Lazy loading images and scripts

If you have ever waited for a bloated webpage to crawl to life, you already know the enemy: everything loads at once. **Lazy loading images and scripts** is the tactical decision to hold back assets the user cannot see yet. Instead of downloading 12 hero images and 8 analytics scripts the second the browser touches the HTML, you wait. You wait until the user scrolls down, or until a specific interaction triggers the resource. It is not complicated, but the implementation details separate a snappy site from a broken one.

The difference between what you see and what you pay for

Every byte that hits the network costs time. On a 3G connection, a 2 MB hero image can stall the entire page for seconds. The mental model is simple: the browser has a single main thread. If it is busy decoding a massive JPEG below the fold, it cannot render the text the user actually wants to read. Deferred loading shifts that work to idle time.

There are two distinct categories here. Images are relatively safe to defer because they are passive content. Scripts are dangerous because they can modify the DOM, fire events, or depend on execution order. You handle them differently. An <img loading="lazy"> attribute is a native browser instruction. A <script defer> or <script async> changes when the JavaScript runs. Mixing these up causes layout shifts, missing functionality, or broken carousels.

Native lazy loading vs. Intersection Observer vs. library-based approaches

You have three real paths for images. Native lazy loading is the simplest: <img src="photo.jpg" loading="lazy" alt="..."><. It works in all modern browsers and requires zero JavaScript. The trade-off is that you cannot control the threshold precisely. The browser decides when to start fetching based on its own heuristics, which are usually fine but not always optimal for above-the-fold content.

Intersection Observer gives you surgical control. You write a small JavaScript snippet that watches an element and triggers loading when it enters a defined viewport margin. This is the right choice if you need to load images 200px before they appear, or if you want to add a fade-in effect. The cost is code complexity and a tiny performance overhead from the observer itself.

Library-based solutions like lazysizes or lozad are overkill for most projects today. They were useful back when loading="lazy" did not exist. Now they add kilobytes of JavaScript to solve a problem the browser already solved. Unless you need polyfills for very old browsers, skip the library. Stick to native attributes and reserve Intersection Observer for edge cases like background images in CSS.

Script deferment: the order of operations matters more than you think

Scripts block parsing. That is the fundamental rule. When the browser encounters a <script> tag without async or defer, it stops building the DOM, downloads the script, executes it, and only then continues. For third-party scripts like analytics, social widgets, or chat widgets, this blocking behavior destroys page speed.

Use defer for scripts that need to run after the document is parsed but before DOMContentLoaded. Use async for scripts that are completely independent, like a tracking pixel that does not touch the DOM. A common mistake is to defer everything, including critical scripts that set up event listeners for the initial view. If your navigation menu depends on a JavaScript file, deferring it might cause a flash of unstyled content or a broken click handler. Test with throttled network conditions before assuming defer is safe.

> Rule of thumb: defer any script that is not required for the first paint. Async any script that is completely isolated. Block only the scripts that render visible UI.

Four mistakes that turn deferred loading into a user experience disaster

First, lazy loading above-the-fold content. If an image is visible when the page loads, do not lazy load it. The browser will see it, defer it, and then load it immediately anyway, adding an unnecessary round trip. Measure the viewport height and apply lazy loading only to elements below that line.

Second, forgetting about cumulative layout shift. When an image loads late, it pushes content down. The user is reading a paragraph, and suddenly it jumps. Set explicit width and height attributes on every lazy-loaded image, or use CSS aspect-ratio boxes. The browser needs to reserve the space before the image loads.

Third, lazy loading everything on a blog page. If a page has 40 small thumbnails in a gallery, lazy loading them is smart. If the page has 3 images and a lot of text, the performance gain is negligible, and you risk delaying the images for no reason. Be selective.

Fourth, loading scripts on every page when they are only needed on one. A contact form widget script should not load on the homepage. Use conditional loading based on the page URL or a data attribute. This is not strictly lazy loading, but it is the same principle: do not fetch what you do not use.

Real-world scenarios: when to push and when to pull back

Consider an e-commerce product listing page with 60 product cards, each containing a thumbnail, a title, a price, and an "add to cart" button. The thumbnails are small, maybe 200x200 pixels. Lazy loading these images with loading="lazy" reduces the initial page weight by roughly 80%, because only the first 8-10 thumbnails are visible. The user scrolls, and the rest load just in time. The "add to cart" JavaScript, however, should be loaded

immediately because it is critical for interaction. Deferring it would mean the button does nothing until the user scrolls down and triggers the script, which feels broken.

Now consider a news article page with a single featured image and a lot of text. The featured image is above the fold. Do not lazy load it. The rest of the page has no images. The only script is a Google Analytics snippet. Defer that snippet. It does not affect the user experience at all, and deferring it shaves off 30-50 milliseconds of blocking time. The difference is measurable in Lighthouse scores but barely perceptible to the user. Still, every millisecond counts when you are competing for attention on a slow connection.

Frequently asked questions about deferred asset loading

Does lazy loading work for background images in CSS?

No, the loading attribute only applies to <img> and <iframe> elements. For CSS background images, you need Intersection Observer to swap the background-image property when the element enters the viewport.

Can lazy loading hurt SEO?

Googlebot processes lazy-loaded content as long as it is accessible through standard HTML or JavaScript. If you hide images behind JavaScript that requires a click or a scroll event that the bot cannot trigger, those images may not be indexed. Use native lazy loading or ensure your Intersection Observer has a generous margin so the bot can see the content.

Should I lazy load all images on a mobile version of a site?

Mobile viewports are smaller, so more images are below the fold. Lazy loading on mobile is generally more beneficial than on desktop, but the same rule applies: do not lazy load the first visible image.

What happens if a user has JavaScript disabled?

Native loading="lazy" works without JavaScript. Intersection Observer approaches fail. If your site depends on JavaScript for lazy loading, provide a fallback with <noscript> tags that load images normally.

Stop optimizing what does not matter and focus on the bottleneck

The real performance win is not in micro-optimizing every image. It is in identifying the single largest asset that blocks the page and deferring it. For most sites, that is a hero image or a third-party script. Run a Lighthouse audit, look at the "Largest Contentful Paint" element, and lazy load everything below it. That single change will move your performance score more than 20 points. The rest is polish.

Technical Verification Node

recommended tool

Report ID: 2C6F99F1 | Signature: 4c33af1930a9f45999566b78f568c427