BEST PRACTICE SERIES

Mobile Monitoring

DATADOG

BEST PRACTICE SERIES

Mobile Monitoring

Best practices for monitoring mobile app performance

In a crowded and competitive market, mobile app developers must offer continuous availability and a frictionless user experience to minimize churn. Monitoring and maintaining mobile apps presents unique challenges. Since mobile apps run on a wide range of devices, it can be difficult to get clear visibility into client-side performance. And, if users are experiencing an issue, while a web app patch can be automatically rolled out to customers instantly, shipping mobile app updates takes time and requires oversight from phone marketplaces and opt-in from users to download them. To ensure that your app consistently delivers a great experience to your users, it's paramount to continually monitor how your app performs as well as how your users are interacting with it and what sort of errors they are experiencing.

In this guide, we'll discuss key metrics and other KPIs that will help you understand the health and performance of your mobile app. Then we'll cover some best practices for getting the most out of your error data to ensure that your team can quickly find and patch bugs to minimize their impact on your customers.

Key metrics for understanding your app's health and performance

While users may be accustomed to waiting a moment after interacting with a web application for the response to be delivered, mobile users expect their apps to react instantly to their taps, swipes, and inputs. A myriad of performance issues can negatively impact your users' experiences and cause them to churn away from your application. Because of the vast array of available devices with wildly different specs and the inherent regional inconsistency of mobile networks, these issues can be hard to identify in pre-production. Therefore, it's important to understand when (and where) your users are experiencing hiccups—such as freezes, crashes, slow rendering of key UI components, application not responding (ANR) errors, and battery drain—in order to spot as quickly as possible issues that need addressing.

In this section, we'll discuss some key metrics that you can continually monitor to form a picture of your app's performance. By tracking these metrics over time and setting alerts on them, you can ensure your team remains informed if your app's performance falls short of SLOs:

- App start time
- Slow renders, frozen frames, and ANRs
- Network performance metrics
- Resource metrics, including CPU, disk, and memory usage
- Custom metrics from user actions

APP START TIME

The initial startup time of your app forms a key first impression that will influence the user's satisfaction and thus affect your app's store ratings and user retention. In some cases, like payment apps or mobile boarding passes, slow starts can even be a significant inconvenience. Faster startups lead to more sustained user interaction with your app and less churn. It's important to monitor your app's load performance during both **cold starts**, in which the app starts up without any of its processes already running, and **warm starts**, in which some app state is available from the previous session.

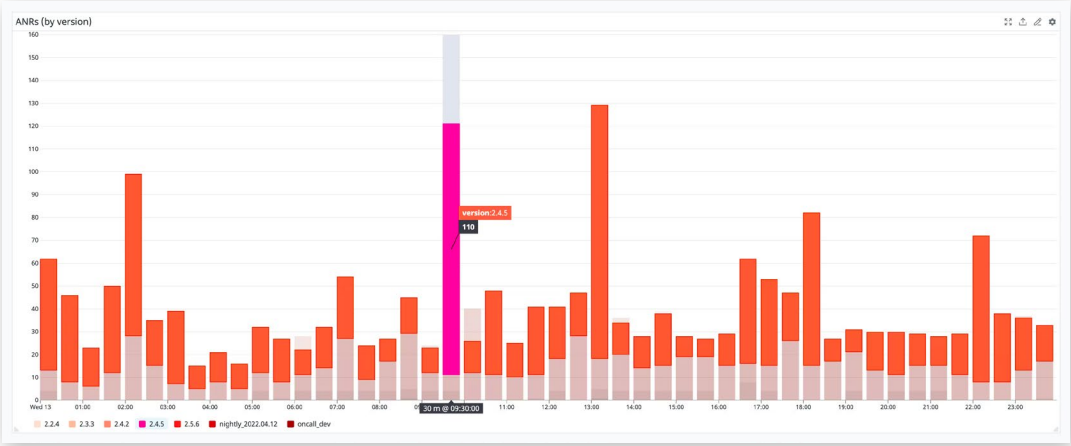
Cold starts are the main limiting factor in your app's load performance. In a cold start, all of your app's resources must be rendered. Apple suggests that cold starts should occur in under 400 ms, while Google suggests five seconds to accommodate the much broader range of Android device specifications. If you see long cold start times for your app, breaking down user data by factors like device type, app version, and location can help you spot issues that might be affecting your app's startup time. For example, if your app's cold start requires fetching a lot of data with network requests, regional network latency can dramatically affect your app's load time. You might also find that certain phones with more powerful CPUs and GPUs will have a much easier time drawing your app's initial view. If you see that many users are experiencing long cold start times, you can try to reduce the number of assets required to display your app's initial UI (e.g., by colorizing assets programmatically). Or, consider lazy-loading UI components that aren't needed for startup.

Warm starts occur with some of the application state already loaded in memory, meaning that they should be significantly quicker than cold starts. Android recommends a warm start time of under two seconds. The duration of warm starts can differ much more greatly than cold starts depending on the state of the device upon launch. Both Android and iOS dynamically kill background app processes depending on the memory needs of the foreground app. On Android, for example, a saved instance state bundle for the `onCreate()` method will sometimes persist while the app runs in the background, or sometimes the app must call the method again to recreate this state from scratch. By profiling your app using Android's and iOS's provided tools, you can begin to understand how various state conditions affect your app's warm starts.

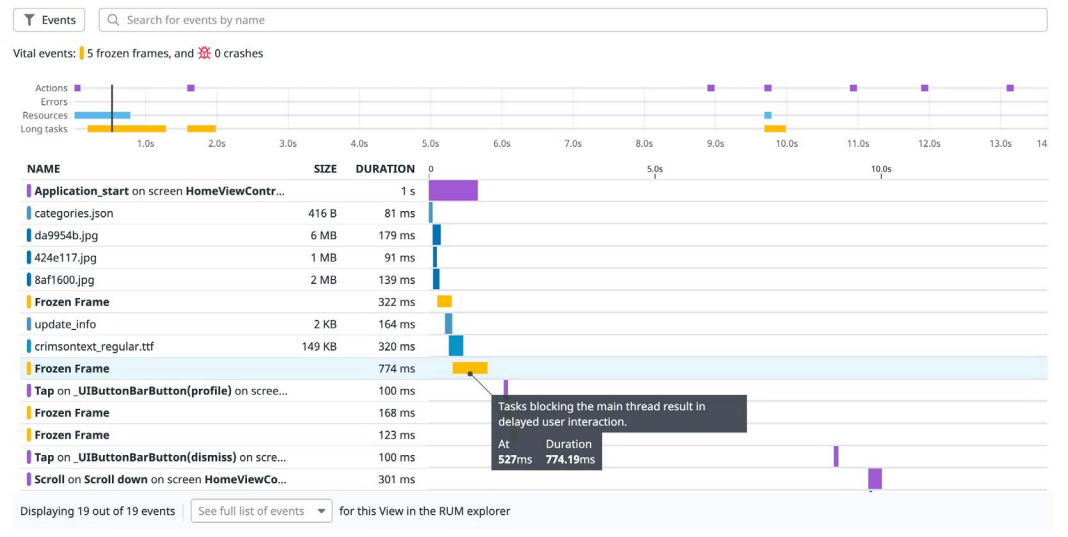
VIEW RENDERING TIME

Slow view rendering, scrolling lag, or delayed responses to touch-based actions will quickly erode your users' sense of immediate control and lead to frustration that may eventually cause them to churn. Both Apple and Google recommend that each frame in your app's runtime should take approximately 17 ms to render. For many reasons, such as code errors or stalled network requests, this frame rendering time can spike dramatically beyond the 17-ms benchmark.

Both Android and iOS provide ways of detecting excessively slow-rendering frames. Google defines a frozen frame as a UI frame that takes longer than 700 ms to render. If a frozen frame extends beyond five seconds, Android will throw an ANR error and a dialogue box will pop up, allowing the user to kill the app. According to Google, app developers should try to make their apps exhibit ANRs in less than 0.47 percent of daily sessions.



Apple tracks app hangs when the main thread is unresponsive for at least 250 ms. Rather than triggering an ANR for even longer freezes, iOS will simply crash your app. Thus, it's particularly important to make sure your iOS apps hang as little as possible.



By tracking the frame rate of views across your app and looking for frozen frames, ANRs, and app hangs, you can stay aware of which parts of your app might exhibit stuttery or slow behavior. Just like with cold starts, it's beneficial to break your frame rate metrics down by device and app version, either to spot compatibility issues or to determine if, for example, certain devices' graphics hardware is unable to efficiently render your app. Where possible, you should try to avoid running long calculations, slow I/O operations, and long network calls from the main thread of your app, which can be another common cause of frozen frames and ANRs. Using distributed tracing, you can trace requests as they propagate through your app and its infrastructure to spot slow network and other service calls that might be causing bottlenecks.

NETWORK PERFORMANCE

Virtually all mobile apps use some kind of backend service infrastructure to handle things like dynamic content fetches, recommendations, authentication, and storing user accounts. Network connectivity can vary wildly depending on where customers are located, meaning that network communication is a common source of degraded performance if your app can't reach its backend services.

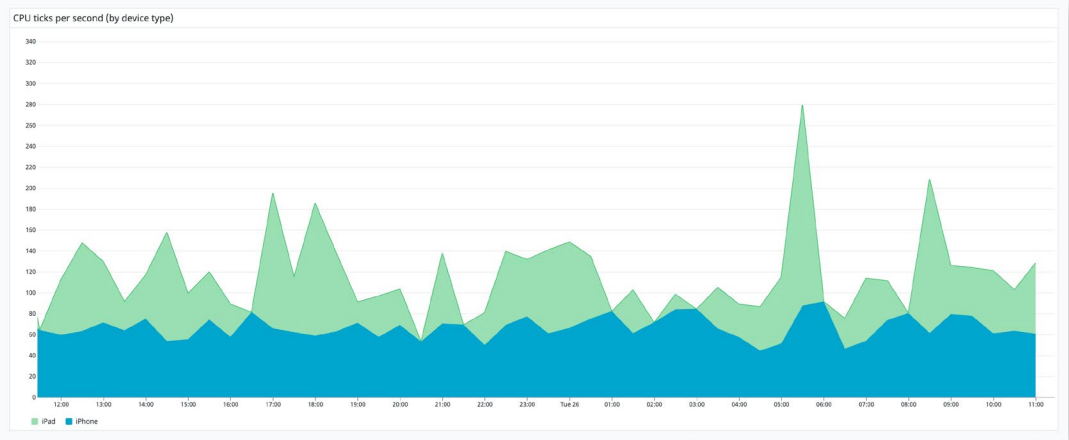
It's important to monitor network traffic not only from your app but also between all your backend services in order to spot outages that will impact your service availability. This can be accomplished by tracking the classic RED (requests, errors, duration) metrics for network requests made by your app and your backend servers. An increase in both average latency and timeout errors—especially if they are within a particular region—can indicate a network outage.

Additionally, you'll want to monitor the volume of requests being sent by your app. Apps constantly sending network requests in the background can lead to hardware resource overconsumption and battery drain, as well as overages on your customers' data bills. If your app is sending too many network requests, you might want to consider persisting more data on customers' devices or having customers manually refresh to load the newest content.

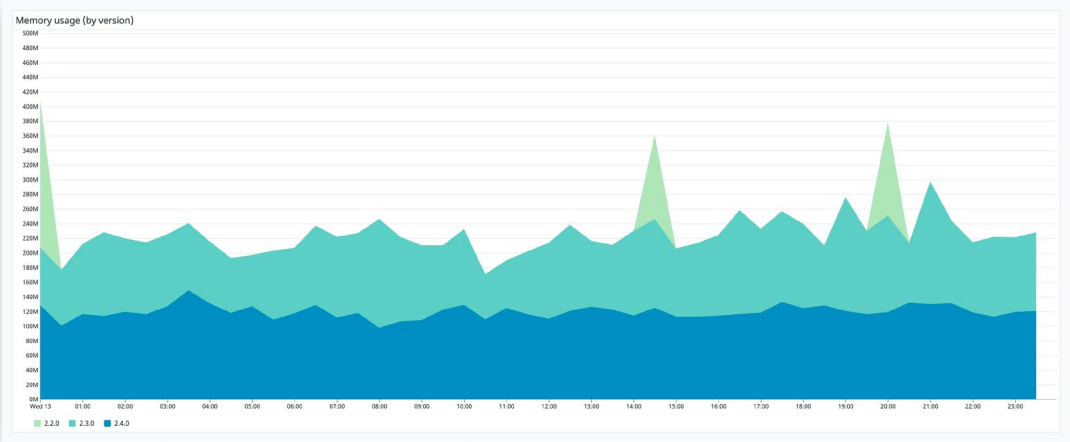
RESOURCE UTILIZATION

Whether your app is running on Android or iOS, your customers will be using a wide variety of devices with different hardware configurations. This means that understanding how resource intensive your app is can provide insight into its performance across devices. For example, by monitoring how much CPU and memory your application uses, you can spot issues that can cause overheating, battery drain, high disk usage, and, when your app exceeds the OS's available memory, out-of-memory errors.

One way to track the CPU usage of your app is by measuring the average number of CPU ticks per second for each view within a user session. This way, you can identify which views are consuming the most CPU and focus your optimization efforts accordingly.



By also tracking the average memory utilization of each view, you can then spot problem areas for your app's memory consumption that may be leading to out-of-memory errors or device delays. You can implement logic to check how much memory is available before running a process (Android's `getMemoryInfo()` or iOS's `applicationDidReceiveMemoryWarning(_:)`), which can help you preemptively mitigate out-of-memory errors.



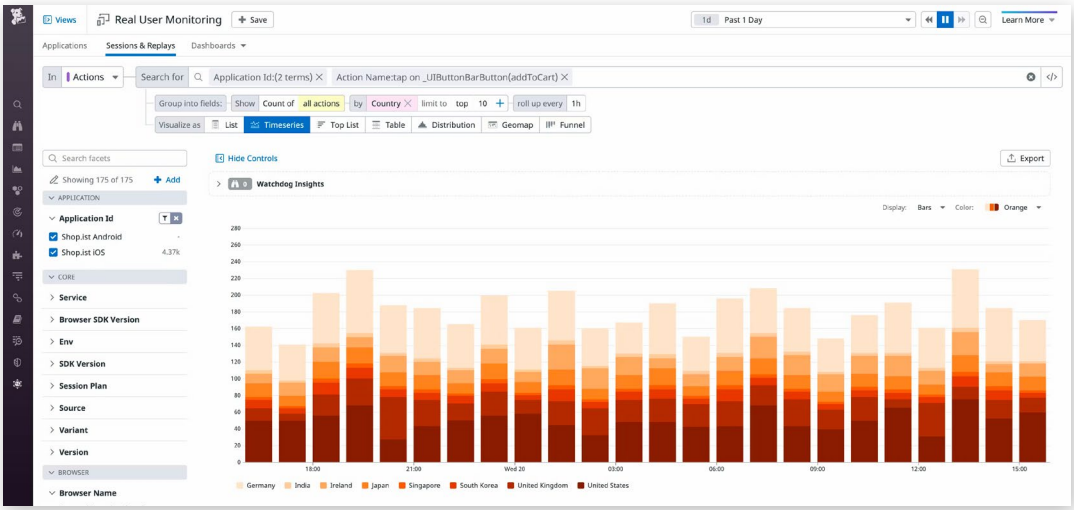
Finally, it's also important to monitor your app's disk usage; if your app is taking up too much hard drive space on your customers' devices, they may decide to delete it to save room for other content. Reads and writes on hard disk storage are significantly slower than reads and writes in RAM, so where possible, it's best to avoid storing dynamic app state on the hard disk. It's also particularly important to monitor hard disk writes from your app, as SSDs can only write data to the same region of the disk a finite number of times before it wears out. On iOS, for example, you can track your app's daily disk write volume in the Xcode Organizer, and then use the Filesystem Activity instrument in Apple's built-in profiler to figure out which parts of your code are contributing the most to your app's disk usage. Whenever possible, store files in a temporary cache in RAM to avoid excessive hard disk operations.

USER ACTIONS

In addition to monitoring how your app is performing for users, it's also important to understand how they are actually using your app. Getting a granular understanding of how users engage with your app will help you focus efforts better on performance and design optimizations to ensure a frictionless user experience. By capturing user actions data—such as taps, scrolls, inputs, and view transitions—you can begin to understand user behavior and observe which workflows might be causing users to drop off.

Collecting data, such as the number of users that complete your app's key workflows, help to quantify these workflows' success rate and answer questions like:

- Are users completing checkout after adding items to their shopping cart?
- Are users paying for in-app purchases, such as an ad-free toggle?
- Are users sharing links from your app via email and SMS?



Having this information makes it easier to determine if a particular workflow is causing more users to churn away due to a bug or poor design. This in turn helps prioritize your development work.

So far, we've looked at some key metrics and KPIs for tracking the health and performance of your mobile app. Next, we'll cover another key aspect of mobile app monitoring, which is understanding what sort of errors your users might be experiencing.

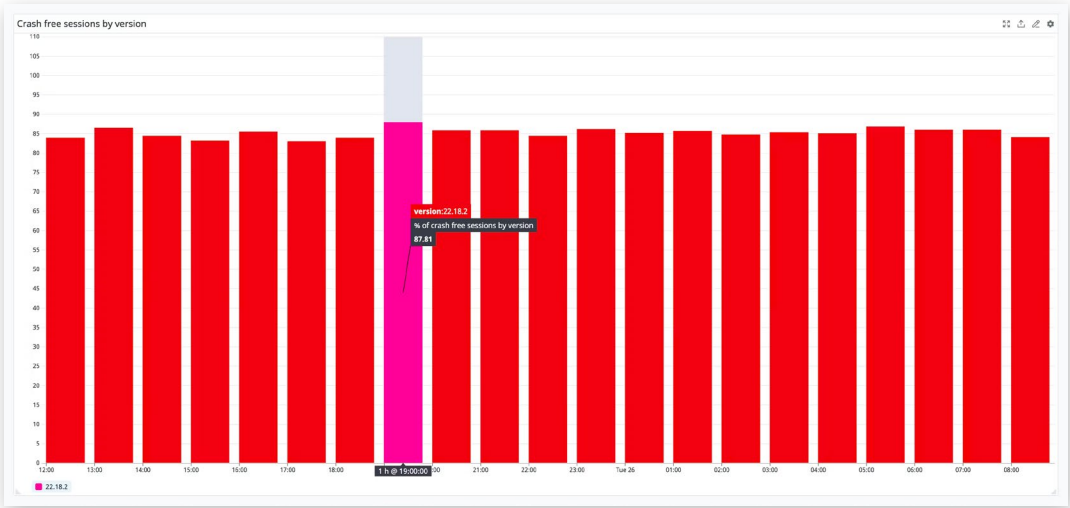
Track and diagnose errors from end to end

As we've discussed, a high volume of errors in your app—or any of its dependent backend services—can lead to lower service availability, ANRs, frozen frames, crashes, and more. Ultimately, these issues will significantly degrade your customers' experiences, leading to lower retention, churn, revenue loss, and reduced discoverability due to low app store ratings. Unlike with a web app, where you can almost instantaneously reroute clients to the newest version of your code, rolling out patches in the mobile space takes time and requires user cooperation. Therefore, it's important to collect as much error information from your users as you can in order to quickly identify new problems affecting your app's usability.

Next, we'll look at some ways to make sure you get the most out of error information so you can both understand the scope of any problems and more quickly identify the root cause.

COLLECT CRASH DATA

As you're collecting code errors from your customers' devices, it's particularly important to track crashes. Recent research shows that crashes cause roughly 20 percent of mobile application uninstalls. A good indicator to watch of crash frequency for your app is the number of crash-free sessions. You can break this data down by app version to figure out if a new patch or incumbent OS incompatibility is causing a fatal issue.



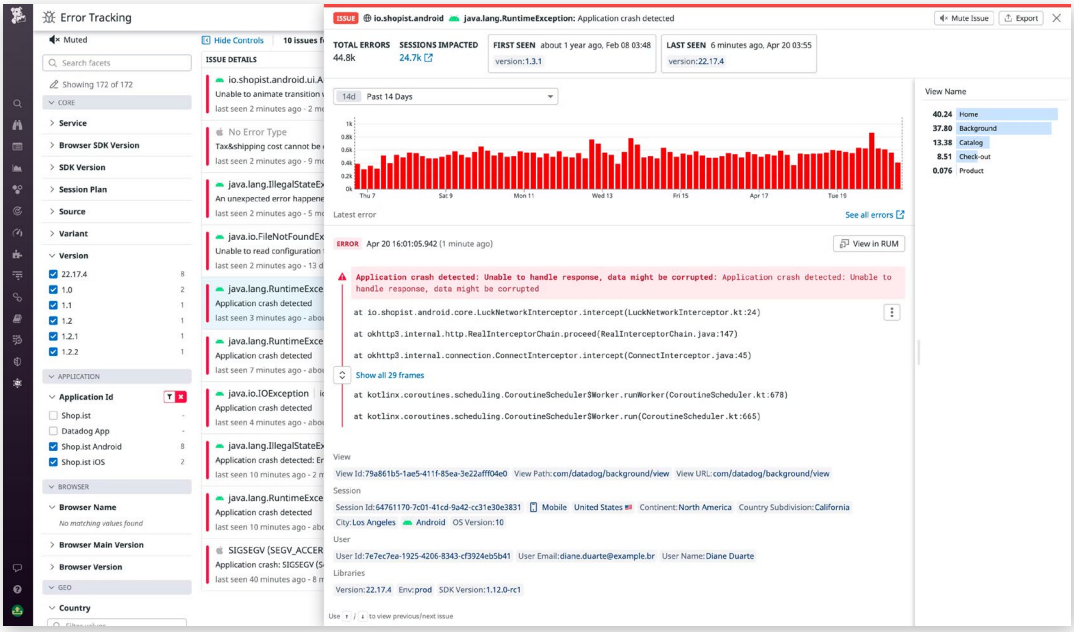
By collecting iOS and Android crash reports and related stack traces from your customers' devices, you can gather the context you need to diagnose crash patterns once you've spotted them. If your application is crashing more frequently than usual across multiple recent versions, for example, there may be a backend issue—such as an API endpoint sending response payloads in an improper format.

CONTEXTUALIZE ERRORS

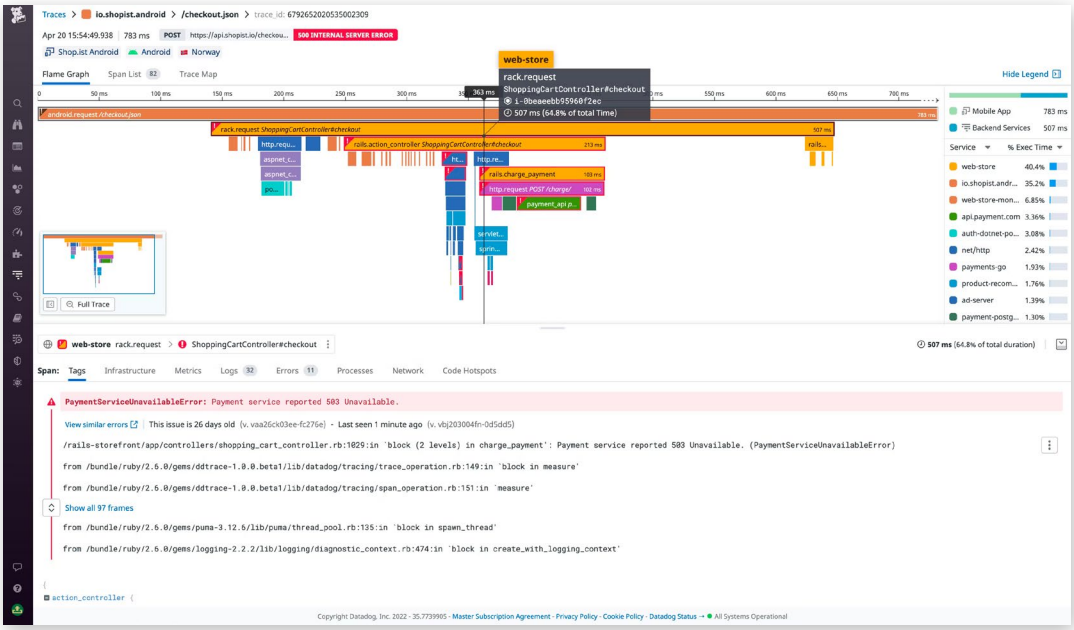
Knowing that crashes and other errors are occurring is important, but without the proper context, it can be difficult to understand how widespread the issue is, or to isolate the root cause. Collecting as much data as possible from user sessions that experience errors provides important insight that can help you reduce MTTR. Key data includes:

- User metadata (e.g., device type, location, app version) and actions (i.e., the steps a user took that resulted in the error)
- Related telemetry including application and infrastructure logs, symbolicated stack traces, and distributed request traces

Another key aspect of tracking errors is to monitor them in aggregate, which enables you to identify trends and see the scope of new issues. With a monitoring tool that includes error tracking, you can easily group crash reports and errors together and slice and dice your data using relevant metadata to determine, for example, whether a specific device type is experiencing crashes, or which views within your app are raising the most errors. By then diving into relevant traces, logs, and crash reports, you have more context to troubleshoot the root cause of the issue and determine where in your stack it lies.



For example, end-to-end request traces in particular allow you to follow requests all the way across your backend services to find errors and bottlenecks. This includes API calls to any third-party managed services your application depends on. This makes it easy to identify whether an error is caused by an issue somewhere else in your environment. By correlating traces with relevant logs, infrastructure metrics (e.g., CPU and memory usage), and network telemetry, you can identify if the root cause of an error is related to your code, underprovisioned hardware, or a network issue—anywhere in your infrastructure.



Keep your apps online and highly performant

In this post, we looked at how, by collecting detailed metrics from your app and its dependencies, gathering relevant context from your users, and effectively tracking errors across your infrastructure, you can stay on top of developing mobile application issues and easily identify where to focus your optimization efforts. With Datadog's comprehensive solutions for mobile real user monitoring, end-to-end tracing, and error tracking, you can leverage a unified platform to help keep your apps online and highly performant. For more information about mobile RUM with Datadog, see our iOS, Android, and React-Native documentation. Or, if you're brand new to the product, get started with a free trial.


DATADOG