Republic of Cameroon

****

Peace-Work-Fatherland

****

Univeristy of Buea

****

Faculty of Engineering and Technology

Republique du Cameroun

****

Paix-Travail-Patrix

****

Univerisite de Buea

****

Faculte d'Engenieurie et de Technologie

********Department of Computer Engineering**********

General Overview of Mobile App Develpment

Presented by Group 13.

- **Members**

- Afayi Munsfu Obase Ngala **FE22A136**
- Atankeu Tchakoute Ange Natanahel **FE22A158**
- Betnessi Grace Blessing **FE22A446**
- Chesseu Njipdi Tertullien **FE22A181**
- James Aristide Massango **FE22A226**

- **Course Detail**

- **Course code/Title:** CEF440-Internet Programming and Mobile Programming
- **Course Instructor:** Dr. Nkemeni Valery
- **Credit Value:** 3

Table of Contents

List of Figures

List of Tables

1.0 COMPARING THE MAJOR TYPES OF MOBILE APPS

The In today's digital landscape, mobile applications have become an integral part of everyday life, serving diverse purposes across industries. However, not all apps are built the same. Developers and businesses must carefully choose the right type of app based on factors such as performance, cost, user experience, and target audience.

Mobile apps generally fall into three major categories: **native apps, progressive web apps (PWAs), and hybrid apps**. Each type offers distinct advantages and limitations, influencing development complexity, performance, and scalability. This section provides a comparative analysis of these mobile app types.

1.1 NATIVE APPS

Definition: Native apps are developed specifically for a particular platform (iOS, Android using platform-specific programming languages such as swift for iOS, Kotlin/Java for Android. They are usually best for high performance applications with extensive device feature requirements.

1.1.1 Pros

- **Performance:** Faster and more efficient as they are purposely optimized for the platform.
- **User Experience:** Excellent UX due to adherence to platform-specific design guidelines.
- **Access to Device Features:** Full access to device hardware and features (camera, GPS, etc.).
- **Offline Functionality:** Can work without internet access.

1.1.2 Cons

- **Development Cost:** Expensive to develop and maintain separate codebases for different platforms.
- **Time-Consuming:** Longer development time as each platform requires separate coding.
- **Updates:** Users must download updates through app stores.

1.1.3 Use Cases

- Applications requiring high performances and complex functionalities, such as gaming or multimedia editing.
- Applications needing deep integration with device hardware.

1.1.4 Some real-world examples include

- Instagram which needs native applications for both iOS and Android.
- Spotify which offers a rich user Experience with full access to device features.

1.2 PROGRESSIVE WEB APPS (WPAs)

Definition: PWAs are web applications that uses modern web technologies to provide a user experience similar to native apps, accessible via a web browser.

Ideal for businesses looking for wide reach with lower development costs ans instant updates.

1.2.1 Pros

- **Cross-Platform:** Works on any device with a browser, reducing development costs.
- **No Installation Required:** Can be accessed instantly without downloading from an app store.
- **Automatic Updates:** Updates happens automatically without user intervention.
- **Offline Capabilities:** Can work offline or on low-quality networks.

1.2.2 Cons

- **Limited Access to Device Features:** Restricted access to hardware compared to native apps.
- **Performance:** May not perform as well as Native apps for complex tasks
- **User Experience:** May not feel as smooth as Native apps, depending on the implementation.

1.2.3 Use Cases

- Applications that need to be quickly deployed and easily maintained.
- Businesses wanting to reach a broad audience without the overhead of app stores.

1.2.4 Some real-world examples include:

- Twitter lite as a PWA provides a fast and lightweight experience.
- Pinterest offers a PWA that is accessible on any device.

1.3 HYBRID APPS

Definition: Hybrid apps combines elements of both native and web apps. They are built using web technologies such as HTML, CSS, JavaScript and wrapped in a native container.

They are suitable for applications needing moderate performance with a single codebase for multiple platforms.

1.3.1 Pros

- **Single Codebase:** Develop once for multiple platforms, reducing development time and costs.
- **Access to Device Features:** Can access certain device features through plugins**.**
- **Faster Development:** Generally quicker to develop than Native apps.

1.3.2 Cons

- **Performance Issues:** May not perform as well as Native apps, especially for graphics-intensive tasks.
- **Depending on WebView:** Relies on web view for rendering, which van affect performance and UX.
- **Potentially Inconsistent UX:** May not fully adhere to platform-specific design guidelines.

1.3.3 Use Cases

- Applications that require a quick launch and have moderate performance needs.
- Businesses wanting to maintain a single codebase for multiple platforms.

1.3.4 Some real-world examples include

- Many applications built using Ionic framework are hybrid.
- Uber uses a hybrid approach for certain components of its app.

2.0 COMPARING MOBILE APPLICATION PROGRAMMING LANGUAGES

This section explores and compares some of the most widely used programming languages for mobile application development, highlighting their strengths, weaknesses, and ideal use cases. By understanding these differences, developers and businesses can make informed decisions when selecting a language that aligns with their project requirements, technical expertise, and long-term goals.

The phases involved in mobile app development includes choosing the right programming language which is crucial for building efficient, scalable, and user-friendly applications. Different programming languages offer distinct advantages depending on factors such as performance, development speed, platform compatibility, and community support.

2.1 OVERVIEW OF MAJOR MOBILE PROGRAMMING LANGUAGES (Kotlin, Swift, Dart, JavaScript, C#)

2.1.1 Kotlin

- **Description:** Kotlin is a modern programming language that runs on the Java Virtual Machine (JVM) and is fully interoperable with Java. It's the preferred language for Android app development.
- **Features:** Concise syntax, null safety, and extension functions.

2.1.2 Swift

- **Description**: Swift is a powerful and intuitive programming language created by Apple for iOS, macOS, watchOS, and tvOS app development**.**
- **Features:** Safety, performance, and modern programming features like closures and generics**.**

2.1.3 Dart

- **Description:** Dart is an open-source language developed by Google, primarily used with the Flutter framework for cross-platform app development**.**
- **Features:** Strongly typed, object-oriented, and designed for client-side development.

2.1.4 JavaScript

- **Description:** JavaScript is a dynamic, high-level scripting language used in web development. It can also be used for mobile app development with frameworks like React Native and Ionic.
- **Features**: Versatile, event-driven, and supports asynchronous programming.

2.1.5 C#

- **Description:** C# is a multi-paradigm programming language developed by Microsoft, primarily used with the Xamarin framework for cross-platform mobile app development**.**
- **Features:** Object-oriented, strong type system, and extensive libraries.

2.2 USE CASE SCENARIOS (WHERE THEY ARE USED)

2.2.1 Kotlin

- Ideal for Android apps; frequently used in enterprise-level applications.
- Supports modern Android features and integrates well with Java.

2.2.2 Swift

- Used for building native iOS applications, including games and consumer apps.
- Preferred for apps leveraging Apple's ecosystem and features.

2.2.3 Dart

- Primarily used for building cross-platform applications with Flutter.
- Used for high-performance apps that run on both iOS and Android from a single codebase.

2.2.4 JavaScript

- Commonly used for hybrid mobile app development with frameworks like React Native or Ionic.
- Suitable for apps that require web-view functionality and real-time data integration.

2.2.5 C#

- Used with Xamarin for cross-platform applications.
- Ideal for game development, enterprise apps, and applications that require a native look and feel on both iOS and Android.

2.3 COMPARISON ANALYSIS OF THE MAJOR MOBILE PROGRAMMINF LANGUAGES

Table 1: Comparison analysis of major mobile programming language

Aspect	Kotlin	Swift	Dart	JavaScript	C#
Performance	High performance due to JVM; optimized for Android.	Very high performance; optimized for Apple hardware.	Good performance; faster with AOT compilation in Flutter.	Variable; depends on runtime (React Native can lag).	Good performance; fast with Xamarin.
Efficiency	Concise syntax reduces boilerplate.	Safe and efficient with memory management.	Efficient for UI-heavy applications.	Moderate; can be inefficient in complex apps.	High efficiency; extensive libraries and frameworks.
Platform Support	Android (native).	iOS (native).	Cross-platform (iOS, Android).	Cross-platform with frameworks (React Native, Ionic).	Cross-platform with Xamarin (iOS, Android).
Popularity	Increasing; favored by Android developers.	Highly popular in iOS development.	Gaining popularity with Flutter.	One of the most popular languages globally.	Popular, especially in enterprise environments.

Ease of Learning	Moderate; familiar for Java developers.	Easy; clean syntax and modern paradigms.	Easy to learn for beginners; Flutter aids UI design.	Moderate; extensive resources available.	Moderate; C# is user-friendly, but Xamarin has a learning curve.
Development Speed	Fast for experienced developers; Kotlin Multiplatform for common code.	Quick development with rich libraries.	Fast due to hot reload in Flutter.	Varies; frameworks can speed up development.	Quick, with extensive support in Visual Studio.

3.0 COMPARING MOBILE APP DEVELOPMENT FRAMEWORKS

A Mobile App Development Framework is a library that provides the necessary foundational structure for creating mobile applications for a specific environment. It offers a structured way to build apps without having to write everything from scratch. Examples of such frameworks include Flutter, React Native, Ionic, Xamarin, and NativeScript. This section of the report provides a detailed comparison of three frameworks: Flutter, React Native, and Ionic, based on key technical and practical factors relevant to mobile application development. The frameworks are compared based on the following criteria: **architecture, key features, development tools, performance, cost & time to market, UI and UX, complexity.**

3.1 REACT NATIVE

Created by Facebook in 2015, React Native is an open-source framework used for mobile app development on Android and iOS platforms. It is based on React and JavaScript.

3.1.1 Architecture: React Native uses a bridge approach to invoke native platform components through JavaScript.

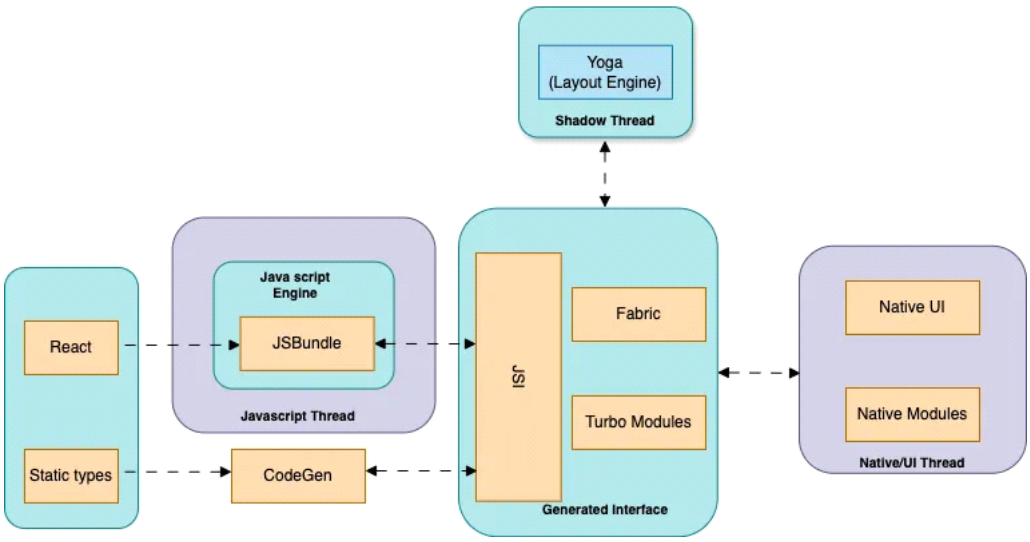

Figure 1: React Native architecture

3.1.2 Development Tools:

- Development Language: JavaScript
- IDEs: Atom, Nuclide, Visual Studio Code
- Tools: Expo, Redux, Ignite, Flow, Reduxsauce, ESLint, React Navigation

3.1.3 Performance

- Startup Time: React Native apps typically take 1.5x to 2x longer to start compared to native apps due to the JavaScript bridge.
- Frame Rate: Achieves 55–60 FPS in most scenarios but may drop below 40 FPS for complex animations.
- Memory Usage: Uses 20–30% more memory than native apps due to JavaScript execution overhead.

3.1.4 Cost & Time to Market: React Native reduces development costs by 30-50% vs. native apps. Development is 30-40% faster due to hot reloading and reusable components. For a medium-complexity app, expect 4-8 months and $50K-$120K, with ongoing maintenance at 15-20% of the initial cost annually.

3.1.5 UI and UX: React Native delivers high-quality UI/UX by combining native performance with flexible styling, allowing pixel-perfect designs that adapt seamlessly across iOS and Android. Libraries like Reanimated and React Navigation ensure smooth animations and intuitive user flows.

3.1.6 Complexity: React Native's complexity scales with app needs: simple apps use ready components (low complexity), mid-level apps need native bridges and optimizations (moderate), and advanced apps require native code/JSI expertise (high complexity).

3.1.7 Community Support:

- Backed by Meta and used by Fortune 500 companies with regular updates.
- 110K+ GitHub stars and 500K+ Stack Overflow questions indicate a vibrant activity.
- Key support channels: React Native Docs, GitHub Issues, Expo Forums, Stack Overflow, React Native Discord (50K+ members), DEV Community.

3.1.8 Use Cases:

- Social Media & Content Platforms: Facebook, Instagram, Pinterest.
- E-Commerce & Marketplaces: Shopify, Walmart, AliExpress.
- Fintech & Digital Banking: Coinbase, PayPal, Venmo.
- On-Demand Services: Uber Eats, Delivery.com.

3.2 FLUTTER

Flutter, created by Google in 2017, is an open-source UI toolkit for building natively compiled applications for mobile, web, and desktop from a single codebase. It uses the Dart programming language and the Skia rendering engine for high performance.

3.2.1 Architecture: Flutter compiles to native ARM code, resulting in faster startup times and improved performance. It uses a custom rendering engine (Skia) to draw widgets, ensuring pixel-perfect designs that adapt to different platforms.

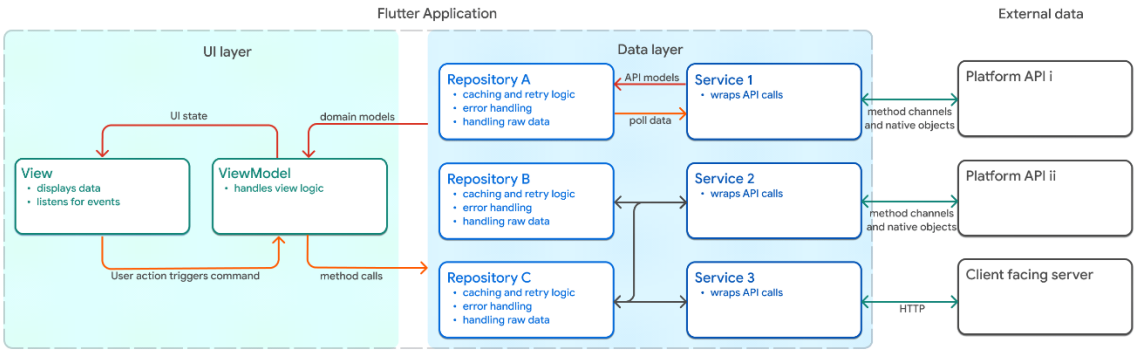

Figure 2: Flutter architecture

3.2.2 Development Tools

- Development Language: Dart
- IDEs: Android Studio, IntelliJ IDEA, Visual Studio Code
- Tools: Flutter DevTools, Dart Analyzer, Firebase, GetX for state management.

3.2.3 Performance

- Startup Time: Faster than React Native due to no reliance on a JavaScript bridge.
- Frame Rate: Achieves a consistent 60 FPS for most applications and supports 120 FPS for higher-end devices.
- Memory Usage: More memory-efficient than React Native due to direct compilation to native code.

3.2.4 Cost & Time to Market: Flutter development is efficient, thanks to its hot reload and pre-built widgets. For a medium-complexity app, development typically takes 4-7 months, with costs ranging from $50K-$110K. Maintenance costs are typically 10-15% of the initial cost annually.

3.2.5 UI and UX: Flutter offers excellent UI with customizable widgets. It uses Material Design for Android and Cupertino for iOS apps, ensuring platform consistency. Its custom Skia rendering engine ensures smooth animations, transitions, and pixel-perfect UI elements.

3.2.6 Complexity: Flutter is moderately complex, especially for large applications. Developers need to understand Dart, Flutter's widget tree system, and how to optimize performance for complex UIs. However, for smaller apps, Flutter is relatively easy to learn and use.

3.2.7 Community Support:

- Supported by Google with a growing community and regular updates.
- Over 150K stars on GitHub.
- Key support channels: Official documentation, GitHub, Stack Overflow, Flutter Community Slack, developer meetups.

3.2.8 Use Cases

- Fintech: Google Pay, Nubank.
- High-Performance Apps: BMW, eBay.
- Gaming: Google Stadia, Reflectly.
- Cross-Platform Enterprise Apps: Alibaba, Tencent.

3.3 IONIC

Ionic, created by Drifty Co. in 2013, is an open-source framework for building hybrid mobile applications. It uses standard web technologies (HTML, CSS, JavaScript) and runs inside a WebView, meaning it renders the app within a native wrapper.

3.3.1 Architecture: Ionic relies on WebView to render mobile apps and does not compile code to native ARM code like Flutter or React Native. It uses web technologies to build apps that run in a browser-like environment on mobile devices.

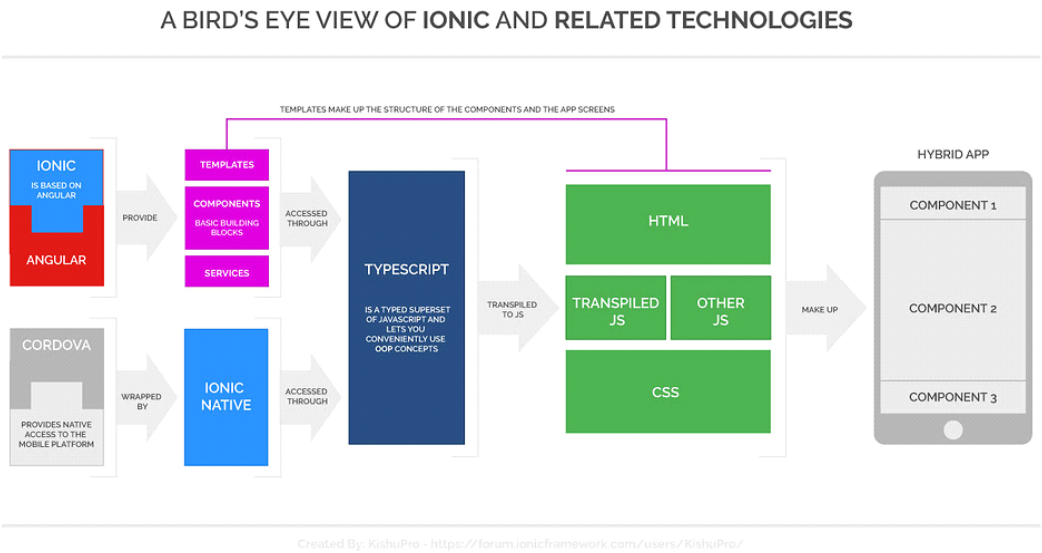

Figure 3: Ionic architecture

3.3.2 Development Tools

- Development Language: HTML, CSS, JavaScript (with Angular, React, or Vue)
- IDEs: Visual Studio Code, WebStorm
- Tools: Ionic CLI, Capacitor, Cordova, Stencil for building web components.

3.3.3 Performance

- Startup Time: Slower than Flutter and React Native due to WebView rendering.
- Frame Rate: Typically runs at 30-60 FPS, but complex apps may experience lag due to the WebView's limitations.
- Memory Usage: Uses more memory compared to React Native and Flutter due to reliance on WebView.

3.3.4 Cost & Time to Market: Ionic allows for rapid development, reducing costs and speeding up time to market. For a medium-complexity app, development time typically ranges from 3-6 months, with costs between $40K-$90K. Maintenance is usually lower, around 5-10% of the initial cost annually.

3.3.5 UI and UX: Ionic uses web-based UI components that mimic the native look and feel. However, since the app is rendered in WebView, animations and transitions are not as fluid as in Flutter or React Native. The UI may not always feel as native-like.

3.3.6 Complexity: Ionic is the easiest to learn among the three frameworks, especially for developers familiar with web development. The complexity increases when integrating native device features, but overall complexity remains relatively low.

3.3.7 Community Support

- A smaller, but dedicated community compared to Flutter and React Native.
- Key support channels: Ionic Forum, GitHub, Stack Overflow, Ionic Discord.

3.3.8 Use Cases

- Hybrid Mobile Apps: MarketWatch, Sworkit.
- Progressive Web Apps (PWAs): Ionics's own website.
- Business Apps: Internal apps and MVPs.
- Rapid Prototyping: Startups or businesses needing fast-to-market solutions.

3.4 COMPARISON SUMMARY

Table 2: Comparison summary of major programming language

Aspects	Flutter	React Native	Ionic
Feature overview	the best performance and native-like UI/UX, making it suitable for apps requiring high performance and custom UI elements	It's the most popular choice, allowing faster development with a single codebase for both iOS and Android, but may struggle with complex animations.	It is best for hybrid mobile apps, PWAs, and rapid prototyping, though it has performance and native UI limitations.
Performance	Leads in performance, especially with complex animations and smoother UI rendering.	Provides good performance but falls behind Flutter in complex cases due to its reliance on JavaScript.	Lags behind both due to its WebView-based approach.
Cost & Time to market	Flutter generally offers faster development times and reduced costs compared to native apps,	React Native generally offer faster development times and reduced costs compared to native apps,	Ionic provides a fast development cycle but at a potentially lower cost.

UI/UX	provides the most customizable and polished UI for both iOS and Android	Delivers a good native-like experience but may need additional native code for highly customized UIs	offers basic native-like UI components but lacks the smooth animations and customizability of the other two
Community support	It is rapidly growing, supported by Google, and has a vibrant community	Has the largest community, extensive resources, and enterprise backing	Has a smaller, but solid community, making it a good choice for simpler applications.

3.5 RECOMMENDATIONS

Choose Flutter for:

- High-performance apps with complex animations and custom UI.
- Applications requiring a consistent look and feel across both iOS and Android.
- Projects needing smooth user interactions and pixel-perfect designs.

Choose React Native for:

- Projects with moderate complexity where time to market is a priority.
- Teams with experience in JavaScript and React.
- Applications needing native-like performance with a shared codebase across platforms.

Choose Ionic for:

- Hybrid apps or MVPs where rapid development and quick time to market are important.
- Businesses looking for PWAs or simpler mobile apps without the need for high performance.
- Developers familiar with web technologies (HTML, CSS, JavaScript) who want to build cross-platform apps quickly.

4.0 STUDY OF MOBILE APPLICATION ARCHITECTURE AND DESIGN PATTERNS

Most often when we build apps as students, we tend to forget to take some time to determine and design the architecture of that app for one or more of these reasons. It may be due to;

- We don’t know which app architecture fits our mobile app requirements.

- We find it not important or necessary.

But in this study, we will discover the importance of mobile app architecture and design patterns which turn to ease app **maintainability, scalability and flexibility**, creating a clear difference between Enterprise-built apps and Home-built apps (mobile apps developed by students).

First, let's dive into the details of mobile app architecture.

4.1 MOBILE APP ARCHITECTURE

I. Definition

Mobile app architecture is the blueprint for building an app, defining how its components (like UI, backend, and database) interact to **process inputs** and **produce outputs**, ensuring scalability, maintainability, and a smooth user experience.

Why is App Architecture Essential?

Quality architecture helps with risk management and enables cost reductions. An application with robust, well-planned architecture is more likely to succeed in its target market. Any mobile app project starts with the planning and designing phase, and choosing the right architecture is a core priority. An insufficient approach to this step can slow down the development process and make it more extensive. It can also lead to various performance issues and system failures.

II. Examples of Mobile App Architecture

Some Examples of mobile app architecture include*;* **Mobile View Controller (MVC), Mobile View Presenter (MVP), Mobile View View-Model (MVVM), Layered Architecture and Clean Architecture.** Amongst these, the most commonly used architectures include **MVC, MVP, MVVM** which will be discussed in detail in this section.

Note: These patterns are widely used to moderate complex codes and simplify UI code by making it neater and more manageable. MV(X) architectures divide the visualizing, processing, and data management functions for UI applications, which increases an app's modularity, flexibility, and testability.

Let us take a closer look at each model to understand their differences.

4.1.1 Model-View-Controller (MVC)

MVC is commonly used when designing simple applications as it's more readily modified than the other two and makes implementing changes to the app simple.

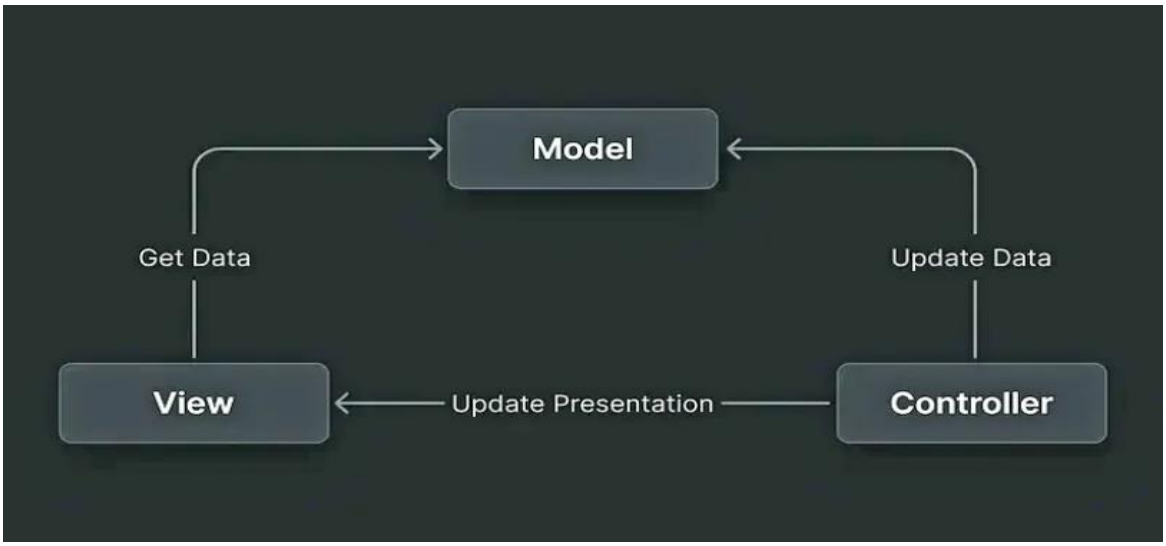

Figure 4: MVC schema

MVC consists of three components: **Model**, **View**, and **Controller**. "Model" is subject to the app's business logic and manages the state of an application and handles data changes and manipulations. "View" manages UI elements by presenting data to users and managing user interactions. "Controller" mediates between view and model by processing incoming requests. Depending on an app's requirements, there may be one or more controllers.

This pattern enables a faster development process and offers multiple views for the model.

i. Strengths

- **Scalability:** MVC architecture allows for easier scaling of the application as it grows.
- **Faster Development:** MVC supports rapid and parallel development.
- **Easy Collaboration:** MVC makes it easier for multiple developers to collaborate and work together.
- **Supports multiple views:** MVC allows for multiple views of the same model.
- **Flexibility:** MVC offers greater flexibility in terms of URL routing, result types, and view engines.

ii. Weaknesses

- **Overhead:** MVC can add unnecessary complexity and overhead for small or simple applications.
- **Complexity of Understanding:** Understanding the nuances of each component and how they interact requires training and experience.

- **Inefficiency of data access in view:** The view component might be overburdened with update requests if the model changes frequently.

Examples

- **Instagram (older versions)**
- **Pinterest (older versions)**
- **Twitter (before switching to MVVM and React)**

4.1.2 Model View Presenter (MVP)

MVP is derived from the MVC pattern, and here, the "**controller**" is replaced by "**presenter**". Performance-wise this pattern offers high reliability as there is less hindrance than with the other two models in terms of rendering frames.

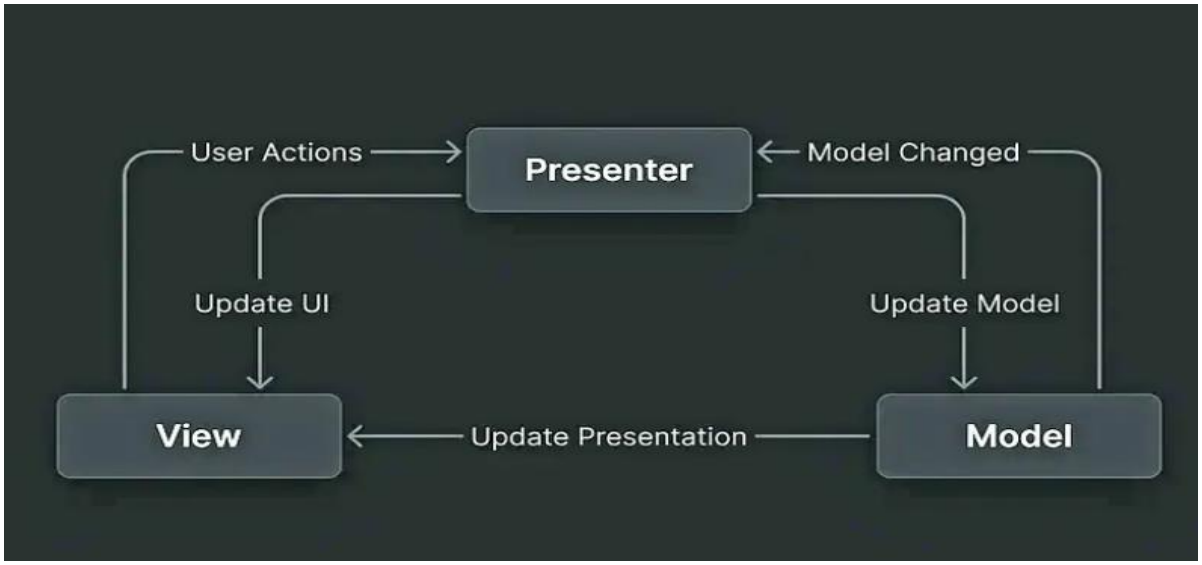

Figure 5: MVP schema

Similar to MVC, the model covers the app's business logic and how data is handled while the view is separated from the logic implemented in the process. The presenter's major function is to manipulate the model and update the view. When receiving input from a user via view, the presenter processes the data and sends the results back to view.

MVP offers easier debugging and allows code reusability.

MVP architecture in mobile app development, with its Model-View-Presenter structure, offers improved testability and maintainability by separating UI logic from business logic, but can introduce some complexity and tight coupling

i. Strengths of MVP Architecture:

- **Enhanced Testability:** The Presenter can be easily mocked for unit testing, allowing developers to test the business logic independently of the UI.
- **Improved Maintainability:** The separation of concerns makes it easier to understand, modify, and maintain the codebase.
- **Modularity and Reusability:** Components are more independent, leading to easier reuse of code across different screens or features.
- **Clear Separation of Concerns:** The Model handles data and business logic, the View displays the UI, and the Presenter mediates interactions between them, promoting a cleaner codebase.
- **Better handling of complex logic:** The Presenter can handle complex logic and communication between different components, making the code more organized.

ii. Weaknesses of MVP Architecture:

- **Tight Coupling:** The Presenter has a direct reference to the View, which can lead to tight coupling between components.
- **Increased Complexity:** The MVP architecture can be more complex than simpler architectures like MVC, requiring more boilerplate code and potentially increasing the learning curve.
- **Potential for Performance Issues:** If not implemented correctly, the direct interaction between Presenter and View can lead to performance bottlenecks.
- **Lifecycle Awareness:** The Presenter doesn't contain anything related to lifecycle aware of Views like Activities or fragments.

Examples

- **Android Google Apps (Older versions)** – Gmail, YouTube (before moving to MVVM).
- **Spotify (Android)** – Earlier versions used MVP to separate UI logic from business logic.
- **Uber** – Previously relied on MVP for structuring its app efficiently before moving to MVVM.

4.1.3 Model-View-ViewModel (MVVM)

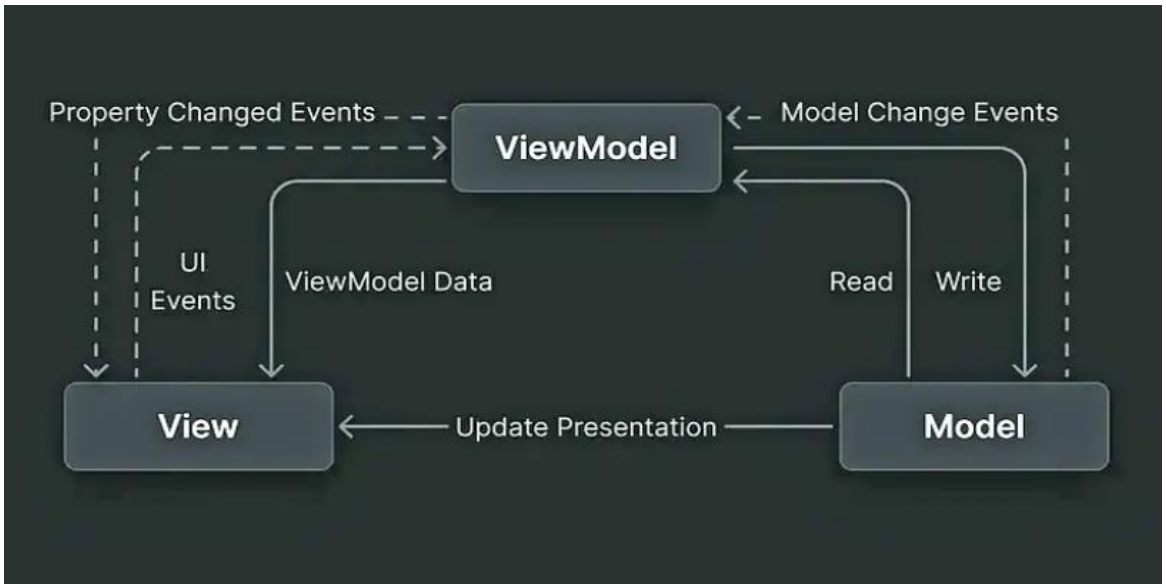

Figure 6: MVVM schema

Designed for more explicit separation of UI development from business logic, MVVM is similar to MVC. The “model” here handles basic data and “view” displays the processed data. The View-Model element discloses methods and commands that help maintain the state of view and control the model. Two-way data binding synchronizes models and properties with the view. Due to data binding, this pattern has higher compatibility than others.

MVVM’s significant advantages include easier testing and maintenance as it allows developers to readily implement changes because the different kinds of code are separated.

i. Strengths

- **Simplified Maintenance:** The separation of concerns makes it easier to understand, modify, and maintain the application's codebase.
- **Easier Development:** Separating the View from the logic makes it possible to have different teams work on different components simultaneously.
- **Lifecycle Awareness:** MVVM leverages lifecycle-awareness and data binding, making it easier to handle events and data updates.
- **Scalability:** MVVM allows developers to build scalable and robust applications with ease.

ii. Weaknesses

- **Increased Complexity:** MVVM can be more complex to implement and understand than simpler architectures like MVC, especially for smaller projects.
- **Steeper Learning Curve:** Developers need to learn the MVVM pattern and its related concepts, which can take time.
- **Potential Over-Engineering:** The separation of concerns can sometimes lead to over-engineering, especially for small projects.
- **Debugging Challenges:** Because data binding is declarative, it can be harder to debug than traditional, imperative code.
- **ViewModel to ViewModel binding:** Can be complex.
- **Service & IOC are complex for midsize applications**.

Examples

- **Google Apps (Modern versions)** – Gmail, YouTube, Google Maps (Android uses Jetpack MVVM).
- **Microsoft Outlook (Android & iOS)**
- **Netflix** – Uses MVVM for managing streaming UI and recommendations.
- **Airbnb** – Uses MVVM for reactive UI updates with data-binding techniques.

To sum up, MVP and MVVM allow developers to break an app down into modular, single-purpose components. At the same time, these two patterns add more complexity to an application. If you intend to build a simple application with one or two screens, MVC may be a better solution. MVVM, meanwhile, works well in more complex applications that handle data from elsewhere, be it a database, file, web service, and so on.

4.2 MOBILE APP DESIGN PATTERNS

I. Definition

In mobile app development, a design pattern is a reusable solution to a common problem, serving as a blueprint for creating maintainable, extensible, and efficient applications by addressing recurring design challenges.

II. Examples

Here are some mobile App design patterns commonly used;

1. Singleton
2. Factory Method

3. Observer
4. Adapter
5. Strategy
6. Repository

Table 3: Design patters

Pattern	Key Use Case	Examples
Singleton	Single instance for global access	Spotify, Google Maps (Playback Manager, Location)
Factory Method	Creating different objects dynamically	Uber (Ride types), Amazon (Shipping options)
Observer	Event-driven updates	Facebook (Live notifications), WhatsApp (Status updates)
Adapter	Bridging incompatible interfaces	Amazon (Product lists), Netflix (Movie lists)
Strategy	Switching between multiple algorithms	PayPal (Payment methods), Google Search (Sorting)
Repository	Clean data management layer	Netflix (Caching + API), Google Drive (Syncing)

4.2.1 SINGLETON PATTERN

☑ Contribution to Code Reusability, Modularity & Maintainability

- Ensures **a single instance** of a class is shared across the app.
- Reduces memory footprint by avoiding multiple object instantiations.
- **Decouples dependencies** by providing a central access point.

◇ Best Practices for Mobile Platforms

- **iOS (Swift)**: Use static let shared = SingletonClass ( ).
- **Android (Kotlin/Java)**: Use object (Kotlin) or private constructor (Java).
- **Avoid storing UI elements** in a Singleton to prevent memory leaks.

⚡ Performance Cost

- If used improperly, it **can lead to tight coupling**, making testing difficult.
- Can cause **memory leaks** if it holds long-lived references to UI components.

4.2.2 FACTORY METHOD PATTERN

☑ Contribution to Code Reusability, Modularity & Maintainability

- Centralizes **object creation logic**, making code easier to maintain.
- Promotes **loose coupling** between classes.
- Makes it easier to introduce **new object types** without modifying existing code.

◇ Best Practices for Mobile Platforms

- **iOS (Swift)**: Implement Factory with protocols for different object creation.
- **Android (Kotlin/Java)**: Use sealed classes or interface for object creation.
- **Use Dependency Injection (DI)** with the Factory to improve testability.

Performance Cost

- Factory Method can introduce **slight overhead** due to abstraction.
- If overused, can lead to **excessive class creation**, reducing readability.

4.2.3 OBSERVER PATTERN

☑ Contribution to Code Reusability, Modularity & Maintainability

- Implements **event-driven programming**, allowing **multiple components to react** dynamically.
- Decouples **event emitters from event listeners**.
- Improves **scalability** for UI updates.

◇ Best Practices for Mobile Platforms

- **iOS (Swift)**: Use Combine or NotificationCenter.
- **Android (Kotlin/Java)**: Use **LiveData**, RxJava, or Flow.
- Use **weak references** to prevent memory leaks.

Performance Cost

- **Frequent updates** can cause UI performance issues.
- Observers that are **not removed properly** can lead to memory leaks.

4.2.4 ADAPTER PATTERN

☑ Contribution to Code Reusability, Modularity & Maintainability

- Acts as a **bridge between incompatible interfaces**, making code more reusable.
- Useful for integrating **third-party libraries** or legacy code.
- Increases maintainability by providing a **consistent interface**.

◇ Best Practices for Mobile Platforms

- **iOS (Swift)**: Use Protocols & Extensions to create adapters.
- **Android (Kotlin/Java)**: Used extensively in RecyclerView Adapter.

- Ensure **Adapter does not introduce performance overhead** due to data conversion.

Performance Cost

- Can introduce **slight overhead** if too many conversions are required.
- Complex adapters can **increase code complexity** and **reduce readability**.

4.2.5 STRATEGY PATTERN

Contribution to Code Reusability, Modularity & Maintainability

- Allows **different algorithms** to be interchangeable dynamically.
- Improves **code maintainability** by **separating business logic** from implementation.
- Makes the code more **flexible and testable**.

Best Practices for Mobile Platforms

- **iOS (Swift)**: Use **protocol-oriented programming**.
- **Android (Kotlin/Java)**: Use **sealed classes or interfaces** for different strategies.
- Implement Strategy with **Dependency Injection** for better modularity.

Performance Cost

- Can lead to **code duplication** if too many strategy classes are created.
- **Extra object creation** may introduce **memory overhead**.

4.2.6 REPOSITORY PATTERN

Contribution to Code Reusability, Modularity & Maintainability

- Provides **a clean separation** between data access logic and business logic.
- Makes the application **independent of the data source** (e.g., local database, API, cache).
- Improves **code organization** and **testability**.

Best Practices for Mobile Platforms

- **iOS (Swift)**: Implement Repository with CoreData or Realm.
- **Android (Kotlin/Java)**: Use Room + Repository in MVVM architecture.
- Implement **caching strategies** for optimized data fetching.

Performance Cost

- If **not optimized**, can **increase API calls**, causing **battery drain**.

- Can introduce **latency issues** if implemented with poor data caching.

Conclusion: Differences Between These Patterns

Table 4: Differences between design patterns

Pattern	Purpose	Key Benefit	Common Use Case
Singleton	Global instance	Memory efficiency	Network manager, logging, database connections
Factory Method	Object creation	Flexibility	Different payment methods, notifications
Observer	Event-driven updates	Decoupling	UI event listeners, real-time updates
Adapter	Interface compatibility	Reusability	RecyclerView, API bridging
Strategy	Dynamic algorithm selection	Modularity	Payment methods, sorting strategies
Repository	Data management	Maintainability	Offline storage, API caching

Final Thoughts

- **Singleton** is good for global states but can **cause tight coupling**.
- **Factory Method** simplifies object creation but can **introduce complexity**.
- **Observer** is great for UI updates but **must be managed properly** to avoid memory leaks.
- **Adapter** makes **legacy code integration easier**, but overuse can reduce performance.
- **Strategy** increases **flexibility** but can create **too many classes**.
- **Repository** is **best for handling data** but must **be optimized for performance**.

5.0 COLLECTING AND ANALYZING USER REQUIREMNTS FOR MOBILE APPLICATION (REQUIREMENT ENGINERING)

Requirement engineering is the discipline that involves establishing and documenting requirements.

User requirements for a mobile app encompasses the specific needs, expectations, and desires of the target audience guiding the design and development process to ensure the app meets their needs and deliver a positive user experience. They can also been seen as valuable insights into what your target audience needs and wants.

5.1 PROCESS OF COLLECTING USER REQUIREMNTS

Step 1- Assigning roles: The first step is to identify and engage with all relevant stakeholders. Stakeholders can include end-users, clients, project managers, subject matter experts, and anyone else who has a vested interest in the software project. Understanding their perspectives is essential for capturing diverse requirements.

Step 2- Define Project Scope: Clearly define the scope of the project by outlining its objectives, boundaries, and limitations. This step helps in establishing a common understanding of what the software is expected to achieve and what functionalities it should include.

Step 3- Conduct Stakeholder Interviews: Schedule interviews with key stakeholders to gather information about their needs, preferences, and expectations. Through open-ended questions and discussions, aim to uncover both explicit and implicit requirements. These interviews provide valuable insights that contribute to a more holistic understanding of the project.

S**tep 4- Document Requirements:** Systematically document the gathered requirements. This documentation can take various forms, such as user stories, use cases, or formal specifications. Clearly articulate functional requirements (what the system should do) and non-functional requirements (qualities the system should have, such as performance or security).

Step 5- Verify and Validate Requirements: Once the requirements are documented, it's crucial to verify and validate them. Verification ensures that the requirements align with the stakeholders' intentions, while validation ensures that the documented requirements will meet the project's goals. This step often involves feedback loops and discussions with stakeholders to refine and clarify requirements.

Step 6- Prioritize Requirements: Prioritize the requirements based on their importance to the project goals and constraints. This step helps in creating a roadmap for development, guiding the team on which features to prioritize. Prioritization is essential, especially when resources and time are limited.

The above points can be summarized using the figure below:

Figure 7: Processes involved in collecting user requirements

5.2 METHODS OR TECHNIQUES FOR COLLECTING USER REQUIREMENTS

Here are some commonly used techniques to collect user requirements:

1. **Interviews:** Conducting one-on-one or group interviews with stakeholders, including end-users, clients, and subject matter experts. This allows for direct interaction to gather detailed information about their needs, expectations, and concerns.
2. **Surveys and Questionnaires:** Distributing surveys and questionnaires to a broad audience to collect information on a larger scale. This technique is useful for gathering feedback from a diverse set of stakeholders and can be particularly effective in large projects.
3. **Workshops:** Organizing facilitated group sessions or workshops where stakeholders come together to discuss and define requirements. Workshops encourage collaboration, idea generation, and the resolution of conflicting viewpoints in a structured environment.
4. **Observation:** Directly observing end-users in their work environment to understand their workflows, pain points, and preferences. Observational techniques help in uncovering implicit requirements that users might not explicitly state.
5. **Prototyping:** Creating mockups or prototypes of the software to provide stakeholders with a tangible representation of the proposed system. Prototyping

allows for early visualization and feedback, helping to refine requirements based on stakeholders' reactions.

6. **Use Cases and Scenarios:** Developing use cases and scenarios to describe how the system will be used in different situations. This technique helps in understanding the interactions between users and the system, making it easier to identify and document functional requirements**.**
7. **Document Analysis:** Reviewing existing documentation, such as business process manuals, reports, and forms, to extract relevant information. This technique provides insights into the current processes and helps identify areas for improvement.

The above points can be summarized with the below illustration:

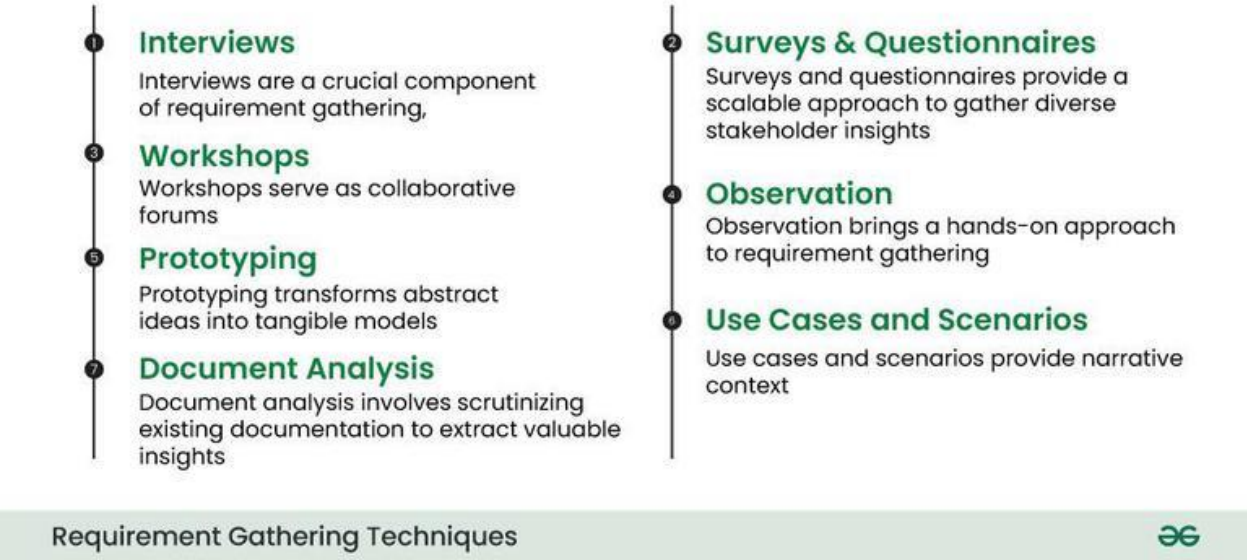

Figure 8: Techniques used for collecting user requirements

5.3 BENEFITS OF COLLECTING USER REQUIREMENTS

Collecting user requirements immensely help in mobile app development for several reasons like;

- Clarity of project objectives
- Customer satisfaction
- Reduced misunderstandings
- Risk mitigation
- Cost optimization by removing unnecessary features

5.4 METHODS FOR ANALYZING USER REQUIREMENTS

Analyzing user requirements refers to the process of examining, refining, and validating the needs and expectations of users for a mobile application. It ensures that the final product meets needs, business goals, and technical feasibility. This step is critical in the requirement engineering phase of mobile app development and helps avoid costly revisions later in the project.

Some key aspects of analyzing user requirements include;

1. **Understanding user needs**: User requirement analysis is important because it helps to identify and understand what users want to achieve with the mobile app. This also helps to differentiate between features, which can be **essential** and **optional** features.
2. **Classification of requirements**: The analysis of user requirements helps to be able to classify it into **functional** (what the mobile app should do. E.g. login, notifications, signup) and **non-functiona**l (performance, security, scalability and UI/UX standards) requirements.
3. **Prioritizing requirements**: When the requirements have been identified, it need to be categorized as features such as **Must-Have, Should-Have, Could-Have** and **Won't-Have (MoSCoW Method).** This ensures critical functionalities are developed first.
4. **Validating Requirements:** The user requirements can be analyzed by checking feasibility in terms of cost, time, and technology. This ensures alignment with business objectives and user expectations.
5. **Documenting Requirements:** To ease the analysis, tools like use case diagrams and prototypes can be done. The creation of a proper documentation is also done to provide a summarized view of the analyzed user requirements.

Some best practices that should be done in the analysis of user requirements include:

1. **Involve stakeholders from the start:** Engage with clients, users, subject matter experts, and other relevant stakeholders from the beginning. They can provide insights, clarify expectations, and contribute valuable ideas which might not be considered otherwise. Regular communication with stakeholders throughout the analysis process ensures their needs are being addressed.
2. **Prioritize requirements:** Prioritize the user requirements based on their importance and impact on the mobile app's success. This helps in managing expectations and focusing on delivering the most essential features first
3. **Use Visual Aids and Tools:** Visual aids, such as diagrams, flowcharts, and wireframes, significantly enhance the user requirements analysis. They provide a

clearer understanding of how different components will interact and how the final product will function. Visuals can bridge the gap between technical teams and non-technical stakeholders, making discussions more productive.

6.0 ESTIMATING MOBILE APP DEVELOPMENT COST

The process of developing a mobile app for personal or business use, generally includes the aspect of cost estimation for the realization of the development of the mobile application. Estimating the cost of developing a mobile app involves several factors, and this guide will walk you through seven steps to help you get a rough estimate of the expenses.

Steps to Calculate the Cost for Mobile App Development

Step 1: Define Your App's Objective and Features

To estimate the cost of your app, start by determining its goal and features. What do you want your app to achieve? What functionalities do you need? Make a list of features, prioritizing them based on importance and complexity. This will lay the groundwork for estimating the cost.

For example, if an app has aim to help users find local restaurants and make reservations. Key features could include searching for restaurants by cuisine, location or name, viewing restaurant details and booking reservations directly app. More complex features like online food ordering may require additional development hours. Clearly defining your app's purpose and minimum viable product will allow for a more accurate cost estimate.

Step 2: Choose the Platform

Deciding between iOS, Android, or both platforms is crucial for determining development costs. Developing separate apps for each platform requires more resources compared to creating a cross-platform app. Each platform has unique design principles and development complexities that impact the overall cost.

For instance, building native iOS and Android apps would mean employing developers skilled in Swift/Objective-C and Java/Kotlin respectively, whereas a cross-platform framework like React Native or Flutter allows using the same codebase for both stores with some customization. Consider your target users and distribution strategy when deciding on platforms.

Table 5: Cost estimation based on choice of platform

Single-Platform Apps (iOS or Android)	Cross-Platform Apps
Developed specifically for one platform using native languages.	Uses a single codebase to run on both iOS and Android.
Best performance, better hardware integration, smoother UI.	Faster development, lower cost.
Higher cost if developing for both platforms separately.	Slight performance trade-off, limited access to certain native features
Estimated Cost: $20,000 - $100,000 per platform	**Estimated Cost:** $30,000 - $150,000 (covers both platforms).

Step 3: Evaluate App Design Complexity

The design of your app plays a significant role in its appeal and usability. Consider whether your app would benefit from a simple and minimalistic design or a more elaborate and visually captivating interface. Keep in mind that intricate designs with custom animations and interactions can increase development costs. For example, a basic design with one or two screen views per section may take 50–100 hours whereas an immersive design with advanced motion graphics could easily exceed 200 hours for the visual design work alone.

Table 6: Cost estimation based on complexity

Simple Apps	Medium Complexity Apps	Complex Apps
Basic UI with minimal screens	Custom UI and animations	Advanced UI/UX with custom animations
No complex backend or real-time features	Basic backend with database connectivity	AI-powered features, real-time chat, or video streaming
-	Integration with APIs (e.g., social login, payment processing)	Complex backend architecture with cloud computing
Example: To-do list apps, calculators	Example: Fitness tracker apps, food delivery apps	Example: E-commerce platforms, social media apps, enterprise apps
$10,000 - $30,000	$30,000 - $100,000	$100,000 - $300,000+

Step 4: Estimate Development Hours

Once you have a clear understanding of your app's features and design, estimate the number of development hours needed for each task. Break down the project into smaller components and allocate hours based on complexity. Using industry benchmarks or seeking guidance from developers can help you make a more accurate estimation.

For instance, a login screen may take 10 hours whereas integrating payment processing could take 30–50 hours depending on the provider. Don't forget to factor in testing, bug fixes and other non-development tasks into your hour estimates.

Step 5: Research Development Rates

Hourly rates for mobile app development vary depending on regions and development teams. Take the time to research average rates in major cities like San Francisco, New York, London and Bangalore which range from $50–150 per hour depending on the seniority and expertise of the developers. Make sure to check rates on reputable freelancing sites and job boards.

Remember that a higher rate doesn't always guarantee better quality. Read reviews from past clients, examine examples of applications developed by the team in their portfolio, and evaluate the specific technical skills and experience levels of the core team members before making a decision. Pay close attention to reviews that comment on factors like responsiveness, ability to work within budgets and timelines, and quality of project management.

Step 6: Consider Additional Expenses

In addition to development hours, there are other costs to consider. Quality assurance and testing are essential for ensuring all features and functionality of the application work as intended across different device types and operating system versions, so allocate at least 15–20% of the development budget for thorough testing and fixing any bugs found.

Also, factor in the current publishing fees which are **$99 per year for the Apple App store** and **$25 per upload for the Google Play Store** when estimating total costs. Regular updates to address compatibility issues, bugs and add new features should also be budgeted for, estimating at least 2–3 updates per year.

Step 7: Include a Contingency Budget

Unexpected challenges like changes to project scope, integration issues between features, or bugs that take significant time to resolve can arise during development, leading to delays and extra expenses. It's wise to include **a contingency budget of 15–20% of the total estimated development costs** to help account for any unforeseen obstacles that may come up over the course of the project. This helps reduce financial risks.

In the analysis, the integration of some third parties should be thought. Some of these may include:

Integrating external services adds functionality but also increases development costs.

API Integrations

- **Common APIs:** Payment gateways (Stripe, PayPal), Google Maps, social media login.
- **Cost Factor:** $1,000 - $10,000 depending on complexity.

Database & Cloud Storage

- **Database Choices:** Firebase (NoSQL), PostgreSQL, MySQL.
- **Cloud Providers:** AWS, Google Cloud, Microsoft Azure.
- **Cost Factor:** $5,000 - $50,000 (includes server setup, maintenance).

Push Notifications & Real-Time Features

- **Cost Factor:** $1,000 - $5,000 (Firebase Cloud Messaging, OneSignal).