responsive design has been around for quite some time and this should be more widely spread.

Lazy-loading

An easy way to get a faster first load is to load images progressively: only load what you need when you need it. This is done through lazy-loading and most browsers now support this natively⁷⁸⁰. Not all users will scroll your page in its entirety so you should avoid loading images that might never be seen by the current user. As such, this is a quick win for sustainability and your users.

780. <https://caniuse.com/loading-lazy-attr>

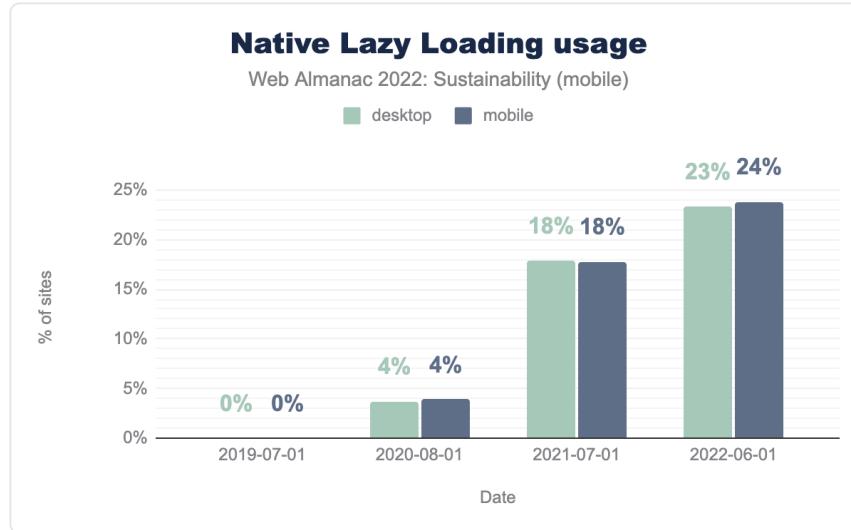


Figure 20.12. Native lazy loading usage

On this graph, we see that native lazy-loading has been more and more widely adopted since its implementation. Around one out of four websites use it. Some might still be using JavaScript libraries to implement this behavior and do not appear on this graph. Switching to native lazy-loading could be a great opportunity for them to slightly reduce requests and avoid some JavaScript processing.

A quick note on iframes: lazy-loading could also be natively applied to iframes, although, for sustainability reasons, you should consider avoiding iframes altogether. Most of the time, facade⁷⁸¹ is the good pattern for you, whether you want, for example, to include embedded videos or interactive maps. Directly including external content on your page has a bad habit of increasing the weight and requests of the page and often causes accessibility issues.

Video

Videos are some of the most impactful resources⁷⁸² you can include on a website. More info on these in the Media chapter. To integrate third-party videos, you should use facades⁷⁸³. On top of this, you should set them up wisely⁷⁸⁴. For instance, avoid preloading and autoplay. You could also learn how to quickly reduce the size of your videos⁷⁸⁵.

781. <https://web.dev/third-party-facades/>

782. <https://theshiftproject.org/en/article/unsustainable-use-online-video/>

783. <https://web.dev/third-party-facades/>

784. <https://www.smashingmagazine.com/2021/02/optimizing-video-size-quality/>

785. <https://theshiftproject.org/en/guide-reduce-weight-video-5-minutes/>

preload

Automatically preloading videos (or audio files) involves retrieving data that might not be useful for all users. On a page including such content and having many visitors, it can quickly add up. As such, preloading should be avoided and only done on user interaction.

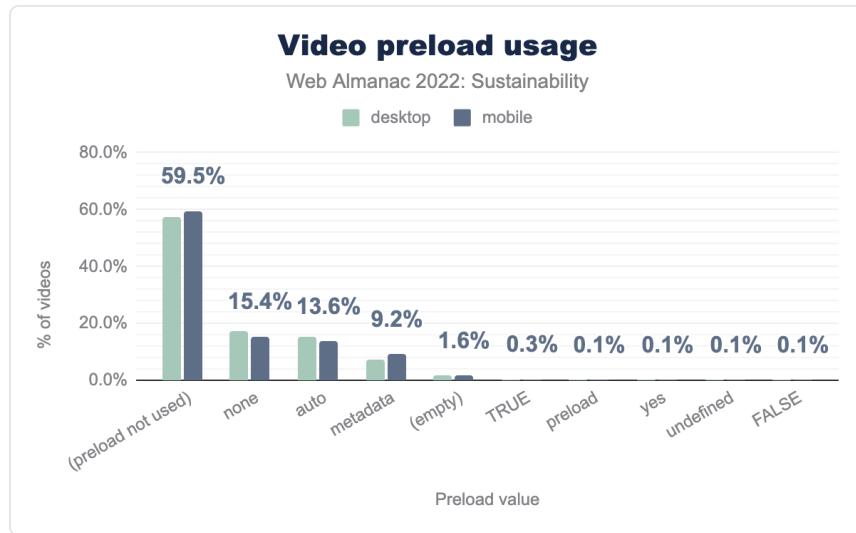


Figure 20.13. Video preload usage

Looking at this graph, one should keep in mind that the `preload` attribute only has 3 possible values: `none`, `auto` and `metadata` (default). Using the `preload` attribute with no value or with an erroneous value might be the same as using the `metadata` value. It still involves loading as much as 3% of the video to get these metadata and can thus be quite impactful.

`none` is still the best way to go for sustainability. But you have to keep in mind that this is only a hint for the browser. In the end, the browser has its own way of handling the preloading of the video and it might not fit with what you had in mind.

For more on this, you should check the article from Steve Souders (2013)⁷⁸⁶ and another one from web.dev (2017)⁷⁸⁷. Even if you can configure your browser or device to save data, video preload is something that browsers should handle more sustainable by default.

786. <https://www.stevesouders.com/blog/2013/04/12/html5-video-preload/>

787. <https://web.dev/fast-playback-with-preload/>

Autoplay

Most of the considerations we made on preload also apply with autoplay. In addition to the fact that it involves loading data and displaying content to users who might not be interested, it can cause accessibility issues. For some users, unsolicited moving pictures and/or sound might be bothersome and hinder their browsing experience.

Also, this attribute can override your `preload` setting since autoplaying requires loading (obviously).

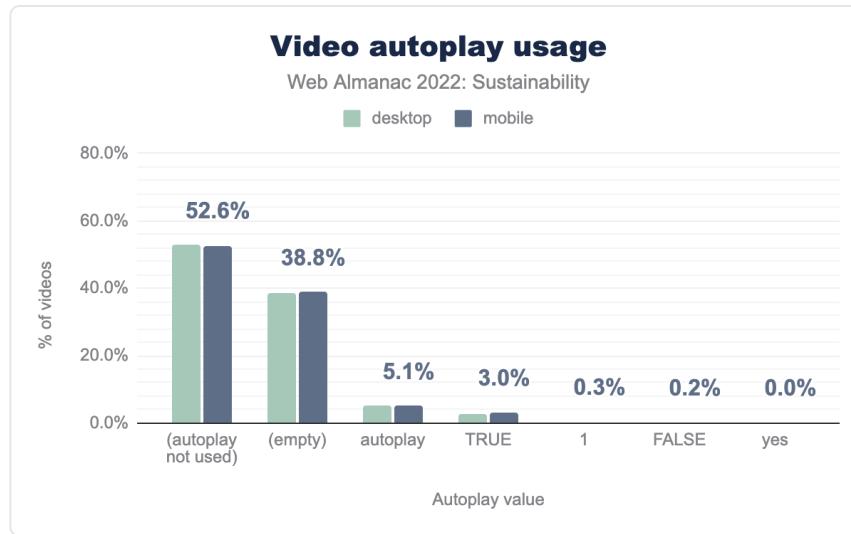


Figure 20.14. Video autoplay usage

More than half of the websites don't use autoplay, which is great. But this is a Boolean attribute so having it, even with an empty value (or wrong value), triggers autoplay. For all the reasons mentioned above, this should be avoided for both sustainability and accessibility.

Animations

For accessibility, moving and blinking parts should be avoided unless users have some control on them. Regarding sustainability, animations are costly: they tend to increase the battery discharge speed and CPU consumption—which might in the end reduce the autonomy of a smartphone. They also involve retrieving and running some code, which might delay rendering.

The (infamous) case of carousel is documented on these pages:

- Why you shouldn't use them⁷⁸⁸
- What can be done instead⁷⁸⁹

If you must use animations you should also avoid GIF⁷⁹⁰ or at least convert them to optimized videos, since animated GIFs can get really heavy.

Favicon and error pages

By default, your browser will look for a favicon upon arriving on a website. If it's missing, most servers will return a 404 error and the HTML for the 404 page of said website. So, some things to consider:

- Don't forget your favicon—and cache it!
- Don't forget to optimize the HTML of your 404 page to make it as light as possible or, even better, configure your server to make sure it only sends some text rather than the HTML of your 404 page.

For more details on all this, see this article from Matt Hobbs⁷⁹¹.

Optimizing external content

One of the great things about web development is that you can easily rely on external content: frameworks and libraries but also content. However, just because it is easy to implement doesn't make it useful or any less impactful. For each external element that you want to add, try to ponder whether it is really needed by the users. If so, then try to integrate it as efficiently as possible. And also keep in mind that each piece of content comes at a cost—requests and additional code, but also sometimes vulnerabilities or at least increasing the attack surface.

Third parties

Third-party requests account for 45% of all requests, with 94% of mobile websites having at least one identifiable third-party resource. This is not surprising, given that third-party code is often used to deliver complex functionality on web pages. It also serves as a quick fix for including cross-platform content onto a website.

⁷⁸⁸. <https://shouldiuseacarousel.com/>

⁷⁸⁹. <https://www.smashingmagazine.com/2022/04/designing-better-carousel-ux/>

⁷⁹⁰. <https://web.dev/efficient-animated-content/>

⁷⁹¹. <https://nooshu.com/blog/2020/08/25/you-should-be-testing-your-404-pages-web-performance/>

91%

Figure 20.15. Percent of third-party requests on mobile pages that are served from green hosting.

With third-party requests making up such a large portion of requests on the web, it is reassuring to see that the vast majority of these requests are being served from green hosting providers.

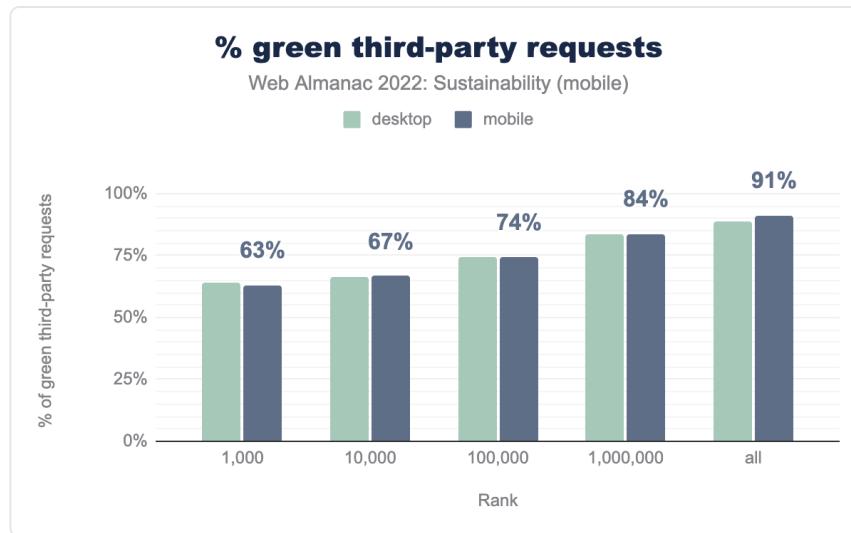


Figure 20.16. Percents of green third party requests

The chart above shows the percentage of third-party requests served from green hosting providers. It's interesting to note that the trend here is opposite to green website hosting. One might expect this to be the case, however, since the top five most requested third parties are all Google entities (fonts, analytics, accounts, tag manager, and ads). URLs associated with these entities are almost entirely listed as served from green hosting.

Making third-party requests more sustainable

As we have seen, most third-party requests are being served from green hosting. However, there is still room for improvement, especially for higher ranked sites. If you are interested in the sustainability of third-party services used on your website, Are my third parties green?⁷⁹² is

792. <https://aremythirdpartiesgreen.com/>

an online testing tool, directory, and API that can help you get started. *For the sake of transparency, it should be noted that this tool was created by one of the chapter authors.*

Beyond hosting, we should also be considering the impact of data transfer for third parties. While providers of third-party services make it relatively easy to integrate their content on another website, that doesn't mean it is always optimized to reduce the amount of data being transferred. For example, the Third Parties chapter of the 2022 Almanac uncovered that:

Google fonts are the most-popular third party on mobile devices being used by 62.6% of all websites. The CSS they provide is not minified. The data shows the average page which has Google Fonts could save 13.3 KB from minifying it.

In the case of fonts, self-hosting and subsetting are two techniques that, when combined, can help reduce this waste. However, most third-parties come in the form of scripts. These incur a cost when transferring data over the network, but also utilize processing power on the end-user's device. For these, we can reduce their impact by loading them "just in time".

This pattern is known as *Import on Interaction*, which sees static facades used in place of interactive content when the page first loads. The content then gets requested and loaded just before the user interacts with an element. This can result in less data being transferred initially, and also reduces the processing required when viewing the page—especially if the script is never requested.

Implementing technical optimizations

We have just seen a lot about the sustainability of the content of websites—even external content. This leaves us with all the other technical optimizations. There is a lot to be done here too and most of this could and should be automated. Once again, this might intersect with some other chapters from the Web Almanac but the idea is to offer you a whole chapter about sustainability and how to make websites more sustainable.

The web performance experts have done a lot in the field of technical optimizations so there is a lot to learn from them. Just keep in mind that some of their best practices don't necessarily make your websites more sustainable. However, making things lighter and simpler is great for sustainability AND performance—not to mention accessibility.

JavaScript

There is a lot to be said about JavaScript and how it helped the web grow (and how it sometimes slows it down). Let's stick to some quick wins: easy to implement and great for sustainability. If you want to learn more about all this, you should check the JavaScript chapter.

Minification

Minifying JavaScript involves removing unnecessary characters for the browser, making your files lighter.

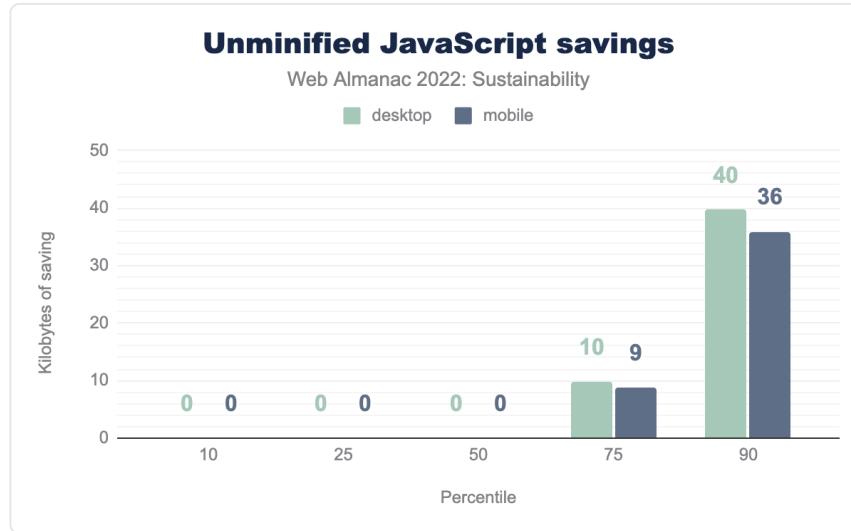


Figure 20.17. Unminified JavaScript savings

On this graph, we notice that most websites already do a great job at minifying JavaScript and that benefits from minifying are not so big. However, why not do it since it's easy to implement and always beneficial?

Including as little as possible directly in HTML

Inlining code is bad practice, even more for sustainability. Making your HTML heavier to load and process is not something you want. Inlining JavaScript might also make it sometimes more difficult to optimize (and maintain).

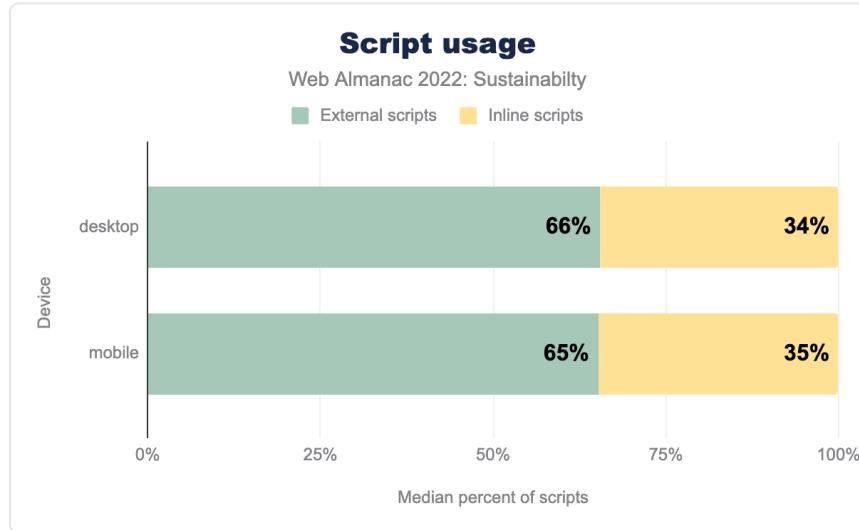


Figure 20.18. Script usage

Almost one third of websites inline JavaScript. This is also something you see a lot with CMS.

CSS

CSS could be a great lever for sustainability, especially if you want to limit the number of images on your website or create some animations as mentioned earlier in this chapter. You can find documentation on how to write efficient CSS—and should definitely look for this—but we'll stick to standard optimizations that should be implemented everywhere. If you want to learn more about all this, see the CSS chapter.

Minification

As with CSS, minifying JavaScript involves removing unnecessary characters for the browser, making your files lighter.

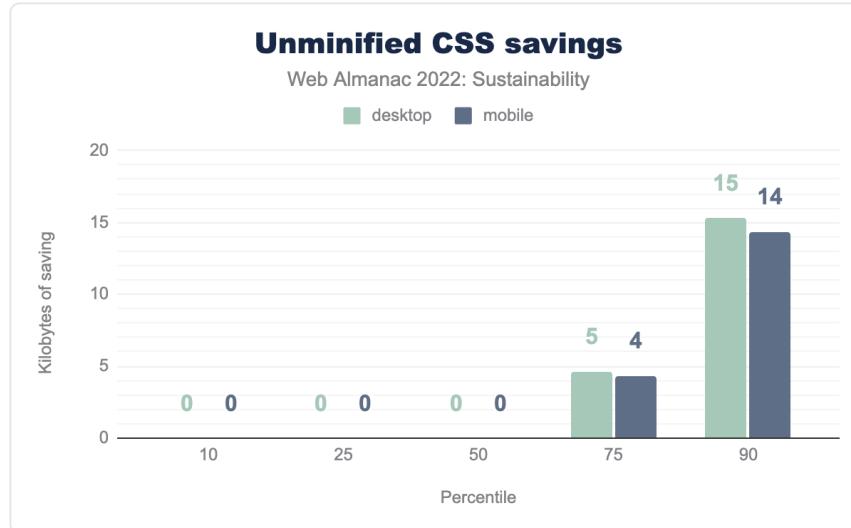


Figure 20.19. Unminified CSS savings

Unminified CSS is absent from most of the websites and the potential gains appear really light. However, it is still beneficial to minify CSS and this should be implemented on all websites.

Including as little as possible directly in HTML

As with JavaScript, inlining CSS could prove detrimental for the size of your HTML file and for the performance of your website. This is often found on websites built with CM and those relying on the Critical CSS method⁷⁹³.

⁷⁹³. <https://web.dev/extract-critical-css/>

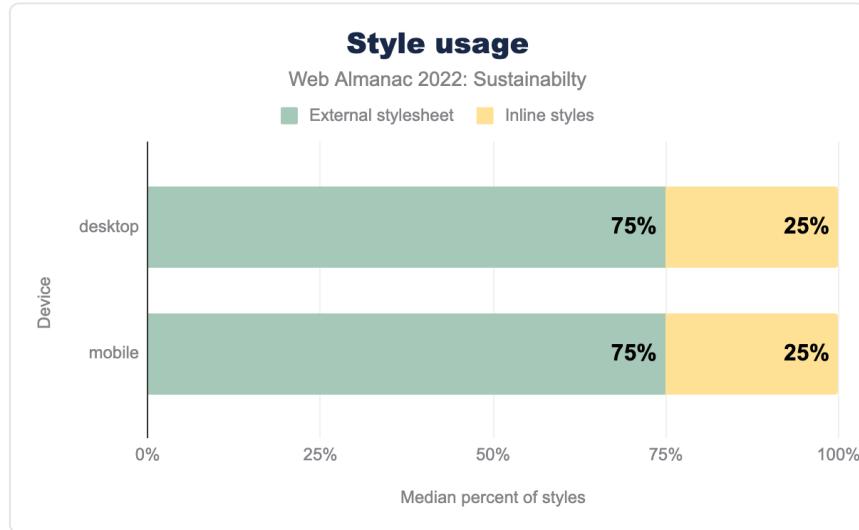


Figure 20.20. Style usage

On this graph, it appears that a quarter of websites still use inline CSS. For sustainability reasons, this should be avoided.

CDN

Implementing a CDN can help make your website more sustainable. It helps get your assets as close to your users as possible and sometimes automatically helps optimize them.

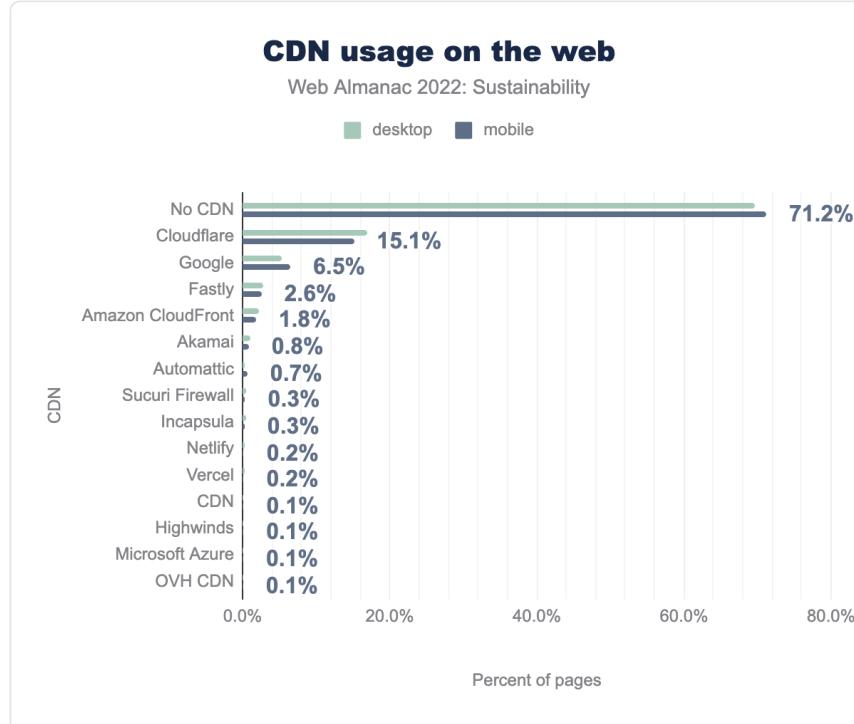


Figure 20.21. Cdn usage on the web

Despite these obvious benefits, more than 70% of websites still don't use a CDN.

Text compression

Compressing the text assets for a website⁷⁹⁴ could require some (easy) server-side configuration. Text files such as HTML, JavaScript and CSS are then compressed (in Brotli or Gzip format), which can easily make them lighter.

⁷⁹⁴. <https://web.dev/uses-text-compression/>

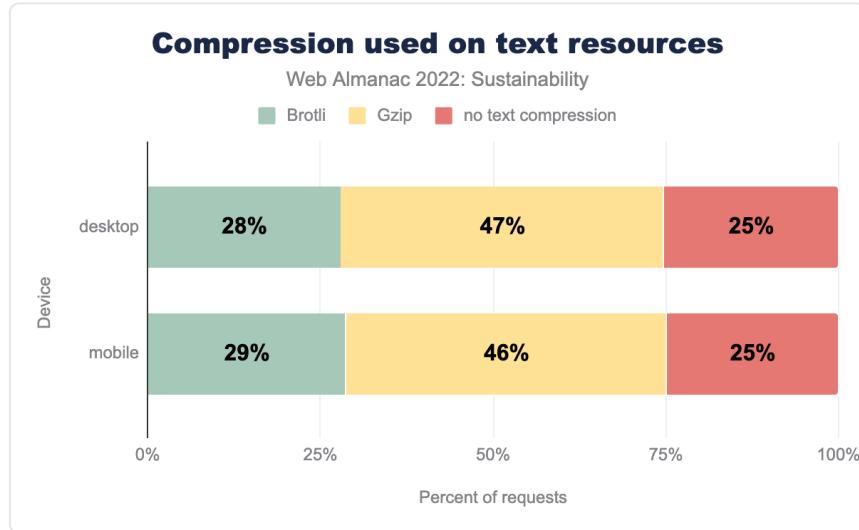


Figure 20.22. Compression used on text resources

However, a quarter of websites still don't implement text compression. Gzip is unanimously supported so feel free to use it.

Caching

Caching⁷⁹⁵ is one of the killer features of browsers but not always easy to implement flawlessly. Caching is great for sustainability since it prevents browsers from requesting all resources every time.

795. <https://web.dev/uses-long-cache-ttl/>

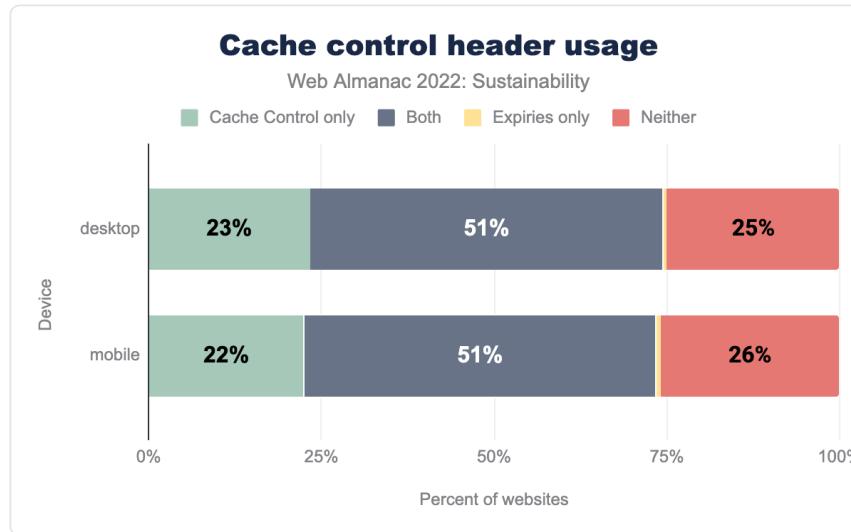


Figure 20.23. Cache control header usage

On this page, we see that more than a quarter of websites don't use caching at all. This is a huge loss for sustainability and performance and—for obvious reasons, users.

SEO and sustainability

Similar to accessibility, estimating emissions specifically related to search engine optimization efforts is challenging. However, SEO does have significant sustainability implications:

- Keyword research, a foundation of SEO, helps authors align the content they create with specific target user needs.
- Structured data helps search engines better understand page content, allowing them to serve more relevant information in results.
- Search-optimized content is typically easier to find, quicker to skim due to its formatting, and clearly written, making it easier to understand.
- Analyzing content performance over time (bounce rates, scroll depth, etc.) helps authors improve the content they have published so it better serves user needs (and improves search results). Depending on tools used for analysis, privacy implications may arise.

Collectively, these efforts reduce the amount of time a user spends searching for information that is relevant to their needs. This reduces their energy use.

Conversely, search engines often reward long-form content like tutorials and guides, which can use more bandwidth (and energy) than listicles or other short-form content. As with many sustainability-related concepts, the key is finding the right balance between creating useful and compelling content and optimizing for performance and efficiency.

Many years ago, Google noted that the energy required for a single search could power a 60W light bulb for 17 seconds. In 2022, more research is needed to quantify the specific environmental impact of search engine optimization. Still, the sustainability implications of good SEO work are clear: optimized content reduces end user energy consumption (not to mention frustration).

Sustainable data and content management

As noted above, structured data helps search engines better understand web pages to produce more relevant results. However, our collective relationship with data has sustainability implications beyond SEO. For example:

- Unused, duplicated, outdated, or incomplete data and poorly managed content use up server space, cause errors for users, and require energy to host and maintain.
- Regular content audits⁷⁹⁶ and a clear content governance plan⁷⁹⁷ can help you measure content performance and prune outdated or underperforming content over time to keep your website lean, efficient, and well-organized.
- While third-party services may only inject a small snippet of code into an individual web page, the data they collect can be very resource-intensive. A single digital ad, for instance, can produce as much as 323 tons of CO₂e⁷⁹⁸.
- Data tool makers—like marketing automation, email marketing, and CRM systems—often focus their product management efforts on collecting data rather than optimizing it. In fact, some of these platforms charge for data use, making their business models at odds with sustainability principles.
- Similarly, many organizations don't have clear data disposal policies nor do they train their teams on effective data management. This is not only a sustainability issue but a privacy and security issue as well.

796. <https://www.mightybytes.com/blog/how-to-run-a-content-audit/>

797. <https://www.mightybytes.com/blog/content-governance/>

798. [https://www.businessinsider.com/making/net-zero/possible/the-hidden-impact-of-digital-ads-2022-7](https://www.businessinsider.com/making-net-zero-possible-the-hidden-impact-of-digital-ads-2022-7)

These are just several examples from a long list of sustainability issues associated with poorly managed content and data. Organizations should regularly audit their content and data management practices to improve efficiency and reduce resource use⁷⁹⁹.

Popular frameworks, platforms, and CMSs

Online platforms and CMS tools help lower the barrier to entry for those wishing to publish or do business on the web. Likewise, development frameworks and site generators allow those who build for the web to get started on projects faster, and enable them to take advantage of defaults and solutions that solve common development problems.

The charts below show the median page weight of the top five most popular eCommerce platforms, CMS tools, and site generator tools.

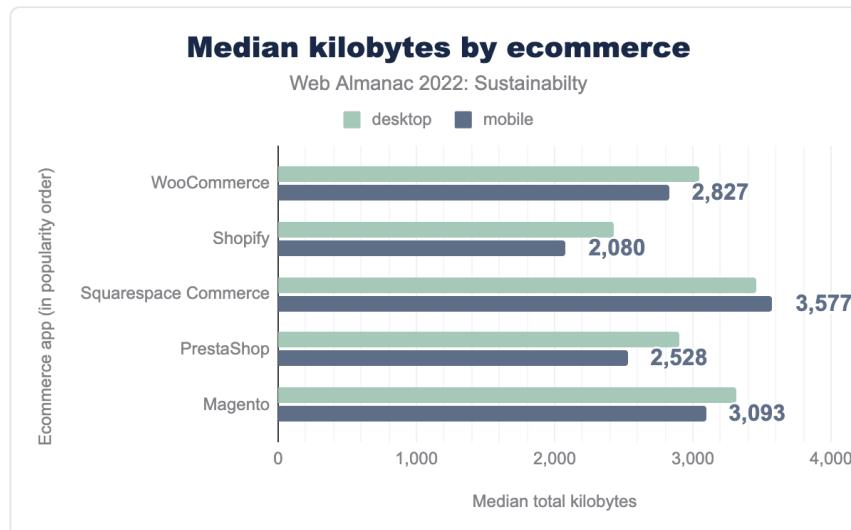


Figure 20.24. Median kilobytes by ecommerce

799. <https://www.mightybytes.com/blog/design-a-sustainable-data-strategy/>

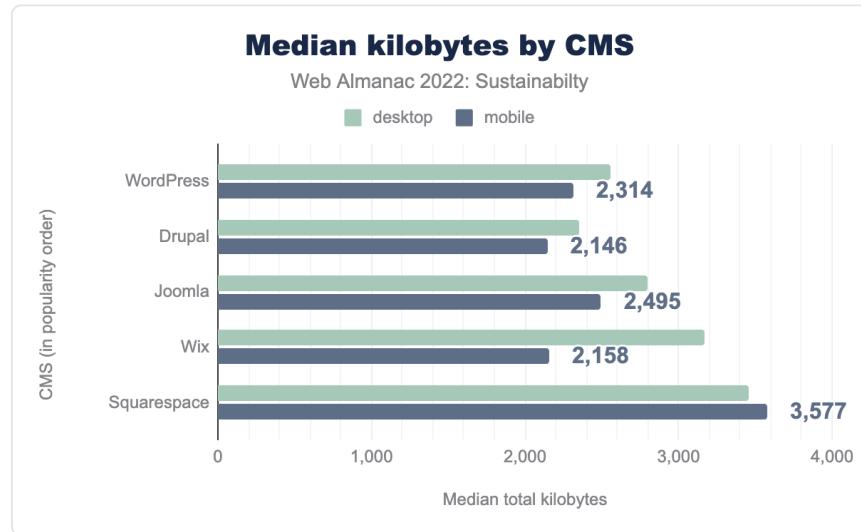


Figure 20.25. Median kilobytes by CMS

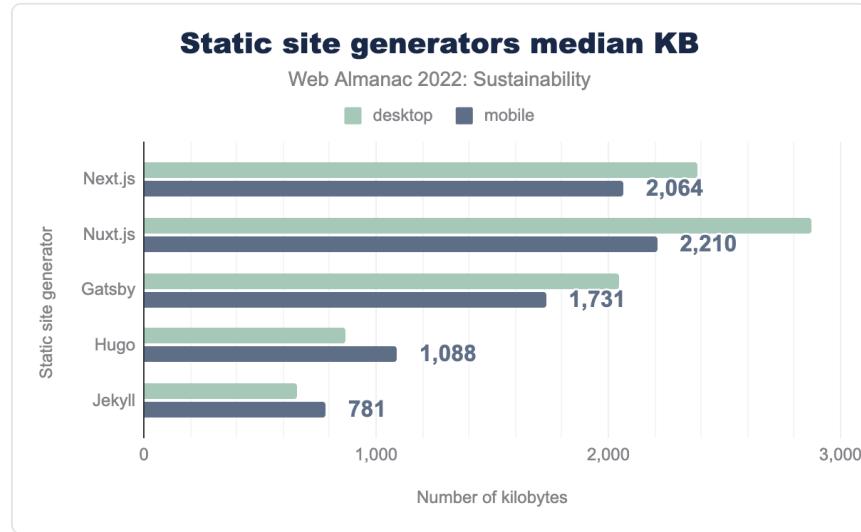


Figure 20.26. Static site generators median KB

Of interest here is that all but three of the platforms/tools listed have a median mobile page weight that is less than the overall median (2,019 KB). These are all in the static site generator category, and especially in the case of Hugo and Jekyll, it can likely be attributed to the kinds of websites these tools are used to create—namely mostly blog and textual content, with much less reliance on JavaScript. It should also be noted that SSGs are often used with performance

in mind, which makes them more likely to be further optimized than the average website using a CMS only for commodity reasons.

Another area of interest when looking across the three segments is that some show a bigger gap between desktop and mobile page size. On closer inspection, this seems to be largely down to image optimizations that some platforms seem to be applying for mobile devices. To highlight this, let's look at the CMS category, where Wix shows a big difference between desktop and mobile size compared to the other popular platforms.

CMS	Device	HTML	JavaScript	CSS	Image	Fonts
WordPress	Desktop	40	521	117	1,202	166
WordPress	Mobile	37	481	115	1,100	137
Drupal	Desktop	23	416	68	1,279	114
Drupal	Mobile	23	406	66	1,158	92
Joomla	Desktop	26	452	86	1,690	104
Joomla	Mobile	22	401	83	1,504	82
Wix	Desktop	123	1,318	86	647	197
Wix	Mobile	118	1,215	9	290	148
Squarespace	Desktop	27	997	89	1,623	214
Squarespace	Mobile	27	990	89	1,790	202

Figure 20.27. Median kilobytes by CMS, device, and resource type

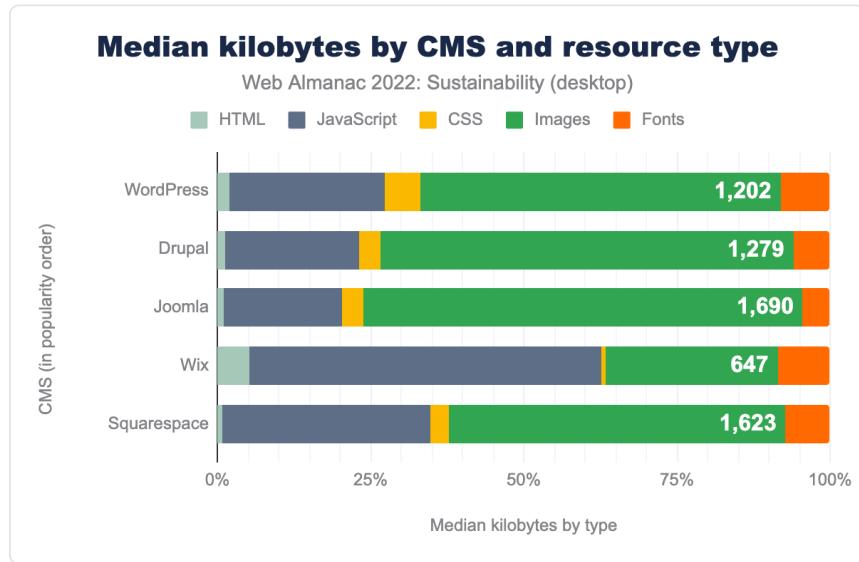


Figure 20.28. Median kilobytes by cms and resource type (desktop)

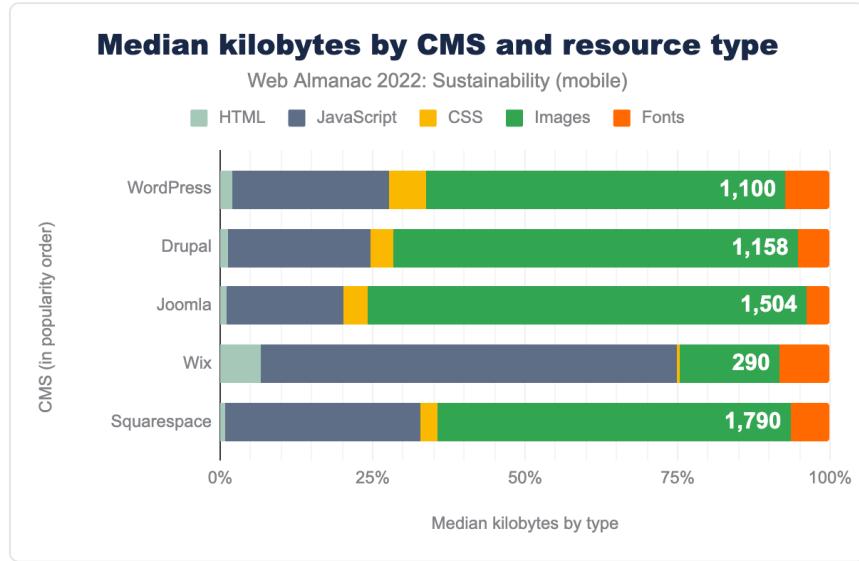


Figure 20.29. Median kilobytes by cms and resource type (mobile)

The table and charts above highlights that Wix, as part of their platform, appear to be applying much more aggressive mobile image optimizations. A similar pattern is seen in the site generator segment, especially when looking at frameworks like Next.js and Nuxt.js.

This seemingly small insight helps to capture the important role that platforms and frameworks can play in helping deliver more sustainable websites. By applying sensible defaults, platform developers and framework authors can help developers leverage their tools to make sites that are *green by default*⁸⁰⁰.

Conclusion

This is the first Web Almanac chapter ever on sustainability and quite a symbolic year to do so with all the droughts, heat waves and other climatic events all over the world. There's definitely something wrong and it's getting harder and harder to look the other way. The web plays a part in this and it's a priority to help everyone understand its environmental impacts. Given all the accessible data from HTTP Archive, this chapter is a unique occasion to gather metrics and take a step back to look at the state of the web regarding sustainability.

Based on available metrics, we see that some best practices are already being adopted and slowly spreading. However, there is still a lot to be done. Some of these actions are easy to implement but could still prove really beneficial. Also, both best practices and measures—preferably on real devices—are essential for continuous sustainability improvement.

It's up to everyone to gain awareness about sustainability, discover the best practices and implement them. It's essential to spread the word about all this and generate discussion. As with accessibility, this concerns all of us and we have everything we need to make the web more sustainable.

Most of what we have seen here shows what developers can do to make websites more sustainable. But we can and must go further. We have to reach for sobriety through design. Project managers need to make sustainability a priority and make sure that this is not something that can always be handled later. Companies need to think about making their business models more sustainable.

With this chapter, we hope to make you more aware about web sustainability, help you understand how sustainable websites are today and give you some tools and clues to handle this topic and spread the word.

Actions you can take

The sustainability of websites needs to be taken into account. As of today, there is still a lot to be done. If needed, you should start with some of the resources recommended above to gain

800. <https://screenspan.net/blog/green-by-default/>

awareness on this subject—and spread the word.

To get started on an existing website, you can:

- Optimize images (WebP + lazy-loading + responsiveness + cache + quality) and make sure this is done automatically
- Avoid implementing videos. If they are necessary, do not autoplay nor preload them
- Look for a more sustainable hosting

You should then:

- Clean up your 3rd-parties.
- Optimize your CSS and JavaScript starting with the easy technical optimizations and automating them.
- Review the design to make your page more sober (less visual content, less animations, etc) and streamline the user journey(s)

Making your websites more sustainable is part of continuous improvement. Not everything can—or should—be done at once. Rely on best practices and measurements to make sure you’re going the right way. Whether you’re working on an existing website or creating a new one from scratch, keep everyone in the team involved or at least aware of this topic.

Some of your users would love to know that your website is more sustainable and how you achieved it. And all of them would benefit from this.

With special thanks to Tom Greenwood⁸⁰¹, Hannah Smith⁸⁰², Eugenia Zegisova⁸⁰³, Rick Viscomi⁸⁰⁴ and all the other wonderful people who made this chapter possible.

⁸⁰¹. <https://www.wholegraindigital.com/team/tom-greenwood/>

⁸⁰². <https://twitter.com/hanopcan>

⁸⁰³. <https://twitter.com/jeveniazi>

⁸⁰⁴. https://twitter.com/rick_viscomi

Authors



Laurent Devernay

Twitter: @ldevernay GitHub: ldevernay Website: <https://ldevernay.github.io/>

Laurent Devernay is a Digital Sobriety Expert for Greenspector⁸⁰⁵. You can find him blogging on his own⁸⁰⁶ or for this company⁸⁰⁷ but almost always about web sustainability. Which makes him either an enthusiast or a monomaniac.



Gerry McGovern

Twitter: @gerrymcgovernireland

Gerry has published eight books. His latest, *World Wide Waste*⁸⁰⁸, examines the impact digital is having on the environment. He developed Top Tasks⁸⁰⁹, a research method which helps identify what truly matters to people.



Tim Frick

Twitter: @timfrick GitHub: timfrick Website: <https://www.mightybytes.com/>

Tim Frick⁸¹⁰ started his digital agency Mightybytes⁸¹¹ in 1998 to help nonprofits, social enterprises, and purpose-driven companies solve problems, amplify their impact, and drive measurable business results. Mightybytes is a Certified B Corp⁸¹² that uses business for good. Certified B Corps meet the highest verified standards of social and environmental performance, transparency, and accountability. Tim is the author of four books, including *Designing for Sustainability: A Guide to Building Greener Digital Products and Services*⁸¹³. A seasoned public speaker, he regularly presents at conferences and offers workshops on sustainable design, measuring impact, and problem solving in the digital economy. He has also served on the boards of several nonprofit organizations, including Climate Ride⁸¹⁴, B Local Illinois⁸¹⁵, and the Alliance for the Great Lakes⁸¹⁶.

805. <https://greenspector.com/en/home/>
806. <https://ldevernay.github.io/>
807. <https://greenspector.com/en/blog-2/>
808. <https://gerrymcgovern.com/books/world-wide-waste/>
809. <https://gerrymcgovern.com/books/top-tasks-a-how-to-guide/>
810. <https://www.mightybytes.com/teammember/tim-frick/>
811. <https://www.mightybytes.com/>
812. <https://www.mightybytes.com/b-corporation/>
813. <https://www.oreilly.com/library/view/designing-for-sustainability/9781491935767/>
814. <https://www.climatehive.org/>
815. <https://www.illinoisbcorps.org/>
816. <https://greatlakes.org/>

Part IV Chapter 21

Page Weight



Written by Jamie Indigo and Dave Smart

Reviewed by Chris Steele

Analyzed by Danielle Rohe

Introduction

We shall show that over the last ten years, mobile page weight has increased 594%. In this time, we've seen performance-enhancing technologies emerge to buffer the impact. More recent performance measuring methodologies like Core Web Vitals eschew page weight as a factor.

While the Google-driven Core Web Vitals initiative shifts the perception of performance to how quickly above-the-fold content becomes visible and usable, the large network payloads are still correlated to long load times.

Many building websites have the luxury of high-speed desktop connections and don't experience the limited, and often expensive mobile network access.

According to the International Telecommunication Union's Global Connectivity Report⁸¹⁷, 66% of global households have internet access. In low-income countries, only 22% have access compared to 91% in high-income countries. Often in rural areas of developing countries, only

817. <https://www.itu.int/itu-d/reports/statistics/2022/05/30/gcr-chapter-2/>

3G is available despite 4G being the minimal connection for meaningful connectivity.

In more than half of the world's low- and middle-income countries, residents pay more than 2% of their average monthly income for 1GB of mobile broadband data⁸¹⁸.

Page weight still matters. Whether you're experiencing a weak network connection at an inopportune moment or live in a market where access to the internet is charged by the megabyte, inflated page weight decreases the availability of information.

What is page weight?

Page weight refers to the byte size of a web page. As it's no longer 1994, a web page is rarely only the HTML of the URL viewed in a browser's address bar. Rather, a web page as viewed and rendered in a browser uses specific elements and assets.

This is why page weight is inclusive of all assets used to create the page. These include:

- The HTML that makes up the page itself
- Images and other media (video, audio, etc) embedded into the page
- Cascading Style Sheets (CSS) used for styling the page
- JavaScript to provide interactivity
- Third-Party resource containing one or more of the above.

Each resource adds to the byte of the page as well as computational resources involved in the transmission, processing, and rendering of the page. Some resources like scripts have additional overhead in the form of CPU usage as each must be downloaded, parsed, compiled, and executed.

As page weight balloons, valiant efforts and methods have been proposed to mitigate the impact. Still, the machinations of page weight's complex relationship to resource allocation is invisible to most users.

Let's inspect closer the three impacts of page weight for resources: storage, transmission, and rendering.

⁸¹⁸. <https://digital-world.itu.int/ministerial-roundtable-cutting-the-cost-can-affordable-access-accelerate-digital-transformation/>

Storage

Ultimately every resource that goes to make up a web page needs to be stored somewhere. For a website, that often means multiple places, all of which bring their own costs and overheads.

Storage on the web server tends to be relatively low cost when on disk storage, and relatively scalable. It's perhaps a little more expensive if the web server is serving from memory. These resources might also be duplicated on intermediate caches and CDNs.

That's just the source, ultimately the resources will need to be stored in some form on the client side too, where storage is potentially more limited, especially for mobile devices. Serving huge, bloated files may well fill the user's cache, pushing out other useful resources.

Unoptimized images, at resolutions better suited to print media at multiple megabytes, and huge video files can still be routinely found.

A lot of this can be mitigated by picking the right formats and codecs for media and being mindful of size as well as quality. Services like Squoosh⁸¹⁹ are great for getting the most out of your images at the smallest possible size, and there are specialist image CDNs⁸²⁰ that can automate much of this.

Media, although regularly the weightiest elements, are not the only place savings can be made—text resources can be compressed and minified too.

Putting your resources on a diet has never been easier!

Transmission

When you visit a web page for the first time, all the resources that the page requests need to be transmitted across the internet from the server to your device.

That can be across a superfast, high monthly usage broadband connection, but it could be from a slow, expensive, capped mobile connection, or even satellite.

The larger the page weight, the longer this will take. It can also be more expensive for those users with lower-capped data plans.

There are optimizations with things like `preconnect`, `preload`, and Priority Hints⁸²¹ that can manage the order things are loaded and help perceived load times, but ultimately the resources still need to be transmitted and received, and the best optimization of all is serving smaller

⁸¹⁹ <https://squoosh.app/>

⁸²⁰ <https://web.dev/image-cdns/>

⁸²¹ <https://web.dev/priority-hints/>

resources.

Rendering

The fetching of resources is just the first step of getting a website painted on-screen and viewable by the user. To do that, a web browser needs to render the page, using all the relevant resources.

Page weight plays an important role here—in a number of ways. First, if the transmission is slow—because the files are large—the longer it is until the browser gets the chance to even start working on these to render.

Large files have an effect even after they are received over the network. Larger files take more processing power and memory to be read, processed, and rendered. This in turn leads to longer delays from some—or even all—of the content from being displayed to the user. The longer the delay, the more likely a user is to abandon the page and seek the information from a more responsive site.

JavaScript is of special concern here, as it not only needs to be downloaded: it also needs to be parsed and executed.

Huge files can have an ongoing performance penalty, even if your user does wait for them to be initially downloaded. They can eat up all client-side resources available to the browser, leading to slow performance, or even crashing the browser entirely.

Modern, high-end smartphones, laptops, and tablets might have the power to handle these large files without noticeable performance issues but older or lower-powered devices may well struggle. These are also often the same devices with slow and potentially expensive mobile connections—leading to a ‘double penalty’ for people who potentially need simple, reliable access the most.

What are we shipping?

Prior to the introduction of HTML 2.0 in 1995, page weights were predictable and manageable. The only asset to weigh was the HTML. RFC1866⁸²² introduced the `` tag. Page weight dramatically increased once images could be included on web pages.

Further versions of the HTML spec added even more features that could add weight—like external CSS, allowing consistent styling across pages.

⁸²². <https://www.rfc-editor.org/rfc/rfc1866>

1996 saw the first emergence of JavaScript, 2005 brought XHR, and 2006 saw the birth of libraries like jQuery, followed by frameworks like Angular, React, Vue, and many more, fully unleashing the leviathan in waiting: JavaScript.

The swell of page weight brought a proliferation of file types intended to improve performance while retaining functionalities. Examining them by asset type will highlight the trade-offs.

Images

Images are the poster child for balance between performance-enhancing technologies and asset byte size. These static files serve as resources to build out and render web pages. The increasingly visual nature of the web ensures the media type will retain its title as most ubiquitous asset.

Older formats like PNG, JPEG, and GIF enjoy their legacy of broader “historical” browser support. Performance-focused file type WebP⁸²³ gained significant browser support and is now available to 97% of global users⁸²⁴.

To delve into the findings and implications of image use on the web, refer to the Media chapter.

JavaScript

JavaScript is the most popular client-side scripting language on the web. We see that 98% of websites use it to create interactive online content—and other sources agree it’s that high⁸²⁵. While magnificent when used in moderation, the intoxicating allure of JavaScript can also lead to serious performance, search engine optimization, and user experience issues.

Refer to JavaScript chapter for detailed insights into the internet’s favorite monkey paw.

Third-party services

A page’s weight is not limited to the assets hosted on the origin. Third-party resources requested by the page pile onto the weight in the form of analytics, chatbots, forms, embeds, analytics, A/B testing tools, and data collection.

According to the Third Parties chapter, 94% of all websites on mobile devices use at least one third-party resource! Each of these contributes to the byte size of page weight.

823. <https://developers.google.com/speed/webp>

824. <https://caniuse.com/webp>

825. <https://w3techs.com/technologies/details/cp-javascript>

Other assets

The basic building blocks of the web have remained relatively unchanged for over 25 years. As the richness of web experiences increases, so does the use of fonts and videos.

These increases are in step with the other file weight increases with a notable expectation to the 100th percentile.

110 MB

Figure 21.1. The largest font use on mobile page.

In 2021, the 100th percentile of mobile sites used 20,452 kilobytes of font files. In 2022, these outliers swelled to 110 megabytes. This 540% growth was not seen in the year-over-year comparison for desktop which sat at 66,257 kilobytes in 2021 and 68,285 in 2022.

However, the 100th percentile—while fun to investigate—will always show the worst of the web. At the 90th percentile, the mobile font weight was less extreme—though still large—401 kilobytes.

More insights into the typographical nature of the web can be found in the Fonts chapter.

Page weight by the numbers

Now we know what we are primarily interested in when considering page weight, let's dive into the details.

Requests volume

It's not only the total number of bytes requested—the number of requests made to create a page can affect the performance of a page. We, therefore, consider this as part of page weight.



Figure 21.2. Distribution of requests.

The median page (50th percentile) makes 76 requests for desktop page loads, and 70 on mobile page loads. At all percentiles the difference between desktop and mobile is minimal.

Last year's median desktop request was 74, so no significant difference over last year.

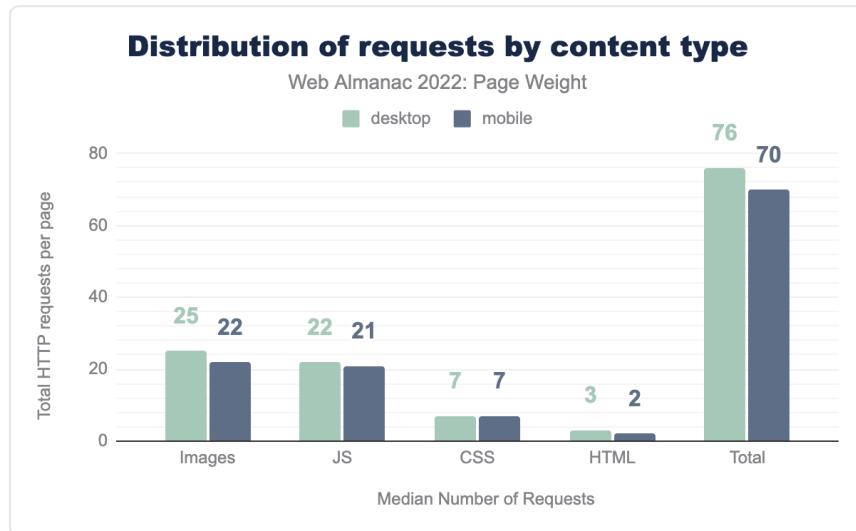


Figure 21.3. Median number of requests by content type.

Breaking down requests by type shows that images are the leading resource requests, with the median page requesting 25 images for desktop page loads; 22 for mobile. This is nearly identical to last year's 25 for desktop; 23 for mobile.

JavaScript is the next largest in request count, 22 requests for desktop page loads, 21 for mobile, again a very close match to 2021, where there were 21 for desktop and 20 for mobile.

In general, there's little difference between desktop and mobile, other than images being slightly lower on mobile—perhaps attributable to lazy-loading not firing on smaller initial viewports.



Figure 21.4. Page weight distribution by percentile.

At the 50th percentile, desktop pages were over 2 MB, mobile pages just under that. By the 90th percentile, this has grown to a nearly 9.0 MB for desktop, and nearly 8.0 MB for mobile.

Overall page weight is remarkably close when looking at what is served desktop versus mobile user-agents, although the gap grows slightly in the higher percentile (larger) pages. Given that mobile devices tend to have fewer local resources and more constrained network capabilities, this is concerning.

678 MB

Figure 21.5. The weight of the largest desktop page

At the 100th percentile, the largest pages we detected, desktop users were faced with eye-watering 678 MB pages, and mobile users 390 MB.

Let's dig a little deeper into what is making up these large sizes.

Request bytes



Figure 21.6. Median page weight over time.

Looking at the median page weight over time, it remains clear that the trend remains disappointingly consistent, with the median weight only growing over time.

594%

Figure 21.7. Growth in mobile page weight over 10 years

In the 10-year period between June 2012 to June 2022, the median page weight increased by 221%, or 1.6 MB, for desktop page loads, 594%, or 1.7 MB for mobile page loads

Year on year, (June 2022 versus June 2021) desktop increased from 2,121 KB to 2,315 KB on desktop, 1,912 KB to 2,020 KB on desktop.

Content type and file formats

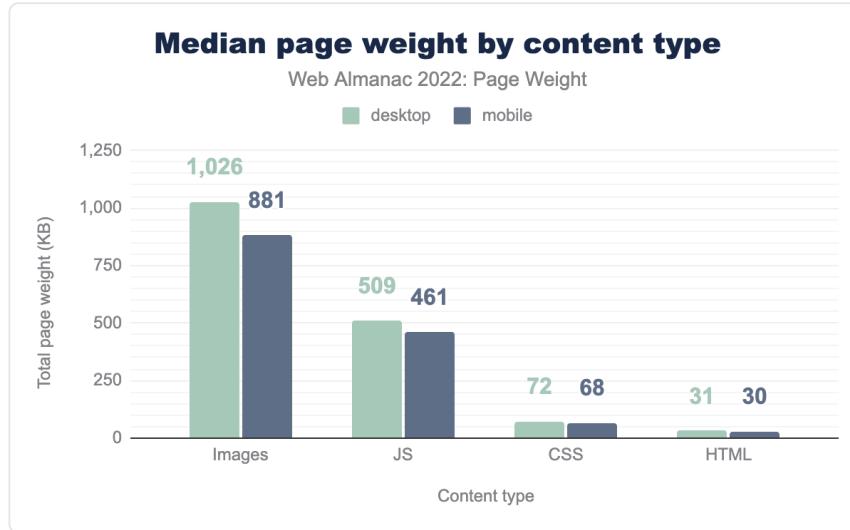


Figure 21.8. Median page weight by content type.

A look at the median weight of the most common resource content types making up the weight of pages shows images are the largest contributor, at 1,026 KB for desktop pages; 811 KB for mobile. JavaScript is the next largest contributor for both desktop and mobile page loads.

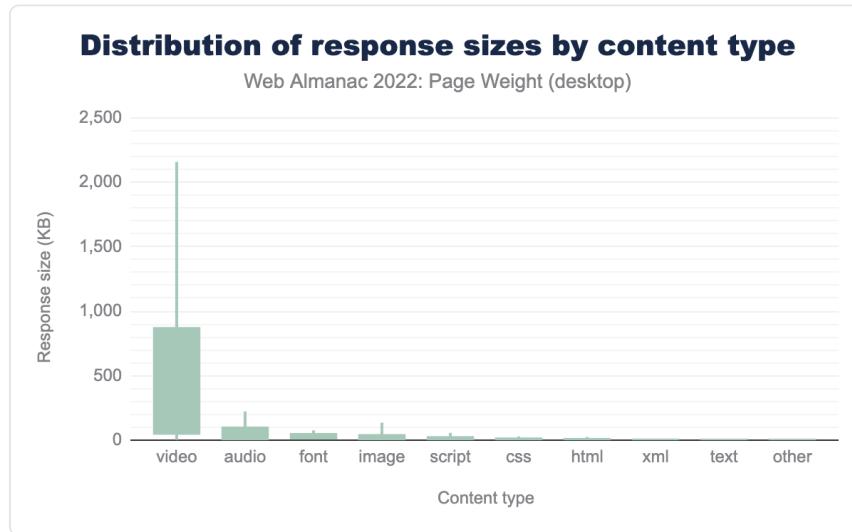


Figure 21.9. Distribution of response sizes by content type.

Whilst images are overall the biggest contributors to page weight across the internet, the biggest contributors in sheer size per request are video, audio, and fonts. At the 90th percentile, video requests weigh in at 2,158 KB, four times larger than all the other 90th percentile types combined.

Like images, there are a number of opportunities with more modern formats, and better encoding, sizing, and compression that could help slim this down. But it's worth noting that video by its nature tends to be weightier, and there's a balance between size and acceptable quality that needs to be struck. For more information, see the video section of the Media chapter.

Median response size by content type

Web Almanac 2022: Page Weight

desktop mobile

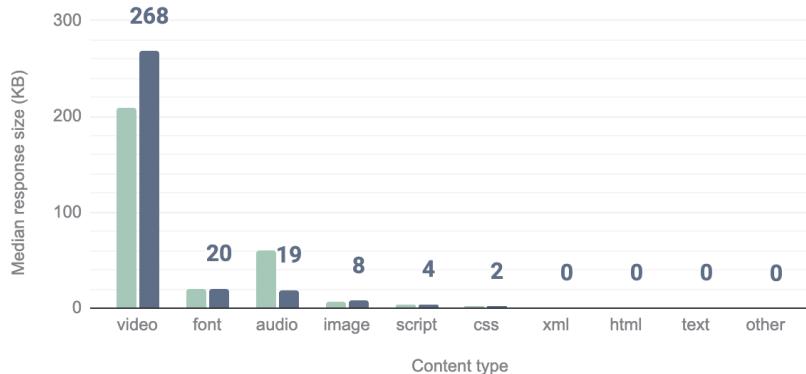


Figure 21.10. Median response size by content type.

Looking at the median response size for each content type, it's perhaps surprising to see that video content is larger at 268 KB on mobile page loads than desktop ones, at 208 KB. It's quite surprising that the median weight of fonts is higher than images, over double at 20 KB versus 8 KB on mobile.

Median response size by format

Web Almanac 2022: Page Weight

desktop mobile

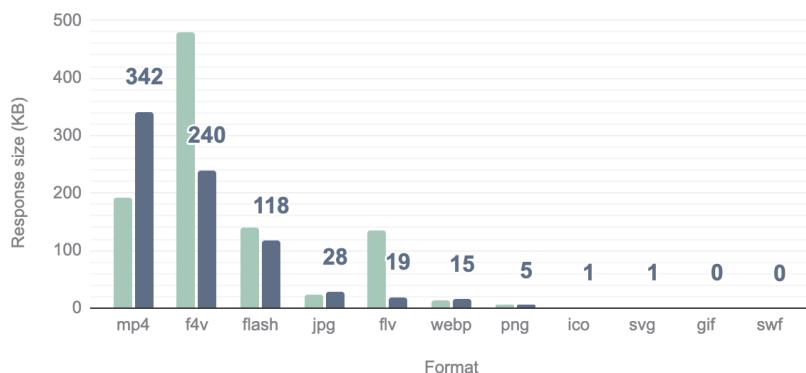


Figure 21.11. Median response size by format.

Focusing on file formats, it's disappointing to see f4v, flash, and flv adding significant weight to pages, the flash player plugin was discontinued in 2021, and removed from major browsers like Chrome⁸²⁶, meaning these bytes are most likely entirely wasted.

Image bytes

Since the inception of the Web Almanac images have represented the largest percentage of page weight by bytes, so it's worth seeing what formats we are using for them.

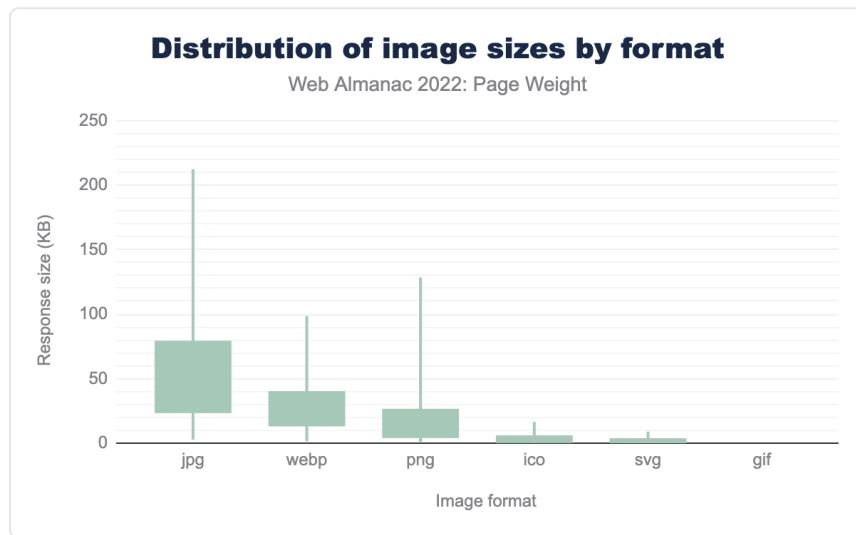


Figure 21.12. Distribution of image sizes by format.

Distribution of image sizes by formats shows us that JPG, WebP, and PNG file formats retain their 2021 status as top sources of image weight.

826. <https://support.google.com/chrome/answer/6258784>

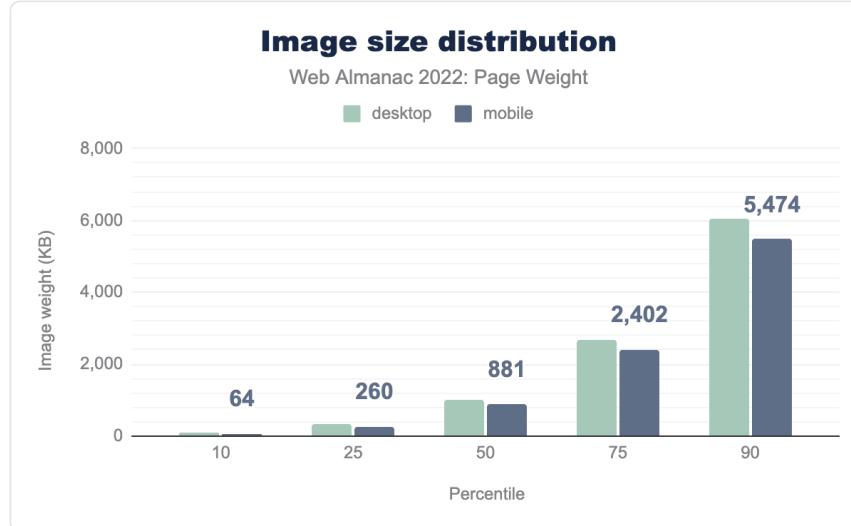


Figure 21.13. Image size distribution.

The median desktop image weight for 2022 was 1,026 kilobytes, a mere 44 kilobyte increase from 2021. Mobile barely shifted at 881 kilobytes.

The year-over-year consistency is only disrupted by the extremities of the 100th percentile. The largest desktop page contained 672MB of images, a significant increase from the hefty max of 186MB in 2021. Similarly, the mobile 100th percentile saw a 959% increase to 385MB.

Video bytes

According to the media chapter of the mobile web, 5% of mobile pages include the `<video>` element. This information aligns with the 100th percentile of other file type in overall page weight (as video files are grouped in the set). Pages bringing video experiences take on a corresponding increase in weight.

MP4s, which represent 51.5% of videos on the web, also represent the capacity for largest response size. At the 50th percentile, mp4 response sizes come in at 534 kilobytes.

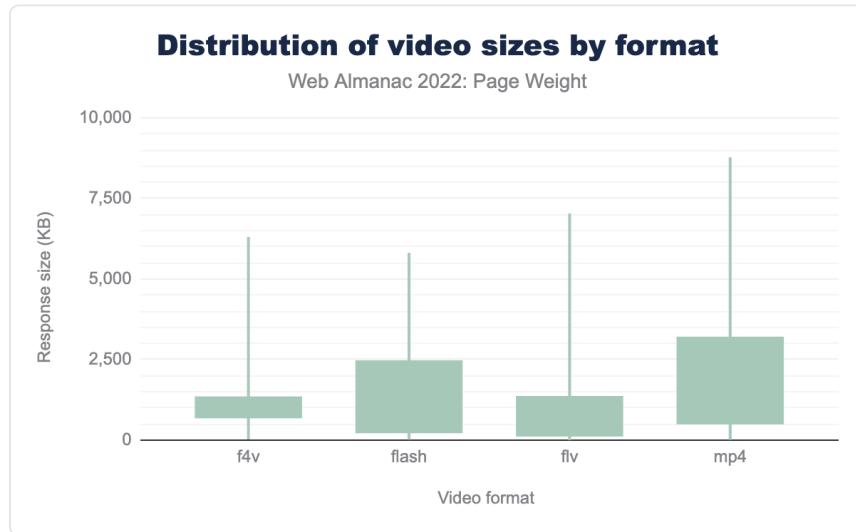


Figure 21.14. Image size distribution.

Adoption rates of byte-saving technologies

So what can we do about all those bytes we're sending? Well obviously we could just stop sending them, but obviously they are being sent for a reason—you'd hope! So let's look at how to keep the content, but send it in more efficient ways than just stuffing kilobytes and kilobytes down the pipes.

Facades for videos & other embeds

Videos and other interactive embeds can massively increase the overall weight of a page. Videos by their nature can be large in terms of bytes, but other content—like social media embeds used to embed a tweet for example—can bring in a substantial amount of JavaScript and other data to enable these to become interactive.

A good design pattern is the use of facades, a form of lazy-loading. This is basically showing a graphical representation of the element, and not loading the full thing in until required. For example, for a YouTube video, the initial load could be just the poster image for the video, an approach taken by the popular `lite-youtube-embed`⁸²⁷ library, which changes to the actual full YouTube embed on click. Alternatively, it can even behave more like traditional image lazy-

827. <https://github.com/paulirish/lite-youtube-embed>

loading and change when in or near the viewport.

Whilst there are drawbacks to this approach, as detailed in the [web.dev article on third-party facades](#)⁸²⁸ the benefits to the user in terms of data sent over the wire are clear, they only need to pay that cost if they want to watch the video, or interact with the live chat app.

In practice, adoption here is hard to track. Lighthouse offers a test where it looks at the use of a limited set of third-party resources, and if these are requested, points out there might be a facade available.

If a site is successfully using a facade, these resources would not be requested, and therefore isn't something lighthouse could test for.

The facade pattern doesn't need to be limited to third-party resources either—although these do come with the additional downside of additional lookups and connections—the approach can work well for large self-hosted resources too.

Analysis showed that there are a number of sites where Lighthouse was able to detect that a facade might be beneficial:

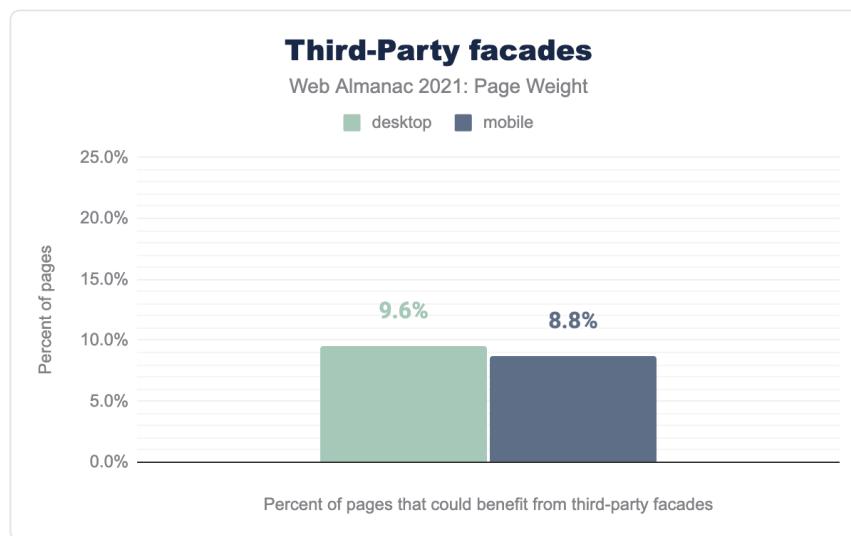


Figure 21.15. Third-party facades.

9.6% of desktop pages tested could have benefited from implementing a facade, a slightly better 8.8% for mobile visits.

⁸²⁸. <https://web.dev/third-party-facades/>

Compression

Compressing resources before serving them to the client can save bytes that have to be sent across the network, and with fewer bytes. In theory, and usually in practice, this makes for faster loads.

For text, non-media, files like HTML, CSS, JavaScript, JSON, or SVG, as well as for ttf and ico files, HTTP compression is a powerful tool, using gzip or Brotli compression to squeeze down file size. Media like images and video tend not to see any benefit as they are already compressed.

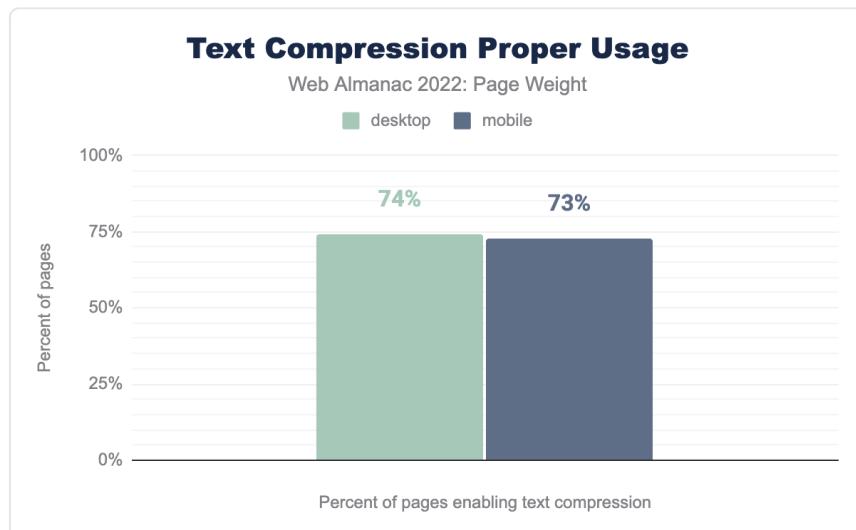


Figure 21.16. Text compression proper usage.

We detected that 74% of page loads on Desktop loads, and a slightly lower 73% of Mobile page loads.

The slightly lower proportion of mobile usage is a disappointing result because they are more likely to have slower and more expensive network connections.

Ultimately compression doesn't reduce the whole impact of page weight, because these resources have to be decompressed on the client before they are used.

It's not an entirely free process, either. There is processing overhead on the server to compress—although this might be cacheable for static resources—and likewise a cost on the client side to decompress these resources before use.

It's about tradeoffs and tackling the worst bottleneck, which is often the network. Compression techniques are remarkably efficient, and the net benefit is usually worth it.

Minification

Minification⁸²⁹ helps to reduce the overall size of text-based resources by removing unnecessary characters, like whitespace, code comments, and other things that play no part in how a browser interprets and uses these resources.

CSS and JavaScript are great candidates for minification, and we looked at both, using Lighthouse's test for these resources.

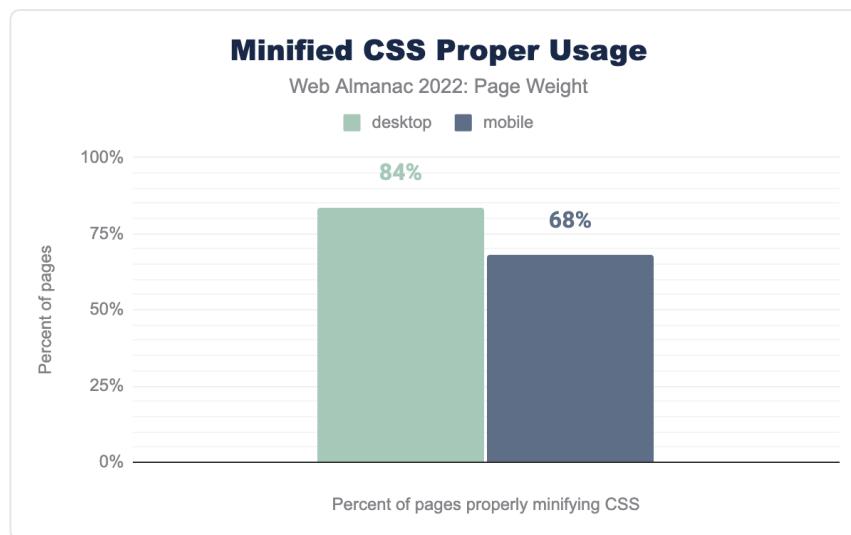


Figure 21.17. Minified CSS proper usage.

84% of Desktop page loads correctly minified the CSS served, and a smaller 68% of mobile page loads.

⁸²⁹. [https://en.wikipedia.org/wiki/Minification_\(programming\)](https://en.wikipedia.org/wiki/Minification_(programming))

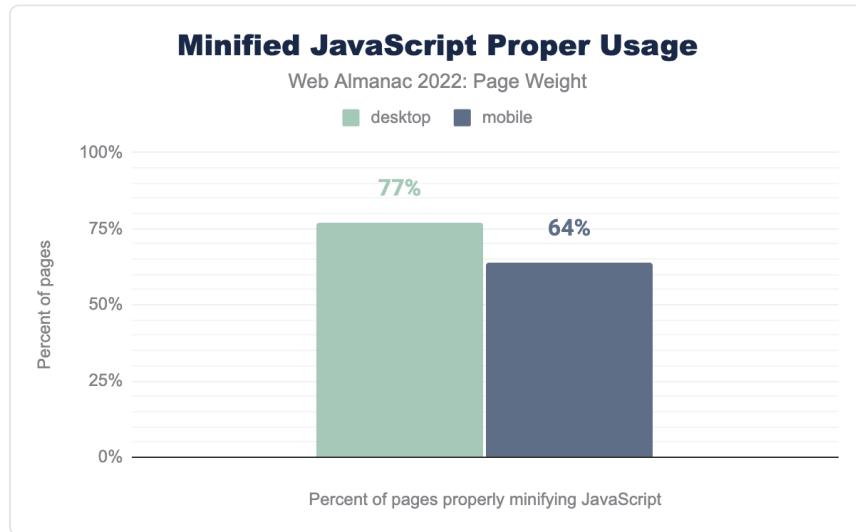


Figure 21.18. Minified JavaScript proper usage.

77% of Desktop page loads correctly minified the JavaScript resources served, and a smaller 64% of mobile page loads.

Whilst minification for both CSS and JavaScript is thankfully popular, with the majority of sites getting it right, there's room for improvement still.

It's disappointing that like in compression, minification for mobile users lags behind desktop. Like compression, saving bytes is especially helpful on mobile devices.

Unlike compression, there's no overhead on the client side to minification, resources don't need to be 'unminified' to be used. There can be overhead on the server-side if the minification is done on-the-fly at serving time, but CSS and JavaScript are likely to be static files, and minification should be done at build time, or before publishing the resource, meaning there is no further overhead.

Caching and CDNs

Caching allows a resource to be reused until a specified expiration. Caches are used in browsers and on servers.

CDNs are a popular example. These interconnected servers are geographically distributed in order to send cached content from a network location closest to the user. CDNs do not reduce page weight but rather reduce the delay by reducing the distance between requestor and

resource.

As such, we did not investigate this in this chapter, but the CDN chapter covers this in more detail and last year's Caching⁸³⁰ chapter gives more detail on that.

Conclusion

A hefty page weight results in longer user wait. The expense of ever-inflating web pages is paid in cost of data-access, cost of devices to meet the technical requirements, and time.

While images and JavaScript remain the largest contributors to byte size, 2022 revealed surprising increases such as a greater prevalence of byte-heavy videos on mobile and the median font byte size being higher than that of its image counterpart.

While they thankfully are extreme compared to most of the web, the absurd outliers seen in the 100th percentile show the potential for unchecked bloat as new file types and functionalities are introduced to the digital experience.

Byte-saving technologies have alleviated some of the pressure but with higher adoption rates on desktop despite their use being more impactful to mobile users. Unchecked, this continued inflation will further the gap of digital inequality.

Authors



Jamie Indigo

Twitter: @Jammer_Volts | GitHub: fellowhuman1101 | Website: <https://not-a-robot.com>

Jamie Indigo isn't a robot, but speaks bot. As a technical SEO at DeepCrawl⁸³¹, they study how search engines crawl, render, and index the web. They love to tame wild JavaScript and optimize rendering strategies. When not working, Jamie likes horror movies, graphic novels, and Dungeons & Dragons.

830. <https://almanac.httparchive.org/en/2021/cdn>
831. <https://www.deepcrawl.com>

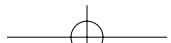
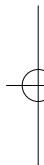
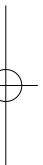


Dave Smart

🐦 @davewsmart ⚡ dwsmart 🌐 <https://tamethebots.com>

Dave Smart is a developer and technical search engine consultant at Tame the Bots⁸³². They love building tools and experimenting with the modern web, and can often be found at the front in a gig or two.

^{832.} <https://tamethebots.com>



Part IV Chapter 22

CDN



Written by Haren Bhandari and Joe Viggiano

Reviewed by Yutaka Oka

Analyzed by Haren Bhandari and Joe Viggiano

Edited by Barry Pollard

Introduction

This chapter provides insights regarding the current state of CDN usage. CDNs are playing an increasingly important role in delivering content to users around the globe—even for smaller sites by facilitating the delivery of static and third-party content such as JavaScript libraries, Fonts and other content. Another key aspect of the CDNs that we will discuss in this chapter is the role CDNs play in adoption of new standards such as TLS and HTTP versions.

We think that CDNs will continue play a vital role in the future not just for content delivery but for content security as well. We recommend that users look at CDNs from both a performance and a security viewpoint.

What is a CDN?

A Content Delivery Network (CDN) is a geographically distributed network of proxy servers. The

goal of a CDN is to provide high availability and performance for web content. It does this by distributing content closer to the end users as well as supporting advanced technologies to delivery content optimally.

Due to the explosion of web content such as videos and images, CDN has been a vital part of many websites to provide a smooth user experience. Post COVID-19, the need for CDN has only increased due to many brick and mortar businesses moving online, increase in web conferencing, online gaming and video streaming.

During the early days, a CDN was a simple network of proxy servers which would:

1. Cache content (like HTML, images, stylesheets, JavaScript, videos... etc.)
2. Reduce network hops for end users to access content
3. Offload TCP connection termination away from the data centers hosting the web properties

They primarily helped web owners to improve the page load times and to offload traffic from the infrastructure hosting these web properties.

Over time, the services offered by CDN providers have evolved beyond caching and offloading bandwidth/connections. Due to its distributed nature and large distributed network capacity CDNs have proved to be extremely efficient at handling large scale Distributed Denial-of-Service (DDoS)⁸³³ attacks. Edge computing is another service that has gained popularity in the recent years. Many CDN vendors provide compute services at the edge that allows the web owners to run simple code at the edge.

Other services offered by the CDN vendors include:

- Cloud-hosted Web Application Firewalls (WAF)⁸³⁴
- Bot Management solutions
- Clean pipe solutions (Scrubbing Data-centers)
- Image and video management solutions

There are benefits to web owners in pushing web application logic and workflows closer to the end user. This eliminates the round trip and bandwidth that a HTTP/HTTPS request would take. It also handles near-instant scalability requirements for the origin. A side-effect of this is that Internet Service Providers (ISPs) benefit from the scalability management as well, which improves their infrastructure capacities.

833. https://en.wikipedia.org/wiki/Denial-of-service_attack#Distributed_attack

834. https://en.wikipedia.org/wiki/Web_application_firewall

Caveats and disclaimers

As with any observational study, there are limits to the scope and impact that can be measured. The statistics gathered on CDN usage for the Web Almanac are focused more on applicable technologies in use and not intended to measure performance or effectiveness of a specific CDN vendor. While this ensures that we are not biased towards any CDN vendor, it also means that these are more generalized results.

These are the limits to our testing methodology:

- **Simulated network latency:** We use a dedicated network connection that synthetically shapes traffic.
- **Single geographic location:** Tests are run from a single datacenter and cannot test the geographic distribution of many CDN vendors.
- **Cache effectiveness:** Each CDN uses proprietary technology and many, for security reasons, do not expose cache performance.
- **Localization and internationalization:** Just like geographic distribution, the effects of language and geo-specific domains are also opaque to these tests.
- **CDN detection:** This is primarily done through DNS resolution and HTTP headers. Most CDNs use a DNS CNAME to map a user to an optimal data center. However, some CDNs use Anycast IPs or direct A+AAAA responses from a delegated domain which hide the DNS chain. In other cases, websites use multiple CDNs to balance between vendors, which is hidden from the single-request pass of our crawler.

All of this influences our measurements.

Most importantly, these results reflect the support of specific features (for example TLSv1.3, HTTP/2) per site, but do not reflect actual traffic usage. YouTube is more popular than `www.example.com` yet both will appear as equal in our dataset.

With this in mind, here are a few statistics that were intentionally not measured in the context of a CDN:

1. Time To First Byte (TTFB)
2. CDN Round Trip Time
3. Core Web Vitals
4. “Cache hit” versus “cache miss” performance

While some of these could be measured with HTTP Archive dataset, and others by using the CrUX dataset, the limitations of our methodology and the use of multiple CDNs by some sites, will be difficult to measure and so could be incorrectly attributed. For these reasons, we have decided not to measure these statistics in this chapter.

We did not see any notable differences between mobile and desktop so, to avoid repeating ourselves, the data provided in this chapter is primarily for mobile usage unless otherwise noted.

CDN adoption

A web page is composed of following key components:

1. Base HTML page (for example, `www.example.com/index.html`—often available at a more friendly name like just `www.example.com`).
2. Embedded first-party content such as images, css, fonts and javascript files on the main domain (`www.example.com`) and the subdomains (for example, `images.example.com`, or `assets.example.com`).
3. Third-party content (for example, Google Analytics, advertisements) served from third-party domains.

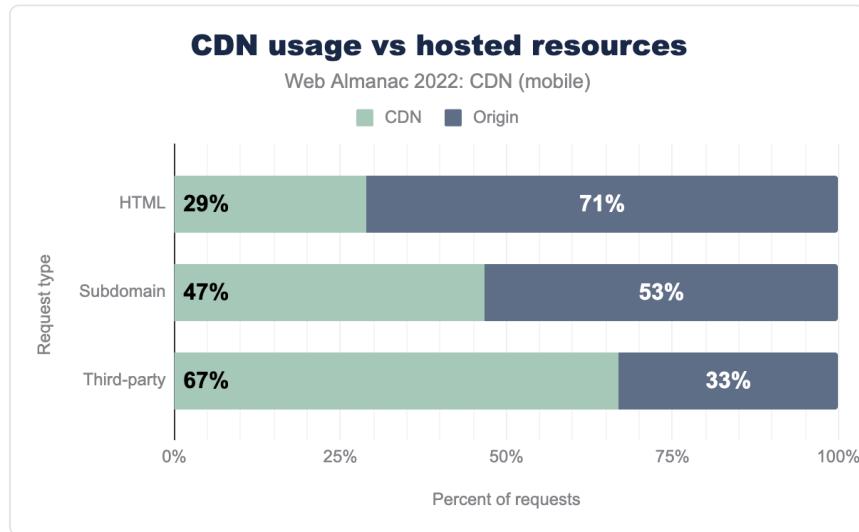


Figure 22.1. CDN usage vs hosted resources on mobile.

The above chart shows the split of each type of request for CDNs versus hosted resources for mobile—as mentioned in the introduction almost identical figures were seen for desktop. We see CDNs are often utilized for delivering static content such as images, stylesheets, JavaScript, and fonts. This kind of content doesn't change frequently, making it a good candidate for caching on a CDNs proxy servers. We still see CDNs are used more frequently for this type of resource—especially for third-party content.

CDNs can provide better performance for delivering non-static content as well as they often optimize the routes and use most efficient transport mechanisms, but we see the usage of serving the HTML still lags considerably behind the other two types.

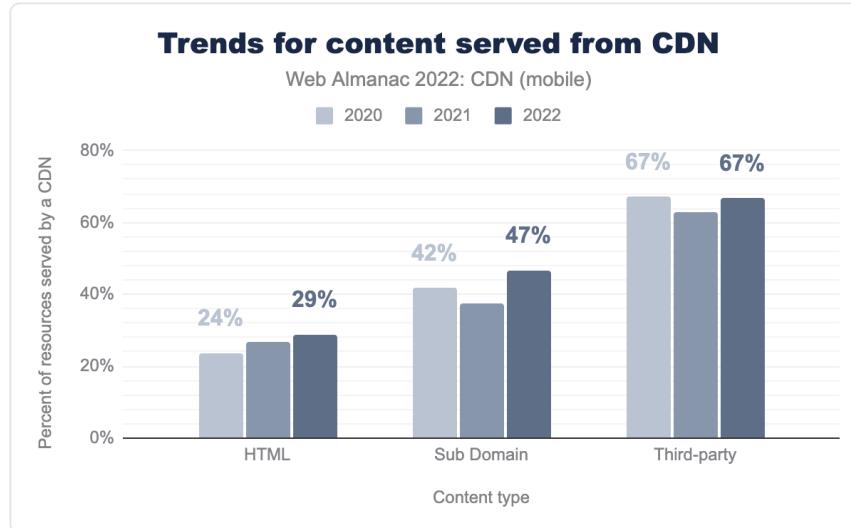


Figure 22.2. Trends for content served from CDN for mobile

Compared to the year 2021⁸³⁵ we found that the usage of CDN has been steadily increasing. There was a large bump in CDN usage for the content served from subdomains, and a smaller one for HTML while third-party CDN usage remained relatively static.

These are some of the potential reasons that can be attributed to this rise:

- Post pandemic, many businesses took a large portion of their physical business online. This put a lot of strain on their servers and found that it was much more efficient to serve the static content through CDNs for offloading through caching and faster delivery.
- This increase was not seen in 2021 as many businesses were still trying to figure out the optimal solution for their problem. In fact we saw a dip in CDN usage for the subdomain and third-party type.
- Sites relied on serving third-party content through third-party domains instead of their own domains. The fact that the amount of content served from third-party domains increased by 3% during this period supports this assumption.

Regarding the base HTML page, the traditional pattern has been to serve the base HTML from the origin and this pattern has continued as majority of base pages continue to be served from the origin. However, there has been a 4% increase in the base pages being served from CDNs.

⁸³⁵. <https://almanac.httparchive.org/en/2021/cdn>

The trend of base HTML pages being served from the CDN is clearly on the rise.

These are some of the likely reasons behind the rise:

- CDNs can improve load time of the base HTML page that can be of high importance to improve customer experience and keep users engaged.
- Using distributed DNS from by CDN providers is simpler and faster.
- It is easier to plan Disaster Recovery if most of the content including the base HTML page is pushed through CDNs. CDNs often provide a failover functionality to automatically switch to the alternative site once the primary site becomes unstable or unavailable.

While we observed CDN adoption across different types of content, we will look at this data from a different point of view below—based on the site popularity.

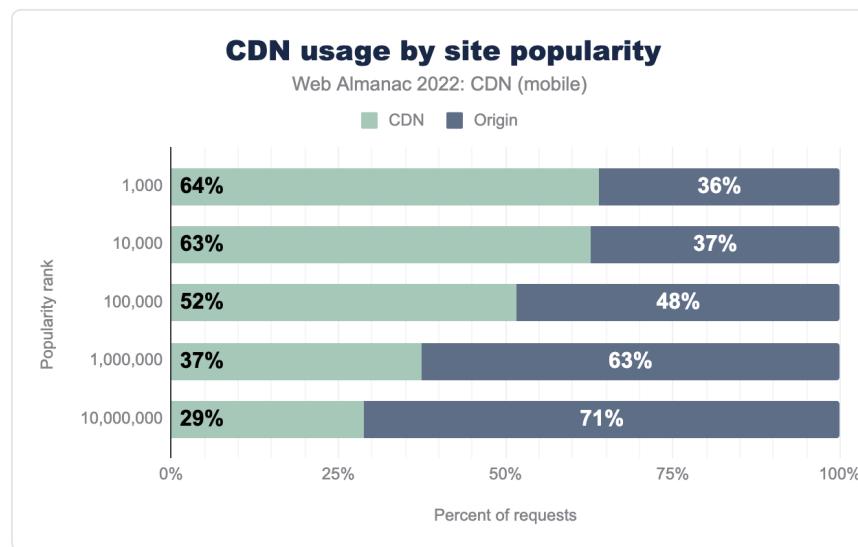


Figure 22.3. CDN usage by site popularity on mobile.

Looking at CDN usage for websites based on their popularity—sourced from Google's Chrome UX Report—the top 1,000-10,000 contribute to the highest usage of CDN. For the high ranked sites, it is understandable that owner companies are investing in CDN for performance and other benefits but even for the top 1,000,000 sites, there has been about a 7% increase in the amount of content delivered through CDNs compared to 2021⁸³⁶. This increase in CDN usage

⁸³⁶ <https://almanac.httparchive.org/en/2021/cdn#fig-3>

for lower popularity sites can be attributed to the fact that there has been an increase in free and affordable options for CDNs and many hosting solutions have CDNs bundled with the services.

Top CDN providers

CDN providers can be broadly classified into 2 segments:

1. Generic CDN (Akamai, Cloudflare, CloudFront, Fastly... etc.)
2. Purpose-built CDN (Netlify, WordPress... etc.)

Generic CDNs address the mass market requirements. Their offerings include:

- Web site delivery
- Mobile app API delivery
- Video streaming
- Edge computing services
- Web security offerings

This appeals to a larger set of industries and is reflected in the data.

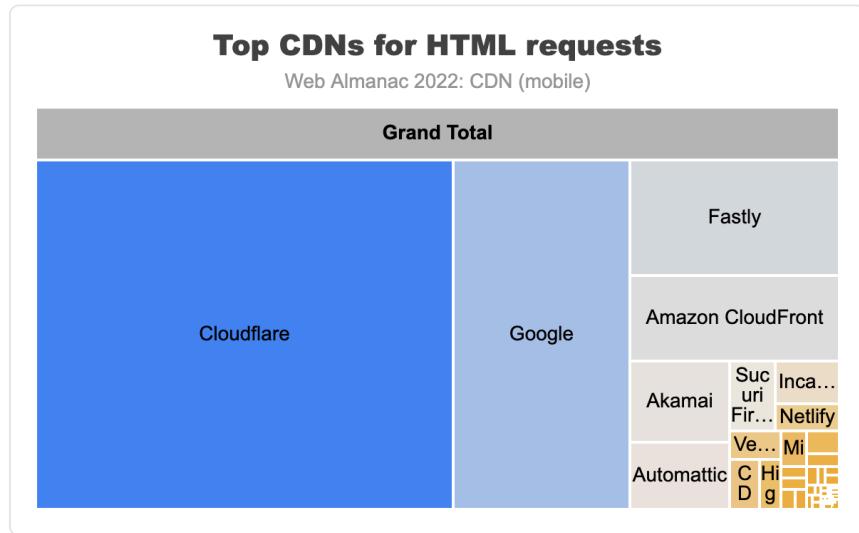


Figure 22.4. Top CDNs for HTML requests on mobile.

The above figure shows the top CDN providers for base HTML requests. The top vendors in this category are Cloudflare, Google, Fastly, Amazon CloudFront, Akamai and Automattic.

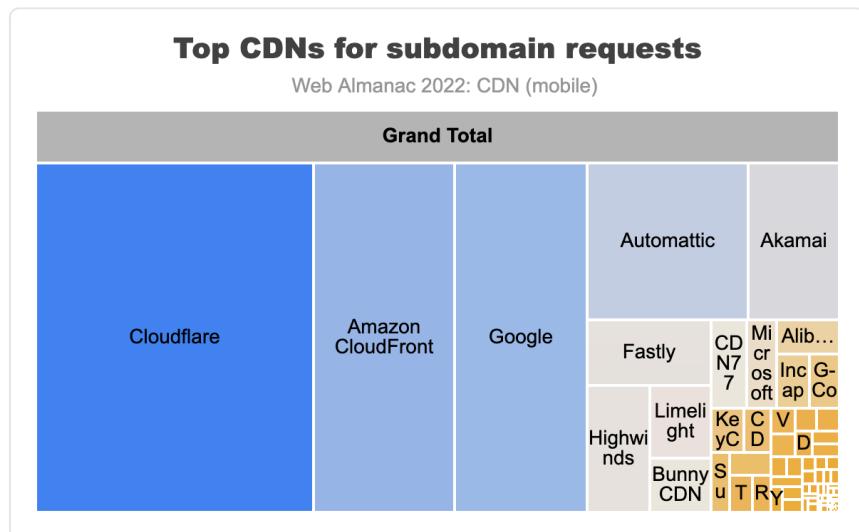


Figure 22.5. Top CDNs for subdomain requests on mobile.

For the subdomain requests we can see greater usage of providers like Amazon and Google.

This is because many users have their content hosted in the cloud services they provide and the users utilize CDN offerings along with their cloud services. This helps the users to scale their applications and increase the performance of their application.

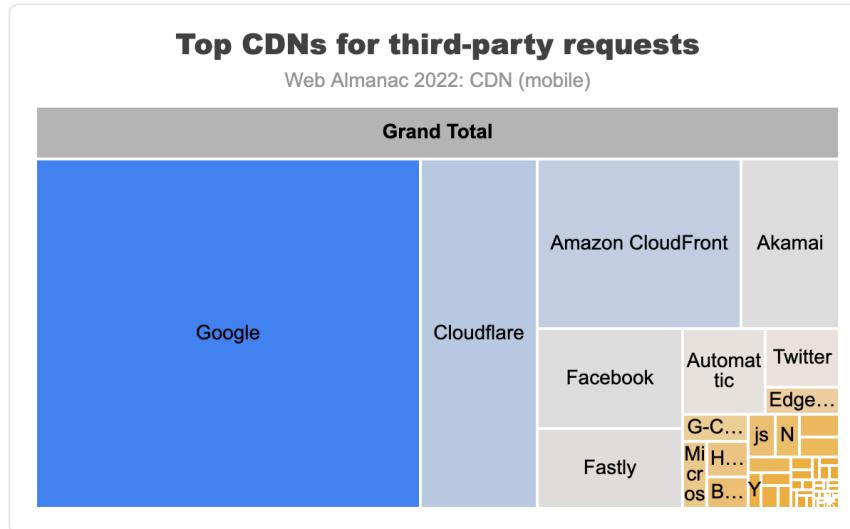


Figure 22.6. Top CDNs for third-party requests on mobile.

Looking at third-party domains, a different trend in top CDN providers is seen. We see Google top the list before the generic CDN providers. The list also brings Facebook into prominence. This is backed by the fact that a lot of third-party domain owners require CDNs more than other industries. For the larger third-party providers—like Google, like Facebook—this necessitates them to invest in building a purpose-built CDN. A purpose-built CDN is one which is optimized for a particular content delivery workflow.

For example, a CDN built specifically to deliver advertisements will be optimized for:

- High input-output (I/O) operations
- Effective management of long tail⁸³⁷ content
- Geographical closeness to businesses requiring their services

This means purpose-built CDNs meet the exact requirements of a particular market segment as opposed to a generic CDN solution. Generic solutions can meet a broader set of requirements but are not optimized for any particular industry or market.

⁸³⁷. https://en.wikipedia.org/wiki/Long_tail

TLS usage

With CDNs set up in the request-response workflows, the end-user's TLS connection terminates at the CDN. In turn, the CDN sets up a second independent TLS connection and this connection goes from the CDN to the origin host. This break in the CDN workflow allows the CDN to define the end-user's TLS parameters. CDNs tend to also provide automatic updates to internet protocols. This allows web owners to receive these benefits without making changes to their origin.

TLS adoption impact

The charts below show that the adoption of the latest version of TLS has been much higher for the content served from CDN compared to origin

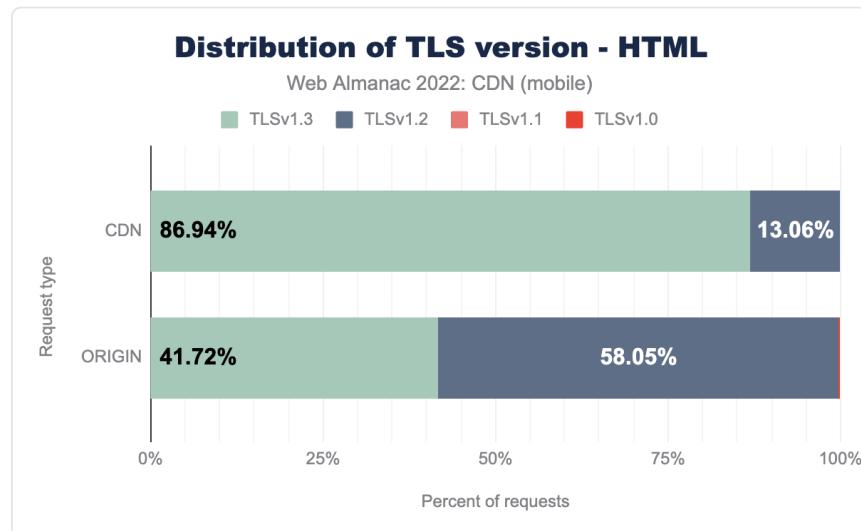


Figure 22.7. Distribution of TLS version for HTML (mobile).

Compared to the year 2021⁸³⁸, for mobile HTML content the adoption of TLS v1.3 has increased by 5% while for the content served from origin the TLS v1.3 adoption has increased by 10%.

⁸³⁸. <https://almanac.httparchive.org/en/2021/cdn#tls-adoption-impact>

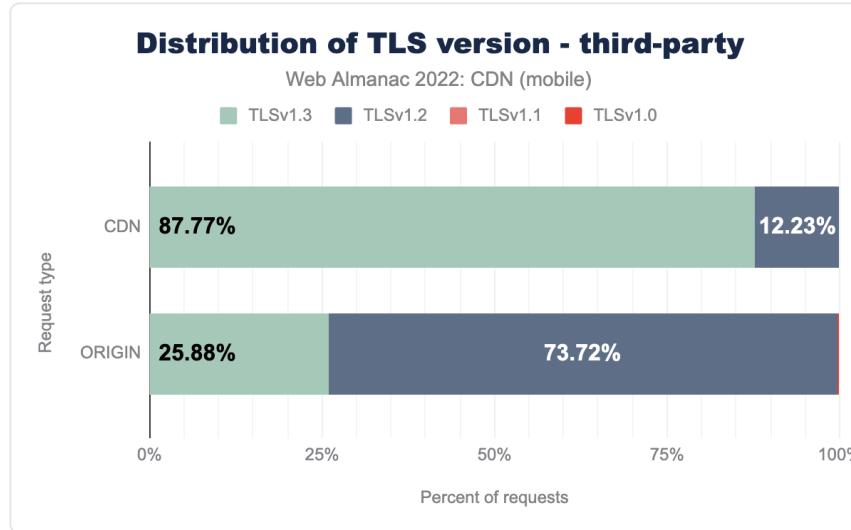


Figure 22.8. Distribution of TLS version for third-party requests (mobile).

In the current security landscape it is important for the content to be delivered via the latest TLS version. It can be seen from the data above that the move to TLS v1.3 was much faster for CDNs compared to the origin. This shows the added security benefit of using CDNs for content delivery.

TLS performance impact

Common logic dictates that the fewer hops it takes for a HTTPS request-response to traverse, the faster the round trip would be.

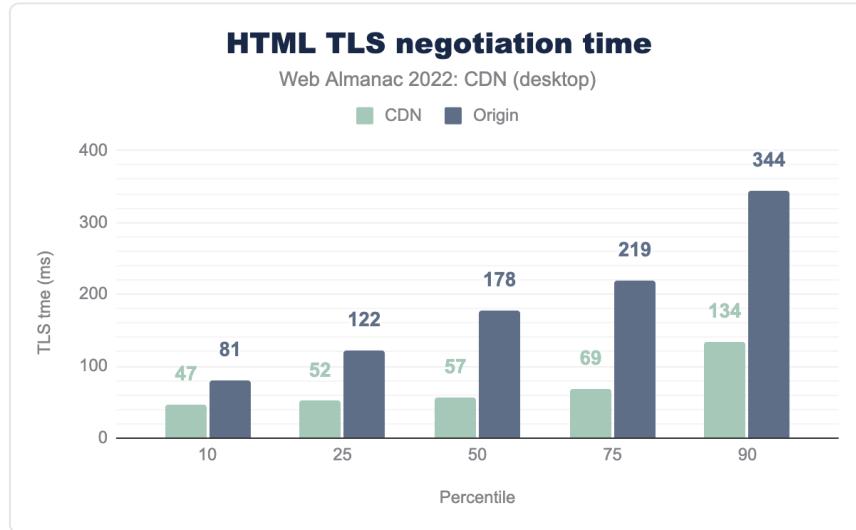


Figure 22.9. HTML TLS negotiation - CDN vs origin (desktop)

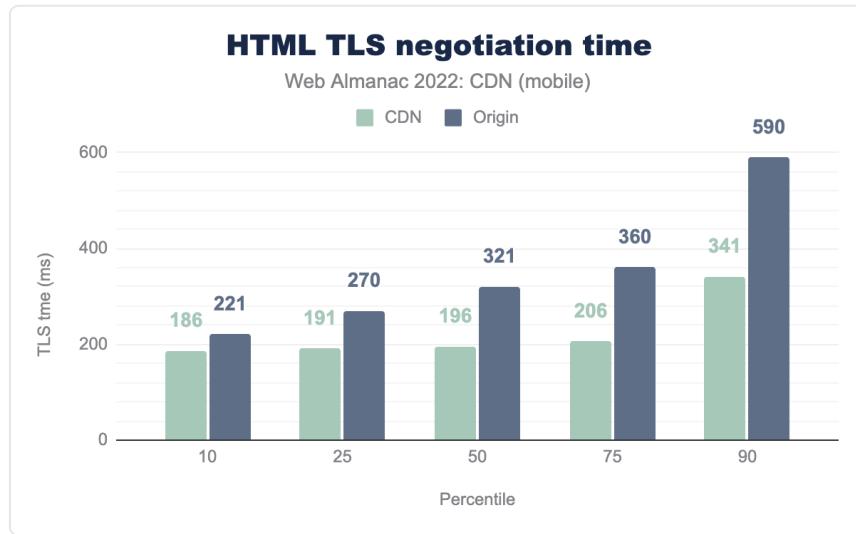


Figure 22.10. HTML TLS negotiation - CDN vs origin (mobile)

As it can be seen from the figures above, the TLS negotiation time is generally much better when with CDNs. This is even more so when comparing the desktop and mobile data, where the greater *Round Trip Time* (RTT) used by our mobile crawler results in much greater TLS negotiation times.

CDNs have helped slash the TLS connection times. This is due to their proximity to the end user and adoption of newer TLS protocols that optimize the TLS negotiation. CDNs hold the edge over origin at all percentiles here. At P10 and P25, CDNs are nearly 1.5x faster than origin in TLS set up time. The gap increases even more once we hit the median and above, where CDNs are nearly 2x faster (mobile) and nearly 3x faster (desktop). 90th percentile users using a CDN will have better performance than 50th percentile users on direct origin connections.

This is particularly important when you consider that all sites have to be on TLS these days. Optimal performance at this layer is essential for other steps that follow TLS connection. In this regard, CDNs are able to move more users to lower percentile brackets compared to direct origin connections.

HTTP/2+ adoption

The HTTP/2 specification was first introduced in 2015 and saw broad support with most major browsers adopting before the end of the year. The HTTP application layer protocol had not been updated since HTTP 1.1 in 1997 and since then the web traffic trend, content-type, content size, website design, platforms, mobile apps and more have evolved significantly. Thus, there was a need to have a protocol which can meet the requirements of the modern-day web traffic and that protocol was realized with HTTP/2.

Despite the hype of HTTP/2 and the promise of reduced latency and other functionality, adoption relied on server side updates to support the newer application protocol. CDNs can help bridge the challenge of newer implementations for web owners, and this is also the case for the even newer HTTP/3 protocol. An HTTP connection terminates at the CDN level, and this provides web owners the ability to deliver their website and subdomains over HTTP/2 and HTTP/3 without the need to upgrade their own infrastructure to support it. Similar benefits were also seen with the adoption of newer TLS versions.

CDNs act as the proxy to bridge the gap by providing a layer to consolidate hostnames and route traffic to relevant endpoints with minimal change to their hosting infrastructure. Features like prioritizing content in the queue and server push can be managed from the CDNs side and a few CDNs even provide hands-off automated solutions to run these features without any inputs from website owners, thus providing a boost to HTTP/2 adoption.

Note that due to the way HTTP/3 works (see the HTTP chapter for more information), HTTP/3 is often not used for first connections which is why we are instead measuring "HTTP/2+", since many of those HTTP/2 connections may actually be HTTP/3 for repeat visitors (we have assumed that no servers implement HTTP/3 without HTTP/3).

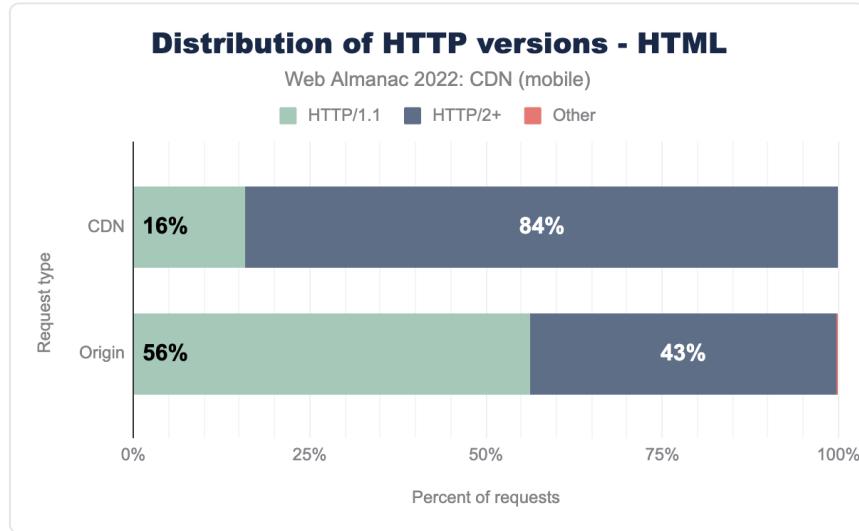


Figure 22.11. Distribution of HTTP versions for HTML (mobile).

There are stark contrasts in the graph above with high HTTP/2+ adoption by domains on CDNs compared to the ones not using a CDN.

In 2021 nearly 40% of the content served from origin had adopted HTTP/2⁸³⁹ while during the same time 81% of the content served from CDNs were served through HTTP/2. For origins this number has grown by 3% points while for the CDN it has grown by 6% points—further widening the considerable gap that was already present. This shows how CDNs were able to allow the website owners to take advantage of newer protocols from very early stage without making any changes in the origin.

839. <https://almanac.httparchive.org/en/2021/cdn#http2-or-better-adoption>

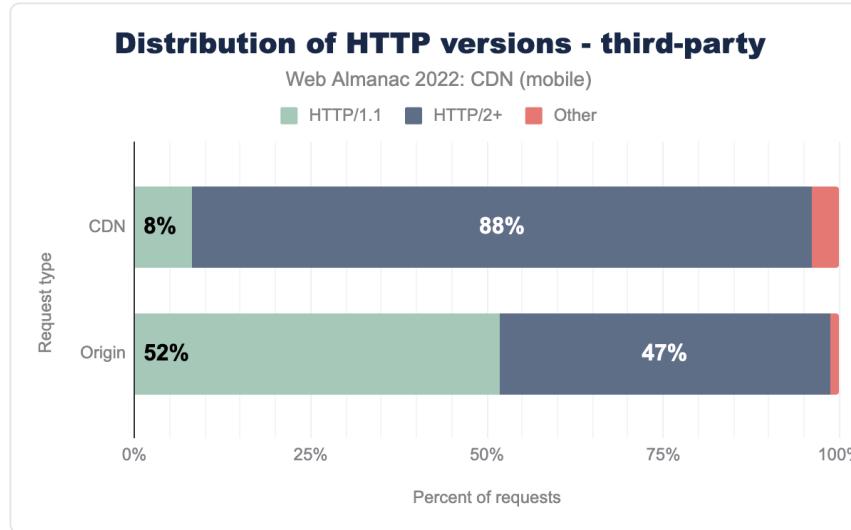


Figure 22.12. Distribution of HTTP versions for third-party requests (mobile).

Third-party domains have been even quicker to support new protocols as we saw in our previous study. In 2022 we saw a further decline in the share of HTTP/1.1 for the third-party domains, though our data was unable to identify a larger number of the protocol used this year, which warrants further investigation.

Third-party domains need to have consistent performance across all network conditions, and this is where HTTP/2+ adds value. In June of 2022, the Internet Engineering Task Force (IETF) published the HTTP/3 RFC⁸⁴⁰ to take the web from TCP to UDP. Many CDN providers have been quick to adopt HTTP/3 support, some before its formal RFC publication, and over time we should see web owners adopting HTTP/3, especially with mobile network traffic having a higher contribution to the total internet traffic. Stay tuned for more insights in 2023.

Brotli adoption

Content delivered over the internet employs compression to reduce the payload size. A smaller payload means it's faster to deliver the content from server to end user. This makes websites load faster and provide a better end-user experience. For images, this compression is handled by image file formats like JPEG, WEBP, AVIF—refer to the Media chapter for more on this. For textual web assets—such as HTML, JavaScript, and stylesheets—compression was traditionally handled by the Gzip file format. Gzip has been in existence since 1992. It did a good job of

840. https://www.theregister.com/2022/06/07/http3_rfc_9114_published/

making text asset payloads smaller, but a new text asset compression can do better than Gzip by using the newer *Brotli* compression format.

Similar to TLS and HTTP/2 adoption, Brotli went through a phase of gradual adoption across web platforms. At the time of this writing, Brotli is supported by over 96%⁸⁴¹ of the web browsers globally. However, not all websites compress text assets in Brotli format. This is because of both lack of support and of the longer time required to compress a text asset in Brotli format compared to Gzip compression. Also, the hosting infrastructure needs to have backward compatibility to serve Gzip compressed assets for older platforms which do not support the Brotli format, which can add complexity.

The impact of this is observed when we compare websites which are using CDN against the ones not using CDN.

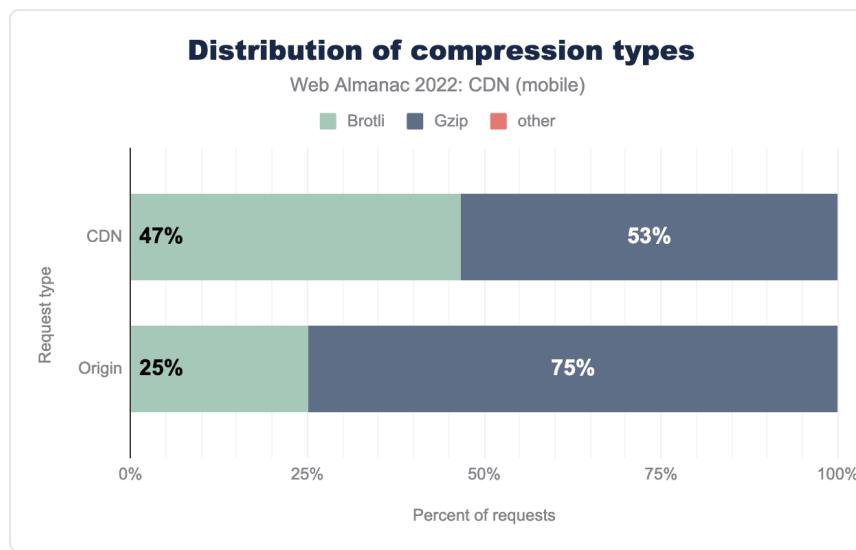


Figure 22.13. Distribution of compression types (mobile).

Both CDN and Origin have shown an increase in adoption of Brotli compared to the previous year⁸⁴². We have seen the adoption of Brotli on CDN grow by 5% points while the Origin grew by almost 4% points. We will be able to see if this trend will continue in year 2023 or we have reached the saturation point.

841. <https://caniuse.com/brotli>

842. <https://almanac.httparchive.org/en/2021/cdn#brotli-adoption>

Client Hint adoption

Client Hints allows a web server to proactively request data from the client and are sent as part of the HTTP headers. The client may provide information such as device, user-agent preferences and networking. Based on the provided information, the server can determine the most optimal resources to respond with to the requesting client. Client Hints were first introduced on the Google Chrome browsers and while other Chromium based browsers have adopted it, other popular browsers have limited or no support for Client Hints.

The CDN, origin servers, and client browser must all support Client Hints to be utilized properly. As part of the flow, the CDN can present the `Accept - CH` HTTP header to the client in order to request which Client Hints a client should include in subsequent requests. We measured clients responses where the CDN provided this header inside the request and measured it across all CDN requests recorded as part of our testing.

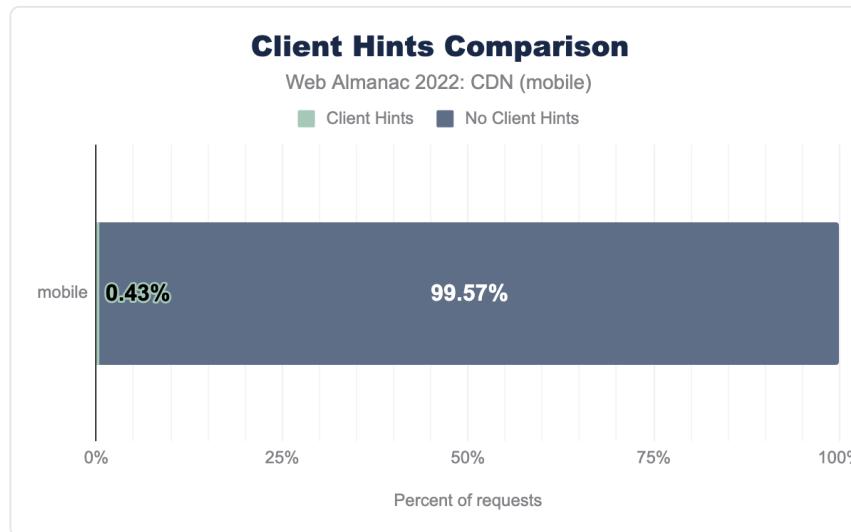


Figure 22.14. Client Hints Comparison (mobile).

For both desktop and mobile clients we saw less than 1% usage of Client Hints, showing that Client Hints adoption is still in its infancy.

Image format adoption

CDNs have traditionally been used to cache, compress and deliver static content such as

images since their inception. Since then many CDNs have added the ability to dynamically change images in both format and sizing on the fly to optimize the image for different use cases.

This may be performed automatically, based on the user agent or client hints, or by sending additional parameters in the query string whereby compute at the edge will interpret and modify the image in the response accordingly. This allows site operators to upload a single high resolution image and modify it on the fly for when lower quality or lower resolution images are needed such as in thumbnails.

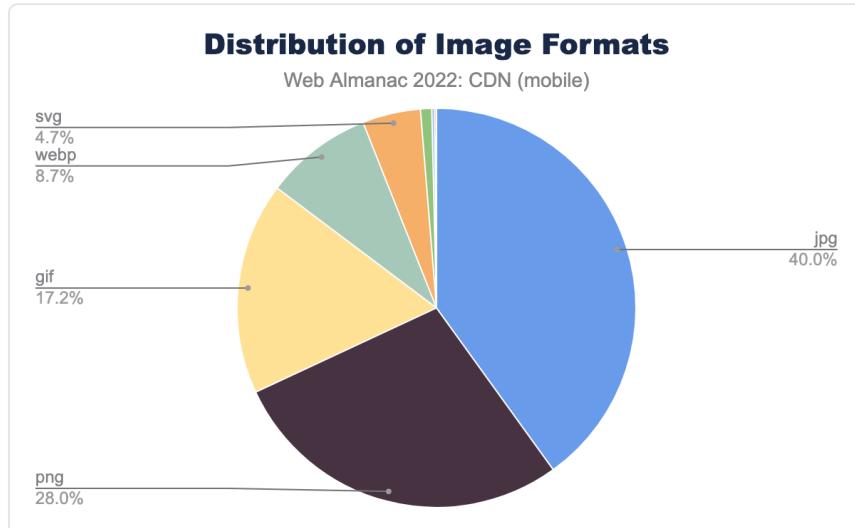


Figure 22.15. Distribution of Image Formats (mobile).

Across both desktop and mobile the dominant image formats were JPG (JPEG) and PNG. JPG provides a lossy compressed file format optimizing for size. PNG or Portable Graphics Format supports lossless compression meaning no data will be lost when the image is compressed, however the image overall is larger in size than a JPG. For more information on JPG vs PNG visit Adobe's website⁸⁴³.

Conclusion

From our continued study in the past years, we can see that the CDNs have not only been vital to website owners in order to reliably deliver content from origin to any user across the globe, they have also played a major role in new security and web standards adoption.

843. <https://www.adobe.com/creativecloud/file-types/image/comparison/jpeg-vs-png.html>

In general, we have seen the rise in the usage of CDNs across the board. We saw that the CDN greatly facilitated the adoption new web security standards such as TLS 1.3 where we saw much higher percentage of traffic using TLS 1.3 came from CDN.

When it comes to the adoption of new web standards and new web technologies such as HTTP/2, Brotli compression we again saw CDNs leading the way. Much higher percentages of websites served out of CDN saw these new standards being adopted. From the end-user perspective this is very positive development as they will be able use the site securely while getting the optimal user experience.

We are also looking at new metrics like Client Hints and image format adoption starting this year. We will be able to get more insights as we collect more data for the next year.

There are limitations to the insights we can deduce about CDNs from the outside, since it is hard to know the secret sauce powering them behind the scenes. However, we have crawled the domains and compared the ones on CDNs against those who are not. We can see that CDNs have been an enabler for websites to adopt new web protocols, from the network layer to the application layer.

This role of CDNs is highly valuable and this will continue to be the case. CDN providers are also a key part of the Internet Engineering Task Force⁸⁴⁴, where they help shape the future of the internet. They will continue to play a key role aiding internet-enabled industries to operate smoothly, reliably and quickly.

We look forward to the next year to collect more data and provide useful insights to our readers.

Authors



Haren Bhandari

harendra

Haren Bhandari is a Solutions Architect at Amazon Web Services. Before joining Amazon Web Services, Haren used to work at Akamai Technologies and has deep experience with CDNs.

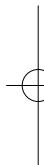
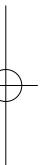
⁸⁴⁴. <https://www.ietf.org/>



Joe Viggiano

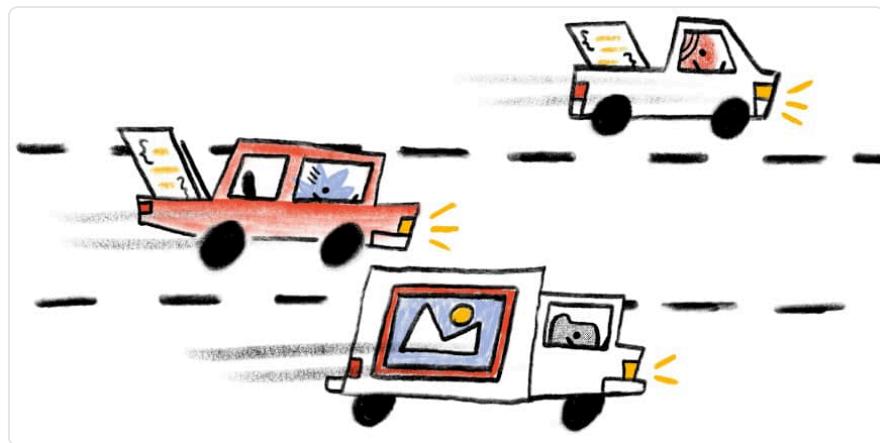
 [joeviggiano](#)

Joe Viggiano is a Media & Entertainment Solutions Architect at Amazon Web Services helping customers deliver media content at scale.



Part IV Chapter 23

HTTP



Written by Vaspol Ruamviboonsuk

Reviewed by Barry Pollard, Robin Marx, and Lucas Pardue

Analyzed by Vaspol Ruamviboonsuk

Edited by Barry Pollard

Introduction

HTTP is the cornerstone of the web ecosystem, providing the foundation for exchanging data and enabling various types of internet services. It has gone through several evolutions especially in the last few years with the introduction of HTTP/2 and, more recently, HTTP/3. HTTP/2 attempted to address shortcomings of HTTP/1.1 such as the limited number of concurrent requests. At first glance, HTTP/3 is similar to HTTP/2 as the semantics are the same, but under the hood HTTP/3 is radically different from its predecessors in that it utilizes QUIC as the transport protocol instead of TCP.

As HTTP/2 provides the basis for HTTP/3, we analyze key features of HTTP/2 such as HTTP/2 Push, prioritization, and multiplexing to understand how much they are still used. We also present case studies from various deployment experiences of these features. For example, HTTP/2 Push allows web servers to preemptively send the response of a resource before the client requests it.

While both push and prioritization should theoretically be beneficial to end users, they can be challenging to use. We discuss new technologies that can potentially be used as alternatives to underperforming HTTP/2 features. For example, 103 Early Hints responses provides an alternative to HTTP/2 server push that achieves the same performance goal of preemptive resource fetches.

Finally, we dive into HTTP/3 by discussing how it is an improvement over HTTP/2 and by analyzing the current support for HTTP/3, where we observe some increase from 2021. We hope that the data points provided in this chapter will provide some insights on future trends and pointers to new technologies that developers can experiment with to improve user experiences.

Evolution of HTTP

HTTP is one of the most important Internet protocols because it powers communications for the web. It began as a text-based protocol through the first three versions: 0.9, 1.0, and 1.1. With its extensibility, HTTP/1.1 was the current HTTP version for 15 years until 2015.

HTTP/2 was a major milestone of HTTP as it evolved from a text-based protocol to a binary-based one. Where HTTP/1.1 supports only serial request and response exchanges, HTTP/2 supports concurrency. Clients and servers represent requests and responses as streams of binary frames. Streams have unique identifiers, which allows frames to be multiplexed and interleaved.

The latest version of HTTP is HTTP/3, which was recently standardized by the IETF in June 2022⁸⁴⁵. While HTTP/3 implements the same features as HTTP/2, there is a vital difference in how it is transported across the internet. HTTP/3 is built on top of QUIC, a UDP-based protocol, which alleviates some of the performance issues that HTTP/2 can face on a lossy network.

HTTP/2 adoption

With various HTTP versions introduced over the years, we begin by describing the current state of HTTP version adoption. We measure the HTTP version usage by taking all resources in the 2022 Web Almanac dataset and identify which version of HTTP each resource was served on.

However, with our setup, it is not trivial to accurately count resources delivered on HTTP/3, as

845. <https://www.rfc-editor.org/rfc/rfc9114.html>

clients have to discover HTTP/3 support, typically via the `alt-svc` HTTP response header. By the time the client receives the `alt-svc` header however, it has already completed the protocol negotiation for HTTP/1.1 or HTTP/2. Only after this point can the client upgrade the protocol to HTTP/3 on subsequent requests or page loads. As such, our data never captures a full HTTP/3 page load.

With the discovery of HTTP/3 being so delayed via the `alt-svc` HTTP header mechanism, our measurements may undercount resources that would have been delivered on HTTP/3 for normal browsing users. Thus, we group resources delivered on HTTP/2 and HTTP/3 together as HTTP/2+.

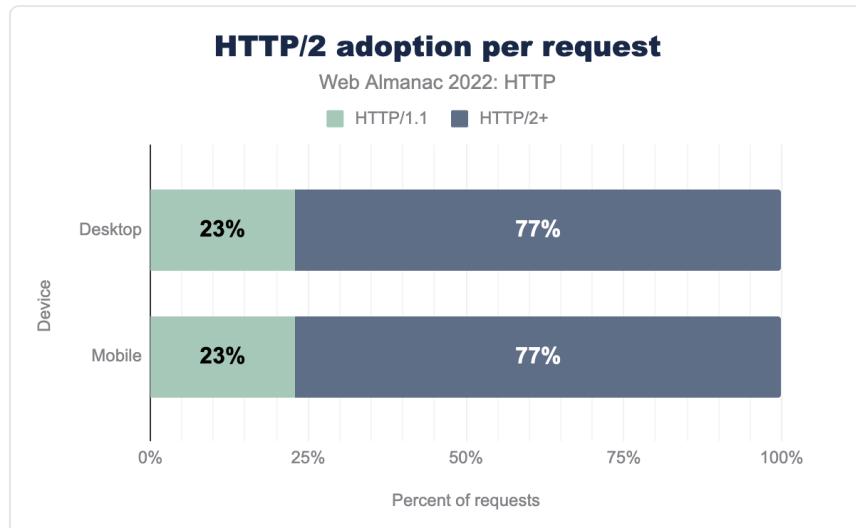


Figure 23.1. Adoption of HTTP/2 and above as a percentage of requests.

First, to understand the status quo, we measure the prevalence of HTTP/2+ adoption. In June 2022, we observed that roughly 77% of requests from our loads use HTTP/2+. This is a 5% increase in HTTP/2+ adoption from July 2021⁸⁴⁶, where we observed that 73% of the requests were on HTTP/2+.

⁸⁴⁶. <https://almanac.httparchive.org/en/2021/http#adoption-by-request>

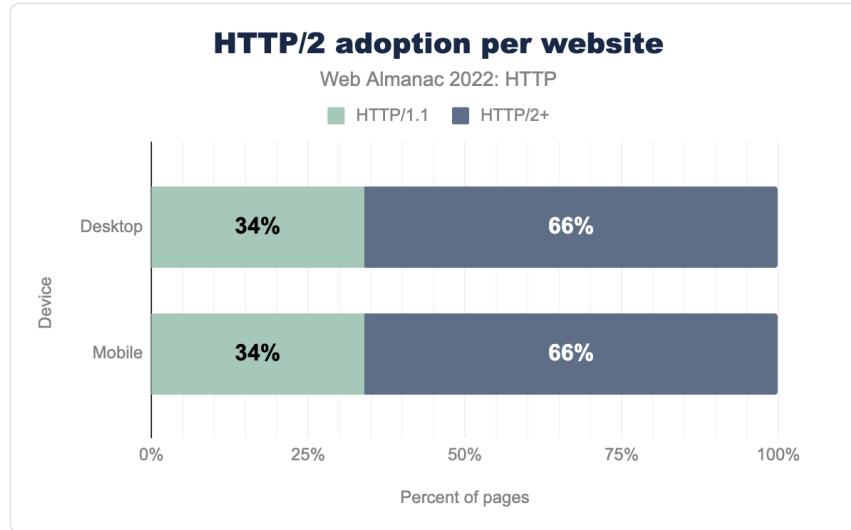


Figure 23.2. Adoption of HTTP/2 and above as a percentage of websites.

With the increase in HTTP/2+ adoption, we would like to understand the driving forces that enable the increase. First, we analyze the HTTP/2+ adoption at per-website granularity by checking whether the landing page of the website was served on HTTP/2+. We observed that approximately 66% of the websites from our dataset, on both desktop and mobile settings, were served on HTTP/2+, whereas this was only true for approximately 60% of the websites in our dataset in 2021⁸⁴⁷. This increase is a positive trend which suggests that websites are ready and moving towards an up-to-date version of HTTP.

847. <https://almanac.httparchive.org/en/2021/http#adoption-of-http2>

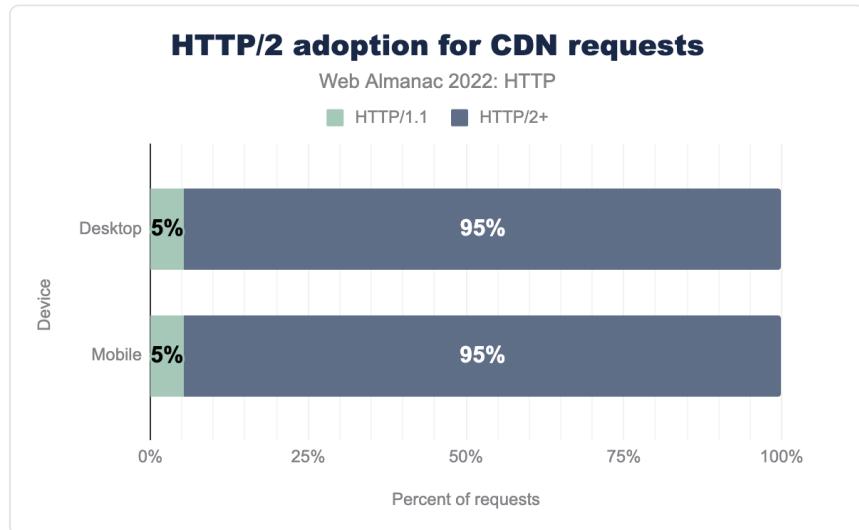


Figure 23.3. Adoption of HTTP/2 and above as a percentage of requests served from a CDN.

Another factor in enabling HTTP/2+ adoption is resources served from CDN. Similar to the observation in our 2021 analysis⁸⁴⁸, we noticed that most resources served from a CDN were on HTTP/2+. The figure above shows that 95% of the requests served from CDN were delivered on HTTP/2+.

Benefits of HTTP/2 and HTTP/3

Next, we focus on how features that HTTP/2 introduced are being adopted. We primarily focus on three notable concepts: *multiplexing* requests over a single network connection, resource prioritization, and *HTTP/2 Push*.

Multiplexing requests over a single connection

An important feature of HTTP/2 is multiplexing requests over a single TCP connection. This is a substantial improvement to earlier versions of HTTP where only one concurrent request is allowed on a TCP connection and, in most cases, only six parallel TCP connections are allowed to a hostname. HTTP/2 introduces the concept of a stream; a logical representation of a connection that is used for resource delivery. Multiple HTTP/2 streams can then be multiplexed onto a single TCP connection thereby removing the per-connection concurrency limitations.

⁸⁴⁸. <https://almanac.httparchive.org/en/2021/http#adoption-by-cdns>

An implication of multiplexing requests into one TCP connection is the reduction of connections required during page loads. Similar to our findings in 2021⁸⁴⁹, pages with HTTP/2 enabled we observe fewer connections than pages that do not have HTTP/2 enabled.

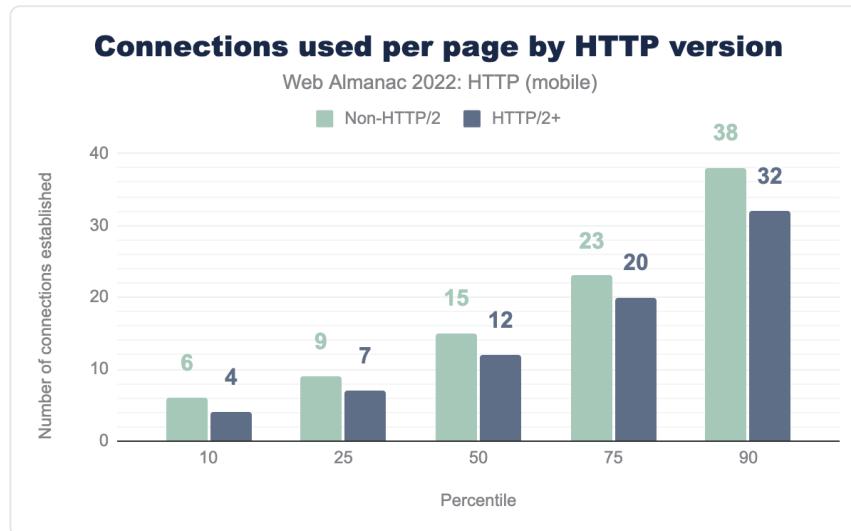


Figure 23.4. Connections used per page by HTTP version.

The figure above shows that the median mobile page has 12 connections established during the page load when HTTP/2 is enabled. In contrast, the median page without HTTP/2 has 15 connections established—an overhead of 3 additional connections. However, the overhead worsens at higher percentiles. The page at the 90th percentile with HTTP/2 enabled has 32 connections, whereas the 90th percentile page without HTTP/2 enabled has 38 connections—a 6 additional connection overhead. These trends are the same between desktop and mobile versions of websites.

Given that we observe an increase in HTTP/2 adoption over the year, it is unsurprising that the number of TCP connections overall has been gradually decreasing over the years. A longitudinal view from HTTP Archive⁸⁵⁰ shows that the median number of connections established decreased by 9 connections, from 22 connections in 2017 to 13 connections in 2022.

Resource prioritization

With HTTP/2, clients can multiplex⁸⁵¹ multiple requests on the same connection. An implication

849. <https://almanac.httparchive.org/en/2021/http#number-of-connections>

850. <https://httparchive.org/reports/state-of-the-web#tcp>

851. <https://stackoverflow.com/questions/36517829/what-does-multiplexing-mean-in-http-2/36519379#36519379>

is that this can negatively impact delivery of render-blocking resources if many resources are inflight at the same time. This can lead to poor user experience. In its original standard⁸⁵², HTTP/2 attempts to address this by proposing a priority tree that clients construct during page load and which web servers can use to prioritize sending more important responses. However, this approach is difficult to use and many web servers and CDNs either did not correctly implement it or ignored it.⁸⁵³ Because of this, it was suggested in a later iteration of HTTP/2⁸⁵⁴ that a different scheme should be used.

With the challenges to HTTP/2 priorities, a new prioritization scheme was needed. The Extensible Prioritization Scheme for HTTP⁸⁵⁵ was developed separately from HTTP/3 and was standardized in June 2022. In this scheme, clients can explicitly assign a priority composed of two parameters via the `priority` HTTP header or the `PRIORITY_UPDATE` frame. The first parameter, `urgency`, tells the server the priority of the requested resource. The second parameter, `incremental`, tells the server whether a resource can be partially used at the client (for example, partially displaying an image as parts of it arrive). Defining the scheme as a HTTP header and as the `PRIORITY_UPDATE` frame makes it extensible as both formats were designed to provide future extensibility. At the time of writing, the scheme has been deployed for HTTP/3 in Safari, Firefox, and Chrome.

While most of the resource priorities are decided by the browser itself, developers can now also use the new priority hints⁸⁵⁶ to tweak the priority of a particular resource. Priority hints can be specified via the `fetchpriority` attribute in the HTML. For example, suppose that a website would like to prioritize a hero image, it can add `fetchpriority` to the image tag:

```

```

Priority hints can be very effective in improving user experience. For example, Etsy conducted an experiment⁸⁵⁷ and observed a 4% improvement in Largest Contentful Paint (LCP) on product listing pages that included priority hints for certain images.

1.2%

Figure 23.5. Mobile pages using Priority Hints.

852. <https://www.rfc-editor.org/rfc/rfc7540#page-25>
 853. <https://github.com/andydavies/http2-prioritization-issues>
 854. <https://www.rfc-editor.org/rfc/rfc9113.html#section-5.3>
 855. <https://httpwg.org/specs/rfc9218.html>
 856. <https://web.dev/priority-hints/>
 857. <https://www.etsy.com/codeascraft/priority-hints-what-your-browser-doesnt-know-yet>

While Priority Hints was only recently released at the end of April 2022 as part of Chrome 101, it has a very promising adoption as we observe that approximately 1% of desktop web pages and 1.2% of mobile web pages have already adopted priority hints in August 2022. With its potential to improve user experience with relative ease, it may be a good feature to experiment with.

HTTP/2 Push

HTTP/2 Push allows web servers to pre-emptively send a response to a request before that request is even sent by the client. For example, a website provider can push a resource that will be used during a page load to the end user along with the main HTML.

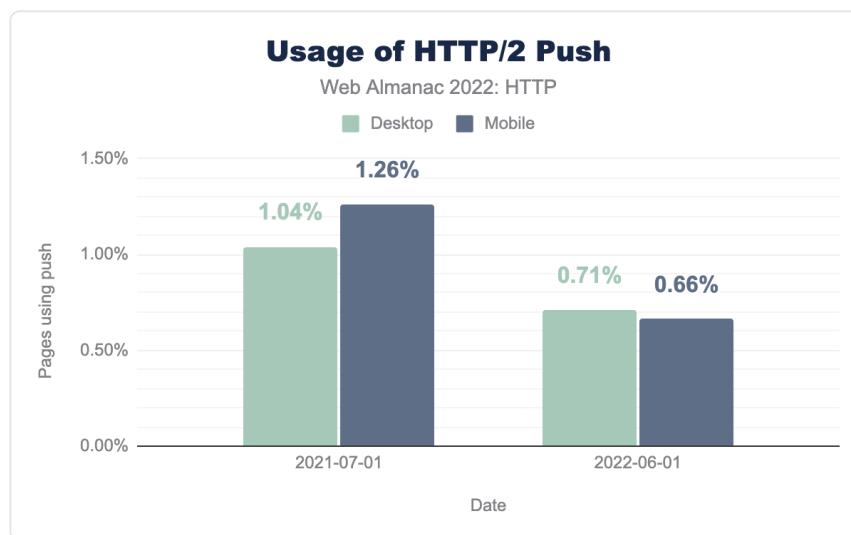


Figure 23.6. Usage of HTTP/2 Push.

In 2021, as shown in the figure above, the percentage of websites using push was very low at 1.26% for mobile. However, in this year's analysis, the number of websites using push decreases to 0.66% for mobile websites. This marks the first decrease in push usage since 2020.

The decrease in websites using push is likely because it is difficult to use effectively⁸⁵⁸. For example, websites cannot accurately know whether a resource being pushed already exists in the client's cache. If it is in the client's cache, the bandwidth used for pushing the resource is wasteful.

⁸⁵⁸. <https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/>

With the difficulties, Chrome decided to deprecate HTTP/2 Push⁸⁵⁹ starting from Chrome version 106⁸⁶⁰. Despite push officially still being a part of the HTTP/3 standard, Chrome—which the HTTP Archive crawler uses—never implemented push for HTTP/3 connections, which might further explain the reduction in usage as sites moved to that version and lost the ability to push.

Alternatives to HTTP/2 Push

Given the challenges to using HTTP/2 Push, and its deprecation from Chrome, developers may be wondering about alternatives.

Preload

Developers can use Preload as one alternative to pre-emptively request a resource that will be used later in a page load. This is enabled by including `<link rel="preload">` in the `<head>` section of the HTML. For example:

```
<link rel="preload" href="/css/style.css" as="style" />
```

Or as a `Link` HTTP header:

```
Link: </css/style.css>; rel="preload"; as="style"
```

Either option allows web servers to include additional URLs or important resources. The client can then issue requests for the provided URLs before the resources are normally discovered during the page load.

While not quite as fast as proactively pushing resources, this is a lot safer in allowing the browser to choose whether to fetch those resources or not if—for example—it already has a copy in the cache.

⁸⁵⁹ <https://groups.google.com/a/chromium.org/g/blink-dev/c/K3rYLvmQUBY/m/ho4qP49oAwAJ>

⁸⁶⁰ <https://developer.chrome.com/blog/removing-push/>

25%

Figure 23.7. Pages using `<link rel="preload">`.

We observed a large number of pages in our dataset include a `<link rel="preload">` tag—approximately 25% on both desktop and mobile.

103 Early Hints

In 2017, the 103 Early Hints status code was proposed⁸⁶¹ and Chrome added support for it this year⁸⁶².

Early Hints can be used to send interim HTTP responses before the final response of the requested object. They can boost performance by leveraging the fact that web servers often require some time to prepare a response, especially for the main HTML document if it is dynamically rendered.

One use case of Early Hints is to deliver `Link: rel="preload"` for resources to preemptively fetch, or `Link: rel="preconnect"` to preemptively connect to other domains. Other headers can conceptually also be conveyed, though this is not supported by any browser.

Early hints can be a better alternative than push because clients retain greater control over how the resources are fetched, but still allow an improvement on just adding preloads and preconnects to the main document HTML. Furthermore, Early Hints can potentially be used for 3rd party resources, which was not possible with push, though again this is not yet supported on any browser⁸⁶³.

1.6%

Figure 23.8. Desktop pages using 103 Early Hints.

There are studies showing that adopting Early Hints can improve user experience. For example, Shopify observed 20-30% LCP improvements⁸⁶⁴ in their lab study. We observe that

861. <https://www.rfc-editor.org/rfc/rfc8297>

862. <https://developer.chrome.com/blog/early-hints/>

863. <https://developer.chrome.com/blog/early-hints/#current-limitations>

864. <https://blog.cloudflare.com/early-hints-performance/>

approximately 1.6% of websites in our dataset have adopted Early Hints even at this early stage and most of the adoption (1.5%) stems from websites running on Shopify's platform.

With the 25% of websites including `<link rel="preload">` with the page response, there is significant potential to convert such link nodes to Early Hints.

HTTP/3

In the remainder of this section, we turn our focus to HTTP/3, as it is the future of HTTP and was standardized in June 2022⁸⁶⁵. In particular, we explore the improvements of HTTP/3 over its predecessors and how much support it currently has. For a more detailed explanation on HTTP/3, you can refer to this excellent series of posts⁸⁶⁶ from Robin Marx⁸⁶⁷, who also helped review this chapter.

Benefits of HTTP/3

While HTTP/2 introduced various improvements over its predecessor, there remain further challenges and opportunities for the protocol. For example, even though multiplexing of requests onto a single TCP connection alleviated head-of-line blocking issues for each request, delivering requests using this method can still be inefficient⁸⁶⁸. This is because lost TCP packets can prevent properly received later TCP packets from being processed until their retransmission arrives—even if the later TCP packet is for a separate HTTP stream. TCP has no concept of the multiplexing happening in the higher, HTTP protocol and so holds up all streams.

HTTP/3 aims to improve upon the shortcomings of HTTP/2 and to do that it is built on QUIC, a UDP-based transport protocol. QUIC addresses TCP head-of-line blocking by implementing reliable packet delivery on top of UDP at a per-stream granularity.

HTTP/3 support

To advertise that HTTP/3 is supported, web servers rely on the `alt-svc` in the HTTP response header. The value of `alt-svc` header contains a list of protocols supported by the server.

⁸⁶⁵. <https://www.rfc-editor.org/rfc/rfc9114.html>

⁸⁶⁶. <https://www.smashingmagazine.com/2021/08/http3-core-concepts-part1/>

⁸⁶⁷. <https://twitter.com/programmingart>

⁸⁶⁸. <https://calendar.perfplanet.com/2020/head-of-line-blocking-in-quic-and-http-3-the-details/>

The screenshot shows a browser's developer tools Network tab with the Headers section selected. The request URL is `https://www.cloudflare.com/`, method is GET, status code is 200 (green), remote address is `104.16.123.96:443`, and referrer policy is `strict-origin-when-cross-origin`. The Response Headers section shows `age: 29`, `alt-svc: h3=":443"; ma=86400, h3-29=":443"; ma=86400` (highlighted in red), and `cache-control: max-age=120`.

Figure 23.9. `alt-svc` response header example.

For example, in September 2022, the `alt-svc` value in the response for `https://www.cloudflare.com` is `h3=":443"; ma=86400, h3-29=":443"; ma=86400` as shown in the screenshot below. `h3` and `h3-29` tell us that Cloudflare supports HTTP/3 and IETF draft 29 of HTTP/3 over UDP port 443. There is also a proposal to speed up the discovery of HTTP/3 as part of DNS lookup; for more details see this post from Cloudflare⁸⁶⁹.

We analyze HTTP/3 adoption by identifying a resource that was served on HTTP/3 or its response header contained an `alt-svc` header with either `h3` or `h3-29` as one of the protocols advertised. This allows us to understand if HTTP/3 could be used, and ignores the limitations mentioned above, of the fresh instance run by our crawler, which has yet to see the `alt-svc` header.

⁸⁶⁹ <https://blog.cloudflare.com speeding-up-https-and-http-3-negotiation-with-dns/>

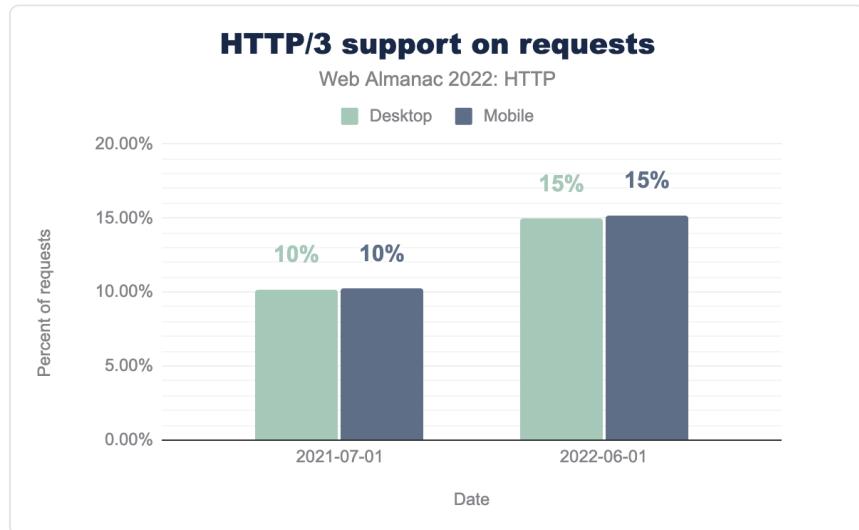


Figure 23.10. HTTP/3 support on requests.

The figure above shows that there is a 5 percentage point increase, from 10% to 15%, in the percentage of requests with HTTP/3 support since last year's Web Almanac. The same increase was observed on both desktop and mobile requests.

Similar to HTTP/2+ adoption, most of the HTTP/3 support originates from CDNs. We expect the support to grow in the future when more CDNs start to support HTTP/3.

Conclusion

This past year was an eventful year for the HTTP protocol especially with HTTP/3 being standardized. We continue to observe high HTTP/2 utilization and see a strong upcoming HTTP/3 support from web servers.

In addition, we have seen strong growth in the ecosystem for technologies that address some of the critical challenges in HTTP/2. 103 Early Hints provides a safer alternative for Server Push and its client support has taken a large step forward with Chrome now supporting it. A new standard for HTTP Prioritization was finalized; major browsers and some web servers already support it for HTTP/3. Furthermore, Priority Hints was added to the web platform to allow clients to further refine prioritization on both HTTP/2 and HTTP/3 and early experiments have yielded promising user experience improvements.

This is an exciting time going forward for the HTTP protocol and the web ecosystem. We are

excited to see how these new technologies will get adopted and what effects they will have on user experience.

Author



Vaspol Ruamviboon

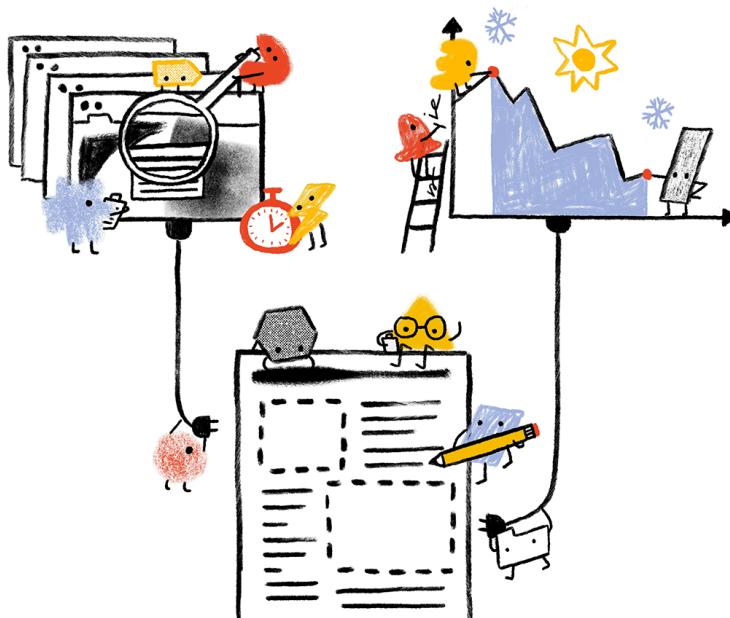
@paivaspol paivaspol vaspol-ruamviboon-7898b824

Vaspol Ruamviboon is a Software Engineer at Microsoft. He completed his PhD from the University of Michigan conducting research on systems to make web pages load faster. You can connect with him on LinkedIn⁸⁷⁰.

870. <https://www.linkedin.com/in/vaspol-ruamviboon-7898b824/>

Appendix A

Methodology



Overview

The Web Almanac is a project organized by HTTP Archive⁸⁷¹. HTTP Archive was started in 2010 by Steve Souders with the mission to track how the web is built. It evaluates the composition of millions of web pages on a monthly basis and makes its terabytes of metadata available for analysis on BigQuery⁸⁷².

The Web Almanac's mission is to become an annual repository of public knowledge about the state of the web. Our goal is to make the data warehouse of HTTP Archive even more

871. <https://httparchive.org>

872. <https://httparchive.org/faq#how-do-i-use-bigquery-to-write-custom-queries-over-the-data>

accessible to the web community by having subject matter experts provide contextualized insights.

The 2022 edition of the Web Almanac is broken into four parts: content, experience, publishing, and distribution. Within each part, several chapters explore their overarching theme from different angles. For example, Part II explores different angles of the user experience in the Performance, Security, and Accessibility chapters, among others.

About the dataset

The HTTP Archive dataset is continuously updating with new data monthly. For the 2022 edition of the Web Almanac, unless otherwise noted in the chapter, all metrics were sourced from the June 2022 crawl. These results are publicly queryable⁸⁷³ on BigQuery in tables prefixed with `2022_06_01`.

All of the metrics presented in the Web Almanac are publicly reproducible using the dataset on BigQuery. You can browse the queries used by all chapters in our GitHub repository⁸⁷⁴.

Please note that some of these queries are quite large and can be expensive⁸⁷⁵ to run yourself. For help controlling your spending, refer to Tim Kadlec's post [Using BigQuery Without Breaking the Bank](#)⁸⁷⁶.

For example, to understand the median number of bytes of JavaScript per desktop and mobile page, see `bytes_2021.sql`⁸⁷⁷:

```
#standardSQL
# Sum of JS request bytes per page (2022)
SELECT
    percentile,
    _TABLE_SUFFIX AS client,
    APPROX_QUANTILES(bytesJs / 1024, 1000)[OFFSET(percentile * 10)] AS js_kilobytes
FROM
```

873. https://github.com/HTTPArchive/httparchive.org/blob/main/docs/gettingstarted_bigquery.md

874. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2022>

875. <https://cloud.google.com/bigquery/pricing>

876. <https://timkadlec.com/remembers/2019-12-10-using-bigquery-without-breaking-the-bank/>

877. https://github.com/HTTPArchive/almanac.httparchive.org/blob/main/sql/2021/javascript/bytes_2021.sql

```

`httparchive.summary_pages.2022_06_01_*`,
UNNEST([10, 25, 50, 75, 90, 100]) AS percentile
GROUP BY
percentile,
client
ORDER BY
client,
percentile

```

Results for each metric are publicly viewable in chapter-specific spreadsheets, for example JavaScript results⁸⁷⁸. Links to the raw results and queries are available at the bottom of each chapter. Metric-specific results and queries are also linked directly from each figure.

Websites

There are 8,360,179 websites in the dataset. Among those, 7,905,956 are mobile websites and 5,428,235 are desktop websites. Most websites are included in both the mobile and desktop subsets.

HTTP Archive sources the URLs for its websites from the Chrome UX Report. The Chrome UX Report is a public dataset from Google that aggregates user experiences across millions of websites actively visited by Chrome users. This gives us a list of websites that are up-to-date and a reflection of real-world web usage. The Chrome UX Report dataset includes a form factor dimension, which we use to get all of the websites accessed by desktop or mobile users.

The June 2022 HTTP Archive crawl used by the Web Almanac used the most recently available Chrome UX Report release for its list of websites. The 202204 dataset was released on May 3, 2022 and captures websites visited by Chrome users during the month of April.

Due to resource limitations, the HTTP Archive previously could only test one page from each website in the Chrome UX report and only home pages were included. Be aware that this will introduce some bias into the results because a home page is not necessarily representative of the entire website. This year, we introduced secondary pages⁸⁷⁹, after the Web Almanac project was beginning and some chapters use this new data. Most chapters, however, used just the

878. https://docs.google.com/spreadsheets/d/1vOeFUYfEtWRen3Xj57zsWav40n5tlcJoVOHmAxmNE_I/edit?usp=sharing

879. <https://discuss.httparchive.org/t/improving-the-http-archive-pipeline-and-dataset-by-10x/2372>

home pages. We expect future analysis to make much more use of this new dataset.

HTTP Archive is also considered a lab testing tool, meaning it tests websites from a datacenter and does not collect data from real-world user experiences. All pages are tested with an empty cache in a logged out state, which may not reflect how real users would access them.

Metrics

HTTP Archive collects thousands of metrics about how the web is built. It includes basic metrics like the number of bytes per page, whether the page was loaded over HTTPS, and individual request and response headers. The majority of these metrics are provided by WebPageTest, which acts as the test runner for each website.

Other testing tools are used to provide more advanced metrics about the page. For example, Lighthouse is used to run audits against the page to analyze its quality in areas like accessibility and SEO. The Tools section below goes into each of these tools in more detail.

To work around some of the inherent limitations of a lab dataset, the Web Almanac also makes use of the Chrome UX Report for metrics on user experiences, especially in the area of web performance.

Some metrics are completely out of reach. For example, we don't necessarily have the ability to detect the tools used to build a website. If a website is built using create-react-app, we could tell that it uses the React framework, but not necessarily that a particular build tool is used. Unless these tools leave detectable fingerprints in the website's code, we're unable to measure their usage.

Other metrics may not necessarily be impossible to measure but are challenging or unreliable. For example, aspects of web design are inherently visual and may be difficult to quantify, like whether a page has an intrusive modal dialog.

Tools

The Web Almanac is made possible with the help of the following open source tools.

WebPageTest

WebPageTest⁸⁸⁰ is a prominent web performance testing tool and the backbone of HTTP

⁸⁸⁰ <https://www.webpagetest.org/>

Archive. We use a private instance⁸⁸¹ of WebPageTest with private test agents, which are the actual browsers that test each web page. Desktop and mobile websites are tested under different configurations:

Config	Desktop	Mobile
Device	Linux VM	Emulated Moto G4
User Agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/102.0.5005.61 Safari/537.36 PTST/220609.133020	Mozilla/5.0 (Linux; Android 8.1.0; Moto G (4)) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/102.0.5005.115 Mobile Safari/537.36 PTST/220609.133020
Location	Google Cloud Locations, USA	Google Cloud Locations, USA
Connection	Cable (5/1 Mbps 28ms RTT)	4G (9 Mbps 170ms RTT)
Viewport	1376 x 768px	512 x 360px

Desktop websites are run from within a desktop Chrome environment on a Linux VM. The network speed is equivalent to a cable connection.

Mobile websites are run from within a mobile Chrome environment on an emulated Moto G4 device with a network speed equivalent to a 4G connection.

Test agents run from various Google Cloud Platform locations⁸⁸² based in the USA.

HTTP Archive's private instance of WebPageTest is kept in sync with the latest public version and augmented with custom metrics⁸⁸³, which are snippets of JavaScript that are evaluated on each website at the end of the test.

The results of each test are made available as a HAR file⁸⁸⁴, a JSON-formatted archive file containing metadata about the web page.

Lighthouse

Lighthouse⁸⁸⁵ is an automated website quality assurance tool built by Google. It audits web pages to make sure they don't include user experience antipatterns like unoptimized images

881. <https://docs.webpagetest.org/private-instances/>

882. <https://cloud.google.com/compute/docs/regions-zones/#locations>

883. <https://github.com/HTTPArchive/custom-metrics>

884. https://en.wikipedia.org/wiki/HAR_file_format

885. <https://developers.google.com/web/tools/lighthouse/>

and inaccessible content.

HTTP Archive runs the latest version of Lighthouse for all pages. This is the first year that Lighthouse testing is done for both mobile and desktop pages. As of the June 2022 crawl, HTTP Archive used the 9.6.2⁸⁸⁶ versions of Lighthouse.

Lighthouse is run as its own distinct test from within WebPageTest, but it has its own configuration profile:

Config	Desktop	Mobile
CPU slowdown	N/A	1x/4x
Download throughput	1.6 Mbps	1.6 Mbps
Upload throughput	0.768 Mbps	0.768 Mbps
RTT	150 ms	150 ms

For more information about Lighthouse and the audits available in HTTP Archive, refer to the Lighthouse developer documentation⁸⁸⁷.

Wappalyzer

Wappalyzer⁸⁸⁸ is a tool for detecting technologies used by web pages. There are 98 categories⁸⁸⁹ of technologies tested, ranging from JavaScript frameworks, to CMS platforms, and even cryptocurrency miners. There are over 3,805 supported technologies (an increase from 2,600 last year).

HTTP Archive runs the latest version of Wappalyzer for all web pages. As of July 2022 the Web Almanac used the 6.10.26 version⁸⁹⁰ of Wappalyzer.

Wappalyzer powers many chapters that analyze the popularity of developer tools like WordPress, Bootstrap, and jQuery. For example, the CMS chapter relies heavily on the respective CMS⁸⁹¹ category of technologies detected by Wappalyzer.

All detection tools, including Wappalyzer, have their limitations. The validity of their results will always depend on how accurate their detection mechanisms are. The Web Almanac will add a note in every chapter where Wappalyzer is used but its analysis may not be accurate due to a

886. <https://github.com/GoogleChrome/lighthouse/releases/tag/v9.6.2>

887. <https://developers.google.com/web/tools/lighthouse/>

888. <https://www.wappalyzer.com/>

889. <https://www.wappalyzer.com/technologies>

890. <https://github.com/AlasI/O/Wappalyzer/releases/tag/v6.10.26>

891. <https://www.wappalyzer.com/categories/cms>

specific reason.

Chrome UX Report

The Chrome UX Report^{[892](#)} is a public dataset of real-world Chrome user experiences. Experiences are grouped by websites' origin, for example <https://www.example.com>. The dataset includes distributions of UX metrics like paint, load, interaction, and layout stability. In addition to grouping by month, experiences may also be sliced by dimensions like country-level geography, form factor (desktop, phone, tablet), and effective connection type (4G, 3G, etc.).

The Chrome UX Report dataset includes relative website ranking data^{[893](#)}. These are referred to as *rank magnitudes* because, as opposed to fine-grained ranks like the #1 or #116 most popular websites, websites are grouped into rank buckets from the top 1k, top 10k, up to the top 10M. Each website is ranked according to the number of eligible^{[894](#)} page views on all of its pages combined. This year's Web Almanac makes extensive use of this new data as a way to explore variations in the way the web is built by site popularity.

For Web Almanac metrics that reference real-world user experience data from the Chrome UX Report, the June 2022 dataset (202206) is used.

You can learn more about the dataset in the Using the Chrome UX Report on BigQuery^{[895](#)} guide on web.dev^{[896](#)}.

Blink Features

Blink Features^{[897](#)} are indicators flagged by Chrome whenever a particular web platform feature is detected to be used.

We use Blink Features to get a different perspective on feature adoption. This data is especially useful to distinguish between features that are implemented on a page and features that are actually used. For example, the CSS chapter's section on Grid layout uses Blink Features data to measure whether some part of the actual page layout is built with Grid. By comparison, many more pages happen to include an unused Grid style in their stylesheets. Both stats are interesting in their own way and tell us something about how the web is built.

Blink Features are reported by WebPageTest as part of our regular testing.

⁸⁹² <https://developers.google.com/web/tools/chrome-user-experience-report>
⁸⁹³ <https://developers.google.com/web/updates/2021/03/crxux-rank-magnitude>
⁸⁹⁴ <https://developer.chrome.com/docs/crxu/methodology/#eligibility>
⁸⁹⁵ <https://web.dev/chrome-ux-report-bigquery>
⁸⁹⁶ <https://web.dev/>
⁸⁹⁷ https://chromium.googlesource.com/chromium/src/+/HEAD/docs/use_counter_wiki.md

Third Party Web

Third Party Web⁸⁹⁸ is a research project by Patrick Hulce, author of the 2019 Third Parties chapter, that uses HTTP Archive and Lighthouse data to identify and analyze the impact of third party resources on the web.

Domains are considered to be a third party provider if they appear on at least 50 unique pages. The project also groups providers by their respective services in categories like ads, analytics, and social.

Several chapters in the Web Almanac use the domains and categories from this dataset to understand the impact of third parties.

Rework CSS

Rework CSS⁸⁹⁹ is a JavaScript-based CSS parser. It takes entire stylesheets and produces a JSON-encoded object distinguishing each individual style rule, selector, directive, and value.

This special purpose tool significantly improved the accuracy of many of the metrics in the CSS chapter. CSS in all external stylesheets and inline style blocks for each page were parsed and queried to make the analysis possible. See this thread⁹⁰⁰ for more information about how it was integrated with the HTTP Archive dataset on BigQuery.

Rework Utils

This year's CSS chapter revisits many of the metrics introduced in 2020's CSS chapter, which was led by Lea Verou. Lea wrote Rework Utils⁹⁰¹ to more easily extract insights from Rework CSS's output. Most of the stats you see in the CSS chapter continue to be powered by these scripts.

Parsel

Parsel⁹⁰² is a CSS selector parser and specificity calculator, originally written by 2020 CSS chapter lead Lea Verou and open sourced as a separate library. It is used extensively in all CSS metrics that relate to selectors and specificity.

898. <https://www.thirdpartyweb.today/>

899. <https://github.com/reworkcss/css>

900. <https://discuss.httparchive.org/t/analyzing-style-sheets-with-a-js-based-parser/1683>

901. <https://github.com/LeaVerou/rework-utils>

902. <https://projects.verou.me/parsel/>

Analytical process

The Web Almanac took about a year to plan and execute with the coordination of more than a hundred contributors from the web community. This section describes why we chose the chapters you see in the Web Almanac, how their metrics were queried, and how they were interpreted.

Planning

The 2022 Web Almanac kicked off in March 2022 with a call for contributors⁹⁰³. We initialized the project with all 26 chapters from previous years and the community suggested additional topics that became two new chapters this year: Interoperability and Sustainability.

As we stated in the inaugural year's Methodology:

One explicit goal for future editions of the Web Almanac is to encourage even more inclusion of underrepresented and heterogeneous voices as authors and peer reviewers.

To that end, this year we've continued our author selection process⁹⁰⁴:

- Previous authors were specifically discouraged from writing again to make room for different perspectives.
- Everyone endorsing 2022 authors were asked to be especially conscious not to nominate people who all look or think alike.
- The project leads reviewed all of the author nominations and made an effort to select authors who will bring new perspectives and amplify the voices of underrepresented groups in the community.

We hope to iterate on this process in the future to ensure that the Web Almanac is a more diverse and inclusive project with contributors from all backgrounds.

Analysis

In April and May 2022, data analysts worked with authors and peer reviewers to come up with a list of metrics that would need to be queried for each chapter. In some cases, custom metrics⁹⁰⁵

903. <https://twitter.com/HTTPArchive/status/1508506002383069192>

904. <https://github.com/HTTPArchive/almanac.httpsarchive.org/discussions/2165>

905. <https://github.com/HTTPArchive/custom-metrics>

were created to fill gaps in our analytic capabilities.

Throughout June 2022, the HTTP Archive data pipeline crawled several million websites, gathering the metadata to be used in the Web Almanac. These results were post-processed and saved to BigQuery⁹⁰⁶.

Being our fourth year, we were able to update and reuse the queries written by previous analysts. Still, there were many new metrics that needed to be written from scratch. You can browse all of the queries by year and chapter in our open source query repository⁹⁰⁷ on GitHub.

Interpretation

Authors worked with analysts to correctly interpret the results and draw appropriate conclusions. As authors wrote their respective chapters, they drew from these statistics to support their framing of the state of the web. Peer reviewers worked with authors to ensure the technical correctness of their analysis.

To make the results more easily understandable to readers, web developers and analysts created data visualizations to embed in the chapter. Some visualizations are simplified to make the points more clearly. For example, rather than showing a full distribution, only a handful of percentiles are shown. Unless otherwise noted, all distributions are summarized using percentiles, especially medians (the 50th percentile), and not averages.

Finally, editors revised the chapters to fix simple grammatical errors and ensure consistency across the reading experience.

Looking ahead

The 2022 edition of the Web Almanac is the fourth in what we hope to continue as an annual tradition in the web community of introspection and a commitment to positive change. Getting to this point has been a monumental effort thanks to many dedicated contributors and we hope to leverage as much of this work as possible to make future editions even more streamlined.

If you're interested in contributing to the 2023 edition of the Web Almanac, please fill out our interest form⁹⁰⁸. Let's work together to track the state of the web!

906. <https://console.cloud.google.com/bigquery?p=httparchive&d=almanac&page=dataset>
907. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2022>
908. <https://forms.gle/zmk6wXfDrmkk2Xa8>

Appendix B

Contributors



The Web Almanac has been made possible by the hard work of the web community. 114 people have volunteered countless hours in the planning, research, writing and production phases of the 2022 Web Almanac.



Aaron Gustafson

✉ @AaronGustafson
⌚ aarongustafson
🌐 https://www.aaron-gustafson.com
Reviewer



Allen O'Neill

✉ @DataBytesAI
⌚ DataBytzAI
📠 allenoneill
🌐 https://webdataworks.io/
Author



Abel Mathew

✉ @DesignrKnight
⌚ DesignrKnight
🌐 http://designrknight.com/
Editor



Alon Kochba

✉ @alonkochba
⌚ alonkochba
📠 alonkochba
Reviewer



Adriana Jara

✉ @tropicadri
⌚ tropicadri
Reviewer



Andrea Volpini

✉ @cyberandy
⌚ cyberandy
🌐 https://wordlift.io/blog/en/entity/
andrea-volpini
Author



Akshay Ranganath

✉ @rakshay
⌚ akshay-ranganath
📠 akshayranganath
🌐 https://akshayranganath.github.io/
Analyst and Author



Arik Smith

⌚ 4upz
Analyst



Alex Denning

✉ @AlexDenning
⌚ alexdenning
🌐 https://getellipsis.com/
Reviewer



Barry Pollard

✉ @tunetheweb
⌚ tunetheweb
📠 tunetheweb
🌐 https://www.tunetheweb.com
Analyst, Developer, Editor, Project Lead,
and Reviewer



Alex N. Jose

✉ @4x13
⌚ alexnj
📠 alexnj
🌐 https://alexnj.com
Reviewer



Belem Zhang

✉ @ibelem
⌚ ibelem
Translator



Chris Lilley

✉ @svgesus
⌚ svgeesus
🌐 https://svgesus.us
Reviewer



Ben Smith

⌚ binji
🌐 https://binji.github.io
Reviewer



Chris Steele

⌚ CSteele-gh
Reviewer



Beth Pan

✉ @beth_panx
⌚ beth-panx
Analyst and Reviewer



Christian Liebel

✉ @christianliebel
⌚ christianliebel
🌐 https://christianliebel.com
Reviewer



Bram Stein

✉ @bram_stein
⌚ bramstein
🌐 http://www.bramstein.com/
Analyst and Author



Cindy Krum

✉ @suzzicks
⌚ Suzzicks
🌐 https://mobilemoxie.com/
Author



Brian Clark

✉ @_clarkio
⌚ clarkio
🌐 https://www.clarkio.com/
Author



Colin Eberhardt

✉ @ColinEberhardt
⌚ ColinEberhardt
🌐 https://blog.scottlogic.com/
ceberhardt/
Author



Brian Kardell

✉ @briankardell
⌚ bkardell
🌐 https://bkardell.com
Author and Reviewer



Colt Sliva

✉ @signorcolt
⌚ csliva
Analyst



Caleb Queern

✉ @httpsecheaders
⌚ cqueern
Reviewer



Dan Knauss

⌚ dknauss
🌐 https://newlocalmedia.com
Editor and Reviewer



Cameron Casher

⌚ camcash17
Analyst



Danielle Rohe

✉ @d4ni_s
⌚ drohe
🌐 https://www.digital-danielle.com
Analyst



Chris Adams

⌚ mrchrisadams
🌐 https://chrisadams.me.uk
Reviewer



Dave Smart

✉ @davewsmart
⌚ dwsmart
🌐 https://tamethebots.com
Author and Reviewer

	<p>David Fox Twitter @theoboto GitHub foxdavidj Website https://www.lookzook.com Project Lead and Reviewer </p>		<p>Gerry McGovern Twitter gerrymcgovernireland Author </p>
	<p>Derek Perkins Twitter derekperkins Website http://derekperkins.com Analyst </p>		<p>Gertjan Franken Twitter GJFR Analyst </p>
	<p>Diego Gonzalez Twitter @diekus GitHub diekus Website https://diek.us Author </p>		<p>Giulia Laco Twitter @webmatter_it GitHub webmatter-it Website https://www.webmatter.it/ Translator </p>
	<p>Edmond de Tournadre Twitter Djohn12 Reviewer </p>		<p>Haren Bhandari Twitter harendra Analyst and Author </p>
	<p>Eric A. Meyer Twitter meyerweb Website http://meyerweb.com/ Reviewer </p>		<p>Hemanth HM Twitter @gnumanth GitHub hemanth Website http://h3manth.com Reviewer </p>
	<p>Eric Portis Twitter eeps Website https://ericportis.com Analyst and Author </p>		<p>Houssein Djirdeh Twitter housseindjirdeh Website https://oussein.me Reviewer </p>
	<p>Estelle Weyl Twitter estelle Website http://standardista.com Reviewer </p>		<p>Ingvar Stepanyan Twitter @RReverser GitHub RReverser Website https://rreverser.com/ Reviewer </p>
	<p>Eugenia Zsigisova Twitter @jevgeniazi GitHub imeugenia Website https://github.com/imeugenia/speaking/blob/main/README.md Author </p>		<p>Iskander Sanchez-Rola Twitter iskander-sanchez-rola GitHub https://iskander-sanchez-rola.com/ Reviewer </p>
	<p>Fershad Irani Twitter @fershad GitHub fershad Website https://www.fershad.com Analyst </p>		<p>Itamar Blauer Twitter @ItamarBlauer GitHub itamarblauer Website https://www.itamarblauer.com/ Author </p>

	JR Oakes ✉ @jroakes 👤 jroakes Analyst		Jono Alderson ✉ @jonoalderson 👤 jonoalderson 🌐 https://www.jonoalderson.com Reviewer
	Jamie Indigo ✉ @Jammer_Volts 👤 fellowhuman1101 🌐 https://not-a-robot.com Author		José Solé ✉ @jmsoleb 👤 jmsole 🌐 https://www.jmsole.cl/ Reviewer
	Jamie Macdonald ✉ JamieWhitMac Analyst		Kai Hollberg ✉ @schweinepriestr 👤 Schweinepriester Reviewer
	Jasmine Drudge-Willson ✉ JasmineDW Editor		Kanmi Obasa ✉ @kanmiobasa 👤 konfirmed 🌐 https://www.knfrm.com Analyst and Reviewer
	Jens Oliver Meiert ✉ @j9t 👤 j9t 🌐 https://meiert.com/en/ Author and Reviewer		Kevin Farrugia ✉ @imkevdev 👤 kevinfarrugia 🌐 https://imkev.dev Analyst and Reviewer
	Jeremy Wagner ✉ @malchata 👤 malchata 🌐 https://jlwagner.net/ Author		Kirsty Simmonds ✉ @keinegurke_ 👤 dereknahman 🌐 https://kirsty.codes Editor
	Joe Viggiano ✉ joeviggiano Analyst and Author		Kushal Das ✉ @kushaldas 👤 kushaldas 🌐 https://kushaldas.in Reviewer
	John Murch ✉ @johnmurch 👤 johnmurch 🌐 http://www.johnmurch.com Reviewer		Laurent Devernay ✉ @ldevernay 👤 ldevernay 🌐 https://ldevernay.github.io/ Author
	Jonathan Wold ✉ @sirjonathan 👤 sirjonathan 🌐 https://jonathanwold.com Author		Laurie Voss 👤 seldo 🌐 http://seldo.com Analyst and Author

	Liran Tal ✉ @liran_tal 👤 lirantal 🌐 https://twitter.com/liran_tal Author		Mobeen Ali ✉ @mobeenali97 👤 mobeenali97 🌐 https://siffar.com Reviewer
	Lucas Pardue ✉ @SimmerVigor 👤 LPardue 🌐 https://lucaspardue.com Reviewer		Mordy Oberstein 👤 mordy-oberstein Author
	Max Ostapenko ✉ @themax_o 👤 max-ostapenko 🌐 https://maxostapenko.com Analyst		Nicolas Hoizey ✉ @nhoizey 👤 nhoizey 🌐 https://nicolas-hoizey.com/ Reviewer
	Maxim Salnikov ✉ @webmaxru 👤 webmaxru 🌐 https://medium.com/@webmaxru Reviewer		Nishu Goel ✉ @TheNishuGoel 👤 NishuGoel 🌐 https://unravelweb.dev/ Analyst and Reviewer
	Melissa Ada ✉ @mel_melificent 👤 mel-ada 🌐 mel-ada Author		Nurullah Demir ✉ @nrllah 👤 nrllh 🌐 https://www.internet-sicherheit.de/ team/demir-nurullah.html Author
	Michael Lewittes 👤 MichaelLewittes Editor		Pankaj Parkar ✉ @pankajparkar 👤 pankajparkar 🌐 https://pankajparkar.dev Reviewer
	Michael Solati ✉ @MichaelSolati 👤 MichaelSolati 🌐 https://michaelsolati.com Author		Patrick Meenan ✉ @patmeenan 👤 pmeenan Reviewer
	Michelle O'Connor Designer		Patrick Stox ✉ @patrickstox 👤 patrickstox 🌐 https://patrickstox.com Reviewer
	Minko Gechev ✉ @mgechev 👤 mgechev 🌐 https://blog.mgechev.com/ Reviewer		Paul Calvano ✉ @paulcalvano 👤 paulcalvano 🌐 https://paulcalvano.com Project Lead



Philip Jägenstedt

⌚ foolip
🌐 https://foolip.org/
Reviewer



Salma Alam-Naylor

🐦 @whitep4nth3r
⌚ whitep4nth3r
🌐 https://whitep4nth3r.com/
Author



Pilar Mera

🐦 @DecreceFeliz
⌚ decrecementofeliz
🌐 http://pi-comunicacion.com/
Translator



Saptak Sengupta

🐦 @Saptak013
⌚ SaptakS
🌐 https://saptaks.website/
Author



Prathamesh Rasam

⌚ 25prathamesh
🌐 prathameshrasam/
Analyst and Reviewer



Scott Davis

🐦 @scottdavis99
⌚ scottdavis99
🌐 http://thirstyhead.com/
Author



Rachel Andrew

🐦 @rachelandrew
⌚ rachelandrew
🌐 https://rachelandrew.co.uk/
Author



Shaina Hantsis

⌚ shantsis
Designer, Editor, and Reviewer



Rick Viscomi

🐦 @rick_viscomi
⌚ rviscomi
Analyst, Author, Editor, and Project Lead



Sia Karamalegos

🐦 @TheGreenGreek
⌚ siakaramalegos
🌐 karamalegos
🌐 https://sia.codes/
Analyst and Project Lead



Rob Teitelman

🐦 @teitelmanrob
⌚ SeoRobt
Reviewer



Simon Pieters

🐦 @zcorpan
⌚ zcorpan
Reviewer



Robin Marx

🐦 @programmingart
⌚ rmarx
🌐 http://internetonmars.org/
Reviewer



Siwin Lo

⌚ siwinlo
Editor



Roel Nieskens

⌚ RoelN
🌐 http://pixelambacht.nl/
Reviewer



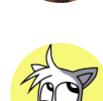
Sophie Brannon

🐦 @SophieBrannon
⌚ SophieBrannon
Author



Sakae Kotaro

🐦 @beltway7
⌚ ksakae1216
🌐 https://ksakae1216.com/
Translator



Thibaud Colas

🐦 @thibaud_colas
⌚ thibaudcolas
🌐 https://thib.me/
Analyst and Author

**Thomas Steiner**

✉ @tomayac
⌚ tomayac
🌐 https://blog.tomayac.com/
Reviewer

**Xavier Jouvenot**

✉ @10xLearner
⌚ Xav83
📠 xavier-jouvenot-98787794
🌐 https://10xlearner.com/
Translator

**Tim Frick**

✉ @timfrick
⌚ timfrick
🌐 https://www.mightybytes.com/
Author

**Yana Dimova**

⌚ ydimova
Analyst

**Tom Van Goethem**

✉ @tomvangoethem
⌚ tomvangoethem
🌐 https://tom.vg/
Author

**Yoav Weiss**

✉ @yoavweiss
⌚ yoavweiss
🌐 https://blog.yoav.ws
Reviewer

**Tushar Pol**

⌚ TusharPol
Reviewer

**Yutaka Oka**

⌚ yokoka
Reviewer

**Vaspol Ruamviboonsuk**

✉ @paivaspol
⌚ paivaspol
📠 vaspol-ruamviboonsuk-7898b824
Analyst and Author

**Zhiwei Li**

⌚ Levix
Translator

**Victor Le Pochat**

✉ @VictorLePochat
⌚ VictorLeP
📠 victor-le-pochat
🌐 https://lepochat.at
Analyst

**Zongchao Bai**

⌚ luckybai
Translator

**Vik Vanderlinden**

⌚ vikvanderlinden
Analyst