



UNIVERSITY OF PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER THESIS IN ICT FOR INTERNET AND MULTIMEDIA

INTEGRATION OF FLUTTER FRAMEWORK IN REAL-LIFE APPLICATIONS: TECHNICAL AND DEVELOPMENT PRACTICES

SUPERVISOR

PROF. LEONARDO BADIA
UNIVERSITY OF PADOVA

MASTER CANDIDATE

YEŞİM UZUN

ACADEMIC YEAR

2023-2024

GRADUATION DATE

03/07/2024

Abstract

The Flutter framework has become very popular in recent years due to its cross-platform capabilities and fast development processes. Flutter, an open-source SDK allows developers to create reliable applications, for platforms like mobile, web, and desktop using a unified code-base. This study looks into the aspects and procedures involved in incorporating Flutter into real-world applications. The analysis discusses the advantages and obstacles of utilizing Flutter compared to development approaches. It also investigates elements like enhancing performance reusing code and integrating with specific platforms. Additionally, it provides suggestions, on how to create, build and maintain applications using Flutter, in environments. The thesis highlights the hot reload, a unique feature of Flutter, that allows developers to build and iterate quickly. Also, it allows to update code and see changes almost instantly, without losing state. The findings of this study contribute to our understanding of integrating the Flutter framework into software development practices.

Acknowledgments

First of all, I would like to dedicate this thesis to my beloved parents. To my father, who I lost three weeks ago, and to my mother, who I lost eight years ago. Their love, efforts, and teachings shaped the person I am today. Their memories and the values they taught me will always inspire and guide me. Even though they are no longer with me, their love and support have been a source of strength and motivation throughout my journey.

I also express my sincere thanks to my supervisor, Leonardo Badia. During these hard times, your advice, kindness, and help were very helpful. Your encouragement helped me focus and complete this dissertation. I could not have imagined having a better advisor for my master's study.

To my brother Ozcan Uzun, whose unwavering support and guidance mean a lot to me. Your confidence in my skills and constant encouragement have been crucial to my success. You have been my rock of strength, and I appreciate everything you do.

I would also like to express my deepest gratitude to my fiancée, Deniz Balsever. A lot of hard times have been easier for me because you always believed in me and were there for me. Your assistance has been a huge source of strength for me, and I am deeply grateful for your support and affection.

Finally, I want to thank everyone who supported me in this way, especially my friends and coworkers. Thank you to everyone who has supported and helped me get to this stage in my life.

Contents

ABSTRACT	iii
ACKNOWLEDGMENTS	v
LIST OF FIGURES	ix
LISTING OF ACRONYMS	xi
1 INTRODUCTION	I
2 LITERATURE REVIEW	5
2.1 Overview of Flutter Framework	5
2.1.1 History and Evolution	7
2.1.2 Core Principles	8
2.1.3 Architecture	9
2.1.4 Key Features	9
2.1.5 Adoption and Community	10
2.2 Technical Aspects of Flutter	11
2.2.1 Framework and Widgets	11
2.2.2 Flutter's Popularity and Advantages	12
2.3 Development Practices with Flutter	14
2.3.1 Learning and Development with Flutter	15
2.3.2 Coding Conventions and Best Practices	16
2.3.3 State Management	17
2.3.4 Testing and Debugging	25
2.3.5 Kinds of Testing	25
2.3.6 Continuous Integration and Continuous Deployment (CI/CD)	30
2.3.7 Performance Optimization	32
2.3.8 Code Reusability and Modularization	33
2.3.9 Version Control Systems	36
2.3.10 Performance Metrics Comparison: Flutter vs. React Native	39
2.4 Case Studies on Flutter Integration	41
2.4.1 Case Study 1: Alibaba's Xianyu App	42
2.4.2 Case Study 2: Google Pay	43

3	METHODOLOGY	45
3.1	Performance and Speed	45
3.2	Performance Optimization with Method Channels	46
3.3	Chat Screen Performance Metrics	49
3.4	Integrating Flutter Projects as a Package In iOS Native Applications	50
3.4.1	Modularization and Integration Ease	50
3.4.2	React Native or Flutter	52
3.5	Overview of Results	53
3.5.1	Performance Metrics	53
3.5.2	Resource Efficiency	53
3.5.3	Discussion on Implications	53
4	CONCLUSION	55
	REFERENCES	57

Listing of figures

2.1	Execution time, energy and memory data running the benchmarks [1].	6
2.2	List of technologies [2]	7
2.3	Architecture of the Flutter Framework [3].	10
2.4	Example of Stateless Widget	12
2.5	Example of Stateful Widget	13
2.6	Interest Over Time [4]	14
2.7	Technologies have trended over time [5]	15
2.8	Example of setState [6]	18
2.9	BloC State Management [7]	20
2.10	The Redux components [8]	22
2.11	The final comparison table of approaches [9]	24
2.12	Unit Test Example	26
2.13	Widget Testing [10]	26
2.14	Counter app - Part 1	27
2.15	Counter app - Part 2	28
2.16	Widget Test Example	29
2.17	CI/CD Pipeline [11]	30
2.18	Example of Modularization	33
2.19	Example of Reusability	35
2.20	Comparison between CVCS and DVCS [12]	36
2.21	Benefits of Version Control [13]	38
2.22	Performance Metrics [14]	39
2.23	Performance comparison Between Flutter and React Native [15]	40
2.24	[16]	40
2.25	Companies That Use Flutter [17]	42
2.26	Flutter's benefit of Alibaba [18]	43
2.27	Google Pay [18]	44
3.1	Architectural overview: platform channels [19].	46
3.2	notificationActionService.swift	47
3.3	notificationActionService.kt	48
3.4	Chat Screen Performance Metrics	49
3.5	Dependencies of the Flutter Project	51
3.6	package.swift	52
3.7	Swot Analysis	54

Listing of acronyms

SPM	Swift Package Manager
CI/CD	Continuous Integration and Continuous Deployment
UI	User Interface
CPU	Central Processing Unit
CLBG	Computer Language Benchmark Game
SDK	Software Development Kit
API	Application Programming Interface
AOT	Ahead of Time
OOP	Object Oriented Programming
MAUI	Multi-platform APP UI
CVCS	Centralized Version Control Systems
DVCS	Distributed Version Control Systems
CVS	Concurrent Versions System
SVN	Subversion
GPU	Graphics processing unit item[IoT] Internet of Things

1

Introduction

”Mobile app development frameworks lower the effort to write and deploy apps across different execution platforms. At the same time, their use may limit native optimizations and impose overhead, increasing resource usage.” [1] As technology changes quickly these days, there is a huge need for efficient, cross-platform mobile app creation. Developers and companies search for strategies to reduce development time and expenses, they also want to create apps that fit various platforms and are simple and effective. This has resulted in the development of several cross-platform frameworks; Flutter is one of the most prominent contenders.

Flutter enables developers to create natively compiled applications for mobile, web, and desktop from a single codebase. It is particularly useful in developing modern mobile systems with real-time applications due to its fast performance and expressive UI capabilities. One main application of Flutter is to build robust real-time messaging apps, leveraging WebSockets for instant message delivery [20]. In the same line, Flutter can be used to integrate live streaming and video conferencing applications [21], as standalone elements or as parts of Web-based Interactive Collaborative Environments, [22], i.e., online platforms that enable multiple users to work together in real-time, regardless of their physical locations. These environments facilitate seamless collaboration through tools like shared documents, real-time messaging, video conferencing, and interactive whiteboards.

They use web technologies to allow users to edit, comment, and discuss content simultaneously, providing live updates, user presence indicators, and version control [23]. A toolkit for real-time interactive collaborative environments is essential for building applications that support seamless, distributed, and interactive user experiences for collaborative editing, real-time communication, and data synchronization, with conflict resolution and real-time notifications[12].

All of these developments can possibly leverage edge computing frameworks, i.e., systems that process data closer to where it is generated, rather than relying on centralized cloud servers, which is essential for enabling real-time interaction for optimal performance and reliability [24]. With these capabilities, developers can create robust and efficient real-time collaborative applications.

Flutter can also be used to create mobile apps that interact with IoT (Internet of Things) devices, providing real-time monitoring and control (e.g., smart home systems [25] or vehicular control [26]). Real-time navigation or ride-sharing apps can be built with Flutter, using its capabilities to handle real-time location tracking and updates. Similarly, financial apps that require real-time stock prices, trading data, and financial news need up-to-date information for a smooth and responsive user experience [27].

A very successful application of Flutter is the development of real-time multiplayer games with seamless performance, where multiple players interact with each other simultaneously in a shared virtual environment without noticeable delays or interruptions. These games rely on efficient networking protocols, low-latency communication, and robust server infrastructure to ensure that actions taken by one player are instantly reflected for all other players. Advanced synchronization techniques, predictive algorithms, and real-time data exchange ensure a smooth and immersive gaming experience[28]. More in general, Flutter can enable the access to the Metaverse for real-time applications, either related to multiplayer gaming, but also serious games, and virtual reality applications for digital twinning [29].

Finally, Flutter is also important for e-health applications [30]. Real time patient monitoring apps that provide instant health data updates to doctors and caregivers can be developed using Flutter, especially thanks to features such as live chats or AI-driven chatbots [31].

To sum up, Flutter's ability to deliver high performance, combined with its expressive and flexible UI, makes it an excellent choice for developing multiple real-time applications across various domains.

A comparison examination of Flutter and other popular frameworks, such as React Native and Ionic, was carried out for the purpose of this thesis. I compared the performance metrics, resource use, and development efficiency of each framework. Analyzing many case studies and actual implementations in this work helps one to have an overview of the benefits and difficulties of using Flutter in real-world scenarios.

2

Literature Review

2.1 OVERVIEW OF FLUTTER FRAMEWORK

Flutter, an open-source UI software development toolkit developed by Google allows developers to create applications, for mobile, web and desktop using a codebase [32]. Since its launch in May 2017 Flutter has become popular, among developers for its structure user interface and wide range of tools available [33]. Flutter is a robust and flexible framework for developing cross-platform apps, making it a preferred choice for many development projects. The Flutter framework revolves around the use of the Dart language. Built on top of Dart Flutter is known for its user interface and efficient performance.

”Our results show that cross-platform and hybrid frameworks can be competitive in CPU-intensive applications. In five of the ten benchmarks, at least one framework-based version exhibits lower energy consumption and execution time than its native counterpart. Overall, Flutter usually imposes the least overhead in execution time and energy, while React Native imposes the highest in all the benchmarks” [1]. This study shows how efficiently Flutter uses resources, which makes it a good choice for developers who care about speed and energy use. With the recent addition of web support and its adoption by various industries, Flutter’s future prospects in the app development landscape are bright.

The adoption of Flutter by various industries, including finance, e-commerce, and entertainment, highlights its versatility and potential. Notable companies such as Alibaba, BMW, and Google itself have successfully used Flutter to develop high-quality applications [34].

To provide a comprehensive comparison of various frameworks, I referenced the performance and resource usage data presented by Oliveira et al [1]. Figure 2.1 shows how long it takes to run the Computer Language Benchmark Game (CLBG) benchmarks, how much energy it uses, and how much memory it uses for Java, Flutter, React Native, and Ionic. Each chart has its own scale. This study provides us with several of important insights regarding these frameworks' effectiveness. "Regarding energy consumption, Java had lower energy consumption than Flutter in eight out of the twenty benchmark-input pairs, while Flutter had lower energy consumption than Java in ten pairings. Ionic had lower energy consumption than Java in only two pairs. When analyzing memory utilization, Java had the lowest consumption in fifteen instances, while Flutter had the lowest consumption in one instance, and Ionic had the lowest consumption in four instances. With the exception of two situations, React Native exhibited the highest energy consumption. In terms of execution time, Java was faster in ten of the twenty sets of benchmark inputs, Flutter was faster in eight, and Ionic was faster in two. Every single measure that React Native was able to finish was the slowest." [1]

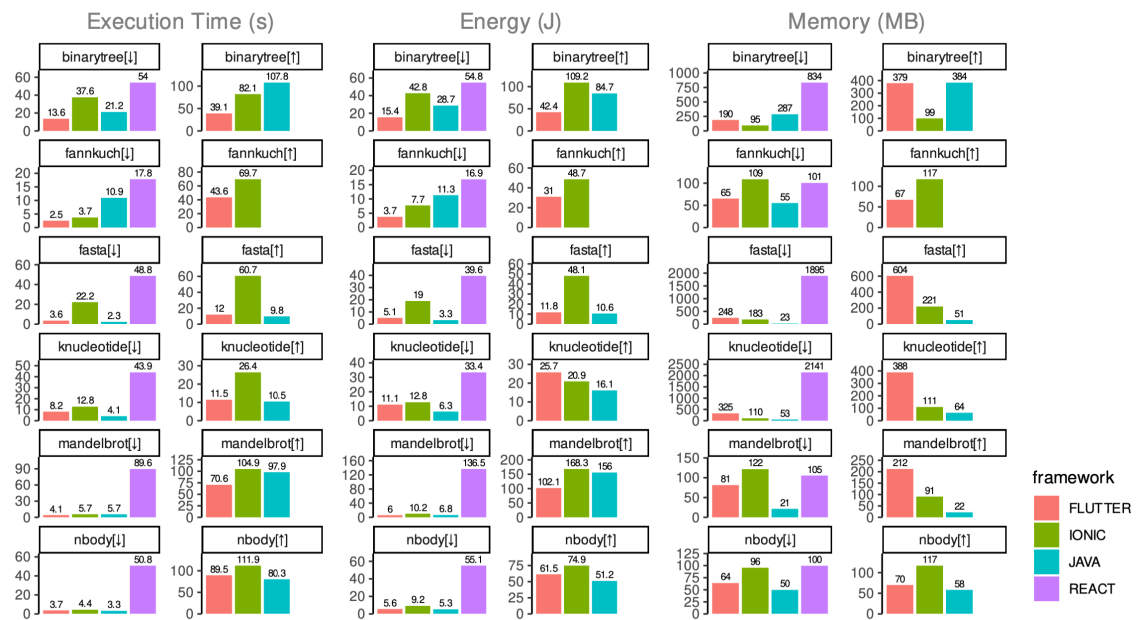


Figure 2.1: Execution time, energy and memory data running the benchmarks [1].

The research by [2] looks into performance overhead in cross-platform mobile development frameworks in great depth. When making mobile apps, using cross-platform tools might make them run less quickly than when developed using the native development method.

Framework	Version	Associated approach	Programming language	APK size
Ionic	v3.9.2	Hybrid (<i>Cordova-based</i>)	TypeScript	10.3MB
React Native	v0.53.2	Interpreted	JavaScript	9.7MB
NativeScript	v3.4.1	Interpreted	JavaScript	30.2MB
Flutter	v0.5.1	Cross-compiled	Dart	32.8MB
MAML / MD ₂	v2.0.0	Model-Driven Development	DSL	3.2MB
Native Android	–	Native	Java	2.7MB

Figure 2.2: List of technologies [2]

”Figure 2.2 lists six technologies which have been used in the development of the artifacts. Of these, one belongs to the Native approach, i.e., it does not support cross-platform deployment. It serves as the baseline benchmark. The remaining five technologies allow for the creation of iOS and Android apps based on a common code base. They vary in terms of programming language, associated development approach, industry adoption, among other aspects”. [2] On some criteria, however, some cross-platform frameworks can outperform native, emphasizing the significance of clearly specified technical requirements and specifications for the thoughtful selection of a cross-platform framework or overall development approach.

2.1.1 HISTORY AND EVOLUTION

Flutter was launched with the goal of creating a tool for building high-quality user interfaces across platforms. The first version of Flutter was known as ”Sky” [35] and ran on the Android operating system. It was first introduced in 2015 but gained significant attention when Google released its first commercial Flutter app in 2017. Since that time it has seen advancements in popularity. Has progressed further enhancing its reliability and features. At the Google Developer Days keynote in Shanghai in September 2018, Flutter Release Preview 2 was announced by Google. Flutter 1.0 was then released on December 4, 2018, during the Flutter event. Subsequently, on December 11, 2019, Flutter 1.12 made its debut at the Flutter Interactive event [36]. On May 6, 2020, the Dart software development kit (SDK) version 2.8 and Flutter 1.17.0 were launched, introducing Metal API compatibility along, with Material wid-

gets and network monitoring tools, for developers. On March 3, 2021, Google released Flutter 2 during an online Flutter Engage event. It added a Canvas Kit renderer for web as opposed to the HTML version before it, web specific widgets, early-access desktop application support for Windows, macOS, and Linux and improved Add-to-App APIs [37]. It also shipped with Dart 2.0 which included partial null-safety, which caused many breaking changes and issues with many external packages; however, the Flutter team included instructions and tools to mitigate these issues [38]. In the following releases more features have been added performance has been enhanced and support, for platforms has been expanded, turning Flutter into a framework, for creating contemporary applications.

2.1.2 CORE PRINCIPLES

The core principles guiding Flutter's design include:

1. **Single Codebase:** In general, cross-platform development uses a single code base that can be executed on multiple platforms. Platforms in this sense typically refer to different operating systems provided by software or hardware vendors, e.g., Android, or iOS. In addition, device fragmentation might cause different versions of the same underlying operating system to be considered as distinct platforms [2]. Flutter allows developers to write code once and deploy it across multiple platforms, including iOS, Android, web, and desktop. This approach significantly decreases the time and effort required for development.
2. **Expressive and Flexible UI:** Flutter's widget-based architecture enables developers to create highly customizable and expressive user interfaces. Widgets are the building blocks of a Flutter application, providing a consistent and flexible way to design and layout UI elements [3].
3. **Fast Development Cycles:** Flutter's hot reload feature allows developers to see the results of their code changes in real-time without restarting the application. This capability enhances productivity and facilitates rapid prototyping and iteration (Flutter, n.d.) When building in debug mode, a Flutter app additionally contains the Dart VM needed for enhanced developer experience, including functionality such as hot reload [2].
4. **Native Performance:** Flutter apps are compiled directly to native ARM code using Dart's Ahead-of-Time (AOT) compilation, ensuring smooth performance and high responsiveness. This approach eliminates the performance overhead typically associated with interpreted languages [3].

2.1.3 ARCHITECTURE

Flutter's architecture consists of three main layers:

1. **Framework:** Flutter's top layer provides many pre-designed widget sets, libraries and APIs for building user interfaces. The framework is built with the Dart language, which offers a robust and flexible environment for developing Flutter applications.
2. **Engine:** The engine is responsible for rendering the application, managing input, and handling platform-specific functionalities. Written in C++, the engine utilizes Skia, a 2D graphics library, to provide fast and efficient rendering.
3. **Embedder:** The embedder is the platform-specific layer that integrates the Flutter engine with the underlying operating system. It handles tasks such as input/output, event handling, and platform-specific plugins, enabling Flutter to run seamlessly on different platforms.

2.1.4 KEY FEATURES

1. **Widgets:** Flutter widgets are constructed using a modern framework inspired by React. The core idea is to create your UI out of widgets. Flutter's widget system is the cornerstone of its architecture, allowing developers to create complex UIs using a composable approach. Widgets can be stateless or stateful, providing a clear separation between the UI and application logic.
2. **Hot Reload:** Flutter framework designed to provide a fast application development experience. For this purpose, it has two important features, such as Hot Reload and Hot Restart. This feature allows to developers to make changes to the code and see the results instantly without losing the current application state. This speeds up the process of finding bugs in the code Hot reload substantially accelerates the development process and increases developer efficiency.
3. **Dart Language:** Flutter uses Dart, a modern, object-oriented programming language developed by Google. Dart's features include an extensive standard library, robust typing and asynchronous programming support, making it ideal for developing high-performance applications.
4. **Cross-Platform Support:** Flutter simplifies cross-platform development. Rather than creating separate code for each platform, developers can take advantage of Flutter's single codebase. This reduces the effort needed to produce versions of an application for both Android and iOS.

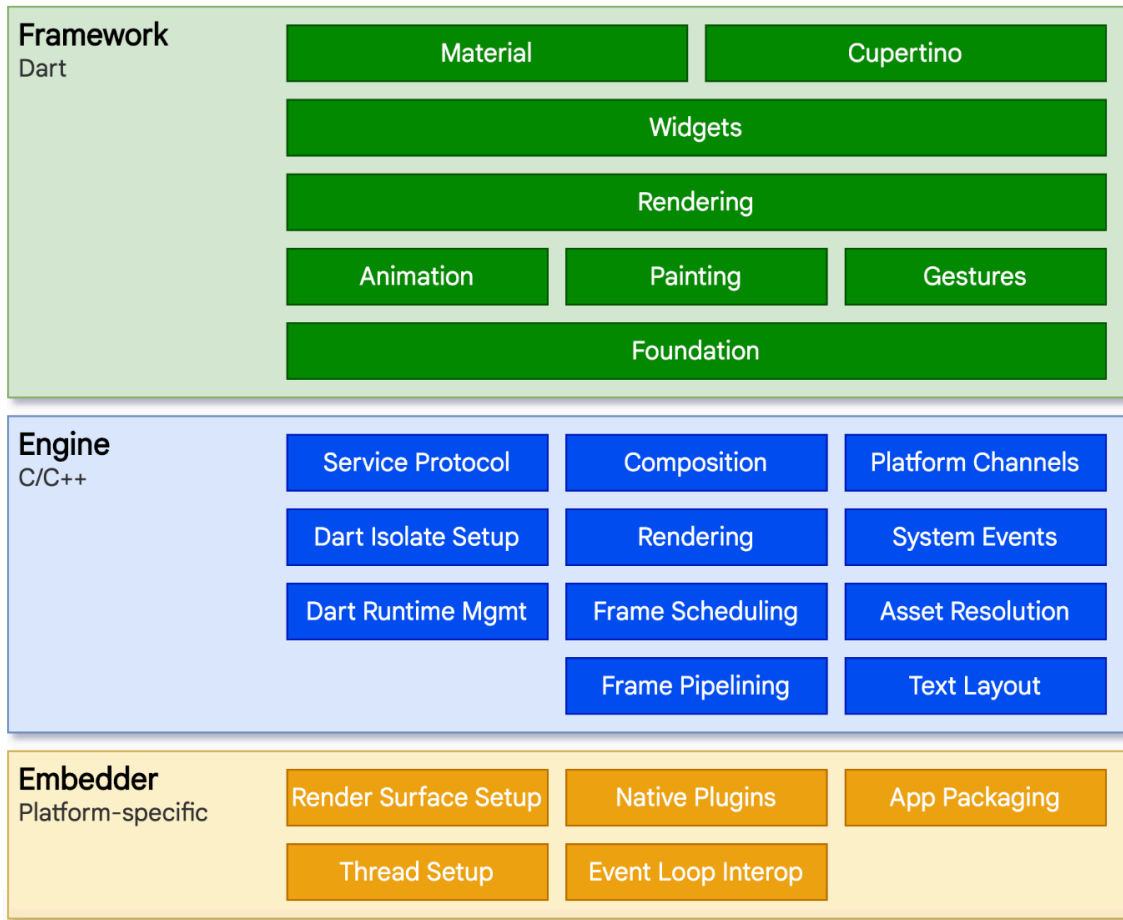


Figure 2.3: Architecture of the Flutter Framework [3].

5. **Extensive Libraries and Plugins:** Flutter’s ecosystem has a large number of libraries and plugins that extend its functionality. This reducing overall coding time. And reduced coding time leads to reduced development costs. The Flutter community actively contributes to this ecosystem, providing solutions for various use cases and integrating with third-party services.

2.1.5 ADOPTION AND COMMUNITY

Flutter is open source, it allows developers from around the world to interact and innovate, enabling continuous improvements to the framework. The framework’s community helps with its development, provides educational materials, and provides support to other develop-

ers. Flutter has been popular across industries, with many important companies and developers using it for their projects. Many well-known companies have successfully implemented Flutter into their development processes. For instance, Alibaba, one of the world's largest e-commerce companies, uses Flutter to power parts of its mobile app, allowing for a seamless and consistent user experience across Android and iOS platforms. The famous automotive manufacturer BMW has also adopted Flutter to create digital experiences in its vehicles. Google itself has been using Flutter in many of its products. Thus confirming its reliability and functionality. Products like Google Ads and Stadia use Flutter to deliver smooth, responsive user interfaces. In addition to technical contributions, the Flutter community helps provide educational resources. Numerous tutorials, documentation, and courses are available online, helping new developers learn Flutter quickly and efficiently. YouTube, Medium, Udemy and many blogs provide a wealth of information about Flutter development techniques, information and case studies.

2.2 TECHNICAL ASPECTS OF FLUTTER

2.2.1 FRAMEWORK AND WIDGETS

Flutter's framework is built around the concept of widgets, which are the core building blocks of a Flutter application. Widget is a blueprint that tells Flutter how to build a particular UI element, be it a simple title, a button, or even a complex layout structure. Widgets can be stateless or stateful, allowing developers to create highly dynamic and interactive user interfaces. The framework includes a rich set of pre-designed widgets for both Material Design and Cupertino (iOS-style) applications. This allows developers to maintain the platform-specific look and feel while using a single codebase.

- **Stateless Widgets:** A widget can be stateless or stateful. Stateless widgets are static widgets that describe a part of the UI which is immutable. Once you construct a stateless widget, it never changes. Stateless widgets are useful for elements that do not change over an app's lifecycle. Examples of stateless widgets include text labels, icons, and static images. Stateless widgets are defined by extending the StatelessWidget class and implementing the build method, which returns the widget's layout. In Figure 2.2, The Text widget is a simple example of a Stateless widget. The text stays the same once its style has been defined [39].
- **Stateful Widgets:** Stateful widgets are dynamic. They are mutable and can change their properties during the lifetime of the application. These widgets are suitable for parts of

```

1
2  import 'package:flutter/material.dart';
3
4  class MyStatelessWidget extends StatelessWidget {
5      @override
6      Widget build(BuildContext context) {
7          return Text(
8              'Hello, Flutter!',
9              style: TextStyle(fontSize: 24),
10         );
11     }
12 }

```

Figure 2.4: Example of Stateless Widget

the UI that need to update dynamically based on user interactions or other changes in the application's state. A widget's state is stored in a State object, separating the widget's state from its appearance. The state consists of values that can change, like a slider's current value or whether a checkbox is checked. When the widget's state changes, the state object calls `setState()`, telling the framework to redraw the widget. In Figure 2.3 there is an example of a simple stateful widget in Flutter [39].

2.2.2 FLUTTER'S POPULARITY AND ADVANTAGES

Flutter, an open-source framework from Google, enables developers to construct feature-rich, cross-platform applications with a single code base, saving time and resources. Its primary benefit is the ability to deliver a highly interactive and visually appealing user interface (UI) with fully customized widgets [23]. Flutter's 1.0 version was released in December 2018, about 6 years ago. Currently, Flutter officially supports six platforms [40]:

- Mobile: iOS and Android
- Desktop: Windows, MacOS, Linux
- Web


```

1  import 'package:flutter/material.dart';
2
3  class MyStatefulWidget extends StatefulWidget {
4      @override
5      _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
6  }
7
8  class _MyStatefulWidgetState extends State<MyStatefulWidget> {
9      int _counter = 0;
10
11     void _incrementCounter() {
12         setState(() {
13             _counter++;
14         });
15     }
16
17     @override
18     Widget build(BuildContext context) {
19         return Column(
20             mainAxisAlignment: MainAxisAlignment.center,
21             children: <Widget>[
22                 Text('Counter: $_counter'),
23                 ElevatedButton(
24                     onPressed: _incrementCounter,
25                     child: Text('Increment'),
26                 ),
27             ],
28         );
29     }
30 }

```

Figure 2.5: Example of Stateful Widget

The graph titled "Interest over time Figure 2.6" shows the relative popularity of various technologies over a given period. This is obtained from Google Trends. It reflects the search volume for each technology, indicating the amount of interest among developers and technologists. Flutter's Rapid Rise, the blue line representing Flutter shows a significant spike in interest starting around September 2021. This sharp increase shows that Flutter is increasingly recognized and adopted among the developer community. Flutter's ability to provide a unified codebase for multiple platforms, coupled with its rich set of customizable widgets and strong

performance, is likely contributing to this rapid rise. The red and yellow lines represent other competing technologies. While these technologies maintain a relatively stable level of interest, their growth is not as pronounced as Flutter's. This stability may indicate established user bases with consistent usage, but perhaps a slower rate of new adoption compared to Flutter. This graph is really good in understanding general trends about the adoption of technology. The trend in a sharp rise in adoption rates of Flutter points to a present upward surge in uptake of the framework, thereby ranking it at the top for mobile and cross-platform development. For a developer or an organization, the trend of Flutter's popularity says that there will be a continuously increasing pool of Flutter resources, community support, and job opportunities. As observed from the statistics of interest varying with time, Flutter has already influenced the developer community to a huge extent. Flutter's rapid adoption and growing popularity show us how useful and promising it is for real-world uses.

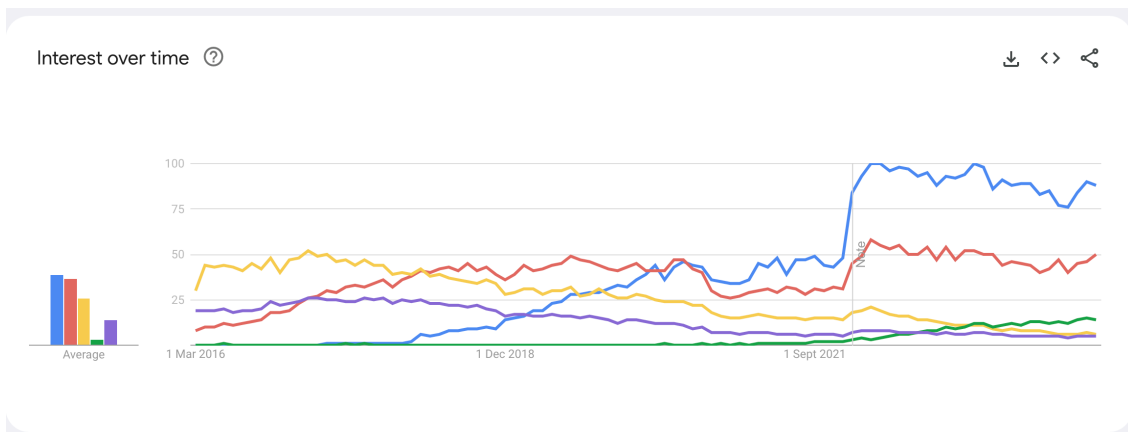


Figure 2.6: Interest Over Time [4]

2.3 DEVELOPMENT PRACTICES WITH FLUTTER

Following best programming practices becomes more crucial as software development gets more complex and technology advances. Developers can write high quality code that is easy to read, maintain and robust by adopting the best practices. Besides that, according to best practices will help you to have better developers collaboration and more efficient workflow because everyone is following the same coding standards and norms.

2.3.1 LEARNING AND DEVELOPMENT WITH FLUTTER

Before you move to Flutter, make sure you understand Dart! Without a doubt, understanding a programming language should be the first step towards mastering Flutter. It's Dart in this instance. Google created the programming language Dart, which was created by Lars Bak and Kasper Lund [41]. It can be utilized to create desktop, server, and mobile applications in addition to online applications. Dart uses Object-oriented programming (OOP) concepts with C#-style syntax, so to fully understand all the mechanisms of a language, it is crucial to know and understand these concepts [42]. Dart is a clean, simple class-based, object-oriented language with more structure than JavaScript, which it is strongly based on. It's ideal for developers who want a structured programming language that allows them to easily restructure and build huge web apps [42]. Without it, we will be unable to develop a Flutter application, such as when using the REST API [43].

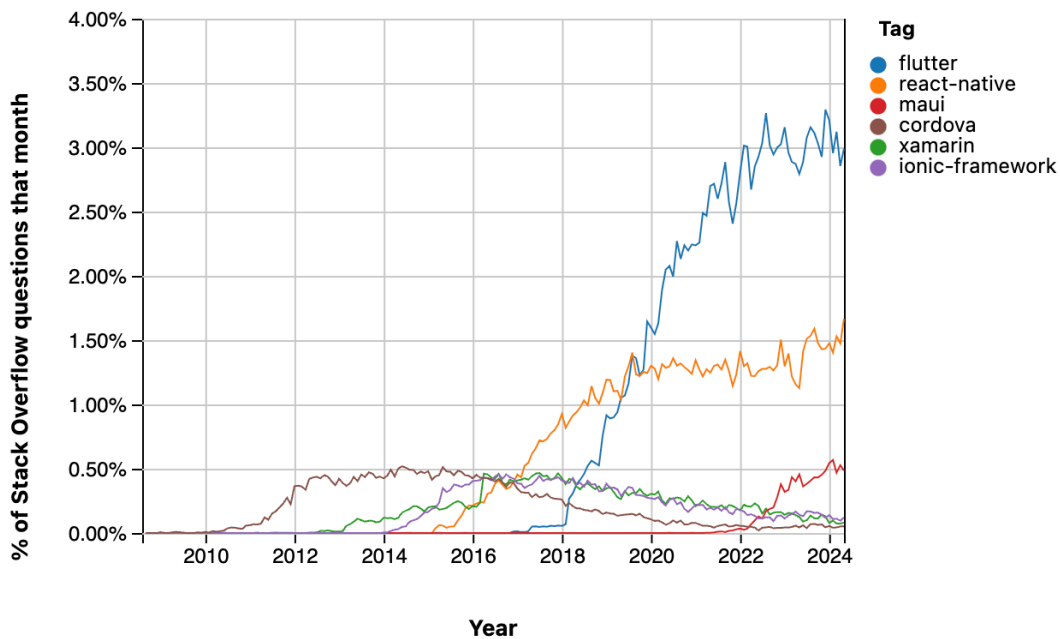


Figure 2.7: Technologies have trended over time [5]

The figure 2.7 shows the percent of questions in Stack Overflow, as of date, that are tagged to a plethora of mobile development frameworks like Xamarin, Ionic Framework, Cordova, Re-

act Native, Flutter, and MAUI. This could be taken as a comparator to fill in the popularity and community involvement of each of these frameworks. The blue line representing Flutter shows a strong and rapid increase in the percentage of Stack Overflow requests beginning around 2018. A case in point is in, By 2024 more than 4 percent of all Stack Overflow questions will be Flutter-related, evidentially shows how readily the platform has remained relevant to that mobile development landscape.

React Native shows a steady increase in questions with the orange line, peaking around 1.5% to 2%. The red line representing MAUI (Multi-platform App UI) starts to grow around 2022. The recent surge shows that developers start to explore and adopt MAUI; using the same logic, it still is Flutter and React Native in terms of general popularity. The green and brown lines for Cordova and Xamarin respectively show a decline over time. This pattern indicates that interest is decreasing, or that developers have moved to newer frameworks such as Flutter and React Native. The purple line representing the Ionic Framework remains reasonably low and stable. While it still has a small user base, it is not growing as quickly as Flutter or React Native. According to this figure, it shows a trend in developer preferences for newer and more efficient frameworks such as Flutter. This trend underscores the industry's shift to frameworks that provide faster development cycles and improved performance. Additionally, it reflects a move towards frameworks that offer a more cohesive development experience.

2.3.2 CODING CONVENTIONS AND BEST PRACTICES

Adopting consistent coding conventions is essential for maintaining code quality and readability in Flutter projects. In this sub section I am talking about following the standard practices such as using Dart style guide, naming conventions, and creating a modular structure for the project. Following these rules helps to make sure that the codebase is clean, maintainable and easy to understand for all team members.

- **Dart Style Guide:** Following the Dart style guide is a fundamental practice in Flutter development. The Dart style guide gives you detailed advice on writing easy to read, clean and maintainable code. The important keys of the Dart style guide include proper indentation, consistent use of whitespace, and clear naming conventions for variables, functions, and classes. Following these steps ensures that your project has a consistent coding style and different developers who may have different opinions working on the project can easily read each others code.

- **Meaningful Naming Conventions:** Following the correct naming conventions as you develop makes your Flutter code easier to read. Taking these proper coding guidelines and tips into account when coding may seem time-consuming and laborious at first, but it will save you a lot of time in the long run. Using meaningful naming conventions in the flutter codes plays a crucial role in improving code readability. The names of variables, functions and classes should be descriptive and convey the purpose or function of the entity they represent. There are some tips for naming conventions in Flutter. For instance, you can use snake_case (lowercase with underscore) for libraries, directories, packages and source files. Use UpperCamelCase for classes, extension names, types, and enums and use lowerCamelCase for constants, variables, parameters, and named parameters. For special variable names, start with an underscore. Additionally, to make code easier to comprehend, always give names a clear and significant meaning. Naming conventions also help avoid confusion and make the code readable in some sense without comments or additional explanations.
- **Project Structure and Modularization:** Another best practice is to organize the project in a modular way, which helps the maintainability of the codebase. Separate concerns and related functionality should be grouped together in modules or packages. Organizing the project in this way makes the codebase more manageable and scalable so that developers can work on individual modules without affecting other parts of the project. In Flutter, it is common to organize the project into directories such as lib, src, widgets, models, services, assets, helpers and utils, each serving a specific purpose.
- **Code Documentation:** One of the important considerations in coding conventions should be proper documentation. Code should have comments and be properly documented. Comments are added to explain complex logic, what a particular piece of code is trying to accomplish so that there is no doubt for future you or the developer who may work on your project. A good practice is to use Dart's documentation comments to create API documentation. Inline comments can be useful to explain a part of the code that is not immediately obvious.

2.3.3 STATE MANAGEMENT

Managing state is key when building responsive, easy to maintain Flutter apps. State management is a way to handle and maintain state of an application or data that might change. There are multiple state management options. Choosing the right state management solution can significantly impact the complexity and performance of an application. In this sub-section, we will look into different state management solutions available in Flutter. Starting from the most

basic `setState` we will look into advanced solutions like Provider, Riverpod, Bloc and Redux. We will also learn when to use them and their pros and cons if any.

setState

This is the most basic and simple way to manage state in Flutter. `setState` method notify the framework that the internal state of this object has changed. Flutter re-renders the widget with the updated state. `setState` is ideal for managing local state within a single widget or a small subtree of widgets. It is best suited to small applications or cases in which state changes do not need to be exchanged between several widgets.

There are advantages of using `setState` such as easy to implement and understand. Also, you do not need to any additional dependencies required for it. It has advantages as well as disadvantages. It is unsuitable for handling complicated states or states that must be shared by several application components. `setState` can lead to tightly coupled code and potential performance issues if over used [6].

```
setState(() { _myState = newValue; });
```

Figure 2.8: Example of `setState` [6]

Provider

In this section, I will talk about the Provider architecture recommended by the Flutter team and frequently used by the Flutter developer community. The Provider package was created by Remi Rousselet. It aims to handle the state as cleanly as possible. It is a state management solution that is built on top of `InheritedWidget`.

It allows developers to manage state in a more scalable and decoupled manner by providing a way to pass state down the widget tree. In Provider, widgets listen to changes in the state and update as soon as they are notified. Thus, when a state change occurs, only the impacted widget needs to be rebuilt rather than the entire widget tree, which saves effort and improves

the speed and functionality of the application.

Provider is suitable for medium to large applications where state needs to be shared across multiple widgets. Provider separates state management from the user interface and makes application code more modular. It also supports dependency injection and makes it possible to easily access state from anywhere in the widget tree. This feature makes Provider well integrated with other Flutter libraries and tools, thus making the development process more efficient and flexible.

Provider can result in a lot of boilerplate code. So this can create unnecessary complexity, especially for small and medium-sized applications. Additionally, the learning curve may be steeper as it requires familiarity with the Provider pattern and concepts. For these reasons, Provider may not always be the most appropriate solution and other state management solutions may be more practical depending on the needs of the application.

Riverpod

Riverpod is a modern state management solution. Riverpod builds on the concepts of Provider but offers additional features and improvements. It is designed to be simpler, safer, and more testable than Provider. Riverpod is a reactive caching framework for Flutter/Dart. It can automatically fetch, cache, combine, and recompute network requests while also handling errors [44]. Riverpod is well for applications of all sizes. Particularly those requiring a more robust and scalable state management system. It is suited for tasks where testability and modularity are critical.

Riverpod eliminates common issues like context scoping in Provider and supports a more modular approach to state management. Additionally, it increases testability and makes it easier to write unit tests, which helps developers make their code more reliable and maintainable. These features make Riverpod a more flexible and powerful state management solution.

Riverpod is newer and less mature compared to Provider, so it may have fewer resources and community support. It also requires learning new concepts and patterns. This can lead to a steeper learning curve. Therefore, some developers might find it more challenging to use initially.

Bloc (Business Logic Component)

The BLoC (Business Logic Component) pattern is a robust state management solution for Flutter that has grown in popularity among developers because to its ability to split business logic from the user interface layer, making the codebase more intuitive, maintainable, and testable. BLoC depends on reactive programming techniques, which use streams and sinks to manage an application's state and events.

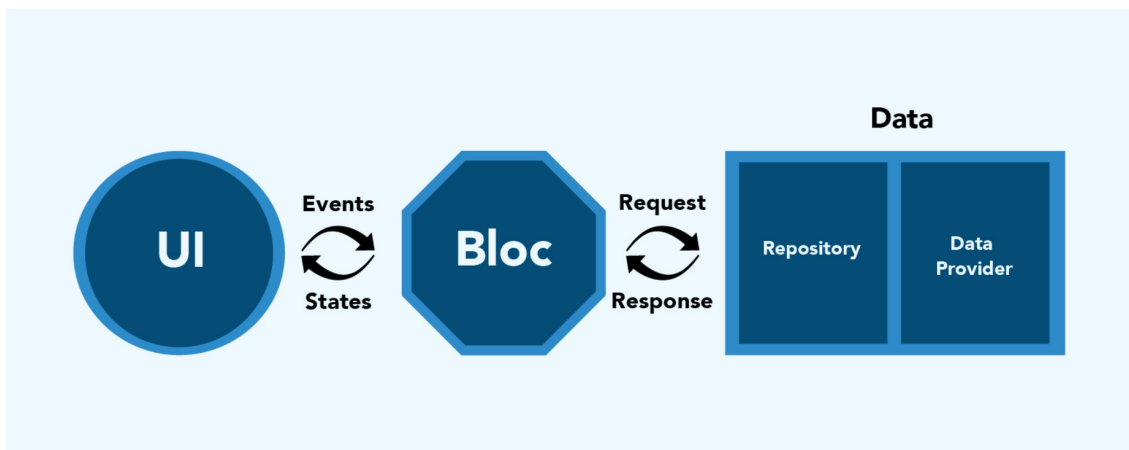


Figure 2.9: BloC State Management [7]

Let's talk about Core Components of the BLoC Pattern:

- **Events:** Events are triggered by user interactions or other actions in the app. They serve as the input to the BLoC, initiating state changes.
- **States:** States are the output of a BLoC. They represent the current state of the application and are consumed by the UI to reflect changes.
- **BLoC:** The BLoC itself acts as an intermediary that processes events and produces states. It contains all the business logic required to handle these events and determine the appropriate state.

The program is more modular and testable as a result of the distinct separation of concerns that guarantees the business logic is kept separate from the UI code. Reactive programming concepts are necessary to comprehend the BLoC paradigm because it primarily uses streams for state management [45].

The BLoC (Business Logic Component) pattern follows a set of well-defined steps to achieve a clear separation between business logic and UI components. A user interaction such as a button press, triggers an event within the application. This event is then sent to BLoC, which acts as an intermediary by processing the incoming event. BLoC uses business logic to handle the event and then creates a new state based on the result. The UI then listens for these state changes and emits this new state. When a new state is published, the user interface automatically updates to reflect the current state and ensuring that the interface is always synchronized with the underlying data and logic. This structured approach improves the application's testability and flexibility to maintain it while providing a reactive programming style that is appropriate in cases of complex case management and real-time changes.

There are advantages of BLoC. For instance, BLoC creates a more modular and manageable codebase by keeping business logic and user interface apart. Because it leverages streams, BLoC is a good fit for applications that need real-time data updates. Also, BLoC manages state efficiently in both small and big applications. Furthermore, because of its vibrant community and variety of information, BLoC makes it simpler for developers to understand and apply the pattern.

Naturally, there are both pros and cons. First, implementing BLoC can involve a significant amount of template code which can be burdensome for smaller projects. Also, understanding and properly implementing streams and reactive programming principles might be difficult for developers unfamiliar with these paradigms. For simpler applications, the cost of setting up BLoC may not be justified, and less complex state management techniques may be preferable. To summarize, BLoC provides a powerful and systematic method for state management in Flutter applications. However, its advantages should be weighed against its complexity and boilerplate needs to decide if it is the best choice for your project.

Redux

Redux is a predictable state management library. Redux groups application states and actions in a modular format, making it easier for application developers to manage the state of the application. Redux is known for being simple and predictable. This makes it simple to debug and test. With Redux, you can structure your application so that the state is extracted in a centrally-located store. The data in this centralized store can be accessed by any widget that requires it,

without having to navigate a chain of other widgets in the tree [46].

Now we will mention about Redux Elements. The main components of Redux include Actions, Reducers, Store, Middleware, and Components. The interaction of these components ensures in a smooth state management process. The diagram provided in the Figure 2.8, these interactions and the flow of data within a Redux application.

Components

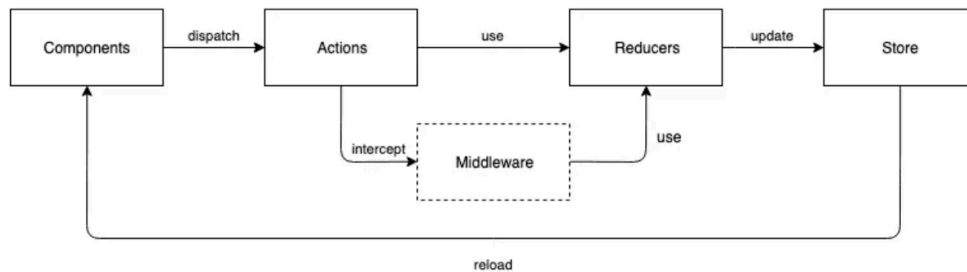


Figure 2.10: The Redux components [8]

Components are parts of the user interface in a Redux application. They dispatch actions in response to user interactions or other events. The Redux repository then processes these actions. This is the final step, in which we start our store and consume data to create our outcome.

Action

When a state is stored, widgets and sub-widgets are typically placed across the program to monitor the state and its current values. An action is the object that controls which event is done on the state. Following the event on this state, the widgets that track data in the state are re-generated, and the data they generate is updated to reflect the state's current values. Any event provided to the store that updates the app's state is considered an action. When a user interacts with a component, for example by clicking a button, an action is dispatched to the store.

Reducers

Reducers are pure functions that take two arguments, transforming the information received from the actions and setting the related values in the store. They indicate how the state of the app changes in response to actions sent to the store. Reducers process various actions and update the state accordingly. Since reducers are pure functions, they do not produce side effects, which makes state changes predictable.

Store

The store contains the whole state tree of the program. In a Redux application, it is the only reliable source of information about the state. The store is generated by passing the root reducer to the `createStore` function. The store includes methods for dispatching actions, subscribing to changes, and retrieving the current state.

Middleware

Middleware provides a third-party extension point between the dispatch of an action and when it reaches the reducer. If we want to construct independent levels for data handling, we can utilize middleware. It can be used for logging, crash reporting, performing asynchronous operations, etc. This allows us to intercept actions and do additional operations before they are sent to reducers. To summarize the data flow in Redux, the data flow starts with a user interaction, such as a button press, that triggers an action within a component. We send this action, represented as a flat JavaScript object, to the Redux repository using the `dispatch` method. Middleware can intercept the action before it reaches the reducer, performing responsibilities such as logging or dealing with asynchronous processes. Then the action reaches the reducer after going through the middleware. The reducer processes the action type and payload to determine how to update the state. After executing the operation, the reducer produces a new state that updates the store. Finally, the store notifies all subscribed components of the state change, causing them to rebuild. It also causes them to reflect the new state, ensuring that the user interface is always in sync with the application's current state.

Redux offers a straightforward and predictable state management strategy. In Redux, the centralized store makes it simple to manage and debug the application's state. Additionally, it

provides middleware to handle side effects and asynchronous activities. Redux, which also has disadvantages, contains a lot of boilerplate code. Also, it can be excessive for small and medium-sized applications.

Approach	Complexity	Boilerplate code	Code generation	Time travel	Scalability	Testability
setState()	easy	a lot	no	no	bad	bad
InheritedWidget	difficult	average	no	no	average	bad
Provider	easy	a little	no	no	good	average
GetX	easy	a little	no	no	good	average
BLoC	difficult	a lot	no	no	the best	good
MobX	easy	a little	yes	no	average	average
Redux	difficult	a lot	no	yes	good	good

Figure 2.11: The final comparison table of approaches [9]

The table in Figure 2.11 compares alternative state management systems in Flutter based on several critical factors, including complexity, boilerplate code, code generation, time travel, scalability, and testability. The comparison is crucial for understanding the advantages and disadvantages of each strategy and determining which state management approach is best suited for different sorts of Flutter projects. If we take a look at based on complexity, 'setState()', 'Provider', 'GetX', 'MobX' marked as easy. So that makes them accessible for beginners and suitable for basic applications. On the other hand 'InheritedWidget', 'BLoC' and 'Redux' marked as difficult so that means a higher learning curve and a greater requirement for experience. When it comes to boilerplate code, 'Provider', 'GetX', and 'MobX' require minimal boilerplate, which helps reduce the initial setup time and simplifies maintenance. So developers may easily get up and going with these user-friendly ways without having to deal with complex configuration. However, 'setState()', 'BLoC' and 'Redux' require a large amount of boilerplate code, which might delay development at first. In these approaches only 'MobX' supports code generation, which can help to speed up development and eliminate errors caused by human coding. If the time travel feature is important, we prefer Redux. BLoC, Redux, Provider and GetX are our options if we need good scalability. For easier testing and debugging, consider BLoC or Redux based on your preferred programming style (reactive or functional) [9].

2.3.4 TESTING AND DEBUGGING

Clean development and testing are critical issues when our project is large and several developers are working on it at the same time. This helps ensure that your software works properly before you release it [47]. In Flutter, we can use the built-in testing framework called Flutter Test to write unit, integration, and widget tests [48]. Evaluating the accuracy of your application is a testing process. It checks that your application's conduct corresponds to what you anticipate through different scenarios, encounters, and stimuli [49].

2.3.5 KINDS OF TESTING

- **Unit Testing:** Unit testing is the process of ensuring your application logic functions as expected before releasing it for general usage by creating additional testing code to assess quality, performance, or reliability [50]. Unit testing aims to test the smallest possible piece of code, which can include functions and classes among other things. Unit tests often execute in an isolated environment, where services are simulated using fictitious data to verify the output of the tested unit [10]. The provided image in Figure ??, illustrates the setup for unit testing with isolated environments, emphasizing the use of mock data to test backend services independently. These used to validate specific functions or methods independently [51]. In the Figure 2.12 unit testing example, the function definition involves the add function, which takes two integers as parameters and returns their sum. The test definition uses the test function to define a unit test case, with the description 'Unit Test' specifying what the test is intended to verify. The expect function tests whether the result of add(2, 3) equals 5. The equals(5) matcher compares the actual output to the expected value. This simple example of unit test ensures that the add method works correctly. If the function's implementation changes in the future, running this test will instantly disclose whether the change caused any issues.

```

1 // A simple function to add two numbers
2 int add(int a, int b) {
3     return a + b;
4 }
5
6 void main() {
7     // Define a test group
8     test('Unit Test', () {
9         // Verify that the add function correctly adds two numbers
10        expect(add(2, 3), equals(5)); // Checks if 2 + 3 equals 5
11    });
12 }

```

Figure 2.12: Unit Test Example

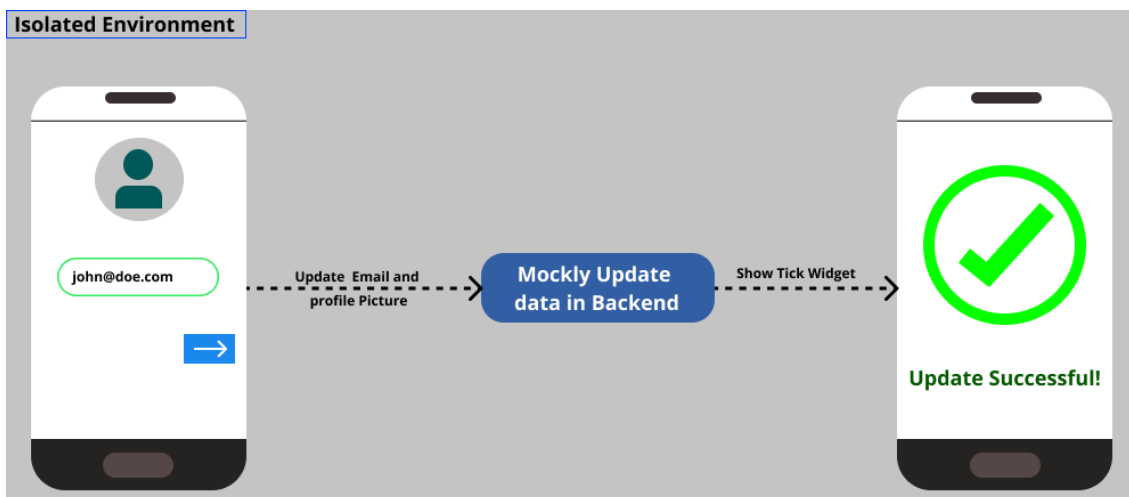


Figure 2.13: Widget Testing [10]

```

1  import 'package:flutter/material.dart';
2
3  void main() {
4    runApp(MyApp());
5  }
6
7  class MyApp extends StatelessWidget {
8    @override
9    Widget build(BuildContext context) {
10     return MaterialApp(
11       home: CounterPage(),
12     );
13   }
14 }
15
16 class CounterPage extends StatefulWidget {
17   @override
18   _CounterPageState createState() => _CounterPageState();
19 }
20
21 class _CounterPageState extends State<CounterPage> {
22   int _counter = 0;
23
24   void _incrementCounter() {
25     setState(() {
26       _counter++;
27     });
28   }
29

```

Figure 2.14: Counter app - Part 1

```

30  @override
31  Widget build(BuildContext context) {
32    return Scaffold(
33      appBar: AppBar(
34        title: Text('Counter App'),
35      ),
36      body: Center(
37        child: Column(
38          mainAxisAlignment: MainAxisAlignment.center,
39          children: <Widget>[
40            Text(
41              'You have pushed the button this many times:',
42            ),
43            Text(
44              '$_counter',
45              style: Theme.of(context).textTheme.headline4,
46            ),], ), ),
47      floatingActionButton: FloatingActionButton(
48        onPressed: _incrementCounter,
49        tooltip: 'Increment',
50        child: Icon(Icons.add),
51      ), ); }
52

```

Figure 2.15: Counter app - Part 2

- **Widget Testing:** Widget testing is used for checking the functionality of user interface components (widgets) [50]. The purpose of a widget test is to ensure that the widget's UI looks and behaves properly [51].

The Figure 2.13, depicts a widget testing scenario for changing a user's email and profile picture in an isolated environment. Here, the user starts with a screen that displays their current email and the profile picture. Also, this screen has an input area for the email and button to save the changes. When clicking the button, the app will update the email and picture data in the backend side. This mock update is handled by a test framework that simulates the backend response, allowing the frontend to be tested separately from the actual backend services. After the mock update, the app shows a success message and a tick widget to show that the update was successful. In order to ensure that the widget accurately represents the successful update, this visual feedback is important.


```

1 import 'package:flutter/material.dart';
2 import 'package:flutter_test/flutter_test.dart';
3 import 'package:my_app/main.dart';
4
5 void main() {
6   testWidgets('Counter increments smoke test', (WidgetTester tester) async {
7     // Build the app and trigger a frame.
8     await tester.pumpWidget(MyApp());
9
10    // Verify that the counter starts at 0.
11    expect(find.text('0'), findsOneWidget);
12    expect(find.text('1'), findsNothing);
13
14    // Tap the '+' icon and trigger a frame.
15    await tester.tap(find.byIcon(Icons.add));
16    await tester.pump();
17
18    // Verify that the counter has incremented.
19    expect(find.text('0'), findsNothing);
20    expect(find.text('1'), findsOneWidget);
21  });
22 }

```

Figure 2.16: Widget Test Example

Writing good test cases saves you from performing repetitive manual testing [52]. Below in the Figure 2.14 and Figure 2.15 , there is an example of a simple widget test for a counter app with a button that increments a displayed counter in Flutter. Now, let us create a widget test for the counter app in Figure 2.16 to ensure that the counter increases when the button is pressed. The `testWidgets` function is used to create a widget test case called 'Counter increments smoke test'. The

```
tester.pumpWidget(MyApp());
```

command builds the app and sets up an initial frame. Firstly, the test verifies that the counter starts at 0 using expect

```
(find.text('0'), findsOneWidget)
```

and ensures that the value 1 is not present with expect

```
(find.text('1'), findsNothing)
```

To simulate user interaction,

```
tester.tap(find.byIcon(Icons.add));
```

taps the '+' icon, followed by `tester.pump()`, which updates the user interface. Finally, the test confirms that the counter has been incremented by ensuring that the starting value 0 is no longer there and that the value 1 is now displayed using `expect`

```
(find.text('0'), findsNothing)
```

and `expect`

```
(find.text('1'), findsOneWidget).
```

This test shows that the counter functionality works as expected and that the UI responds correctly to user interactions.

- **Integration test:** Software testing methods such as integration testing combine separate software modules and test them collectively. Finding errors in the way integrated units interact with one another is the main goal of integration testing [53]. Integration tests help to ensure that your application is performing successfully [54]. This kind makes sure that nothing interferes with the smooth operation of the parts or causes faults [51].

2.3.6 CONTINUOUS INTEGRATION AND CONTINUOUS DEPLOYMENT (CI/CD)

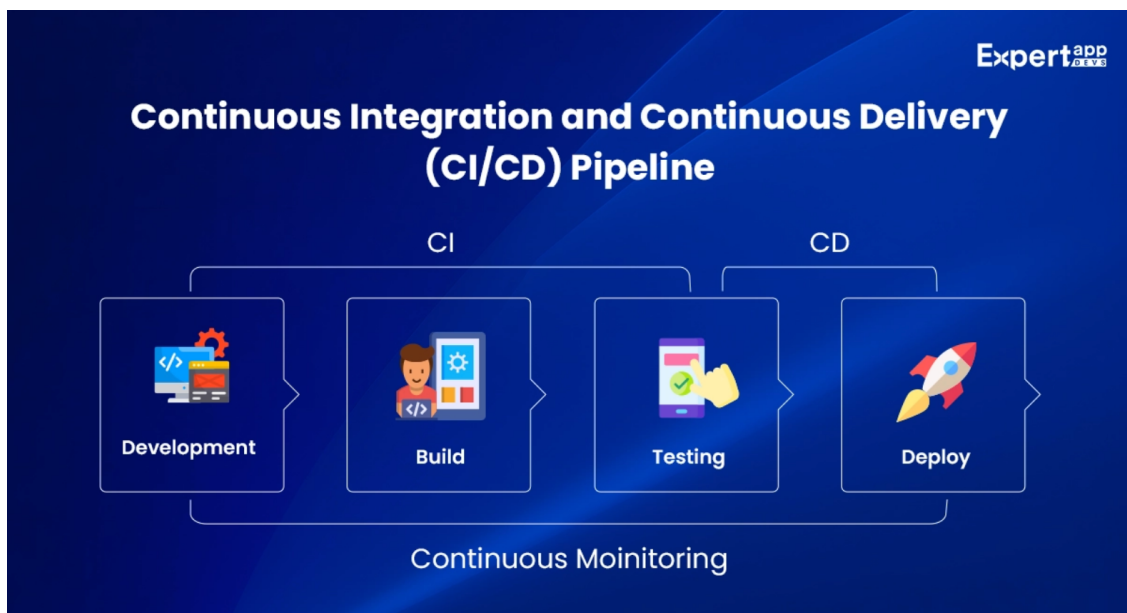


Figure 2.17: CI/CD Pipeline [11]

Continuous integration(CI) is a process which lets you find out the effect of every change you do in your project on other modules and thus help you in constant improvement in your code quality. After the development, different parts of your application may be affected or even rendered inoperable. By running all unit tests within the CI process, this issue can be detected and resolved immediately. You can utilize continuous integration by automating code change, and also make sure that the changes have been integrated into a common repository [11]. In contrast, continuous deployment automates the frequent release of software functionalities [55].

The goal of CI/CD is to increase software quality while also reducing the time it takes to deploy new features and issue fixes. There are several tools and platforms support CI/CD practices such as Bitrise, CircleCI, GitLab CI/CD, Codemagic. Implementing CI/CD in Flutter projects fosters a more efficient development workflow, ensuring that new features and bug fixes are delivered quickly and reliably, ultimately improving the overall application quality and user experience [11]. As depicted in Figure 2.17, The Continuous Integration and Continuous Deployment (CI/CD) pipeline comprises four key stages: Development, Build, Testing, and Deployment. During the Development stage that new code is written and committed, which afterwards needs to integrate bien with the existing codebase. The committed code is compiled during the Build stage to make sure it is error-free and ready to be turned into an executable program. The Testing phase is involves running automated tests to verify the functionality, performance, and reliability of the code. The phase is therefore vital in the process of detecting bugs and correcting them. In the end, during the Deployment stage, the tested code is moved to the production environment and made available to end users.

Previous works have also indicated challenges and barriers in CI/CD adoption implying that setting up a CI/CD pipeline is not an easy task. Furthermore, CI/CD can be done incorrectly, which might reduce its effectiveness and pose maintainability issues [56]. Implementing CI/CD (Continuous Integration/Continuous Deployment) for Flutter app development is crucial for organizations aiming for success in the highly competitive environment. It can fill in the gaps in the way growth has usually been done. You can speed up the time it takes to get an app to market and make it better with this combination.

2.3.7 PERFORMANCE OPTIMIZATION

It is quite important to optimize performance in mobile application development since it guarantees smooth usage and optimal utilization of resources. One interesting study that evaluated and compared several different cross-platform frameworks, including Flutter, React Native, and Ionic, provides insights useful to apply for performance overhead specific to these frameworks.

”React Native Performance Evaluation,” Rasmus Eskola’s thesis, says that React Native slows down significantly when compared to native code. This is particularly noticeable on older devices where common operations such as application launch and component rendering can have up to ten times longer latency than the native equivalent. On modern devices, the overhead is less noticeable but still present, making React Native more suitable for newer hardware [57].

A study by Wellington Oliveira et al. (2023) analyzed the resource usage overhead of mobile app development frameworks, comparing Flutter, React Native, and Ionic to native Java implementations. Comparing Flutter to React Native and Ionic, the results show that Flutter often imposes the least overhead in execution time and energy consumption. In particular, Flutter used less energy and ran faster in a number of benchmarks, which makes it a better choice for making mobile apps. However, in an app that continuously animates multiple images on the screen, without interaction, the React Native showed at least CPU and energy usage. These findings highlight the importance of analyzing expected application behavior before committing to a specific framework. [1]

As shown in Figure 2.1, the execution time, energy consumption, and memory usage of different frameworks provide a comprehensive view of their performance characteristics. These tips are useful for developers who want to make their apps run faster and use resources more efficiently.

On the other hand, Flutter usually has faster speed than native apps because it compiles to native ARM code. Skia, Flutter’s drawing engine, makes it easy for it to draw UI elements quickly and smoothly, which makes animations run more smoothly and loads faster [58].

The results of these studies show that Flutter is better in a number of important ways. First of all, It runs faster and has less latency compared to frameworks like React Native, due to its native processing and efficient rendering engine. Since Flutter usually uses less memory and energy, developing mobile apps with it is more environmentally friendly. Last but not least, Flutter offers a better optimal platform for creating beautiful, responsive mobile applications because to its strong performance and reduced resource consumption. These advantages shows how well-suited Flutter is as a top framework for creating cross-platform mobile applications. It also provides an appealing balance between effectiveness and performance.

2.3.8 CODE REUSABILITY AND MODULARIZATION

Code modularization and reusability are important concepts for minimize expenses, time and effort when developing a project. Modularity is the practice of separating your code into smaller, self-contained parts that execute certain jobs or operations. These self-contained parts are often referred modules, components, or classes, depending on the programming language and paradigm you use. Plan and identify the components of your software that require subdivision first. A software system should be divided into several distinct modules, with each class or method having a single purpose.

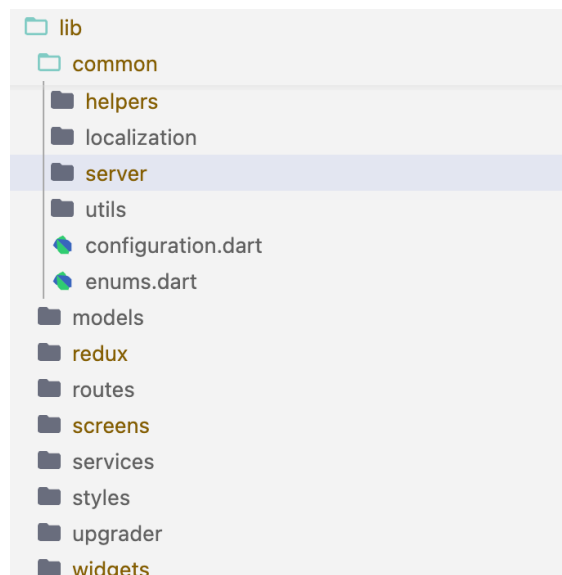


Figure 2.18: Example of Modularization

Figure 2.18 provides an example of a structure. Here I have divided the project into different files to find and manage my project easily related to their parts. This way, modularization will help you understand and manage your code better. Also, this structure enhances development efficiency. Procedures and functions can be called from anywhere in the program to do specific jobs. This makes code more reusable.

Let's talk about reusability too. Reusability refers to ability of your code in a way that can be used again in different parts of an application or projects without needing major changes. This can really help improve efficiency and stability. Also, reusable code is easier to maintain. Let's assume you find a bug, or you want to improve something in your reusable code or component, the changes will apply automatically wherever the code is used.

Figure 2.19 an example of reusable code. I have created a vertical button widget stored inside 'widgets' file. This button can be used across different pages or in the same page in a list component. By using this widget, we can promote code reusability and we know that any changes to the button such as changing text color, height of button, or text, will be only in this widget place. So this example structure shows good practices for modularization and reusability in our projects. So use the separation of concerns concept, which states that every module should have a distinct goal and a single job.

Dart ▾

```
import 'package:my_example/styles/example_colors.dart';
import 'package:flutter/material.dart';

class VerticalFullRaisedButton extends StatelessWidget {
  final String text;
  final double containerHeight;
  final Color color;
  final Color textColor;
  final Color disabledColor;
  final Color disabledTextColor;
  final Function()? onPressed;
  final bool isDisabled;
  VerticalFullRaisedButton(this.text,
    {this.color = ExampleColors.blue50Default,
    this.textColor = ExampleColors.white,
    this.disabledColor = ExampleColors.gray30Disabled,
    this.disabledTextColor = ExampleColors.gray50,
    this.onPressed,
    this.isDisabled = false,
    this.containerHeight = 48});
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      child: Container(
        height: containerHeight,
        child: Center(
          child: Text(
            this.text,
            style: TextStyle(
              fontWeight: FontWeight.w700,
              color: isDisabled ? this.disabledTextColor : this.textColor,
              fontSize: 16.0,
              fontStyle: FontStyle.normal),
          ),), ),
      style: ButtonStyle(
        surfaceTintColor: MaterialStateProperty.all(
          isDisabled ? this.disabledColor : this.color),
        backgroundColor: MaterialStateProperty.all(
          isDisabled ? this.disabledColor : this.color),
        shape: MaterialStateProperty.all(
          RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(4.0),
          ),
        )),
      onPressed: () {
        if (!this.isDisabled && this.onPressed != null) {
          this.onPressed!();
        } }, ); }}

```

Figure 2.19: Example of Reusability

2.3.9 VERSION CONTROL SYSTEMS

In this section, we will see the essential tool in modern software development such as version control systems. Version control system helps to developers to maintain changes, follow the project history, and collaborate efficiently with team members. In addition, developers can revert to previous version of the code if they need. That's way version control systems are important part of development process. They help keep code safe and make the process of making software faster. When you want to choose a version control system there are key factors to consider such as the maximum number of users can access per account, the capacity of the storage, and the features of the tool.

Version Control System	CVCS	DVCS
Repository	There is only one central repository which is the server.	Every user has a complete repository which is called local repository on their local computer.
Repository Access	Every user who needs to access the repository must be connected via network.	DVCS allows every user to work completely offline. But user need a network to share their repositories with other users.
Example of VCS Tools	Subversion, Perforce Revision Control System	Git, Mercurial, Bazaar, BitKeeper
Software Characteristics that suitable	<ul style="list-style-type: none"> i. Projects that allow only several users to contribute to the software development. ii. Team located in a single site. 	<ul style="list-style-type: none"> i. DVCS is suitable for a single or more developers because the project repository is distributed to all the developers and this ability offer a great improvement for the projects. ii. It also can be applied for a small or big software projects because it makes less difficult for normal users to contribute to the development. iii. Team located in multiple site or different countries and different timezones.

Figure 2.20: Comparison between CVCS and DVCS [12]

According to the study by Zolkifli et al., there are two primary approaches to version control systems: centralized version control systems (CVCS) and distributed version control systems (DVCS). The figure 2.20 summarizes the key differences between CVCS and DVCS. CVCS, such as Concurrent Versions System (CVS) and Subversion (SVN), use a single central repository that all users must connect to for accessing and committing changes.

Another version control system, DVCS, such as Git, Mercurial, Bazaar, and BitKeeper, provides each user with a complete local copy of the repository, enabling offline work and better handling of branching and merging. While the CVCS model is suitable for projects where only a few users contribute and the team is located at a single site, DVCS is suitable for both small and large projects and allows contributions from developers located across multiple sites or different time zones. Figure 2.21 illustrates the benefits of using a version control system.

Version control systems support collaborative framework that make it easier for more efficient teamwork between developers. So multiple team members can work on the same project without intervention to each other. When a developer make a mistake or want to recover any error, can revert the previous version of the code. I think the best thing is the branching and merging benefit. Because it allows the creation of different branches for storing different features, improvements, or bug fixes, which can later be merged into master codebase.

I've learned from the bugs I've made and the teammates I've worked with that small and frequent commits can be more manageable and reversible. In addition, writing clear commit messages is important to explain the goal of the changes to help other team members. Make code reviews frequently to maintain the code quality of the executed project and facilitate sharing knowledge and experience among team members. And this is something that should not be overlooked, merge changes regularly to avoid conflicts and keep the master codebase up to date.

BENEFITS OF VERSION CONTROL

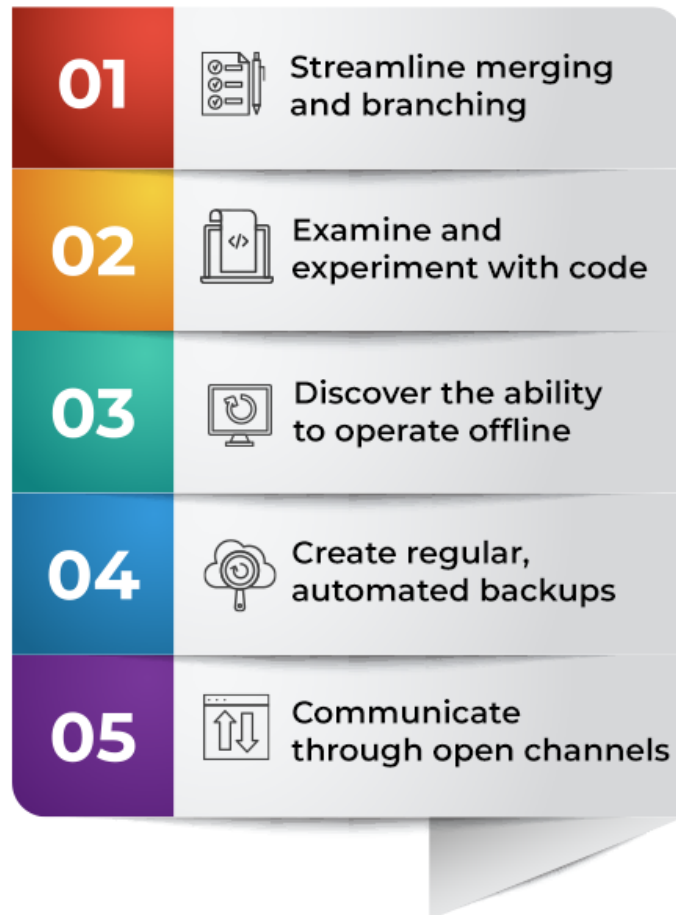
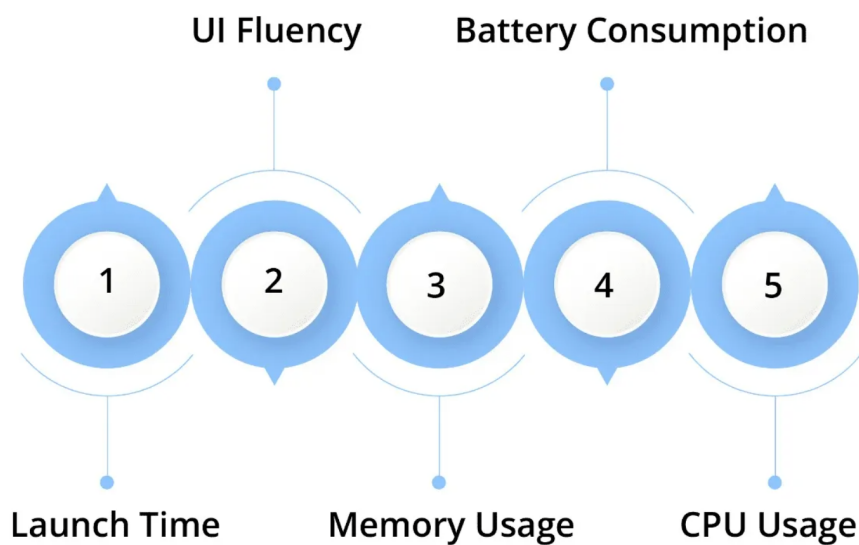


Figure 2.21: Benefits of Version Control [13]

2.3.10 PERFORMANCE METRICS COMPARISON: FLUTTER VS. REACT NATIVE



TECHAHEAD

Figure 2.22: Performance Metrics [14]

Beyond the advantages of Flutter for developers, it is also a good solution from a business field. There are benefits for customers and businesses with beautiful UI, more quickly getting-to-market, and cost-effectiveness.

Performance metrics of Flutter and React Native include factors such as memory usage, launch time, UI rendering speed, CPU usage and battery consumption, as shown in figure 2.22. Flutter's architecture gives it an advantage in terms of performance. In contrast to React Native, which necessitates a JavaScript communication bridge to interact with native components, Flutter does not require this bridge. Rather, it employs the robust Skia rendering engine. Performance may be lowered by the bridge in React Native [14].

The Google Team said this about Flutter’s speed: ”Flutter is fast. It’s powered by the same hardware-accelerated Skia 2D graphics engine that underpins Chrome and Android. We architected Flutter to be able to support glitch-free, jank-free graphics at the native speed of your device. Flutter code is powered by the world-class Dart platform, which enables compilation to native 32-bit and 64-bit ARM code for iOS and Android.”

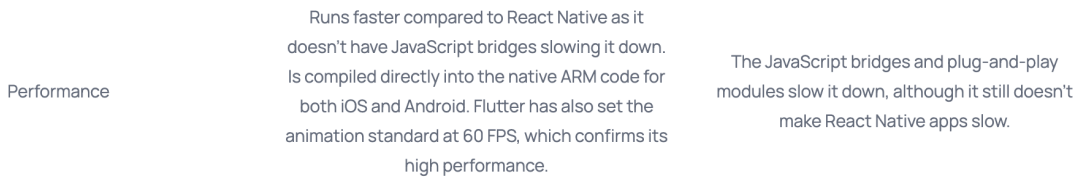


Figure 2.23: Performance comparison Between Flutter and React Native [15]

Performance Comparison of Flutter & React Native		
PERFORMANCE METRICS	FLUTTER	REACT NATIVE
App Startup Time	Faster	Slower
UI Rendering Speed	Faster	Slower
App Memory Usage	Lower	Higher
Overall Responsiveness	Better	Lesser

Figure 2.24: [16]

The UI fluency metric evaluates the app's ability to respond to user input, such as page transitions, screen scrolls, and button presses. The fact that the interface is instantaneous and fluid supports the user experience. While React Native provides good performance, it may not be as performant as Flutter in certain scenarios that involve heavy animations or complex UI interactions [16].

From the blogpost of SPEC INDIA shown in Figure 2.24, there are comparisons the performance of Flutter and React Native based on specific performance metrics. So based on this comparison, I can say that Flutter outperforms React Navite in startup time, UI rendering speed, and overall responsiveness due to its native code compilation and architectural advantages.

2.4 CASE STUDIES ON FLUTTER INTEGRATION

In this section, I will present several case studies that show how the Flutter framework is integrated into practical applications. These case studies show the useful uses, benefits, and difficulties that develop along the integration process. Case studies let the developer see how Flutter is positioned for real-world use and give an idea of how the framework handles common problems. Flutter offers a wide range of advantages that, based on my experience, make it an effective tool for any organization.

Flutter has great customization possibilities and may be adapted to meet specific business requirements. There are many things you can achieve with Flutter. The most important thing is to be able to use code on multiple platforms, such as iOS, Android, desktop, web, and embedded. "Flutter's unique features, value, and increasing adoption make it a solid business choice today. This is why top executives worldwide are keen on harnessing the power of Flutter for their digital projects [59]". Several firms shown in Figure 2.25 and in different sectors such as e-commerce, finance, healthcare, logistics, and education have taken advantage of Flutter to optimize their application development procedures and provide their users with excellent experiences.

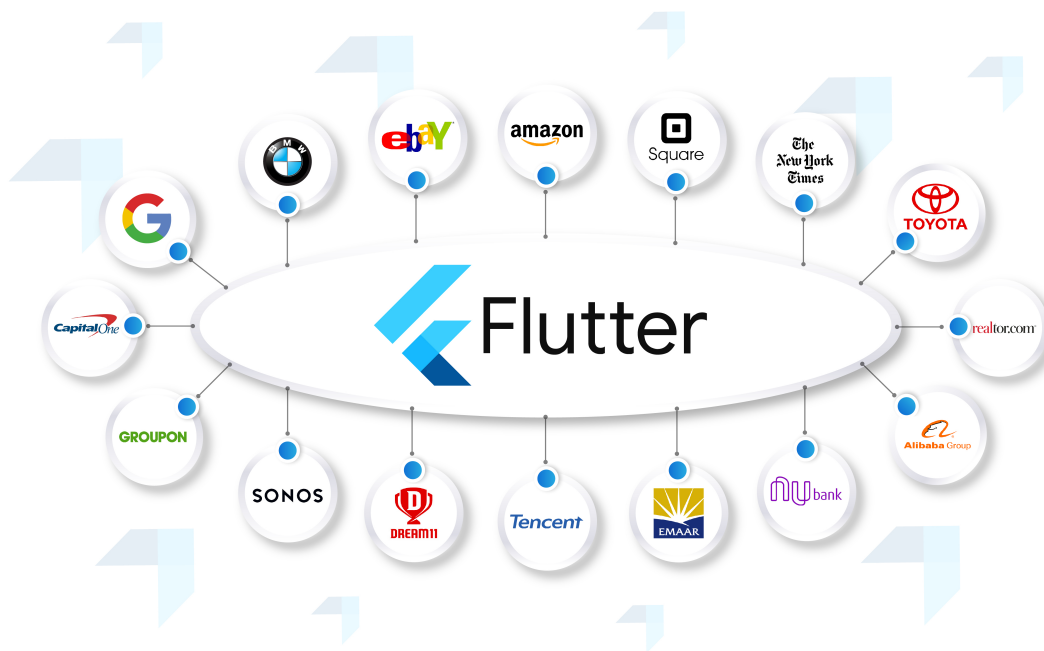


Figure 2.25: Companies That Use Flutter [17]

2.4.1 CASE STUDY 1: ALIBABA’S XIANYU APP

Alibaba, one of the world’s largest e-commerce companies, used Flutter to create the Xianyu app. Xianyu is a popular second-hand marketplace. The goal was to create an area where buyers and sellers of secondhand products without the need for an intermediary. Alibaba’s Xianyu app (iOS, Android) was another key early adopter of Flutter. This app demonstrated Flutter’s scalability and performance capabilities with millions of users, [60]. They had limited time to fulfill their goals. They decided to go with Flutter. Because of the maintain single codebase they had enough time to build new features. UI widgets allowed the developers to create a smooth UI. So the platform helped them achieve this goals [61].



Flutter significantly reduced the time we need to develop for new feature from 1 month down to 2 weeks.

Bruce Chen
Senior Development Engineer, Alibaba

Figure 2.26: Flutter's benefit of Alibaba [18]

Alibaba needed a solution that provided fast development and consistent performance on both iOS and Android. They used Flutter as an e-commerce software development, resulting in a highly interactive user interface with high performance and the ability to reuse code across platforms, vital for retaining and engaging customers in the e-commerce space [59].

This is the most popular Flutter case study because it demonstrates how a brand improved user experiences through interface development. It was important for the company to extend intuitive solutions so that the users could move around easily [61]. So, using Flutter allowed to significantly reduce the development process and enable faster iterations. Fifty million users use the current app, and Flutter has paved the path for growth [61]. According to Bruce Chen, Senior Development Engineer at Alibaba, "Flutter significantly reduced the time we need to develop for new feature from 1 month down to 2 weeks" (see Figure 2.26).

2.4.2 CASE STUDY 2: GOOGLE PAY

Google Pay is created by Google for mobile payment service. It is used to facilitate contactless in-app, online, and in-person transactions using Android phones, tablets, and watches. Google Pay has a user base of 100 million individuals across numerous countries. Flutter's own site reports, "But to do that, they relied on 1.7 million lines of code between their Android and iOS apps — an amount that didn't feel sustainable as Google Pay continued to expand to new countries, each of which would require its own unique features" [18].

Since iOS users are an ever-expanding user base and the majority of its own users are Android users, it was important that these two environments work in both iOS and Android environments with code written in one go. "Above all, migrating to Flutter would enable fast, resource-efficient scaling of Google Pay around the world. Whereas building out features on both An-

droid and iOS required double the effort, Flutter would only require about 1.2 times as much work. So they decided to take the plunge [18].”

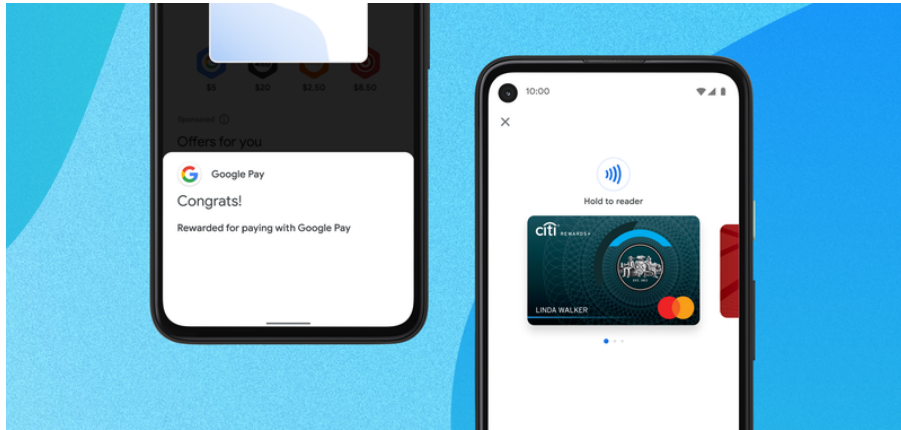


Figure 2.27: Google Pay [18]

“Everyone loved Flutter — you could see the thrill on people’s faces as they talked about how fast it was to build a user interface.”

— David Ko, Engineering Director, Google Pay

As a result, the Google Pay application was able to be developed more efficiently, more compactly, and more readily updated by utilizing Flutter. Since Flutter is much easier to manage, it is a great advantage that there is less code written and developers save time.

3

Methodology

In this section, I will explain my preferences for the methods, codes, and strategies employed in the Flutter mobile applications I have worked on, highlighting the reasons behind my choices based on my gained experience. As I mentioned in the previous chapter, Flutter, developed by Google, allows developers to build natively compiled applications for mobile, web and desktop from a single code base. One of the reasons, Flutter allows my code changes to be reflected instantly thanks to its "hot reload" feature. This made my development process faster and more efficient. Additionally, Flutter gives the advantage of being able to use a single codebase for iOS and Android. Flutter's performance is remarkable for its cross-platform speed. React Native's performance may be slightly lower than Flutter due to its compilation process and bridging layers.

React Native, on the other hand, allows web developers to use familiar JavaScript. This can increase code reusability. However, there may be issues with cross-platform compatibility and some components may differ.

3.1 PERFORMANCE AND SPEED

Performance metrics are important to analyze and ensure an application's smooth running and positive user experience, in mobile development. In order to evaluate this experience in an objective way, developers utilize measurable performance indicators.

3.2 PERFORMANCE OPTIMIZATION WITH METHOD CHANNELS

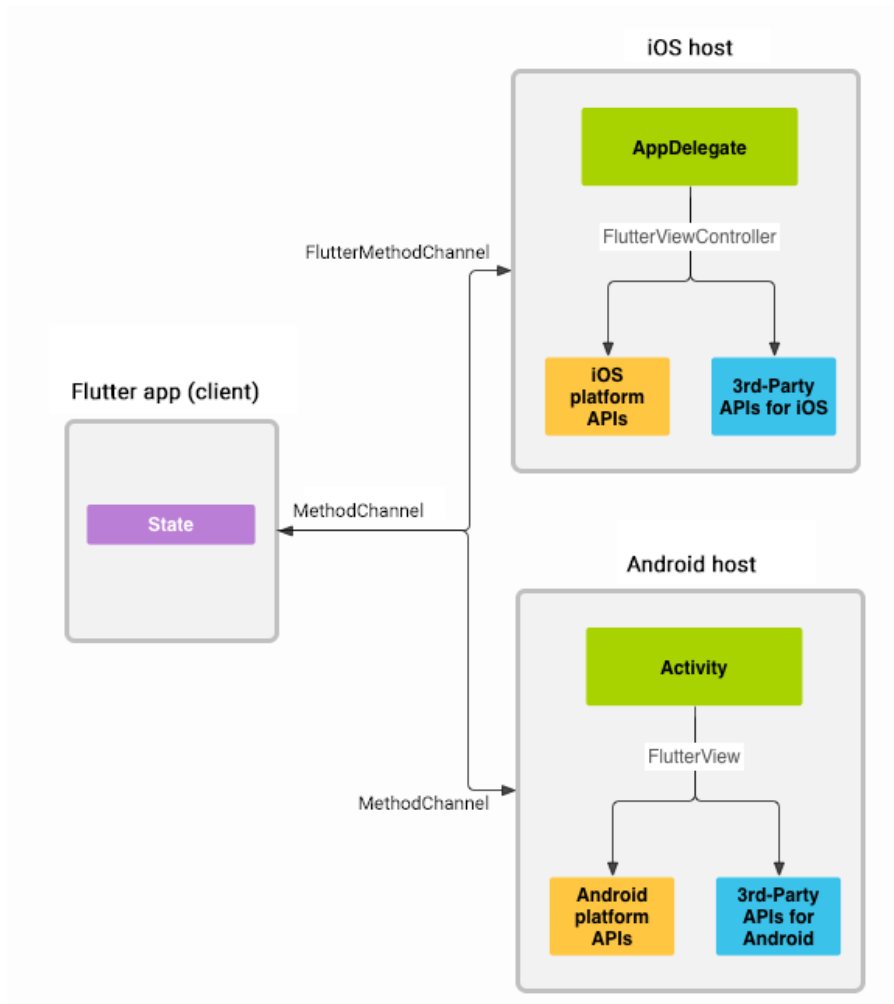
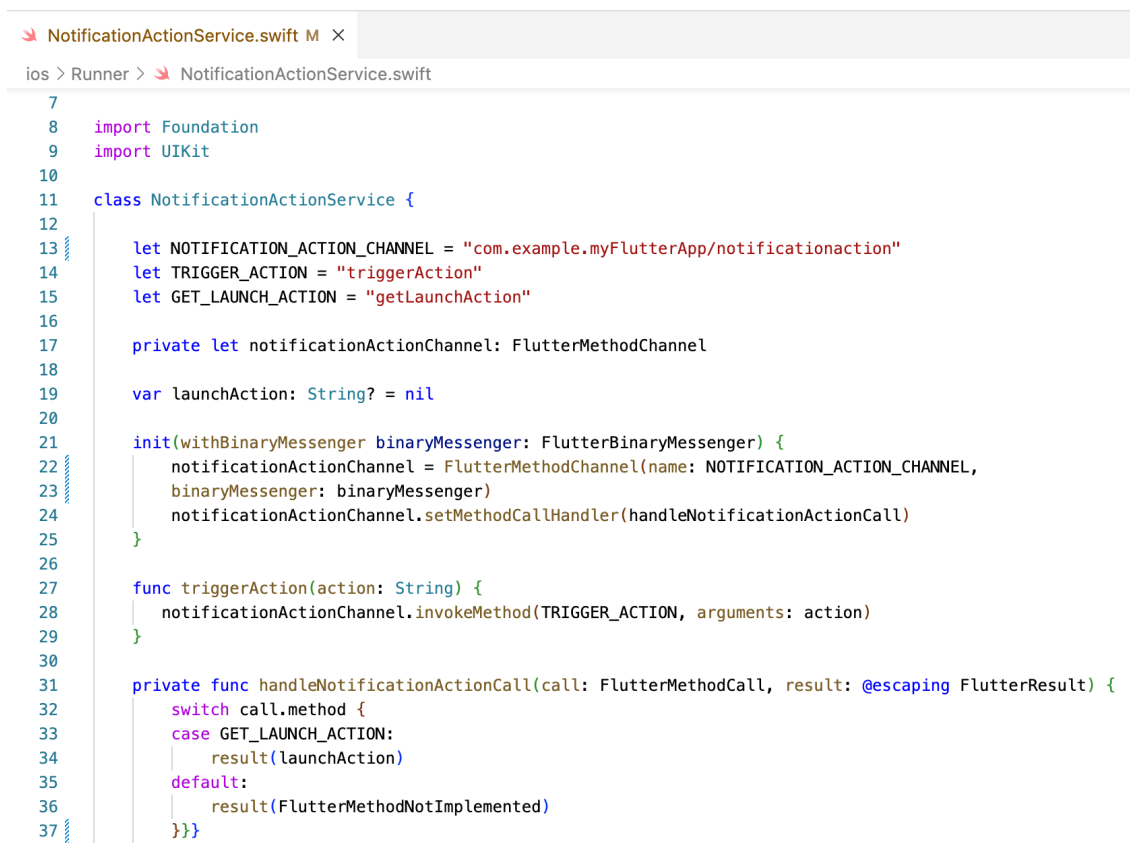


Figure 3.1: Architectural overview: platform channels [19].

In view of the development of the project I am working on, I needed to ask the question "How can we speed up and simplify the way Flutter talks to its main platform?" When I review Flutter community pages and documentation, the connection between Flutter and the platform is usually made via the platform's channels. You can check the section "Writing custom platform-specific code" from the Flutter documentation website. From this page, we can experience how to write custom platform-specific code. Flutter uses a flexible system that allows you to

call platform-specific APIs in a language that works directly with those APIs: Kotlin or Java on Android, Swift or Objective-C on iOS, C++ on Windows, Objective-C on macOS, C on Linux [19].

Figure 3.1 shows the architecture of method channels in Flutter, highlighting how the Flutter app communicates with platform-specific APIs on both iOS and Android. According to this architecture, the Flutter app uses MethodChannel to send and receive messages. On iOS the 'AppDelegate' via 'FlutterViewController' and on Android 'Activity' via 'FlutterView' can manage the messages. So this helps us to use native functionalities from iOS and Android platforms.



```
NotificationActionService.swift M X
ios > Runner > NotificationActionService.swift
7
8 import Foundation
9 import UIKit
10
11 class NotificationActionService {
12
13     let NOTIFICATION_ACTION_CHANNEL = "com.example.myFlutterApp/notificationaction"
14     let TRIGGER_ACTION = "triggerAction"
15     let GET_LAUNCH_ACTION = "getLaunchAction"
16
17     private let notificationActionChannel: FlutterMethodChannel
18
19     var launchAction: String? = nil
20
21     init(withBinaryMessenger binaryMessenger: FlutterBinaryMessenger) {
22         notificationActionChannel = FlutterMethodChannel(name: NOTIFICATION_ACTION_CHANNEL,
23             binaryMessenger: binaryMessenger)
24         notificationActionChannel.setMethodCallHandler(handleNotificationActionCall)
25     }
26
27     func triggerAction(action: String) {
28         notificationActionChannel.invokeMethod(TRIGGER_ACTION, arguments: action)
29     }
30
31     private func handleNotificationActionCall(call: FlutterMethodCall, result: @escaping FlutterResult) {
32         switch call.method {
33             case GET_LAUNCH_ACTION:
34                 result(launchAction)
35             default:
36                 result(FlutterMethodNotImplemented)
37         }
38     }
39 }
```

Figure 3.2: notificationActionService.swift

```

NotificationActionService.kt M x
android > app > src > main > kotlin > com > example > helloworld > services > NotificationActionService.kt
1 package com.example.myFlutterAppservices
2 import io.flutter.embedding.engine.FlutterEngine
3 import io.flutter.plugin.common.MethodCall
4 import io.flutter.plugin.common.MethodChannel
5
6 class NotificationActionService {
7     companion object {
8         const val NOTIFICATION_ACTION_CHANNEL = "com.example.myFlutterApp/notificationaction"
9         const val TRIGGER_ACTION = "triggerAction"
10        const val GET_LAUNCH_ACTION = "getLaunchAction"
11    }
12
13    private var notificationActionChannel : MethodChannel
14    var launchAction : String? = null
15
16    constructor(flutterEngine: FlutterEngine) {
17        notificationActionChannel = MethodChannel(flutterEngine.dartExecutor.binaryMessenger,
18            NotificationActionService.NOTIFICATION_ACTION_CHANNEL)
19        notificationActionChannel.setMethodCallHandler { call, result -> handleNotificationActionCall(call, result) }
20    }
21
22    fun triggerAction(action: String?) {
23        notificationActionChannel.invokeMethod(NotificationActionService.TRIGGER_ACTION, action)
24    }
25
26    private fun handleNotificationActionCall(call: MethodCall, result: MethodChannel.Result) {
27        when (call.method) {
28            NotificationActionService.GET_LAUNCH_ACTION -> {
29                result.success(launchAction)
30            }
31            else -> {
32                result.notImplemented()
33            } } }

```

Figure 3.3: notificationActionService.kt

The following Figures 3.2 and 3.3 are example of implementing Method Channels for Notifications in iOS and Android. The following swift and kotlin codes set up a method channel in an Android and iOS app to trigger and handle notification actions from the Flutter app. In this examples, the NotificationActionService class set up a method channel named com.example.myFlutterApp/notificationaction. I initialized the channel with 'Flutter Engine' to enable communication between Flutter and Android and for iOS 'FlutterBinaryMessenger'. When the triggerAction function is called, it calls a method on the Flutter side with the action that was given. The handleNotificationActionCall function is responsible for receiving method calls from the Flutter app. The getLaunchAction method is handled by getting the stored launchAction and returning the MethodChannel. So this showcase shows how to handle notifications efficiently.

Why do I prefer to use Flutter? When we compare React Native and Flutter to create method channels, both frameworks offer robust solutions, but there are key differences. React Native developers has to be knowledgeable in JavaScript for the React Native side and Java or

Objective-C/Swift for the native side but Flutter uses Dart for both the Flutter app and method channel implementation. That helps to simplify the development process. Using the JavaScript bridge in React Native introduces additional latency and performance overhead as it needs to be transferred asynchronously. In contrast, Flutter uses direct platform channels, resulting in more efficient communication with less overhead. Another difference is that Flutter can be preferred over React Native as it provides a more unified and consistent approach with its own plugin system and use of platform channels. When we consider these comparisons, we can effectively see Flutter's strengths in addressing method channels. For projects that prioritize performance and a streamlined development process, Flutter's method channels may be more advantageous.

3.3 CHAT SCREEN PERFORMANCE METRICS

In this section, I analyzed the performance metrics of the chat screen in my flutter application. The key metrics evaluated include Elapsed Time, Build Time, and Raster Time over a series of frames.

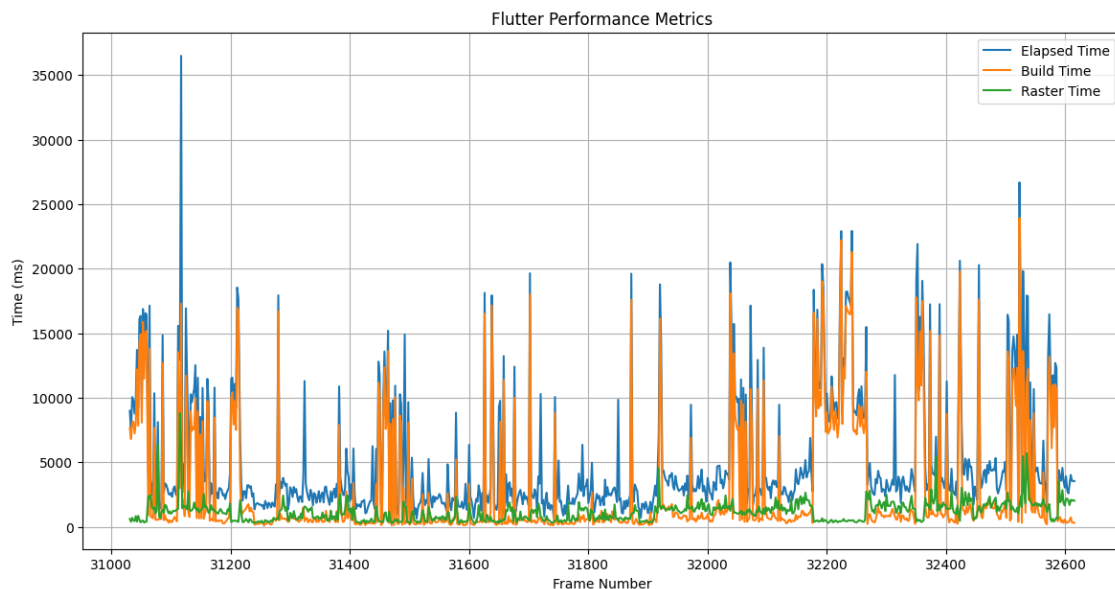


Figure 3.4: Chat Screen Performance Metrics

If we interpret the analysis of this measurement according to the graph shown in Figure 3.4, the constantly reduced raster time indicates that the GPU rendering process is efficient

and stable, resulting in smooth and responsive visual rendering. Most build times are rather short, with sporadic spikes showing otherwise effective management of UI modifications in most cases. When we analyze the build time metric, we can say that it is a manageable build time. On the other metric which is the elapsed time, it provides a low average elapsed time. The average elapsed time is low despite the high spikes, which improves user experience by reducing latency and guaranteeing seamless interactions. The chat screen shows strong performance in terms of raster time and generally stable build times. This study shows the useful parts of how well the chat screen works.

3.4 INTEGRATING FLUTTER PROJECTS AS A PACKAGE IN IOS NATIVE APPLICATIONS

Flutter projects or components can be inserted into any iOS application as embedded frameworks. As described in Flutter's docs, there are three methods to do that. Although the Flutter documentation recommends using CocoaPods, due to the advantages of using SPM (Swift Package Manager), I prefer to follow the SPM (Swift Package Manager) package method because CocoaPods may take longer to resolve dependencies, especially when there are a lot of dependencies or the project is big. Another disadvantage of CocoaPods, Installing and maintaining an extra tool is necessary for it, which might increase complexity and cause potential conflicts. So using SPM (Swift Package Manager) for integrating the Flutter module into an iOS native app offers a faster and native experience compared to CocoaPods.

3.4.1 MODULARIZATION AND INTEGRATION EASE

SPM (Swift Package Manager) supports modularization and versioning, which simplifies dependency management and updates. You can use the same code in more than one project if you turn your Flutter project into an SPM package. Using modules makes it easy to maintain and update the code since changes to one package are made to all the places that use that package. In addition, makes it clear that the Flutter module is separate from the rest of the iOS app, which can make it easier to handle the codebase. SPM (Swift Package Manager) is built into Xcode and provides a seamless way to add, update and manage dependencies. This helps to avoid the requirement for manual configuration and guarantees the consistent management of all dependencies. Also with SPM (Swift Package Manager), you can avoid the complexity of manually configuring your project to include Flutter modules. SPM performs the installa-

tion automatically, reducing potential errors and setup time. There are several platforms for distributing a Swift Package Manager(SPM) package such as Azure or GitHub Actions. These offer benefits such as automated CI/CD (Continuous Integration and Continuous Deployment), version control, collaboration. These platforms provide robust tools to improve the development process and collaboration.

Swift Package Manager (SPM) supports versioned packages, making it easy to manage and update new versions of your Flutter module. It helps ensure compatibility and stability by specifying full releases or release ranges. SPM can increase build performance by rebuilding only the elements of the project that have changed. This leads to quicker building periods, particularly in large projects. Foremost, SwiftPM is a native Apple tool integrated into Swift. Other advantages include a quick and simple configuration, easier control over packages and their sub-dependencies, and a GUI built into Xcode for managing package configurations [62].

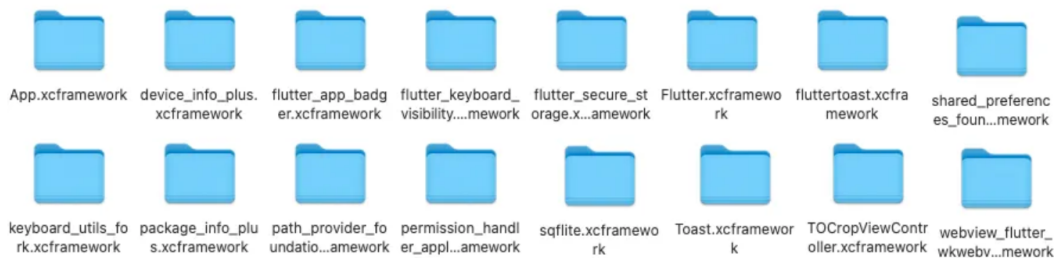


Figure 3.5: Dependencies of the Flutter Project

Teams are able to collaborate more effectively when they utilize Swift Package Manager (SPM). Team members have the ability to autonomously work on various components of the project and seamlessly integrate their modifications. On the other hand, it makes it easier to integrate open-source packages into your project. With SPM, all dependencies are declared in a single Package.swift file, making it easier to manage and understand the dependencies of your project. With Swift Package Manager (SPM), all dependencies are declared in a single Package.swift file, making it easier to manage and understand the dependencies of your project. As shown in Figure 3.5, let's assume your project have these dependencies so you can add all these depen-

dependencies into package.swift as shown in Figure 3.6. Only you need to configure it based on your dependencies. After that, if the build is successful that means the .xcframework files are ready for use.

```

14  targets: [
15    .target(
16      name: "MyFlutterPackage",
17      dependencies: [
18        "App", "Flutter", "path_provider_foundation", "uni_links", "permission_handler_apple",
19        "shared_preferences_foundation", "flutter_keyboard_visibility", "flutter_secure_storage",
20        "fluttershare", "device_info_plus", "image_picker_ios", "keyboard_utils_fork",
21        "package_info_plus", "sqlite", "Toast", "TOCropViewController", "url_launcher_ios",
22        "webview_flutter_wkwebview", "image_cropper", "flutter_app_badger"]],
23    .binaryTarget(name: "App", path: "./MyFlutterFrameworks/App.xcframework"),
24    .binaryTarget(name: "Flutter", path: "./MyFlutterFrameworks/Flutter.xcframework"),
25    .binaryTarget(name: "path_provider_foundation", path:
26      "./MyFlutterFrameworks/path_provider_foundation.xcframework"),
27    .binaryTarget(name: "uni_links", path: "./MyFlutterFrameworks/uni_links.xcframework"),
28    .binaryTarget(name: "permission_handler_apple", path:
29      "./MyFlutterFrameworks/permission_handler_apple.xcframework"),
30    .binaryTarget(name: "shared_preferences_foundation", path:
31      "./MyFlutterFrameworks/shared_preferences_foundation.xcframework"),
32    .binaryTarget(name: "flutter_keyboard_visibility", path:
33      "./MyFlutterFrameworks/flutter_keyboard_visibility.xcframework"),
34    .binaryTarget(name: "flutter_secure_storage", path:
35      "./MyFlutterFrameworks/flutter_secure_storage.xcframework"),
36    .binaryTarget(name: "fluttershare", path: "./MyFlutterFrameworks/fluttershare.xcframework"),
37    .binaryTarget(name: "device_info_plus", path: "./MyFlutterFrameworks/device_info_plus.xcframework"),
38    .binaryTarget(name: "image_picker_ios", path: "./MyFlutterFrameworks/image_picker_ios.xcframework"),
39    .binaryTarget(name: "keyboard_utils_fork", path: "./MyFlutterFrameworks/keyboard_utils_fork.xcframework"),
40    .binaryTarget(name: "package_info_plus", path: "./MyFlutterFrameworks/package_info_plus.xcframework"),
41    .binaryTarget(name: "sqlite", path: "./MyFlutterFrameworks/sqlite.xcframework"),
42    .binaryTarget(name: "Toast", path: "./MyFlutterFrameworks/Toast.xcframework"),
43    .binaryTarget(name: "TOCropViewController", path: "./MyFlutterFrameworks/TOCropViewController.xcframework"),
44    .binaryTarget(name: "url_launcher_ios", path: "./MyFlutterFrameworks/url_launcher_ios.xcframework"),
45    .binaryTarget(name: "webview_flutter_wkwebview", path: "./MyFlutterFrameworks/webview_flutter_wkwebview
46      .xcframework"),
47    .binaryTarget(name: "image_cropper", path: "./MyFlutterFrameworks/image_cropper.xcframework"),
48    .binaryTarget(name: "flutter_app_badger", path: "./MyFlutterFrameworks/flutter_app_badger.xcframework"),
49  ]

```

Figure 3.6: package.swift

3.4.2 REACT NATIVE OR FLUTTER

Deciding between converting Flutter or React Native for integration into a native iOS app depends on several factors such as performance, codebase, SPM integration. Flutter might be the better choice if performance is very important, and you need graphics that run quickly and smoothly. Also, if you prefer using a single language (Dart) for both UI and logic you can use Flutter. React native integration has benefits but as I said before it depends on what you expect in your project. When we consider the disadvantages of React native, due to performance and integration complexity, Flutter might be better choice.

3.5 OVERVIEW OF RESULTS

The findings of this study, many aspects of the Flutter framework in mobile application development have been highlighted. Comparative analysis with frameworks such as React Native also with Ionic provided important insights.

3.5.1 PERFORMANCE METRICS

According to the performance evaluation, Flutter performs smoother and with reduced latency than React Native, especially on older devices. As shown by several benchmarks, Flutter's native compilation and effective rendering engine, Skia, help to explain its outstanding performance. For instance, Flutter consistently demonstrated faster execution times and lower energy consumption in CPU-intensive applications, compared to React Native and Ionic.

3.5.2 RESOURCE EFFICIENCY

The resource efficiency of Flutter was also emphasized by the analysis. In multiple benchmarks, Flutter consumed less energy and memory than its counterparts. This is especially important for uses where battery life and making good use of resources are important. The reduced overhead in execution time and energy consumption positions Flutter as a more sustainable option for mobile app development.

3.5.3 DISCUSSION ON IMPLICATIONS

The companies thinking about cross-platform systems as well as developers, these results have significant relevance. Flutter is a desirable option for projects needing a quick time to market and excellent user experiences because to its capacity to produce high performance and resource-efficient applications. The unified codebase approach not only streamlines development but also reduces maintenance efforts

Moreover, the utilization of method channels in Flutter, as explored in the case studies, shows the framework's capacity to effectively manage platform-specific capabilities. Compared to React Native, which introduces additional latency through its JavaScript bridge, Flutter's direct platform channels ensure more efficient communication and integration with native APIs.

Generally, the study confirms that Flutter provides a robust, efficient and flexible platform for developing cross-platform mobile applications. Major organizations like Alibaba and Google Pay have adopted it, which shows that it can fit different development objectives while still being very effective and satisfying for users.



Figure 3.7: Swot Analysis

The swot analysis provided in Figure 3.7 provides a comprehensive overview of the Flutter framework based on the research findings. This analysis visually encapsulates the strengths, weaknesses, opportunities, and threats associated with the Flutter framework, providing a clear summary of the research findings.

4

Conclusion

This study aims to review the integration of the Flutter framework in real-life applications from a technical and development practice point of view. Such a detailed comparison with certain other leading cross-platform frameworks like React Native and Ionic explained several major key ideas and advantages of Flutter.

Flutter's architecture makes it easy to build high-performance applications thanks to the Dart programming language and Skia graphics engine. The benchmarks conducted in this study continuously showed that Flutter frequently displays lower latency, faster execution speeds, and reduced energy consumption in comparison to React Native and Ionic. The performance advantage is important for applications that prioritize efficiency and responsiveness.

Compared to React Native's JavaScript bridge, Flutter's method channels make it easier for Dart and native code to communicate with each other and this capability shows Flutter's ability to handle platform-specific functionalities with low overhead, improving its appeal to mobile developers.

This paper describes the advantages and practical benefits of using the Flutter interface, the importance of its cross-platform use and its overall positive contribution to mobile development. By showcasing Flutter's advanced functionality, economical use of resources and simplified development procedure, this thesis promotes its use in various application contexts.

This study shows how Flutter is important for the mobile development community and for developers and companies looking for good and effective solutions in their mobile applications. Flutter provides high-performance applications across multiple platforms, which makes it preferable in modern application development.

In conclusion, there are many benefits of using Flutter in real-world applications, such as better speed, better resource utilization, and a faster development process. The mobile app landscape continues to evolve day by day, however Flutter's toolkit and strong community support ensure that it will remain an important framework for developers want to build the next generation of cross-platform applications.

References

- [1] W. Oliveira, B. Moraes, F. Castor, and J. a. P. Fernandes, “Analyzing the resource usage overhead of mobile app development frameworks,” in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 152–161. [Online]. Available: <https://doi.org/10.1145/3593434.3593487>
- [2] 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,” *Empirical Software Engineering*, vol. 25, no. 4, pp. 2997–3040, 2020. [Online]. Available: <https://doi.org/10.1007/s10664-020-09827-6>
- [3] Flutter architectural overview. [Online]. Available: <https://docs.flutter.dev/resources/architectural-overview>
- [4] Google Trends. Interest over time. [Online]. Available: <https://trends.google.com/trends/explore?cat=31&date=2016-03-17%202024-03-17&q=Flutter,React%20Native,Xamarin,MAUI,Ionic>
- [5] Stack overflow trends. [Online]. Available: <https://insights.stackoverflow.com/trends?tags=flutter%2Creact-native%2Cxamarin%2Cmaui%2Cionic-framework%2Ccordova>
- [6] setState method. [Online]. Available: <https://api.flutter.dev/flutter/widgets/State/setState.html>
- [7] Kiran S. (Oct 30, 2023) Mastering flutter app architecture with bloc. [Online]. Available: <https://gurzu.com/blog/mastering-flutter-app-architecture-with-bloc/>
- [8] Rajeswari S. (Aug 23, 2021) Flutter redux-state management with redux in flutter. [Online]. Available: <https://medium.com/@rajeswari3699/flutter-redux-state-management-with-redux-in-flutter-7a6a13515f69>
- [9] Dmitrii Slepnev, “State management approaches in flutter,” 2020. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/355086/Dmitrii_Slepnev.pdf

- [10] Monikinderjit Singh. (Feb 16 2022) Integration testing in flutter: Getting started. [Online]. Available: <https://www.kodeco.com/29321816-integration-testing-in-flutter-getting-started>
- [11] Jignen Pandya. (June 13, 2024) Top 5 ways ci/cd streamlines your flutter app development (for faster delivery lower costs). [Online]. Available: <https://www.expertappdevs.com/blog/ways-ci-cd-streamlines-your-flutter-app-development#what-is-the-ci/cd-pipeline?>
- [12] Nazatul Nurlisa Zolkifli and Amir Ngah and Aziz Deraman, “Version control system: A review,” *Procedia Computer Science*, vol. 135, pp. 408–415, 2018, the 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877050918314819>
- [13] Chiradeep BasuMallick. What is version control? meaning, tools, and advantages. [Online]. Available: <https://www.spiceworks.com/tech/devops/articles/what-is-version-control/>
- [14] Deepak Sinha. What is the performance of flutter vs. native vs. react-native? [Online]. Available: <https://www.techaheadcorp.com/blog/what-is-the-performance-of-flutter-vs-native-vs-react-native/>
- [15] Oleg Goncharenko. Flutter vs. react native - detailed framework comparison. [Online]. Available: <https://brocoders.com/blog/flutter-vs-react-native/>
- [16] SPEC INDIA. Flutter vs react native: Choosing the right framework for your project. [Online]. Available: <https://www.linkedin.com/pulse/flutter-vs-react-native-choosing-right-framework-your-project-sp3vc/>
- [17] Quintagroup. Companies that use flutter.jpg. [Online]. Available: <https://quintagroup.com/services/service-images/companies-that-use-flutter.jpg/view>
- [18] Flutter Development. Flutter apps in production. [Online]. Available: <https://flutter.dev/showcase>
- [19] Flutter Documentation. Writing custom platform-specific code. [Online]. Available: <https://docs.flutter.dev/platform-integration/platform-channels>

- [20] A. Vats and S. Azim and A. S. Chauhan, "Chat messenger app using flutter," in *Proc. Int. Conf. Adv. Computing, Communication Control and Networking (ICAC₃N)*, 2023, pp. 1531–1535.
- [21] H. Chang and M. Varvello and F. Hao and S. Mukherjee, "A tale of three videoconferencing applications: Zoom, webex, and meet," *IEEE/ACM Transactions on Networking*, vol. 30, no. 5, pp. 2343–2358, 2022.
- [22] O. Schmid and A. Lisowska Masson and B. Hirsbrunner, "Real-time collaboration through web applications: an introduction to the toolkit for web-based interactive collaborative environments (twice)," *Personal and Ubiquitous Computing*, vol. 18, pp. 1201–1211, 2014.
- [23] Y. W. Syaifudin, D. D. Yapenrui, Noprianto, N. Funabiki, I. Siradjuddin, and H. N. Chasanah, "Implementation of self-learning topic for developing interactive mobile application in flutter programming learning assistance system," in *2024 ASU International Conference in Emerging Technologies for Sustainability and Intelligent Systems (ICETISIS)*, 2024, pp. 1103–1107.
- [24] F. Shirin Abkenar and L. Badia and M. Levorato, "Selective data offloading in edge computing for two-tier classification with local domain partitions," in *Proc. IEEE PerCom Workshops*, 2023, pp. 56–61.
- [25] M. Hasan and P. Biswas and M. T. I. Bilash and M. A. Z. Dipto, "Smart home systems: Overview and comparative analysis," in *Proc. IEEE ICRCICN*, 2018, pp. 264–268.
- [26] U. Michieli and L. Badia, "Game theoretic analysis of road user safety scenarios involving autonomous vehicles," in *Proc. IEEE PIMRC*, 2018, pp. 1377–1381.
- [27] L. Badia, "Analysis of age of information under sr arq," *IEEE Communications Letters*, vol. 27, no. 9, pp. 2308–2312, 2023.
- [28] S. S. Sefati and S. Halunga, "Ultra-reliability and low-latency communications on the internet of things based on 5G network: Literature review, classification, and future research view," *Trans. Emerging Telecommunications Technologies*, vol. 34, no. 6, p. e4770, 2023.

- [29] M. Favero, C. Schiavo, L. Verzotto, A. Buratto, T. Marchioro, and L. Badia, “Strategic Cooperation in the Metaverse: A Game Theory Analysis with Age Of Information,” in *Proc. IEEE IWCMC*, 2024.
- [30] A. Zancanaro, G. Cisotto, and L. Badia, “Modeling value of information in remote sensing from correlated sources,” *Computer Communications*, vol. 203, pp. 289–297, 2023.
- [31] K. Akarsu and O. Er, “Artificial intelligence based chatbot in e-health system,” *Artificial Intelligence Theory and Applications*, vol. 3, no. 2, pp. 113–122, 2023.
- [32] Ron Amadeo. ((2018-02-27)) Google starts a push for cross-platform app development with flutter sdk. [Online]. Available: <https://arstechnica.com/gadgets/2018/02/google-starts-a-push-for-cross-platform-app-development-with-flutter-sdk/>
- [33] Godwin Alexander Ekainu. ((2024-01-18)) Google starts a push for cross-platform app development with flutter sdk. [Online]. Available: <https://www.intelivita.com/en-tr/blog/top-flutter-app-development-tools/>
- [34] Flutter Development. Companies using flutter. [Online]. Available: <https://flutter.dev/>
- [35] Kulinski, Swav. ((2018-12-07)) Flutter — the sky’s the limit. [Online]. Available: <https://medium.com/kinandcartacreated/flutter-the-skys-the-limit-84887c8f650d>
- [36] Flutter blog. ((2019-12-11)) Flutter: the first ui platform designed for ambient computing. [Online]. Available: <https://developers.googleblog.com/en/flutter-the-first-ui-platform-designed-for-ambient-computing/>
- [37] Cubettech. ((2022)) Everything you need to know about flutter 2.0. [Online]. Available: <https://cubettech.com/resources/blog/everything-you-need-to-know-about-flutter-2/>
- [38] Migrating to null safety. Everything you need to know about flutter 2.0. [Online]. Available: <https://dart.dev/null-safety/migration-guide>
- [39] Add interactivity to your flutter app. [Online]. Available: <https://docs.flutter.dev/ui/interactivity>

- [40] Pavel Sulimau. Flutter vs competitors: Popularity. [Online]. Available: <https://pasul.medium.com/flutter-vs-competitors-popularity-f79536688ec3>
- [41] Educative: Interactive Courses for Software Developers. A bit about dart - learn dart: First step to flutter. [Online]. Available: <https://www.educative.io/courses/learn-dart-first-step-to-flutter/a-bit-about-dart>
- [42] Introduction to dart. [Online]. Available: <https://dart.dev/language#important-concepts>
- [43] M. Kochmański. 7 things you need to learn as a beginner flutter developer. [Online]. Available: <https://www.monterail.com/blog/flutter-guide-for-beginners>
- [44] V. V. Paridhi Wadhvani. (June 3, 2024) Flutter riverpod tutorial with usage advantages. [Online]. Available: <https://www.bacancytechnology.com/blog/flutter-riverpod-tutorial#:~:text=What%20is%20Riverpod%20Flutter%3F,request%20while%20also%20handling%20errors.>
- [45] B. Kapadiya. (Feb 1, 2024) Complete flutter bloc tutorial: Understanding state management in flutter. [Online]. Available: <https://www.dhiwise.com/post/flutter-bloc-tutorial-understanding-state-management>
- [46] D. Jolayemi. (Oct 25, 2021) Flutter redux: Complete tutorial with examples. [Online]. Available: <https://blog.logrocket.com/flutter-redux-complete-tutorial-with-examples/>
- [47] Dart testing. [Online]. Available: <https://dart.dev/guides/testing>
- [48] Muhammadumarch. (March,2023) Dart testing. [Online]. Available: <https://medium.com/@muhammadumarch321/testing-and-debugging-in-flutter-49d6ee7b4440>
- [49] PAUL. (Aug 29, 2023) Beginner guide to debugging testing in flutter — flutterconf23 talk. [Online]. Available: <https://edemekong.medium.com/beginner-guide-to-debugging-testing-in-flutter-flutterconf23-talk-5772a3a998df>
- [50] L. Tan. (Jan 29 2020) Unit testing with flutter: Getting started. [Online]. Available: <https://www.kodeco.com/6926998-unit-testing-with-flutter-getting-started>

- [51] D. Sinha. (Published: Jul 26, 2023) Testing and debugging flutter apps: A comprehensive approach. [Online]. Available: <https://www.techaheadcorp.com/blog/testing-debugging-flutter-apps-a-comprehensive-guide/>
- [52] J. Wogu. Flutter widget testing. [Online]. Available: https://medium.com/@Ikay_codes/flutter-widget-testing-68b32ccc93c8
- [53] Edward Chopskie. Unit testing vs. integration testing: 4 key differences and how to choose. [Online]. Available: <https://brightsec.com/blog/unit-testing-vs-integration-testing-4-key-differences-and-how-to-choose/>
- [54] Testing flutter apps. [Online]. Available: <https://docs.flutter.dev/testing/overview#unit-tests>
- [55] Aboubacar Abdou Abarchi. (Apr 14, 2023) Flutter continuous integration (ci) and continuous deployment (cd) pipeline with github actions. [Online]. Available: <https://medium.com/@ab3masta/flutter-continuous-integration-ci-and-continuous-deployment-cd-pipeline-with-github-actions-4fdcdb5>
- [56] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 471–482.
- [57] Eskola, Rasmus, "React native performance evaluation," Master's thesis, Aalto University, 2018. [Online]. Available: <https://aaltodoc.aalto.fi/items/5ad555c0-057b-4c11-a776-4aa4ff809ebb>
- [58] Priyansh Shah. (FEB 1, 2024) How does skia contribute as one of the key graphics engines in flutter apps. [Online]. Available: <https://www.dhiwise.com/post/how-does-skia-contribute-as-graphics-engines-in-flutter-apps>
- [59] OLEG SVET. Top industries and cases for using flutter. [Online]. Available: <https://computools.com/top-industries-and-cases-for-using-flutter/>
- [60] Inez Bartosińska. Apps made with flutter. [Online]. Available: <https://www.thedroidsonroids.com/blog/apps-made-with-flutter>

- [61] J. Pandya. Top real-world flutter marketplace apps you should know. [Online]. Available: <https://www.expertappdevs.com/blog/top-3-real-world-flutter-marketplace-apps>
- [62] InRhythm™. A comprehensive introduction to swift package manager. [Online]. Available: <https://medium.com/@GetInRhythm/a-comprehensive-introduction-to-swift-package-manager-20d248ec0066>