Linköpings universitet/Linköping University | IDA
Examensarbete på grundnivå, 16hp | Innovativ programmering
HT 2023 | LIU-IDA/LITH-EX-G--23/054--SE

React Native vs. Flutter: A performance comparison between cross-platform mobile application development frameworks

Gustav Tollin
Marcus Lidekrans

Handledare: Anders Fröberg
Examinator: Erik Berglund

LINKÖPINGS UNIVERSITET

Upphovsrätt

Copyright

React Native vs. Flutter: A performance comparison between cross-platform mobile application development frameworks

Marcus Lidekrans
marli691@student.liu.se

Gustav Tollin
gusto027@student.liu.se

ABSTRACT

This study compares the performance of two popular cross-platform mobile application development frameworks, Flutter and React Native. As the number of mobile users continues to grow, the ability to target multiple platforms using a single codebase is increasingly important for developers and companies. We conducted three manual UI tests; scrolling through a list, testing the camera, and filtering a large dataset to measure the performance of the frameworks in terms of CPU usage, memory usage, and janky frames on an Android device. The results indicate that Flutter may provide better performance in specific situations when compared to React Native. The study contributes to the existing research by providing additional insights into the performance of these frameworks under specific test scenarios.

INTRODUCTION

Mobile app development has become increasingly important as the number of mobile users continues to grow each year, surpassing desktop users since 2014 [11]. With this trend, the demand for cross-platform development has also increased, especially as mobile users are split between Android and iOS. This has led to the development of various cross-platform frameworks that allow developers to create mobile apps that can be easily distributed to different platforms.

The work of Delia et al. [3] explains that native applications developed for iOS and Android using their respective software development kit (SDK) allow for unimpeded access to all the system features (e.g., the camera, GPS) as well as high performance. These benefits, however, do not come without drawbacks. Writing native applications for both iOS and Android requires two separate code bases and thus doubles the development effort. Furthermore, both codebases must be maintained, and the developed app should ideally have the same design, user experience, and functionality. This requires highly skilled developers accustomed to both ecosystems (programming language, tools, OS-specific implementation details, etc.). For many companies, these drawbacks are not justifiable or feasible, and thus instead opt to use a cross-platform development framework that allows for ease of distribution to different platforms.

Multiple types of cross-platform development frameworks seek to accomplish the same goal of distributing the same app to different platforms but accomplish this through different approaches. One such approach is to develop a mobile web application. However, running the app in a web browser comes with the limitations of not being able to access all native features and slower performance; companies have therefore developed frameworks that more closely resemble native solutions [3]. This paper will compare two such frameworks, React Native and Flutter.

React Native was released as open-source by Facebook in 2015, allowing developers to use React's popular UI framework to create natively rendered mobile applications with the already familiar web language JavaScript [1]. Using a "bridge", the JavaScript code can communicate with the native rendering APIs and thus create mobile applications that both look and feel native. It is one of the most popular frameworks for developing cross-platform mobile applications for good reasons, a large and thriving community, low bar to entry for experienced web developers, and high code reusability when developing to different platforms.

Flutter is another open-source framework that was introduced by Google in 2018. Flutter applications are created with the programming language Dart and compiled natively to each target platform ahead of time. One of the main benefits of using Flutter to develop mobile applications is that neither a JavaScript bridge nor a WebView is needed. There is a large community of both developers and organizations supporting the project. [2]

As mentioned briefly above, the two frameworks operate in fundamentally different ways. Comparing two such different frameworks is a daunting task, as countless metrics could be used to evaluate them. It is crucial to determine which metrics to prioritize and which to exclude to ensure a fair and meaningful comparison. Furthermore, when attempting to develop the same application using two different frameworks, inevitably, some components will not be identical or even remotely similar. So how do you account for these differences when evaluating performance?

[1] https://reactnative.dev/
[2] https://docs.flutter.dev/resources/faq

Given the variation and potential for optimization, we made two important decisions. First, we made a general performance comparison, focusing on CPU usage, memory usage, and the number of janky frames. A janky frame is when the app's framerate drops below its target. Secondly, where the two components were not identical, we chose the most similar ones.

Aim

This thesis aims to compare two popular cross-platform mobile application development frameworks, React Native and Flutter, to understand which framework is preferable when developing mobile apps. There are many factors to consider when comparing two such frameworks, such as type of application, project scope, and previous developer experience. This paper will seek to answer this question by considering important performance metrics.

Research question

What are the performance differences between mobile apps developed in React Native compared to Flutter in terms of:

- CPU usage
- Memory usage
- The number of janky frames

THEORY

In this section, we present key concepts related to cross-platform mobile application development frameworks relevant to our study and previous research.

Frameworks

React Native and Flutter are the two most popular open-source mobile app development frameworks [3]. Below we present some essential key concepts and differences between the two frameworks.

List implementations

Both React Native and Flutter provide different implementations of lists for displaying data. These implementations differ in performance, functionality, and how the data is displayed.

In React Native, the FlatList component is used to display large datasets because of its lazy-loading strategy and removal of items that go off-screen. By default, ten items are initially rendered (initalNumToRender prop), and 21 items (unit of visible length) are asynchronously loaded off-screen (windowSize prop). This results in rendering items ten screens above and below the currently visible screen. The ScrollView component is also available, which uses the more naive loading strategy of rendering all its child components at once. [4] [5]

Flutter has a ListView widget that can display large and small datasets, depending on the constructor used. The ListView.builder constructor is more suitable for displaying larger data sets due to its on-demand creation of child widgets. Just like the FlatList component, items that go off-screen are destroyed. However, one big difference between the FlatList component and the Listview widget is that the ListView widget does not asynchronously load items off-screen. [6]

Native features

Both React Native and Flutter allow developers to access the different native features of the device. However, the implementation of accessing native features differs between the frameworks, which may impact the app's performance.

As mentioned previously, React Native uses a bridge to enable communication between JavaScript and native code. This bridge acts as an intermediary between the JavaScript code and the native code and may introduce some latency. Developers can create native modules or use a third-party solution to access native platform APIs.

On the other hand, Flutter provides access to native features through plugins that are compiled into the app and can directly access the native features of the device. These plugins are written in each platform's language and provided by the official Flutter team or the community.

Expo

Expo is an open-source framework that is used for building React Native apps [7]. It provides developers various tools and features to quickly and efficiently create high-quality mobile apps. The Expo SDK is a set of APIs and components that allow developers to access device features such as the camera and the GPS [8]. By leveraging the Expo SDK, developers can build apps that take full advantage of the capabilities of modern mobile devices. Expo Go is a client designed to test React Native apps [9]. After starting a development server using Expo CLI, developers can connect to the dev server using the Expo Go app on iOS or Android. This makes testing and debugging apps easier during development, saving developers time and effort.

Cross-platform development

Web Applications

Creating a web application is a simple way to distribute a mobile app across multiple platforms. These applications run in a web browser and are designed to work on various mobile devices. Typically written in HTML, CSS, and JavaScript, they are easy to develop, deploy, and maintain. One of the advantages of web applications is that they do not require approval from an app store. They also need only an Internet connection to work. However, performance can be slow [15] compared to other options, and the native features accessed through web APIs are limited.

Hybrid Apps

Another approach to building cross-platform mobile apps is to use a hybrid app development framework. Hybrid apps are essentially web applications that are packaged as native apps

[3] https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/
[4] https://reactnative.dev/docs/flatlist
[5] https://reactnative.dev/docs/scrollview
[6] https://api.flutter.dev/flutter/widgets/ListView-class.html
[7] https://docs.expo.dev/faq/
[8] https://docs.expo.dev/versions/latest/
[9] https://docs.expo.dev/get-started/expo-go/

and provide access to the native features of the device through a web view [12]. Popular hybrid frameworks include Apache Cordova and Ionic. Applications built with these frameworks are easy to distribute, but their performance is typically slower than native apps.

Cross-compiled Apps

Cross-compiled application development frameworks enable developers to build applications that run on multiple platforms by compiling code ahead of time for each platform. This means the code runs directly on the device, improving performance and preserving the native user experience [15]. Flutter is an example of a cross-compiled framework, where the code is compiled into native code for the targeted mobile platform, which can be either Android or iOS.

Interpreted Apps

With an interpreted application development framework, developers can build mobile apps using an interpreted language like JavaScript. The code runs in a runtime environment and is then interpreted into native code, which enables access to the native features of the device while still maintaining a platform-independent logic. Interpreted applications are easy to distribute while retaining the look and feel of native applications. However, performance may not be optimal due to the runtime interpretation of the code [12]. For example, React Native is considered an interpreted framework because it executes JavaScript code at runtime using a JavaScript engine, with some platform-specific components compiled ahead of time for the target platform.

Evaluation

User retention for mobile apps is generally low, and good app performance is crucial for maintaining users. It is troubling that in previous studies of mobile app performance, not much work has been done in evaluating React Native and Flutter. [7]

When evaluating the performance of React Native and Flutter, we consider several key metrics, including central processing unit (CPU) usage, random access memory (RAM) usage, and the number of janky frames. These terms and their significance in mobile app development are explained briefly in this section.

Both CPU and memory usage are important metrics to consider when evaluating the performance of mobile apps. CPU usage measures the amount of processing power the framework uses at any given time, while memory usage refers to the amount of RAM an app uses at a given time. High usage of either of these metrics implies that the device resources are strained [7], leading to slow performance, reduced battery life, and resource bottlenecks for other applications running on the device. Therefore, it is important to measure these metrics to ensure that the app utilizes the device's resources efficiently.

The fluidity of an application is defined by the number of frames displayed per second. Android and iOS devices operate at 60 frames per second (FPS). This results in the system having 16.67ms to generate the static image for that interval [10]. Failing to do this will cause frames to get dropped (janky frames), resulting in a less fluid and unresponsive app. A high FPS rate indicates smooth animations and transitions in the user interface (UI), while a low FPS rate can lead to stuttering and an overall poor user experience. [7]

[10]https://reactnative.dev/

Related works

We first present previous research comparing frameworks where at least one is React Native or Flutter and then present related research more broadly.

Framework comparisons

A prior study by Wu [14] has compared React Native and Flutter. The author made a case study where he analyzed and compared a React Native open-source app rewritten in Flutter. The work included a performance comparison where they implemented a list using both frameworks and measured dropped frames while scrolling. The results showed that Flutter had a slight advantage in fewer frames dropped, while React Native performed better in an I/O speed test.

In another study conducted by Jagiello [5], the performance of the two frameworks was compared by measuring the number of dropped frames within a specific time frame. The author measured the number of dropped frames under a certain time for two different applications to determine which framework performs better. The results revealed that React Native had fewer dropped frames, although the difference was not statistically significant.

A comparative study by Danielsson [2] evaluated the development process, user experience, and performance of React Native applications in relation to native Android applications. The study assessed performance by measuring GPU frequency, CPU load, memory usage, and power consumption using Android Studio Profiler and Xcode Instruments profiling tools. To evaluate performance, three test cases were conducted to replicate various user scenarios within the app. While the results indicated that React Native applications were slower when compared to native Android applications, the former did not hurt the overall user experience and had a shorter development time.

Olsson [10] conducted a comparative study to evaluate the performance and look between Flutter and native applications. The author developed four Flutter, Kotlin, and Swift applications and measured CPU performance on Android and iOS platforms. CPU usage was measured using Android Studio Profiler and Xcode Instruments. The results indicated that there was minimal difference in CPU usage between Flutter and native applications. However, the author noted that further testing is necessary to demonstrate the variance in performance between the two comprehensively.

Mahendra and Anggorojati [7] conducted a study comparing five performance metrics (CPU usage, memory usage, response time, frame rate, and application size) for Flutter and React Native with native Android as the baseline. The test application used in the study was a simple social media app with a timeline and profile pages. The experiment involved running two tests (infinite scroll and taking pictures with the camera) on an Android emulator, and each test was repeated 30 times for each app development framework. The results

showed that Flutter had better performance than React Native but that Native Android performed best.

Nawrocki et al. [9] explain that the quality of a cross-platform framework is hard to measure regarding developer experience, but through a survey posted on Stack Overflow 2019, some interesting data was collected. According to the survey, Flutter is the most popular framework, with 75.4% of developers having used it before. React Native is in second place with 62.5%, and in third place with 48.3% came Xamarin, a framework developed by Microsoft. After developing two mobile apps to measure the performance and the development experience, the authors concluded that Flutter had the upper hand over the other frameworks. React Native performed well but was not as easy to develop as Flutter or Xamarin. This was mainly due to React Native's dependence on community-maintained libraries, which affected the documentation quality.

Huber et al. [4] conducted a quantitative performance analysis of three typical UI interactions and developed one app natively and two using mobile cross-platform development (MCPD) approaches (React Native and Ionic/Capacitor). They measured CPU usage, main memory usage, janky frames, and GPU memory usage using automated tests. The study confirmed the results of previous studies, indicating that compared to natively developed apps, apps developed using MCPD approaches put a higher load on mobile devices regarding CPU usage, main memory, and GPU memory consumption. Additionally, the study showed that the better the mobile device, the lower the additional load on resources.

After reviewing these previous studies, it can be observed that the general trend in terms of performance is that native applications outperform those developed using Flutter, followed by React Native and hybrid frameworks; however, the actual performance difference may depend on the specific test used and may not always be noticeable to users.

Broadening the scope

Liu et al. [6] discovered that more than one-third of the 70 real-world performance bugs collected from eight large-scale and popular Android applications required special user interactions to manifest. They concluded that these performance bugs can easily escape traditional testing, and developers need to pay more attention to them. Through scenario-based testing, these hard-to-find bugs can more easily be found.

Mercado et al. [8] conducted an empirical study examining user reviews of apps on app marketplaces to determine whether there was a relationship between the approach used to develop apps and their perceived quality. The study found that hybrid apps on both Android and iOS platforms were more likely to receive user complaints. Additionally, Android users tended to have a more unfavorable perception of the performance, reliability, and usability of apps developed using hybrid approaches compared to traditional native approaches.

In an empirical study by Selakovic et al. [13], 98 performance issues in 16 popular JavaScript projects were fixed, most by optimizations that changed only a few lines of code. The use of JavaScript has evolved from simple scripts to complex programs, thanks to the recent improvements in just-in-time compilers. However, despite these improvements, developers still need to optimize their code manually. One such manual performance fix that developers can apply is to favor for-loops over functional-style processing of arrays. For example, use a regular for loop instead of the reduce function.

METHOD

Our study builds upon Mahendra and Anggorojati's previous work and introduces several changes and additions. We used an actual device instead of an emulator to provide more authentic and reliable results depicting real-world application performance. Additionally, each test was carried out in release mode, which is expected to outperform debug mode [2][5]. Although we ran all tests for Flutter in release mode, we were unable to measure janky frames. Therefore, we had to repeat the test scenarios in profile mode to obtain this data for Flutter.

We also incorporated networking into our apps to better simulate real-world usage scenarios and added an extra test case to filter large datasets. These improvements aim to reflect the outcomes observed in an actual app more accurately.

App description

Using both frameworks to measure performance differences, we developed a social media app inspired by Twitter. See Figures 1 and 2. The app has three main routes, a home page with an infinite scroll list, a page where the user can take a photo, and a search page where the user can filter posts. The infinite scroll list will resemble the home feed seen in social media apps. The list dynamically loads additional items from a server as the user scrolls down, ensuring an endless content stream. This approach offers two key benefits. First, it more accurately reflects the performance of real-world social media apps with multiple network connections. Second, it makes the list infinite by allowing new data to be fetched on demand.

We chose to develop a social media app prototype to evaluate the performance of frameworks for building such apps. Social media apps have resource-intensive features, like the features discussed above, making them useful test cases for measuring performance. By creating an app with a social media app's broad scope and external dependencies, we could simulate these technical demands and compare how different frameworks handle them, even though our prototype does not have social functionality.

To accurately capture objective and comparable data, the app implemented in both frameworks was developed to be as similar as possible in design and functionality. Therefore, we used Material Design for both apps to avoid inconsistency in the UI. Flutter has built-in support that we used, and for React Native, we used a community UI framework called React Native Paper that offered similar components.

Implementation

This section describes how the different features were implemented in the app.

Home page

The posts displayed in the home page are loaded using HTTP GET requests from a local Go server. Each post in the list

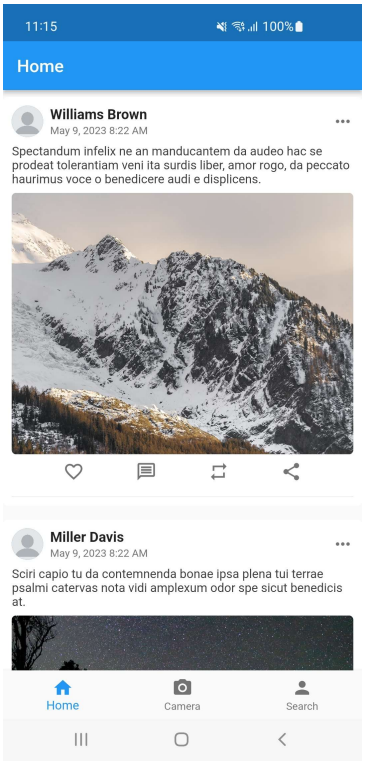

Figure 1. Flutter app

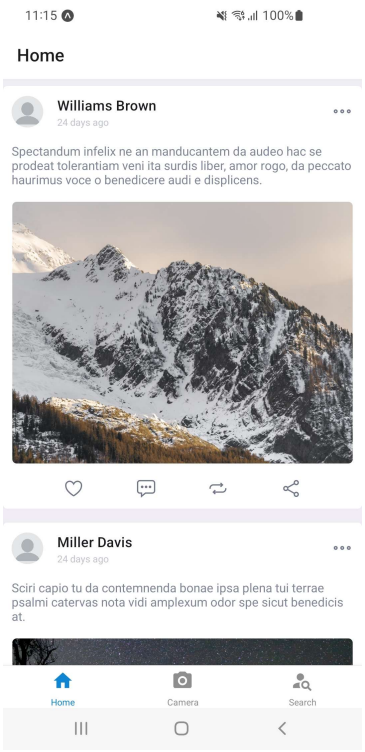

Figure 2. React Native app

contains a username, randomly generated text, a profile picture, an image, and four icons. To display the posts, we first make an HTTP GET request to our go server at /posts/get/:id/:count, which returns the count number of posts, starting from id as JSON data. The client then parses this JSON data, and additional HTTP GET requests are made to our server for each image when they are to be displayed. See Figure 3 for a sequence diagram. The icons displayed under each post are stored on the device and not loaded from the server.

The profile picture is small (16KB), and the embedded images are much larger (between 145KB and 1.3MB). Additionally, each request processed by the local server has an artificially added delay (20ms - 150ms) to simulate the conditions one would see in a production environment.

When the user scrolls down the list and reaches the bottom, an additional request is sent, just like the initial request, with the id, increased and processed accordingly.

Camera

Flutter and React Native provide ways to integrate camera functionality into mobile apps. The implementation process for both frameworks involves utilizing a camera plugin or component to access the device's camera and allow users to take photos. Once a photo is taken, it can be saved to the device's gallery using a package or library specific to the framework.

In Flutter, the ImagePicker plugin allowed users to take new photos using the camera. The GallerySaver package was also used to save the taken image to the device's gallery.

In React Native, the Expo Camera component was used to access the device's camera and allow users to take photos within the app. In addition, the Expo Media Library was used to save the image to the device's gallery.

Search page

The route containing the search page loads the posts in the same manner as described above in the home page section, but with the difference that instead of loading new posts when the user scrolls to the bottom of the list, we load all the data needed by sending a GET request for 100 000 posts. This data is parsed and saved in a list when the user clicks a button. After the user presses the button, a text input field is displayed from where the user can search for and filter posts based on the username.

Building and installing

Building the apps in both frameworks required different processes. To build the Flutter app, we used the "flutter build apk" command to build an Android APK package. The resulting APK file was installed on the Android device using the "flutter install" command.

For building the React Native app, we used EAS Build, which is a hosted service for building binaries for Expo and React Native projects [11]. When the build was done, we installed it on the device using the command "adb install."

[11] https://docs.expo.dev/build/introduction/

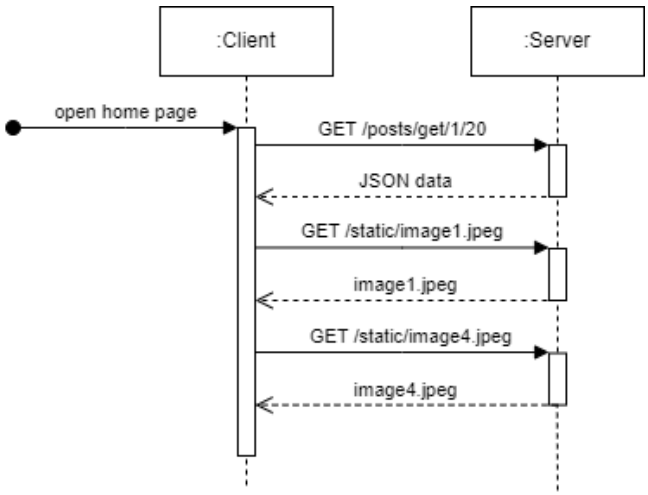

Figure 3. Example of how the client communicates with the server

Experiment

A Google Pixel 2 smartphone operating on the Android platform was used to conduct the experiment. The experiment involved executing three test scenarios manually on the smartphone five times each. We used React Native version 0.71.6 and Flutter version 3.10.0 for this experiment. As the Android Studio Profiler only provided visual representations of CPU and memory usage, we obtained data samples via the Android Debug Bridge (ADB) to better determine maximum, mean, and minimum values.

While ADB allowed us to sample frame data for determining the number of janky frames in the React Native app, this method was not viable for measuring frame data within the Flutter app, as it employs its own rendering engine. Consequently, we resorted to running the Flutter app in profile mode and counting the number of janky frames with the help of Dart DevTools.

```
file="$1.csv"
echo "cpu%,mem%" > $file
pid=‘adb shell pidof $1‘
i=1
adb shell dumpsys gfxinfo "$1" reset
start_time=$(date +"[%T]")
end_time=$((SECONDS+30))

while [ $SECONDS -lt $end_time ];
do
  res=‘adb shell top -b -n 1 -p $pid | tail -
    n 1 awk ’/^ *[0-9]/ {print $9 "," $10}’‘
  echo "[$(date +"%T")], $start_time, $i:
    $res"
  echo "$res" >> $file
  ((i++))
done

adb shell dumpsys gfxinfo "$1" framestats >
    framestats.txt
```

Listing 1. Shell script

We conducted a performance analysis of React Native and Flutter by implementing a shell script (Listing 1) on an Android device. The script utilized two commands, "adb shell top" and "adb shell dumpsys gfxinfo package.name framestats," to gather detailed statistics on CPU usage, memory usage, and frame rate performance.

The "adb shell top" command was repeatedly executed for 30 seconds to generate CPU and memory usage data, which we extracted and compiled into a visual graph. Meanwhile, the "adb shell dumpsys gfxinfo package.name framestats" command provided detailed frame rate statistics, including the total number of frames rendered, janky frames, and the frames per second.

As ADB could not be used for the Flutter app to generate frame rate statistics, we measured the number of janky frames separately through Dart DevTools. This involved opening the performance profiler in Dart DevTools while the app ran in profile mode. By doing so, we could view the app's frame rate over time in a chart and obtain the app's average frame rate throughout the recording.

Test scenario

We conducted three test scenarios where we measured the metrics stated in the research question, CPU usage, memory usage, and the number of janky frames. We measured all performance metrics for each test case, allowing us to determine which test scenario had the most significant impact on each metric. Before each test, we deleted the app cache and allowed a 10-second wait period to ensure that the starting conditions for all tests were equal.

Test Scenario 1

The first scenario involved scrolling down the home page containing images and text, with the test running for 30 seconds. This allowed us to evaluate the application's ability to handle heavy loads and assess its responsiveness under such conditions.

1. Scroll down the list for 30 seconds

Test Scenario 2

In the second scenario, we focused on the performance of accessing hardware functionality by testing the implementation of the camera. Specifically, we repeated taking a photo and saving it to the device's gallery for 30 seconds. This test provided valuable insights into how accessing native functionality differed between the two frameworks.

1. Navigate to the route with camera functionality
2. Press the camera button to take a photo
3. Save the taken photo
4. Take a new photo
5. Repeat steps three and four for 30 seconds.

Test Scenario 3

The third scenario involved testing the application's search feature, with the user navigating to the search page by pressing the search icon on the navigation bar and then loading posts from the server. The test involved entering an account name in the search field at the top to filter and display posts

related to the searched account. This allowed us to evaluate the performance when filtering large data.

1. Navigate to the search page
2. Click the button to load posts
3. Search for an existing account name

Automatic vs. manual testing

For the execution of the three test scenarios, we deliberated on utilizing automated testing frameworks for both applications to ensure consistent and objective test runs. Flutter's built-in automatic testing functionality, Flutter Driver, provided an extensive framework. However, for React Native, we had to resort to an external testing framework to conduct the test. As it transpired, there was no straightforward means of automatically running tests on both applications. Moreover, the scrolling speed of the two varying testing frameworks resulted in further inconsistency. Ultimately, we concluded that manual testing was the most appropriate solution.

RESULTS

The results of the test scenarios conducted on React Native and Flutter are presented in this section.

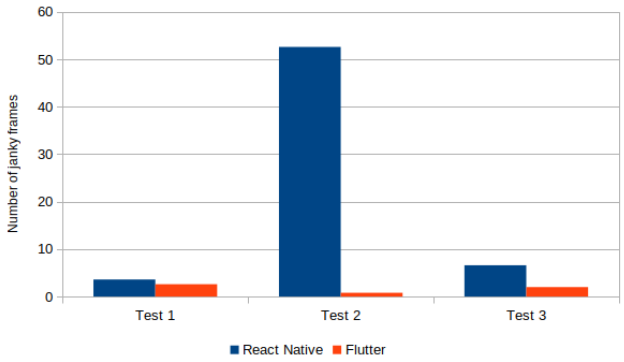

Figure 4. Janky Frames

Test 1 Results

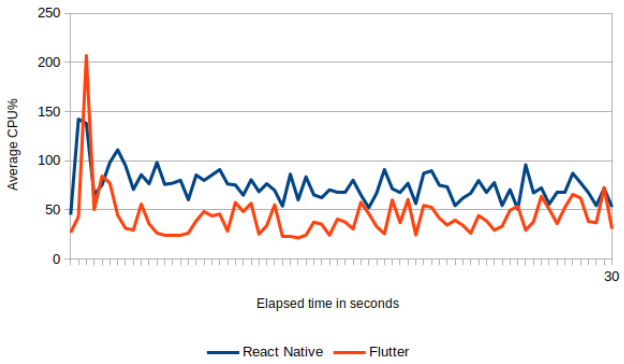

Figure 5. CPU Test 1

Test 1 focused on rendering a scrolling list and evaluated CPU usage, memory usage, and janky frames. Flutter outperformed

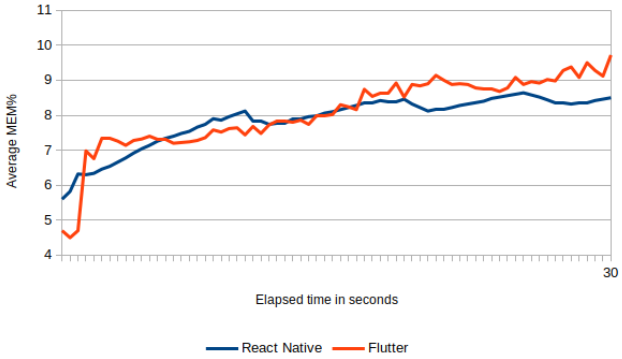

Figure 6. MEM Test 1

React Native in terms of CPU usage, with an average CPU usage of 43.42% compared to React Native's average of 52.92%. This difference in CPU usage was 66 throughout the test scenario, with Flutter spiking higher in the beginning, as shown in Figure 5. On the other hand, React Native had slightly lower memory usage than Flutter, with an average memory usage of 7.85% compared to Flutter's average of 8.06%, as shown in Figure 6. Moreover, the test results showed that Flutter had fewer janky frames than React Native, with an average of 2.6 janky frames compared to React Native's average of 3.6, as shown in Figure 4.

Test 2 Results

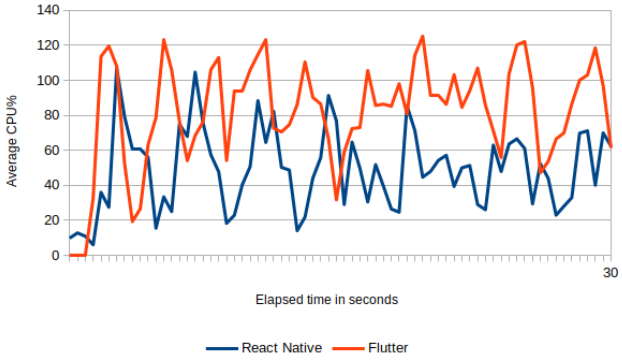

Figure 7. CPU Test 2

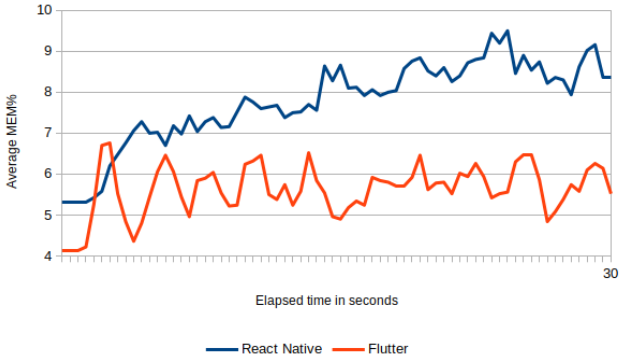

Figure 8. MEM Test 2

The second test scenario focused on evaluating the performance of accessing hardware functionality through the camera implementation. The results indicate that Flutter performed better than React Native on average regarding memory usage and janky frames. Figure 7 shows that React Native had lower CPU usage than Flutter, with an average CPU usage of 49.06% for React Native and 81.63% for Flutter. On the other hand, Figure 8 demonstrates that Flutter had lower average memory usage, with an average of 5.62%, compared to React Native's average of 7.76%. Additionally, Figure 4 shows that Flutter had significantly fewer janky frames, with an average of 0.8, compared to React Native's average of 52.6.

Test 3 Results

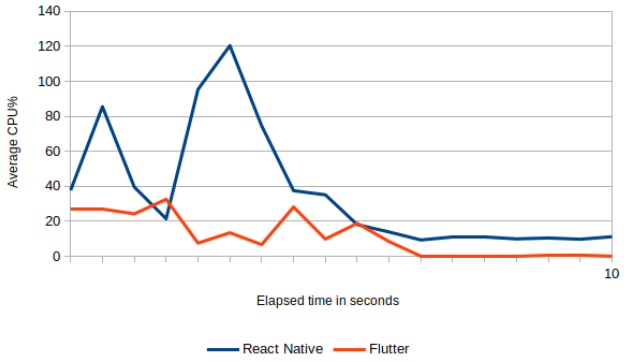

Figure 9. CPU Test 3

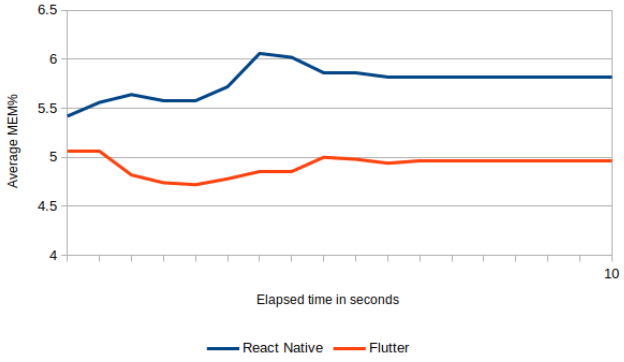

Figure 10. MEM Test 3

In the third test scenario, the application's search feature was tested, and the performance of the frameworks in terms of CPU usage, memory usage, and janky frames was evaluated. The results showed that, on average, Flutter performed better than React Native in all three areas, as shown in Figure 9, 10, and 4. Specifically, the average CPU usage for Flutter was 11.34%, which was significantly lower than the average CPU usage for React Native, which was 36.21%. The average memory usage for Flutter was 4.91%, which was also lower than the average memory usage for React Native, which was 5.77%. Finally, the average number of janky frames for Flutter was 2, lower than the average number for React Native, 6.6.

DISCUSSION

The discussion section of this study addresses several important points related to the performance tests conducted on React Native and Flutter.

Method

First, it is worth noting that the janky frames in Flutter were measured using the Flutter profiler tool. This could have impacted the results obtained since the profiling mode can negatively impact the performance.

Another limitation of this study is the absence of automated tests, which could have improved the consistency and reliability of the measurements. Additionally, each test was only performed five times for each scenario, which may not have been sufficient to obtain accurate data. Furthermore, most tools for measuring performance are primarily designed for debugging performance issues and not for extracting easily readable data. This makes it challenging to ensure the measurement values are accurate and reliable.

It should also be noted that the performance tests were conducted exclusively on an older Android device. Certain tests may have yielded different results on newer Android or iOS devices. Previous research indicates no significant differences, at least not in the general trend.

Results

According to Cheon and Chavez [1], developing a cross-platform app can be challenging due to subtle platform differences that can significantly impact its design and coding. Even with prior research, unexpected challenges can arise during development that requires additional problem-solving. We encountered such challenges while working with React Native and Flutter, platforms with which we had limited experience. Despite this, we remained committed to delivering a high-quality app by following best practices and applying the knowledge we gained through our research and development efforts. Cheon and Chavez also note that platform differences can be complex and difficult to anticipate at the early stages of development, making it essential to stay adaptable and resourceful throughout the process.

Test 1

We performed a scrolling test for the first test, but we did not evaluate how the frameworks handle networking. The addition of networking was mainly intended to capture more realistic data rather than to examine its effect on performance precisely. It is possible that networking had a more significant negative impact on one of the frameworks than the other. It is also worth noting that the difference in the way the scrolling list work between the two frameworks could have had a significant impact on performance, particularly with the inclusion of networking. React Native loads content asynchronously off-screen, which may have considerably affected performance.

Test 2

For the app written in Flutter, we used an official plugin for the camera, while we relied on an Expo plugin for React Native. Our findings indicate that React Native produced more janky frames than Flutter, which exhibited a smoother performance.

Additionally, Flutter performed better than React Native by capturing approximately seven more pictures within 30 seconds. During our testing, we experimented with disabling the autofocus option temporarily in React Native and observed a slight improvement in the speed of picture-taking. However, it's worth noting that Flutter had higher CPU usage, perhaps due to the more significant number of pictures taken and saved to disk.

Test 3

The last test in our experiment involved fetching one hundred thousand posts and filtering them based on username. To accomplish this, we used the built-in filter function for both frameworks. However, we knew this approach might have hurt performance for React Native due to a lack of compiler optimization, compared to using a for loop. Despite this concern, we decided to stick with the built-in functions since we wanted to assess how the built-in functions compare. Our intentional lack of optimization might help explain why we see such a big difference in CPU usage in test 3, and it is conceivable that the results could be more equal through optimization.

Finally, it is worth considering whether performance is the most crucial factor when choosing between React Native and Flutter for mobile application development. While there were differences in performance between the two frameworks, they were not significant enough to suggest that one is inherently better than the other. Other factors, such as developer experience and community support, may also be important considerations when choosing a framework. Additionally, for applications where performance is critical, developing a native application may be a better option.

Source criticism

We had difficulty finding relevant research with many citations, as many articles on this topic have been published recently. However, we used various sources, including bachelor-level papers, and also discovered noteworthy academic papers that we incorporated into our research. These sources provided valuable insights into the performance evaluation of cross-platform development frameworks, and we deemed them credible because they were consistent with more established sources. Although there is limited research on this topic, we took a comprehensive approach to gathering insights from multiple sources to ensure the reliability and validity of our findings.

The work in the broader context

The focus of this paper is on performance. However, it's important to note that optimizing for performance without considering potential environmental impacts is not ideal. Given the significant number of smartphone users worldwide, opting for greater battery consumption to achieve better performance may have adverse environmental effects.

While cross-platform development tools offer several advantages, such as shorter development timelines than native development, they might consume more computing resources. Hence, it is crucial to consider the trade-offs between these two approaches and their resource usage implications.

CONCLUSION

In conclusion, this thesis aimed to compare React Native and Flutter, two popular cross-platform mobile application development frameworks, to determine which framework is preferable for mobile app development. The research question focused on performance differences between the two frameworks in terms of CPU usage, memory usage, and the number of janky frames.

In comparison with previous research, our findings are generally consistent with the results of other studies that have compared the performance of React Native and Flutter. However, our study contributes to the existing research by providing additional insights into the performance of these frameworks under specific test scenarios.

Based on the performance tests conducted, it can be concluded that both frameworks have their strengths and weaknesses in mobile application development. The results suggest that Flutter may perform better in certain scenarios, particularly regarding CPU and memory usage and janky frames. However, it is essential to note that these results are based on specific test scenarios and may not necessarily hold for other use cases. Developers should consider the particular requirement of their application to determine which framework would best fit their needs.

It is worth noting that our study has some limitations. For example, we only tested performance on limited devices and scenarios. Future research could explore the performance of these frameworks on a wider range of devices and scenarios. It is also essential to continue monitoring and comparing the performance of these frameworks as they evolve and new updates are released.

REFERENCES

[1] Yoonsik Cheon and Carlos Chavez. 2021. Converting Android Native Apps to Flutter Cross-Platform Apps. In *Proceedings - 2021 International Conference on Computational Science and Computational Intelligence, CSCI 2021*. Institute of Electrical and Electronics Engineers Inc., 1898–1904. DOI: http://dx.doi.org/10.1109/CSCI54926.2021.00355

[2] William Danielsson. 2016. *React Native application development : A comparison between native Android and React Native*. Master's thesis. http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-131645

[3] Lisandro Delia, Nicolas Galdamez, Pablo Thomas, Leonardo Corbalan, and Patricia Pesado. 2015. Multi-platform mobile application development analysis. *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)* (2015). DOI: http://dx.doi.org/10.1109/RCIS.2015.7128878

[4] Stefan Huber, Lukas Demetz, and Michael Felderer. 2020. Analysing the performance of mobile cross-platform development approaches using ui interaction scenarios. In *Communications in Computer and Information Science*, Vol. 1250 CCIS. Springer, 40–57. DOI: http://dx.doi.org/10.1007/978-3-030-52991-8{_}3

[5] Jakub Jagiello. 2019. *Performance comparison between React Native and Flutter*. Bachelor's thesis. Umeå University. https://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-163190

[6] Yepang Liu, Chang Xu, and Shing Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 1013–1024. DOI: http://dx.doi.org/10.1145/2568225.2568229

[7] Mohammad Mahendra and Bayu Anggorojati. 2020. Evaluating the performance of Android based Cross-Platform App Development Frameworks. *ACM International Conference Proceeding Series* (11 2020), 32–37. DOI:http://dx.doi.org/10.1145/3442555.3442561

[8] Iván Tactuk Mercado, Nuthan Munaiah, and Andrew Meneely. 2016. The impact of cross-platform development approaches for mobile applications from the user's perspective. In *WAMA 2016 - Proceedings of the International Workshop on App Market Analytics, co-located with FSE 2016*. Association for Computing Machinery, Inc, 43–49. DOI: http://dx.doi.org/10.1145/2993259.2993268

[9] Piotr Nawrocki, Krzysztof Wrona, Mateusz Marczak, and Bartlomiej Sniezynski. 2021. A Comparison of Native and Cross-Platform Frameworks for Mobile Applications. *Computer* 54, 3 (3 2021), 18–27. DOI: http://dx.doi.org/10.1109/MC.2020.2983893

[10] Matilda Olsson. 2020. *A Comparison of Performance and Looks Between Flutter and Native Applications : When to prefer Flutter over native in mobile application development*. Bachelor's thesis. Blekinge Institute of Technology. https://urn.kb.se/resolve?urn=urn:nbn:se:bth-19712

[11] Carlos Manso Pinto and Carlos Coutinho. 2018. From Native to Cross-platform Hybrid Development. *9th International Conference on Intelligent Systems 2018: Theory, Research and Innovation in Applications, IS 2018 - Proceedings* (7 2018), 669–676. DOI: http://dx.doi.org/10.1109/IS.2018.8710545

[12] C. P. Rahul Raj and S. B. Tolety. 2012. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. *2012 Annual IEEE India Conference, INDICON 2012* (2012), 625–629. DOI: http://dx.doi.org/10.1109/INDCON.2012.6420693

[13] Marija Selakovic and Michael Pradel. 2016. Performance issues and optimizations in Java script: An empirical study. In *Proceedings - International Conference on Software Engineering*, Vol. 14-22-May-2016. IEEE Computer Society, 61–72. DOI: http://dx.doi.org/10.1145/2884781.2884829

[14] Wenhao Wu. 2018. *React Native vs Flutter, cross-platform mobile application frameworks*. Bachelor's thesis. Metropolia University of Applied Sciences.

[15] Spyros Xanthopoulos and Stelios Xinogalos. 2013. A comparative analysis of cross-platform development approaches for mobile applications. *ACM International Conference Proceeding Series* (2013), 213–220. DOI: http://dx.doi.org/10.1145/2490257.2490292