



13 ways to improve your iOS shopping app's performance

Contents

Introduction.....	03
01 Memory leaks	04
02 Scroll views.....	09
03 Slow startup	13
04 Slow loading times.....	17
05 Networking.....	20
06 Low Power Mode.....	24
07 Third-party integrations.....	27
08 Discounts or sales.....	31
09 UI freezes or lag.....	34
10 User flows.....	37
11 Compatibility issues	42
12 Lack of error handling.....	46
13 Payment providers.....	50
Embrace helps you deliver amazing shopping experiences	53

Introduction



The rise of mobile has revolutionized the shopping experience, making it more accessible than ever. Brick-and-mortar stores transitioning to online platforms laid the foundation, and mobile devices further expanded every brand's reach, enabling connecting directly with customers wherever they are. The convenience and flexibility offered by mobile commerce have completely transformed the retail landscape.

The recent surge in mobile app users has propelled mobile e-commerce sales to extraordinary heights. In 2022 alone, these sales skyrocketed to a staggering [\\$415.93 billion](#), which amounts to a remarkable 6% share of overall retail sales. In 2023, projections continued upward, with retail m-commerce sales rising to [43.4% of the total retail e-commerce sales](#), surpassing the 41.8% achieved in 2022.

However, navigating the mobile landscape presents challenges for iOS app developers. Users have high expectations and swiftly uninstall apps that fail to meet their standards. Poor app performance quickly leads to user dissatisfaction and eventual app abandonment. Therefore, prioritizing app performance is crucial for delivering smooth and responsive mobile experiences that continually delight users.

For example, sustaining user engagement can be daunting. On average, an app loses a staggering [77% of its daily active users \(DAUs\)](#) within the first three days after installation, and uninstalls can cost retail and e-commerce apps as much as [\\$68,000 per month](#).

Even if users decide to stick around, app performance issues can still drive them away. Research indicates that [62% of people delete an app](#) if they encounter mobile crashes, freezes, or errors. Even seemingly minor issues like slow load times can result in up to [70% of mobile app users abandoning an app](#).

In addition, a mere one-second delay in response time can result in a [7% loss in conversion rate](#).

Gaining and keeping users in the mobile app landscape has never been more challenging. That's why it's vital to aggressively address app performance issues to retain even the most loyal users.

By adhering to best practices, you can ensure that your app is resource-efficient, resilient to failures, and equipped with the necessary tools to swiftly resolve any issues that arise.

In this eBook, we share several ways (13, to be precise) to uncover mobile app performance gaps that can cause slowdowns, freezes, crashes, and other types of negative user experiences in your iOS shopping apps. We'll also cover actionable steps you can take today to prevent them from happening in order to build better mobile experiences.

Let's dive into the 13 common app performance issues that occur in iOS shopping apps.



01 Memory leaks

Shopping apps tend to use a lot of memory because they frequently load many images and videos. Balancing the need to showcase products with rich visuals while remaining within iOS memory limits is essential for delivering smooth user experiences.

Memory leaks can occur when resources are not properly released, leading to increased memory usage. This can result in app slowdowns or crashes.

01

While automatic reference counting (ARC) on iOS eliminates most of the monotony of manual memory management, there are several ways that costly memory leaks can surface in your iOS apps. Let's go over a few of them now.

Technical causes

Unused memory

This can happen due to coding errors, conditional logic, or overlooked scenarios. For example, if a programmer allocates memory for a variable but forgets to assign a value to it or if a certain branch of code that should release allocated memory is never executed, it can lead to unused memory and memory leaks.

Caching issues

Improper cache management can result in memory leaks. You must remove or replace cached objects and data when they are no longer needed. Otherwise, they will continue to occupy memory indefinitely. This can occur due to programming errors, such as failing to update the cache when data changes or not implementing a mechanism to invalidate outdated cache entries. Instead of using a dictionary to hold onto cached items, use Apple's provided `NSCache` for a good in-memory cache solution.

External resources

Interacting with external resources like databases, files, or network connections can also contribute to memory leaks if they are not properly released or closed. An example is opening a file or network connection but failing to close it after use. In that case, the associated memory resources may not be released. Similarly, if database connections are not properly closed or released, it can result in memory leaks over time. Lack of comparative data

Programming language-specific issues

Different programming languages have their own mechanisms that can contribute to memory leaks. Languages like C or C++ require manual memory management, and if deallocation is not performed correctly using functions like `free()` or `delete`, it can result in memory leaks. On the other hand, languages like Java or C# use automatic memory management through garbage collection, which can also have its challenges. For example, accidentally creating cyclic references between objects – where objects reference each other to prevent them from being garbage collected – can lead to memory leaks.

01

Best practices

Track memory usage

Many platforms provide tools and utilities specifically designed for monitoring memory usage. For example, tools such as [Instruments](#), [Heap](#), and [Leaks](#) can be used to detect memory leaks and analyze memory patterns.

Handle cleanup operations gracefully

Always remember to deallocate resources, save data, release memory, and close connections/files at critical points. When an application is about to terminate, it is recommended to save essential state information as soon as possible to ensure that important data is not lost. You can accomplish this by implementing methods like [applicationWillTerminate](#) in the [UIApplicationDelegate](#) protocol on iOS.

Handle strong and weak references properly

Properly managing strong and weak references is crucial for avoiding memory leaks and maintaining optimal memory usage. Use strong references only when necessary for object lifecycle management. Strong references create a retain cycle where objects reference each other, preventing them from being deallocated even when they are no longer needed. In scenarios where you want to avoid memory leaks, such as delegate patterns, use weak references. Weak references do not increase the reference count, allowing objects to be deallocated when they are no longer referenced strongly. Using weak references appropriately can break retain cycles and ensure that memory is efficiently managed.

By following the recommended practices for memory allocation and deallocation, you can minimize the occurrence of memory leaks in your code. Let's consider an example involving a [NetworkRequest](#) class that performs network requests in the code snippet below.

01

By following the recommended practices for memory allocation and deallocation, you can minimize the occurrence of memory leaks in your code. Let's consider an example involving a `NetworkRequest` class that performs network requests in the code snippet below.

```
protocol NetworkRequestDelegate: AnyObject {
    func didCompleteRequest(data: Data)
}

class NetworkRequest {
    weak var delegate: NetworkRequestDelegate?

    func performRequest() {
        // Simulating network request completion
        if let responseData = "Response Data".data(using: .utf8) {
            delegate?.didCompleteRequest(data: responseData)
        }
    }
}

class ViewController: NetworkRequestDelegate {
    let networkRequest = NetworkRequest()

    func startRequest() {
        networkRequest.delegate = self
        networkRequest.performRequest()
    }

    func didCompleteRequest(data: Data) {
        let responseString = String(data: data, encoding: .utf8)
        print("Received response data: \(responseString ?? "")")
    }
}

// Sample usage
let viewController = ViewController()
viewController.startRequest()
```

In this class, there is a weak reference `delegate` property of type `NetworkRequestDelegate`. By using a `Weak` reference for the delegate, the `NetworkRequest` instance does not keep a strong reference to the delegate. This ensures that the delegate, typically a `ViewController`, can be deallocated when no longer needed, preventing a strong reference cycle and potential memory leaks.

Steps you can take today

Improve automated memory leak testing with Xcode (for Swift)

[Automated memory leak testing](#) is an essential practice in Swift-based projects. It involves creating targeted tests that automatically detect and terminate instances that could potentially cause memory leaks. [Xcode](#) provides built-in support for memory leak testing through [XCTest](#), the testing framework for Swift and Objective-C. By writing specific test cases that exercise the code paths where memory leaks are likely to occur, you can enhance the efficiency and effectiveness of memory leak detection.

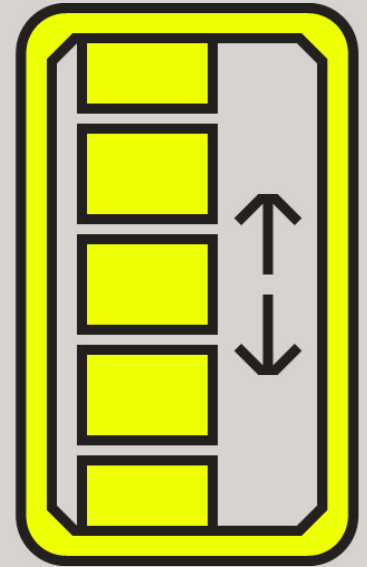


Use available memory management tools

1. **Heap analysis:** The heap command-line tool in Xcode allows you to inspect your application's memory allocation and deallocation patterns. By analyzing heap snapshots, you can identify memory blocks not being properly released or deallocated, indicating potential memory leaks or inefficient memory usage.
2. **Leaks detection:** The leaks command-line tool in Xcode is specifically designed to detect and analyze memory leaks automatically. It scans your application's memory and identifies objects still in memory despite being no longer reachable. This tool helps you pinpoint memory leaks accurately and provides detailed information about the leaked objects.
3. **Instruments:** Xcode's Instruments tool offers a comprehensive set of features for memory analysis and allows you to monitor memory allocations and deallocations in real-time, helping you identify areas of excessive memory usage. The VM Tracker instrument tracks virtual memory usage, providing insights into memory pressure and potential leaks. Additionally, the Zombies instrument helps identify instances where objects are accessed after deallocation, which often cause crashes or unpredictable behavior.
4. **Memory graph debugger:** This powerful tool within Xcode allows you to visualize the object graph and understand the relationships between objects in memory. It helps you track object ownership and identify strong reference or retention cycles that can lead to memory leaks.

Custom diagnostic configuration

Xcode allows you to set up custom diagnostic configurations within your development scheme to enable or disable specific memory management options. For example, you can configure your scheme to include "malloc stack logging" during development. This feature records the stack trace associated with each memory allocation, helping you track down memory-smashing bugs, heap corruption, and other memory-related issues.



02 Scroll views

Shopping apps rely heavily on scroll views to create a seamless browsing experience for users, allowing them to explore extensive product lists effortlessly. To captivate users, these apps must excel in swiftly loading new items, presenting visually appealing images and engaging carousels, and offering a seamless path toward purchasing.

02

Let's go over a few challenges that iOS engineers face when working with scroll views.

Technical causes

Incorrect layout or constraints

Scroll view issues can arise if the layout or constraints applied to the scroll view are incorrect. For example, if the content size is not defined properly or the constraints are misconfigured, it can lead to unexpected scrolling behavior or content clipping. This can cause elements within the scroll view to be incorrectly positioned or extend beyond the visible bounds, resulting in a suboptimal user experience.

Moreover, problems can occur when the content size of a scroll view is miscalculated or not updated correctly based on the actual content. This can result in issues like incomplete scrolling, where users cannot reach the entire content within the scroll view. Suppose the content size does not accurately reflect the content's dimensions. In that case, it can lead to inconsistencies and frustration for users navigating the scroll view.

Memory leaks

Scroll views are often used to display new content dynamically; however, when not managed properly, they can contribute to memory leaks. Each time new content is loaded into the scroll view, it may create additional resources in memory that are not properly released. Over time, this can lead to increased memory usage, causing performance degradation and lag during scrolling.

Interaction conflicts

Scroll views can sometimes conflict with other interactive elements within the application. This can result in unresponsive scrolling or unintended actions triggered during scrolling. Conflicts can arise when touch gestures or event handling are overlapped or improperly handled.

Performance limitations

The performance of a scroll view can be affected if the content it displays is too large or complex. This can exceed the capabilities of the device or the application, leading to sluggish scrolling, rendering delays, or even freezing.

Inefficient rendering or redraw

Inefficient rendering or redraw operations during scrolling can result in choppy or slow scrolling performance. This can occur if the scroll view's content is not properly cached, causing excessive redraw operations, or if the rendering process is not optimized for smooth scrolling.

Best practices

Reduce heavy resources

It's important to reduce heavy resources within scroll views. This includes minimizing the number and complexity of animations, optimizing image sizes and formats to reduce file sizes, and avoiding heavy scripts or computations that may cause performance bottlenecks. Adopting efficient coding practices and prioritizing lightweight resources can significantly improve the scrolling experience. Additionally, implementing lazy loading techniques – where content is loaded only when it is about to become visible on the screen – can enhance scrolling performance and responsiveness, particularly for scroll views with large or image-heavy content.

Maintain consistent scrolling behavior across different app sections

Standardized scrolling libraries or frameworks that provide consistent scrolling interactions and animations throughout the application are beneficial. This ensures that users are familiar with the scrolling behavior and can easily navigate different app sections. Maintaining a consistent scrolling experience makes users less likely to encounter unexpected or jarring scrolling behaviors, resulting in a more cohesive and enjoyable app experience. Lightweight resources can significantly improve the scrolling experience. Additionally, implementing lazy loading techniques – where content is loaded only when it is about to become visible on the screen – can enhance scrolling performance and responsiveness, particularly for scroll views with large or image-heavy content.

Provide visual feedback

Implementing visual elements such as loading spinners, progress bars, or placeholders can inform users about ongoing loading processes. These visual cues reassure users that new content is being loaded seamlessly, keeping them engaged with the app. By providing clear and visually engaging feedback, users are more likely to perceive the scrolling experience as smooth and responsive, even when there are delays in loading new content.

Steps you can take today

Performance testing with XCUITesting

Take advantage of Xcode's built-in testing framework, [XCUITesting](#), to automate performance testing for scroll views. Using the [XCTOSSignpostMetric](#), which measures frame rate, frame count, hitches count, and duration, you can create performance tests to identify issues like scrolling hitches, frame drops, and other performance-related problems. By conducting regular performance tests, you can proactively detect and address performance issues early in the development process, ensuring a smooth scrolling experience for users.

Use collection source data

Consider using more performant collections available in iOS, such as [UICollectionView](#) or [UITableView](#), depending on your specific use case. These collection views are optimized for efficiently displaying large amounts of data, with features like cell reusing and efficient layout rendering. You can improve your app's scrolling performance and responsiveness by leveraging the appropriate collection view for your needs.

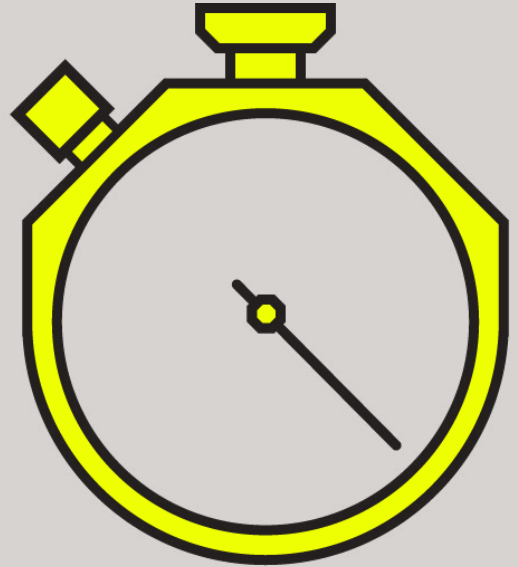
Leverage optimal UI components

Choose the right UI components to optimize the performance of your scroll views. For example, consider using lazy scroll views, which load content on demand as the user scrolls, rather than loading all content upfront. This approach reduces the initial load time and memory footprint, resulting in smoother scrolling experiences.



Take advantage of iOS Simulator tools to reduce GPU work

1. **Color Blended Layers:** Identify areas where layers are blended, indicating the potential for excessive GPU work. Blending layers can be computationally expensive, especially when multiple layers overlap or have transparency. By [optimizing the layer hierarchy or reducing unnecessary blending](#), you can reduce the GPU workload and improve the scrolling performance.
2. **Color Misaligned Images:** Detect misaligned images, which can cause additional GPU work during rendering. Misaligned images may result from incorrect scaling or positioning, leading to suboptimal rendering performance. [Ensuring images are properly aligned and scaled](#) to match the display requirements can reduce GPU workload and enhance efficiency.
3. **Color Copied Images:** Identify areas where images are needlessly copied, leading to unnecessary GPU work. By [optimizing image handling and reducing unnecessary image copying](#), you can decrease GPU load and improve scrolling performance in your app.
4. **Color Off-Screen Rendered:** Identify areas where content is being rendered off-screen, which can consume additional GPU resources. Rendering off-screen content can be costly, especially if it's not visible within the scroll view. By [minimizing off-screen rendering](#) and ensuring that content is only rendered when it becomes visible, you can reduce GPU workload and improve the efficiency of scroll view rendering.



03 Slow startup

When users are forced to endure a sluggish startup experience, their patience wears thin, and their confidence in the application diminishes. This can lead to a negative perception of the overall product, reduced user engagement, and decreased conversion rates.

03

Slow startup can occur [for many reasons](#), so let's go over a few of them now.

Technical causes

Lack of app caching

If the app fails to cache frequently accessed data, it may require repeated network requests or resource loading, leading to longer startup times. Caching involves storing data locally on the device, allowing faster access and reducing reliance on network requests.

UI rendering and layout

If the UI components contain excessive views or require extensive layout operations, it can significantly slow down the rendering process, leading to longer startup times. This can occur when the app loads complex UI structures or applies intricate Auto Layout constraints.

Best practices

Prioritize essential data and functionality

To improve startup performance, it is crucial to prioritize the loading and rendering of essential data and functionality. Identify the core components necessary for the initial app experience, such as product listings, search functionality, or user authentication. Users can then access key features and begin interacting with the app immediately, while other non-critical elements are still loading in the background.

Implement asynchronous loading and progressive rendering

Use [asynchronous loading techniques](#) to fetch data and resources in the background while displaying partial or placeholder content to the user. Asynchronous loading allows the app to render and respond to user interactions while continuing to fetch additional data. Users thus have a responsive and interactive experience throughout the startup process.

Test on realistic network conditions and devices

Testing on different network conditions, such as 3G, 4G, or Wi-Fi, helps assess how the app performs in different bandwidth environments. Similarly, testing different device models with varying processing power and memory capacity provides insights into how the app handles startup on different devices. By testing under realistic conditions, developers can identify and address performance bottlenecks, optimize loading times, and ensure a smooth startup experience for users across a range of network and device environments.

Steps you can take today

Reduce the number of software development kits (SDKs)

Loading fewer SDKs can significantly improve app performance during startup. Each SDK introduces additional network calls and consumes device resources, which can slow down the app's initialization process.



4 tips for evaluating SDK necessity

1. **Analyze SDK performance:** Assess each SDK's initialization time and network call performance. If an SDK significantly impacts startup time or introduces excessive network calls, it may be worth reconsidering its usage.
2. **Opt for lightweight alternatives:** If an SDK contains functionality not crucial for the app's core features, consider exploring lightweight alternatives. Point solutions or smaller libraries that offer the required functionality without unnecessary features can reduce the memory footprint and resource consumption during startup.
3. **Combine functionality:** If your app relies on multiple SDKs to retrieve data from different sources, look for opportunities to consolidate functionality. Instead of using several SDKs for different data sources, consider finding a solution that combines or integrates multiple data sources into a single SDK.
4. **Regularly review and update:** [Continuously review the necessity of each SDK](#) in your app. SDKs may have been added for experiments or for features that have since been deprecated. Regularly testing and evaluating the impact of each SDK on startup performance ensures that you are only using the absolutely necessary SDKs.

03

Use static dependencies whenever possible

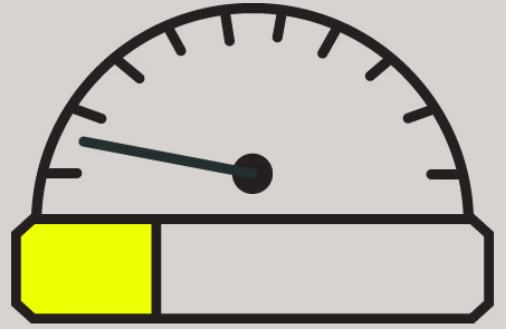
Using static dependencies instead of dynamic dependencies offers several advantages for startup performance. Static dependencies are linked directly into the app binary, reducing dynamic linking overhead during startup. This leads to faster app initialization, improved reliability, and easier deployment.

Use mergeable libraries

Apple introduced the new concept of [mergeable libraries](#) at this year's WWDC in June. By incorporating mergeable dynamic libraries, you can achieve app launch times comparable to static linking in release builds while retaining the advantage of dynamically linked build times in debug builds. This innovation optimizes app performance across different stages of development.

Strive for a concise `didFinishLaunchingWithOptions` method

Keeping the [didFinishLaunchingWithOptions](#) method as concise as possible enhances code readability, maintainability, and startup performance. You can optimize startup performance by modularizing initialization logic and delegating responsibilities to specific methods or components. This approach allows the app to focus on essential setup tasks and adhere to the single responsibility principle, leading to a cleaner and more efficient startup process.



04

Slow loading times

Extensive research shows that slow loading times result in reduced conversion rates. After all, the longer you keep users waiting, the more likely they are to abandon your shopping experience.

04

Note that we're breaking out slow startup and slow loading times into separate sections because optimizing load times within your app is a completely different set of challenges than the ones during app startup. Let us delve deeper into slow loading times now.

Technical causes

Slow third-party integrations

Third-party services, such as payment gateways or inventory management systems, can introduce delays in app loading times. If the response times from these integrated services are slow, they can significantly impact the overall app performance.

Network latency or connectivity issues

Slow internet connections, network congestion, or intermittent connectivity problems can be key factors contributing to slow loading times. When the app relies on network requests to fetch data or content, any issues with network latency or connectivity can result in freezes, blank screens, or long spinners.

Sampling limitations

Data-sampling techniques leveraged by popular tools also introduce the risk of missing instances of failed ad requests. Since these failures can happen sporadically or in specific scenarios, relying solely on sampled data may result in underrepresentation or incomplete detection of the problem. This limitation hampers the ability to fully understand the extent and impact of failed ad requests on the overall ad monetization performance.

Bloated or inefficient code

Poorly optimized code can lead to slower execution times and overall sluggishness. Excessive complexity, redundant calculations, or inefficient algorithms can impact the app's performance during loading.

Unoptimized content

Inefficient content delivery can contribute to slow loading times, especially for frequently accessed content. Missing or ineffective caching strategies can result in redundant data fetches and slower loading speeds. High-resolution images, in particular, can significantly impact loading times, especially on mobile devices or slower internet connections.

Best practices

Optimize loading throughout the app

Reduce the number of network requests by optimizing API calls to provide efficient and compact data payloads. Employ lazy loading techniques to fetch and load content only when needed, which reduces initial loading times. By offloading non-essential content loading to background threads with appropriate [Quality of Service \(QoS\)](#), the main thread can focus on rendering the UI and ensuring a responsive user experience.

Measure dropoff in key user flows

By monitoring user behavior and drop-off points, you can identify bottlenecks and areas of improvement. Optimize slow-loading screens by minimizing data fetching, optimizing database queries, and streamlining the flow to prioritize critical content loading. Address loading delays in key user flows to enhance the overall user experience and prevent users from abandoning the app.

Prewarm web views

[Web views](#), commonly used to display web content within the app, can experience delays during initial loading. Prewarming web views in the background by loading and caching web content before it is needed can significantly reduce perceived loading time. This technique ensures that content is available when requested, resulting in a smoother and faster experience. In addition, provide essential resources (such as fonts) from the app to minimize the need for external requests when launching a web view.

Steps you can take today

Ensure a responsive UI by keeping the main thread free

Heavy processing tasks, such as complex calculations or intensive data fetching, should be offloaded to background threads or queues. By distributing the workload across multiple threads, you prevent the main thread from being overloaded.

Enhance perceived performance with UI skeletons and spinners

UI skeletons, also known as placeholders, simulate the structure and layout of the expected content before it is fully loaded. Users get a visual indication that content is being loaded, which reduces the perceived waiting time. In addition, spinners and progress indicators provide real-time feedback, assuring users that the app is actively loading data as opposed to experiencing an error.

Improve web view performance with prewarming

[Prewarming WKWebView](#) instances can significantly boost web view loading times. WKWebView performs various setup operations during initialization, including loading frameworks and resources. By preallocating and preloading web views in advance, you can minimize the time spent on iOS initialization and resource loading when the web view is first accessed. This optimization reduces the latency associated with initializing web-related processes, resulting in faster web view loading and improved user experience.



05 Networking

In mobile, users are never guaranteed to have a consistent network connection. Therefore, it becomes crucial to deeply understand real-world network conditions and take measures to safeguard the user experience accordingly.

Networking issues can manifest in several ways within your iOS apps, impacting key functionalities like product information loading, checkout processes, and payment transactions. Let's delve into a few of these causes now.

Technical causes

Unstable internet connection

Users with unreliable internet connections may encounter slow loading times, intermittent connectivity, or even complete loss of connection while using iOS shopping apps. Factors such as weak signal strength, network interference, or remote locations with limited network coverage can contribute to an unstable Internet connection. These issues can disrupt the smooth data flow and impact the user experience.

Network congestion

Increased network traffic during peak usage hours or in densely populated areas can result in slower data transfer speeds. The high volume of data being transmitted and received can strain the network infrastructure, causing performance degradation and longer loading times.

API integration issues

Shopping apps often rely on various APIs to provide functionalities such as payment gateways, shipping providers, or inventory management systems. Issues with API integrations can arise, including authentication errors, rate limiting, or API downtime. These issues can disrupt network communication between the app and the corresponding services, leading to incomplete transactions, failed payment processes, or inaccurate inventory information.

Mobile network limitations

Networking issues on mobile devices can stem from limitations inherent to mobile networks. Weak signal strength in certain areas, especially in remote or crowded locations, can result in slower network speeds and connectivity issues. Mobile carriers may also impose restrictions or prioritize certain network traffic, affecting the performance of iOS shopping apps when accessing data or making network requests.

Network and infrastructure variations

Shopping apps interact with diverse network environments and infrastructures, which can introduce compatibility issues. Differences in Internet Service Providers (ISPs), firewalls, proxies, or Content Delivery Networks (CDNs) may impact the app's network protocols, security configurations, or optimizations. Incompatible setups or restrictions can lead to connectivity problems, data transmission errors, or slower loading times.

Best practices

Enable offline functionalities

By caching relevant data locally on the user's device, such as product information, pricing, images, and descriptions, users can continue browsing and accessing product details even when offline. In addition, specific features that do not require real-time data or transactions, such as viewing the wishlist or order history, should be accessible offline. When the device reconnects to the Internet, the app should automatically sync any user's offline changes, ensuring the data remains up-to-date and consistent across devices.

Test devices and operating systems

To ensure optimal performance and compatibility, iOS shopping apps should be thoroughly tested on different devices, screen sizes, and operating systems. Testing on various devices, including iPhones and iPads, allows developers to identify device-specific issues and optimize the app's design and functionality accordingly.

Use HTTP cache tools

Implementing [HTTP cache mechanisms](#) can significantly improve performance and reduce network dependency in iOS shopping apps. Techniques such as using [ETags](#) and [cache control headers](#) can optimize caching of server responses, reducing the need for frequent network requests. By efficiently caching static data, such as product images, the app can load content quickly and reduce the reliance on network connectivity. This improves the overall app performance and enhances the user experience, especially in scenarios with limited or unstable network connections.

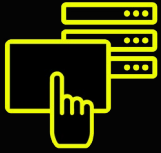
Steps you can take today

Optimize network efficiency with backend collaboration and frontend patterns

Collaborating with the backend team to consolidate API calls can lead to significant improvements in network efficiency. Implementing frontend API gateways or backend-for-frontend (BFF) patterns can help streamline network communication, optimize data transfers, and enhance overall performance and responsiveness.

Optimize backend performance with HTTP/2 support

Ensuring that your backend supports [HTTP/2 protocol](#) is essential for optimizing performance and minimizing resource consumption. HTTP/2 brings several benefits, including multiplexing and improved header compression, which enable the simultaneous transmission of multiple requests and responses over a single connection. This reduces latency and creates more efficient data transmission, resulting in faster loading times and improved network performance.

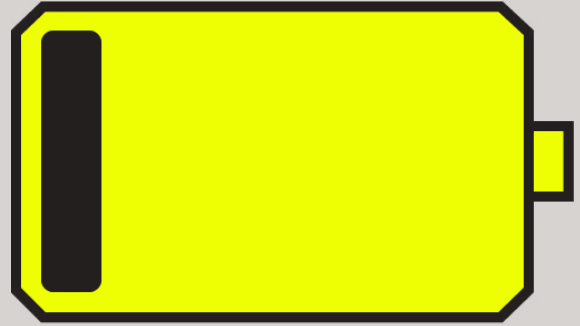


6 benefits of using a server-driven UI

1. **Reduced client-side processing:** The server delivers the pre-rendered UI to the client. This reduces client-side processing, especially for complex or dynamic UIs, resulting in faster rendering times on the client.
2. **Optimized data transfers:** By transferring only the necessary UI components and data from the server to the client, you minimize the amount of data that needs to be transmitted. This can significantly reduce loading times.
3. **Efficient updates:** You can enable efficient updates by allowing the server to dynamically generate and deliver updated UI components to the client as needed. This eliminates the need for the entire page to be reloaded, reducing the time and resources required for UI updates.
4. **Adaptive content delivery:** You can adapt the content and UI components based on network conditions, device capabilities, or user preferences. This ensures users have a tailored experience that minimizes slow loading times.
5. **Better user experience:** With server-driven UI, users can start interacting with the app faster, even if some components or data take longer to load, resulting in a smoother and more responsive user interface.
6. **Easier performance optimization:** You can make performance optimizations like caching, compression, or offloading processing to more powerful servers. These optimizations can help mitigate the effects of slow loading times and improve the overall performance of the UI.

Use the Network Link Conditioner

[The Network Link Conditioner](#) is a powerful tool available in the “Developer” settings of iOS devices. It allows you to simulate adverse networking conditions to test your app’s behavior. You can evaluate and optimize your app’s performance under conditions like 100% packet loss, 3G, DSL, Edge, or a bad network. This tool helps identify potential issues and allows you to fine-tune your networking strategies to ensure optimal performance and responsiveness for users, regardless of their network environment.



06

Low Power Mode

At any given time, it's important to recognize that a significant portion of your user base, around 20%, may be in Low Power Mode.

06

When a device enters Low Power Mode, various functionalities and system settings are adjusted to conserve battery life. These modifications can significantly impact the performance of your shopping app, potentially resulting in a suboptimal user experience. Let's explore a few of these potential challenges in iOS apps.

Technical causes

Background process limitations

When a device enters [Low Power Mode](#), iOS imposes restrictions on background activity to conserve battery life. Certain background tasks, such as network requests or location updates, may experience delays or even be suspended. iOS also reduces the frequency of background app refresh, which can cause delays in retrieving updated data from the server. This can result in the app displaying outdated information to users.

Reduced CPU and GPU performance

When a device is in Low Power Mode, iOS reduces the performance of the CPU and GPU to conserve energy. This reduction in processing power can lead to slower app performance and decreased graphical capabilities.

Network connectivity restrictions

Low Power Mode limits network activities to save power. This can impact real-time data updates, push notifications, and content refreshing in your iOS shopping app.

Location services limitations

In Low Power Mode, iOS disables or restricts location-based services to conserve battery life. This can affect features in your shopping app that rely on location, such as store locators or personalized recommendations.

Best practices

Limit background activity

Identify critical functionalities in your iOS shopping app that require background activity, such as push notifications or background data synchronization, and ensure they are optimized for efficiency. Use the [background task framework](#) provided by iOS, allowing the operating system to allocate resources appropriately based on system conditions.

Optimize network usage

Minimize unnecessary data transfers and reduce the frequency of network requests whenever possible. Consider implementing techniques such as request batching, which combines multiple requests into a single request to reduce network communication overhead. Implement intelligent data caching to reduce the reliance on real-time data fetching, ensuring that essential information is readily available even when network connectivity is limited.

06

Adjust power-intensive features

Identify power-intensive features in your iOS shopping app, such as animations or resource-heavy operations, and adjust these features to work optimally in Low Power Mode. For example, you can reduce the complexity or frequency of animations, implement more efficient algorithms, or use system-provided optimizations like lowering the frame rate or disabling unnecessary visual effects. By adapting power-intensive features, you can balance functionality and battery efficiency, providing a better user experience in Low Power Mode.

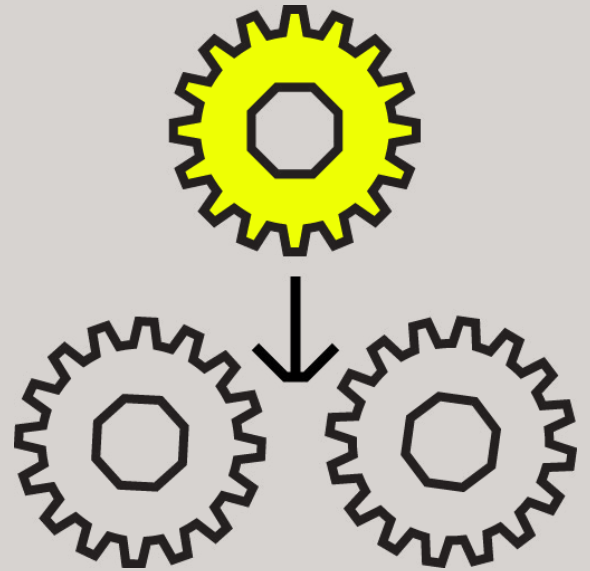
Steps you can take today

Understand background process limitations in Low Power Mode

By understanding the limitations of background app refresh, CPU and GPU performance, resource utilization, and location services, you can effectively design and optimize your iOS shopping app to work within the constraints of Low Power Mode.

Detect Low Power Mode to optimize app behavior

Implement mechanisms to detect when users have entered Low Power Mode and dynamically adjust your app's behavior to optimize power efficiency. This can include reducing animations or visual effects, minimizing network activity, suspending non-essential background tasks, or modifying the frequency of data refreshing. By adapting your app's behavior based on the user's power state, you can conserve battery life and provide a seamless user experience.



07 Third-party integrations

In mobile, seamless integration with third-party services is crucial for enhancing app functionality and user experience. However, various challenges can arise when integrating with external services.

Third-party integration issues can impact critical functionalities like payment processing, analytics, social media integrations, or ad placements. These issues can significantly impact your app's performance, reliability, and overall user experience.

Let's explore a few ways third-party integration issues can surface in your iOS apps.

Technical causes

Incorrect API usage

Integration issues can occur when the app improperly uses the APIs provided by third-party services. This may involve using outdated API versions, providing incorrect parameters, disregarding request/response formats, or lacking compatibility with the latest specifications. Such misuses can result in conflicts, unexpected behavior, or errors during API communication.

Swizzling abuse and/or misuse

Swizzling is a technique used by SDKs to modify the behavior of existing methods in Foundation classes. However, conflicts can arise when multiple SDKs attempt to swizzle the same method, leading to unpredictable behavior, crashes, or even security vulnerabilities.

Authentication and authorization problems

Issues can arise when authentication mechanisms are improperly implemented or when credentials, tokens, or API keys used for authentication are invalid or expired. In addition, authorization problems can occur if the app lacks the necessary permissions to access certain functionalities or data from the third-party service.

Unstable or unreliable third-party services

Integration issues can stem from the performance problems, downtime, or inconsistent response times of third-party services. Relying on unreliable services can lead to slow loading times, failed requests, or degraded functionality within the app, impacting the user experience.

iOS dependency conflicts

Compatibility issues can arise if the third-party integration is not aligned with the targeted iOS versions for the application or violates the [guidelines set by the App Store](#). Certain integrations may have minimum iOS version requirements, and using them on older iOS versions can result in compatibility problems or unexpected behavior.

Best practices

Payment provider optimization

Optimize your payment provider integration by thoroughly testing their performance and monitoring their capabilities to handle peak traffic. This ensures a seamless and reliable payment experience for your users, minimizing disruptions or slowdowns during the transaction process.

Test ad vendor performance across devices

By evaluating how ads perform on different devices and screen sizes, you can identify vendors that deliver ads efficiently without negatively impacting app performance. This allows you to balance monetization through advertisements while maintaining a smooth user experience.

Enable graceful degradation

Implement error handling mechanisms and prioritize critical features that must remain functional even if the third-party integration fails. By enabling graceful degradation, you ensure that users can still access essential features and services, even in the event of a failure or disruption in the third-party integration. Thorough testing and continuous monitoring of the integration's performance are essential to promptly detect and address any issues.

Implement failover caching

Identify critical data or content that needs to be cached and develop a caching strategy. Determine when and how the data should be cached, considering factors such as expiration time and cache update frequency. Implement a cache layer within your app that seamlessly switches to cached data if the third-party integration fails. Regularly update and refresh the cache to ensure up-to-date information is available to users. Handle cache invalidation gracefully to avoid serving stale data and provide a reliable and consistent experience.

Implement retry policies

Define a strategy to handle temporary failures in the third-party integration by implementing retry policies. Identify the API calls or operations that require retry logic and determine appropriate retry intervals and the maximum number of retries based on the integration. Implement error handling mechanisms to detect and capture errors, and use algorithms like exponential backoff to gradually increase the interval between retries. Consider incorporating network reachability checks to ensure retries are attempted only when the device has a stable internet connection.

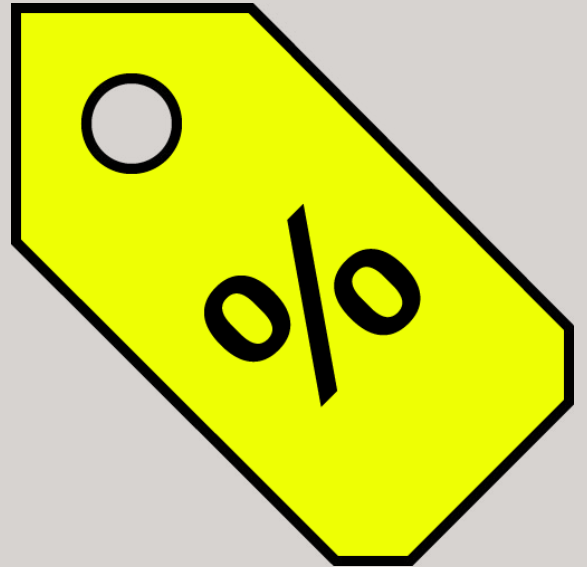
Steps you can take today

Adhere to third-party rules and guidelines for better integration

It's important to thoroughly review and follow the documentation and guidelines provided by the third-party service you're integrating with. This ensures a smooth integration process and minimizes the risk of failures or compatibility issues. Understanding and adhering to the recommended practices can optimize the integration and provide a seamless user experience.

Handle SDK failures and payment provider issues with graceful fallbacks

In the event of an SDK failure, you can provide alternative functionality or features that ensure the app remains usable. For example, if a specific SDK is unavailable, you can offer a simplified version of the feature or substitute it with a similar solution. Similarly, in case of payment provider issues, you can seamlessly switch to a backup provider or offer alternative payment methods to ensure a smooth transaction process for your users. By planning ahead and implementing these fallback strategies, you can mitigate the impact of potential failures.



08 Discounts or sales

In shopping apps, promotions involving discounts play a crucial role in attracting customers and boosting sales. From push notifications to emails to text messages, there are many channels to directly reach your customers and present them with enticing deals.

However, discount or sales issues can significantly affect the shopping experience and business operations. These issues can result in inaccurate final prices at checkout, leading to customer dissatisfaction and abandoned purchases.

Let's explore a few common ways discount or sales issues can surface in your app.

Technical causes

Inconsistent data synchronization

When the app and backend systems fail to synchronize discount data effectively, discrepancies can arise. Outdated or incorrect discount information may be presented to users, leading to confusion and dissatisfaction.

Inaccurate coupon handling

Improper validation or processing of coupon codes can result in various issues. For example, users may experience incorrect discount calculations. On the other hand, users might accidentally use the same code multiple times or add expired or invalid discount codes, leading to inconsistent and unfair discounts.

Fraudulent activity

Discounts and sales can attract fraudulent activity, including fake discounts, coupon abuse, and unauthorized stacking of discounts. Such activities undermine the integrity of the promotion and can result in financial losses for the business.

Caching and invalidation problems

Caching plays a crucial role in optimizing app performance. However, improper management of cached discount data can lead to outdated or incorrect information being presented to users. Inadequate cache invalidation strategies may result in users seeing expired or no longer valid discounts.

Integration issues with payment gateways

Problems can arise when integrating payment gateways with discount calculations during the checkout. If the integration is not properly implemented, it can result in incorrect discount calculations or failures in applying discounts at the payment stage.

Best practices

Ensure clear backend management

Collaborate closely with the backend team to establish clear communication channels and design efficient systems for handling complex discount scenarios. This includes defining discount rules, validation processes, and synchronization mechanisms to ensure accurate and consistent discount information across the app and backend.

Implement anti-fraud techniques

Implement robust validation and security measures to prevent inconsistencies and fraudulent activities related to discount codes. This can include techniques such as code expiration, usage limits, and user verification to ensure that discounts are used appropriately.

Implement a reliable push notification system

Segment your audience based on user preferences, past purchase history, or other relevant criteria to deliver personalized notifications. Optimize the timing of notifications to ensure they reach users at the most opportune moments. Test and monitor the delivery of notifications to ensure they reach the intended recipients. Provide clear and concise information about the discounts and sales in the notifications, and offer opt-in and opt-out options to respect user preferences.

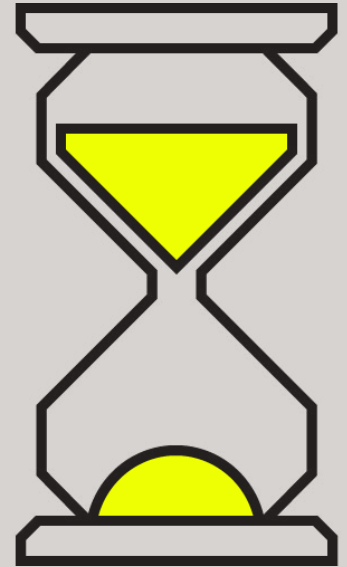
Steps you can take today

Ensure high quality for your mobile app

Delivering a high-quality app is paramount for a positive user experience, to identify and resolve any issues or bugs, and perform thorough testing, including functional, user experience, and performance testing. Regularly update and maintain your app to ensure it remains stable, reliable, and free from glitches, specifically focusing on optimizing the discount and sales features. Pay attention to app responsiveness, loading times, smooth navigation, and intuitive user interfaces that make finding and redeeming discounts effortless. By delivering a high-quality app tailored to discounts and sales, you'll establish trust with your users, encourage increased engagement, and maximize the effectiveness of your promotional campaigns.

Implement functional fallbacks for payment methods

When integrating multiple payment methods into your mobile app, it's important to have functional fallbacks in place. If a specific payment method encounters issues or is temporarily unavailable, users should have alternative payment options to complete their transactions seamlessly. Consider offering options like credit/debit cards, mobile wallets, or other payment gateways to provide flexibility to your users.



09

UI freezes or lag

Shopping apps often encounter UI freezes or lag, which can be attributed to inefficient code execution, bottlenecks in UI rendering, or excessive UI updates. These issues can hinder the smooth and responsive user experience crucial for a successful shopping app.

Many factors can contribute to UI freezes or lag in iOS apps. Let's explore some of the common causes now.

Technical causes

Heavy UI rendering

Complex elements, such as high-resolution images, animations, or dynamic content, can require significant processing power and memory to render. These elements can strain device resources if not optimized properly, leading to UI freezes or lag. For example, loading multiple large images simultaneously or rendering complex animations can cause rendering delays and degrade the app's overall performance.

Excessive UI updates

Having frequent and unnecessary updates to the UI, even for minor changes, can overload the rendering system and result in UI freezes or lags. Excessive UI updates include unnecessary layout recalculations, redundant view updates, or continuous reloading of data that hasn't changed.

Device performance limitations

UI freezes and lags can occur in iOS shopping apps when resource-intensive operations push the limits of low-end or older devices. These devices may have limited processing power, memory, or graphics capabilities, making it challenging to handle demanding UI rendering and updates. As a result, the UI may become unresponsive or exhibit noticeable lag, negatively impacting the user experience.

Resource-intensive background tasks or processes

Certain background tasks or processes, such as data synchronization, content updates, or complex calculations, can consume significant device resources. If these tasks are not properly managed, they can compete with UI rendering and responsiveness.

Improper state handling by SwiftUI

[SwiftUI](#), Apple's declarative UI framework, may exhibit issues with improper state handling, which can contribute to UI freezes or lags. Sometimes, the framework may cause the same screen to load multiple times, leading to unnecessary UI updates and performance degradation. This can be particularly problematic in iOS shopping apps where frequent UI updates are common, such as when adding or removing items from a cart or updating product availability.

Best practices

Code optimization

Analyzing and optimizing code is essential for addressing UI freezes and lags. This involves identifying and improving performance bottlenecks, using efficient algorithms and data structures, and eliminating redundant code. By optimizing the code, you can reduce processing overhead and improve the overall responsiveness of the app. In addition, profiling tools and performance monitoring can help identify areas of improvement and guide optimization efforts.

Move time-consuming tasks onto background threads

It's important to offload time-consuming tasks onto background threads. This allows resource-intensive operations, such as data fetching, image processing, or complex calculations, to be performed without blocking the main thread. Minimize UI freezes and lag by distributing the workload across multiple threads, and leverage techniques like [Grand Central Dispatch \(GCD\)](#), [Operation Queues](#), or asynchronous APIs to more effectively handle background tasks.

Reduce UI updates

Excessive and redundant UI updates can overload the rendering system and lead to UI freezes or lag. Techniques like batched updates, where multiple UI updates are grouped, can reduce the frequency of updates and optimize the rendering process. In addition, leveraging algorithms like diffing or selective data binding can help identify and update only the specific parts of the UI that have changed, minimizing the overall workload and improving UI responsiveness.

Steps you can take today

Use SwiftUI for UI rendering and avoid cyclic changes

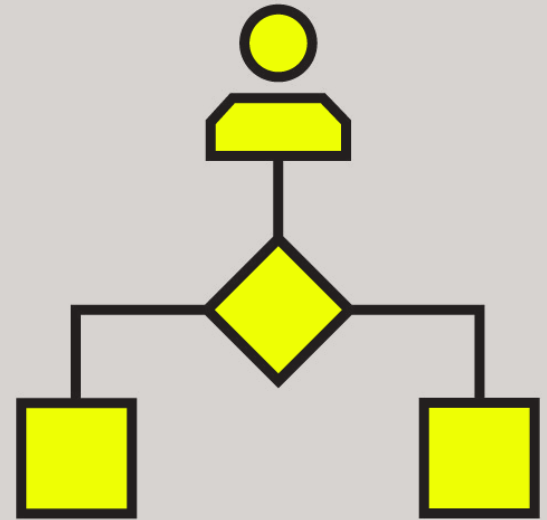
When using SwiftUI, it's important to leverage its features effectively while avoiding cyclic changes in the UI. Cyclic changes occur when a view triggers updates that cause subsequent updates, leading to an infinite loop. You can create a stable and efficient UI rendering process by understanding SwiftUI's principles and employing best practices, such as using state management effectively and structuring your views appropriately.

Use main threads/actors effectively

Use the main thread/actor for UI updates, user interactions, and time-sensitive tasks that require sequential execution. It should be dedicated to tasks directly affecting the UI, such as updating views, responding to user input, or performing animations. Keeping these tasks on the main thread/actor ensures that UI updates are timely and the user interface remains interactive.

Use global threads/actors effectively

Use global threads/actors for computationally intensive or long-running tasks that should not block the main thread. Examples include data processing, complex calculations, or network requests. However, ensuring proper synchronization and data access is crucial when multiple threads/actors access shared data. Employ synchronization mechanisms such as locks, queues, or Swift actors to guarantee thread safety and data integrity.



10

User flows

Users may encounter challenges when navigating key flows within your app, like search, onboarding, adding items to the cart, and the checkout process. When users encounter roadblocks, it can lead to frustration and negatively impact the overall user experience.

10

Identifying and addressing issues with user flows is crucial for ensuring a seamless and intuitive app experience. Let's delve into a few examples to better understand these challenges and how they can impact your app's usability and user experience.

Technical causes

Too many screens

Users are more likely to abandon an app if they have to navigate an excessive number of screens. Lengthy onboarding processes or requiring too many taps to access essential product information can lead to user frustration and disengagement.

Checkout or payment issues

Problems during the checkout process, such as issues with payment gateways, validation errors, or coupon code malfunctions, can disrupt the user flow and result in cart abandonment.

Inconsistencies in user flow

In larger organizations or complex development environments, different teams may be responsible for different parts of the user flow, such as cart management, checkout, or payment processing. This division of responsibilities can introduce inconsistencies, miscommunications, or unexpected errors in the user flow.

Lack of personalization or tailored recommendations

When a shopping app fails to provide personalized experiences or tailored product recommendations, this can result in reduced engagement, missed cross-selling and upselling opportunities, decreased conversion rates, limited customer retention, and missed opportunities for data-driven insights.

Device and orientation support

The diverse range of iOS devices, including iPhones and iPads, come in many screen sizes, aspect ratios, and resolutions. Each device has unique characteristics and display properties, impacting how your app is experienced by users. In addition, users can switch between portrait and landscape orientations, further complicating the design and functionality considerations. Failure to account for device and orientation support can result in inconsistent user experiences, visual distortions and misalignment, limited usability, and, ultimately, user frustration and abandonment.

Best practices

Divide user flows into funnels

Analyzing and categorizing user flows into funnels allows you to focus on specific stages of the app, such as search, onboarding, cart, and checkout. By understanding the user journey within each funnel, you can identify pain points and areas for improvement. Optimize the product screens, marketing and notification strategies, and seller and buyer interactions to enhance the overall user flow experience.

Streamline your shopping cart

The process of adding items to the cart should be seamless and intuitive. Implement prominent buttons and visual feedback to ensure users can easily add products to their carts. Consider using persistent indicators or floating buttons to provide convenient cart management options, such as viewing the cart contents or accessing the checkout process.

Reduce friction during checkout

Minimize the steps and input fields required for checkout. Provide guest checkout options to eliminate the need for account creation, allowing users to complete purchases quickly. To accommodate user preferences, offer multiple, convenient payment options like Apple Pay, PayPal, and credit/debit cards. Use progress indicators and provide clear instructions at each step to guide users through the checkout process efficiently.

Implement proper testing

Conduct unit tests to verify the functionality of individual components, and acceptance or functional tests to check the end-to-end flow. Thorough testing helps identify and resolve any issues or inconsistencies, ensuring a smooth and reliable user experience.

Use intuitive navigation

Design the user flow to follow an intuitive navigation structure that aligns with iOS design patterns. Use familiar navigation elements such as tab bars, navigation bars, and swipe gestures to provide a seamless and intuitive navigation experience. Keep the navigation hierarchy simple and consistent throughout the app, ensuring users can easily navigate between different screens and sections.

10

Steps you can take today

Automated testing for user flows

By setting up automated tests, you can validate acceptance criteria and functional aspects of your app's user flows. This allows you to identify and address any issues or inconsistencies in the flow, ensuring a smooth and seamless user experience.

Restore app state

Users frequently switch between apps, so you should gracefully handle these transitions and restore the app state when they return. By saving key information like cart items or checkout progress, you can enable users to quickly pick up where they left off. This enhances convenience and reduces friction in key user flows, ultimately improving engagement and conversion rates.

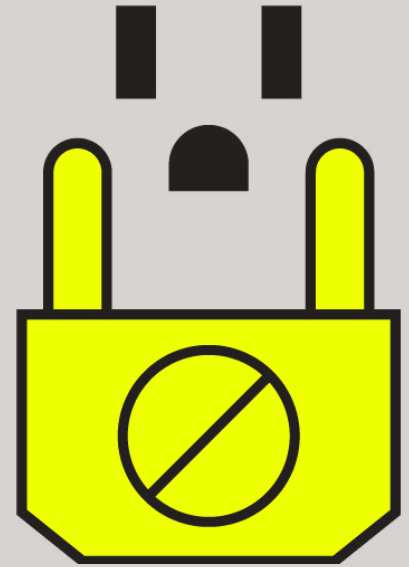
Track app exits for user insights

You can gain valuable insights into user frustrations and pain points by tracking technical and behavioral events leading up to app exits. Analyzing this data allows you to isolate and address specific issues that cause users to abandon carts and purchases. These insights are crucial for spotting optimization opportunities to significantly boost conversion rates.



Take advantage of iOS automated testing tools to improve user flows

1. **XCTest:** The native testing framework provided by Apple for iOS app testing. It is built into Xcode and offers a comprehensive set of tools for UI testing, performance testing, and unit testing. You can write tests in Swift or Objective-C to automate user flows, including navigating through product listings, adding items to the cart, applying discounts, and completing the purchase process.
2. **Earl Grey:** A [powerful UI automation framework](#) developed by Google specifically for iOS apps. It integrates seamlessly with XCTest and provides additional functionality for testing complex user flows. EarlGrey supports interactions with UI elements, synchronization, and performance testing, making it suitable for automating shopping-related user flows and ensuring their reliability.
3. **KIE (Keep It Functional):** An [open-source iOS integration testing framework](#) that allows you to write tests in Objective-C. It focuses on providing a human-readable syntax for testing user interactions and flows. KIF supports simulating user actions like tapping buttons, entering text, and navigating between screens. With KIF, you can create tests that simulate the entire shopping process, from product browsing to order placement.
4. **Appium with XCUITest:** A [popular cross-platform automation framework](#). When integrated with XCUITest, Appium becomes a powerful tool for automating user flows in iOS shopping apps. With Appium, you can write tests using various programming languages and leverage XCUITest's features to interact with UI elements and validate user flows in your shopping app.



11 Compatibility issues

In the dynamic landscape of mobile devices, shopping apps must be compatible with a diverse range of device models, screen sizes, and operating systems. Failing to thoroughly test the app on different devices can result in compatibility issues that adversely affect the user experience, such as UI inconsistencies, layout problems, or functional errors.

Compatibility issues can manifest in many ways within iOS apps. Let's go over a few of them now.

Technical causes

App functionality is not optimized for device variations

Compatibility issues can arise when shopping apps are not optimized for a wide range of device variations, including screen sizes, resolutions, and hardware capabilities. This can result in layout issues, feature inconsistencies, or rendering errors. For example, if the app's layout is not responsive and adaptable, UI elements may be misaligned or improperly scaled on certain devices, leading to a disjointed and unappealing user interface.

API version mismatches

Shopping apps integrate with various APIs and services, such as payment gateways or shipping providers. Compatibility issues can occur when the app relies on unsupported or non-backward compatible API versions. This can lead to errors, failed requests, or unexpected behavior. It is important to keep the app's API integrations current and ensure compatibility with the required API versions to maintain smooth and uninterrupted functionality.

User customization preferences

Shopping apps often allow users to customize themes, fonts, or accessibility settings according to their preferences. However, compatibility issues can arise if the app doesn't handle these customizations properly or they cause conflict with the app's design or functionality. This can result in visual inconsistencies, layout problems, or functional limitations.

Backend system incompatibility

Shopping apps rely on backend systems, databases, and external services for transactions and inventory management. Compatibility issues can occur if there are inconsistencies or gaps between the app's frontend and the backend systems it interacts with. This can lead to data synchronization problems, transaction failures, or incorrect information displayed to users.

Best practices

Implement a responsive design

Ensure your shopping app's UI design is responsive and can adapt to different screen sizes, resolutions, and orientations. This involves creating flexible layouts, scalable UI components, and adaptive UI techniques like Auto Layout or SwiftUI's layout system. By implementing a responsive design, you can provide a consistent and visually appealing experience across various devices, enhancing compatibility and usability.

Ensure iOS version compatibility

Determine the minimum iOS version your shopping app will support. While it can be tempting to target the latest iOS version to leverage its newest features and APIs, it's crucial to carefully assess the trade-off between functionality and compatibility. Consider the user base and device usage statistics to understand the impact of excluding older devices that may be unable to upgrade to the latest iOS version.

Continuously update and address bugs

Stay vigilant in updating your app based on user feedback and bug reports to address compatibility issues that may arise with new devices or OS versions. Regularly releasing updates that address these issues demonstrates your commitment to providing a reliable app. Conduct thorough testing on different devices and OS versions, so you can ensure your app remains compatible with evolving technologies.

Steps you can take today

Leverage new APIs for localization, asset management, and color schemes

Embrace Apple's latest APIs and technologies for localization, asset management, and color schemes in your iOS shopping app. By staying updated, you can enhance the user experience and streamline development. Leverage APIs to effortlessly translate your app into multiple languages, adapt content, currency formats, and date/time representations for different regions. Optimize multimedia management with APIs, effortlessly handling images, videos, and audio files. Utilize asset catalogs to organize app-specific assets, support various resolutions, and ensure consistency across different device screen sizes. Embrace color scheme advancements, enabling dynamic color schemes that adapt to user preferences and system settings like dark mode.



APIs to enhance your app's user experience:

1. **Themes:** [UIAppearance](#) enables you to customize the appearance of UI controls and views throughout your app. You can set custom colors, fonts, and other visual attributes to create a cohesive, branded look and feel across your entire app. This allows you to maintain consistency and provide a visually appealing experience for your users.
2. **Fonts:** [UIFont](#) offers flexibility in specifying custom fonts for text elements in your app. You can choose from a wide range of system-provided fonts or add custom fonts to your project and use them programmatically. By carefully selecting and implementing fonts, you can create a visually pleasing typography that aligns with your app's design language and enhances readability.
3. **Accessibility Settings:** [UIAccessibility](#) provides a comprehensive set of accessibility APIs that enable you to make your app inclusive and accessible to users with disabilities. With these APIs, you can enable essential features like VoiceOver, which provides spoken feedback for visually impaired users, and Dynamic Type, which allows users to adjust text size based on their preferences. In addition, you can create custom accessibility labels for UI elements, ensuring that users with assistive technologies can navigate and interact with your app effectively.
4. **Dynamic Type:** [UIFontDescriptor](#) is a powerful iOS feature that allows users to adjust text size within their device's settings. By leveraging this API, you can support Dynamic Type in your app by providing font styles that adapt to different text size preferences. This ensures your app's content remains readable and accessible to users with varying visual needs.
5. **Dark Mode:** [UITraitCollection](#) allows you to adapt your app's UI to Dark Mode, providing a seamless and visually pleasing experience for users who prefer darker themes. Responding to changes in the overall visual and design elements of the UI allows your app to seamlessly transition and maintain consistency across different appearance modes.



12

Lack of error handling

Inadequate error handling in your shopping app can have detrimental effects, such as app crashes, unexpected behaviors, and incomplete transactions. Failing to handle errors properly undermines the reliability and smooth functioning of the app, ultimately impacting the overall shopping experience.

Here are a few examples illustrating how poor error handling can significantly impact app performance.

Technical causes

Insufficient exception handling

This issue poses a significant risk, as it can result in unhandled exceptions or errors being disregarded, leading to unpredictable behavior and even app crashes.

Lack of comprehensive error reporting

Inadequate logging, error tracking, or analytics can hinder error identification and resolution, impacting the app's stability. Without detailed error reporting, developers may struggle to pinpoint the root causes of issues, making it challenging to address them effectively.

Missing error recovery mechanisms

When errors occur, it is essential to provide users with a graceful recovery process. For example, offering alternative actions, guiding users through troubleshooting steps, or automatically restoring the app to a stable state.

Inadequate input validation

Insufficient validation of user inputs can lead to security vulnerabilities and data corruption, which also compromises app stability. It is important to validate and sanitize user inputs to prevent malicious inputs or unexpected data from causing errors or compromising the app's integrity.

Lack of error handling in external integrations

By using the appropriate error-handling techniques, developers can effectively respond to potential exceptions and make informed decisions on resolving them.

When integrating with external services, it is crucial to consider potential errors that may arise and implement appropriate error-handling strategies.

12

This code snippet below demonstrates the importance of implementing proper error handling when an **API** indicates that it may **throw** an exception.

```
enum APIError: Error {
  case networkError
  case invalidResponse
  case authenticationError

  var localizedDescription: String {
    switch self {
      case .networkError:
        return "There was an error produced in the network"
      case .invalidResponse, .authenticationError:
        return "Something went wrong while processing response"
    }
  }
}

struct API {
  func fetchData() async throws -> Data {
    // Simulating a network request with random successful state
    let success = Bool.random()

    if !success {
      throw APIError.networkError
    }

    let responseData = "Response Data".data(using: .utf8)!
    return responseData
  }
}

func processResponse(data: Data) {
  print("Processing response data: \(String(data: data, encoding: .utf8) ?? "")")
}

func fetchDataAndProcessResponse() async {
  let api = API()

  do {
    let data = try await api.fetchData()
    processResponse(data: data)
  } catch let exception {
    print("Error: \(exception.localizedDescription)")
  }
}

Task {
  await fetchDataAndProcessResponse()
}
```


Best practices

Catch and handle exceptions

Use robust exception handling mechanisms with try-catch blocks to catch and handle exceptions effectively. This helps prevent app crashes and ensures that errors are gracefully handled, providing users with a smooth and reliable shopping experience.

Implement error logging and monitoring

Logging error information, such as error messages, stack traces, and relevant contextual data, allows developers to identify and analyze issues more effectively. Reviewing error logs helps identify patterns or trends, enabling proactive troubleshooting and issue resolution.

Improve response error-handling

Implement a centralized error-handling mechanism encompassing standardized error responses, consistent exception handling, clear and informative error messages, and appropriate error logging and monitoring. This ensures errors are handled consistently across the app, providing a cohesive and user-friendly experience. In addition, continuous testing and quality assurance practices should be employed to verify the effectiveness of the error-handling mechanism and identify any potential issues before they impact users.

Steps you can take today

Implement error handling in URLSession

Prioritize building robust error-handling mechanisms within URLSession to handle errors that may occur during network requests in your app. This involves writing error-handling code that captures and analyzes response codes, detects network connectivity issues, handles timeouts, and gracefully manages other potential errors during network interactions.

Design for failure

Adopt a mindset of designing for failure throughout the development process of your mobile app. This means anticipating and planning for failures at every system level, including network requests, data processing, and user interactions. By designing with failure in mind, you can implement resilient strategies such as error recovery, fallback mechanisms, and graceful degradation to mitigate the impact of failures and provide a smoother user experience even in challenging circumstances.

Model errors for robust error handling

Use language-specific error handling features like the Error protocol in Swift or NSError in Objective-C to create structured and informative error models. By defining custom error types that encapsulate relevant information about encountered issues, such as error codes, localized descriptions, and contextual data, you can provide meaningful and actionable error messages to users. Well-modeled errors facilitate better error handling, enable clearer communication with users, and enhance the overall robustness of your app.



13 Payment providers

Payment providers are crucial components of mobile apps. Issues can significantly impact the seamless processing and completion of payment transactions within your iOS app, leading to user frustration and loss of revenue.

13

Here are some key technical issues that can arise with payment providers in iOS apps.

Technical causes

Payment integration challenges

Integrating payment providers into iOS shopping apps can be a complex process involving setting up connections, handling authentication protocols, and implementing the necessary APIs. Challenges regarding compatibility, authentication protocols, or handling specific payment methods or currencies can arise.

Payment processing errors

During the payment process, errors can occur, leading to transaction failures, declined payments, or issues with currency conversions or international payments. These errors can stem from connectivity problems, incorrect payment details, insufficient funds, or issues with the payment provider's systems.

Inconsistent payment statuses

Shopping apps may encounter delays or discrepancies in receiving accurate payment status updates or notifications from payment providers. These inconsistencies can result in confusion among users about the success or failure of their transactions.

Security and fraud prevention

Vulnerabilities and weaknesses in the app's security measures can result in unauthorized access, data breaches, or fraudulent activities. When a shopping app lacks robust security and effective fraud prevention mechanisms, it can significantly affect the app and its users, including compromised user data, loss of user trust, financial losses, legal and regulatory consequences, and damage to brand reputation.

Compatibility and versioning conflicts

Payment providers may introduce updates or new features to their APIs or SDKs, which can create issues for shopping apps that were integrated with older versions. Compatibility and versioning conflicts can significantly affect the app's functionality, stability, and user experience.

Best practices

Error handling and logging

Implement robust error-handling mechanisms to capture and log payment processing errors, both on the client and server sides. This includes capturing error codes, messages, and relevant transaction details. Detailed logging helps diagnose and troubleshoot issues promptly, enabling faster resolution and improving the overall payment experience for users.

Sandboxes

Use sandbox environments provided by payment providers to simulate transactions without affecting actual finances. Test scenarios such as successful payments, declined transactions, and errors to ensure your app handles these cases gracefully. Regular testing helps identify and address issues early, ensuring a smooth payment process for users.

Communication with payment providers

Maintain regular contact via email, phone, or support channels. When encountering issues, promptly contact the payment provider with detailed information, including error codes, logs, and steps to reproduce the issue. Clear and concise communication helps providers understand and resolve problems efficiently, reducing downtime and improving the reliability of payment processing.

Payment gateway compatibility

Verify that the provider offers a well-documented and reliable iOS SDK or API for seamless integration. Review the provider's documentation and SDK features thoroughly to understand any limitations or specific requirements. Choosing a reputable and trusted payment provider with a solid track record of compatibility and support for iOS will help ensure a smooth integration process and a reliable payment experience for your app users.

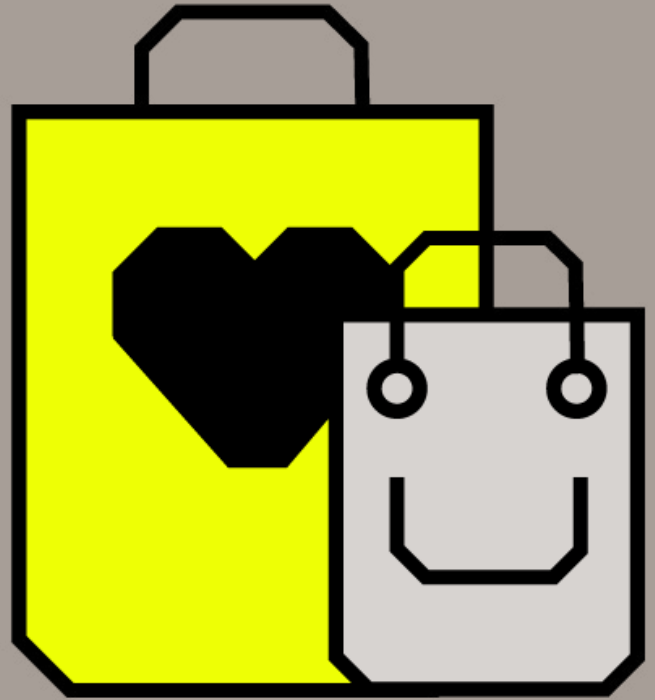
Steps you can take today

Thoroughly test for payment provider integration

Conduct exhaustive testing to ensure a correct and seamless integration. This involves testing standard payment scenarios, considering various edge cases, and performing end-to-end (E2E) testing. Collaborating with the technical staff of the payment provider can provide valuable insights and help validate the integration's robustness. By simulating different scenarios, such as successful payments, declined transactions, and network failures, you can ensure that your app handles these cases gracefully and provides a reliable payment experience for your users.

Implement contingency plans for payment provider failures

Build your app to handle potential failures or issues with the payment provider's services. These failures could include service outages, API changes, or disruptions in network connectivity. You can ensure your application continues functioning smoothly even if the primary payment provider encounters problems by considering alternative options, such as backup payment gateways or manual payment processing. Implementing proper error-handling mechanisms, including informative error messages and fallback options, allows your app to handle unexpected situations gracefully and guide users through the payment process without frustration or confusion.



Embrace helps you deliver amazing shopping experiences

We've explored how app performance issues can significantly impact user experiences. If you're tired of working with limited and fragmented tooling, Embrace is here to help. With Embrace, you can eliminate data gaps and guesswork, enabling you to build better shopping experiences.

Let's explore a few ways to leverage Embrace in your iOS apps.

Auto-capture 100% of your mobile telemetry

Embrace automatically captures, enriches, and transforms signals for every technical and behavioral detail leading to a crash, error, or issue. That means not only do you know the entire user journey throughout your app, you also have the underlying system and app events like network calls, logs, and device state changes. For example, if you're running live promotions, you can uncover exactly what happened leading up to an error in real time before it impacts other customers. With proactive network alerts and custom real-time dashboards tracking the metrics you care about, you'll always be the first to know when something isn't right.

Get the full impact of every issue

Embrace ensures you can access unsampled data to understand the full impact of any issues on your users and business. As iOS engineers, you want to minimize disruptions that hinder feature development. Embrace provides the most reliable way to determine whether an issue requires immediate attention or can be addressed later. With Embrace, you can prioritize your efforts effectively and focus on what matters most.

Resolve issues faster with full user session insights

Embrace offers a play-by-play account of every user experience, giving valuable insights into technical snags and how users interacted with them. You can see what led up to an issue, how the user responded, and what steps you need to take to address it. What's even better is that Embrace provides visibility into both foreground and background sessions. Did a push notification from the background cause a crash? Did a user lose their shopping cart progress after backgrounding the app to take a call? With Embrace, you're not limited to a single session view, but instead gain a holistic picture of what's happening.

Let's break down the 13 common app performance issues and how Embrace can help combat them.

Embrace product offerings:	Use cases Embrace can combat:
<p>Crash Reporting: With Embrace's Crash Reporting feature, teams can effectively investigate and address underlying causes by identifying specific crashes. This ensures faster remediation time, better stability, and an improved user experience. Embrace goes beyond crash tracking and extends its capabilities to identify memory leak-related issues and specific crashes associated with discount/sale periods. By empowering you to effortlessly pinpoint the exact source of the problem, Embrace enables swift resolution.</p>	<ul style="list-style-type: none"> • Memory Leaks • Discount/sales • Lack of error handling
<p>Real-time, proactive Alerting: With Embrace, teams can effortlessly create personalized alerts, guaranteeing prompt notifications for various regressions in performance, stability, or health metrics. This comprehensive monitoring encompasses potential memory leaks, integration-related problems, specific error conditions, performance thresholds, payment provider issues, and discount/sale concerns. By enabling swift identification and resolution of these issues, you can effectively prevent any negative impact on your app's performance.</p>	<ul style="list-style-type: none"> • Memory Leaks • Third-party integrations • Lack of error handling • Payment providers • Discounts/sales

<p>Custom dashboards: Gain a comprehensive birds-eye view to monitor the occurrence and severity of memory leaks over time. Seamlessly compare metrics across different app versions, allowing teams to swiftly identify regressions requiring further investigation. In addition, teams can break out issues based on the operating system or version. This valuable information helps identify compatibility issues when your app runs on specific devices or older OS versions. With these insights, you can proactively address compatibility concerns and ensure a seamless user experience across different platforms.</p>	<ul style="list-style-type: none"> • Memory leaks • Compatibility issues
<p>Network Monitoring: Embrace identifies potential bottlenecks in the startup process, such as slow network calls or excessive network activity. It diligently tracks network requests, measuring their durations and providing valuable insights into latency, throughput, and success rates. By leveraging this information, Embrace helps you swiftly identify issues like slow responses, connectivity problems, or even slow or blocked network requests. With Embrace's monitoring capabilities, you can proactively address these network-related challenges and ensure optimal performance for your application.</p>	<ul style="list-style-type: none"> • Slow startup • Slow loading times • Networking • Third-party integrations • UI freezes and lag
<p>App Performance: Embrace offers comprehensive metric tracking for optimal app performance. It monitors scroll frame rates, smoothness, and responsiveness, providing visibility into potential slowdowns and user frustration. In addition, Embrace tracks response times, network requests, server-side processing metrics, frame rates, rendering times, and UI thread utilization. It also monitors responsiveness, load times, and performance metrics across iOS devices and OS versions. With insights into CPU usage, memory consumption, and network activity, Embrace helps teams swiftly identify and address performance bottlenecks.</p>	<ul style="list-style-type: none"> • Scroll views • Slow loading times • Discounts/sales • UI freezes and lag • User flows • Compatibility issues • Lack of error handling • Payment providers
<p>User Session Insights: Embrace provides a comprehensive view of user interactions, capturing every technical and behavioral detail that led to an issue. It auto-captures and enriches all necessary context, including scroll view interactions, deeplinks, battery issues, integrated services issues, and more. This allows for analysis of specific instances where scroll view issues occur, along with the user's complete context. In addition, Embrace enables instant lookup of user complaints during sales or discounts, reproduction of issues with PII-compliant User-ID Lookup, and analysis of the full play-by-play of each user session. This in-depth analysis helps teams identify bottlenecks, obstacles, and usability or compatibility issues on different platforms, leading to effective issue resolution and an improved understanding of user experiences.</p>	<ul style="list-style-type: none"> • Scroll views • Low Power Mode • Discounts/sales • User flows • Compatibility issues • UI freezes and lags • Third-party integrations


<p>Background Session Monitoring: Embrace offers comprehensive tracking of app activity when it transitions to the background, including duration. This provides users with valuable insights into server-side and device-side monitoring, allowing for a deeper understanding of the factors contributing to increased startup times. Moreover, Embrace identifies excessive or unnecessary background tasks that may result in power drain when the device is in Low Power Mode. This comprehensive monitoring capability helps optimize app performance, reduce power consumption, and enhance the user experience.</p>	<ul style="list-style-type: none"> • Slow startup • Low Power Mode
<p>Error Tracking: Embrace offers valuable insights into networking issues, and analyzes error logs to identify recurring issues, determine their root causes, and prioritize resolution efforts. It also automatically captures detailed error logs, exceptions, and crashes, including those related to third-party integrations. By leveraging OS Log data, Embrace enhances user sessions and crash reports, allowing teams to discover runtime exceptions that impact users without crashing the application. This comprehensive approach empowers teams to identify common errors, prioritize them based on frequency and impact, and proactively improve error handling.</p>	<ul style="list-style-type: none"> • Networking • Third-party integrations • Lack of error handling


Shopping apps provide a highly personalized touchpoint for engaging with consumers. To encourage users to shop within your app, creating a seamless and enjoyable experience is crucial. Throughout this eBook, we have explored numerous factors that can contribute to bad user experiences, which quickly lead to lower conversion rates, app abandonment, and loss of brand loyalty. By incorporating the following tips into your app development process – building robust apps, conducting thorough testing, monitoring app performance, and promptly addressing issues – you can significantly enhance the quality of your iOS shopping app..





Embrace helps mobile engineers identify and prioritize user-impacting app issues with the detailed technical context to resolve them instantly. We provide full user session insight to solve any issue, whether it's a crash, slowdown, ANR or something else. When users are finding problems first, and mobile teams are finding out last, using Embrace empowers quick, early detection of issues before they become widespread. Engineers can focus on building the best player experiences possible and leave the fire drills behind.

Contact

 8569 Higuera St, Culver City, CA 90232

 (424)-326-9004

 contact@embrace.io

 embrace.io

Learn more at embrace.io or follow Embrace on [LinkedIn](#), [Facebook](#), or [Twitter](#).