Script evaluation and long tasks

- What is script evaluation?
- The relationship between scripts and the tasks that evaluate them
  - Loading scripts with the <script> element
  - Loading scripts with the <script> element and the type=module attribute
    - Chromium-based browsers
    - Safari and Firefox
  - Loading scripts with dynamic import()
  - Loading scripts in a web worker
- Trade-offs and considerations
  - Compression efficiency
  - Cache invalidation
  - Nested modules and loading performance
- Conclusion

When it comes to optimizing Interaction to Next Paint (INP), most of the advice you'll encounter is to optimize interactions themselves. For example, in the optimize long tasks guide, techniques such as yielding with `setTimeout`, `isInputPending`, and so forth are discussed. These techniques are beneficial, as they allow the main thread some breathing room by avoiding long tasks, which can allow more opportunities for interactions and other activity to run sooner, rather than if they had to wait for a single long task.

However, what about the long tasks that come from loading scripts themselves? These tasks can interfere with user interactions and affect a page's INP during load. This guide will explore how browsers handle tasks kicked off by script evaluation, and look into what you may be able to do to break up script evaluation work so that your main thread can be more responsive to user input while the page is loading.

What is script evaluation?

If you've profiled an application that ships a lot of JavaScript, you may have seen long tasks where the culprit is labeled **Evaluate Script**.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/WZyF71IiPWyG8vwKQcdP.png", alt="Script evaluation work as visualized in the performance profiler of Chrome DevTools. The work causes a long task during startup, which blocks the main thread's ability to respond to user interactions.", width="697", height="107" %}
> Script evaluation work as shown in the performance profiler in Chrome DevTools. In this case, the work is enough to cause a long task that blocks the main thread from taking on other work—including tasks that drive user interactions.

Script evaluation is a necessary part of executing JavaScript in the browser, as JavaScript is compiled just-in-time before execution. When a script is evaluated, it is first parsed for errors. If the parser doesn't find errors, the script is then compiled into bytecode, and then can continue onto execution.

Though necessary, script evaluation can be problematic, as users may try to interact with a page shortly after it initially renders. However, just because a page has *rendered* doesn't mean that the page has finished *loading*. Interactions that take place during load can be delayed because the page is busy evaluating scripts. While there's no guarantee that the desired interaction can take place at this point in time—as a script responsible for it may not have loaded yet—there could be interactions dependent on JavaScript that *are* ready, or the interactivity doesn't depend on JavaScript at all.

{% Aside %} One metric that might give you insight into whether you have excessive script evaluation occurring during page load is Total Blocking Time (TBT), as it is a load responsiveness metric. Because TBT correlates well with INP, a page with a high TBT is a reasonable indicator that there may be high INP values during load that may be tied to script evaluation work. {% endAside %}

The relationship between scripts and the tasks that evaluate them

How tasks responsible for script evaluation are kicked off depends on whether the script you're loading is loaded via a regular `<script>` element, or if a script is a module loaded with the `type=module`. Since browsers have the tendency to handle things differently, how the major browser

engines handle script evaluation will be touched upon where script evaluation behaviors across them vary.

Loading scripts with the `<script>` element

{% Aside 'important' %} This section deals with loading scripts using the `<script>` element *without* the `type=module` attribute. {% endAside%}

The number of tasks dispatched to evaluate scripts generally has a direct relationship with the number of `<script>` elements on a page. Each `<script>` element kicks off a task to evaluate the requested script so it can be parsed, compiled, and executed. **This is the case for Chromium-based browsers, Safari, *and* Firefox.**

Why does this matter? Let's say you're using a bundler to manage your production scripts, and you've configured it to bundle everything your page needs to run into a single script. If this is the case for your website, you can expect that there will be a single task dispatched to evaluate that script. Is this a bad thing? Not necessarily—unless that script is *huge*.

You can break up script evaluation work by avoiding loading large chunks of JavaScript, and load more individual, smaller scripts using additional `<script>` elements.

{% Aside 'important' %} As devices vary in their capability, it's very difficult to define a set limit for how large individual scripts should be. To achieve a decent balance between compression efficiency, download time, and script evaluation time, a limit of 100 kilobytes per individual script is a good target. {% endAside %}

While you should always strive to load as little JavaScript as possible during page load, splitting up your scripts ensures that, instead of one large task that may block the main thread, you have a greater number of smaller tasks that won't block the main thread at all—or at least less than what you started with.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/dajGS6urKufQoweVhJQh.png", alt="Multiple tasks involving script evaluation as visualized in the performance profiler of Chrome DevTools. Because multiple smaller scripts are loaded instead of fewer larger scripts, tasks are less likely to become long tasks, allowing the main thread to respond to user input more quickly.", width="800", height="174" %}
> Multiple tasks spawned to evaluate scripts as a result of multiple `<script>` elements present in the page's HTML. This is preferable to sending one large script bundle to users, which is more likely to block the main thread.

You can think of breaking up tasks for script evaluation as being somewhat similar to yielding during event callbacks that run during an interaction. However, with script evaluation, the yielding mechanism breaks up the JavaScript you load into multiple smaller scripts, rather than a smaller number of larger scripts than are more likely to block the main thread.

{% Aside 'important' %} Currently, Chromium-based browsers will execute all loaded scripts with the `defer` attribute in the same task as the `DOMContentLoaded` event. While this minimizes overall layout work, the cost is the increased likelihood of a longer task, which may cause other performance problems. A potential solution is currently being explored. This behavior also takes place with scripts loaded using the `type=module` attribute, as such scripts are deferred by default. {% endAside %}

Loading scripts with the `<script>` element and the `type=module` attribute

{% Aside 'important' %} If you're not bundling ES modules and loading them using the `type=module` attribute, you could actually end up slowing down page startup. For more information, read the trade-offs and considerations section later on in this guide. {% endAside %}

It's now possible to load ES modules natively in the browser with the `type=module` attribute on the `<script>` element. This approach to script loading carries some developer experience benefits, such as not having to transform code for production use—especially when used in combination with import maps. However, loading scripts in this way schedules tasks that differ from browser to browser.

Chromium-based browsers

In browsers such as Chrome—or those derived from it—loading ES modules using the `type=module` attribute produces different sorts of tasks than you'd normally see when not using `type=module`. For example, a task for each module script will run that involves activity labeled as **Compile module**.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/aEGTjOqtruKet5I6sCLM.png", alt="Module compilation work in multiple tasks as visualized in Chrome DevTools.", width="800", height="140" %}
> Module loading behavior in Chromium-based browsers. Each module script will spawn a **Compile module** call to compile their contents prior to evaluation.

Once the modules have compiled, any code that subsequently runs in them will kick off activity labeled as **Evaluate module**.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/CQpGnJmXh9JSUIMuBpeu.png", alt="Just-in-time evaluation of a module as visualized in the performance panel of Chrome DevTools.", width="800", height="184" %}
> When code in a module runs, that module will be evaluated just-in-time.

The effect here—in Chrome and related browsers, at least—is that the compilation steps are broken up when using ES modules. This is a clear win in terms of managing long tasks; however, the resulting module evaluation work that results still means you're incurring some unavoidable cost. While you should strive to ship as little JavaScript as possible, using ES modules—regardless of the browser—provides the following benefits:

- All module code is automatically run in strict mode, which allows potential optimizations by JavaScript engines that couldn't otherwise be made in a non-strict context.
- Scripts loaded using `type=module` are treated as if they were deferred by default. It's possible to use the `async` attribute on scripts loaded with `type=module` to change this behavior.

Safari and Firefox

When modules are loaded in Safari and Firefox, each of them is evaluated in a separate task. This means you could theoretically load a single top-level module consisting of only static `import` statements to other modules, and every module loaded will incur a separate network request and task to evaluate it.

Loading scripts with dynamic `import()`

Dynamic `import()` is another method for loading scripts. Unlike static `import` statements that are required to be at the top of an ES module, a dynamic `import()` call can appear anywhere in a script to load a chunk of JavaScript on demand. This technique is called code splitting.

Dynamic `import()` has two advantages when it comes to improving INP:

1. Modules which are deferred to load later reduce main thread contention during startup by reducing the amount of JavaScript loaded at that time. This frees up the main thread so it can be more responsive to user interactions.
2. When dynamic `import()` calls are made, each call will effectively separate the compilation and evaluation of each module to its own task. Of course, a dynamic `import()` that loads a very large module will kick off a rather large script evaluation task, and that can interfere with the ability of the main thread to respond to user input if the interaction occurs at the same time as the dynamic `import()` call. **Therefore, it's still very important that you load as little JavaScript as possible.**

Dynamic `import()` calls behave similarly in all major browser engines: the script evaluation tasks that result will be the same as the amount of modules that are dynamically imported.

Loading scripts in a web worker

Web workers are a special JavaScript use case. Web workers are registered on the main thread, and the code within the worker then runs on its own thread. This is hugely beneficial in the sense that—while the code that registers the web worker runs on the main thread—the code within the web worker doesn't. This reduces main thread congestion, and can help keep the main thread more responsive to user interactions.

In addition to reducing main thread work, web workers *themselves* can load external scripts to be used in the worker context, either through `importScripts` or static `import` statements in browsers that support module workers. The result is that any script requested by a web worker is evaluated off the main thread.

{% Aside 'objective' %} **Read to learn more:** Use web workers to run JavaScript off the browser's main thread. {% endAside %}

Trade-offs and considerations

While breaking up your scripts into separate, smaller files helps limit long tasks as opposed to loading fewer, much larger files, it's important to take some things into account when deciding how to break scripts up.

Compression efficiency

Compression is a factor when it comes to breaking up scripts. When scripts are smaller, compression becomes somewhat less efficient. Larger scripts will benefit much more from compression. While increasing compression efficiency helps to keep load times for scripts as low as possible, it's a bit of a balancing act to ensure that you're breaking up scripts into enough smaller chunks to facilitate better interactivity during startup.

Bundlers are ideal tools for managing the output size for the scripts your website depends on:

- Where webpack is concerned, its `SplitChunksPlugin` plugin can help. Consult the `SplitChunksPlugin` documentation for options you can set to help manage asset sizes.
- For other bundlers such as Rollup and esbuild, you can manage script file sizes by using dynamic `import()` calls in your code. These bundlers—as well as webpack—will automatically break off the dynamically imported asset into its own file, thus avoiding larger initial bundle sizes.

Cache invalidation

Cache invalidation plays a big role in how fast a page loads on repeat visits. When you ship large, monolithic script bundles, you're at a disadvantage when it comes to browser caching. This is because when you update your first-party code—either through updating packages or shipping bug fixes—the entire bundle becomes invalidated and must be downloaded again.

By breaking up your scripts, you're not just breaking up script evaluation work across smaller tasks, you're also increasing the likelihood that return visitors will grab more scripts from the browser cache instead of from the network. This translates into an overall faster page load.

{% Aside 'important' %} In order for caching to be both efficient and to avoid stale resources from being served from the cache, be sure that your bundler is generating resources with a hash in the file name. {% endAside %}

Nested modules and loading performance

If you're shipping ES modules in production and loading them with the `type=module` attribute, you need to be aware of how module nesting can impact startup time. Module nesting refers to when an ES module statically imports another ES module that statically imports another ES module:

```
// a.js
import {b} from './b.js';

// b.js
import {c} from './c.js';
```

If your ES modules are not bundled together, the preceding code results in a network request chain: when `a.js` is requested from a `<script>` element, another network request is dispatched for `b.js`, which then involves *another* request for `c.js`. One way to avoid this is to use a bundler—but be sure you're configuring your bundler to break up scripts to spread out script evaluation work.

If you don't want to use a bundler, then another way to get around nested module calls is to use the `modulepreload` resource hint, which will preload ES modules ahead of time to avoid network request chains. Beware, however: this hint is currently only supported in Chromium-based browsers.

Conclusion

Optimizing evaluation of scripts in the browser is no doubt a tricky feat. The approach depends on your website's requirements and constraints. However, by splitting up scripts, you're spreading the work of script evaluation over numerous smaller tasks, and therefore giving the main thread the ability to handle user interactions more efficiently, rather than blocking the main thread.

To recap, here are some things you can to do to break up large script evaluation tasks:

- When loading scripts using the `<script>` element without the `type=module` attribute, avoid loading scripts that are very large, as these will kick off resource-intensive script evaluation tasks that block the main thread. Spread out your scripts over more `<script>` elements to break up this work.
- Using the `type=module` attribute to load ES modules natively in the browser will kick off individual tasks for evaluation for each separate module script.
- Reduce the size of your initial bundles by using dynamic `import()` calls. This also works in bundlers, as bundlers will treat each dynamically imported module as a "split point," resulting in a separate script being generated for each dynamically imported module.
- Be sure to weigh trade-offs such as compression efficiency and cache invalidation. Larger scripts will compress better, but are more likely to involve more expensive script evaluation work in fewer tasks, and result in browser cache invalidation, leading to overall lower caching efficiency.
- If using ES modules natively without bundling, use the `modulepreload` resource hint to optimize the loading of them during startup.
- As always, ship as little JavaScript as possible.

It's a balancing act for sure—but by breaking up scripts and reducing initial payloads via dynamic `import()`, you can achieve better startup performance and better accommodate user interactions during that crucial startup period. This should help you score better on the INP metric, thus delivering a better user experience.

Hero image from Unsplash, by Markus Spiske.

□□□ #2
□ □□□ □□□ 2026-10-31 22:42:34 UTC
□ □□□ □□□ 2026-10-31 22:42:36 UTC