

Performance



- [Review Your Browser Setup](#)
 - [Audit Configured Vite Plugins](#)
 - [Reduce Resolve Operations](#)
 - [Avoid Barrel Files](#)
 - [Warm Up Frequently Used Files](#)
 - [Use Lesser or Native Tooling](#)
-

While Vite is fast by default, performance issues can creep in as the project's requirements grow. This guide aims to help you identify and fix common performance issues, such as:

- Slow server starts
- Slow page loads
- Slow builds

Review Your Browser Setup

Some browser extensions may interfere with requests and slow down startup and reload times for large apps, especially when using browser dev tools. We recommend creating a dev-only profile without extensions, or switch to incognito mode, while using Vite's dev server in these cases. Incognito mode should also be faster than a regular profile without extensions.

The Vite dev server does hard caching of pre-bundled dependencies and implements fast 304 responses for source code. Disabling the cache while the Browser Dev Tools are open can have a big impact on startup and full-page reload times. Please check that "Disable Cache" isn't enabled while you work with the Vite server.

Audit Configured Vite Plugins

Vite's internal and official plugins are optimized to do the least amount of work possible while providing compatibility with the broader ecosystem. For example, code transformations use regex in dev, but do a complete parse in build to ensure correctness.

However, the performance of community plugins is out of Vite's control, which may affect the developer experience. Here are a few things you can look out for when using additional Vite plugins:

1. Large dependencies that are only used in certain cases should be dynamically imported to reduce the Node.js startup time. Example refactors: [vite-plugin-react#212](#) and [vite-plugin-pwa#244](#).
2. The `buildStart`, `config`, and `configResolved` hooks should not run long and extensive operations. These hooks are awaited during dev server startup, which delays when you can access the site in the browser.
3. The `resolveId`, `load`, and `transform` hooks may cause some files to load slower than others. While sometimes unavoidable, it's still worth checking for possible areas to optimize. For example, checking if the `code` contains a specific keyword, or the `id` matches a specific extension, before doing the full transformation.

The longer it takes to transform a file, the more significant the request waterfall will be when loading the site in the browser.

You can inspect the duration it takes to transform a file using `vite --debug plugin-transform` or [vite-plugin-inspect](#). Note that as asynchronous operations tend to provide inaccurate timings, you should treat the numbers as a rough estimate, but it should still reveal the more expensive operations.

::: tip Profiling You can run `vite --profile`, visit the site, and press `p + enter` in your terminal to record a `.cpuprofile`. A tool like [speedscope](#) can then be used to inspect the profile and identify the bottlenecks. You can also [share the profiles](#) with the Vite team to help us identify performance issues. :::

Reduce Resolve Operations

Resolving import paths can be an expensive operation when hitting its worst case often. For example, Vite supports "guessing" import paths with the `resolve.extensions` option, which defaults to `['.mjs', '.js', '.mts', '.ts', '.jsx', '.tsx', '.json']`.

When you try to import `./Component.jsx` with `import './Component'`, Vite will run these steps to resolve it:

1. Check if `./Component` exists, no.
2. Check if `./Component.mjs` exists, no.
3. Check if `./Component.js` exists, no.
4. Check if `./Component.mts` exists, no.
5. Check if `./Component.ts` exists, no.
6. Check if `./Component.jsx` exists, yes!

As shown, a total of 6 filesystem checks is required to resolve an import path. The more implicit imports you have, the more time it adds up to resolve the paths.

Hence, it's usually better to be explicit with your import paths, e.g. `import './Component.jsx'`. You can also narrow down the list for `resolve.extensions` to reduce the general filesystem checks, but you have to make sure it works for files in `node_modules` too.

If you're a plugin author, make sure to only call `this.resolve` when needed to reduce the number of checks above.

::: tip TypeScript If you are using TypeScript, enable `"moduleResolution": "bundler"` and `"allowImportingTsExtensions": true` in your `tsconfig.json`'s `compilerOptions` to use `.ts` and `.tsx` extensions directly in your code. :::

Avoid Barrel Files

Barrel files are files that re-export the APIs of other files in the same directory. For example:

```
export * from './color.js'
export * from './dom.js'
export * from './slash.js'
```

When you only import an individual API, e.g. `import { slash } from './utils'`, all the files in that barrel file need to be fetched and transformed as they may contain the `slash` API and may also contain side-effects that run on initialization. This means you're loading more files than required on the initial page load, resulting in a slower page load.

If possible, you should avoid barrel files and import the individual APIs directly, e.g. `import { slash } from './utils/slash.js'`. You can read [issue #8237](#) for more information.

Warm Up Frequently Used Files

The Vite dev server only transforms files as requested by the browser, which allows it to start up quickly and only apply transformations for used files. It can also pre-transform files if it anticipates certain files will be requested shortly. However, request waterfalls may still happen if some files take longer to transform than others. For example:

Given an import graph where the left file imports the right file:

```
main.js -> BigComponent.vue -> big-utils.js -> large-data.json
```

The import relationship can only be known after the file is transformed. If `BigComponent.vue` takes some time to transform, `big-utils.js` has to wait for its turn, and so on. This causes an internal waterfall even with pre-transformation built-in.

Vite allows you to warm up files that you know are frequently used, e.g. `big-utils.js`, using the `server.warmup` option. This way `big-utils.js` will be ready and cached to be served immediately when requested.

You can find files that are frequently used by running `vite --debug transform` and inspect the logs:

```
vite:transform 28.72ms /@vite/client +1ms
vite:transform 62.95ms /src/components/BigComponent.vue +1ms
vite:transform 102.54ms /src/utils/big-utils.js +1ms
```

```
export default defineConfig({
  server: {
    warmup: {
      clientFiles: [
        './src/components/BigComponent.vue',
        './src/utils/big-utils.js',
      ],
    },
  },
})
```

Note that you should only warm up files that are frequently used to not overload the Vite dev server on startup. Check the `server.warmup` option for more information.

Using `--open` or `server.open` also provides a performance boost, as Vite will automatically warm up the entry point of your app or the provided URL to open.

Use Lesser or Native Tooling

Keeping Vite fast with a growing codebase is about reducing the amount of work for the source files (JS/TS/CSS).

Examples of doing less work:

- Use CSS instead of Sass/Less/Stylus when possible (nesting can be handled by PostCSS / Lightning CSS)

- Don't transform SVGs into UI framework components (React, Vue, etc.). Import them as strings or URLs instead.

Examples of using native tooling:

While Vite core is based on native tooling, some features still use non-native tooling by default to provide better compatibility and feature set. But it may be worth the cost for larger applications.

- Try out the experimental support for [LightningCSS](#)

👤	#2	
👤	👤	2026-10-31 13:14:40 UTC
👤	👤	2026-10-31 13:14:42 UTC