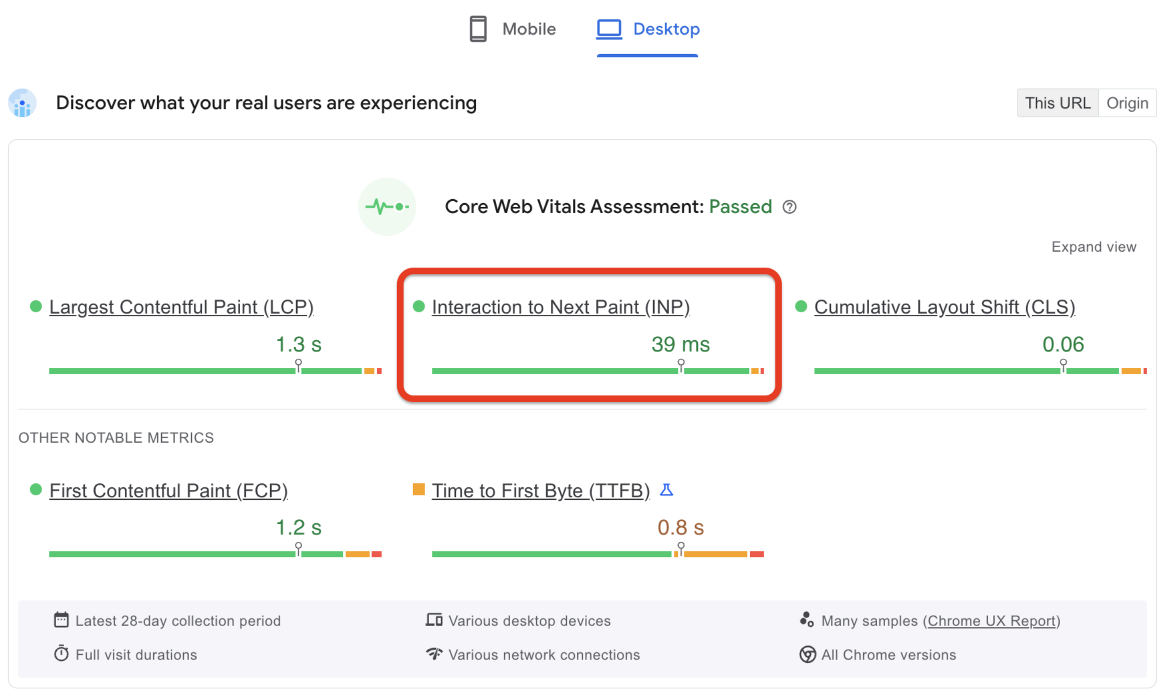
Interaction to Next Paint (INP)

TL;DR: Interaction to Next Paint (INP) is the time between when a user clicks on a page and sees a response. To make this faster, optimize how JavaScript works, minimize unnecessary tasks, and prioritize important actions. Use features like requestIdleCallback to make sure user interactions get quick and smooth responses for an enjoyable browsing experience.

Interaction to Next Paint (INP) is now **an official Core Web Vital**, replacing First Input Delay (FID) as the new responsiveness metric.



Discover what your real users are experiencing

Core Web Vitals Assessment: Passed

Largest Contentful Paint (LCP) 1.3 s

Interaction to Next Paint (INP) 39 ms

Cumulative Layout Shift (CLS) 0.06

OTHER NOTABLE METRICS

First Contentful Paint (FCP) 1.2 s

Time to First Byte (TTFB) 0.8 s

Latest 28-day collection period
Various desktop devices
Many samples (Chrome UX Report)
Full visit durations
Various network connections
All Chrome versions

What is Interaction to Next Paint (INP)?

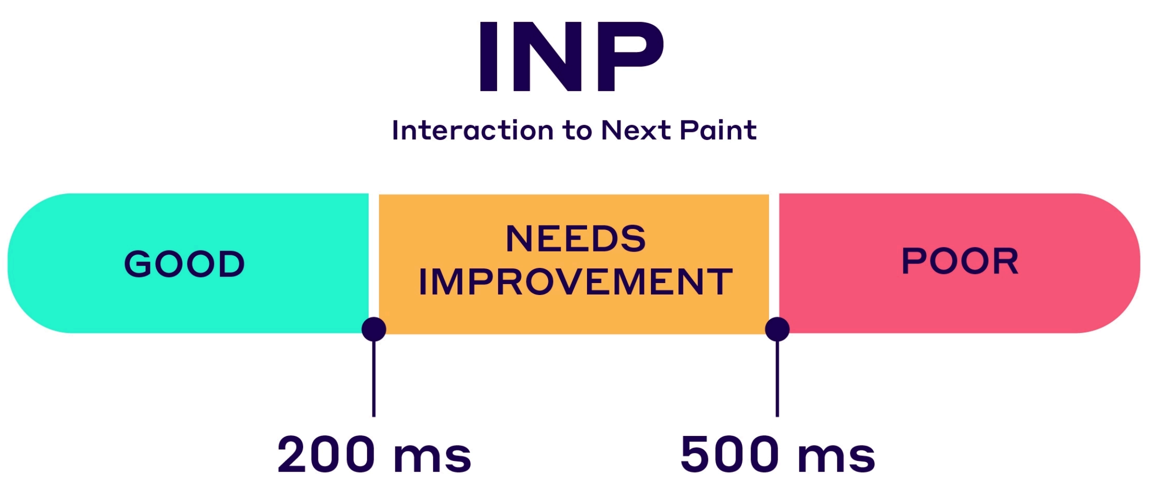
INP assesses a page's overall responsiveness to user interactions by observing the latency of all qualifying interactions during a user's visit to a page. **The final INP value is the longest interaction observed.**

The interactions that play a part in INP's calculations are:

- Clicking with a mouse;
- Tapping on a device with a touchscreen;
- Pressing a key on a physical or digital keyboard.

Similar to the other Core Web Vitals, your INP score can be in one of the three thresholds:

- Good: 0-200ms
- Needs Improvements: 200-500ms
- Poor: >500ms



To guarantee that you achieve this objective for the majority of your users, it is recommended to assess the 75th percentile of page loads, segmented across mobile and desktop devices.

If you want to learn more or brush up your knowledge about INP, read our article on the new responsiveness metric.

Understanding Interaction Latency

If you want your INP score to go from poor to good, you need to understand interaction latency.

So what exactly is interaction latency?

Interaction latency refers to the delay or lag experienced between a user's input or action and the resulting response or output on the screen. It is a crucial factor in determining your site's responsiveness and perceived performance.

Three primary components contribute to interaction latency:

Input Delay

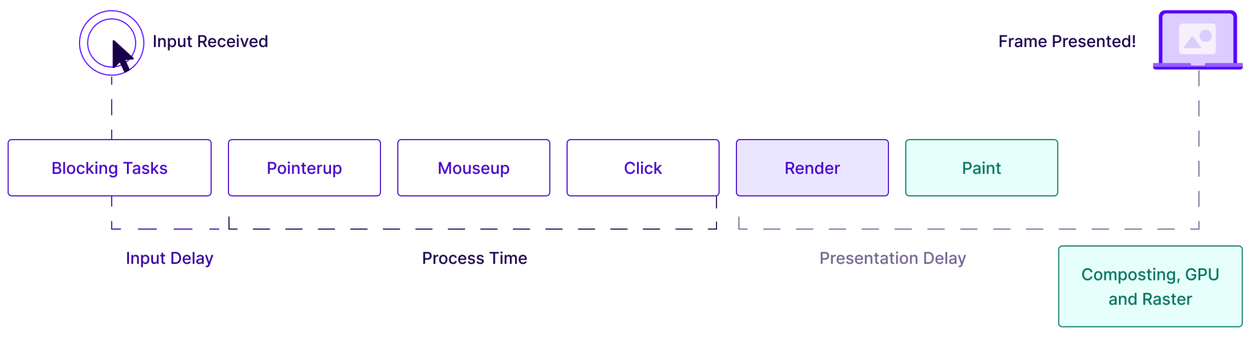
Input delay refers to the time between when a user starts interacting with the page and when the associated actions or event callbacks begin to execute. It includes the physical or technical delays caused by the input device (e.g., keyboard, mouse, touchscreen) and the system's input processing mechanisms.

Processing Time

Once the user input is received, the system must process it to determine the appropriate response or action. Processing time refers to the duration required for the system to analyze and interpret the input data, perform any necessary calculations or operations, and generate the output or response.

Presentation Delay

After the system has generated the output or response, there is typically a delay before it is presented to the user. Presentation delay encompasses the time it takes for the system to update the display, render graphics or user interfaces, and deliver the output to the user interface or output device.



Understanding and optimizing the interaction latency can provide a seamless user experience and fix your INP scores.

But before that, let's take a look at the main culprits for high interaction latency and bad INP scores…

Core Web Vitals INP issue detected on your website: What might be causing it?

Although INP is labeled as *pending,* this does not mean you should enter the optimization process without a strategy.

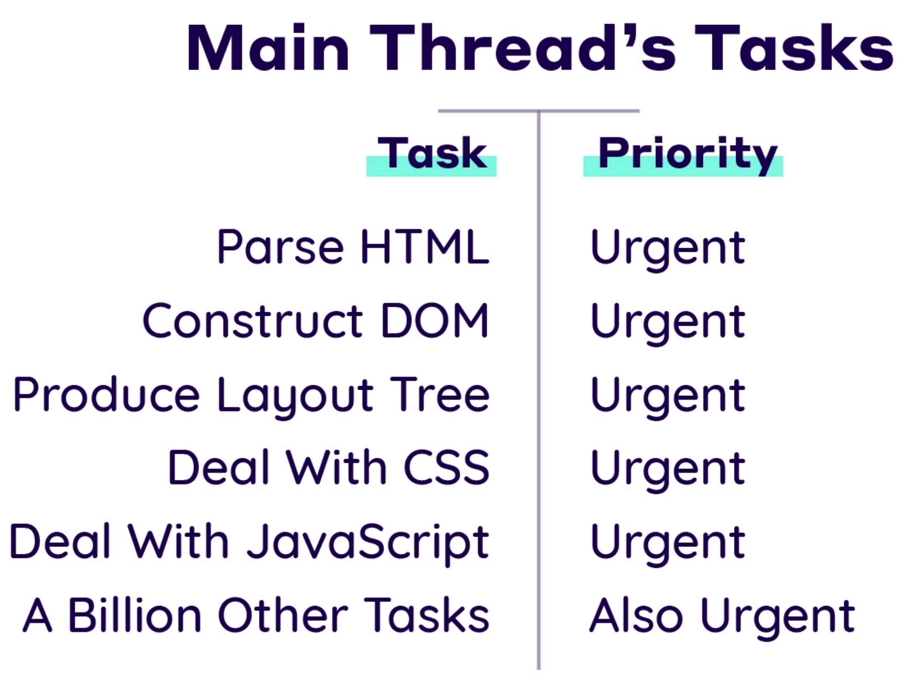
The first thing you need to do is learn what the main INP culprits are, so you can handle them effectively.

Here are the main reasons for the *"INP issue: longer than 200ms"* error message:

Long tasks

Everything that a browser does is considered a task. This includes rendering, parsing HTML, running JavaScript, and anything you may or may not have control over.

The main thread is where the browser does most of the work needed to display a page. And while there might be dozens of tasks that need to be executed, **the main thread can only process one task at a time.**



Main Thread's Tasks

Task	Priority
Parse HTML	Urgent
Construct DOM	Urgent
Produce Layout Tree	Urgent
Deal With CSS	Urgent
Deal With JavaScript	Urgent
A Billion Other Tasks	Also Urgent

But that's only half of the problem.

The other half is that **if a task takes more than 50 milliseconds to be executed, it's classified as a *long task.***

Considering that the main thread can handle one task at a time, the longer the task is, the longer the browser will be blocked processing it.

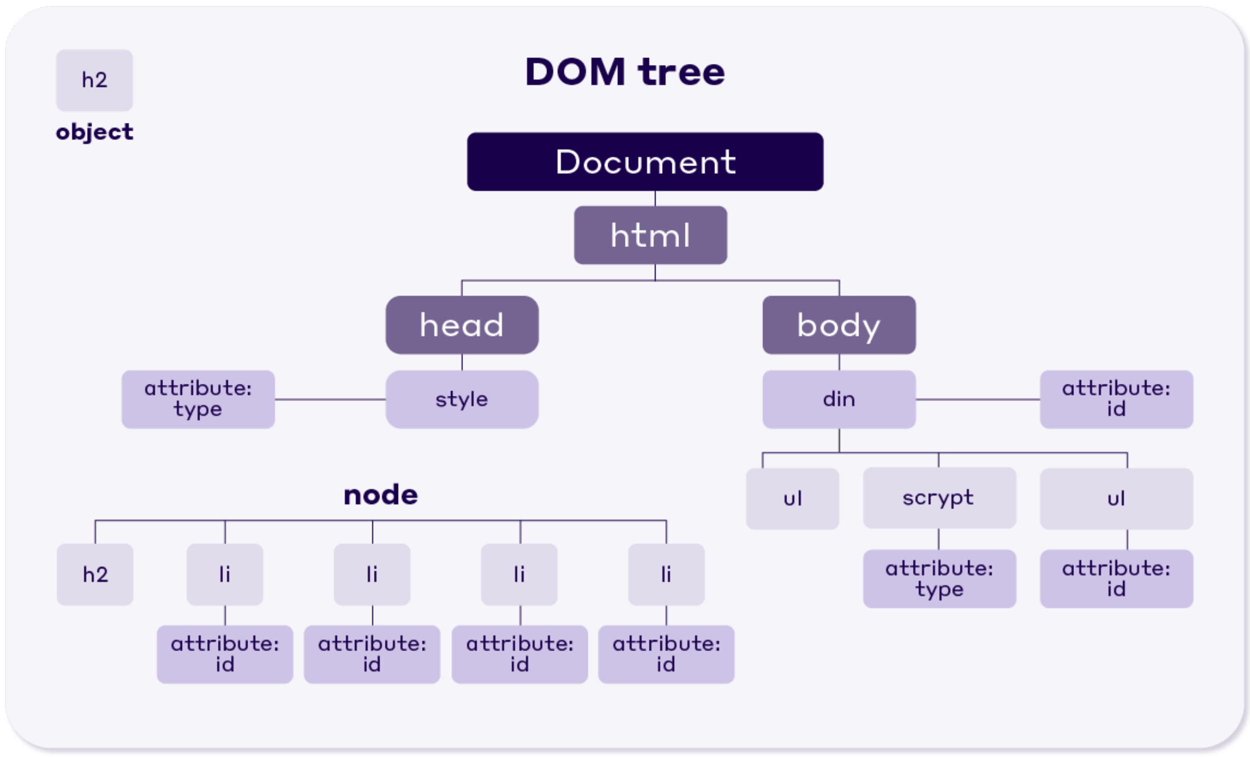
In other words, if the user is attempting to interact with the page while a long task runs, the browser will be delayed in fulfilling the request.

The result is - interaction latency and a lower INP score.

Large DOM size

Another reason for failing INP is having a large DOM size.

The Document Object Model (DOM) is an inseparable part of every web page. The DOM is a representation of an HTML document structured as a tree. Each branch in the tree terminates at a node, and each node contains objects. Nodes can represent different parts of the document, such as elements, text strings, or comments.



The DOM itself isn't a problem, but its size might be. A large DOM size impacts a browser's ability to render a page quickly and efficiently.

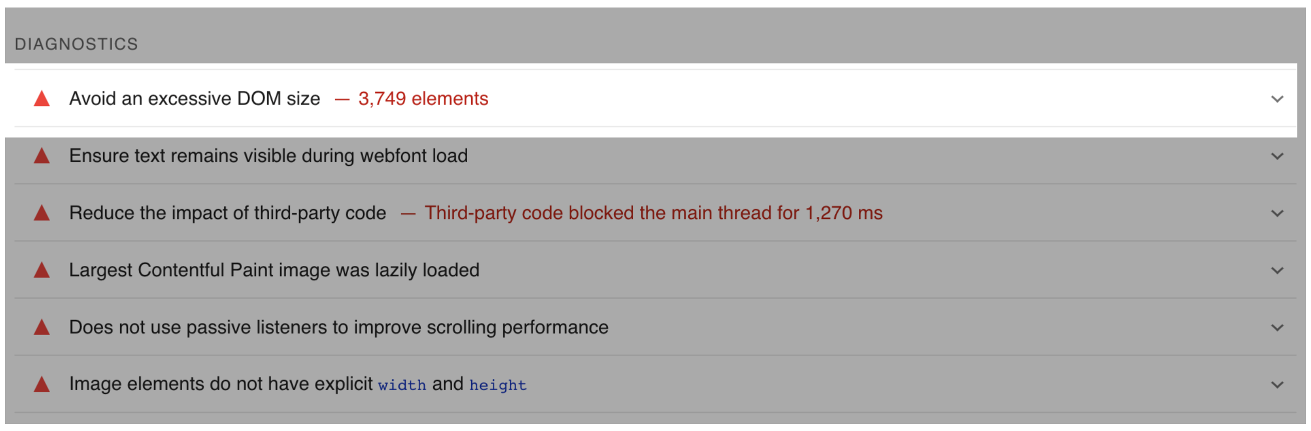
The larger the DOM, the more resource-intensive it is to initially display the page and make subsequent updates during the page's lifecycle.

Simply put:

If you want a page to respond quickly to user interactions, ensure your DOM includes only the necessary elements.

You might be wondering what "necessary" means. According to Lighthouse, a page's DOM size is excessive when it **exceeds 1,400 nodes**.

So make sure to stay within this limit. Otherwise, you might see the following error in your PageSpeed Insights report:



DIAGNOSTICS

- ▲ Avoid an excessive DOM size — 3,749 elements
- ▲ Ensure text remains visible during webfont load
- ▲ Reduce the impact of third-party code — Third-party code blocked the main thread for 1,270 ms
- ▲ Largest Contentful Paint image was lazily loaded
- ▲ Does not use passive listeners to improve scrolling performance
- ▲ Image elements do not have explicit `width` and `height`

Client-side rendering of HTML

To understand why client-site rendering might cause poor INP scores, we need to explain how it differs from the server-side rendering of HTML.

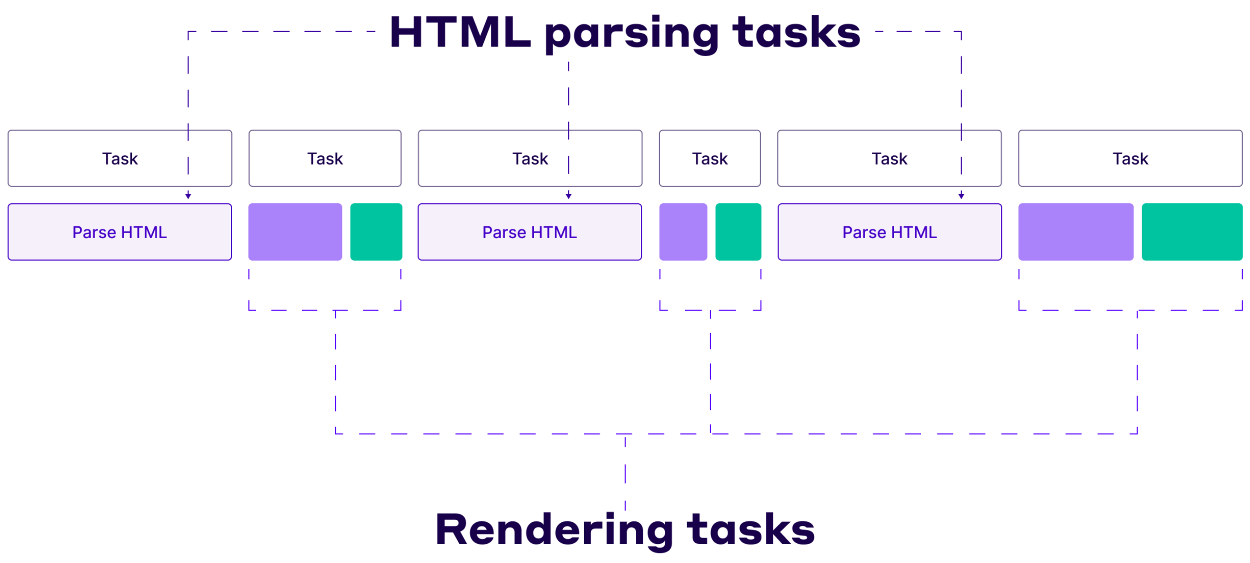
The traditional page load involves the browser receiving HTML from the server on every navigation. What happens in the background when a person decides to load a page is:

1. The browser sends a navigation request for the URL provided.
2. The server responds with HTML in chunks.

The key here is *"in chunks."*

When the browser receives the first chunk of HTML, it can start parsing it. But as we know, parsing HTML is a task the main thread handles.

However, after each chunk is processed, the browser takes a break from parsing and allows other tasks to be performed. This prevents long tasks that could slow down the browser.



Instead, it can start rendering the parts of the page that have already been parsed, so the user sees a partially loaded page sooner. It can also handle any user interactions that occur during the initial loading of the page.

In other words:

This approach translates to a better Interaction to Next Paint (INP) score for the page.

On the contrary, if your website uses the single page application (SPA) pattern, which dynamically creates large parts of the HTML/DOM on the client with JavaScript, you can expect negative effects on your INP score.

In client-side rendering, the server sends a small chunk of basic HTML to the client. Then the client takes care of filling in the main content of the page using data it fetches from the server.

All future navigations are handled by JavaScript, fetching new HTML from the server and dynamically updating the page without fully reloading it. Unfortunately, when it comes to JavaScript tasks on the client side, they are not automatically chunked up.

This can lead to long tasks that block the main thread, potentially affecting your page's Interaction to Next Paint score. Therefore, client-side rendering can hurt the loading and interactivity of your page.

If you need additional info on the pros and cons of server-side vs. client-side rendering, check out this video:

Now that you know some of the main culprits, let's proceed with measuring your INP score and identifying slow interactions.

How to identify slow interactions using field data and debug them in the lab

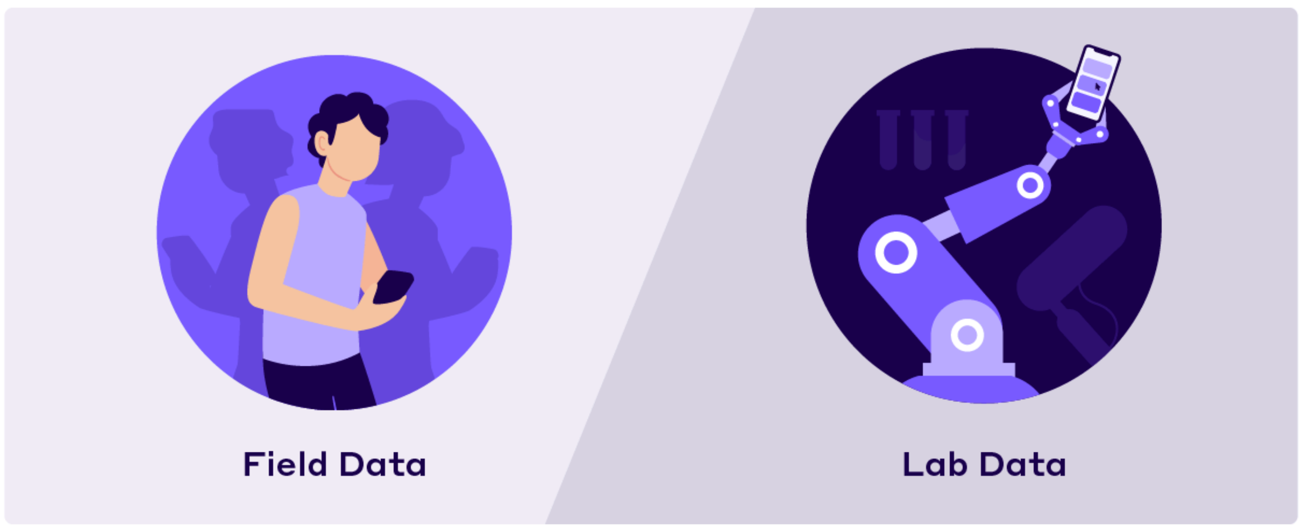
The next step in the INP optimization journey is to measure your site's performance and identify any slow interactions.

Similar to First Input Delay, INP is best measured in the field – examining how real users experience your website.

The optimal testing process would look like this:

1. Gather field data for INP
2. Identify the exact actions responsible for INP
3. Use lab tools to work out why those interactions are slow

We said *optimal* because, in some cases, your site might not have field data (also known as Real User Monitoring (RUM) data). However, this doesn't mean you should give up optimizing your INP score. You need to take a different approach and leverage the available lab tools.



Let's take a look at both scenarios and explain how to take most of your field and lab data.

Field Data

Ideally, you'll want to have field data when you start improving your site's responsiveness. Relying on RUM data saves you a lot of time figuring out which interactions need to be optimized.

Furthermore, lab-based tools can simulate certain interactions but cannot fully replicate real-world user experiences.

When gathering INP field data, you'll want to capture the following to give context to why interactions were slow:

- **The INP value –** The distribution of these values will determine whether the page meets the INP thresholds.
- **The element selector string responsible for the page's INP –** Solely knowing a page's INP value isn't enough. You want to know which elements are responsible for the interactions.
- **The loading state of the page for the interaction that is the page's INP –** Understanding whether an interaction occurs during page load or afterward helps determine whether you should optimize the main thread or if the interaction itself is slow, regardless of the page's initial loading.
- **The interaction's startTime –** Ensure to log the interaction's startTime as it lets you know when the interaction occurred on the performance timeline.
- **The event type –** Knowing the interaction event type – click keypress, other eligible events – allows you to pinpoint which event callback in the interaction took the longest to run.

If you're asking yourself:

How am I supposed to capture all of these things?

Don't worry. All of the data is exposed in the web-vitals JavaScript library. You can check Google's step-by-step guide on how to leverage the web-vital library and even how to transmit INP data straight to your Google Analytics.

Also, even if you're already collecting data with a third-party RUM provider, consider comparing it with Chrome UX Report (CrUX) data, as there are differences in the methodologies they use.

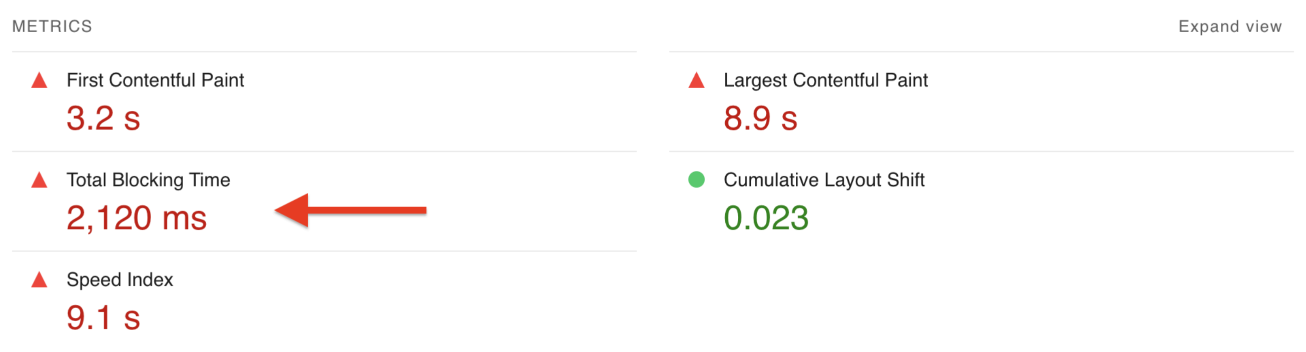
Lab Data

Field data is the most reliable source for measurement. However, as we said, it is not always available.

But there's no need to worry because you can still measure and identify interactions to improve with the help of lab data.

You can use either Lighthouse or PageSpeed Insights to run some performance tests. The metric you should keep an eye out for is Total Blocking Time (TBT).

TBT is a metric that assesses page responsiveness during load and correlates very well with INP. A poor TBT score is a signal there are interactions that might be slow during page load.


METRICS
Expand view

▲ First Contentful Paint 3.2 s	▲ Largest Contentful Paint 8.9 s
▲ Total Blocking Time 2,120 ms	● Cumulative Layout Shift 0.023
▲ Speed Index 9.1 s	

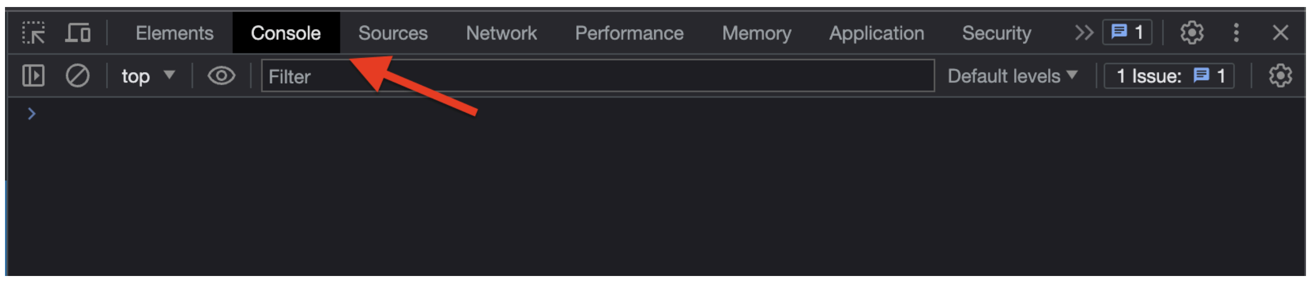
Here's how you can reproduce slow interaction with lab tools:

- **Use the Web Vitals Chrome Extension**

The Web Vitals Chrome Extension is one of the easiest ways to measure your site's interaction latency. Here's what you need to do to retrieve useful information:

1. In Chrome, click the extensions icon to the right of the address bar.
2. Locate the Web Vitals extension in the drop-down menu.
3. Click the icon at the right to open the extension's settings.
4. Click Options.
5. Enable the Console logging checkbox in the resulting screen, then click Save.

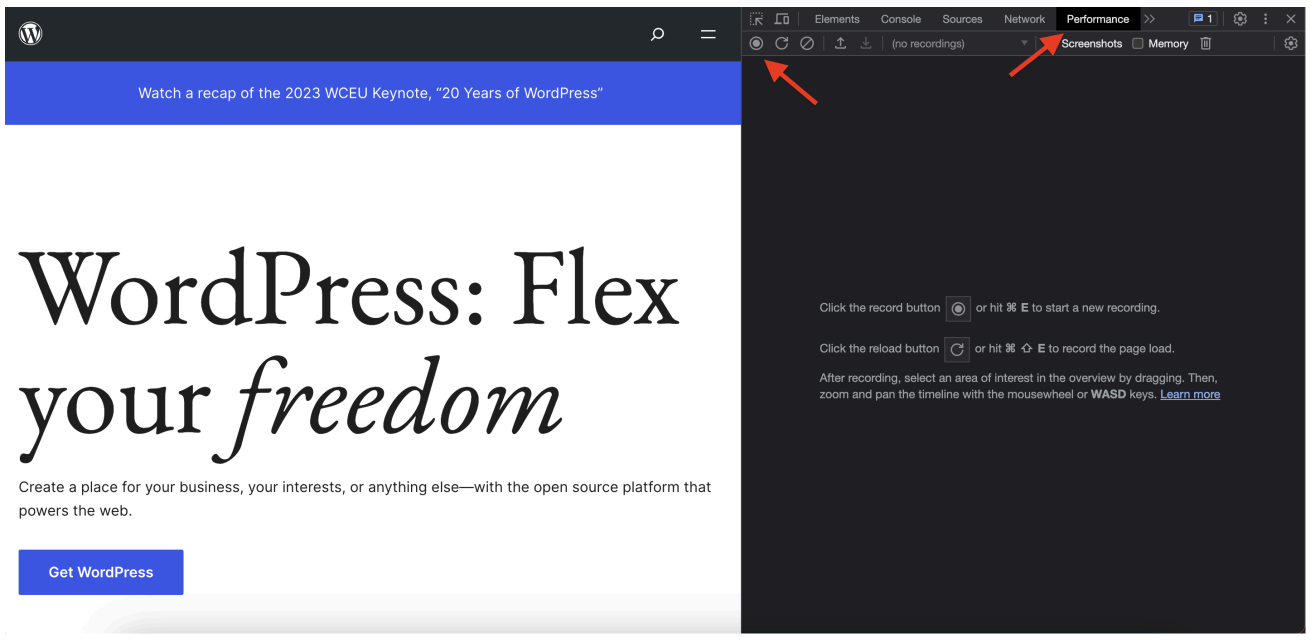
Finally, open the Chrome DevTools console and begin testing. You'll receive helpful console logs giving you detailed diagnostic information for the interaction.



- **Record a trace with Chrome DevTools**

To get even more information about why interaction is slow, you can use the performance profiler in Chrome DevTools. Just do the following:

1. Open Chrome DevTools and go to the Performance panel.
2. Click the Record button at the upper left of the panel to start tracing.



Watch a recap of the 2023 WCEU Keynote, "20 Years of WordPress"

WordPress: Flex your *freedom*

Create a place for your business, your interests, or anything else—with the open source platform that powers the web.

Click the record button or hit ⌘ E to start a new recording.

Click the reload button or hit ⌘ ⇧ E to record the page load.

After recording, select an area of interest in the overview by dragging. Then, zoom and pan the timeline with the mousewheel or WASD keys. Learn more

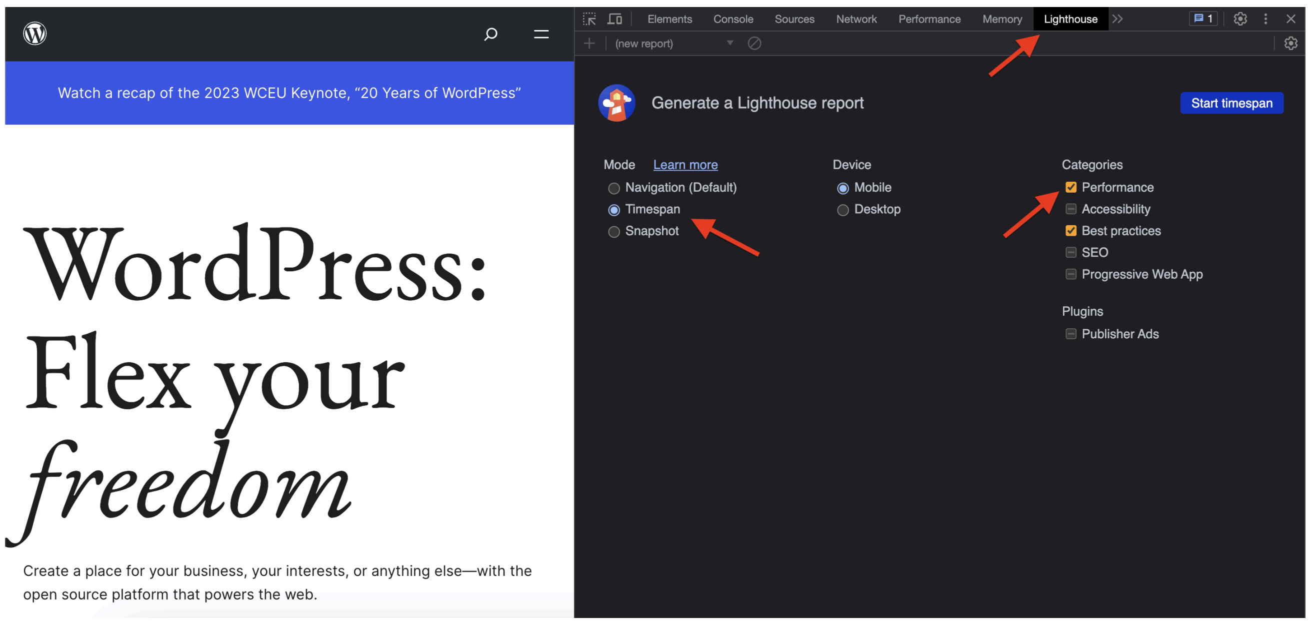
3. Perform the desired interaction.
4. Click the Record button again to stop tracing.

To quickly identify performance issues, check the activity summary at the top of the profiler when the profile populates. Look for red bars in the activity summary, indicating instances of long tasks during the recording. These red bars help you pinpoint problem areas and focus your investigation.

- **Use Lighthouse timespans**

Lighthouse's timespans mode is the less intimidating alternative to the DevTools performance profiler. Here's how to use it:

1. Go to the Lighthouse tab in DevTools.
2. Under the section labeled Mode, select the Timespan option.
3. Select the desired device type under the section labeled Device.
4. Ensure at least the checkbox labeled Performance is selected under the Categories label.
5. Click the Start timespan button.



Watch a recap of the 2023 WCEU Keynote, "20 Years of WordPress"

WordPress: Flex your *freedom*

Create a place for your business, your interests, or anything else—with the open source platform that powers the web.

Generate a Lighthouse report

Start timespan

Mode [Learn more](#)
- ○ Navigation (Default)
- ◉ Timespan
- ○ Snapshot

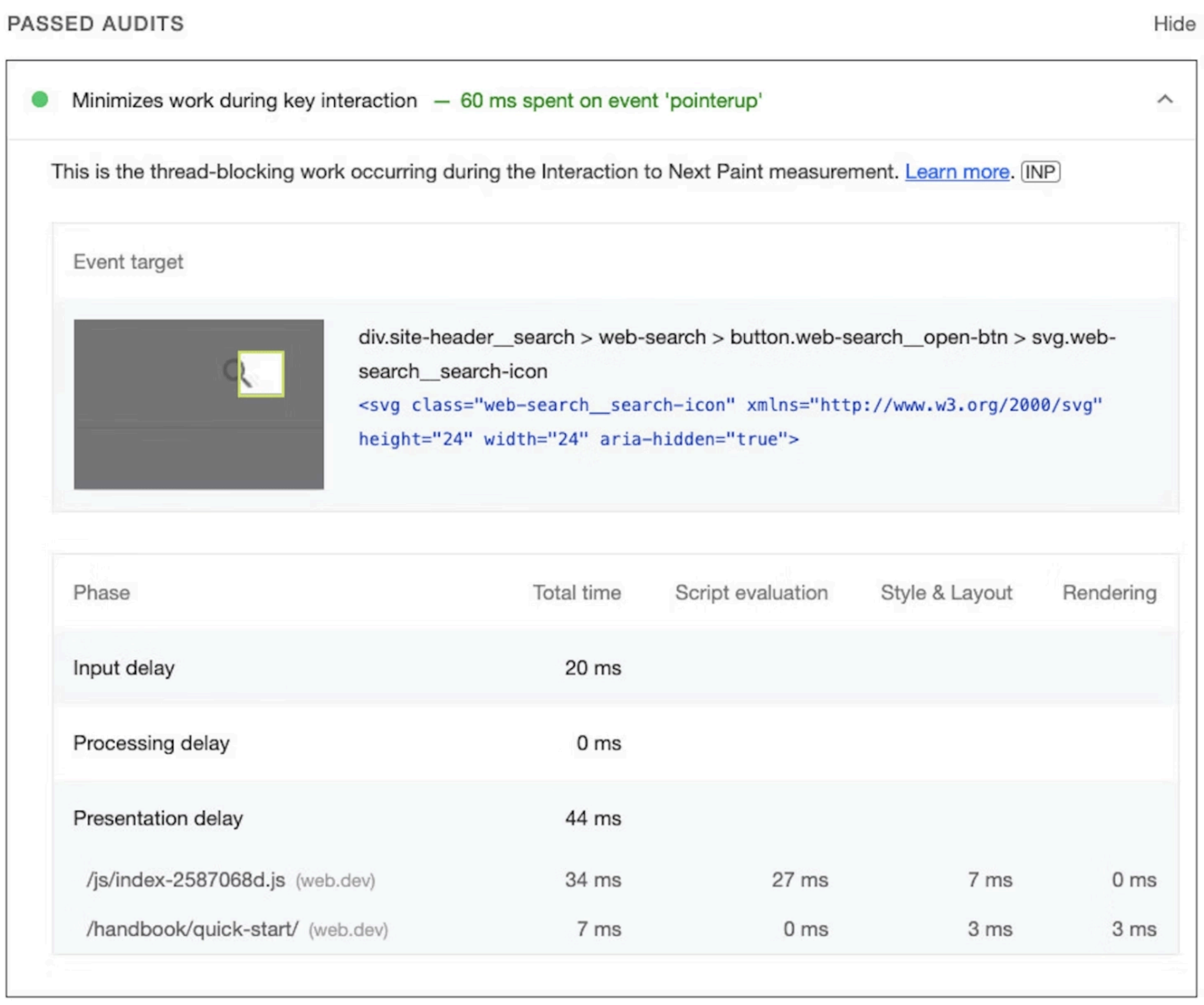
Device
- ◉ Mobile
- ○ Desktop

Categories
- [✓] Performance
- [ ] Accessibility
- [✓] Best practices
- [ ] SEO
- [ ] Progressive Web App

Plugins
- [ ] Publisher Ads

6. Test the desired interaction(s) on the page.
7. Click the End timespan button and wait for the audit to appear
8. Once the audit populates, filter it by INP.

You will be presented with a list of failed and passed audits. When you click on them, a drop-down menu will appear, and you can see a breakdown of time spent during interaction divided by input delay, processing time, and presentation delay.



PASSED AUDITS Hide

● Minimizes work during key interaction — 60 ms spent on event 'pointerup'

This is the thread-blocking work occurring during the Interaction to Next Paint measurement. [Learn more](#). INP

Event target

div.site-header__search > web-search > button.web-search__open-btn > svg.web-search__search-icon

```
<svg class="web-search__search-icon" xmlns="http://www.w3.org/2000/svg"
height="24" width="24" aria-hidden="true">
```

Phase	Total time	Script evaluation	Style & Layout	Rendering
Input delay	20 ms			
Processing delay	0 ms			
Presentation delay	44 ms			
/js/index-2587068d.js (web.dev)	34 ms	27 ms	7 ms	0 ms
/handbook/quick-start/ (web.dev)	7 ms	0 ms	3 ms	3 ms

Source: [Google](#)

Now that you know what to work on, it's time to roll up your sleeves and begin optimizing.

To guarantee your site a *good* INP score, you need to ensure that each interaction event runs as fast as possible. Here's how to achieve it:

Reduce input delay

1. Avoid recurring timers that overwork the main thread

[setTimeout](#) and [setInterval](#) are commonly used JavaScript timer functions that can contribute to input delay.

`setTimeout` schedules a callback to run after a specified time, and while it can help avoid long tasks, it depends on when the timeout occurs and if it coincides with user interactions.

setInterval, on the other hand, schedules a callback to run repeatedly at a specified interval. Because of that, it is more likely to interfere with user interactions. Its recurring nature increases input delay and can affect the responsiveness of interactions.

If you have control over the timers in your code, evaluate their necessity and reduce their workload as much as possible.

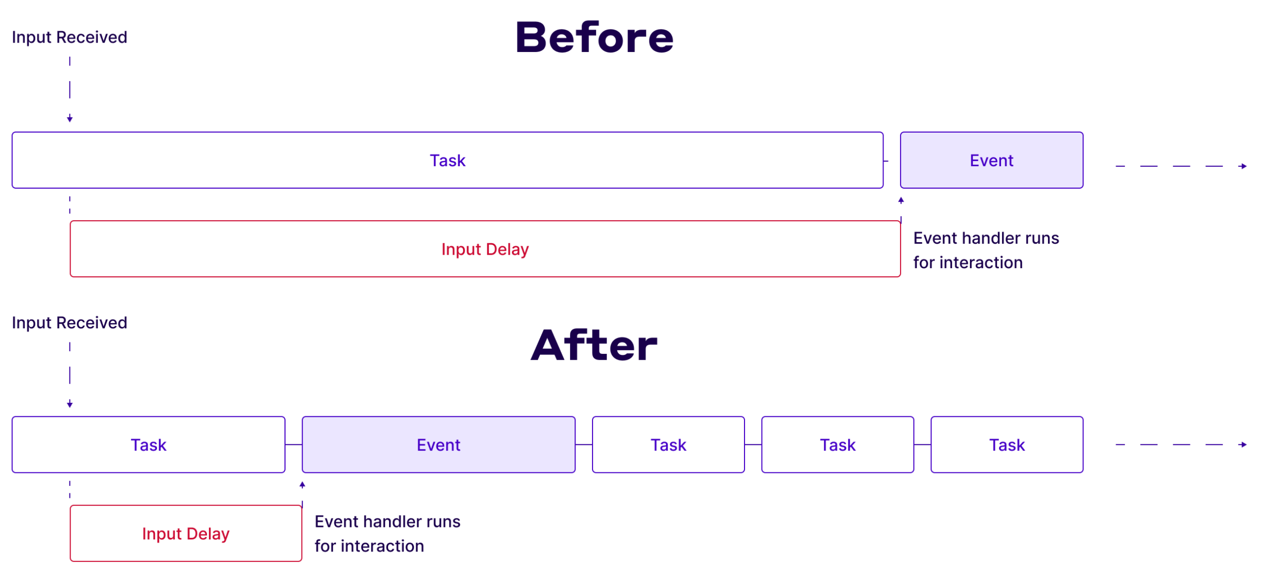
2. Avoid long tasks

As you already know, long tasks block the main thread, preventing the browser from executing the interaction events.

To enhance your site's responsiveness, it is important to minimize the workload on the main thread and consider breaking up long tasks.

By breaking up long tasks into smaller chunks, the main thread gets an opportunity to handle other tasks and respond to user interactions more quickly.

Additionally, breaking up long tasks helps avoid the "jank" effect, where animations and transitions on the page become choppy or stutter due to the overwhelmed main thread. By ensuring that each task completes within a shorter timeframe, the page can maintain a smoother visual experience for the user.



3. Avoid interaction overlap

Interaction overlap means that after a visitor interacted with one element, they make another interaction with the page before the initial interaction has had a chance to render the next frame.

For instance, this can happen when users are typing in form fields, leading to numerous keyboard interactions within a brief timeframe. You can optimize the process by:

- Debouncing inputs to limit the number of times an event callback executes in a given period of time.
- Using AbortController to cancel outgoing fetch requests so the main thread doesn't become overworked handling fetch callbacks.

Optimize event callbacks (processing time)

1. Consider removing the unnecessary callback

Is the expensive event callback truly necessary? If not, consider removing the code entirely, or if that's not possible, delay its execution until a more suitable time.

Key term: Event callback
Think of it like a chain reaction. When you perform an action on a website, like clicking a button, the website recognizes that action as an event. The website then looks for a specific piece of code, called a callback

function, that is connected to that event. Once the callback function is found, it is executed, and it determines what should happen next.

2. Defer non-rendering work

Long tasks can be broken up by yielding to the main thread. When you yield to the main thread, you essentially pause the current task and split the remaining work into a separate task. This allows the renderer to handle updates to the user interface that were processed earlier in the event callback. By yielding, you enable the renderer to execute pending changes and ensure a smooth and timely user interface update.

Key term: Yielding
Yielding the main thread refers to temporarily pausing the execution of a task running on the main thread to allow other tasks to be processed. When a task on the main thread yields, it means that it voluntarily gives up control and allows other pending tasks to be executed. This mechanism prevents long-running tasks from monopolizing the main thread and causing unresponsiveness in the user interface.

Minimize presentation delay

1. Reduce DOM size

A large DOM size is a surefire way to fail the INP assessment. So to ensure that won't happen, you need to reduce your DOM size, or to put it another way – you need to reduce DOM depth.

Aim for a DOM depth of no more than 1,400 nodes.

You can achieve it by incorporating the following techniques:

- Avoid poorly coded plugins and themes
- Minimize JavaScript-based DOM nodes
- Move away from page builders that generate bloated HTML
- Don't copy/paste text into the WYSIWYG editor
- Break down your one-page website into multiple pages
- Don't hide unwanted elements using display:none
- Avoid using complicated CSS declarations and JavaScript

2. Avoid excessive or unnecessary work in requestAnimationFrame callbacks

The `requestAnimationFrame` method tells the browser that you wish to perform an animation and requests that the browser calls a specified function to update an animation right before the next repaint.

The requestAnimationFrame() callback is part of the rendering phase in the event loop and needs to finish before the next frame can be displayed. If you're utilizing requestAnimationFrame() for tasks unrelated to user interface changes, it's essential to recognize that you might be causing a delay in the rendering of the subsequent frame.

So avoid using them when unnecessary.

3. Defer ResizeObserver callbacks

The ResizeObserver interface reports changes to the dimensions of an Element's content or border box or the bounding box of an SVGElement.

ResizeObserver callbacks run before rendering and can potentially postpone the presentation of the next frame if they involve resource-intensive tasks. Similar to event callbacks, it is advisable to defer any unnecessary logic not required for the immediate upcoming frame.