University of Buea

Faculty of Engineering & Technology

Department of Computer Engineering

CEF 440: Internet & Mobile Programming

Task 1:

Introduction to Mobile Programming

Group 14

	Name	Matricule
1	Agyingi Rachel Mifor	FE22A140
2	Jane Ahone Eloundou	FE22A227
3	Ndintawel Sophie Te-e	FE22A253
4	Dione Melina Makoge	FE22A188
5	Timah Berry Naura Enyauh	FE22A318

Project Supervisor: Dr Nkemeni Valery

Table of Content

Chapter 1: Major Types of Mobile Applications 3
1.1 Types of Mobile Applications 3
1.2 Similarities: 4
1.3 Differences 4
Chapter 2: Mobile App Programming Languages 8
2.1. Native Mobile App Languages 8
2.2. Cross-Platform Mobile App Languages 8
2.3. Web-Based Mobile App Languages (for PWAs and Hybrid Apps) 9
Chapter 3: Mobile Development Frameworks 10
3.1 Types of Mobile Development Frameworks 10
3.2 A Review of Some Frameworks 11
3.3 Comparative Review: SwiftUI vs Flutter vs Ionic vs PWA 22
3.4 Best Use Cases for Each Framework 23
Chapter 4: Mobile App Architecture & Design Patterns 26
4.1 Introduction 26
4.2 The Fundamental Layers of Mobile App Architecture 26
4.3 Key Principles of a Good App Architecture 30
4.4 How To Choose the Right Architecture 31
Chapter 5: Requirement Engineering 32
5.1 Introduction 32
5.2 Importance of Requirements Engineering to Mobile App Development32
5.3 Key Challenges and Trends to Requirements Engineering 33
5.4 Requirement Engineering Process 34
5.5 Tools & Techniques for Requirement Engineering 36
Chapter 6: Steps to estimate cost for a mobile app development: 37
Conclusion: 39

Chapter 1: Major Types of Mobile Applications

In the age of technology, smartphones are widely used by billions of people. A key feature of that makes a smartphone “smart” is the ability of their operating systems to run various types of mobile applications such as: email, browsers and social media applications.

A mobile application is a software program designed to run on a mobile device such as a smartphone or a tablet PC rather than desktop or laptop computers. They provide specific functionalities and services to users.

The three major types of mobile apps include:

- Native applications:
- Progressive web applications:
- Hybrid applications:

The decision to develop a mobile application involves choosing between different development approaches. Each approach has distinct advantages and disadvantages across key factors such as performance, user experience, development, complexity, accessibility , offline usage and security. This report provides a detailed comparison of these app types to help highlight the best options for a developer’s needs.

1.1 Types of Mobile Applications

Native applications:

These are mobile applications built for a specific platform, such as iOS or Android, using its native programming language and development tools. Languages such as Kotlin/Java are used for Android and Swift/Objective-C for iOS. They have a direct access to device capabilities, ensuring high performance and smooth user experience.

Progressive web applications:

These are web applications that behave like native applications while being accessible through a web browser. PWAs use modern web technologies (HTML, CSS and

JavaScript) and offer app-like features on various platforms such as: offline functionality, push notifications and home screen installation.

Hybrid applications:

They are mobile applications that combine elements from both native apps and web apps. They are built using web technologies but is wrapped in a native container and deployed on various platforms.

1.2 Similarities:

- Native applications, hybrid applications and PWAs are designed for mobile devices and offer a user-friendly interface for smartphones and tablets
- They can work offline to some extent although offline support varies
- They can access device features such GPS and push notifications.
- All three approaches allow users to install the app on their home screen

1.3 Differences

1.3.1 Architectural differences:

FEATURE	NATIVE APPS	PWAS	HYBRID APPS
Technology	Platform specific languages(Swift, Kotlin)	Web based(HTML, CSS, JS)	Web-based with native wrapper
Platform compatibility	Platform Specific	Cross Platform	Cross Platform
Access to native APIs	Full Access	Limited	Partial access via bridges
Installation	Downloaded from app stores	No app store required, accessed via a url	Downloaded from app stores

Table 1: Architectural differences of different types of mobile apps

1.3.2 Performance:

Performance is critical for ensuring smooth interactions and fast response times.

PWAs : Can achieve near-native performance through service workers and caching but may lag in graphics-intensive applications.

Native Apps : Offer the highest performance as they are compiled specifically for the platform, enabling seamless animations and interactions.

Hybrid Apps : Performance depends on the web view rendering; may struggle with complex UI or animations, making them slower than native apps.

1.3.3 User experience:

The user experience determines how well an app interacts with users and meets their expectations. User experience varies across the various types of applications.

PWAs : Provide a consistent experience across platforms, load quickly, and support push notifications but may lack advanced gestures and animations.

Native Apps : Deliver the best UX with fully optimized UI elements, platform-specific interactions, and smooth transitions.

Hybrid Apps : Offer a UX closer to native but may experience lags due to reliance on web technologies within a native container.

1.3.4 Development Considerations:

Developing each type of app comes with its own set of adavantages and disadvantages from a developer's perspective:

PWAs : Require standard web development skills and have the fastest development cycle. They do not have access to all native device capabilities out-of the box, so for advanced features, additional plugins or libraries may be needed.

Native Apps : Require expertise in platform-specific languages, making them more resource-intensive to develop. There is also a steeper learning curve compared to web development

Hybrid Apps : Strike a balance by enabling cross-platform development with some native features, reducing costs and development time.

1.3.5 Accessiblity:

This refers to how easily people, including those with disabilities can use an app. Ensuring accessibility is vital for a diverse user base.

PWAs : Leverage web standards (WAI-ARIA) for accessibility across browsers but may face limitations on certain devices.

Native Apps : Provide the best accessibility, with built-in APIs for screen readers, captions, and voice control.

Hybrid Apps : Can be accessible when built properly, but accessibility depends on how well the web view integrates with native accessibility features.

1.3.6 Offline Usage:

Offline functionality is crucial for usability in low or no-network conditions.

PWAs : Use service workers for caching, enabling offline access. However, full offline capabilities are limited by browser constraints.

Native Apps : Fully support offline usage, storing data locally and enabling seamless offline workflows.

Hybrid Apps : Support offline mode through local storage but may face limitations due to reliance on web views.

1.3.7 Maintenance:

Maintenance and updates affect long-term app-sustainability:

PWA's require very little maintenance as the app codebase lives on a server. As long as the server is maintained, the PWAs will be updated automatically for all users.

Native apps require releasing new app versions when new features and fixes are implemented. Thus maintenance is more complex than in PWAs. Automated CI/CD pipelines and testing are used to streamline these processes.

Hybrid apps are the most complex to maintain as they require app store deployments like native apps and the web codebase that renders within the native wrapper. Any changes in the application codebase requires repackaging and redeploying the app.

1.3.8 Security:

PWAs : Depend on HTTPS, Content Security Policy, and authentication mechanisms but lack native security features.

Native Apps : Offer the highest security through OS-level encryption, secure storage, and app store validation.

Hybrid Apps : Benefit from app store vetting but are limited by the security constraints of web-based code

Chapter 2: Mobile App Programming Languages

Different programming languages are used for mobile app development depending on the platform, performance needs, and development approach. Below is a detailed comparison of the major mobile app programming languages.

2.1. Native Mobile App Languages

These languages are platform-specific and provide the best performance, access to hardware features, and native UI.

Language	Platform	Pros	Cons
Swift	iOS (Apple devices)	High performance, optimized for iOS, easy to read and maintain, modern syntax	Limited to Apple ecosystem
Objective-C	iOS (legacy)	Mature and stable, long-standing support in Apple ecosystem	Verbose, outdated syntax, harder to maintain
Kotlin	Android	Official Android language, concise, null safety, interoperable with Java	Newer than Java, so fewer resources
Java	Android	Mature, large community, highly portable	Verbose syntax, can be slower compared to Kotlin

Table 2: Overview of Some Native Languages

2.2. Cross-Platform Mobile App Languages

These languages allow developers to write a single codebase that runs on both iOS and Android.

Language	Used In	Pros	Cons
Dart	Flutter	Fast performance, great UI with customizable widgets, growing popularity	Less community support compared to JavaScript
JavaScript/ TypeScript	React Native, Ionic	Large developer community, reusable codebase, quick development	Performance issues for complex apps, limited access to native features
C#	Xamarin	Strong Microsoft ecosystem, allows sharing code across platforms	Requires Microsoft tools, heavier apps

Table 3: Overview of Some Cross Platform Languages

2.3. Web-Based Mobile App Languages (for PWAs and Hybrid Apps)

These languages are mainly used for Progressive Web Apps (PWAs) and hybrid apps.

Language	Used In	Pros	Cons
HTML, CSS, JavaScript	PWAs, Ionic, Cordova	Easy to learn, fast development, no need for App Store deployment	Limited access to device features, slower performance

Table 4: Overview of Some Web Based Languages

Summary of Comparisons

- Best for iOS: Swift (best performance and Apple support).

- Best for Android: Kotlin (modern, optimized for Android).

- Best for Cross-Platform: Dart (Flutter) or JavaScript (React Native).

- Best for Web-Based Apps: JavaScript with HTML & CSS.

Chapter 3: Mobile Development Frameworks

A mobile application development framework is a set of tools, libraries, and APIs that simplify and accelerate the process of creating mobile applications for various platforms like iOS and Android, providing a structured environment and pre-built components.

How are they different from programming languages?

These frameworks are built on-top (using) programming languages and offer a standardized way to build and deploy apps. This allows developers to focus on implementing business logic and user experience rather than foundational code reducing development time and improving efficiency.

3.1 Types of Mobile Development Frameworks

1. **Native Frameworks:**

   Native frameworks are designed for building apps exclusively for a single platform, such as iOS or Android.

2. **Cross-Platform Frameworks:**

   These frameworks enable developers to create applications that can run on multiple platforms (iOS, Android, etc.) using a single codebase. They save time and resources by allowing code reuse across different platforms, although they may sometimes sacrifice performance compared to native apps.

3. **Hybrid Frameworks:**

   Hybrid frameworks combine elements of both native and web applications. They are built using web technologies like HTML, CSS, and JavaScript and are encapsulated within a native container.

4. **Progressive Web Apps (PWAs):**

   They are web applications that provide a native app-like experience on mobile devices.

3.2 A Review of Some Frameworks

3.2.1 Native Framework - SwiftUI

SwiftUI is a declarative framework for building user interfaces for iOS, iPadOS, watchOS, tvOS, visionOS and macOS, developed by Apple Inc. for the Swift programming language.

Language Characteristics

- **Declarative Syntax:** You *declare* what the UI should look like, not how to build it step by step.
- **Composable Views:** Everything is a View, and views are composed of smaller views. Encourages modular, reusable UI components.
- **Built-in DSL (Domain Specific Language):** SwiftUI's syntax feels like a mini-language within Swift. It abstracts a lot of UIKit's complexity into readable, expressive code.

Performance

Strengths:

- **Memory Management:** Uses structs (value types), so no reference counting needed, Immutable by default, lightweight, and cheap to copy.Swift enforces memory safety, reducing memory leaks.
- **Efficient Re-Renders:** Uses property wrappers like @State, @Binding, and @ObservedObject to track state. Only views affected by state changes are updated. A built-in diffing algorithm ensures minimal UI updates.

Weakness:

- **Runtime Performance Issues:** Deeply nested or large view hierarchies can suffer if state is mismanaged.Overuse of @State may trigger excessive re-renders. SwiftUI’s abstraction can sometimes cause unnecessary view rebuilding, slowing performance.

Cost & Time to Market

Strengths:

- **Fast UI Development:** Declarative syntax + Xcode Previews = faster prototyping and iteration.
- **Adaptive UI with Minimal Effort:** Automatically supports Dark Mode, Dynamic Type, and Accessibility features. Less custom code needed for system-level settings = consistent native experience.

Weaknesses:

- **No Cross-Platform Support:** Only works on Apple platforms. Android requires a separate codebase and team, increasing cost and complexity.
- **Edge Case Limitations:** Complex UIs may require UIKit workarounds. Can slow down development if hitting SwiftUI's current limitations.

UX & UI

Strengths:

- **Native Look & Feel:** Uses Apple's design system by default, so apps look and feel native with minimal styling effort. Users get a familiar experience out of the box.
- **Responsive & Adaptive:** Automatically adjusts layouts for different devices and orientations. Builds inclusive, multi-platform UI with little conditional logic.

Weaknesses:

- **Limited Fine-Grained Control:** No direct access to low-level tools like Auto Layout or CALayer. Complex interactions may need more code or workarounds.
- **Some Features Still Maturing:** Missing or inconsistent controls/modifiers across platforms or iOS versions. Occasional UX bugs that rely on future updates to resolve.

Complexity

- **Starts Simple**: SwiftUI is beginner-friendly with clear, concise syntax. You can build basic UIs quickly and easily.
- **Smart State Management**: Built-in tools like @State, @Binding, and @EnvironmentObject handle UI updates automatically, reducing boilerplate and bugs.

Where It Gets Complex:

- **State Management at Scale**: Multiple state types can interact in unexpected ways. Managing state across large apps can lead to bugs and confusing update flows.
- **Custom Layouts & Interactions**: Going beyond default components (custom gestures, animations, layouts) introduces more complexity and often requires advanced tools or UIKit integration.
- **Tooling & Debugging**: SwiftUI's debugging and developer tools are still maturing, making troubleshooting harder in complex projects.

Community Support

- **Strong Apple Support:** Apple consistently updates SwiftUI with major improvements (e.g. new layout tools, performance boosts), showing long-term commitment.
- **Expanding Open-Source Ecosystem:** Many open-source libraries now support SwiftUI with custom components, animations, forms, and navigation tools.

Current Gaps:

- **Limited UI Libraries:** Compared to UIKit, SwiftUI lacks mature, ready-made third-party UI kits.

3.2.2 Cross Platform - Flutter

Flutter, developed by Google, is a UI toolkit to build native applications for mobile apps, desktop & web platforms. It is a cross-platform mobile app development framework that works on one code base using Dart to develop Android as well as iOS applications.

The framework provides a large range of fully customizable widgets that helps to build native applications in a shorter span. Moreover, Flutter uses the 2D rendering engine called Skia for developing visuals and its layered architecture ensures the effective functioning of components.

Language Characteristics

- **Declarative Syntax:** Flutter uses a declarative style where the UI is rebuilt from scratch when the state changes. You describe what the UI looks like at any given state, and Flutter handles the rendering.
- **Composable Widgets:** Everything in Flutter is a **Widget**. Widgets describe how a part of the UI should look. Widgets can be composed of other widgets, allowing reusable, modular design.
- **Custom DSL via Dart Syntax:** While not a dedicated DSL, Dart's flexibility enables widget trees that act like a declarative mini-language. Layouts are deeply nested but can be structured with custom widgets and builder patterns.

Performance

Strengths:

- **High Performance Rendering:** Flutter uses its own high-performance rendering engine (Skia) for drawing UI, bypassing native platform UI components.This gives consistent 60fps+ performance across platforms.
- **Hot Reload:** Real-time code changes without restarting the app. Extremely fast for testing UI tweaks and logic changes.

Weaknesses:

- **Large Binary Size:** Apps have a larger footprint due to bundling the rendering engine.
- **Platform-Specific Performance Issues:** Although Flutter is optimized, certain platform-specific behaviors (keyboard input, scroll physics) may require manual tuning.
- **Custom Rendering Engine Limitations:** Since Flutter bypasses native controls, accessibility and native integration can sometimes lag behind.

Cost & Time to Market

Strengths:

- **Single Codebase for All Platforms:** One Dart codebase runs on Android, iOS, Web, Desktop, and more. Shared business logic, UI, and testing = less time and fewer developers.
- **Hot Reload & Previews:** Very fast iteration and prototyping with minimal downtime.

Weaknesses:

- **Learning Curve for Dart:** Dart is relatively new to many developers and not widely used outside of Flutter.
- **Occasional Platform-Specific Workarounds:** Writing platform-specific code for features like Bluetooth, GPS, or Camera may still require native platform knowledge.

UX & UI

Strengths:

- **Pixel-Perfect Custom UIs:** Full control over every pixel — easily build beautiful, custom UIs that look identical across platforms.
- **Rich Widgets & Animation Support:** Extensive widget library and seamless animations out-of-the-box. Lottie, Hero transitions, and physics-based animations are built-in.

Weaknesses:

- **Non-Native Feel by Default:** Because it doesn't use native controls, some apps may "feel" different unless carefully styled.
- **Accessibility Can Be Tricky:** While improving, Flutter's accessibility tools aren't always as mature as native solutions.
- **Platform Conventions Must Be Manually Followed:** Must manually implement things like swipe-to-go-back, tab gestures, or navigation patterns if differing by platform.

Complexity

Starts Simple:

- **Intuitive Widget-Based System:** Clear UI structure using widgets and layout trees. Easy to pick up for designers and junior developers.
- **Unified Styling & Theming:** Material and Cupertino theming is centralized and easy to manage.

Where It Gets Complex:

- **Deep Widget Trees:** UI hierarchies can get deeply nested and hard to read/maintain.Requires refactoring into custom widgets early on.
- **State Management Choices:** Many options: setState, Provider, Riverpod, Bloc, GetX, etc. Choosing and maintaining the right state solution is crucial for scaling apps.
- **Platform-Specific Code:** You may still need Kotlin/Swift/Java/Obj-C for plugins or integrations, which adds complexity.

Community Support

Strengths:

- **Strong Google Backing:** Actively developed and maintained by Google with frequent updates and a strong vision.
- **Large Plugin Ecosystem:** Pub.dev has thousands of packages for everything from Firebase to custom charts.

Current Gaps:

- **Dart Ecosystem Still Growing:** Outside of Flutter, Dart is less popular, which means fewer developers and packages compared to JS or Swift.

3.2.3 Hybrid Framework - Ionic

Ionic is an open-source UI toolkit for building cross-platform applications (iOS, Android, Web, and Desktop) using web technologies like HTML, CSS, and JavaScript/TypeScript. It is built on top of Capacitor (modern native runtime) and integrates with frameworks like Angular, React, and Vue.

Language Characteristics

- **Declarative Syntax:** Depends on the chosen front-end framework (e.g., Angular, React, Vue). UIs are typically built declaratively through component-based syntax, especially with React and Vue.
- **Composable Components:** Ionic provides a rich set of UI components (buttons, modals, tabs, etc.) that can be composed like building blocks. Encourages reusable, modular components through standard web component architecture.

- **Web-Based DSL (HTML/CSS/JS):** Uses familiar web tech; no new language to learn if you're coming from web dev. Ionic components are just HTML custom elements styled with CSS and controlled via JS/TS.

Performance

Strengths:

- **Near-Native Performance for Many Use Cases:** Modern devices and web views (especially WKWebView on iOS) offer strong performance.Great for apps that are UI-centric and not performance-intensive.
- **Capacitor Native Plugin Support:** Integrates with native functionality via Capacitor, allowing direct access to device features like Camera, GPS, Filesystem, etc.
- **Web Optimizations:** Leverages lazy loading, tree-shaking, and service workers to improve performance on the web.

Weaknesses:

- **Performance Overhead from WebView:** Apps run inside a WebView (like a browser inside a native shell), so performance isn't on par with true native apps in demanding use cases.
- **Heavy DOM Manipulations Can Lag:** Complex animations, heavy lists, or game-like interactions may result in jank or slower frame rates.
- **Limited Canvas & Graphics Power:** Not ideal for intensive rendering tasks like AR, real-time gaming, or rich 3D animations.

Cost & Time to Market

Strengths:

- **Code Once, Run Everywhere:** Shared codebase for web, Android, and iOS, significantly reducing time and development cost.
- **Familiar Tools & Stack:** Web developers can build mobile apps without learning a new language or paradigm.
- **Capacitor Plugin Ecosystem:** Simplifies native functionality access without needing deep platform-specific code.

Weaknesses:

- **Native Code May Still Be Needed:** For complex features, you may need to write or modify native iOS/Android code.
- **Platform-Specific Styling:** Ionic components are styled to feel native, but sometimes manual tweaking is needed to match exact platform guidelines.
- **Capacitor Plugin Maturity:** Some plugins may be community-driven and lack polish or complete platform parity.

UX & UI

Strengths:

- **Prebuilt Cross-Platform Components:** Ionic provides styled components that mimic native iOS and Android designs using Material Design and Cupertino styles.
- **Responsive Design with CSS:** Easy to build layouts that adapt to screen size and orientation using standard CSS.
- **Web Standards-Based:** Full use of web accessibility APIs, media queries, and browser tools.

Weaknesses:

- **WebView Limitations:** Some native behaviors (like keyboard handling or gesture responsiveness) aren't as smooth as in native frameworks.
- **Non-Native Feel in Some Areas:** Subtle differences in animation timing or system gestures may cause apps to "feel" non-native.
- **Manual Responsiveness Effort:** Responsive layouts still require developer implementation with media queries or grid systems.

Complexity

Starts Simple:

- **Low Barrier to Entry:** If you know HTML/CSS/JS (and maybe Angular/React/Vue), you can get started quickly.
- **Component-Based Architecture:** Clean structure, encourages reusable UI components and modular design.

Where It Gets Complex:

- **State Management Depends on Framework:** Ionic itself doesn't manage state. You must rely on your chosen framework (Redux, Vuex, RxJS, etc.), which can introduce complexity.
- **Native Integration Requires Platform Knowledge:** You may hit roadblocks when plugins are missing or need custom native code in Swift/Kotlin/Java.
- **Plugin Compatibility:** Not all Capacitor plugins work flawlessly across platforms; testing and debugging can be time-consuming.

Community Support

Strengths:

- **Large Web Dev Community:** Since Ionic is built on web standards, it benefits from a huge global pool of web developers and open-source contributions.
- **Strong Documentation & Resources:** Ionic and Capacitor have detailed docs, official tutorials, and starter templates.
- **Capacitor Plugins & Open Source Libraries:** Growing ecosystem of plugins and libraries for native features, PWA support, authentication, etc.

Current Gaps:

- **Smaller Native Mobile Community:** Compared to Flutter or Swift/Android, fewer native developers contribute to the ecosystem.
- **Debugging WebView-Based Apps:** Debugging hybrid apps can be harder when dealing with performance, native bridge errors, or inconsistent behavior across platforms.

3.2.4 Progressive Web Apps

PWAs are web applications that use modern web capabilities to deliver an app-like experience to users. They run in the browser but can be installed on devices, work offline, send push notifications, and access some native device features—all using standard web technologies (HTML, CSS, JS/TS).

Language Characteristics

- **Framework-Agnostic:** PWAs can be built using any front-end framework (React, Angular, Vue, Svelte, or even Vanilla JS).

- **Modern Web Standards:** Typically leverage declarative, component-based UI (e.g., JSX in React or SFCs in Vue).

Performance

Strengths:

- **Highly Optimized for Web:** Uses HTTP/2, service workers, and lazy loading to improve load time and responsiveness.
- **Instant Loading with Caching:** Service workers cache resources efficiently for fast repeat visits and offline support.
- **Runs on Any Device with a Browser:** No install needed; can be added to the home screen without app stores.

Weaknesses:

- **No Full Native Performance:** Still runs inside a browser context; heavy animations or complex computations may lag compared to native.
- **Limited Access to Device APIs:** Only supports a subset of native features (e.g., no Bluetooth, limited camera access).
- **Battery & Memory Usage:** Browsers impose limits on background processes and memory use, especially on mobile.

Cost & Time to Market

Strengths:

- **Single Codebase Across All Platforms:** One web app can target desktop, mobile web, and even be installed like an app on phones.
- **No App Store Submission Needed:** Skip the app store approval process; deploy instantly via web.
- **Low Dev Overhead:** Web devs don't need to learn native languages or platform-specific tooling.

Weaknesses:

- **App Store Discoverability Lost:** Not listed in iOS App Store or Google Play by default (unless wrapped in TWA or similar).

- **Still Needs Responsive Design:** Building for multiple screen sizes and interactions still requires planning and effort.
- **Offline Support Takes Work:** Service workers need to be carefully implemented and tested for consistent offline UX.

UX & UI

Strengths:

- **Responsive Design with Web Tech:** Uses CSS media queries, Flexbox, and Grid for adaptable UI across screen sizes.
- **Installable with App-Like Feel:** PWAs can run full-screen, have icons, and launch from home screens.
- **Access to Push Notifications & Offline Capabilities:** Enhances engagement and usability even when connectivity is poor.

Weaknesses:

- **Lacks Full Native Feel:** Scroll inertia, gestures, and animations may feel “browser-y” compared to native apps.
- **Limited OS Integration:** Can't access things like the phone dialer UI, advanced haptics, or biometric authentication.
- **iOS Support Lag:** Apple’s Safari has more limited PWA support than Chrome or Edge (e.g., no push notifications until recently).

Complexity

Starts Simple:

- **Quick to Build:**
  Start with a standard web app and gradually enhance with PWA features.
- **Low Barrier to Entry:** Anyone familiar with basic HTML/CSS/JS can build a functional PWA.

Where It Gets Complex:

- **Service Worker Logic:** Caching strategies and background sync can be tricky to implement and debug.

- **Security Requirements:** Must be served over HTTPS, and handling certificate renewals, secure headers, etc., adds devops overhead.
- **Device Feature Limitations:** Need to design around limited access to native APIs or find polyfills/alternatives.

Community Support

Strengths:

- **Massive Web Ecosystem:** Leverages the entire open web developer community and resources.
- **Strong Support from Google & Microsoft:** Both actively develop tools and best practices for PWA development.
- **Well-Documented APIs:** Extensive MDN documentation and framework integrations for PWA support.

Current Gaps:

- **iOS Limitations:** Apple's slower adoption of PWA features restricts parity across platforms.
- **Still Some Confusion Around PWA "Standard":** Varying implementations lead to inconsistent experiences.
- **Browser-Based Debugging:** While dev tools are powerful, bugs tied to service workers, caching, or installability can be tricky.

3.3 Comparative Review: SwiftUI vs Flutter vs Ionic vs PWA

Criteria	**SwiftUI (Native)**	**Flutter (Cross-platform)**	**Ionic (Hybrid)**	**PWA**
Language Characteristics	Declarative Swift-based DSL; native to Apple platforms	Dart language; declarative widget-based system	Web standards (HTML/CSS/JS) + Angular/React/Vue	Pure web tech (HTML/CSS/JS), no framework lock-in
Performance	High/native performance;	Near-native via Skia engine;	Good for most UI apps; relies	Web-dependent; limited

Criteria	SwiftUI (Native)	Flutter (Cross-platform)	Ionic (Hybrid)	PWA
	uses Swift & system-level rendering	great 60fps UI	on WebView	hardware/API access
Cost & Time to Market	Fast for Apple-only apps; deep integration	Very fast for cross-platform apps	Fast for web teams; strong code reuse	Instant delivery; no app store delay
UX & UI	Native Apple look & feel; adaptive UI	Custom, pixel-perfect UIs; platform-neutral by default	Native-like with Ionic components	Web-styled UI; can look “lightweight” or generic
Complexity	Simple for small apps; complex state handling in large ones	Simple to start; managing state can get tricky	Familiar for web devs; native bridge adds complexity	Easiest dev flow; complexity comes with native-like behavior
Community Support	Growing; Apple-first devs	Huge, vibrant dev community; Google-backed	Strong web dev support; mixed plugin quality	Massive web community; fewer native app examples

Table 5: Comparison of SwiftUI vs Flutter vs Ionic vs PWA

3.4 Best Use Cases for Each Framework

3.4.1 SwiftUI – Best for Apple-Only Native Apps

Use When:

- You’re targeting only iOS/macOS/watchOS/tvOS.
- You need deep native integrations (e.g. HealthKit, ARKit).

- You want the **best performance**, native accessibility, and platform conventions.

Avoid If:

- You need cross-platform support.
- You're doing heavy animations or large state management without UIKit fallbacks (UIKit is an older Swift framework)

3.4.2 Flutter – Best for Custom Cross-Platform UIs

Use When:

- You want one codebase for iOS, Android, Web, Desktop.
- You want full custom UI/UX control.
- You need fast performance (games, animation-heavy apps).

Avoid If:

- You want to strictly follow native platform styles.
- Your team lacks Dart expertise.

3.4.3 Ionic – Best for Web-Focused Teams

Use When:

- Your team already knows web dev (JS/TS + Angular/React/Vue).
- You want to build one app for web + mobile.
- You need access to native features via plugins (camera, GPS, etc.).

Avoid If:

- You need very high-performance graphics.
- You're building apps that require native look/feel without compromise.

3.4.4 PWA – Best for Web-First Experiences

Use When:

- You want to deploy instantly, without app stores.
- Your app doesn't require deep native integration.

- You need offline support, caching, or push notifications in browsers.

Avoid If:

- You need native APIs not available via browsers.
- You want to distribute on app stores or need strong monetization.

Chapter 4: Mobile App Architecture & Design Patterns

4.1 Introduction

What is Mobile App Architecture?

App architecture refers to a set of structural components that frame the system and its inner interactions, the specifics being defined by the application's features and requirements. How smoothly and reliable an app runs depends significantly on the quality of its architecture.

Why is App Architecture Essential?

Quality architecture helps with risk management and enables costs reduction. Hence, an application with robust and well-planned architecture is more likely to succeed in its target market.

Mobile applications usually start with the planning and designing phase, hence and insufficient approach to this step can slow down the development process and make it more extensive. This can also lead to various performance issues and system failures.

Poor mobile app architecture may also lead to:

- Difficulties with development and maintenance.
- Source code testing complications.
- Lower code readability.
- Greater exposure to errors.

4.2 The Fundamental Layers of Mobile App Architecture

The multi-layer approach is widely used in mobile app development as it separates the different application-specific operational layers. This helps developers to solve complicated problems quickly without changing the entire application.

The number of layers varies according to an application's business and functional requirements. The most used architecture pattern is the three-tier structure.

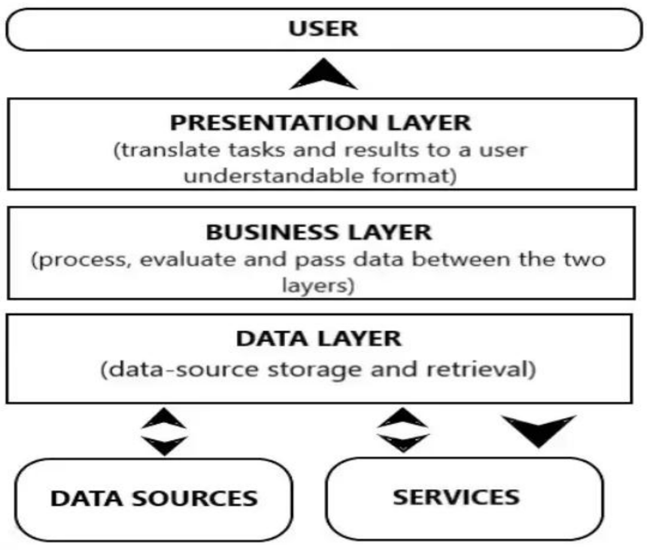

USER
PRESENTATION LAYER
(translate tasks and results to a user understandable format)
BUSINESS LAYER
(process, evaluate and pass data between the two layers)
DATA LAYER
(data-source storage and retrieval)
DATA SOURCES
SERVICES

Image1: Layers of the three-tier architecture

Data Layer

The data layer is the heart of the application, comprised of data access components, server communications, and logic. This layer executes connections to the network and enables the storage of accrued information either locally or on the cloud. The patterns chosen for core operations determine the application's overall performance. The key priority at this level is security and safety of data maintenance.

Business Layer

In this layer, all domain processes and operations take place as the business layer explains the logic, which in turn drives an app's core functionalities. Business logic can take many forms, but it is mainly responsible for business components and the corresponding data flow. Additionally, the business layer includes navigation methods, logging, catching, and other technical processes.

Presentation Layer

The presentation layer focuses on how an app presents itself to the user and includes user interface (UI) elements such as themes, fonts, and colours. Prioritisation of features and functionalities also takes place at the presentation layer, because end-users should be able to easily navigate the app.

4.2.1 Types of Design Patterns in Mobile Development

- **Creational**: Describe how to create or instantiate objects, and the most used ones are Builder and Singleton.
- **Structural**: Describe how objects are composed and combined to form larger structures (Model-View-Controller, and Model-View-ViewModel)
- **Behavioral**: Describe how objects communicate with each other (Delegation, Strategy, and Observer)

4.2.2 Top Architecture Design Pattern: MVC, MVP, and MVVM

The different architecture design patterns are widely used to moderate complex codes and simplify UI code by making it neater and more manageable. Model-View (MV(X)) architectures divide the visualising, processing, and data management functions for UI applications, which increases an app's modularity, flexibility, and testability.

Model-View-Controller (MVC)

MVC is commonly used when designing simple applications. This is because it is more readily modified compared to others and makes implementing changes to the app simple.

MVC consists of three components: Model, View, and Controller. The "Model" is subject to the app's business logic and handles data changes and manipulations. The "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.

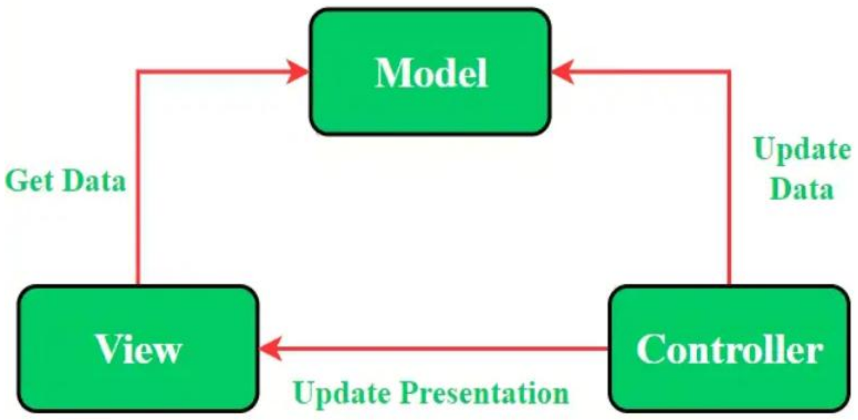


Image2: Model-View-Controller

Model View Presenter (MVP)

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

Like 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.

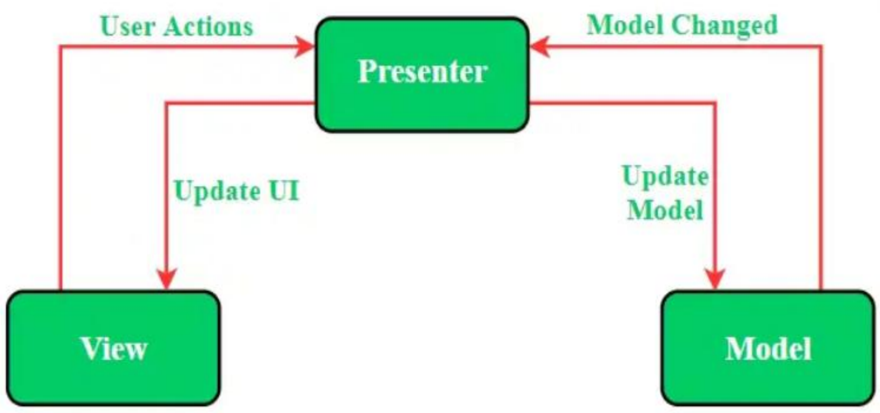


Image 3: Model View Presenter

Model-View-ViewModel (MVVM)

MVVM is designed for more explicit separation of UI development from business logic, much like MVC. The "model" here handles basic data and "view" displays the processed data. The ViewModel element discloses methods and commands that help maintain the state of view and control the model.

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. Overall, 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.

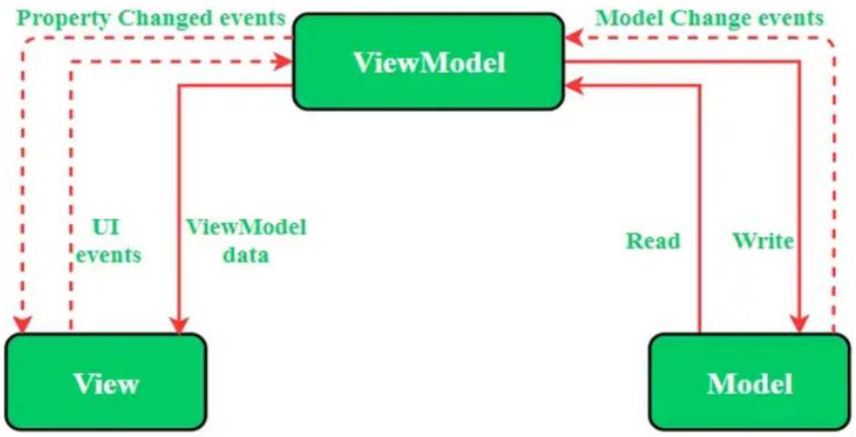


Image 4: Model View View Model

There are other architecture design patterns, some of which include:

- VIPER (View, Interactor, Presenter, Entity, and Router) Architecture.
- Singleton Method Design Pattern.
- Factory Method Design Pattern.
- Observer Method Design Pattern.
- Dependency Injection (DI) Method Design Pattern.
- Adapter Method Design Pattern.

4.3 Key Principles of a Good App Architecture

Suppose you have designed your app architecture per the business and user requirements, how do you verify that the app has been composed correctly and will function as intended? A well-designed architecture should meet a set of specific conditions including:

- **Efficiency:** The application performs the tasks and functions under any condition and handles heavy loads effectively.
- **Portability:** Since frequent environmental (market or technical-specific) changes are natural in mobile applications, good architecture must allow adjustments without crashing the whole system.
- **Scalability:** Post-release updates and further development are a key part of the software development life cycle. Solid architecture results in faster changes and updates because it's developed in separate threads.

- **Testability:** If an app's architecture is easily tested it reduces the number of errors and increases reliability.

4.4 How To Choose the Right Architecture

To choose the effective architecture for a mobile app, the following steps should be followed:

- Analyse the current state of the codebase to define any issues that need to be resolved or code that needs improvement.
- Evaluate various architecture patterns and try a few on for size before settling for one.
- After the decision is made, examine how effectively the chosen architecture solves the key problems.

Chapter 5: Requirement Engineering

5.1 Introduction

Requirements Engineering

A systematic and strict approach to the definition, creation and verification of requirements for a software system is known as requirements engineering. It focuses on identifying, tracing and characterising the needs at each stage of a system design, to ensure their implementation and impact are clearly understood.

5.2 Importance of Requirements Engineering to Mobile App Development

1. Defines the Scope of a Project:

Requirements are essential for scoping out a project. Software requirements help determine what features will be in the final product, how long it will take to develop those features, and how much it will cost.

2. Identify Potential Risks :

Requirements also help identify risks early in the development process and depending on the methodology you’re using, saving considerable time and money later.

3. Give Direction to Developers :

Requirements also provide a roadmap for the development process. They can help developers understand how the different pieces of the software fit together and what needs they should deliver first

4. Requirement Safeguards the End-User Experience :

Software requirements help to safeguard the end-user experience in software development. Having a complete and accurate understanding of what users need, want, and expect from a software application, developers can create products that are more likely to meet user needs.

5. Fosters Communication and Collaboration Between Team Members:

Software and app development is a technical process that involves many different people with various skill sets. Requirements act as a common language between team members, improving communication and collaboration.

6. Avoid Costly Re-Work and Last Minute Surprises :

If you don't gather requirements upfront, you're likely to end up with a product that doesn't meet the needs of your users. This can result in costly rework down the line, as well as last-minute surprises that throw off your development timeline.

5.3 Key Challenges and Trends to Requirements Engineering

1. Challenge 1 - Uncertainty and Change:

Uncertainty arises from incomplete, ambiguous, or conflicting information, as well as unpredictable factors affecting the system. Change results from evolving stakeholder needs, market conditions, regulations, or new technologies. Both uncertainty and change impact software development by causing rework, delays, and errors. Managing them effectively is crucial for project success.

2. Challenge 2 - Complexity and Scale:

Complexity and scale arise from diverse stakeholders, system functionality, architecture, and context. They make it challenging to understand, communicate, and verify requirements while ensuring consistency and feasibility.

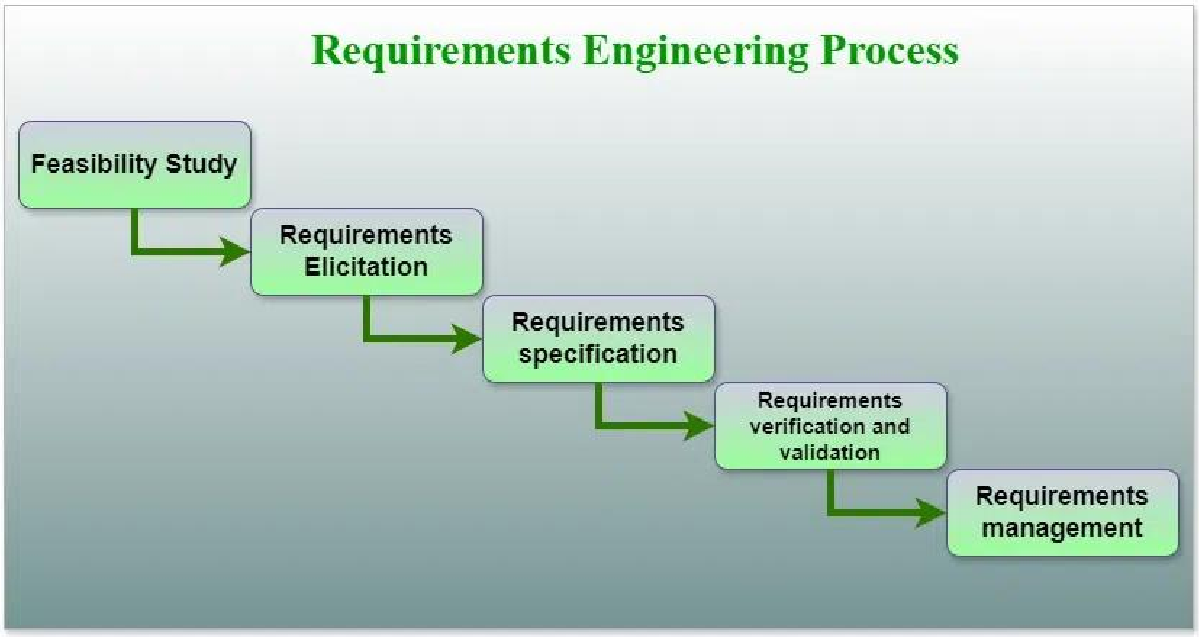
3. Trend 1 - User-Centered and Value Oriented :

One of the current trends in RE is the shift from a system-centered and function-oriented perspective to a user-centered and value-oriented perspective. This means that the focus of RE is not only on what the system does, but also on who uses the system, why they use it, how they use it, and what value they derive from it.

4. Trend 2 - AI-Enabled and Data-Driven :

Another current trend in RE is the emergence of artificial intelligence (AI) and data as enablers and drivers of the software systems and the requirements. This means that the software systems can leverage AI and data to provide intelligent, adaptive, and personalized functionality and quality to the users.

5.4 Requirement Engineering Process


Requirements Engineering Process
Feasibility Study
Requirements Elicitation
Requirements specification
Requirements verification and validation
Requirements management

Overview of Requirement Engineering Process

5.4.1 Feasibility Study

The different feasibility study areas include:

1. Technical Feasibility:

Technical feasibility assesses available hardware, software, and required technologies for project development. It evaluates resource suitability, team capabilities, and the feasibility of maintenance and upgrades.

2. Operational Feasibility:

Operational feasibility evaluates how well the system meets requirements, its ease of use, and maintenance. It also assesses usability and whether the proposed solution is acceptable.

3. Economic Feasibility:

Economic feasibility analyzes the project's costs, including development, resources, and operations. It assesses whether the financial benefits outweigh the expenses for the organization.

4. Legal Feasibility:

Legal feasibility ensures the project complies with laws, regulations, and standards. It identifies legal constraints, reviews contracts, and assesses intellectual property concerns

5. Schedule Feasibility :

Schedule feasibility evaluates if the project timeline is realistic and achievable. It identifies milestones, deadlines, and resource availability to ensure timely completion. Time constraints are also considered to support successful project execution.'

5.4.2 Requirements Elicitation:

Requirements elicitation is the process of gathering information about the needs and expectations of stakeholders for a software system.
Techniques involved in requirements elicitation include:

- Interviews
- Surveys
- Focus Groups
- Observation
- Prototyping

5.4.3 Requirements Specification :

Requirements specification is the process of documenting the requirements identified in the analysis step in a clear, consistent, and unambiguous manner. The models used for Requirement specification include ER diagrams, Data Flow Diagrams (DFDs), Functions Decomposition Diagram (FDDs), Data Dictionaries, etc.

Requirements specified include :
Functional Requirements, Non-Functional Requirements, Constraints, Acceptance Criteria.

5.4.4 Requirements verification and validation (V&V) :

This is the process of checking that the requirements for a software systems are complete, consistent and accurate and that they meet the needs and expectations of the stakeholders.

5.4.5 Requirements Management :

Requirements management is the process of managing the requirements throughout the software development life cycle, including tracking and controlling changes, and ensuring that the requirements are still valid and relevant. Key activities involved in requirements management include:

- Tracking and controlling changes
- Version control
- Traceability
- Communication
- Monitoring and Reporting.

5.5 Tools & Techniques for Requirement Engineering

Somes Tools Involved in Requirement Engineering Include :

- Observation report
- Questionnaire (survey, poll)
- User cases
- User stories
- Requirement workshop
- Mind mapping
- Role Playing
- Prototyping

Some Software Tools used include:

- Jira,
- Trello,
- Figma,
- Balsamiq,
- Google Forms

Chapter 6: Steps to estimate cost for a mobile app development:

The exact project estimate for a mobile app depends on several factors. The following elements have a significant influence on overall cost of app development.

1. Platform Support;

There are billions of smartphone users globally and it is estimated that 69.9% of them are android users while 30.1% are iOS users. If native development is to be chosen due to the excellent performance of native apps, two separate code bases are required which significantly increases development cost. To compromise a hybrid application solution can be chosen but the app would deliver a less impressive performance. Thus target users and distribution strategy need to be considered when deciding on platforms in order to minimize cost and satisfy the users.

2. App Design Complexity:

When designing an application, the target audience needs to be kept in mind such that the choice between minimalistic or elaborate designs is done. More intricate designs with complex animations increase the development time for the UI/UX designer thus increasing the development cost. Apps featuring generic templates on the other hand can be built much faster thus decreasing the development cost.

3. Development Hours estimate:

After getting a clear understanding of App's requirements and design, the hours required to carry out each task need to be estimated. The project can be broken down into smaller components and hours allocated based on complexity. This can be done using industry bench marks or by seeking help from experienced developers. The hours spent developing the project will help significantly in estimating the overall project cost.

4. Development Team experience:

An experienced and diversely skilled app development team often costs more but delivers higher quality applications whereas a team of gloabal capability can help balance speed and affordability.

5. Testing requirements:

Efficient testing is a rigorous process where the app performance is validated across several platforms and varying devices. This is a significant requirement in estimating development cost.

6. Additional expenses:

In addition to development hours and all the other required elements, additional expenses such as cost of API keys, and publishing fees for the various platforms like the App store and Google Play store which are $99 per year and $25 per upload respectively are factored in. The budget for regular updates is also factored in per year.

Conclusion:

Mobile app development is a complex but highly rewarding process that involves multiple layers of decision-making—from selecting the right type of application (native, hybrid, or PWA), to choosing programming languages, frameworks, and architectural patterns. Each option comes with its own set of trade-offs in performance, user experience, development cost, scalability, and maintainability.

Understanding the differences and similarities among app types, languages, and frameworks is crucial in aligning technical choices with business goals. While native apps offer the best performance and user experience, they come with higher costs and longer development times. Cross-platform and hybrid frameworks offer quicker development cycles and lower budgets at the expense of some native advantages. Meanwhile, PWAs provide a web-first, installable experience that bypasses app store barriers—ideal for certain lightweight or content-focused applications.

Good architectural decisions and requirement engineering practices further ensure the app is scalable, maintainable, and aligned with end-user needs. Estimating development costs requires careful planning, taking into account platform choices, design complexity, developer expertise, and testing needs. By following a structured and informed approach, developers and businesses can deliver high-quality mobile applications that meet technical standards and user expectations.

References:

1. Native vs Hybrid vs PWA - A Complete Breakdown: Imaginary Cloud Blog
2. PWA vs Native vs Hybrid App – A Comprehensive Comparison: Medium – Zorbis Inc
3. Mobile App Development Frameworks - Explained: NS804
4. Top 10 Mobile Frameworks: GeeksforGeeks
5. How to Calculate Mobile App Development Cost in 7 Steps: Medium – Bhumi Khokhani
6. Mobile App Development Cost Explained: Cubix Blog
7. Why Are Requirements Important in Software and Apps Development?: NearContact
8. Requirements Engineering – Concepts and Trends: ScienceDirect
9. Software Engineering – Requirements Engineering Process: GeeksforGeeks