Comparative Analysis of View Layout Performance in SwiftUI and Flutter on iOS Devices

Caleb Elson

School of Engineering, Computer and Mathematical Sciences

Auckland University of Technology, New Zealand

Abstract*— With the increasing need for efficient mobile applications, understanding performance differences between native and cross-platform frameworks is essential. This study investigates whether SwiftUI performs layout tasks faster than Flutter on iOS devices by measuring view layout times of equivalent views across eight devices. Results indicate that SwiftUI outperforms Flutter in three out of four views tested, with significant differences observed: the simplest view favoured SwiftUI, while the most complex view favoured Flutter. Future work includes exploring additional performance metrics and views, incorporating user experience evaluations, and extending comparisons to other frameworks and devices.

I. Introduction

The mobile operating systems market is predominantly dominated by iOS and Android, which together hold a combined market share of 99.27% as of 2024 [1]. As of September 2024, 61.66% of all network traffic was generated by mobile devices [2]. This ubiquitous use of mobile devices has created significant pressure for companies to develop applications for both iOS and Android platforms. To meet this demand efficiently, many developers have turned to cross-platform frameworks such as Flutter and React Native, which allow for a single codebase to be used across multiple platforms.

However, the choice between native and cross-platform development presents a critical decision for developers, particularly concerning application performance. Performance is a key factor that influences user experience, retention rates, and overall satisfaction. Native frameworks like SwiftUI are specifically designed for their respective platforms, potentially offering optimised performance and seamless integration with the operating system. In contrast, cross-platform frameworks aim to provide broader reach but may introduce performance overhead due to abstraction layers.

While numerous studies have compared various metrics of cross-platform frameworks to native development, gaps remain in the current knowledge base. The rapid advancement in mobile technology means that studies can quickly become outdated, necessitating continual research to keep pace with evolving frameworks. Additionally, many existing studies focus exclusively on cross-platform frameworks or, if native frameworks are included, often limit their scope to Android, leaving a gap in the literature regarding iOS native frameworks like SwiftUI.

This research aims to address these gaps by conducting a performance comparison between Flutter and SwiftUI on iOS devices. By focusing on the time taken to lay out equivalent views, this study seeks to provide insights into the performance implications of choosing between a native framework and a cross-platform framework on iOS. This is particularly relevant for developers and companies looking to optimise development efficiency and resource allocation.

A. SwiftUI

SwiftUI, first introduced in 2019, is a declarative framework developed by Apple for building user interfaces across their platforms, including iOS and iPadOS. SwiftUI integrates with Xcode, Apple's integrated development environment (IDE) for macOS, and, together with the Swift programming language – also developed by Apple – allows developers to create native applications for iOS, iPadOS, and other Apple platforms. SwiftUI offers a modern approach to UI development, emphasising simplicity and interoperability with Apple's ecosystem. Its declarative syntax enables developers to write less code and provides real-time previews, enhancing productivity and reducing development time.

B. Flutter

Flutter is a UI software framework created by Google, initially released in 2017. Flutter uses the concept of "widgets," which employ the declarative UI paradigm to describe the implementation and design of UI elements. Widgets can contain other widgets, allowing developers to build views of varying complexity. Flutter applications

are written using the Dart programming language, also developed by Google. Flutter is cross-platform, enabling a single codebase to be used on multiple platforms, including iOS, Android, macOS, Windows, and the web. Being cross-platform, Flutter is an appealing option for developers by reducing development effort and increasing reach simultaneously.

II. Literature Review

This section provides a background on research comparing the performance between cross-platform and native mobile development frameworks. This research specifically focuses on Flutter and SwiftUI, which both rank among the top 20 most-used technologies of 2023 according to Stack Overflow's annual survey [3]. Each is the highest-ranked framework in its respective category—Flutter in mobile cross-platform frameworks and SwiftUI in native iOS frameworks.

A 2019 study by Biørn-Hansen *et al.* provides a comprehensive review of the state of research into cross-platform mobile development [4]. This study identified five main development approaches: hybrid, interpreted, cross-compiled, model-driven, and progressive web apps. They found that while some areas were very well-researched, such as the software platform features of hybrid and interpreted applications, there was a need for more studies that measure technical performance, as much of the current body of knowledge was described as conceptual and descriptive. Flutter was specifically mentioned as an area in need of research, and they also highlighted the problem of rapid evolution in the application development market—much of the research frequently cited as state-of-the-art was instead found to be dated and of questionable relevance.

An example of how quickly the state of the art progresses can be observed in a 2012 study by Palmieri *et al.*, which studied frameworks that were in common use at the time, such as DragonRad and MoSync [5]. Neither of these frameworks has been updated in more than 10 years [6], while the currently most popular cross-platform frameworks, Google's Flutter and Meta's React Native, were released in 2017 and 2015, respectively [7]. Native UI frameworks have been progressing as well, with Apple's native SwiftUI releasing in June 2019 [8]. This underscores the continual need for new research into the performance of various mobile development frameworks.

Nanavati *et al.* address several methods to improve the performance of applications developed in Flutter [9]. Their findings show that substantial improvements can be achieved in build time, render time, and shader file size by switching from a StatelessWidget to a StatefulWidget. Additionally, they identified the potential impacts that memory leaks can have on the performance of a Flutter-based application. Most relevant to this research, they found that debug mode was slower than production mode, necessitating that performance profiling be conducted on real devices.

On the native iOS side, a comparison between SwiftUI and UIKit found that UIKit performed better in nearly all tested cases, with views having fewer than 32 components being approximately 25% faster [10]. While it is worth noting that SwiftUI is generally not the faster of the iOS native frameworks, it is clearly positioned to take over as the default UI framework. A search of talks given at Apple's annual Worldwide Developer Conference over 2022, 2023, and 2024 that focused on SwiftUI or UIKit revealed 33 for SwiftUI and only 7 for UIKit [11]. Additionally, there is a clear lack of studies on SwiftUI specifically—an intentionally broad search of EBSCO journal articles performed in October 2024 containing the term "SwiftUI" returned only 28 results.

The overhead introduced by using cross-platform mobile development frameworks was studied by Biørn-Hansen *et al.* in 2020 [12]. This study included five cross-platform frameworks: Ionic, React Native, NativeScript, Flutter, and MAML/MD2, with native Android using Java as a baseline. Benchmarks were developed to test how each of these performed while accessing commonly used features: retrieving data from the accelerometer, adding a contact, accessing a PNG image in the file system, and retrieving the longitude and latitude from the device's GPS sensor or network-based positioning system. While it found that generally there is a performance decrease when using cross-platform frameworks compared to native ones, there were metrics where the cross-platform counterpart exceeded native performance. This study focused on Android and suggested that a similar study could be conducted on iOS in the future. It also focused on the capabilities of the frameworks studied rather than on their abilities to render user interfaces.

Jagiello [13] and Wu [14] each built applications in both React Native and Flutter and made observations regarding frame rate and specifically dropped frames by testing them on Android devices. Biørn-Hansen et al. conducted a similar test but used more cross-platform frameworks and tested on both Android and iOS devices [15]. They found that measuring frames per second (FPS) was not particularly insightful for the kinds of transitions they were testing and found that differences in profiling between the Android and iOS platforms made comparisons difficult or misleading. While this study was mostly focused on measuring transitions, they hypothesised that FPS is insufficient for measuring user experience. Biørn-Hansen *et al.* also noted issues with using the standard profiling tools for iOS to measure the kind of frame-dropping issues studied by Jagiello and Wu.

An evaluation of code complexity, CPU usage, and user-reported assessments of the look and feel between Flutter, native Android using Kotlin, and native iOS using Swift found performance to be similar between Flutter

and the native solutions [16]. However, it noted that since the testing was performed by humans and not scripted, there was potential room for error in the results.

III. Research Question

As seen in the Literature Review section above, there is a gap in the current research for mobile frameworks, and a study focused on comparing cross-platform performance to native iOS performance, especially using SwiftUI, is warranted.

A. *Research Question 1: Does SwiftUI perform layout tasks faster than Flutter when rendering equivalent views on iOS devices?*

This question seeks to determine whether the native iOS framework, SwiftUI, offers superior performance in view layout times compared to the cross-platform framework, Flutter. By measuring the time taken to lay out equivalent views in both frameworks, the study aims to provide empirical data on whether native development offers a performance advantage over cross-platform solutions on iOS devices.

B. *Research Question 2: How do different types of views (simple vs. complex) affect the performance comparison between SwiftUI and Flutter?*

This question explores how the complexity of views influences the performance of each framework. By testing both simple views (such as a single text element) and complex views (such as creating thousands of circles on screen at once), this study examines whether performance differences are consistent across varying levels of view complexity or if certain types of views amplify the performance disparities between SwiftUI and Flutter.

C. *Research Question 3: What is the impact of device hardware and iOS version differences on the performance of SwiftUI and Flutter applications?*

Understanding how hardware specifications and iOS versions affect performance is crucial for developers targeting a wide range of devices. This question investigates whether older devices or different iOS versions impact the performance differences between the two frameworks. By testing on multiple devices released in different years and running various iOS versions, this study assesses the consistency of performance across the hardware and software.

D. *Scope*

This study focuses exclusively on measuring the time taken to lay out views in SwiftUI and Flutter applications on iOS devices. It does not consider other performance metrics such as frame rates, CPU and memory usage, or network-related performance. The research targets devices running iOS 16.0 or later. While the views are made to be highly similar, they are intentionally *equivalent* and not *identical* to give a more realistic measure of how performance varies between a typical Flutter and SwiftUI application.

IV. Research Design

Given that native iOS versus cross-platform performance is constantly evolving, this study focuses on building common views that leverage each of the studied frameworks' designs and testing them in real-world scenarios. Two mobile applications were developed and deployed on a variety of iOS devices: one built entirely with Flutter, and the other with SwiftUI. The time required to lay out the views was measured.

Although the rendering pipelines differ between SwiftUI and Flutter, as will be discussed below, the chosen methodology attempts to account for both the differences in view lifecycles and each platform's best practices for creating views. The only requirements for the devices were the ability to run iOS applications and to be running at least iOS 16.0. This ensures that the research is relevant to the current state of mobile application development and because the SwiftUI application uses APIs that require iOS 16.0 or later.

The initial design for this study involved a loop of multiple views that would take approximately five minutes. However, feedback from testers necessitated scaling this back; put simply, testers were unwilling to spend that much time running a testing loop on their phones. Additionally, early feedback indicated that the most strenuous test, the Squares View test, did not complete even after several minutes of waiting. While the goal of this research was to sufficiently stress the system to differentiate performance between native and cross-platform frameworks, the variety of hardware—including devices from as early as 2017—necessitated a design that would perform

adequately across all tested devices. Therefore, the Squares View test was modified to perform better across all devices.

The literature review highlighted the need to use real devices for testing Flutter. Additionally, Flutter was found to perform better when using StatefulWidget instead of StatelessWidget, so StatefulWidget was used for this study. SwiftUI was chosen as the native framework instead of UIKit, both because of its position as the preferred native UI framework and due to a clear lack of current research on SwiftUI. Measuring FPS was found to be problematic; therefore, an approach was devised that simply measures the time each framework takes to lay out its views, avoiding concerns about exactly matching elements such as transition times. Finally, by scripting the loop itself instead of relying on user interaction, a repeatable process was created.

The final design consists of two applications, each running through the same loop of four views, five times each, taking approximately 30 to 40 seconds in total. Before each view's rendering pipeline is started, the current time is recorded, and once the layout is finished, the current time is recorded again. The difference between these times is used for analysis below. Everything happens locally, so network connection is not a factor. Although this setup is somewhat unrealistic—as nearly every popular mobile application relies on loading data from a network connection—it is necessary because this research focuses on comparing view layout time, and network connection variability would introduce unwanted noise. While the views were made to be similar, they are intentionally not identical. SwiftUI and Flutter both have their own conventions on how views and widgets, respectively, should appear, and this study focuses on the time to lay out *equivalent views* rather than *identical views*.

All application code and collected data can be found on GitHub (see Appendix A).

A. Overview of SwiftUI View Lifecycle

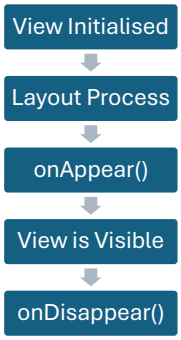

Fig. 1 SwiftUI view lifecycle

When creating a view, SwiftUI provides only three hooks into the lifecycle: onAppear(), which is used to add an action that will be performed before the view appears; task, which also performs an action but asynchronously; and onDisappear(), which is for adding an action after the view disappears [17]. As this study focuses on the time to lay out individual views, only two steps are relevant: right before the view is initialised, and onAppear().

In the SwiftUI application, there is a home view named ContentView, which initialises every subsequent view. A timestamp is saved immediately before the view is initialised, and that timestamp is then passed to the view to be tested. Once the child view is about to be shown to the user, onAppear() is called and a second timestamp is saved. The child view then saves both values to the list of timer values and finally simply dismisses itself, which notifies ContentView. ContentView then adds the next child view to the stack for testing, until five full loops of the four test views are completed, at which point the user is presented with the final view that allows for the test data to be sent.

The onAppear() method is not guaranteed to be called in any specific frame; Apple's documentation notes simply that the action will be completed before the first rendered frame appears [17]. This is partially due to SwiftUI's approach to UI development, which is built around using a collection of individual component views to present to the user. For example, a table view consists of a table containing multiple rows, each of which has an image and a label. Each of these elements is an individual view from SwiftUI's perspective, and onAppear() may be called on any of them. Further analysis reveals that onAppear() is called when the view has finished its layout**, but** before rendering has been completed [18]. Since onAppear() is not called when a view is updated, all four of the test views and the test in general were designed to perform a single layout and render of a view, then dismiss that view and move on to the next one.

B. Overview of Flutter View Lifecycle

Flutter is built around the concept of widgets, which are the fundamental building blocks of a Flutter application's user interface. The two foundational widgets are StatelessWidget and StatefulWidget.

A StatelessWidget is immutable and is generally used for static, non-dynamic views, while a StatefulWidget is mutable and is most useful when a view is expected to change over time. Following the literature review above, the Flutter application used in this study was built entirely with StatefulWidget.

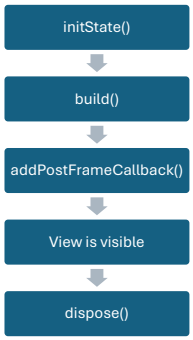

Fig. 2 Flutter view lifecycle

For this research, two stages are relevant: initState() and addPostFrameCallback(). An initial timestamp is recorded just before the view is initialised, after which initState() is called, and subsequently, the build()method begins. The addPostFrameCallback() is then called on the instance of the binding of the created widget, directly after the main rendering pipeline has been flushed. This makes it a suitable counterpart to SwiftUI's onAppear() and is used to measure the time to finish the view layout in this study, as it is called after the layout has been completed.

In the Flutter application, the view is not dismissed at this point, as doing so led to no view ever being shown to the user. This created an issue where the tester did not know if the application was doing anything and assumed it was broken as it appeared to be frozen for 30–40 seconds until all the tests had run. To solve this issue, the Flutter test views are not dismissed until they are actually visible. This is achieved using VisibilityDetector, a widget built by Google that enables a callback once a widget's visibility has changed [17].

C. Devices Tested

TABLE I
Devices tested, ordered by device release date

Release Year	**Device**	**iOS Version**
2017	iPhone X	16.7.10
2020	iPhone 12	17.6.1
2020	iPhone 12 mini	18.0.1
2020	iPhone 12 Pro Max	17.7
2021	iPhone 13 Pro	18.0.1
2021	iPhone 13 Pro Max	18.0.1
2022	iPhone 14 Plus	17.6.1
2022	iPhone 14 Pro Max	17.6.1

These eight devices were used to test both the SwiftUI and Flutter applications. No special setup or conditioning was used; users were simply asked to download the applications from TestFlight, Apple's service for over-the-air installation of mobile applications for testing [19]. Testers ran the loop, requiring a single button press, and then sent the data.

D. Table View

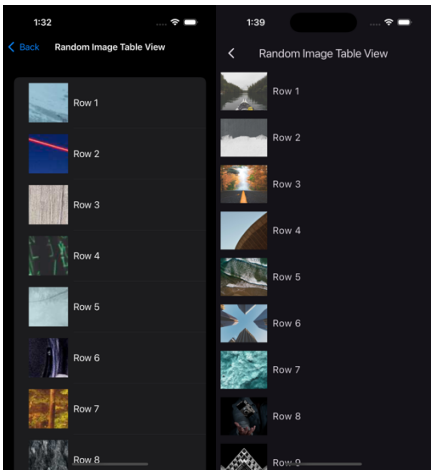

Fig. 3 Table Views in SwiftUI and Flutter, respectively

Table views are among the most common views across all mobile apps and are therefore relevant to test in this study. For this test, a basic table view with minimal customisation was built. Each row contains the row index as a label and a randomly selected image from a royalty-free list [20], which is added at the beginning of the row. The design of the tables and rows is not matched between the two frameworks; instead, we used the native table view implementations provided by SwiftUI and Flutter.

The Table View is expected to load quickly on both frameworks, allowing any observed differences to reflect the overhead introduced by each framework rather than the complexity of the view itself. While 50 rows are programmed to be created, both SwiftUI and Flutter only load rows as needed. Given the ubiquity of table views across mobile applications, it is expected that this will be a highly optimised process.

E. Circles View

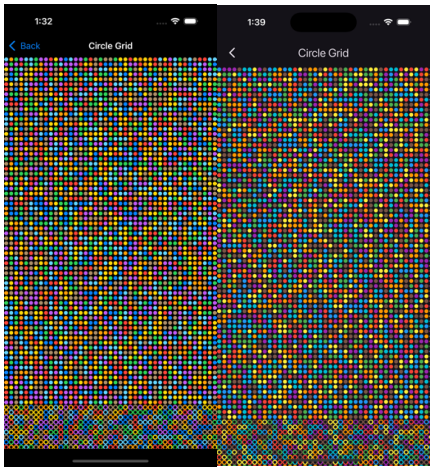

Fig. 4 Circles Views in SwiftUI and Flutter, respectively

Circles View is a step up in complexity from Table View, loading 5,000 circles in 100 rows and 50 columns. The size of each circle is calculated to ensure that all 5,000 circles fit on any screen size. Efforts were made to ensure that the implementation of this view is as similar as possible between SwiftUI and Flutter. It uses a common set of colours from which to randomly select, shared between both frameworks. Both implementations manually create each circle and lay them out in almost identical ways, filling the first 3,500 circles and leaving the rest only outlined.

Both SwiftUI and Flutter lay out the circles using their implementations of Canvas, which is designed to improve performance when drawing views that are not primarily composed of text and do not need to be

interactable[17], [21]. A rectangle is created and placed at a calculated coordinate based on the number of circles to be drawn, the size of the screen, and the current index. Inside the rectangle, a circle is drawn, and then the colour and whether to completely fill it are determined. The only substantial difference between these views is in the UI windowing itself—SwiftUI includes a safe area for the taskbar at the bottom and has a smaller navigation bar at the top.

This view serves the purpose of testing each framework with a relatively complex view to load. However, due to its simple implementation, it is more similar between the two frameworks compared to the other, more complex view, Squares View. This allows for a more straightforward comparison between Flutter and SwiftUI.

F. Squares View

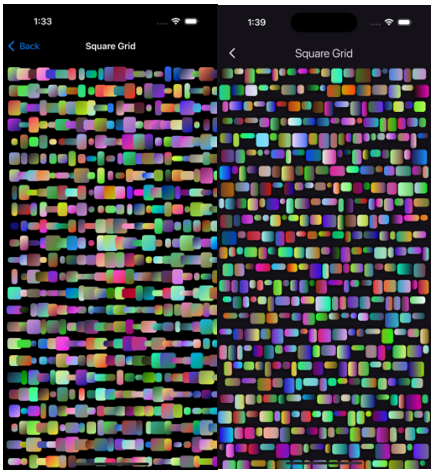

Fig. 5 Squares Views in SwiftUI and Flutter, respectively

The most complex view in this test, the Squares View, was designed to take longer to lay out than any of the other views so that any difference in time would be more noticeable. While an effort was made to implement them as similarly as possible, this proved to be difficult due to the differences in how each framework lays out its views. This view was moved from being the first view to being the third in the loop as it can take multiple seconds before the view is visible and testers simply thought the application was broken when it was the first view and it took multiple seconds to react at all.

An individual block is created in SwiftUI using a Rectangle view, while Flutter uses a Container widget. The width and height are both randomly chosen between 10 and 30 points. The colour of the block is a linear gradient that is chosen as random values for red, green, and blue (RGB)—in Flutter this is a random integer between 0 and 255, while in SwiftUI this is a random double between 0 and 1, inclusive.

This block is placed into the layout, with SwiftUI using a LazyVGrid inside a ScrollView, and Flutter using a List view inside a Wrap. Both implementations are scrollable, vertical views that create child views only as needed [15], [19]. SwiftUI was instructed to create 50 columns, as specifying the number of columns is required for LazyVGrid, while Flutter was allowed to lay out the blocks however it wants, with a margin of 1 point on all sides. It is worth noting that SwiftUI's implementation of LazyVGrid allows for elements to overlap, as can be seen above, while Flutter's Wrap dynamically lays out the blocks until it runs out of space in a line and then simply moves to a new one. In practice, this difference should be largely mitigated by the limited space—the blocks are randomly sized, but their sizes are chosen without respect to the maximum width of the view. This means that approximately the same number of visible blocks should be laid out in each row in both apps. Indeed, the number per row in the screenshots above is similar, and since both SwiftUI and Flutter's implementations only load blocks as needed, the rest of the unseen columns for SwiftUI's LazyVGrid should not have an impact on the layout time. It is still worth noting that the SwiftUI view will end up with more total blocks loaded. This is because, even though the unseen blocks should not have a significant impact on performance, more blocks are loaded per row as blocks are able to be partially visible on the right side.

As noted above in the introduction to this section, this view was intended to be the most **time-consuming** view and needed to be scaled back during testing. Originally, 500,000 blocks were created, but the iPhone X was so slow to load this view that this test was substantially scaled back. The final block count is 25,000, which is still a sufficiently complex view that some devices need multiple seconds each cycle, but it will fully load within a reasonable amount of time on every tested device.

The expectation from this view is that SwiftUI will need to load more blocks than Flutter will, although precisely how many more will vary based on screen size, and this will likely mean that it will be the **worst-performing** in comparison to the Flutter version of all the views.

G. Test View

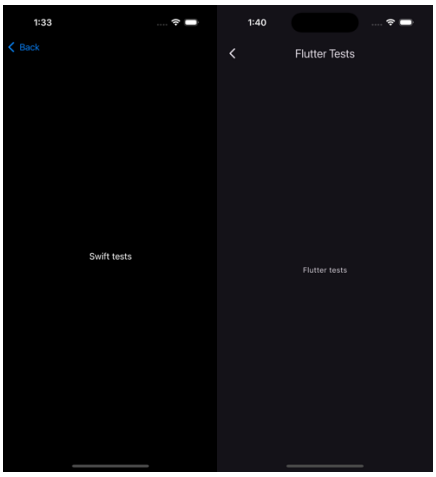

Fig. 6 Test Views in SwiftUI and Flutter, respectively

The final view is the simplest: a single text element, centred horizontally and vertically. This view serves as a baseline to test how long it takes for SwiftUI and Flutter to lay out a single, simple view or widget, respectively, and is useful as it is the easiest to replicate. Unlike some of the other views that have necessary implementation differences, this view was chosen to reduce as many of the implementation details as possible and to provide the clearest possible picture of the overhead introduced by each of the frameworks when creating the simplest possible view.

V. Findings, Analysis, and Discussion

A. Table View

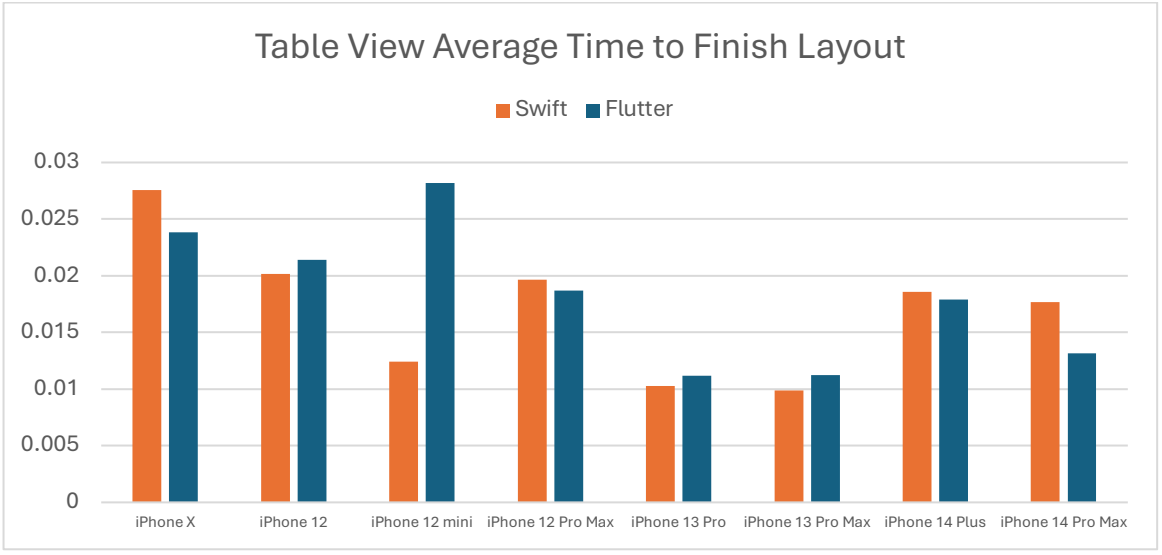

Fig. 7 Average time to finish layouts for all five runs of the Table View, lower is better

Figure 6 above shows a mostly expected result: performance across a range of devices remains consistently close between SwiftUI and Flutter, with most of the per-device differences being in the range of single digit milliseconds – the iPhone 14 Plus measured 0.00067 seconds, or 0.67 milliseconds. While the iPhone 14 Pro Max sees a relatively large advantage of Flutter over SwiftUI at 4.56 milliseconds, clearly the largest difference was on the iPhone 12 mini, with a difference of 15.79 milliseconds. This is curious – the iPhone 12 mini and the iPhone 12 Pro Max have identical processors [22].

TABLE II
Time to load Table View in Flutter

iPhone X	iPhone 12 mini
0.040613	0.041689
0.018695	0.019315
0.018877	0.018836
0.020346	0.045629
0.020561	0.015546

Digging further into the data, the issues is clearer: Table II shows the iPhone 12 mini had two high results above 0.04 seconds out of the five runs. The iPhone X, as a comparison, had only a single result over 0.04, while most of both sets of runs were at or below 0.02. This test clearly has some margin of error, and the iPhone 12 mini's result is thus likely to be inside of it.

Ultimately, the differences here are negligible: averaging over all devices, between SwiftUI and Flutter, the measured time difference is a mere 1.17 milliseconds in favour of SwiftUI. This is a 3.33% advantage for SwiftUI, which while measurable, is not notable. As mentioned in the Research Design section above, this aligns with expectations – it is a simple table view, and table views are so ubiquitous that it is expected that both SwiftUI and Flutter would have spent time optimising them to be highly performant.

B. Circles View

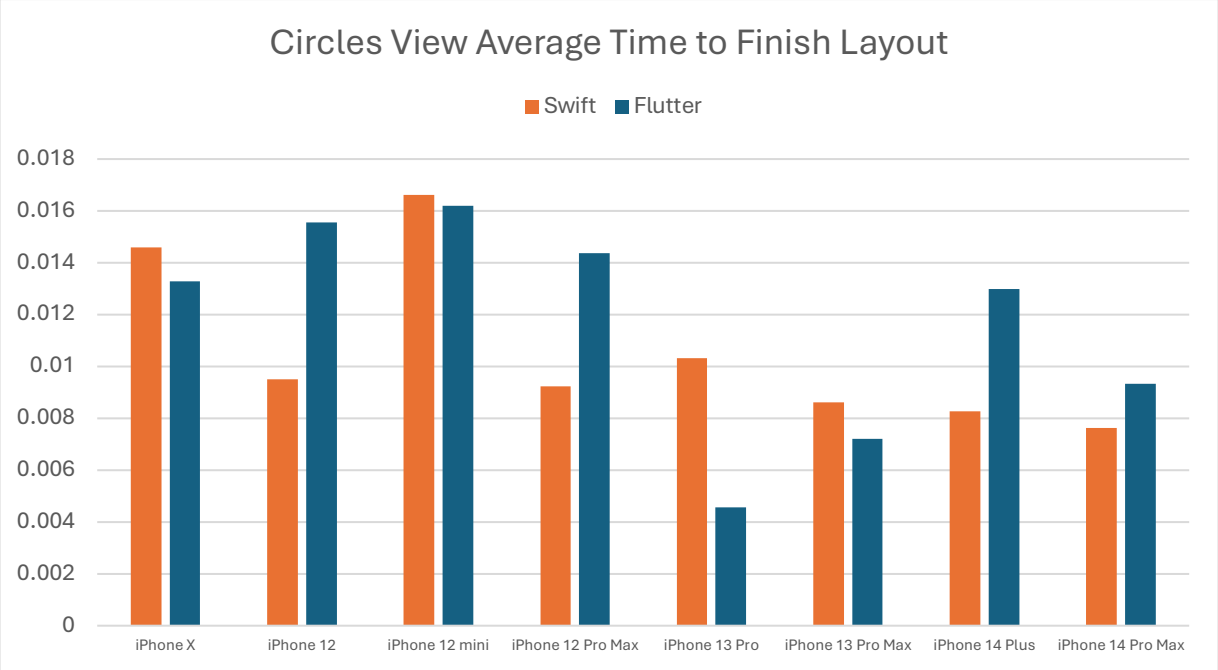

Fig. 8 Average time to finish layouts for all five runs of the Circles View, lower is better

Figure 7 shows the average time for each device in the Circles View test. Note that these are even smaller differences, with every device averaging less than 0.02 seconds.

TABLE III
Time to load Circles View in SwiftUI and Flutter

iPhone 12	iPhone 12	iPhone 12 Pro Max	iPhone 12 Pro Max	iPhone 14 Plus	iPhone 14 Plus
SwiftUI	Flutter	SwiftUI	Flutter	SwiftUI	Flutter
0.010138	0.010288	0.011207	0.012235	0.008915	0.012978
0.010224	0.017579	0.008568	0.015453	0.008224	0.012397
0.010008	0.016134	0.008508	0.014456	0.008236	0.012910
0.008457	0.017288	0.009294	0.014446	0.008010	0.013389
0.008677	0.016526	0.008547	0.015230	0.007953	0.013238

Unlike in the Table View above, Table III shows that the three outlier Flutter devices were just generally higher. In fact, only one of the Flutter results is faster than any of the counterpart SwiftUI results. This shows a much more definitive trend and suggests that for this view SwiftUI may simply be faster across the devices tested.

TABLE IV
Time to load Circles View in SwiftUI

iPhone 13 Pro
Flutter
0.024226
0.007396
0.006999
0.006378
0.006594

Meanwhile, the iPhone 13 Pro is the lone device that shows noticeably faster Flutter results, but just like for the Table view, this appears to be a single outlier run.

Ultimately, the results for this view are closer than the Table View – 1.10 milliseconds overall, in favour of SwiftUI, but the percentage difference was larger, with SwiftUI taking a 4.91% advantage. While SwiftUI has an advantage in the first two views tested, the difference is unlikely to sway the decision when choosing between the two. These views are intentionally simplified, however, and similar but more complex views may show a larger advantage.

C. Squares View

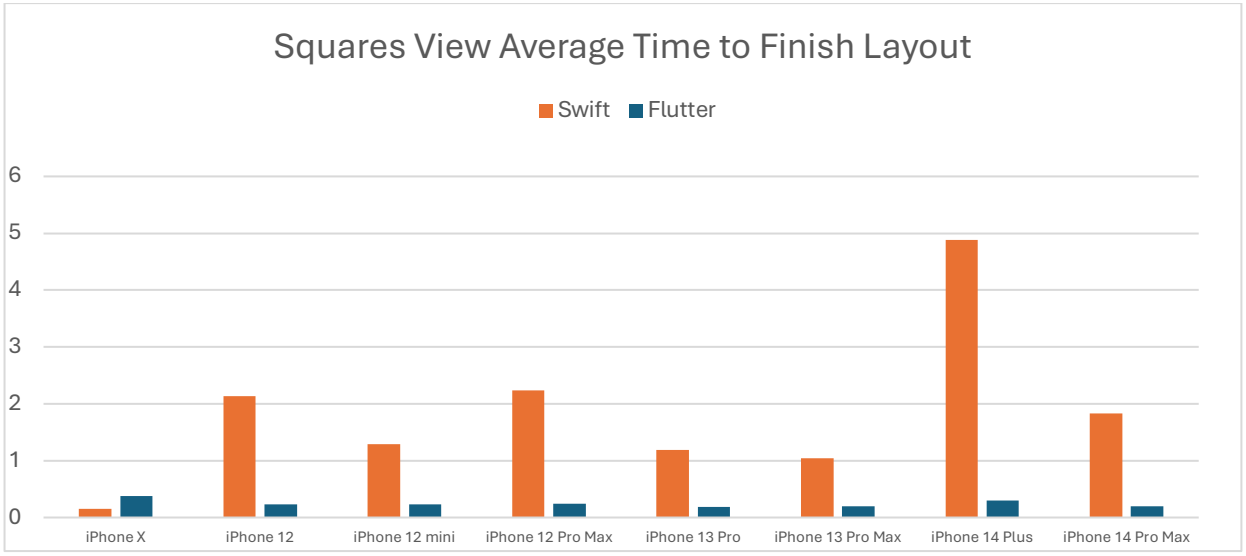

Fig. 9 Average time to finish layouts for all five runs of the Squares View, lower is better

Figure 8 shows a substantial difference, with every device, except one[1], having a large advantage for Flutter. Seven of the devices took over one second in SwiftUI, on average, while the slowest on the Flutter side of those devices was 0.3 seconds. The iPhone 14 Plus in particular fared poorly:

TABLE V
Time to load Squares View in SwiftUI

iPhone 14 Plus

[1] The iPhone X is the only device tested running a version of iOS 16, and while outside of the scope of this study, running this test in an Xcode simulator of an iPhone X running iOS 16 gives similar results to those listed. iOS 17 appears to have made a change that substantially increases the time to lay out this Squares View, with iOS 16 being noticeably faster than Flutter. This is not deemed relevant to this study, as this worse performance carries over to iOS 18 and is thus the more relevant result going forward.

4.779448
4.798462
4.943729
4.941125
4.967490

With an average of 4.88 seconds across all five runs, the iPhone 14 Plus was clearly the worst performer, and helped create an overall average of 1.60 seconds advantage for Flutter, a 76.57% advantage. As stated in the Research Design section, it was anticipated that SwiftUI would be at a disadvantage in this view, as on average it was loading more squares every time than Flutter due to the way each was implemented. This difference is larger than was expected, however, and suggests that there may be advantages within Flutter's layout pipeline that have led to the superior performance.

D. Test View

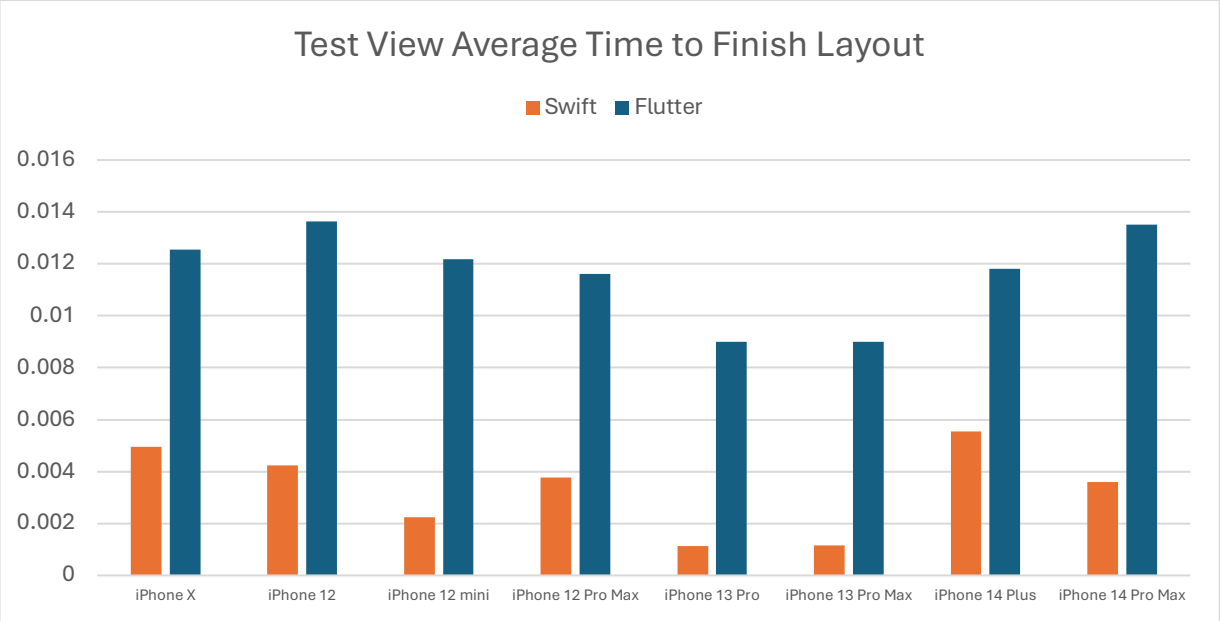

Fig. 10 Average time to finish layouts for all five runs of the Test View, lower is better

In stark contrast to the previous view, Figure 9 shows a very large performance for SwiftUI. Recall that this is the simplest view – a single label in the middle of the screen and was chosen as a pure comparison without needing to worry about implementation. The best performing devices for SwiftUI were the iPhone 12 mini, the iPhone 13 Pro, and the iPhone 13 Pro Max, which also happen to be the only devices running iOS 18, suggesting continued improvements to SwiftUI. While the absolute difference is small – the average time across all devices gave SwiftUI an 8.33 millisecond advantage – the percentage difference is nearly as large as the Squares View above: 55.57% in favour of SwiftUI. Few mainstream mobile applications have a screen this simple, so extrapolating potential performance differences is difficult, but this was an unexpected advantage gained by SwiftUI.

VI. Conclusions, Limitations, and Future Work

A. Conclusion

The primary objective of this study was to investigate the performance differences between SwiftUI and Flutter on iOS devices, specifically to determine whether native SwiftUI performs layout tasks faster than Flutter when rendering equivalent views (Research Question 1). The results indicated that, of the four views tested across eight devices, SwiftUI performed better on average in three. This suggests that SwiftUI generally offers faster layout times compared to Flutter in the contexts examined.

Regarding how different types of views affect the performance comparison between SwiftUI and Flutter (Research Question 2), the study found mixed results. The most substantial differences between the two frameworks' performances were observed in the most complex and the simplest views. The data also suggest that the iOS version specifically impacts performance (Research Question 3).

The future of both frameworks is promising. While cross-platform frameworks like Flutter offer clear advantages for applications targeting both Android and iOS, this study and others, such as by Biørn-Hansen et al. [12], suggest that native development can yield performance benefits. Developers must balance the trade-offs between development efficiency and application performance when choosing the appropriate framework for their projects.

B. Limitations

This research is limited in scope to specifically comparing the layout time of the four views discussed above in SwiftUI and Flutter, and does not consider factors such as frame rates, CPU usage, or network connectivity. Furthermore, it focuses exclusively on the chosen frameworks – SwiftUI and Flutter – and does not study other cross-platform frameworks such as React Native. Additionally, the study is limited by its focus on iOS, specifically recent versions of iOS. User experience was not assessed in this study but could be included in future research, potentially using questionnaires or interviews to determine whether cross-platform frameworks integrate adequately with native frameworks from a design and user experience perspective. While the four views tested were carefully chosen to represent a wide variety, numerous other views were not tested, and therefore, the accuracy of extrapolating these results to all views is uncertain.

C. Future Work

There is much left unexplored regarding performance metrics—objective measurements like frames per second and total time to display the view on screen, as well as subjective differences in user experience between native and cross-platform framework designs. Additional frameworks could also be studied, both cross-platform ones like React Native and Ionic, and iOS native UIKit along with native Android as well. The findings from the Test View above suggest improvements to SwiftUI in the recently released iOS 18, highlighting an issue raised in the literature review: studies on various frameworks quickly become outdated as the frameworks are continually updated and improved. This means that even well-conducted previous work can be re-examined to test the current state of the art.

References

[1] ‘Mobile OS market share worldwide 2009-2024’, Statista. Accessed: Oct. 24, 2024. [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[2] ‘Desktop vs Mobile vs Tablet Market Share Worldwide’, StatCounter Global Stats. Accessed: Oct. 24, 2024. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

[3] ‘Stack Overflow Developer Survey 2023’, Stack Overflow. Accessed: Oct. 24, 2024. [Online]. Available: https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023

[4] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, ‘A Survey and Taxonomy of Core Concepts and Research Challenges in Cross-Platform Mobile Development’, *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–34, Sep. 2019, doi: 10.1145/3241739.

[5] M. Palmieri, I. Singh, and A. Cicchetti, ‘Comparison of cross-platform mobile development tools’, in *2012 16th International Conference on Intelligence in Next Generation Networks*, Oct. 2012, pp. 179–186. doi: 10.1109/ICIN.2012.6376023.

[6] *MoSync/MoSync*. (Sep. 17, 2024). C. MoSync. Accessed: Oct. 23, 2024. [Online]. Available: https://github.com/MoSync/MoSync

[7] ‘Cross-platform mobile frameworks used by global developers 2023’, Statista. Accessed: Oct. 23, 2024. [Online]. Available: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/

[8] A. Inc, ‘SwiftUI - Latest News - Apple Developer’. Accessed: Oct. 23, 2024. [Online]. Available: https://developer.apple.com/news/?id=06032019b

[9] J. Nanavati, S. Patel, U. Patel, and A. Patel, ‘Critical Review and Fine-Tuning Performance of Flutter Applications’, in *2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI)*, Jan. 2024, pp. 838–841. doi: 10.1109/ICMCSI61536.2024.00131.

[10] B. Ronneling, ‘Performance analysis of SwiftUI and UIKit : A comparison of CPU time and memory usage for common user interface elements’, *En Jämf. Av CPU-Tid Och Minnesanvänding För Vanliga Användargränssnittselement*, 2023.

[11] ‘All Videos - Videos - Apple Developer’. Accessed: Oct. 24, 2024. [Online]. Available: https://developer.apple.com/videos/all-videos/

[12] A. Biørn-Hansen, C. Rieger, T.-M. Grønli, T. A. Majchrzak, and G. Ghinea, ‘An empirical investigation of performance overhead in cross-platform mobile development frameworks’, *Empir. Softw. Eng.*, vol. 25, no. 4, pp. 2997–3040, Jul. 2020, doi: 10.1007/s10664-020-09827-6.

[13] J. Jagiello, *PERFORMANCE COMPARISON BETWEEN REACT NATIVE AND FLUTTER*. 2019. Accessed: Oct. 24, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-163190

[14] W. Wu, ‘React Native vs Flutter, cross-platform mobile application frameworks’.

[15] A. Biørn-Hansen, T.-M. Grønli, and G. Ghinea, ‘Animations in Cross-Platform Mobile Applications: An Evaluation of Tools, Metrics and Performance’, *Sensors*, vol. 19, no. 9, Art. no. 9, Jan. 2019, doi: 10.3390/s19092081.

[16] M. Olsson, *A Comparison of Performance and Looks Between Flutter and Native Applications : When to prefer Flutter over native in mobile application development*. 2020. Accessed: Oct. 24, 2024. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:bth-19712

[17] A. Inc, ‘SwiftUI Overview - Xcode - Apple Developer’. Accessed: Oct. 24, 2024. [Online]. Available: https://developer.apple.com/xcode/swiftui/
[18] ‘Timing of onAppear Invocation | Fatbobman’s Blog’, fatbobman.com. Accessed: Oct. 24, 2024. [Online]. Available: https://fatbobman.com/en/posts/onappear-call-timing/
[19] A. Inc, ‘TestFlight’, Apple Developer. Accessed: Oct. 24, 2024. [Online]. Available: https://developer.apple.com/testflight/
[20] ‘ServiceStack/images: Collection of free images and resources for Websites and Apps’. Accessed: Oct. 23, 2024. [Online]. Available: https://github.com/ServiceStack/images
[21] ‘Flutter documentation’. Accessed: Oct. 24, 2024. [Online]. Available: https://docs.flutter.dev
[22] ‘What processor or processors do the iPhone models use?: EveryiPhone.com’. Accessed: Oct. 24, 2024. [Online]. Available: https://everymac.com/systems/apple/iphone/iphone-faq/iphone-processor-types.html

APPENDIX A

The source code and relevant data can be found at: https://github.com/calebelson/TimeTest