Web Performance and Core Web Vitals

Version Core Web Vitals 2026 · verified 2026-11-02

Measure and fix frontend speed by metric — LCP, INP and CLS thresholds, field vs lab data, and the fixes that move each one.

Contents

Mental model 2

The three vitals 2

Lab vs field data 3

LCP and the critical path 4

Images and fonts 5

JavaScript cost 6

INP and long tasks 7

CLS and layout stability 9

Caching and the CDN 10

Budgets in CI 10

Further reading 12

Mental model

Each Core Web Vital points at one bottleneck. LCP is the critical path: how fast the biggest thing on screen arrives and paints. INP is main-thread work: how long the page is too busy to answer a tap. CLS is layout stability: whether space was reserved before content arrived. Diagnose by metric first, then fix that bottleneck, because a fix for one rarely moves the others.

The three vitals

Metric	Good	Poor
LCP — Largest Contentful Paint	≤ 2.5 s	> 4.0 s
INP — Interaction to Next Paint	≤ 200 ms	> 500 ms
CLS — Cumulative Layout Shift	≤ 0.1	> 0.25

Between the two columns is “needs improvement”. A page passes only when all three are good at the **75th percentile** of real visits, measured separately for mobile and desktop. INP replaced First Input Delay as a Core Web Vital in 2024; `web-vitals` 5 removed `onFID()`.

```
LCP  load    -> server, network, render path
INP  respond -> JavaScript on the main thread
CLS  stable  -> missing sizes, late inserts
pass = all three "good" at p75, per device
```

> **Gotcha:** The 75th percentile means a quarter of your users can have a bad experience while you still pass. Look at p75 on mobile first; it is usually the failing one.

Lab vs field data

Source	Kind	Use it to
CrUX, PageSpeed Insights	Field	See what Google sees
`web-vitals` library	Field (RUM)	Find which page and element
Lighthouse, DevTools	Lab	Reproduce and debug
Total Blocking Time	Lab	Proxy for INP

Field data is the truth and the ranking signal; lab data is a single synthetic load you can debug. Lab runs have no real interactions, so they cannot measure INP. The `web-vitals` attribution build tells you which element and which phase caused a bad value.

```
import {
  onCLS, onINP, onLCP,
} from "web-vitals/attribution";

function send(m) {
  const body = JSON.stringify({
    name: m.name, value: m.value, id: m.id,
    rating: m.rating, // "good" | ...
```

```
  });
  navigator.sendBeacon("/rum", body);
}
onCLS(send); onINP(send); onLCP(send);
```

> **Note:** The LCP and Event Timing APIs (behind INP) reached Baseline in December 2025 with Safari 26.2, so RUM for those two now covers all engines. CLS still reports from Chromium only, and CrUX holds Chrome users only.

LCP and the critical path

Hint	Does	Baseline
`fetchpriority="high"`	Boosts one fetch	Newly 2024
`<link rel="preload">`	Fetch a late-found resource early	Widely
`<link rel="preconnect">`	Warm DNS, TCP, TLS	Widely
`<link rel="modulepreload">`	Fetch a module graph early	Widely
`<script defer>`	Run after parse, no block	Widely

LCP splits into four parts: time to first byte, resource load delay, resource load duration and element render delay. Most slow LCPs are load delay: the browser found the hero image late, because it sat in CSS, JavaScript or a lazy-loaded tag.

```
<link rel="preconnect"
  href="https://img.example.com">
<link rel="stylesheet" href="/app.css">
<script src="/app.js" defer></script>

<img src="https://img.example.com/hero.avif"
  width="1200" height="600"
  fetchpriority="high" alt="Spring sale">
```

> **Gotcha:** `loading="lazy"` on the LCP image delays it until layout proves it is in the viewport. Lazy-load only below-the-fold images.

Images and fonts

Technique	Fixes
`width` + `height` on `<img>`	CLS: space reserved
`srcset` + `sizes`	LCP: right bytes per screen
AVIF or WebP	LCP: fewer bytes
`loading="lazy"` below the fold	Bandwidth for the hero
`font-display: swap`	Text visible during font load
Preload one critical font	Earlier font, less swap shift

The browser derives an aspect ratio from `width` and `height` even when CSS makes the image fluid, so declare both on every image. Subset web fonts and ship WOFF2.

```
<link rel="preload" as="font" type="font/woff2"
  href="/fonts/inter.woff2" crossorigin>
<style>
  @font-face {
    font-family: Inter;
    src: url(/fonts/inter.woff2) format("woff2");
    font-display: swap;
  }
  img { max-width: 100%; height: auto; }
</style>
```

> **Gotcha:** Font preloads need `crossorigin` even on your own origin. Fonts are fetched in CORS mode, so without it the preload is not reused and the font downloads twice.

JavaScript cost

Technique	Effect
`import()` on demand	Splits a chunk off the entry
Route-level splitting	Each page ships its own code
`defer` / `type="module"`	Parse HTML without blocking
Drop or delay third parties	Less main-thread work
Bundle analyzer in CI	Catch size regressions

Every kilobyte of JavaScript costs download, parse, compile and execute time, and the last three land on the main thread. Load code when the user needs it, not when the page starts.

```
const btn = document.querySelector("#chart");
btn.addEventListener("click", async () => {
  const { drawChart } = await import(
    "./chart.js"
  );
  drawChart(document.querySelector("#out"));
});
```

> **Tip:** Module scripts are deferred by default. `async` on a classic script runs it as soon as it arrives, which can block parsing at an unpredictable moment.

INP and long tasks

INP phase	Caused by	Fix
Input delay	Other tasks running	Break up long tasks
Processing	Your handlers	Do less, defer the rest
Presentation delay	Rendering, big DOM	Smaller DOM, less layout

INP is the slowest interaction of the visit (ignoring one outlier per 50 interactions), from input to the next frame. A task over 50 ms is a long

task. Paint the visible response first, then yield to the browser before the expensive work.

```
function yieldToMain() {
  if (globalThis.scheduler?.yield) {
    return scheduler.yield();
  }
  return new Promise((r) => setTimeout(r, 0));
}
async function onSave(btn, items) {
  btn.textContent = "Saving…"; // visible first
  await yieldToMain();
  for (const item of items) {
    process(item);
    await yieldToMain();
  }
}
function process(item) { /* heavy work */ }
```

> **Note:** `scheduler.yield()` is not Baseline (no Safari), so keep the `setTimeout` fallback.

CLS and layout stability

Cause	Fix
Image or video without size	`width` / `height` or `aspect-ratio`
Ad, embed, banner inserted late	Reserve a min-height slot
Web font swap	Matched fallback via `size-adjust`
Animating `top` / `height`	Animate `transform` instead

A layout shift counts when a visible element moves without user input; shifts within 500 ms of a tap or keypress are excluded. CLS is the largest burst of shifts within a session window, not the page total.

```
.ad-slot { min-height: 250px; }
.embed { aspect-ratio: 16 / 9; width: 100%; }

@font-face {
  font-family: "Inter Fallback";
  src: local("Arial");
  size-adjust: 107%;
}
body { font-family: Inter, "Inter Fallback"; }
```

> **Gotcha:** Content inserted above the viewport after load, such as a cookie bar pushing the page down, is the classic CLS bug. Overlay it with `position: fixed` instead.

Caching and the CDN

Asset	`Cache-Control`
Hashed JS, CSS, fonts	`public, max-age=31536000, immutable`
HTML	`no-cache` (revalidate every time)
API JSON	`private, max-age=0` or short `s-maxage`

Put a content hash in every static filename so it can be cached for a year and never revalidated; the HTML points to the new hash on deploy. Serve from a CDN close to users to cut TTFB, which is the first part of LCP. HTTP caching semantics are in the HTTP sheet.

```
curl -sI https://example.com/assets/app.3f9a.js \
  | grep -i -E "cache-control|age|content-enc"
```

> **Gotcha:** `no-cache` does not mean “do not store”. It means “revalidate before use”. `no-store` disables caching entirely.

Budgets in CI

Assertion key	Typical budget
`largest-contentful-paint`	`maxNumericValue: 2500`
`cumulative-layout-shift`	`maxNumericValue: 0.1`
`total-blocking-time`	`maxNumericValue: 200`
`categories:performance`	`minScore: 0.9`

Lighthouse CI runs Lighthouse several times per URL and fails the build when an assertion breaks, so regressions surface in the pull request. Lab budgets catch regressions; only field data proves the fix.

```
{
  "ci": {
    "collect": { "numberOfRuns": 3 },
    "assert": {
      "assertions": {
        "largest-contentful-paint":
          ["error", { "maxNumericValue": 2500 }],
        "cumulative-layout-shift":
          ["error", { "maxNumericValue": 0.1 }],
        "total-blocking-time":
          ["warn", { "maxNumericValue": 200 }]
      }
    }
  }
}
```

> **Gotcha:** Lighthouse 12 removed `budget.json` performance budgets, and Lighthouse 13 replaced many performance audits with insights. Assert on metric audit IDs, which survive those changes.

Further reading

- web.dev — Web Vitals
- web.dev — Optimize LCP
- web.dev — Optimize INP
- web.dev — Optimize CLS
- GoogleChrome/web-vitals
- Lighthouse CI — configuration