User-centric Performance Metrics

By Philip Walton
Engineer at Google working on the Web Platform

You've probably heard time and time again that performance matters, and that it's critical your web apps are fast.

But as you try to answer the question: *how fast is my app?*, you'll realize that fast is a vague term. What exactly do we mean when we say fast? In what context? And fast for whom?

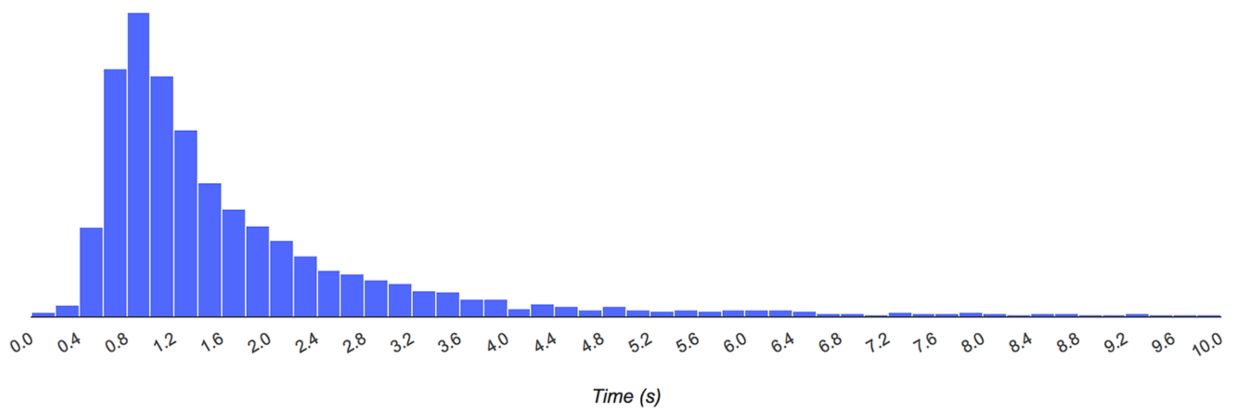
Note: If you'd rather watch a video than read an article, I spoke on this topic at Google I/O 2017 with my colleague Shubhie Panicker.

When talking about performance it's important to be precise so we don't create misconceptions or spread myths that can sometimes lead to well-intentioned developers optimizing for the wrong things—ultimately harming the user experience rather than improving it.

To offer a specific example, it's common today to hear people say something like: ***I tested my app, and it loads in X.XX seconds***.

The problem with this statement is *not* that it's false, it's that it misrepresents reality. Load times vary dramatically from user to user, depending on their device capabilities and network conditions. Presenting load times as a single number ignores the users who experienced much longer loads.

In reality, your app's load time is the collection of all load times from every individual user, and the only way to fully represent that is with a distribution like in the histogram below:



The numbers along the X-axis show load times, and the height of the bars on the y-axis show the relative number of users who experienced a load time in that particular time bucket. As this chart shows, while the largest segment of users experienced loads of less than one or two seconds, many of them still saw much longer load times.

The other reason "my site loads in X.XX seconds" is a myth is that load is not a single moment in time—it's an experience that no one metric can fully capture. There are multiple moments during the load experience that can affect whether a user perceives it as "fast", and if you just focus on one you might miss bad experiences that happen during the rest of the time.

For example, consider an app that optimizes for a fast initial render, delivering content to the user right away. If that app then loads a large JavaScript bundle that takes several seconds to parse and execute, the content on the page will not be interactive until after that JavaScript runs. If a user can see a link on the page but can't click on it, or if they can see a text box but can't type in it, they probably won't care how fast the page rendered.

So rather than measuring load with just one metric, we should be measuring the times of every moment throughout the experience that can have an affect on the user's load *perception*.

A second example of a performance myth is that ***performance is only a concern at load time***.

We as a team have been guilty of making this mistake, and it can be magnified by the fact that most performance tools *only* measure load performance.

But the reality is poor performance can happen at any time, not just during load. Apps that don't respond quickly to taps or clicks and apps that don't scroll or animate smoothly can be just as bad as apps that load slowly. Users care about the entire experience, and we developers should too.

A common theme in all of these performance misconceptions is they focus on things that have little or nothing to do with the user experience. Likewise, traditional performance metrics like load time or DOMContentLoaded time are extremely unreliable since when they occur may or may not correspond to when the user thinks the app is loaded.

So to ensure we don't repeat this mistake, we have to answer these questions:

1. What metrics most accurately measure performance as perceived by a human?
2. How do we measure these metrics on our actual users?
3. How do we interpret our measurements to determine whether an app is "fast"?
4. Once we understand our app's real-user performance, what do we do to prevent regressions and hopefully improve performance in the future?

User-centric performance metrics

When a user navigates to a web page, they're typically looking for visual feedback to reassure them that everything is going to work as expected.

Is it happening?	Did the navigation start successfully? Has the server responded?
Is it useful?	Has enough content rendered that users can engage with it?
Is it usable?	Can users interact with the page, or is it still busy loading?
Is it delightful?	Are the interactions smooth and natural, free of lag and jank?

To understand when a page delivers this feedback to its users, we've defined several new metrics:

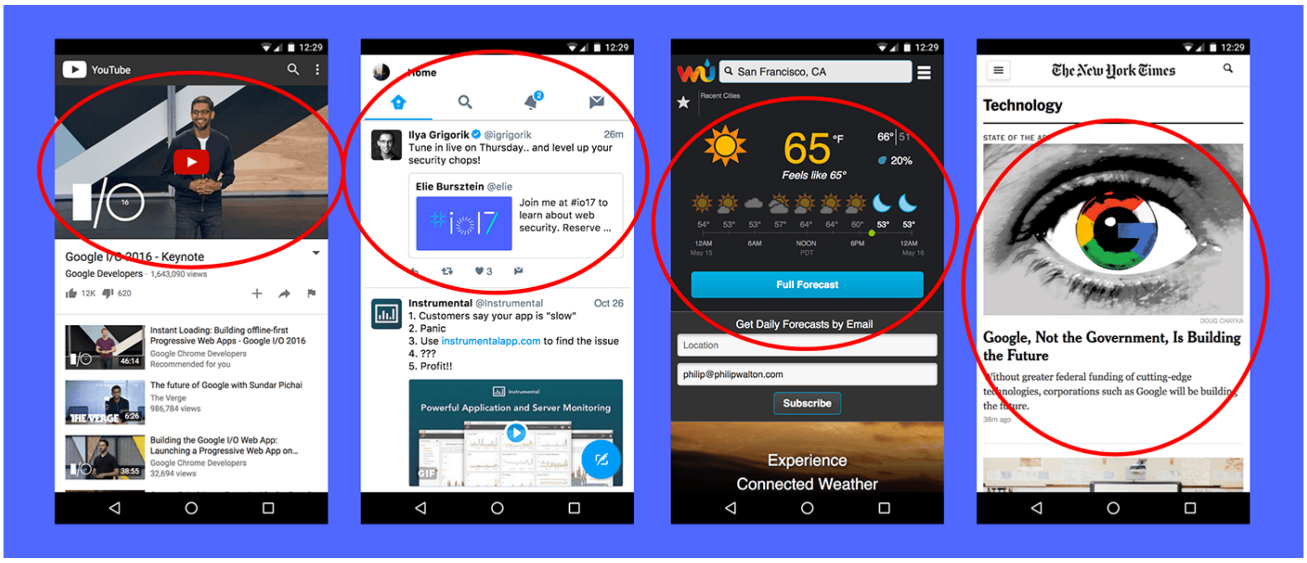
First paint and first contentful paint

The Paint Timing API defines two metrics: *first paint* (FP) and *first contentful paint* (FCP). These metrics mark the points, immediately after navigation, when the browser renders pixels to the screen. This is important to the user because it answers the question: *is it happening?*

The primary difference between the two metrics is FP marks the point when the browser renders *anything* that is visually different from what was on the screen prior to navigation. By contrast, FCP is the point when the browser renders the first bit of content from the DOM, which may be text, an image, SVG, or even a `<canvas>` element.

First meaningful paint and hero element timing

First meaningful paint (FMP) is the metric that answers the question: "is it useful?". While the concept of "useful" is very hard to spec in a way that applies generically to all web pages (and thus no spec exists, yet), it's quite easy for web developers themselves to know what parts of their pages are going to be most useful to their users.



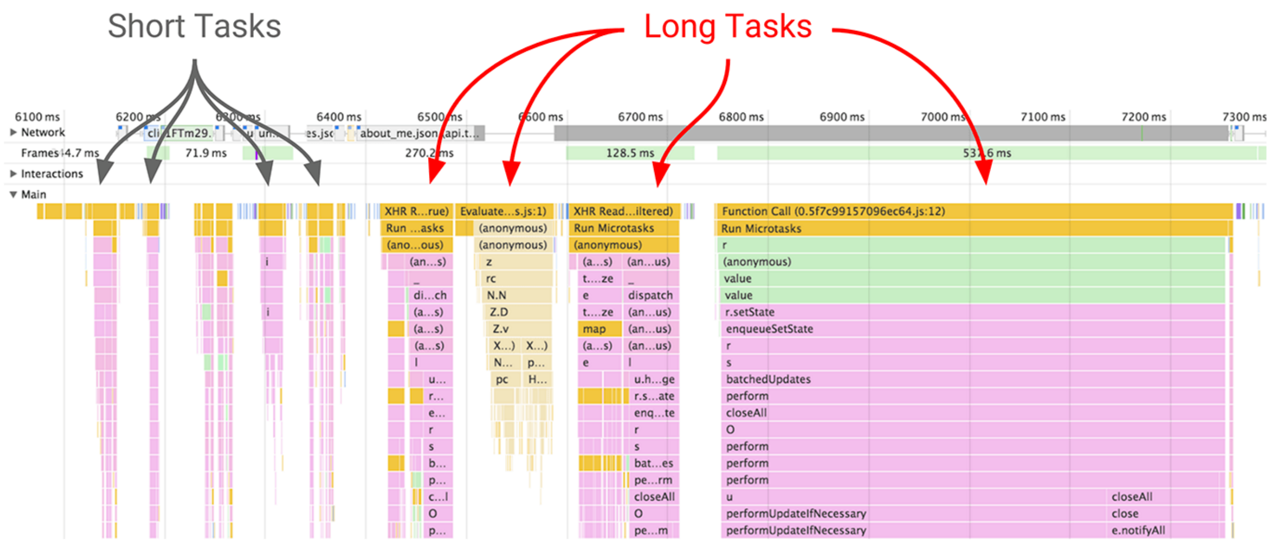
These "most important parts" of a web page are often referred to as *hero elements*. For example, on the YouTube watch page, the hero element is the primary video. On Twitter it's probably the notification badges and the first tweet. On a weather app it's the forecast for the specified location. And on a news site it's likely the primary story and featured image.

Web pages almost always have parts that are more important than others. If the most important parts of a page load quickly, the user may not even notice if the rest of the page doesn't.

Long tasks

Browsers respond to user input by adding tasks to a queue on the main thread to be executed one by one. This is also where the browser executes your application's JavaScript, so in that sense the browser is single-threaded.

In some cases, these tasks can take a long time to run, and if that happens, the main thread is blocked and all other tasks in the queue have to wait.



To the user this appears as lag or jank, and it's a major source of bad experiences on the web today.

The long tasks API identifies any task longer than 50 milliseconds as potentially problematic, and it exposes those tasks to the app developer. The 50 millisecond time was chosen so applications could meet the RAIL guidelines of responding to user input within 100 ms.

Time to interactive

The metric *Time to interactive* (TTI) marks the point at which your application is both visually rendered and capable of reliably responding to user input. An application could be unable to respond to user input for a couple of reasons:

- The JavaScript needed to make the components on the page work hasn't yet loaded.
- There are long tasks blocking the main thread (as described in the last section).

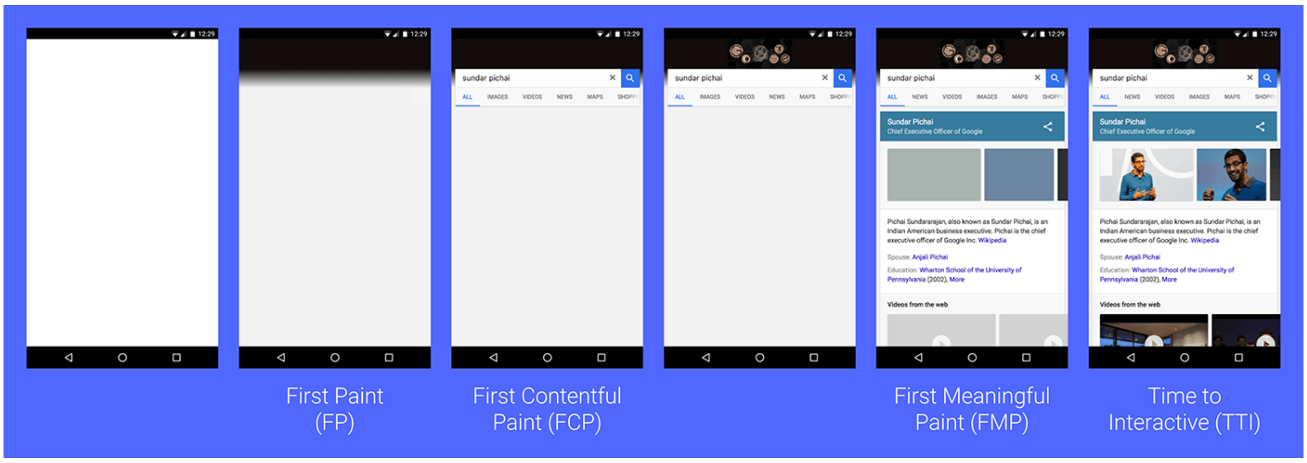
The TTI metric identifies the point at which the page's initial JavaScript is loaded and the main thread is idle (free of long tasks).

Mapping metrics to user experience

Getting back to the questions we previously identified as being the most important to the user experience, this table outlines how each of the metrics we just listed maps to the experience we hope to optimize:

The Experience	The Metric
Is it happening?	First Paint (FP) / First Contentful Paint (FCP)
Is it useful?	First Meaningful Paint (FMP) / Hero Element Timing
Is it usable?	Time to Interactive (TTI)
Is it delightful?	Long Tasks (technically the absence of long tasks)

And these screenshots of a load timeline should help you better visualize where the load metrics fit in the load experience:


First Paint (FP)
First Contentful Paint (FCP)
First Meaningful Paint (FMP)
Time to Interactive (TTI)

The next section details how to measure these metrics on real users' devices.

Measuring these metrics on real users' devices

One of the main reasons we've historically optimized for metrics like load and `DOMContentLoaded` is because they're exposed as events in the browser and easy to measure on real users.

By contrast, a lot of other metrics have been historically very hard to measure. For example, this code is a hack we often see developers use to detect long tasks:

```
(function detectLongFrame() {
  var lastFrameTime = Date.now();
  requestAnimationFrame(function() {
    var currentFrameTime = Date.now();
    if (currentFrameTime - lastFrameTime > 50) {
      // Report long frame here...
    }

    detectLongFrame(currentFrameTime);
  });
}());
```

This code starts an infinite `requestAnimationFrame` loop and records the time on each iteration. If the current time is more than 50 milliseconds after the previous time, it assumes it was the result of a long task. While this code mostly works, it has a lot of downsides:

- It adds overhead to every frame.
- It prevents idle blocks.
- It's terrible for battery life.

The most important rule of performance measurement code is that it shouldn't make performance worse.

Services like Lighthouse and Web Page Test have offered some of these new metrics for a while now (and in general they're great tools for testing performance on features prior to releasing them), but these tools don't run on your user's devices, so they don't reflect the actual performance experience of your users.

Luckily, with the addition of a few new browser APIs, measuring these metrics on real devices is finally possible without a lot of hacks or workarounds that can make performance worse.

These new APIs are `PerformanceObserver`, `PerformanceEntry`, and `DOMHighResTimeStamp`. To show some code with these new APIs in action, the following code example creates a new `PerformanceObserver` instance and subscribes to be notified about paint entries (e.g. FP and FCP) as well as any long tasks that occur:

```
const observer = new PerformanceObserver((list) => {
  for (const entry of list.getEntries()) {
    // `entry` is a PerformanceEntry instance.
    console.log(entry.entryType);
    console.log(entry.startTime); // DOMHighResTimeStamp
    console.log(entry.duration); // DOMHighResTimeStamp
  }
});
```

```
// Start observing the entry types you care about.
observer.observe({entryTypes: ['resource', 'paint']});
```

What `PerformanceObserver` gives us that we've never had before is the ability to subscribe to performance events as they happen and respond to them in an asynchronous fashion. This replaces the older PerformanceTiming interface, which often required polling to see when the data was available.

Tracking FP/FCP

Once you have the data for a particular performance event, you can send it to whatever analytics service you use to capture the metric for the current user. For example, using Google Analytics you might track first paint times as follows:

```
<head>
  <!-- Add the async Google Analytics snippet first. -->
  <script>
  window.ga=window.ga||function(){(ga.q=ga.q||[]).push(arguments)};ga.l=+new Date
  ga('create', 'UA-XXXXX-Y', 'auto');
  ga('send', 'pageview');
  </script>
  <script async src='https://www.google-analytics.com/analytics.js'></script>

  <!-- Register the PerformanceObserver to track paint timing. -->
  <script>
  const observer = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      // `name` will be either 'first-paint' or 'first-contentful-paint'.
      const metricName = entry.name;
      const time = Math.round(entry.startTime + entry.duration);

      ga('send', 'event', {
        eventCategory: 'Performance Metrics',
        eventAction: metricName,
        eventValue: time,
        nonInteraction: true,
      });
    }
  });
  observer.observe({entryTypes: ['paint']});
  </script>

  <!-- Include any stylesheets after creating the PerformanceObserver. -->
  <link rel="stylesheet" href="...">
</head>
```

Important: you must ensure your `PerformanceObserver` is registered in the `<head>` of your document before any stylesheets, so it runs before FP/FCP happens.

This will no longer be necessary once Level 2 of the Performance Observer spec is implemented, as it introduces a **`buffered`** flag that allows you to access performance entries queued prior to the `PerformanceObserver` being created.

Tracking FMP using hero elements

Once you've identified what elements on the page are the hero elements, you'll want to track the point at which they're visible to your users.

We don't yet have a standardized definition for FMP (and thus no performance entry type either). This is in part because of how difficult it is to determine, in a generic way, what "meaningful" means for all pages.

However, in the context of a single page or a single application, it's generally best to consider FMP to be the moment when your hero elements are visible on the screen.

Steve Souders has a great article called User Timing and Custom Metrics that details many of the techniques for using browser's performance APIs to determine in code when various types of media are visible.

Tracking TTI

In the long term, we hope to have a TTI metric standardized and exposed in the browser via PerformanceObserver. In the meantime, we've developed a polyfill that can be used to detect TTI today and works in any browser that supports the Long Tasks API.

The polyfill exposes a **`getFirstConsistentlyInteractive()`** method, which returns a promise that resolves with the TTI value. You can track TTI using Google Analytics as follows:

```
import ttiPolyfill from './path/to/tti-polyfill.js';

ttiPolyfill.getFirstConsistentlyInteractive().then((tti) => {
  ga('send', 'event', {
    eventCategory: 'Performance Metrics',
    eventAction: 'TTI',
    eventValue: tti,
    nonInteraction: true,
  });
});
```

The `getFirstConsistentlyInteractive()` method accepts an optional `startTime` configuration option, allowing you to specify a lower bound for which you know your app cannot be interactive before. By default the polyfill uses DOMContentLoaded as the start time, but it's often more accurate to use something like the moment your hero elements are visible or the point when you know all your event listeners have been added.

Refer to the TTI polyfill documentation for complete installation and usage instructions.

Note: As with FMP, it's quite hard to spec a TTI metric definition that works perfectly for all web pages. The version we've implemented in the polyfill will work for most apps, but it's possible it won't work for your particular app. It's important that you test it before relying on it. If you want more details on the specifics of the TTI definition and implementation you can read the TTI metric definition doc.

Tracking long tasks

I mentioned above that long tasks will often cause some sort of negative user experience (e.g. a sluggish event handler or a dropped frame). It's good to be aware of how often this is happening, so you can make efforts to minimize it.

To detect long tasks in JavaScript you create a new `PerformanceObserver` and observe entries of type `longtask`. One nice feature of long task entries is they contain an attribution property, so you can more easily track down which code caused the long task:

```
const observer = new PerformanceObserver((list) => {
  for (const entry of list.getEntries()) {
    ga('send', 'event', {
      eventCategory: 'Performance Metrics',
      eventAction: 'longtask',
      eventValue: Math.round(entry.startTime + entry.duration),
      eventLabel: JSON.stringify(entry.attribution),
    });
  }
});

observer.observe({entryTypes: ['longtask']});
```

The attribution property will tell you what frame context was responsible for the long task, which is helpful in determining if third party iframe scripts are causing issues. Future versions of the spec are planning to add more granularity and expose script URL, line, and column number, which will be very helpful in determining if your own scripts are causing slowness.

Tracking input latency

Long tasks that block the main thread can prevent your event listeners from executing in a timely manner. The RAIL performance model teaches us that in order for a user interface to feel smooth, it should respond within 100 ms of user input, and if this isn't happening, it's important to know about it.

To detect input latency in code you can compare the event's time stamp to the current time, and if the difference is larger than 100 ms, you can (and should) report it.

```
const subscribeBtn = document.querySelector('#subscribe');

subscribeBtn.addEventListener('click', (event) => {
  // Event listener logic goes here...

  const lag = performance.now() - event.timeStamp;
  if (lag > 100) {
    ga('send', 'event', {
      eventCategory: 'Performance Metric'
      eventAction: 'input-latency',
      eventLabel: '#subscribe:click',
      eventValue: Math.round(lag),
      nonInteraction: true,
    });
  }
});
```

Since event latency is usually the result of a long task, you can combine your event latency detection logic with your long task detection logic: if a long task was blocking the main thread at the same time as `event.timeStamp` you could report that long task's attribution value as well. This would allow you to draw a very clear line between negative performance experiences and the code that caused it.

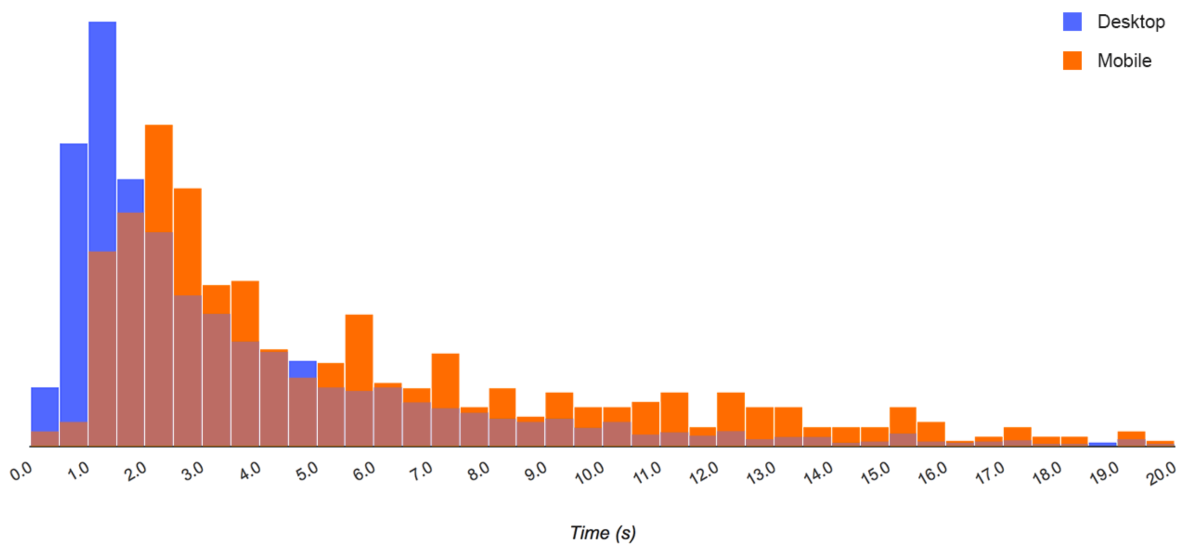
While this technique isn't perfect (it doesn't handle long event listeners later in the propagation phase, and it doesn't work for scrolling or composited animations that don't run on the main thread), it's a good first step into better understanding how often long running JavaScript code affects user experience.

Interpreting the data

Once you've started collecting performance metrics for real users, you need to put that data into action. Real-user performance data is useful for a few primary reasons:

- Validating that your app performs as expected.
- Identifying places where poor performance is negatively affecting conversions (whatever that means for your app).
- Finding opportunities to improve the user experience and delight your users.

One thing definitely worth comparing is how your app performs on mobile devices vs desktop. The following chart shows the distribution of TTI across desktop (blue) and mobile (orange). As you can see from this example, the TTI value on mobile was quite a bit longer than on desktop:



While the numbers here are app-specific (and you shouldn't assume they'd match your numbers, you should test for yourself), this gives you an example of how you might approach reporting on your usage metrics:

Desktop

Percentile	TTI (seconds)
50%	2.3
75%	4.7
90%	8.3

Mobile

Percentile	TTI (seconds)
50%	3.9
75%	8.0
90%	12.6

Breaking your results down across mobile and desktop and analyzing the data as a distribution allows you to get quick insight into the experiences of real users. For example, looking at the above table, I can easily see that, for this app, **10% of mobile users took longer than 12 seconds to become interactive!**

How performance affects business

One huge advantage of tracking performance in your analytics tools is you can then use that data to analyze how performance affects business.

If you're tracking goal completions or ecommerce conversions in analytics, you could create reports that explore any correlations between these and the app's performance metrics. For example:

- Do users with faster interactive times buy more stuff?
- Do users who experience more long tasks during the checkout flow drop off at higher rates?

If correlations are found, it'll be substantially easier to make the business case that performance is important and should be prioritized.

Load abandonment

We know that users will often leave if a page takes too long to load. Unfortunately, this means that all of our performance metrics share the problem of survivorship bias, where the data doesn't include load metrics from people who didn't wait for the page to finish loading (which likely means the numbers are too low).

While you can't track what the numbers would have been if those users had stuck around, you can track how often this happens as well as how long each user stayed for.

This is a bit tricky to do with Google Analytics since the analytics.js library is typically loaded asynchronously, and it may not be available when the user decides to leave.

However, you don't need to wait for analytics.js to load before sending data to Google Analytics. You can send it directly via the Measurement Protocol.

This code adds a listener to the `visibilitychange` event (which fires if the page is being unloaded or goes into the background) and it sends the value of `performance.now()` at that point.

```
<script>
window.__trackAbandons = () => {
  // Remove the listener so it only runs once.
  document.removeEventListener('visibilitychange', window.__trackAbandons);
  const ANALYTICS_URL = 'https://www.google-analytics.com/collect';
  const GA_COOKIE = document.cookie.replace(
    /(?:(?:^|.*;)\s*_ga\s*\=\s*(?:\w+\.\d\.)([^;]*).*$)|^.*$/, '$1');
  const TRACKING_ID = 'UA-XXXXX-Y';
  const CLIENT_ID =  GA_COOKIE || (Math.random() * Math.pow(2, 52));

  // Send the data to Google Analytics via the Measurement Protocol.
  navigator.sendBeacon && navigator.sendBeacon(ANALYTICS_URL, [
    'v=1', 't=event', 'ec=Load', 'ea=abandon', 'ni=1',
    'dl=' + encodeURIComponent(location.href),
    'dt=' + encodeURIComponent(document.title),
    'tid=' + TRACKING_ID,
    'cid=' + CLIENT_ID,
    'ev=' + Math.round(performance.now()),
  ].join('&'));
};
document.addEventListener('visibilitychange', window.__trackAbandons);
</script>
```

You can use this code by copying it into `<head>` of your document and replacing the `UA-XXXXX-Y` placeholder with your tracking ID.

You'll also want to make sure you remove this listener once the page becomes interactive or you'll be reporting abandonment for loads where you were also reporting TTI.

```
document.removeEventListener('visibilitychange', window.__trackAbandons);
```

Optimizing performance and preventing regression

The great thing about defining user-centric metrics is when you optimize for them, you inevitably improve user experience as well.

One of the simplest ways to improve performance is to just ship less JavaScript code to the client, but in cases where reducing code size is not an option, it's critical that you think about *how* you deliver your JavaScript.

Optimizing FP/FCP

You can lower the time to first paint and first contentful paint by removing any render blocking scripts or stylesheets from the `<head>` of your document.

By taking the time to identify the minimal set of styles needed to show the user that "it's happening" and inlining them in the `<head>` (or using HTTP/2 server push), you can get incredibly fast first paint times.

The app shell pattern is a great example of how to do this for Progressive Web Apps.

Optimizing FMP/TTI

Once you've identified the most critical UI elements on your page (the hero elements), you should ensure that your initial script load contains just the code needed to get those elements rendered and make them interactive.

Any code unrelated to your hero elements that is included in your initial JavaScript bundle will slow down your time to interactivity. There's no reason to force your user's devices to download and parse JavaScript code they don't need right away.

As a general rule, you should try as hard as possible to minimize the time between FMP and TTI. In cases where it's not possible to minimize this time, it's absolutely critical that your interfaces make it clear that the page isn't yet interactive.

One of the most frustrating experiences for a user is tapping on an element and then having nothing happen.

Preventing long tasks

By splitting up your code and prioritizing the order in which it's loaded, not only can you get your pages interactive faster, but you can reduce long tasks and then hopefully have less input latency and fewer slow frames.

In addition to splitting up code into separate files, you can also split up large chunks of synchronous code into smaller chunks that can execute asynchronously or be deferred to the next idlepoint. By executing this logic asynchronously in smaller chunks, you leave room on the main thread for the browser to respond to user input.

Lastly, you should make sure you're testing your third party code and holding any slow running code accountable. Third party ads or tracking scripts that cause lots of long tasks may end up hurting your business more than they're helping it.

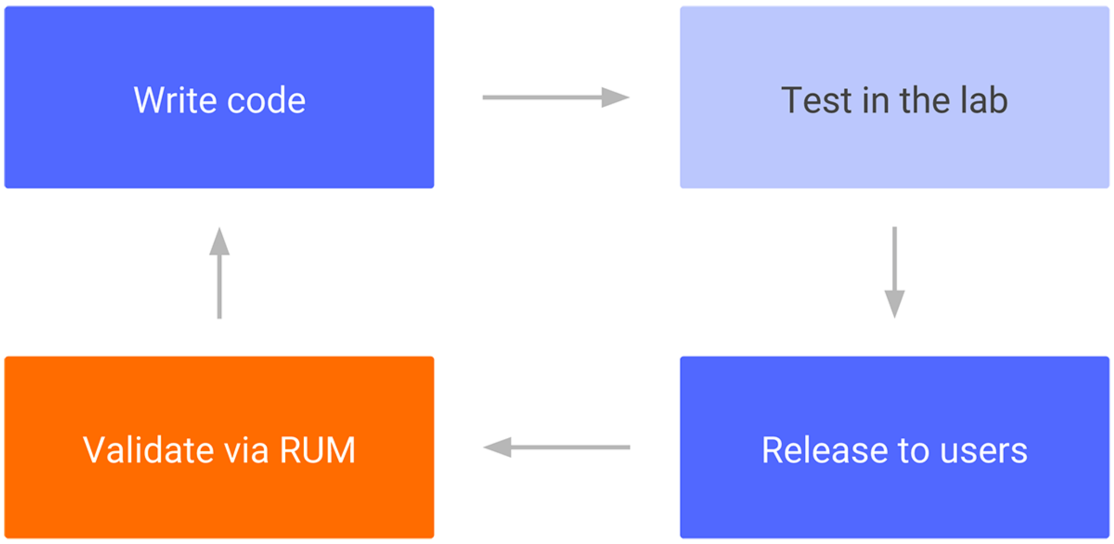
Preventing regressions

This article has focused heavily on performance measurement on real users, and while it's true that RUM data is the performance data that ultimately matters, lab data is still critical in ensuring your app performs well (and doesn't regress) prior to releasing new features. Lab tests are ideal for regression detection, as they run in a controlled environment and are far less prone to the random variability of RUM tests.

Tools like Lighthouse and Web Page Test can be integrated into your continuous integration server, and you can write tests that fail a build if key metrics regress or drop below a certain threshold.

And for code already released, you can add custom alerts to inform you if there are unexpected spikes in the occurrence of negative performance events. This could happen, for example, if a third party releases a new version of one of their services and suddenly your users start seeing significantly more long tasks.

To successfully prevent regressions you need to be testing performance in both the lab and the wild with every new feature releases.



Wrapping up and looking forward

We've made significant strides in the last year in exposing user-centric metrics to developers in the browser, but we're not done yet, and we have a lot more planned.

We'd really like to standardize time to interactive and hero element metrics, so developers won't have to measure these themselves or depend on polyfills. We'd also like to make it easier for developers to attribute dropped frames and input latency to particular long tasks and the code that caused them.

While we have more work to do, we're excited about the progress we've made. With new APIs like `PerformanceObserver` and long tasks supported natively in the browser, developers finally have the primitives they need to measure performance on real users without degrading their experience.

The metrics that matter the most are the ones that represent real user experiences, and we want to make it as easy as possible for developers to delight their users and create great applications.

Staying connected

File spec issues:

- https://github.com/w3c/longtasks/issues
- https://github.com/WICG/paint-timing/issues
- https://github.com/w3c/performance-timeline/issues

File polyfill issues:

- https://github.com/GoogleChrome/tti-polyfill/issues

Ask questions:

- progressive-web-metrics@chromium.org
- public-web-perf@w3.org

Voice your support on concerns on new API proposals:

- https://github.com/w3c/charter-webperf/issues
