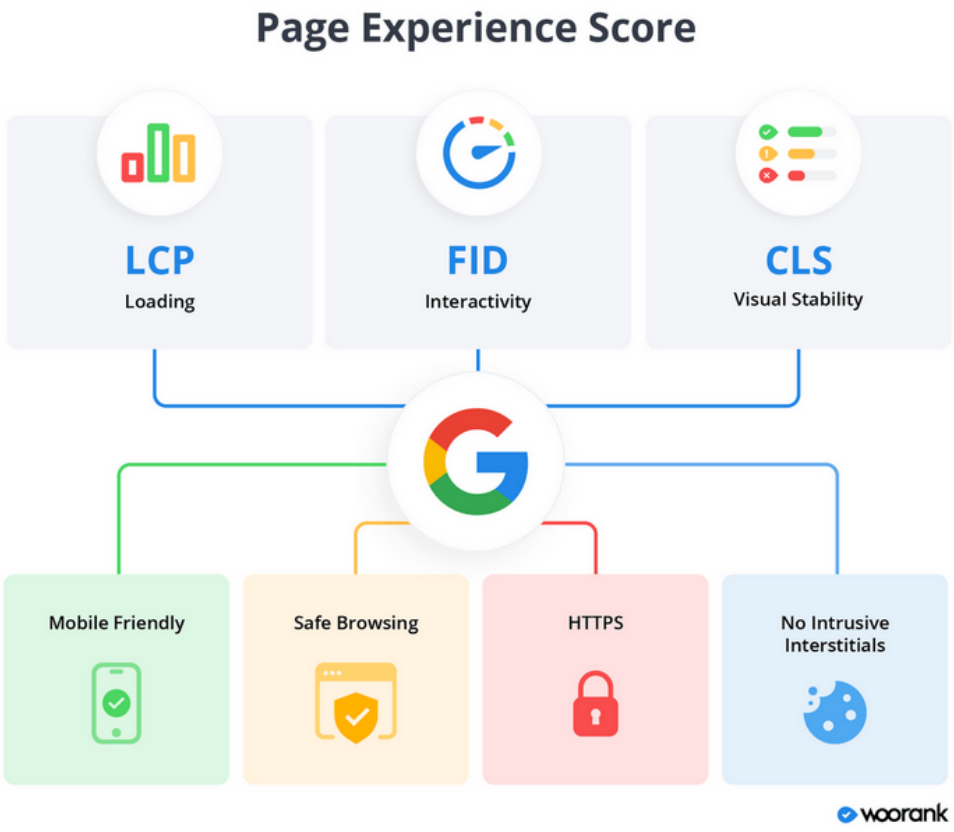
Web Vitals

By : Nour Abd Al-Aziz

Supervised By : Rami Theeb

Google's Page Experience

Page experience is a set of signals that measure how users perceive the experience of interacting with a web page beyond its pure information value, both on mobile and desktop devices. It includes Core Web Vitals, which is a set of metrics that measure real-world user experience for loading performance, interactivity, and visual stability of the page. It also includes existing Search signals: mobile-friendliness, HTTPS, and intrusive interstitial guidelines.



What is ‘ Web Vitals’ ?

Web Vitals is an initiative by Google to provide unified guidance for quality signals that are essential to delivering a great user experience on the web.

The Web Vitals metrics are split into Core Web Vitals and non-Core Web Vitals.

The Core Web Vitals are:	The non-Core Web Vitals are:
• Largest Contentful Paint (LCP) • First Input Delay (FID) • Cumulative Layout Shift (CLS)	• Total Blocking Time (TBT) • First Contentful Paint (FCP) • Speed Index (SI) • Time to Interactive (TTI)

Why did Web Vitals come out?

There is something we should always be doing (and have been doing even before web vitals appeared), that is, making sure that websites are providing a great user experience.

This goal needs some extra efforts from developers to measure and report on performance and understand the quality of user experience. Fortunately, Google has provided a number of tools over the years to achieve that, but the issue is not all developers are experts with these tools, there are always challenges to keep up with.

So, in May 2020, Google announced that they would be releasing a new ranking factor related to overall page experience (how quickly pages load, mobile-friendliness, etc…). Included in the existing page experience ranking signals are a set of metrics called Core Web Vitals.

Hence, there's no worries , with web vitals there's no need to be a performance expert. The Web Vitals initiative aims to simplify the landscape, and help sites focus on the metrics that matter most (Core Web Vitals).

Core Web Vitals

Core Web Vitals, a subset of Web Vitals, helps you to judge your website's UX against a distinct set of UX metrics. Google intends to evolve Core Web Vitals over time. However, as of 2021, the current set of Core Web Vitals focuses on the following three areas of UX: loading, interactivity, and visual stability. These three areas are measured as:

- Largest Contentful Paint (LCP)
- First Input Delay (FID)
- Cumulative Layout Shift (CLS).

Why are Core Web Vitals important ?

Since 2015, Google has been increasing the importance of good website UX as a Search Ranking Signal. This puts more emphasis on website owners to ensure their website UX is excellent if they want to be rewarded with good Google search rankings.

It's important to understand that Core Web Vitals became a Google ranking factor in August 2021. The score you get on these metrics can make a difference in how prominently you appear within Google's own search engine results pages for your targeted search queries.

Largest Contentful Paint (LCP)

- **LCP Definition**

Largest Contentful Paint (LCP) is an important, *user-centric metric* for measuring *perceived load speed* because it marks the point in the page load timeline when the page's main content has likely loaded—a fast LCP helps reassure the user that the page is *useful*.

- User-centric metrics : User-Centric Performance Metrics are those performance metrics that show us how users perceive our applications by measuring what affects user experience the most.
- Perceived load speed: how quickly a page can load and render all of its visual elements to the screen.
- Is useful : Has enough content rendered that users can engage with it?

- **Why did experts choose LCP to measure page speed loading**

There are other metrics to measure how quickly the main content of a web page loads and is visible to users, the following are some of them with their drawbacks:

- **load or DOMContentLoaded**: not good because they don't necessarily correspond to what the user sees on their screen.

- **First Contentful Paint (FCP)** : only capture the very beginning of the loading experience. If a page shows a splash screen or displays a loading indicator, this moment is not very relevant to the user.

- **First Meaningful Paint (FMP) and Speed Index (SI)** : these metrics are complex, hard to explain, and often wrong—meaning they still do not identify when the main content of the page has loaded.

Hence, experts found that a more accurate way to measure when the main content of a page is loaded is to look at when the largest element was rendered.

- **Good Score for LCP**

  To provide a good user experience, sites should strive to have Largest Contentful Paint of 2.5 seconds or less. To ensure you're hitting this target for most of your users, a good threshold to measure is the 75th percentile of page loads, segmented across mobile and desktop devices.

- **What could be causing a poor LCP score?**

There could be any of a myriad of causes for a poor LCP score, such as:

- Slow server response times
- Render-blocking JavaScript and CSS
- Resource load times

- **Optimize Largest Contentful Paint (LCP)**

  - **Reduce server response times**

If your server takes long to respond to a request, then the time it takes to render the page on the screen also goes up. It, therefore, negatively affects every page speed metric, including LCP. To improve your server response time, here is what to do :

1. Analyze and optimize your servers

A lot of computation, DB queries, and page construction happens on the server. You should analyze the requests going to your servers and identify the possible bottlenecks for responding to the requests. It could be a DB query slowing things down or the building of the page on your server.

2. Use a Content Delivery Network (CDN)

While your server sits in one location, a CDN consists of globally distributed servers that can store your website's content (or commonly referred to as "cache" your website's content) and deliver it to your users. So now, if users sitting in California access your website, instead of downloading the content from North Virginia, they can get it from one of the servers of the CDN, a few miles from their location. Shorter distance, super fast load time, happier users, and more sales for you.

3. **Compress text files**

Any text-based data you load on your webpage should be compressed when transferred over the network using a compression algorithm like gzip or Brotli. SVGs, JSONs, API responses, JS and CSS files, and your main page's HTML are good candidates for compression using these algorithms. This compression significantly reduces the amount of data that will get downloaded on page load, therefore bringing down the LCP.

4. Configure Caching

Caching ensures fast delivery to visitors. Without caching, a browser requests assets from the server each time a page loads instead of accessing them from a local or intermediary cache.

There are WordPress plugins that enable storing files locally on a user's computer. The files are then reused during future visits. This practice is called caching. It speeds up loading time and ensures better UX.

HTTP caching can speed up your page load time on repeat visits.

When a browser requests a resource, the server providing the resource can tell the browser how long it should temporarily store or *cache* the resource. For any subsequent request for that resource, the browser uses its local copy rather than getting it from the network.

- **Remove render-blocking resources**

When the browser receives the HTML page from your server, it parses the DOM tree. If there is any external stylesheet or JS file in the DOM, the browser has to pause for them before moving ahead with the parsing of the remaining DOM tree.

These JS and CSS files are called render-blocking resources and delay the LCP time. Here are some ways to reduce the blocking time for JS and CSS files:

1. Do not load unnecessary bundles

Avoid shipping huge bundles of JS and CSS files to the browser if they are not needed. If the CSS can be downloaded a lot later, or a JS functionality is not needed on a particular page, there is no reason to load it up front and block the render in the browser.

Suppose you cannot split a particular file into smaller bundles, but it is not critical to the functioning of the page either. In that case, you can use the defer attribute of the script tag to indicate to the browser that it can go ahead with the DOM parsing and continue to execute the JS file at a later stage. Adding the defer attribute removes any blocker for DOM parsing. The LCP, therefore, goes down.

2. Inline critical CSS

Critical CSS comprises the style definitions needed for the DOM that appears in the first fold of your page. If the style definitions for this part of the page are inline, i.e., in each element's style attribute, the browser has no dependency on the external CSS to style these elements. Therefore, it can render the page quickly, and the LCP goes down.

3. Minify and compress the content

You should always minify the CSS and JS files before loading them in the browser. CSS and JS files contain whitespace to make them legible, but they are unnecessary for code execution. So, you can remove them, which reduces the file size on production. The minified version of a webpage is transmitted instead of the full version when a user requests it, resulting in faster response times and lower bandwidth costs.

Use data compression algorithms to bring down the file size delivered over the network. Gzip and Brotli are two compression algorithms. Brotli compression offers a superior compression ratio compared to Gzip and is now supported on all major browsers, servers, and CDNs.

- **Reduce resource load time**

  The goal of this step is to reduce the time spent transferring the bytes of the resource over the network to the user's device. In general, there are three ways to do that:

  1. Reduce the size of the resource

  On most websites, the above-the-fold content usually contains a large image which gets considered for LCP. It could either be a hero image, a banner, or a carousel. It is, therefore, crucial that you optimize these images for a better LCP.

  To optimize your images, you should use a third-party image CDN like ImageKit.io. The advantage of using a third-party image CDN is that you can focus on your actual business and leave image optimization to the image CDN.

  It helps in delivering your images in lighter formats, providing real-time transformations for responsive images and automatically compressing your images.

  2. Reduce the distance the resource has to travel

  In addition to reducing the size of a resource, you can also reduce the load times by getting your servers as geographically close to your users as possible. And the best way to do that is to use a content delivery network (CDN).

  3. Reduce contention for network bandwidth

  Even if you've reduced the size of your resource and the distance it has to travel, a resource can still take a long time to load if you're loading many other resources at the same time. This problem is know as *network contention*.

  If you've given your LCP resource a high “ fetchpriority ” attribute value, and started loading it as soon as possible then the browser will do its best to prevent lower-priority resources from competing with it.

- **What elements are considered for LCP and how does an element's size determined ?**

  The types of elements considered for Largest Contentful Paint are:

  - <img> elements
  - <video> elements (the poster image is used)
  - An element with a background image loaded via the url() function (as opposed to a CSS gradient)
  - Block-level elements containing text nodes or other inline-level text elements children.

  In general, the *size of the element reported* for the LCP is typically the size that's visible to the user within the viewport.

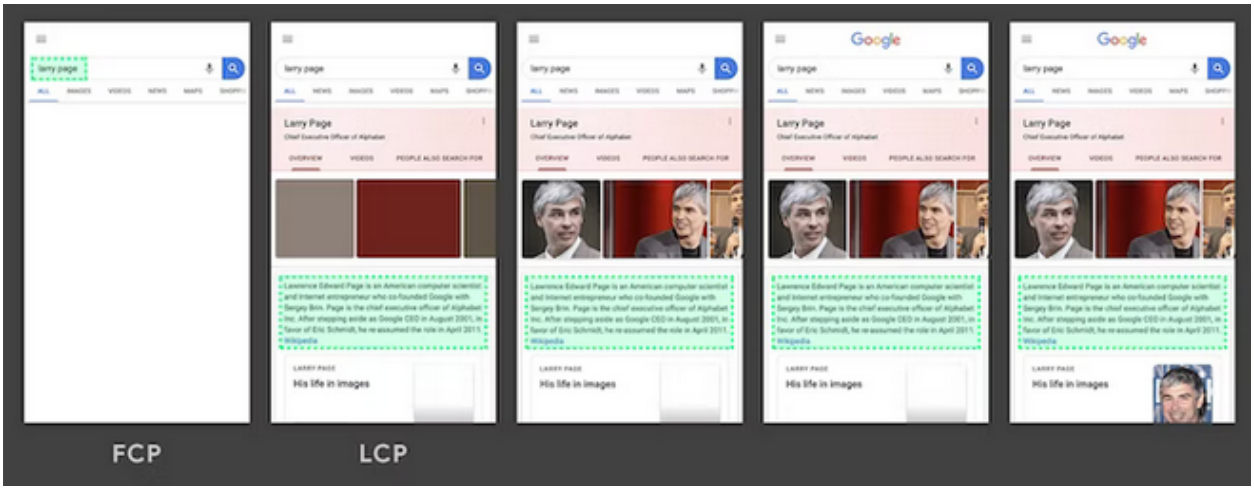
- **When is the largest contentful paint reported?**

  Web pages often load in stages, and as a result, it's possible that the largest element on the page might change.

  To handle this potential for change, the browser dispatches a *PerformanceEntry* of type *largest-contentful-paint* identifying the largest contentful element as soon as the browser has painted the first frame. But then, after rendering subsequent frames, it will dispatch another *PerformanceEntry* any time the largest contentful element changes.

For example, on a page with text and a hero image the browser may initially just render the text—at which point the browser would dispatch a largest-contentful-paint entry whose element property would likely reference a <p> or <h1>. Later, once the hero image finishes loading, a second largest-contentful-paint entry would be dispatched and its element property would reference the <img>.

In the following example, the largest element is a paragraph of text that is displayed before any of the images or logo finish loading. Since all the individual images are smaller than this paragraph, it remains the largest element throughout the load process.



First Input Delay (FID)

- **FID Definition**

In general, FID is an important, *user-centric* metric that captures a user's first impression of a site's interactivity and responsiveness. A low FID helps ensure that the page is *usable*. It measures the speed at which users are able to interact with a page after landing on it.

In other words, FID measures the time from when a user first interacts with a page (click a link, tap on a button, etc.) to the time when the browser is actually able to begin processing event handlers in response to that interaction.

- **Good score for FID**

To provide a good user experience, sites should strive to have a First Input Delay of 100 milliseconds or less.

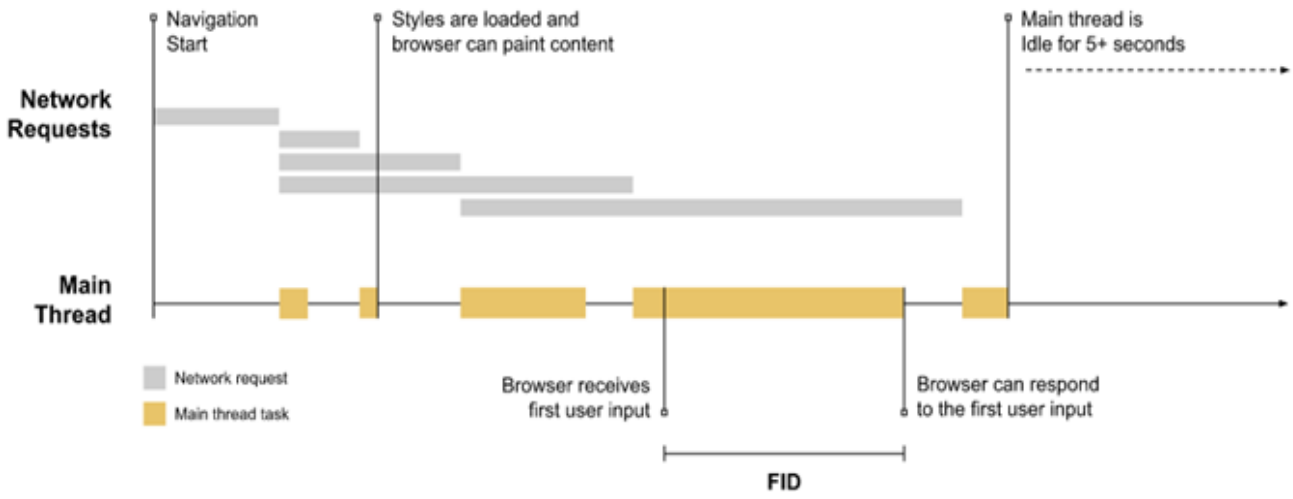
- **Why does input delay happen**

Input delay happens because the browser's main thread is busy doing something else, so it can't (yet) respond to the user.

One common reason this might happen is the browser is busy parsing and executing a large JavaScript file loaded by your app. While it's doing that, it can't run any event listeners because the JavaScript it's loading might tell it to do something else.

If a user tries to interact with the page during that time (e.g. click on a link), there will be a delay between when the click is received and when the main thread is able to respond.

Because the input occurs while the browser is in the middle of running a task, it has to wait until the task completes before it can respond to the input. The time it must wait is the FID value for this user on this page.



- **What counts as a first input**

FID is a metric that measures a page's responsiveness during load. As such, it only focuses on input events from discrete actions like *clicks, taps, and key presses*.

Other interactions, like scrolling and zooming, are continuous actions and have completely different performance constraints (also, browsers are often able to hide their latency by running them on a separate thread).

- **What if an interaction doesn't have an event listener?**

FID is measured even in cases where an event listener has not been registered; that's because:

- FID measures the delta between when an input event is received and when the main thread is next idle.
- Many user interactions do not require an event listener but *do* require the main thread to be idle in order to run.

For example, all of the following HTML elements need to wait for in-progress tasks on the main thread to complete prior to responding to user interactions:

- Text fields, checkboxes, and radio buttons (<input>, <textarea>)
- Select dropdowns (<select>)

- **Optimize First Input Delay (FID)**

  The main cause of a poor FID is heavy JavaScript execution. Optimizing how JavaScript parses, compiles, and executes on your web page will directly reduce FID.

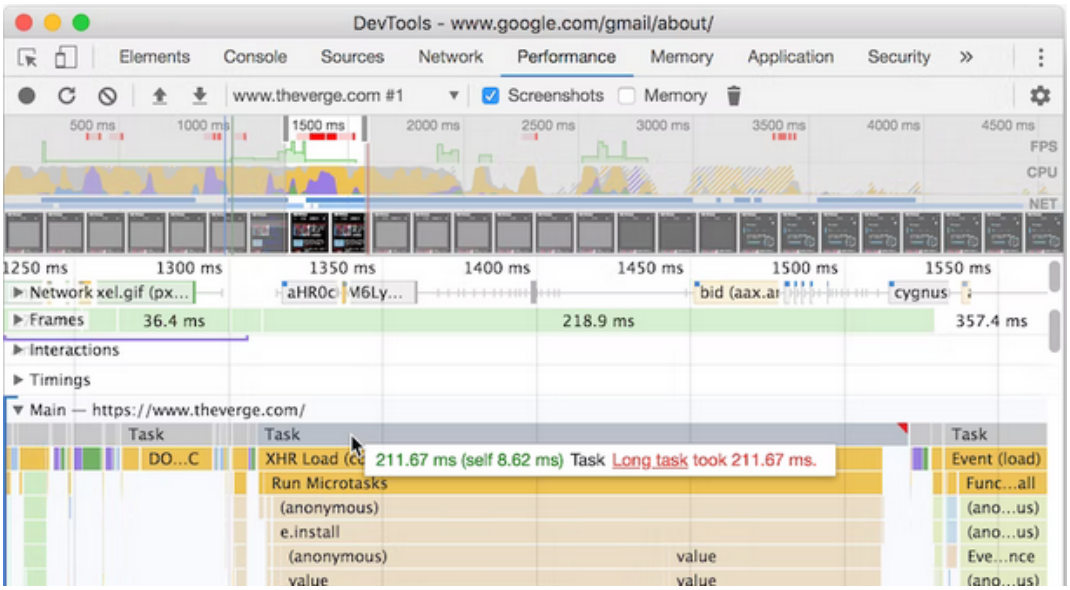
1- Break up Long Tasks:

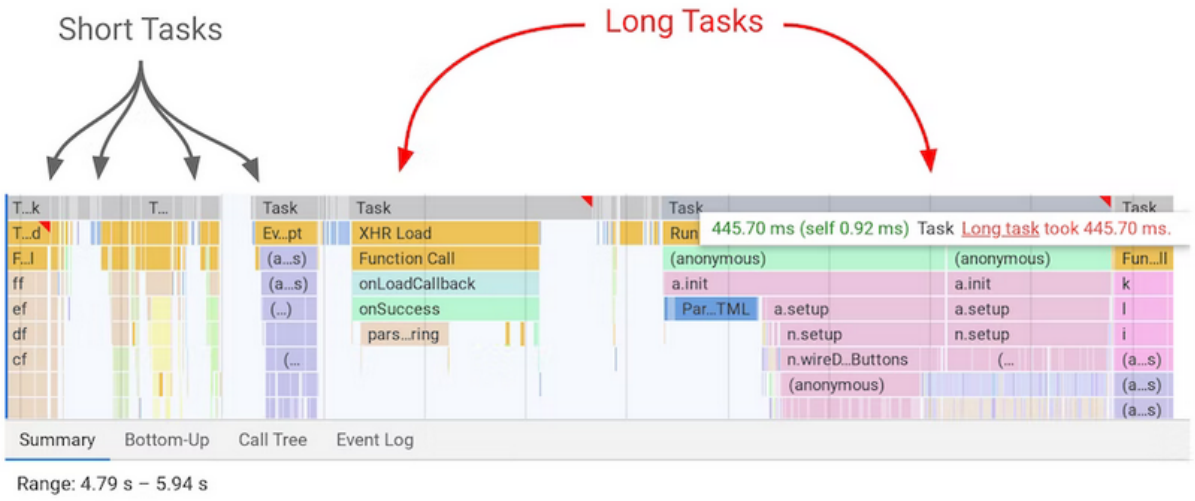
Long task could be any piece of code that blocks the main thread for 50 ms or more due to complex work, causing the UI to "freeze".

While a web page is loading, Long Tasks can tie up the main thread and make the page unresponsive to user input even if it looks ready. Clicks and taps often don't work because event listeners, click handlers etc have not yet been attached.

Are there Long Tasks in my page that could delay interactivity?

DevTools now visualizes Long Tasks, tasks (shown in gray) have red flags if they are Long Tasks (in Chrome 83 the Long Task visualization UI in the Performance panel has been updated. The Long Task portion of a task is now colored with a striped red background) .



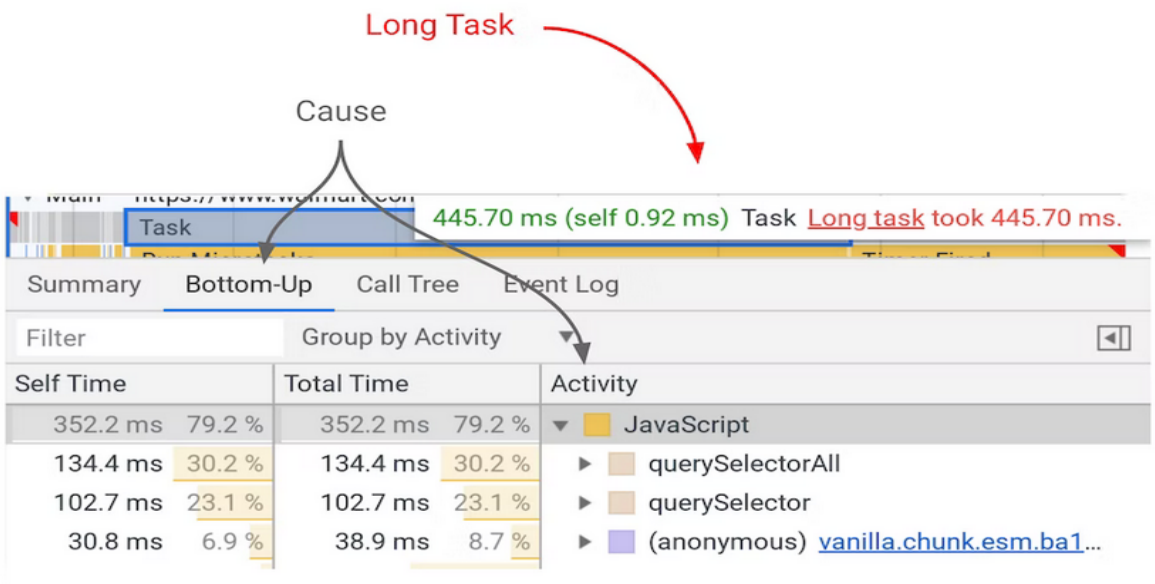


- Record a trace in the Performance panel of loading up a web page.
- Look for a red flag in the main thread view. You should see tasks are now gray ("Task").
- Hovering over a bar will let you know the duration of the task and if it was considered "long".

Visit https://developer.chrome.com/docs/devtools/evaluate-performance/

Knowing What is causing my Long Tasks using Chrome DevTools

To discover what is causing a long task, select the gray Task bar. In the drawer beneath, select Bottom-Up and Group by Activity. This allows you to see what activities contributed the most (in total) to the task taking so long to complete. Below, it appears to be a costly set of DOM queries.



What are common ways to optimize Long Tasks?

- Large scripts are often a major cause of Long Tasks so consider splitting them up.
- Also keep an eye on third-party scripts; their Long Tasks can delay primary content from getting interactive.
- Break all your work into small chunks (that run in < 50ms) and run these chunks at the right place and time; the right place may even be off the main thread, in a worker.

Code splitting by dynamic imports

- Sending large JavaScript payloads impacts the speed of your site significantly. Instead of shipping all the JavaScript to your user as soon as the first page of your application is loaded, split your bundle into multiple pieces and only send what's necessary at the very beginning.

- Code splitting is a technique that seeks to minimize startup time. When we ship less JavaScript at startup, we can get applications to be interactive faster by minimizing main thread work during this critical period.

- Reducing JavaScript payloads downloaded at startup will contribute to better First Input Delay (FID) times. The reasoning behind this is that, by freeing up the main thread, the application is able to respond to user inputs more quickly by reducing JavaScript parse, compile, and execution-related startup costs.

- Split the JavaScript bundle to only send the code needed for the initial route when the user loads an application. This minimizes the amount of script that needs to be parsed and compiled, which results in faster page load times.

- Most newer browsers support dynamic import syntax, which allows for module fetching on demand. Dynamically importing JavaScript on certain user interactions will make sure that code not used for the initial page load is only fetched when needed.

Static import

```
import moduleA from "library";

form.addEventListener("submit", e => {
  e.preventDefault();
  someFunction();
});

const someFunction = () => {
  // uses moduleA
}
```

Dynamic import

```
form.addEventListener("submit", e => {
  e.preventDefault();
  import('library.moduleA')
    .then(module => module.default) // using the default export
    .then(() => someFunction())
    .catch(handleError());
});

const someFunction = () => {
    // uses moduleA
}
```

2- Reduce JavaScript execution time

Limiting the amount of JavaScript on your page reduces the amount of time that the browser needs to spend executing JavaScript code. This speeds up how fast the browser can begin to respond to any user interactions.

To reduce the amount of JavaScript executed on your page:

- Defer unused JavaScript
- Minimize unused polyfills
- Preloading important JavaScript files
- Use workers
- Minify and compress your code.
- Remove unused libraries

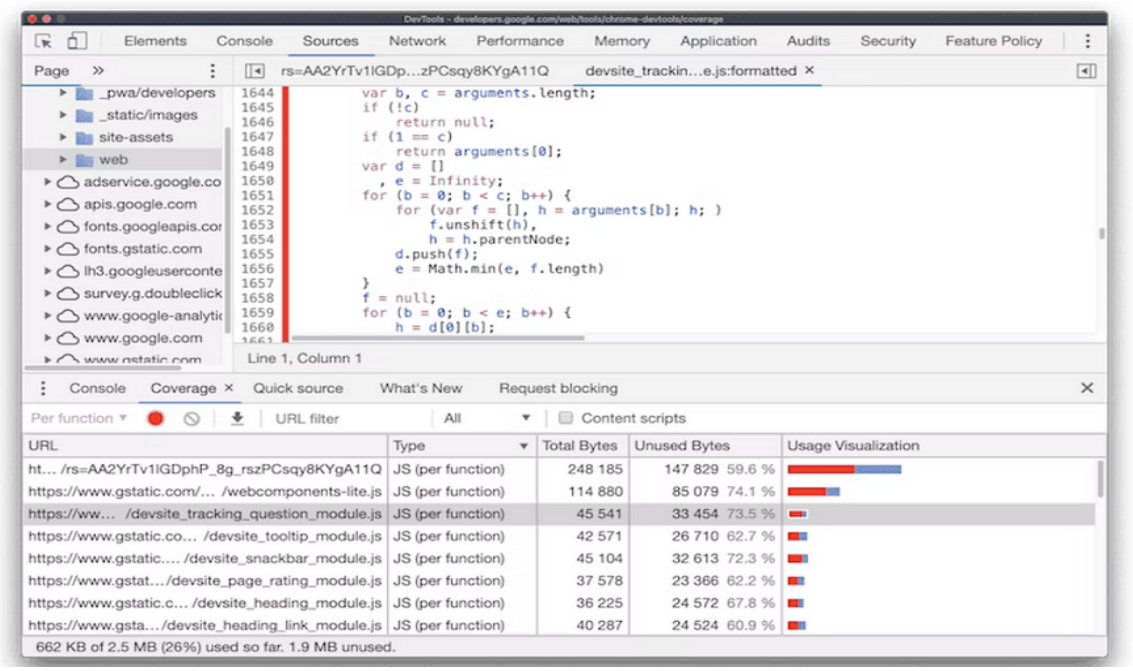
- **Defer unused JavaScript**

By default all JavaScript is render-blocking. When the browser encounters a script tag that links to an external JavaScript file, it must pause what it's doing and download, parse, compile, and execute that JavaScript. Therefore you should only load the code that's needed for the page or responding to user input.

The Coverage tab in Chrome DevTools can tell you how much JavaScript is not being used on your web page.

The table in the Coverage tab shows you what resources were analyzed, and how much code is used within each resource. Click a row to open that resource in the Sources panel and see a line-by-line breakdown of used code and unused code. Any unused lines of code will have a red line at the beginning.

Knowing unused JavaScript using Chrome DevTools :



To cut down on unused JavaScript:

- Code-split your bundle into multiple chunks

  Code-splitting is the concept of splitting a single large JavaScript bundle into smaller chunks that can be conditionally loaded (also known as lazy-loading).

- Defer any non-critical JavaScript, including third-party scripts, using async or defer.

  Unless there is a specific reason not to, all third-party scripts should be loaded with either defer or async by default.

- **Use a web worker**

A blocked main thread is one of the main causes of input delay. Web workers make it possible to run JavaScript on a background thread. Moving non-UI operations to a separate worker thread can cut down main thread blocking time and consequently improve FID.

Web Workers are a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface. Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa).

- **Preloading important JavaScript files**

Preloading key JavaScript files allows scripts to be downloaded beforehand, but executed later. This prevents the scripts from blocking the main-thread at critical phases of the page load, enabling your users to interact faster with the page.

preload our CSS and JavaScript files so they will be available as soon as they are required for the rendering of the page later on.

- **Minify and compress your code**

Minification is the process of removing whitespace and any code that is not necessary to create a smaller but perfectly valid code file. webpack v4 includes a plugin for this library by default to create minified build files.Compression is the process of modifying data using a compression algorithm. Gzip is the most widely used compression format for server and client interactions. Brotli is a newer compression algorithm which can provide even better compression results than Gzip.

Cumulative Layout Shift (CLS)

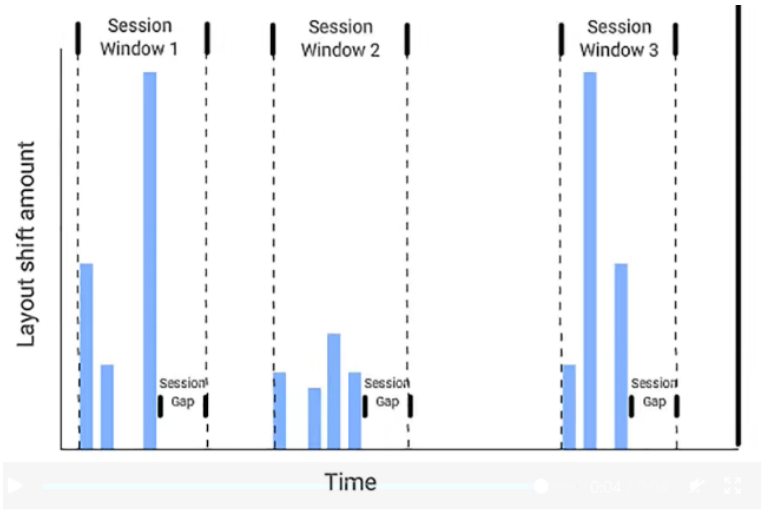
Cumulative Layout Shift (CLS) is an important, user-centric metric for measuring visual stability because it helps quantify how often users experience unexpected layout shifts—a low CLS helps ensure that the page is delightful.

CLS is a measure of the largest burst of *layout shift scores* for every unexpected layout shift that occurs during the entire lifespan of a page.

A *layout shift* occurs any time a visible element changes its position from one rendered frame to the next. (See below for details on how individual layout shift scores are calculated.)

A burst of layout shifts, known as a *session window*, is when one or more individual layout shifts occur in rapid succession with less than 1-second in between each shift and a maximum of 5 seconds for the total window duration.

The largest burst is the session window with the maximum cumulative score of all layout shifts within that window.



Example of session windows. Blue bars represent the scores of each individual layout shift.

- **What is a good CLS score?**

To provide a good user experience, sites should strive to have a CLS score of 0.1 or less.

- **What is layout shift?**

Any time an element that is visible within the viewport changes its start position between two frames. Such elements are considered *unstable elements*.

layout shifts only occur when existing elements change their start position.

If a new element is added to the DOM or an existing element changes size, it doesn't count as a layout shift—as long as the change doesn't cause other visible elements to change their start position.

- **Layout shift score**

To calculate the *layout shift score*, the browser looks at the viewport size and the movement of *unstable elements* in the viewport between two rendered frames :

layout shift score = impact fraction * distance fraction

Impact fraction

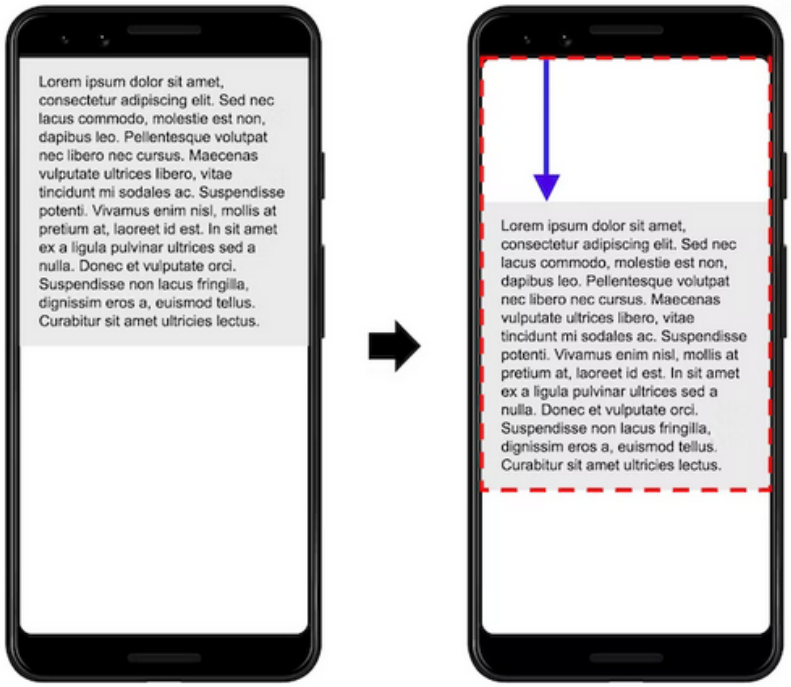
It measures how *unstable elements* impact the viewport area between two frames.

The *impact fraction* for the current frame is the union of the visible areas of all *unstable elements* for the previous frame *and* the current frame—as a fraction of the total area of the viewport—.

Distance fraction

It measures the distance that unstable elements have moved, relative to the viewport. The *distance fraction* is the greatest distance any *unstable element* has moved in the frame (either horizontally or vertically) divided by the viewport's largest dimension (width or height, whichever is greater).

Explained example :



In the image above there's an element that takes up half of the viewport in one frame. Then, in the next frame, the element shifts down by 25% of the viewport height. The red, dotted rectangle indicates the union of the element's visible area in both frames, which, in this case, is 75% of the total viewport, so its *impact fraction* is 0.75.

And the largest viewport dimension is the height, and the unstable element has moved by 25% of the viewport height, which makes the *distance fraction* 0.25.

So, in this example the *impact fraction* is 0.75 and the *distance fraction* is 0.25, so the *layout shift score* is 0.75 * 0.25 = 0.1875.

- **Expected vs Unexpected Layout Shift**

    - An expected layout shift happens in response to a user input.
        - For example, clicking on the search icon to expand an input field.
    - An unexpected layout shift, on the other hand, is usually triggered by third-party content, dimensionless images, or other dynamic content.
        - For example, an ad suddenly appearing and pushing an image or content down the page.

- **Optimize CLS**

1. Specifying image and video elements dimensions

- In the early days of the web, developers would add width and height attributes to their image and video tags to ensure sufficient space was allocated on the page before the browser started fetching images.

```
<img src="puppy.jpg" width="640" height="360" alt="Puppy with balloons" />
```

These "pixel" dimensions would ensure a 640x360 area would be reserved. The image would stretch to fit this space, regardless of whether the true dimensions matched or not.

- When Responsive Web Design was introduced, developers began to omit width and height and started using CSS to resize images instead:

```
img {
  width: 100%; /* or max-width: 100%; */
  height: auto;
}
```

A downside to this approach is space could only be allocated for an image once it began to download and the browser could determine its dimensions.

- This is where aspect ratio comes in:
    - The aspect ratio of an image is the ratio of its width to its height.
    - For an x:y aspect ratio, the image is x units wide and y units high.
    - With aspect ratio, if we know one of the dimensions, the other can be determined.
    - Knowing the aspect ratio allows the browser to calculate and reserve sufficient space for the height and associated area.

- Modern browsers now set the default aspect ratio of images based on an image's width and height attributes so it's valuable to set them to prevent layout shifts.

```
<!-- set a 640:360 i.e a 16:9 - aspect ratio -->
<img src="puppy.jpg" width="640" height="360" alt="Puppy with balloons" />
```

And the User-Agent stylesheets of all browsers add a default aspect ratio based on the element's existing width and height attributes:

```
img {
  aspect-ratio: attr(width) / attr(height);
}
```

- When working with responsive images, srcset defines the images you allow the browser to select between and what size each image is. To ensure <img> width and height attributes can be set, each image should use the same aspect ratio.

```
<img
  width="1000"
  height="1000"
  src="puppy-1000.jpg"
  srcset="puppy-1000.jpg 1000w, puppy-2000.jpg 2000w, puppy-3000.jpg 3000w"
  alt="Puppy with balloons"
/>
```

2. Statically reserve space for the ad slot

In other words, style the element before the ad tag library loads.

Ads are usually requested asynchronously and dynamically add content to your page during or after page load. While ads are being fetched, the rest of the page continues to load and non-ad content may become visible to the user. If you don't reserve sufficient space for the ads being loaded, they can end up displacing visible non-ad content when they are eventually added to the page.

3. Avoid placing ads near the top of the viewport

Ads near the top of the viewport may cause a greater layout shift than those at the middle. This is because ads at the top generally have more content lower down, meaning more elements move when the ad causes a shift.

4. Web fonts causing FOUT/FOIT

Downloading and rendering web fonts can cause layout shifts in two ways:

- The fallback font is swapped with a new font (FOUT - flash of unstyled text)
- "Invisible" text is displayed until a new font is rendered (FOIT - flash of invisible text)

The following tools can help you minimize this:

- font-display allows you to modify the rendering behavior of custom fonts with values such as auto, swap, block, fallback and optional. Unfortunately, all of these values (except optional) can cause a re-layout in one of the above ways.
- The Font Loading API can reduce the time it takes to get necessary fonts.
- Using `<link rel=preload>` on the key web fonts: a preloaded font will have a higher chance to meet the first paint and it ensures text on your page remains visible and stable during the web font load, in which case there's no layout shifting.

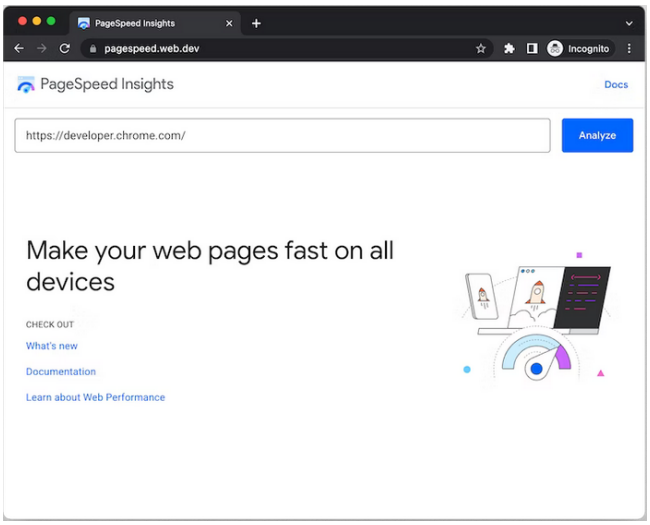
5. Use CSS transform property

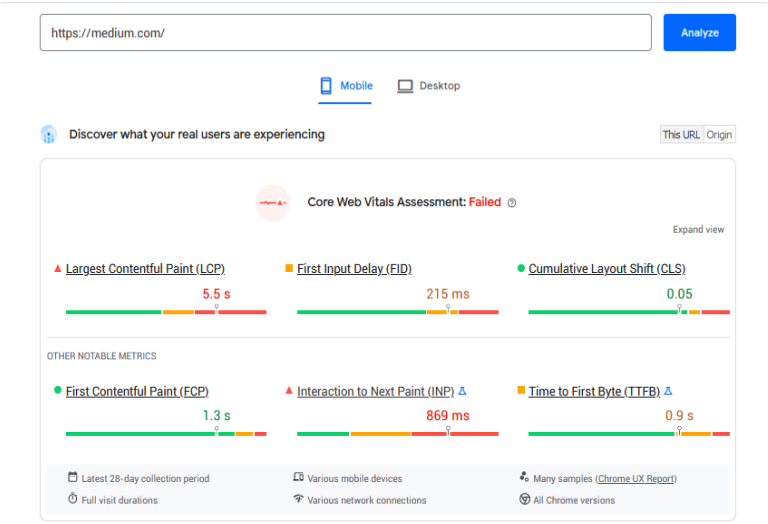
CSS transform property allows you to animate elements without triggering layout shifts:

- Instead of changing the height and width properties, use transform: scale().
- To move elements around, avoid changing the top, right, bottom, or left properties and use transform: translate() instead.

Measure Core Web Vitals using Lighthouse tool

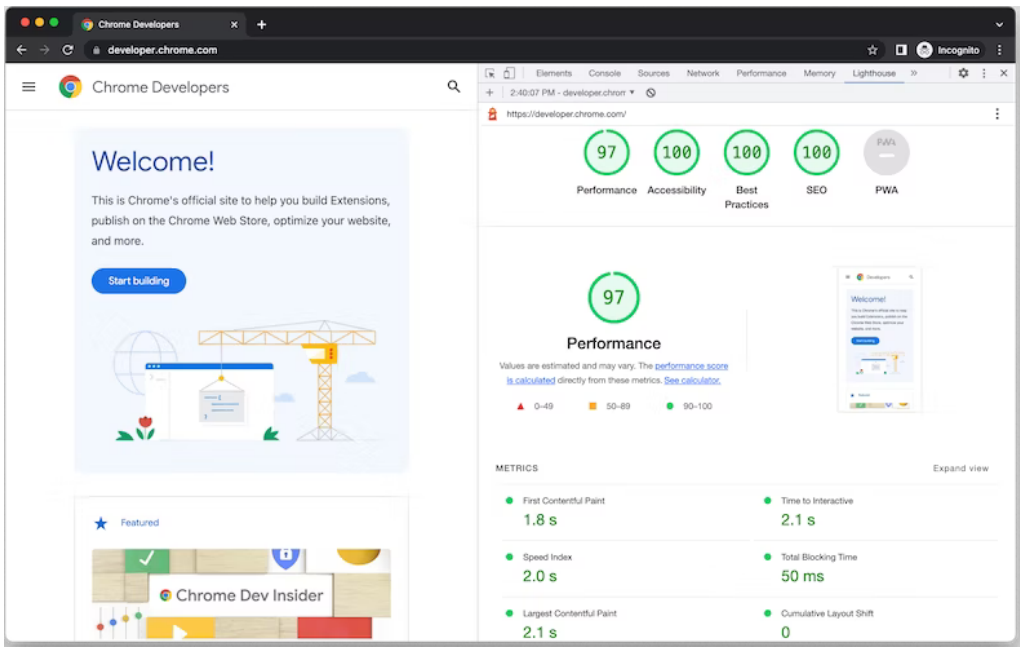
- Lighthouse from PageSpeed Insights





Visit: https://developer.chrome.com/docs/lighthouse/overview/#devtools

- Lighthouse from Chrome DevTools



Visit: https://developer.chrome.com/docs/lighthouse/overview/#psi

References

https://web.dev/lcp/

https://web.dev/fid/

https://web.dev/cls/

https://web.dev/optimize-fid/

https://web.dev/optimize-lcp/

https://web.dev/optimize-cls/

https://developers.google.com/publisher-tag/guides/minimize-layout-shift

https://css-tricks.com/the-difference-between-minification-and-gzipping/

https://css-tricks.com/improve-largest-contentful-paint-lcp-on-your-website-with-ease/

https://imagekit.io/blog/what-is-image-cdn-guide/?utm_source=css-tricks&utm_medium=sponsored_content&utm_campaign=csstricks_LCP

https://web.dev/long-tasks-devtools/

https://web.dev/reduce-javascript-payloads-with-code-splitting/

https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Loading

https://imagekit.io/blog/what-is-image-cdn-guide/?utm_source=css-tricks&utm_medium=sponsored_content&utm_campaign=csstricks_LCP