Malaysian Journal of Computing, 9 (2): 1930-1943, 2024
Copyright © UiTM Press
eISSN: 2600-8238

COMPARATIVE ANALYSIS OF UI CONSTRUCTION AND PERFORMANCE IN ANDROID AND FLUTTER FRAMEWORKS

Zou Donglan[1], **Mohamad Yusof Darus**[2*], **Maslina Abdul Aziz**[3], **Suzana Zambri**[4]

[1]*School of mathematics and computer, Xinyu University, Xinyu, Jiangxi, China*
[2*,3,4]*College of Computing, Informatics and Mathematics ,UiTM, Malaysia*
[1]29877624@qq.com, [2*]yusof_darus@uitm.edu.my, [3]maslina_aziz@uitm.edu.my,
[4]suzana081@uitm.edu.my

ABSTRACT

This article examines the process of building user interfaces (UI) using two leading mobile application development tools: Android and Flutter. By developing common login interfaces, the UI construction between these platforms was compared and contrasted. The study highlights key differences in UI elements, design paradigms, and performance metrics, offering a comprehensive analysis. A detailed experimental plan to assess and compare UI development in Flutter and Android was designed, followed by performance evaluations based on metrics such as application size, runtime speed, and memory usage. The findings aim to help beginners quickly grasp and master these tools. Furthermore, the implications for developers, particularly regarding code complexity, reusability, and the overall development experience were discussed. This research contributes to the ongoing discourse on mobile app development best practices, helping guide tool selection for specific project requirements.

Keywords: *Android, Flutter, Mobile Application Development, Usability, User Interfaces (UI).*

Received for review: 27-08-2024 Accepted: 27-09-2024; Published: 01-10-2024
DOI: 10.24191/mjoc.v9i2.26484

1. Introduction

Familiarity with both Android and Flutter is essential for mobile application developers. Android, developed by Google, is an operating system based on the Linux platform for mobile phones and tablets. Since its inception, Android has received unprecedented attention and has quickly become one of the most popular operating systems on mobile platforms. Flutter, also developed by Google, is an open-source mobile application development framework. Its main goal is to provide a unified, high-performance, and easy-to-learn cross-platform UI toolkit, enabling developers to build iOS and Android applications simultaneously using a single codebase. Flutter adopts a unique "responsive" architecture design, combined with the Dart language and a custom rendering engine, to achieve efficient interface drawing and animation effects.

To enhance mobile programming education and support student self-study, researchers have developed various tools and applications. For instance, the Android Programming Learning Assistance System (APLAS) uses the native programming language for the Android platform to aid learning (Syaifudin et al., 2019). Another example is a mobile application designed for

luggage tracking to help users track misplaced or stolen items in public places (Abdulham & Mohamed Noor, 2023). Moreover, some researchers conducted a study focusing on the functionality and usability of mobile apps for people with dementia, reviewing apps that assist individuals with mild to moderate dementia in their daily lives (Zehang & Sabran, 2023).
Traditionally, mobile app development required separate codebases for each platform, such as iOS (Apple) and Android (Google). However, the advent of cross-platform mobile programming has revolutionized the development landscape. Cross-platform mobile programming allows developers to create mobile applications that run on multiple operating systems using a single codebase. This approach leverages frameworks and tools that enable developers to write code once and deploy it across various platforms, saving time and effort (Isitan & Koklu, 2020). Thus, studying cross-platform app development is crucial for students.

Recently, Flutter has gained popularity as a Software Development Kit (SDK) for creating cross-platform applications compatible with both Android and iOS. Many software developers have adopted Flutter as their preferred tool. Research has explored the Grammar-Concept Understanding Problem (GUP) in mobile programming using Flutter, providing introductory insights for novice students (Syaifudin et al., 2019 & Patta et al., 2023). Additionally, Mahmud et al. (2023) utilized the Flutter framework and Firebase services for front-end and back-end development to create the Mathvision Prototype using predictive analytics.

To support beginners in quickly understanding and mastering these two development tools, this research focuses on the UI design of both platforms. It analyzes and compares the differences in the UI construction process, as well as the differences in running speed and memory consumption under the same functional implementations. The remainder of this paper is organized as follows: Section 2 describes related works, Section 3 discusses the research methodology, Section 4 presents the results and discussions, and Section 5 concludes the paper.

2. Related Works

In recent years, educational applications have revolutionized learning paradigms, prompting significant research into new methodologies for developing learning apps. This study focuses on UI design for Android and iOS platforms.

Perinello and Gaggi (2024) investigated the accessibility of user interfaces in mobile applications developed with Flutter and React Native. Their analysis revealed that while neither framework inherently ensures complete accessibility, they identified effective strategies for enhancing the accessibility of components and widgets. Notably, React Native was found to offer a more streamlined approach, resulting in more readable source code.

In a study focusing on cross-platform mobile programming learning, the researchers examined the Flutter framework. Patta et al. (2023) engaged 109 undergraduate students from Makassar State University, Indonesia, in 22 tasks covering 71 keywords and questions. Their findings underscored the effectiveness of their approach in assessing student comprehension levels, pinpointing areas for improvement, and facilitating targeted learning resource development.

Kishore et al. (2022) conducted a comparative analysis of Flutter and React Native, evaluating application performance and stability across Android and web platforms. Their study highlighted the differences in functionality and performance metrics, providing valuable insights into these two prominent cross-platform development technologies.

Choudhari et al. (2022) proposed an economical mobile application for monitoring electricity usage, offering statistical data on consumption, meter reading uploads, power usage monitoring, and bill estimation to promote awareness of electricity consumption habits. Syaifudin et al. (2022) introduced an automated Dart code verification system aimed at enhancing learning in Flutter-based mobile application programming. Their system integrates automated Dart code verification, drawing from established software testing methodologies prevalent in Android development.

Ammar (2021) proposed an innovative approach for automatically generating mobile user interfaces using standards like EMF, GMF, ATL, and Xpand. This approach provides a structured framework for efficient UI development. Martinez et al. (2020) conducted a systematic mapping study and industry survey to formulate the Mobile Ilities framework, integrating mobile-specific challenges with agile software development practices. This framework aids novice developers by combining Scrum methodologies with mobile development characteristics. Boukhary and Colmenares (2019) introduced a new Flutter architecture based on Uncle Bob's Clean Architecture principles, offering robust state management solutions and serving as a comprehensive framework for Flutter application development.

Despite valuable insights gained from existing studies, few have specifically addressed UI construction for Android and Flutter platforms. This paper aims to bridge this gap by analyzing and comparing UI construction processes across these two platforms.

2.1 Android Architecture

The Android architecture is organized into four main layers: the Application layer, the Application Framework layer, the System Runtime layer, and the Linux Kernel layer, as depicted in Figure 1.

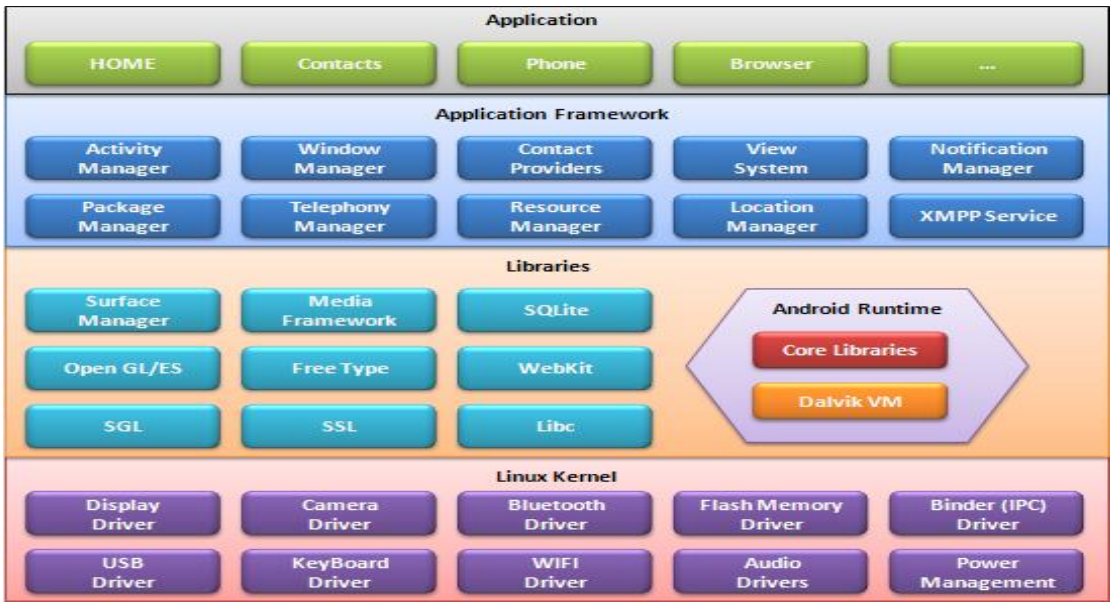


Figure 1. Android Architecture Diagram (Krajci & Cummings, 2013)

Below are some brief descriptions of all the layers:

a. The application layer serves as the topmost tier where users directly interact with apps such as email, messaging, and games.
b. The Application Framework layer provides essential services to applications, including the Activity Manager, Content Providers, Resource Manager, and other critical system services.
c. The System Runtime layer executes Android applications. In earlier Android versions, it used the Dalvik Virtual Machine; however, newer versions utilize the Android Runtime (ART) with Ahead-of-Time (AOT) compilation for improved performance.
d. At the base, the Linux Kernel layer offers core functionalities like hardware drivers, memory management, and other essential system functions.

These layers collectively form the Android architecture, enabling efficient interaction between applications, system services, and the underlying hardware.

According to Mayrhofer et al. (2021), Android applications consist of components that can invoke independent functional modules. These components are categorized into four core types: *Activity*, *Service*, *BroadcastReceiver*, and *ContentProvider*, as depicted in Figure 2. These components play crucial roles in defining the behavior and interaction of applications within the Android system.

Each of these components must be registered with corresponding tags in the project's *AndroidManifest.xml* file. An Android application primarily consists of these four independent components, which can be called and coordinated to form a complete application.

The communication between these components is mainly facilitated by Intents. An Intent describes the action to be performed, the data involved, and any additional information necessary for a single operation within an application. Based on the Intent description, Android locates the corresponding component, passes the Intent to it, and completes the component call. Thus, the Intent acts as a mediator, providing the necessary information for component interaction, effectively decoupling the caller from the called component. The Android component diagram is shown in Figure .

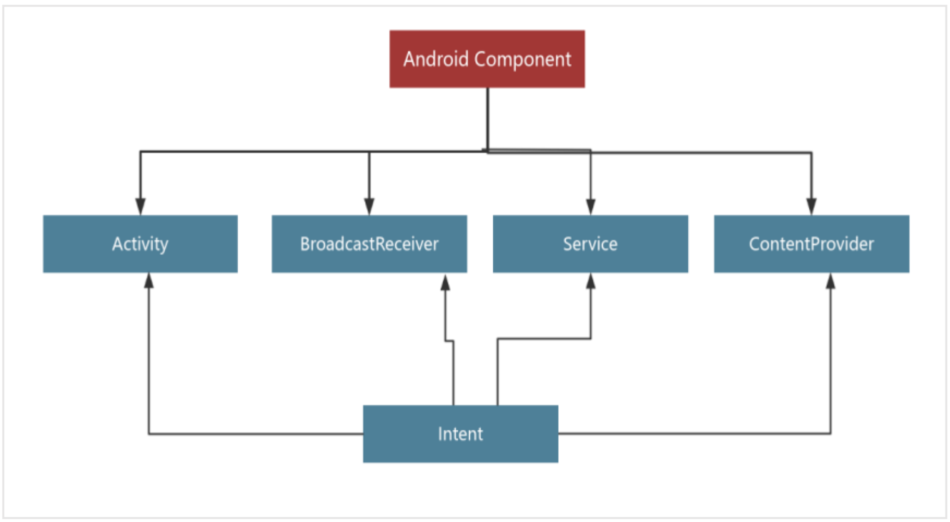


Figure 2. Android Component Diagram (Mayrhofer et al., 2021)

2.2 Android UI Building

In Android, UI elements are constructed using View and *ViewGroup* components. *ViewGroup* acts as a container for organizing controls within the interface, encompassing both standard *View* controls and other *ViewGroup* containers, as illustrated in Figure 3.

a. **View**: This is the base class for all UI components in Android. Examples include *TextView*, *ImageView*, Button, etc. Views are responsible for drawing and handling user interactions.
b. **ViewGroup**: ViewGroup is a subclass of View that acts as a container for other Views. It provides layout parameters to its child Views and organizes them spatially on the screen. Examples include *LinearLayout, RelativeLayout, FrameLayout*, etc.

Together, View and ViewGroup form the foundation for building the user interface in Android applications, allowing developers to create complex layouts and interactive elements

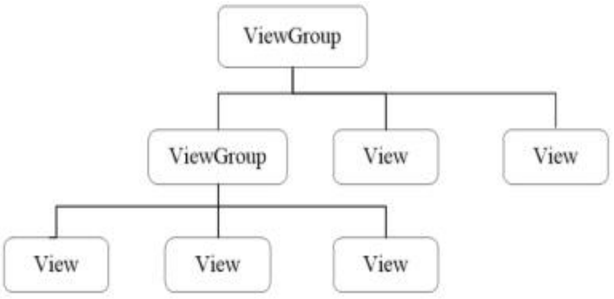
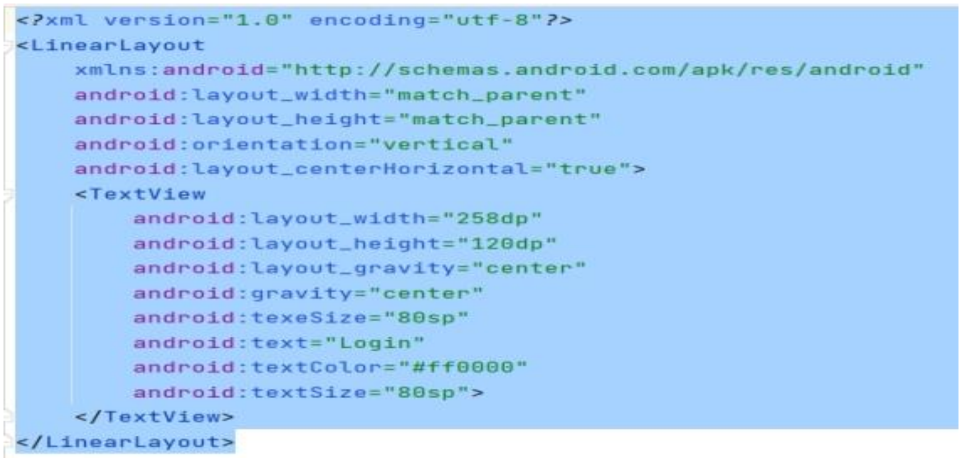


Figure 3. Android View Hierarchy Chart (Allen , 2021)

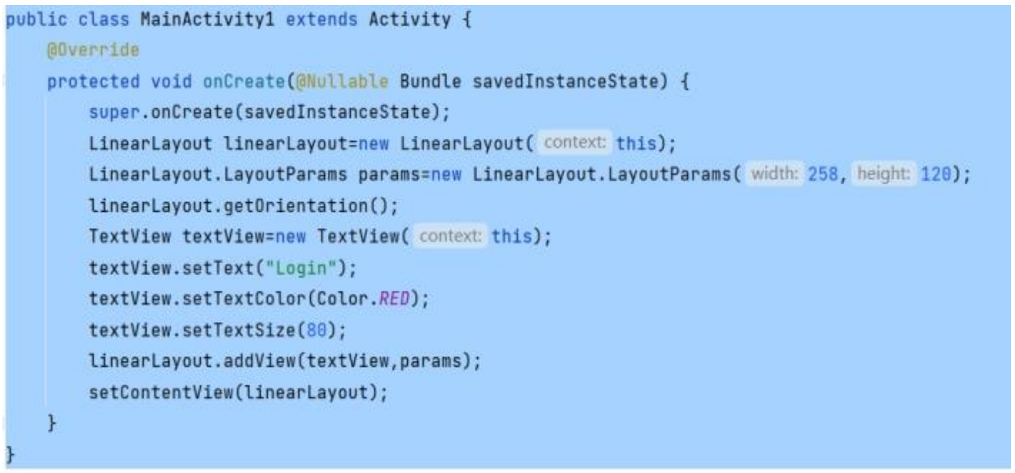
In Android, there are two primary methods for designing user interface layouts. The most used method involves writing the layout in XML files. This approach effectively separates the UI layout from the Java code, enhancing program structure and clarity, as depicted in Figure 4.



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_centerHorizontal="true">
    <TextView
        android:layout_width="258dp"
        android:layout_height="120dp"
        android:layout_gravity="center"
        android:gravity="center"
        android:texeSize="80sp"
        android:text="Login"
        android:textColor="#ff0000"
        android:textSize="80sp">
    </TextView>
</LinearLayout>
```

Figure 4. Writing Layouts in *XML* Files

The second method is to write layouts in *Java* code, where all layout and control objects can be created using the "*new*" keyword. The created *View* control can be added to the *ViewGroup* layout to display the *View* control in the layout interface, as shown in Figure.

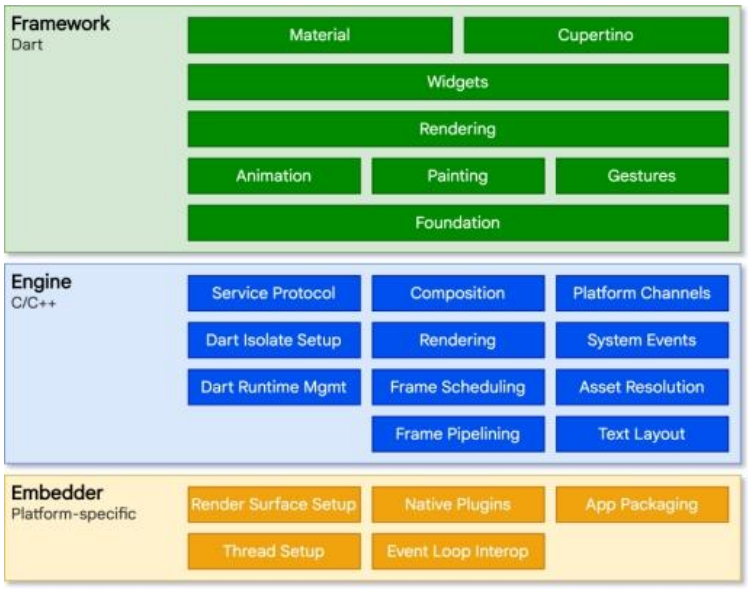


```
public class MainActivity1 extends Activity {
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        LinearLayout linearLayout=new LinearLayout( context: this);
        LinearLayout.LayoutParams params=new LinearLayout.LayoutParams( width: 258, height: 120);
        linearLayout.getOrientation();
        TextView textView=new TextView( context: this);
        textView.setText("Login");
        textView.setTextColor(Color.RED);
        textView.setTextSize(80);
        linearLayout.addView(textView,params);
        setContentView(linearLayout);
    }
}
```

Figure 5. Writing Layouts in Java Files

2.3 Flutter Architecture

In Flutter, the primary goal of its architecture is to offer a unified, high-performance, and user-friendly cross-platform UI toolkit. Developers can build iOS and Android applications concurrently using a single codebase. Flutter employs a unique "responsive" architecture design, utilizing the Dart programming language and a custom rendering engine to achieve efficient interface rendering and animations, as described by Durai et al., (2022) The system framework diagram is depicted in Figure 6, illustrating how Flutter components integrate to support its cross-platform capabilities.



.

Figure 6. Flutter System Framework Diagram (Nagaraj et al., 2022)

Flutter architecture is structured into three main layers: Embedded, Engine, and Framework. The Framework layer, implemented in Dart language, includes Foundation, Animation, Painting, and Gestures layers tailored by Google for developers. The Rendering layer manages the UI tree, recalculating and updating UI elements efficiently. Flutter development revolves around six key components:

a. Dart Language: Google's Dart language provides Flutter with rich APIs and library support for building UI components and managing business logic.
b. Widget System: Widgets are the core concept in Flutter, representing reusable UI elements. The Widget tree forms the application's view hierarchy, with each Widget managing its state and UI through efficient updates.
c. Material Design & Cupertino Widgets: Flutter supports both Material Design and Cupertino styles, facilitating platform-specific UI development for Android and iOS.
d. Flutter Engine: Powered by the Skia rendering engine, the Flutter Engine converts Dart bytecode into graphics commands, enabling high-performance UI rendering and animations.
e. Platform Channels: Enables communication between Flutter's Dart code and native platform code (Java/Kotlin for Android, Objective-C/Swift for iOS), supporting integration with device-specific features.
f. Hot Reloading & DevTools: Features like Hot Reloading allow developers to preview and deploy code changes in real-time without disrupting the app's state. DevTools offer debugging capabilities for diagnosing issues during development.

In Flutter, everything is a component built on Widgets, forming a hierarchical structure that defines the application's UI layout and behavior as illustrated in Figure 7 (Syaifudin et al., 2024).

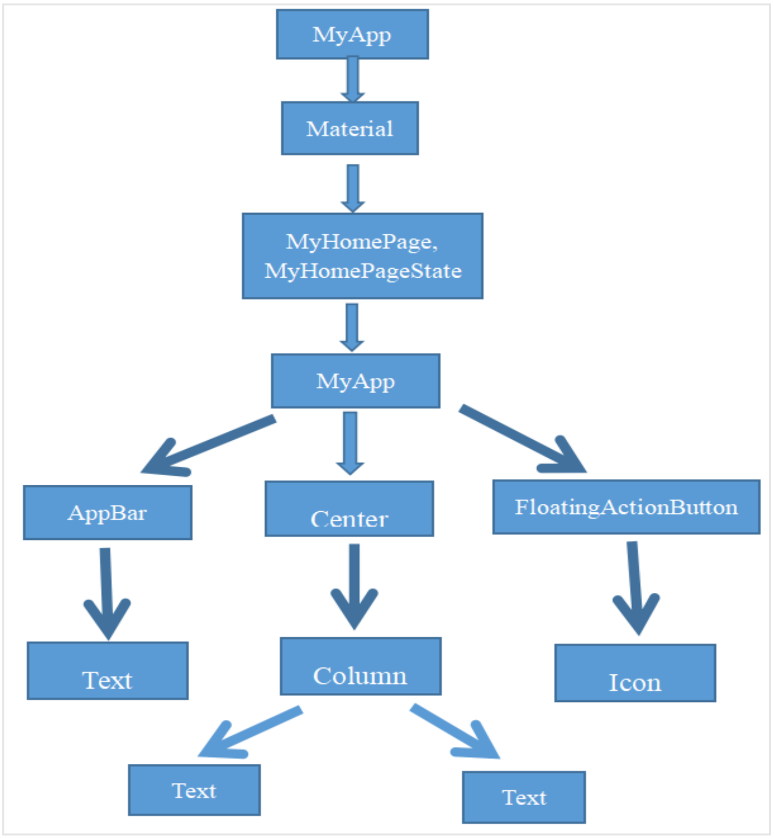


Figure 7. Flutter Program Development Hierarchy Diagram (Syaifudin et al., 2024)

Widgets in Flutter can be categorized into two main types:

a. *StatelessWidget*: The widget's UI representation doesn't depend on any mutable state.
b. *StatefulWidget*: The widget needs to manage some form of mutable state that affects its UI.

The primary responsibility of a Widget is to implement a build function, which defines how the widget should render itself on the screen. Widgets often compose other lower-level Widgets, and this nesting continues until the lowest-level Widgets, known as *RenderObjects*, are reached. *RenderObjects* are responsible for calculating and describing the geometry of the UI elements they represent. This structured approach allows Flutter to efficiently build and manage complex UI layouts by nesting Widgets and utilizing RenderObjects for precise layout calculations.

3. Methodology

The research is executed in chronological phases as shown in Figure 8 as Methodology Framework. Firstly, the researcher has developed a detailed experimental plan to compare and analyze the UI construction of Flutter and Android development tools.

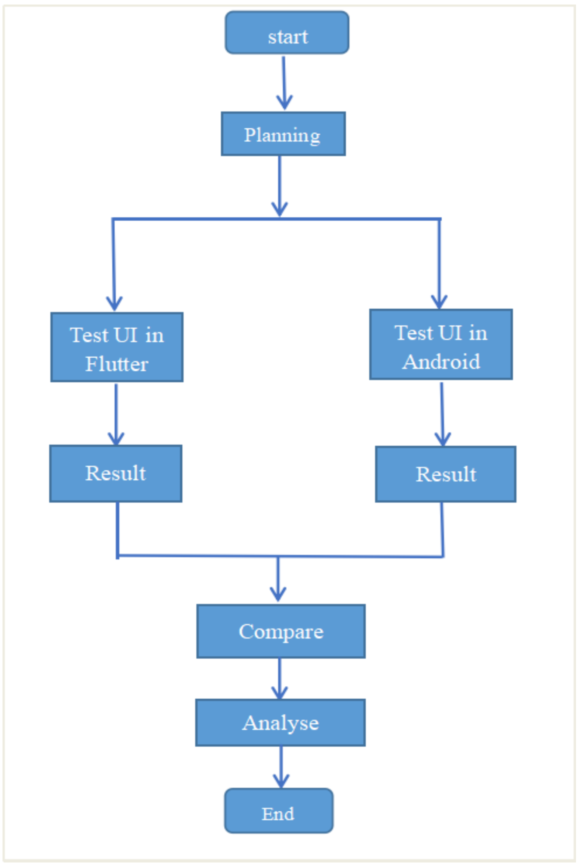


Figure 8. Research Methodology

The researchers integrated the Android Studio development environment and implemented a small module for registration and login functions separately using Java for Android development and Dart for Flutter development. To compare the features of these two frameworks, the study selected three metrics: runtime performance, memory usage, and application size, as criteria for measurement and comparison.

In app development, runtime metrics typically measure an application's time to execute specific tasks while running. This metric is crucial for evaluating the performance and efficiency of the app's code and algorithms during execution. Monitoring runtime metrics helps developers identify bottlenecks, optimize code, and ensure the app meets performance expectations, delivering a smooth user experience.

Memory usage refers to the amount of system memory (RAM) an application consumes while running. Excessive memory usage can lead to performance issues such as slowdowns, freezes, or crashes, especially on devices with limited memory resources like smartphones or tablets. Monitoring and optimizing memory usage is essential for ensuring the app runs smoothly across different devices and platforms.

App size indicates the amount of space an application occupies on a device's storage, typically measured in megabytes. It varies based on the app's complexity, features, and content. App size is critical as it impacts user experience, download speed, and storage capacity. Large app sizes can deter users from installing or updating apps due to slower download speeds and may prevent installation on devices with limited storage. Moreover, larger apps may require more system resources, potentially leading to slower load times and diminished performance, negatively affecting user retention and satisfaction. Therefore, managing app size is vital for optimizing user experience and maximizing app accessibility and performance.

4. Testing & Result

In this research, we compare Android and Flutter app development, focusing on application size, runtime performance and memory usage. Application size refers to the total size of the app, including image resources, while runtime performance is measured using profilers to assess execution efficiency. In Android development, memory usage is a critical aspect to consider to ensure smooth performance and avoid issues like crashes or sluggish behavior.

4.1 Testing UI in Android

In this research, the experiment was conducted using the Android Studio 3.2 integrated development environment. Initially, an XML layout file was created for the registration and login function module. The logical function implementation was then written in *MainActivity*. Finally, the app was downloaded and installed on a Vivo U10 Android smartphone. The running effect diagram is depicted in Figure 9.

During testing, the following results were observed:

a. Application Size: The total size of the application, including image resources is 21.6 MB.
b. Memory usage: The *login.xml* interface layout file specifically occupies 2.65 KB, while image resources contribute 18.8 KB to the overall size.
c. Runtime: The application's runtime, measured using profilers available on the development platforms, is recorded at 19 seconds.

These metrics play a crucial role in evaluating and optimizing the performance and efficiency of the application across various devices and platforms. They help developers understand resource consumption, identify potential optimizations, and ensure a smooth user experience.

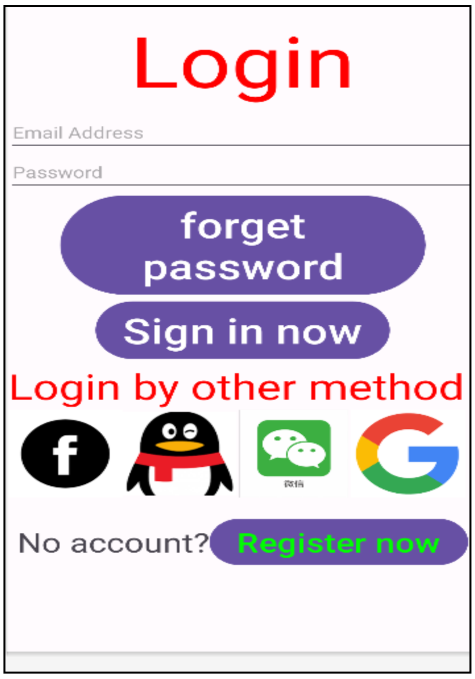


Figure 9. Login Function Module in Android Framework

4.2 Testing UI in Flutter

Researchers tested the Flutter framework version 3.19.5, focusing on application size, runtime, and memory usage. The UI testing effects are shown in Figure 10.

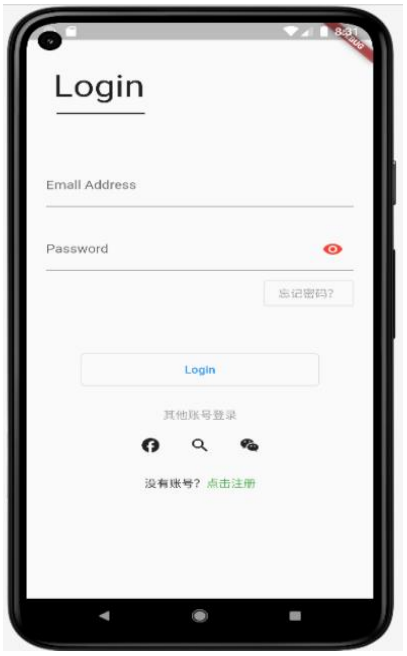


Figure 10. Login Function Module in Flutter Framework

The backend console on the development platform showed that the application runs for a total of 10 seconds. The *login.dart* interface layout file measures 4.45KB, and the entire application size is 93.3MB, including 18.8KB of image resources.

5. Comparative and Analysis

This section discusses the test results obtained after evaluating the applications in Flutter and Android, as presented in Table 1.

Table 1. Experiment Data Comparison

Development framework	Runtime	Memory Usage	App Size
Android	19 seconds	2.65KB	21.6MB
Flutter	10 seconds	4.45KB	93.3MB

Based on the experiment data, Flutter's runtime is shorter than Android's. Flutter generally offers better performance in UI rendering speed compared to traditional Android UI layouts, particularly for complex UIs or animations. This is due to its use of the "Skia" rendering engine, which draws UI elements directly to the screen without relying on the platform's native UI components. Here are a few reasons why Flutter UI rendering might be faster than traditional Android UI rendering:

a. Direct Rendering: Flutter renders UI elements directly to the screen using Skia, bypassing the Android UI toolkit (View hierarchy), eliminating the overhead of traversing the View hierarchy and performing layout calculations.
b. Widget-based Architecture: Flutter's widget-based architecture represents UI components as widgets, efficiently managing the UI hierarchy and optimizing rendering by only redrawing widgets that have changed.

From Table 1, it was found that the file size of *login.dart* in Flutter is slightly larger than *login.xml* in Android, despite using the same image resources. In this research, the comparative data on memory usage refers only to the UI layout file size. Typically, Flutter layout files aren't larger than Android layout files. In this experiment, Flutter uses Dart code to describe UI layouts in the form of .dart files. Additionally, *login.dart* includes the interaction logic code for UI components. Compared to Android's XML layout files, Dart code is often more concise due to the powerful features of the Dart language, enabling the implementation of complex UI layouts and interaction logic more efficiently. In contrast, in native Android, the interaction logic code must be written in the MainActivity.java file.
In this research, the comparative data on app size refers to the space the experimental application occupies on the test device's storage, typically measured in megabytes. From Table 1, it was found that the app size of Flutter is much larger than Android. The larger size of the Flutter APK compared to the Android APK with the same functionality and image resources is primarily due to the additional libraries and runtime required by Flutter applications. The specific reasons include:

a. Flutter Engine: Every Flutter application includes the Flutter engine, responsible for rendering UI and handling interactions, which significantly contributes to the APK size.
b. Dart Runtime: Flutter apps include the Dart runtime, necessary for interpreting and executing the Dart code used in Flutter applications.
c. Embedding Code: Flutter apps contain code to embed the Flutter engine within the native Android app, adding to the APK size.
d. Native Components: While Flutter renders most UI components using its own rendering engine, it still needs to interact with native components for certain functionalities, requiring additional native code and resources.
e. Optimizations: Android Studio and the Android build tools perform various optimizations when building APKs, such as code shrinking, resource optimization, and compression, which may be more effective for native Android apps than for Flutter apps.
f. Multi-Platform Support: Flutter supports multiple platforms, including Android, iOS, web, and desktop. This cross-platform capability means Flutter APKs may include code and resources for multiple platforms, even if the app targets only Android.

6. Conclusion

This study has provided valuable insights into UI construction differences between Android and Flutter frameworks, offering a foundational exploration of Flutter programming for novice developers. While Android's native development simplicity translates to better performance metrics such as smaller app sizes and lower memory consumption, Flutter excels in cross-platform development and rapid UI rendering, albeit with larger APK sizes due to additional runtime and engine components. These findings empower users to select the framework best suited to their specific needs. Moving forward, our research will delve into the nuances of image loading, animations, and other aspects to comprehensively compare Android and Flutter. By expanding our focus, we aim to further aid developers in making informed decisions and mastering these powerful development tools.

Acknowledgement

Many thanks to the College of Computing, Informatics and Mathematics, UiTM, Malaysia

Funding

The author(s) received no specific funding for this work.

Author Contribution

Author1 prepared the literature review and performed the research work, conducted the statistical analysis and interpreted the results. Author2 wrote the research methodology and oversaw the article writing. Author3, Author4 collected experimental data and formatted the manuscript.

Conflict of Interest

The authors have no conflicts of interest to declare.

References

Abdulham, N. F., & Mohamed Noor, N. (2023). Mobile application luggage tracking system using GPS module and NodeMCU. Malaysian Journal of Computing (MJoC), 8(1), 1250-1263.

Allen, G. (2021). Android for absolute beginners: getting started with mobile apps development using the Android Java SDK. Apress.

Ammar, L. B. (2021). An automated model-based approach for developing mobile user interfaces. IEEE Access, 9, 51573-51581.

Boukhary, S., & Colmenares, E. (2019). A Clean Approach to Flutter Development through the Flutter Clean Architecture Package. *International Conference on Computational Science and Computational Intelligence (CSCI),* pp. 1115–1120. https://doi.org/10.1109/CSCI49370.2019.00211

Choudhari, A., Gawai, J., Lekurwale, A., Ranglani, S., & Dhongde, P. (2022, February). A Mobile App for Smart Electricity Usage Monitoring. In 2022 Second International Conference on Artificial Intelligence and Smart Energy (ICAIS) (pp. 1667-1673). IEEE.

Durai, S., Shyamalakumari, C., & Sujithra, T. (2022). Cloud Computing based Multipurpose E-Service Application using Flutter. *In Proceeding 6th International Conference on Computing Methodologies and Communication (ICCMC)*. pp. 1122-1126. https://doi.org/10.1109/ICCMC53470.2022.9753968.

Işıtan, M., & Koklu, M. (2020). Comparison and evaluation of cross platform mobile application development tools. International Journal of Applied Mathematics Electronics and Computers, 8(4), 273-281.

Kishore, K., Khare, S., Uniyal, V., & Verma, S. (2022). Performance and stability Comparison of React and Flutter: Cross-platform Application Development. *International Conference on Cyber Resilience* (ICCR), pp. 1-4.

Krajci, I., & Cummings, D. (2013). History and Evolution of the Android OS. In I. Krajci & D. Cummings (Eds.), *Android on x86: An Introduction to Optimizing for Intel® Architecture* , pp. 1-8. Berkeley, CA: Apress.

Mahmud, Y., Abdul Razak, M.S., Abdul Rahman, S., , Hanafiah, M. &, Suhaimi, A.A. (2023). Mathvision Prototype Using Predictive Analytics. *Malaysian Journal of Computing,* Vol 8(2), pp. 1505-1516. doi:10.24191/mjoc.v8i2.22391

Martinez, D., Ferre, X., Guerrero, G., & Juristo, N. (2020). An Agile-Based Integrated Framework for Mobile Application Development Considering Ilities. in IEEE Access, Vol. *8*, pp. 72461-72470. https://doi:10.1109/ACCESS.2020.2987882

Mayrhofer, R., Stoep, J. V., Brubaker, C., & Kralevich, N. (2021). The android platform security model. ACM Transactions on Privacy and Security (TOPS), 24(3), 1-35.

Nagaraj, K., Prabakaran, B., & Ramkumar, M. O. (2022, October). Application Development for a Project using Flutter. In 2022 3rd International Conference on Smart Electronics and Communication (ICOSEC) (pp. 947-951). IEEE.

Patta A.R, Funabiki N., Lu X. & Syaifudin Y.W. (2023). A Study of Grammar-concept Understanding Problem for Flutter Cross-platform Mobile Programming Learning. *2023 6th International Conference on Vocational Education and Electrical Engineering (ICVEE),* https://doi:10.1109/ICVEE59738.2023.10348237

Perinello, L., & Gaggi, O. (2024). Accessibility of Mobile User Interfaces using Flutter and React Native. *In Proceeding 1st Consumer Communications & Networking Conference (CCNC).* IEEE Xplore: https://DOI: 10.1109/CCNC51664.2024.10454681

Sharma, S., Khare, S., Unival, V., & Verma, S. (2022, October). Hybrid Development in Flutter and its Widgits. In 2022 International Conference on Cyber Resilience (ICCR) (pp. 1-4). IEEE.

Syaifudin, Y. W., Funabiki, N., Kuribayashi, M., & Kao, W. C. (2020). A proposal of Android programming learning assistant system with implementation of basic application learning. International Journal of Web Information Systems, 16(1), 115-135.

Syaifudin, Y. W., Hatjrianto, A. S., Funabiki, N., Liliana, D. Y., Kaswar, A. B., & Nurhasan, U. (2022). An Implementation of Automatic Dart Code Verification for Mobile Application Programming Learning Assistance System Using Flutter. *In Proceeding International Conference on Electrical and Information Technology (IEIT).*

Syaifudin, Y. W., Yapenrui, D. D., Noprianto N., Funabiki, N., Siradjuddin, I., & Chasanah, H. N. (2024). Implementation of Self-Learning Topic for Developing Interactive Mobile Application in Flutter Programming Learning Assistance System. *In Proceeding ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETSIS) 2024,* IEEE Xplore:. https:// DOI:10.1109/ICETSIS61505.2024.10459432

Zehang, C., & Sabran, K. (2023). A systematic literature review of mobile applications to assist people with mild to moderate dementia in their daily lives/Zehang Cheng and Kamal Sabran. Malaysian Journal of Computing (MJoC), 8(1), 1332-1348.