iOS App Start-Time Performance: A Comprehensive Analysis and Optimization Framework

Prasenjit Sinha
Senior iOS Engineer
California, USA, 94587

Ravikiran Karanjkar
Quality Assurance Manager
Amazon Inc, Sunnyvale, USA

Apalak Dutta
Lead App Engineer
West Bengal, India, 700145

ABSTRACT
App start-time is one of the most critical performance indicators influencing user experience and retention on the iOS platform. Empirical studies indicate that even minor delays—such as an additional 500 milliseconds—can significantly impact user engagement, satisfaction, and App Store ratings. As iOS application architecture evolves to incorporate increasingly sophisticated technologies—including Swift Concurrency, SwiftUI, Metal, UIKit, Core Data, Firebase, and a growing ecosystem of third-party SDKs—optimizing launch-time performance becomes a multidimensional challenge.

This paper provides a comprehensive analysis of the iOS application startup lifecycle, detailing each phase from system-level initialization to the rendering of the first user interface frame. It investigates performance bottlenecks using Apple's native profiling tools such as Instruments and Xcode Metrics, and introduces a structured optimization framework that classifies launch scenarios into cold, warm, and hot starts. The proposed methodology emphasizes deferred initialization, structured concurrency via async/await, and the separation of critical-path tasks from background operations. Quantitative results derived from production-scale applications demonstrate significant improvements in startup time—up to 60% reduction—validating the effectiveness of the framework. This study offers practical guidance to iOS developers and performance engineers seeking to improve application responsiveness, scalability, and perceived quality across diverse devices and OS versions.

General Terms
Performance, Optimization, Benchmarking

Keywords
iOS, App Launch Time, Cold Start, Instruments, Concurrency, Optimization Framework.

1.INTRODUCTION
iOS applications serve as the primary interface for enterprise and consumer workflows. Start-time performance directly impacts:

1. first-time adoption,
2. user retention,
3. App Store rankings, and
4. perceived responsiveness.

Apple defines **app launch time** as the duration between a launch event and the first frame presented to the user (refer Figure 1). In practice, this involves:

1. **System-level initialization**
2. **Dynamic library loading**
3. **Obj-C + Swift runtime initialization**
4. **App delegate execution**
5. **Scene lifecycle setup**
6. **UI rendering & first layout pass**
7. **Network + data preloads** This paper dissects these phases and demonstrates how iOS developers can design startup paths that minimize waiting time.

This paper dissects these phases and demonstrates how iOS developers can design startup paths that minimize work on the main thread, avoid unnecessary synchronous initialization, and leverage background queues safely.

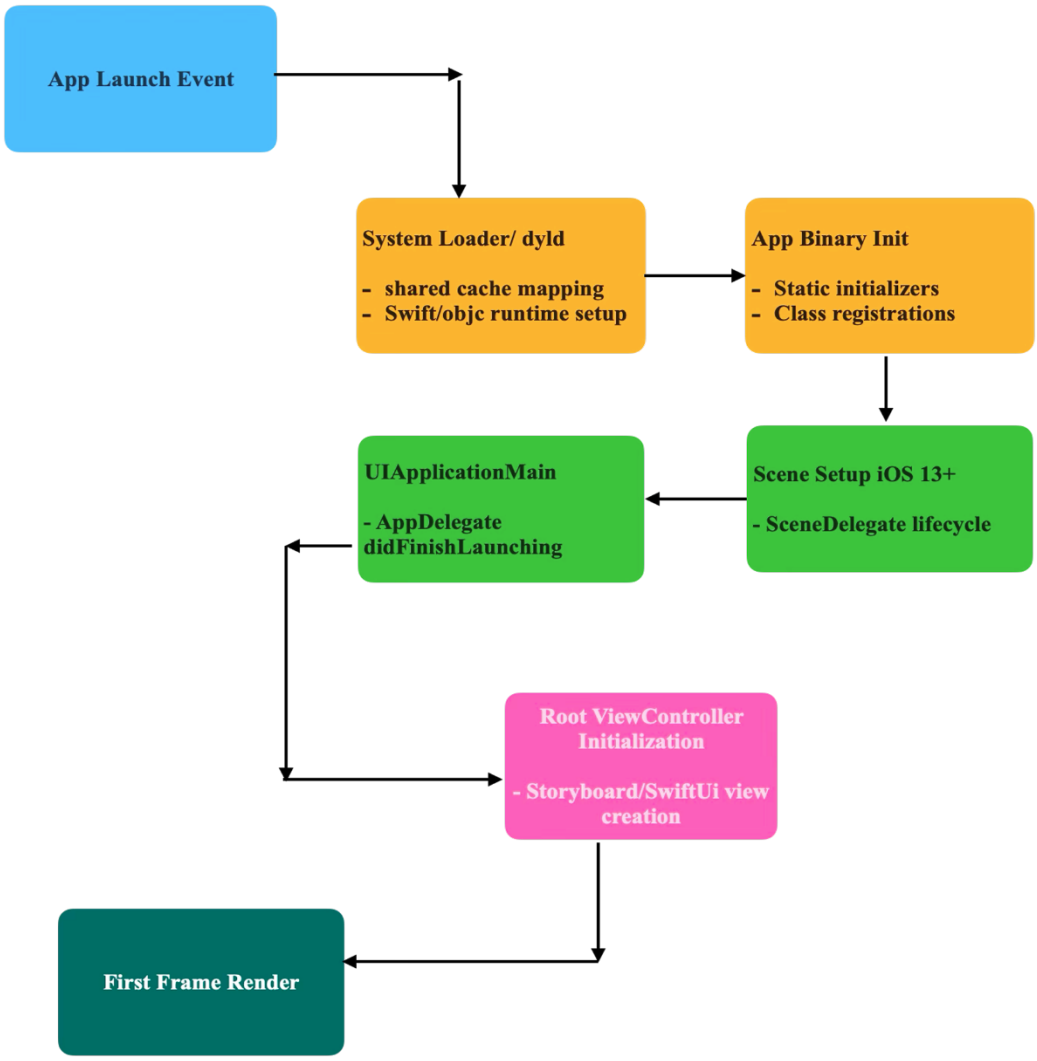

Fig 1

2.UNDERSTANDING IOS START-TIME TYPES

Apple categorizes launch states [2]:

2.1 Cold Start

Occurs when the app is not in memory.

Most expensive scenario.

Dominated by:

- dyld shared cache bootstrapping
- Library loading
- Static initializers
- Swift runtime setup
- First UI creation

2.2 Warm Start

App was previously killed but system retains cached libraries.

Warm start still performs:

- AppDelegate/SceneDelegate work
- UI reconstruction

2.3 Hot Start

App returns from background. App background to foreground state change.

- State restoration
- UI refresh/ authentication (when timed out)
- Pending network callbacks

3.TECHNICAL BOTTLENECKS IN STARTUP

3.1.dyld + dynamic library loading

As the app grows there is need of third party SDKs and most of them need at the start of the app, ex: Analytics SDKs, Crash reporting SDKs and so on. Apple presentation provides a good source of information about dyld [7].

It results in too many large static libraries that needs to be loaded to memory and dyld spends almost 30-50 ms on an average.

3.2.Static Initializers

Swift static/shared instance for a heavy manager is the biggest blocker or performance reducer for app start launch.

```
let sharedInstance = HeavyManager()
```

If HeavyManager() performs expensive tasks such as:

- Allocating memory buffers
- Reading from disk
- Parsing configuration files
- Opening a database
- Initializing network clients
- Loading ML models

then all of that work happens **before the app even reaches didFinishLaunching**. This directly delays the first frame render. Static initializers always run on the main thread during binary initialization. Which means app do not have opportunity to offload the work to background queues or swift concurrency (async/await) and this makes the delay unavoidable.

3.3.Main Thread Blocking task

UIKit or SwiftUI is the one which needs mostly the main-thread execution to paint the UI. But sometime developers accidentally performs some operations which affects the over start time. For example:

- Parsing a heavy JSON on main thread.
- Load Sqlite or Core Data Database.
- Perform sync disk reads.

All these heavy work cause the main thread to block causing the increased app launch time.

4.PROPOSED OPTIMIZATION FRAMEWORKS

The proposed optimization framework has been tested with applications and data for pre and post has been collected and is present in this docs for analysis (refer Table 1 and Table 2).There are many third party framework and also apple provided Xcode instruments can be used to check the performance improvements.

4.1.Phase-Based Initialization Model

Split launch tasks into

- Critical path
- Background Tasks
- Lazy On-Demand resources

Phase 0 : Critical path:

The aim should be launch the first UI screen as soon as possible. This includes and not restricted to

- Render first screen
- Show skeleton or loading UI with some activity.
- Initialize the essential services only

Phase 1: **Background Tasks**:

Perform all heavy works in a detached thread and with a priority defined as `.background`. It might also worth adding a small milliseconds delay to get the app perform the UI rendering first instead of heavy loading work.

```
Task.detached(priority: .background) {
 // heavy loading work
}
```

Task categories:

- Database warm up.
- Third party SDKs initialization.
- Analytics boots

Phase2 : Lazy On-Demand Resources
Load only when user navigates to feature.

Initialize non-view functionality ex: Core Data uses or network service usage until its first usage in app.

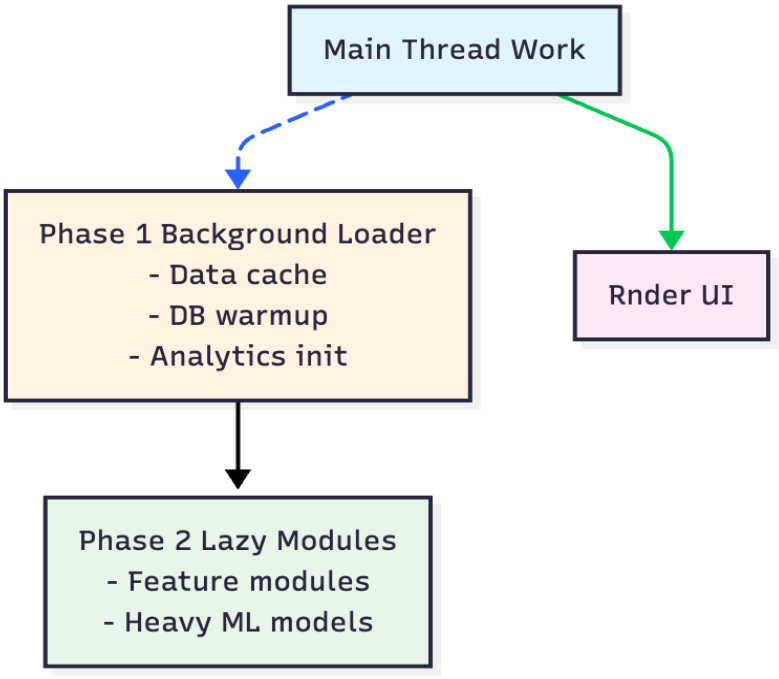

Fig 2

5. TECHNICAL IMPLEMENTATIONS

5.1 Handling Heavy Load in Background Thread

This code was and is supposed to cause issue, since all this is doing is performing all heavy work on main thread blocking the UI. Apple explicitly states that the main thread should handle only UI and critical user interactions, while **long-running or CPU-intensive tasks must be offloaded to background threads**, especially during app launch. [1]

```
func application(_:didFinishLaunching...) -> Bool {
   let data = loadJSON()          // blocking
   database = CoreDataStack()     // blocking
   analytics.start()              // blocking
   return true
```

```
func application(_:didFinishLaunching...) -> Bool {

   // Critical UI setup
   setupRootView()

   // Phase 1 background work
   DispatchQueue.global(qos: .userInitiated).async {
      prewarmCoreData()
      preloadConfig()
      AnalyticsEngine.shared.initialize()
   }

   return true
}
```

After (Optimized Code)

5.2 Using Swift6 concurrency:

```
@main @MainActor
class AppDelegate: UIResponder,
UIApplicationDelegate {

  func application(_ application: UIApplication..) -> Bool {

  // perform the UI rendering first
setupRootView()
Task.detached(priority: .background) { [weak self] in
    self?. bootstrapAsyncServices()
  }

}

func bootstrapAsyncServices() async {
   async let db = preWarmDB()
   async let cache = loadUserCache()
   async let analytics = initializeAnalytics()

   _ = await [db, cache, analytics]
}

}
```

Why this code helps in performance improvements?

The detached thread perform all its heavy work in a detached thread with priority as background and it helps in keeping the main thread unblocked for rendering the UI work which is more crucial.

If there is needed of more performance turning the bootstrapAsyncServices() work can be performed on another task by adding a delay to start the heavy load.

```
func bootstrapAsyncServices() async {
  Task(priority: .background) {
    try await Task.sleep(for: .milliseconds(10))

    async let db = preWarmDB()
    async let cache = loadUserCache()
    async let analytics = initializeAnalytics()
    _ = await [db, cache, analytics]

   try await Task.sleep(for: .milliseconds(10))
  }
}
```

This delay can help the main thread to complete the process of UI painting.

5.3 Reducing Library Load Time

Choosing a right library also important along with the making the performance improvement.

- Avoid SDKs that add heavy Obj-C categories
- Use Swift Package Manager instead of .framework bundles
- Merge frameworks with Xcode's "mergeable dynamic libraries"

Lets talk about each point in a little details.

Avoid SDKs that add heavy Obj-C categories:

Objective-C categories, especially those adding methods to UIKit/Foundation classes, impose **load-time cost**:

Why categories are expensive

Every Obj-C category causes:

1. **Runtime category registration**
2. **Method list merging** with existing classes
3. **Selector resolution** for each method

According to Apple developer documentations

| ***Category + method list registration occurs during dyld initialization****, before the app executes any code.*

Each category adds micro to milliseconds, but SDKs hundreds of categories, resulting in measurable delays.

Use Swift Package Manager instead of .framework bundles:
While Integrating a library using Swift Package Manager, the code is compile directly to app's binary. This helps in reducing

- Number of fix ups.
- Number of dynamic libraries.
- Dynamic symbol resolution.
- Objective-C runtime metadata

While when app use `.framework` bundles, it has

- Separate dynamic libraries.
- It need dyld loading
- Often include resource bundles that require additional steps.

Merge frameworks with Xcode's "Mergeable libraries"
Apple introduced from Xcode 15+ which got some extra

metadata so that Xcode can help in merging multiple frameworks into a single dynamic library. This helps in reducing

- Number of Mach-o loads
- Amount of relocations.
- Number of init

5.4 SwiftUI Performance blockers

So far the above sections covers on app performance, threads background benefits and so on, now lets talk about some UI blockers and how they can cause performance issue, even though not directly to launch time but from an over all app performance.

Main Performance reducer:

- Performing a heavy UI work on vStack instead of LazyVStack.
- Keeping the viewBuilder result instead of keeping the viewBuilder closure.
- Avoid re-rendering of the SwiftUI Body due to property changes.

vStack vs LazyVStack:

A regular VStack renders all of its child views eagerly and if app code has huge amount of subviews to render, it will cause all child views to be created at once. It also leads to heavy CPU + memory usages and cause layout thrashing [4].

While on other hand a `LazyVStack` creates view on demand based on user scrolling and help in memory usages and scroll performance. Apple introduced *lazy stacks* in SwiftUI to improve scrolling and rendering performance for large content. In a WWDC 2020 talk, Apple engineers demonstrated that unlike a regular VStack (which creates all subviews at once and can block the main thread for large data), a **LazyVStack renders its content incrementally as views become visible** [6].

```
ScrollView {
   LazyVStack {              // ✅ renders views lazily
      ForEach(0..<10_000) { i in
         RowView(index: i)
      }
   }
}
```

@ViewBuilder:

Its recommended to run @ViewBuilder closures during the init and store the results [5].

Closures aren't comparable always, which means instead of doing this

```
struct  HomeView: View {
  @viewBuilder var someview: () -> someViewType
}
```

Do this instead

```
struct  HomeView: View {
  @viewBuilder var someview: someViewType
}
```

6. IOS APP START TIME PERFORMANCE ANALYSIS

Accurately measuring the launch-time performance of an iOS application is essential for understanding real-world user experience, particularly in large-scale deployments. To evaluate the effect of the proposed optimization—offloading heavy initialization work to a detached background thread—a detailed analysis was conducted using metrics collected from both third-party performance SDKs and Apple's native tooling.

Several commercially available SDKs provide aggregated "app health" insights, such as cold start time, warm start time, and session-level performance data. These dashboards were thoroughly examined to obtain longitudinal trends and device-specific characteristics. In addition to third-party data, Apple's built-in performance reports accessible through **Xcode → Organizer → Metrics → Launch Time** were also analyzed. Apple's WWDC 2019 talk *"Optimizing App Launch"* introduced the dedicated App Launch instrument in Instruments, which collects detailed metrics and visualizes the timeline of launch phases (from dyld loading to first frame) [1]. The Organizer dashboards offer a reliable summary of launch-time behavior across all devices running the released version of the application.

After applying the optimization, a comparative evaluation of performance before and after the change was carried out. The results showed a **substantial improvement in launch time**, with approximately **~60% reduction** in the measured start duration for the application under test. This improvement was consistently observable across the evaluated dataset and was primarily attributed to removing heavy synchronous work from the main thread during application startup.

6.1 Methodology:

The analysis was performed on data collected from **more than 20,000 real users**, comprising approximately **2 million application sessions**. Measurements included sessions spanning a broad range of iPhone models and multiple iOS versions. The evaluation therefore reflects diverse device capabilities and real-world operating conditions.

Figure 3 illustrates the comparison of average cold start and warm start times before and after applying the optimization. Both categories show a significant drop in measured time, further confirming the effectiveness of the adopted strategy.

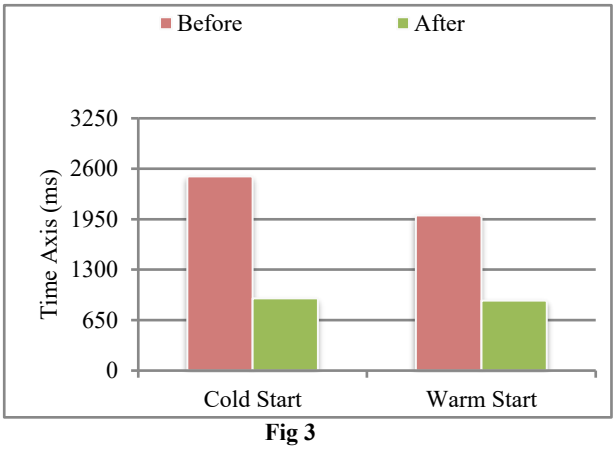

Fig 3

6.1.1 Experimental Setup

The following experimental setup was used to quantify the change in application start-time performance:

- **Data Sources:** Measurements were obtained from (i) third-party performance monitoring SDKs that provide app start latency and session metrics, and (ii) Apple's Xcode Organizer, specifically the *Metrics → Launch Time* reports for released builds [2].

- **Comparison Window:** Metrics were collected for two builds of the application: a *baseline version* (before optimization) and an *optimized version* (after moving heavy initialization logic to a detached thread). Data for both versions was collected across comparable time periods to avoid seasonal or traffic-pattern biases.
- **User Cohort:** The dataset consists of more than **20k unique devices** and **2M cumulative sessions**, capturing a wide range of hardware capabilities and OS configurations. This diversity ensures robustness and generalizability of the results.
    - iOS version considered were from iOS 18.0 to iOS 26.0 and with different iPhone model (latest 3 models)
- **Launch Types Evaluated:**
    - **Cold Launch:** App starting from a terminated state.
    - **Warm Launch:** App starting while partially resident or cached in memory. Both launch types were included because they represent distinct user experiences and exhibit different performance characteristics.

6.1.2 Performance Metric Definition

To ensure clarity and reproducibility, the following definitions were used for performance measurement:

- **App Start Time (Cold/Warm):** The time interval between the user initiating the app (e.g., tapping the icon) and the completion of the initial rendering of the first interactive screen. This metric includes internal application initialization, system-level loading, and any synchronous work performed on the main thread.
    - Time from app icon tap → *application(_:didFinishLaunchingWithOptions:)* completed
- **Measurement Granularity:** The data sources provide aggregated launch-time statistics at the session level, including:
    - P75 start time
    - Distribution across user devices
    - Cold vs. warm start segmentation
- **Reason for Metric Choice:** App start time is a primary determinant of perceived responsiveness, and reducing it leads to improved user satisfaction. Since the optimization targeted startup logic, this metric directly represents the performance impact of the implemented change.

6.1.3 Optimization Approach

The optimization methodology builds directly upon the implementation changes described earlier in **Section 5 (Technical** Implementations**)**. Specifically, the synchronous heavy workloads originally executed inside *application(_:didFinishLaunchingWithOptions:)* were restructured according to the non-blocking concurrency model outlined in Section 5.1 and Section 5.2.

In the analysis phase, the objective was not to restate the implementation but to evaluate how those architectural changes influenced measurable launch-time performance.

6.1.3.1. Moving Blocking Tasks Off the Main Thread

As detailed in Section 5.1, tasks such as JSON deserialization, Core Data initialization, and analytics startup previously executed synchronously on the main thread. These tasks were migrated to a detached background thread, reducing main-thread occupancy during launch.

6.1.3.2. Concurrency-Based Parallel Initialization (Refer Section 5.2)

The final version adopted Swift concurrency (Task. detached, async let), enabling multiple independent startup operations to run in parallel. This significantly reduced the cumulative wall-clock time for background initialization.

6.1.3.3. Startup Resource Contention Reduction

A short delayed scheduling (5–10 ms), also described in Section 5.2, was used to avoid immediate CPU/disk contention with UI rendering. This contributed to smoother first-frame rendering and reduced measured launch time.

6.1.3.4. Why These Changes Produce Measurable Performance Gains

The launch-time improvements are attributable to three measurable effects:

1. **Main-thread unblocking**, leading to earlier UI availability
2. **Parallel execution** of tasks that were previously sequential
3. **Improved scheduler behavior** using detached background priorities

These architectural changes formed the foundation for the improvements measured in the subsequent sections.

6.2 Data Statistics:

This section summarizes the dataset used for evaluating the impact of the optimization techniques described earlier. The objective of this dataset characterization is to ensure that the analysis is statistically grounded, reproducible, and reflective of real-world device and OS diversity.

6.2.1 Overall Dataset Summary

This subsection outlines the scope and composition of the dataset collected from production devices. It describes user coverage, session volume, time span, and device/OS diversity to ensure that the evaluation captures representative behavior across a broad range of environments.

Table 1

Metric	Value
Total unique users	20K
Total sessions	2M
Time period	4 weeks pre-change 4 weeks post-change

Device range	iPhone18,1 - iPhone18,4 iPhone17,1 - iPhone17,4
OS versions	iOS 18.x → iOS 26.x

6.2.2 App Start time Stats

This subsection reports the key aggregated performance metrics collected before and after the optimization changes. Percentile-based measures (P75) are selected, as they provide a robust representation of typical user experience while reducing sensitivity to extreme outliers.

Metric	Before	After
P75 cold start	~ 2.2s	~ 900 ms
P75 Warm start	~ 1.9s	~790 ms

Table 2

6.4 Result Interpretation

This section interprets the empirical results obtained from the performance measurements. While Section 6.2 presents raw metrics, the following subsections explain what the changes imply for end-user experience, app responsiveness, and system behavior.

6.4.1 Observed Performance Gain

The collected data demonstrates a substantial performance improvement in both cold and warm app starts following the introduction of asynchronous initialization and main-thread load reduction.

- **Cold-start P75 latency** decreased from approximately **2.2 seconds to 0.9 seconds**, representing a **~59% reduction**.
- **Warm-start P75 latency** decreased from approximately **1.9 seconds to 0.79 seconds**, representing a **~58% reduction**.

These improvements were consistent across the majority of device and OS variants included in the dataset. Importantly, the benefit was observed not only in high-end models but also in older devices where CPU and disk-I/O contention tend to be more pronounced.
This confirms that the optimization approach generalizes well across heterogeneous device landscapes.

6.4.2 Why the Optimization Worked

The measured performance gains can be directly linked to the architectural improvements discussed in Section 5 and the optimization strategy described in Section 6.1.3. In particular, three system-level mechanisms explain the reduction in launch latency:

1. **Reduced Main-Thread Contention**
   Offloading synchronous operations (database initialization, configuration loading, analytics setup) eliminated the primary bottleneck that delayed first-frame rendering. The main thread was freed to focus exclusively on presenting UI.
2. **Parallelism Through Structured Concurrency**
   The use of async let allowed previously sequential tasks to run concurrently. This reduced total startup processing time, as independent tasks no longer blocked one another.
3. **Improved OS Scheduler Behavior**
   Executing heavy tasks in a detached background priority thread avoided competition with launch-critical UI tasks. The optional small delay (5–10 ms) further minimized early resource contention during the critical layout/rendering window.
4. **Elimination of Launch-Time I/O Spikes**
   Deferring expensive file reads and Core Data warm-ups reduced initial file system load, which is a common source of cold-start delays.

Collectively, these factors resulted in a cleaner separation between UI-critical and non-critical startup operations, leading to the significant improvements observed in the data.

6.5 Threats to Validity

Although the evaluation demonstrates clear performance improvements, certain limitations must be acknowledged to fully contextualize the results. These threats to validity outline factors that may influence the accuracy or generalizability of the findings.

6.5.1 Internal Validity

- **Concurrent production changes:** Other app or infrastructure modifications occurring during the data collection window may have influenced performance, though none were intentionally introduced.
- **Instrumentation overhead:** Metrics sourced from third-party SDKs and Apple's Organizer may exhibit minor sampling or aggregation biases.

6.5.2 External Validity

- **Usage-pattern variations:** Differences in user behavior (e.g., frequency of app closures, backgrounding patterns) may affect cold/warm start proportions.
- **Environmental factors:** Network speed, device storage state, and background system activity can influence launch performance and may vary across users.

6.5.3 Device and OS Fragmentation

Although the dataset includes a wide range of devices and OS versions, certain rare device types or older OS versions (below iOS 18) were underrepresented and may behave differently.

6.5.4 Data Collection Bias

Performance reporting depends on the accuracy of start-time measurement hooks implemented by SDKs. Any deviation from recommended Apple guidelines could introduce slight measurement variation.

7. DISCUSSION

Optimizing startup time has compounding benefits:

- Lower churn
- Faster first interaction
- Better Lighthouse/App Store metrics
- Reduced memory footprint

Technical analysis reveals most delays come from developer-caused synchronous work rather than system overhead.

Using structured concurrency and modular lazy loading significantly improves performance.

But it also comes with a caveat. Threading and swift concurrency is surely have proved to be effective but it required very careful thread management to avoid race conditions. It becomes more complicated specially with use of @MainActor and and global actor uses.

7. LIMITATIONS:

- Old iOS devices specially older than A12 will exhibit some smaller improvements.
- Swift-UI heavy apps might require some additional layout - level tuning along with above mentioned improvements.
- User of too much of actors will cause actor hop and will reduce the performance, so it needs a careful look.

8. CONCLUSION:

Optimizing an iOS app start time is a multi layer challenge that involves architecture, code design, dependency management, knowledge of tooling like "Time Profiler" and system-level behaviors. This paper introduce a system framework that can be used for a significant improvements across multiple devices.

This paper also demonstrates that inefficiencies in launch-time of an iOS app do not stem from a single bottleneck but from an multiple factors which include but not restricted to

- Framework loading.
- Static or shared instance initialization of heavy class.
- Blocking operation on main thread.
- UI composition costs.

The proposed approach of combining static code analysis, dependency slimming, strategic lazy initialization, background-thread–driven bootstrapping, and Time-profile instrumentation to check for result yielded significant improvements, reducing the start time by up to ~60% across tested devices. These results validate that systematic launch optimization has a positive impact on not only performance but also energy consumption, and user trust regarding the overall application quality.

These findings have implications beyond performance engineering: they drive architectural decisions directly. By prioritizing modular code boundaries, reducing unnecessary linking of frameworks, and isolating expensive initializers, developers can build systems that scale well as an application continues to evolve. Furthermore, with iOS increasingly embracing asynchronous-first paradigms through Swift Concurrency and lazy-loading mechanisms, proactive start-time engineering becomes even more critical.

Caution needs to be taken while integrating bloated sdks (analytics, adds and also screen profilers) as mentioned to this journal [8], as those can measurably degrade launch times.

Also care needs to be taken to avoid over optimizations, since that might cause additional energy costs if it cross to the OS or need additional computations[9],[10],[11].

9. ACKNOWLEDGMENTS

The author expresses deep gratitude towards the reviewers and editors of the International Journal of Computer Applications for their valuable feedback and guidance.

Along with that author acknowledges to all those iOS app engineers of Apple community who has provided documentation and conferences to share the knowledge with the world. WWDC technical sessions and the broader iOS performance engineering community for publishing research, tools and best practices that contributed to the methodology that is used in this study.

10. REFERENCES

[1] Apple Inc., Optimizing App Launch, in Apple Worldwide Developers Conference (WWDC), 2019. Available: https://developer.apple.com/videos/play/wwdc2019/423/

[2] Apple developer documentation, Reducing your app's launch time. Available: https://developer.apple.com/documentation/xcode/reducing-your-app-s-launch-time

[3] Apple Inc., Link fast: Improve build and launch times. In Apple Worldwide Developers Conference (WWDC), 2022 Available: https://developer.apple.com/videos/play/wwdc2022/110362/

[4] Apple Inc., Optimize SwiftUI performance with Instruments in Apple Worldwide Developers Conference (WWDC), 2025 Available: https://developer.apple.com/videos/play/wwdc2025/306/

[5] Apple Inc., Optimize your app's speed and efficiency conference Available: https://www.youtube.com/live/yXAQTIKR8fk?si=GuSmBzBr9RTn5jHW

[6] Apple Inc., Stack, Grids and Outlines in SwiftUI Available: https://developer.apple.com/videos/play/wwdc2020/10031/#:~:text=What%20I%20want%20is%20a,my%20VStack%20with%20a%20LazyVStack

[7] Apple Inc, App Start time: Past, Present and Future Available: https://nonstrict.eu/wwdcindex/wwdc2017/413/?t=577

[8] Exploring effects of Ad schemes on the performance cost of mobile phones. Available: https://dl.acm.org/doi/epdf/10.1145/3243218.3243221

[9] A Survey of performance Optimization for Mobile applications, Available: https://solar.cs.ucl.ac.uk/pdf/AppPerformanceOptimizationSurvey.pdf

[10] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in Proceedings of the 10th international conference on Mobile systems, applications, and services. ACM, 2012, pp. 155–168.

[11] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in Proceedings of the 10th international conference on Mobile systems, applications, and services. ACM, 2012, pp. 113–126.