03

3 ‘silent killers’ that hurt your user experience

LOGROCKET.COM

Introduction

Understanding performance in frontend applications is hard. While performance issues are sometimes reported by users or caught in QA, users who have a bad experience often just leave or suffer in silence, causing many problems to go unreported.

There are a host of well-known factors that can affect client-side performance: bundle size, caching practices, external asset usage, content delivery methods, and much more. But there are also more nefarious issues lurking beneath the surface — we call them "silent killers."

These issues have a major negative effect on your users, but they don't show up in traditional monitoring or error reporting tools. While these tools can help developers monitor the backends of server-side apps — infrastructure, services, databases — they are often limited in their frontend telemetry, leaving silent killers to continue unnoticed.

Even tools meant for frontend monitoring often lack information about how users are interacting with your app before a problem occurs, as well as how those users actually experience the problems themselves. Without this context, it can be difficult and time-consuming to determine what's truly affecting your users and what needs to be done next.

Proactively seeking to understand frontend performance issues and how they affect your users can help you identify the silent killers that are hard to efficiently pin down and resolve. With crucial context around user behavior, you can cut through the noise to better understand the root causes of frontend issues, prioritize your resources, and get ahead of problems to provide a smoother user experience overall.

In this guide, we'll introduce three examples of silent killers lurking in your applications, explore what makes them difficult to identify, and illustrate how they may be impacting your users.

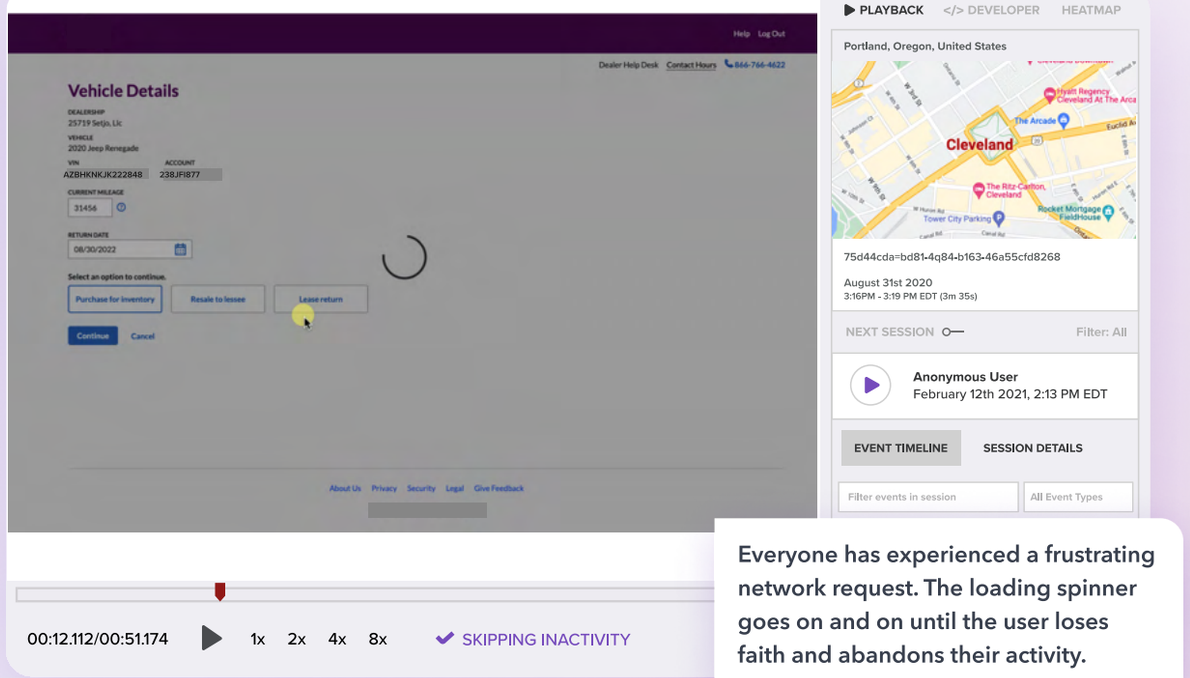
UP NEXT:
01: Frustrating network requests
↓

01

Frustrating network requests

Long or frustrating network requests can impact your app in many ways, from overloading the server to causing users to become impatient and consequently abandon their session.

This resource-intensive silent killer can be difficult to catch because, usually, the request does ultimately succeed while the user is still on the page; thus, it doesn't register as a failed network request in traditional monitoring tools. By the time the request completes, however, the user may no longer want or be able to complete their intended action.



Everyone has experienced a frustrating network request. The loading spinner goes on and on until the user loses faith and abandons their activity.

LogRocket

Typically, this type of slow, blocking network request is related to loading page elements rather than issues with navigation between pages.

If you break HTTP traffic down into time spent loading the HTML document versus page components, you'll often find loading the HTML document takes up less than 20 percent of the total response time. The rest of the time is spent waiting for the browser to download and render the app's components.

Since frustrating network requests typically end up succeeding (albeit slowly), not all long network requests are bad from the user's perspective. However, you need to know how to identify those that impact your users and hurt their experience on your app.

When an app's components load slowly, the user may have to actively wait on important information or interactive elements that take a long time to render.

A survey of 1,250 online shoppers by Digital.com found that half of all respondents expect a website to load in 3 seconds or less and will abandon their purchases if pages load too slowly.

While this expectation is not specific to page element rendering, it stands to reason that app users will generally respond in a similar way to frustrating network requests.

Consider a situation where a CTA button is placed midway through a landing page. If this button does not render in time as the user scrolls through the page, they may miss the opportunity to convert as intended. From the user's perspective, there was no opportunity to take further action at all.

Monitoring your average network request response time can give you some helpful insights into how the majority of your users are experiencing your app. Some more mature engineering organizations may even be monitoring 50th–99th percentile response times on APIs. But the context around the impact of those requests is still missing.

This disconnect between your app's backend and frontend can occur for myriad reasons — user location, poor network connection, device speed, or server-side logic implementation, to name a few. But to fully understand how these factors impact your users, you need to monitor the frontend to connect the dots between the issues they face and what they actually experience.

DEVELOPER TIP

To debug a frustrating network request, you need to understand both **why** this specific request is slow and **how** exactly your users are experiencing it. For example, if the response is taking a long time to download and there is no status indicator for your users, you could reduce the size of the response and add a progress bar with overlay messages.

UP NEXT:

02: Client-side performance problems

↓

A frustrating network request may not always have a significant effect on your user experience. In those cases, you may not have to dedicate time or resources to resolve this issue. But in other cases, timely request completion can be crucial for conversion.

If you can see evidence of a user's impatience during a particularly long network request, or identify where the user is on a page by the time the request succeeds, this may help you better determine exactly how a frustrating network request affects the way users interact with your app.

Start with your users and work backward to understand how this silent killer may be impacting your app's UX.

02

Client-side performance problems

Client-side application performance includes everything from server execution time to browser rendering on the client side. This provides plenty of opportunities for potential problems to occur.

Long threads could block the JavaScript event loop, main thread, or UI thread. High memory usage could cause the browser tab to crash. A page element may not load correctly or behave the way it's supposed to.

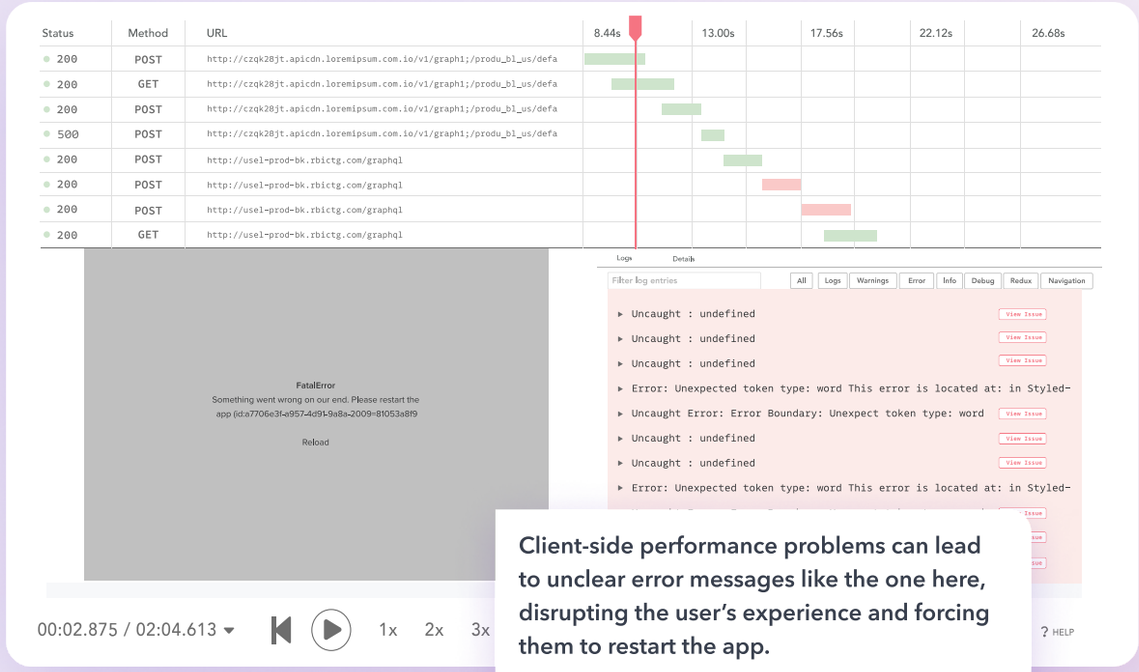
Many developers think primarily in terms of their control over and visibility into the code running in the backend rather than how performance problems may look on the client side. Just because something is running well when you test it on your device, however, doesn't mean the same is true for your users.

For example, if you are based on the east coast of the United States and testing an app you deployed to a nearby server, you're not experiencing your app the same way as your users in other parts of the country or the world. Geography, network strength, device type, and other factors outside the user's control can all affect your app's performance for different users in different ways.

DEVELOPER TIP

If your web app is using a significant amount of memory, you could risk crashing the browser process. You should also think about how you might be blocking the main thread and hurting your users in the process.

This turns client-side performance problems into silent killers that developers may not become aware of unless they notice things like a high drop-off in the checkout flow or important pages that are loading slower than average.



Client-side performance problems can lead to unclear error messages like the one here, disrupting the user's experience and forcing them to restart the app.

These signs may indicate that users are running into client-side performance problems while viewing parts of the app. Users experiencing these problems may abandon their session without ever triggering any actual error message or alerting your team to the problem.

Even if the user reports the issue or seeks support, reproducing the issue to determine what went wrong can be difficult without knowing or being able to simulate the user's location, device type, and numerous other variables.

Imagine you have an app serving an international customer base. In addition to the general factors affecting client-side performance, delivering digital content to multiple countries introduces variables such as different levels of internet access, volumes of traffic, and more.

It's easy to miss client-side performance problems if you're looking at traditional error reporting tools. There are so many particular issues that could result from the same general problems, and no one-size-fits-all solution for any of them.

Using these tools or testing performance manually can also be inefficient and give you results that don't accurately reflect real users' experiences. Typically, your results will reflect performance in a local environment instead, allowing client-side performance problems for real users in different countries to slip by undetected.

Monitoring the frontend could tell you when performance is getting worse but may not help you understand how or why. Trying to navigate through potential solutions without clear insight into who's being affected, what led them to particular issues, and where to focus your efforts can result in a lot of time-consuming guesswork.

Consider a real example. Delivery startup Rappi found that their Core Web Vitals were suffering as they expanded service into different countries. They turned to user session data to narrow down the root cause of their client-side performance problems and learned that the biggest impact was coming from a very high Time to First Byte (TTFB) on one lightweight page, particularly in two high-traffic countries.

Once they recognized that a small improvement in TTFB could optimize their user experience better than more time-consuming changes in other parts of the app, they were able to quickly make the necessary updates, see a significant reduction in their TTFB, and vastly improve the app experience for affected users.

If you start with real user sessions, you can get a clearer picture of the issue and its impact, including everything leading up to the problem, how it affects the experience people have with your app, and what you should do next.

UP NEXT:
03: Canceled and failed requests
↓

03

Canceled and failed requests

Canceled and failed requests are two types of issues that can have a similar negative effect on a user's experience.

When a network request is cut off at a CDN or blocked by an extension, the user may encounter an error message or some other outcome that negatively affects their experience with your app. Certain types of failed requests may also occur where the upstream server may be unhealthy or unreachable.

While you might think this type of issue would get picked up by an error reporting tool, it often ends up as a silent killer. Evidence of canceled or failed requests may not appear in backend monitoring tools or logs, or may otherwise be hard to find through QA testing or tools that monitor the frontend without capturing context around user behavior.

For example, depending on the level at which the request fails, it may be tricky for traditional monitoring tools to detect the error. Most server-side data collection agents run on the application server, at the very end of the line of communication between the user and the application server. If you have a CDN or some other service acting as a reverse proxy, possibly to provide some form of network security such as DDoS protection, you may run into unexpectedly canceled requests.

This may occur because the software incorrectly thinks it's protecting the application from a malicious actor, but the error would be effectively invisible to the data collection agent.

DEVELOPER TIP

Check to see if your CDN, DDoS protection solution, or other request-blocking systems are blocking requests they shouldn't be. Make sure connections between your web servers and upstream servers aren't severed or broken during deployments.

Even if your monitoring tool does catch canceled or failed requests, the context around how these types of errors really affect your users is once again missing. Tracing and reproducing such an error to debug it can also be a challenge.

Additionally, the way your users experience a canceled or failed request may not align with what you're seeing in the backend. Users may see an endlessly loading request rather than an error message when a server request is canceled.

In other cases, they may see an error message that doesn't align with what actually went wrong when a request fails.

Let's say you have an app that allows customers to purchase tickets for an event. Users should be able to view available tickets for a certain date, select the ticket or tickets they want, add or edit their information, view their ticket reservation information, check out, and so on.

Here are some problems that could occur related to canceled or failed network requests and how your user might experience them:

- Your user selects a date to view available tickets. The network request times out, but the user ultimately sees an error message saying there are no tickets available for that date. This could be due to a poorly handled 504 error, where a response was not received in time to complete the request.

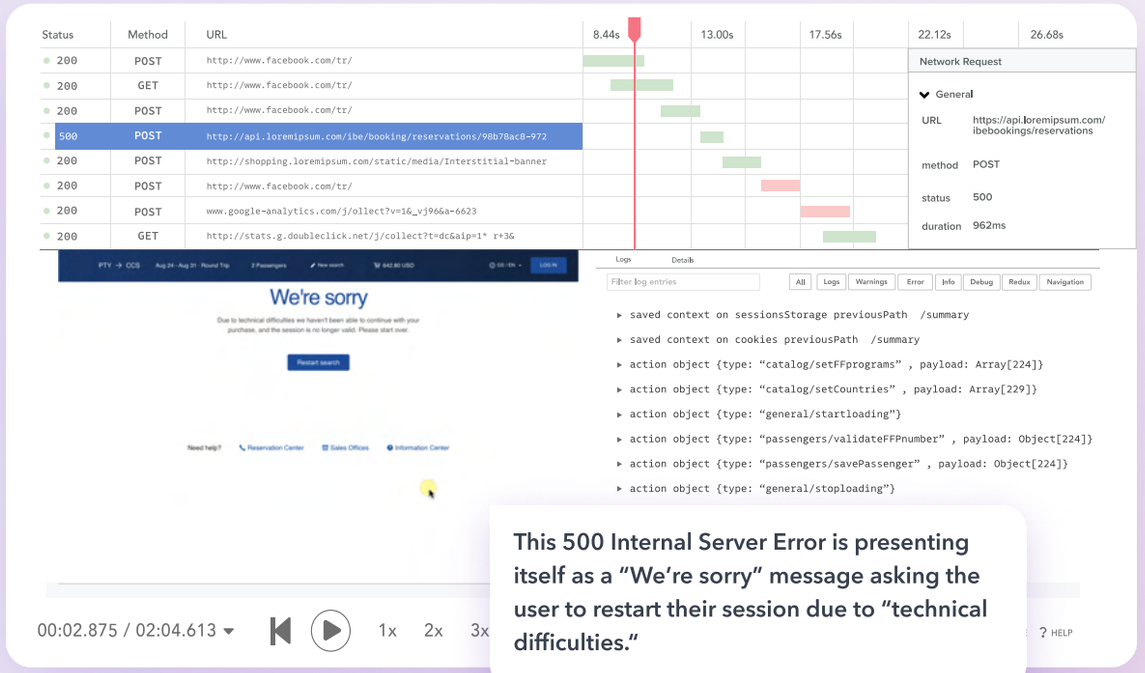
- Your user selects a ticket and proceeds to add or edit their information. However, after submitting their information, they see a message that tells them a technical error occurred. This could be due to a canceled network request.

- The ticket reservation information fails to load when the page loads. Your user sees a loading circle, progress bar, blank page, or an error message, and may not understand what went wrong or why. Potential causes for this problem are varied, but could include a canceled request, a 502 Bad Gateway Error, or a 503 Service Unavailable response code.

- Your user clicks the **Purchase Tickets** button and sees a loading indicator. The network request is canceled, but your user continues to see it loading endlessly, possibly because there was no method in place to handle the lack of response from the server.

Whether or not the user sees any error message, they also might not surface the problem to the developer team, or know how to describe the issue in a way that helps the team resolve it. Instead, they may abandon whatever they were doing entirely.

Ultimately, a canceled or failed request almost always hurts your user, though not always in the ways you might expect. However, the user may not report the issue, a traditional error reporting tool may not recognize it as a problem, and even once found, context around the user's experience of the issue may still be missing.



It's important to recognize trends in user sessions so you can get ahead of canceled and failed network requests, understand their impact, and ensure a smooth experience for your users.

UP NEXT:
Conclusion
↓

Conclusion

Companies engaging with users on digital platforms are now expected — even required — to deliver great user experiences. But as expectations keep rising and frontends continue to become more complex, delivering these experiences gets harder and harder.

New issues will always appear in any modern application, along with known issues that continue to resurface over time. Although monitoring the frontend of your apps can help you debug problems reactively, this alone isn't enough to ensure great user experiences. Silent killers like those we've discussed — errors that are difficult to identify or reproduce — can easily slip through the cracks.

Users are quick to abandon you for alternative products or services if their expectations are not met.

You need a way to cut through the complexity and noise to identify the most important issues that are preventing users from having smooth digital experiences.

Monitoring errors is an important aspect of tracking and refining application performance, but it doesn't tell the whole story. To drive exceptional user experiences for every user at scale, you need to look at how errors, as well as other factors, are affecting users in real life.

Sometimes, unexpected issues may prevent users from completing what they're trying to do. Other times, apparent errors won't actually affect your app's user experience. Look to real user session data to enable your team to pinpoint and prioritize fixing the issues that impact your users the most.

Learn about how LogRocket can help you more efficiently solve issues faced by your users to maximize performance and ensure outstanding digital experiences here.