**Mobile App Development**

## **What is mobile application development?**

Mobile application development is the set of processes and procedures involved in writing software for small, wireless computing devices, such as smartphones and other hand-held devices.

Like web application development, mobile application development has its roots in more traditional software development. One critical difference, however, is that mobile apps are often written specifically to take advantage of the unique features of a particular mobile device. For example, a gaming app might be written to take advantage of the iPhone's accelerometer, or a mobile health app might be written to take advantage of a smartwatch's temperature sensor.

Today, the two most prominent mobile platforms are iOS from Apple and Android from Google. Phones and tablets from Apple come preloaded with essential applications, including a full web browser and the Apple App Store. Android devices also come preloaded with similar apps, and you can install more using the Google Play Store.

## **Types of mobile applications**

In the early years of mobile apps, the only way to ensure an app could perform optimally on any device was to develop the app natively. This meant that new code had to be written specifically for each device's specific processor. Today, the majority of mobile applications developed are device-agnostic.

Before developing an app, you need to determine which type you'll be creating. Here's a breakdown of several types of mobile app development technologies with information about each.

1. **Progressive Web Applications**

A progressive Web Application is a web app that uses modern web capabilities to deliver a native app-like user experience. PWAs are built using standard web technologies like HTML, CSS, and JavaScript. Unlike traditional web apps, PWAs take advantage of features like service workers and web app manifests to provide additional benefits like offline access, push notifications, and an icon on the user’s home screen.

## **Some key characteristics of PWAs:**

* **Progressive** — Work for every user, regardless of browser choice. They are progressive in nature as they work for all browsers.
* **Responsive** — Fit any device screen size: desktop, mobile, tablet, etc.
* **Connectivity independent** — Service workers are allowed to work offline or with low network connectivity.
* **App-like** — It feels like an app with app-style interactions and navigation.
* **Fresh** — Always up to date, thanks to service worker update process.
* **Safe** — Served over HTTPS to prevent snooping and ensure content hasn’t been tampered with.
* **Discoverable** — Identifiable as “applications” through W3C manifest and service worker registration and can be indexed by search engines.
* **Re-engageable** — Web push notifications re-engage users.
* **Installable** — Allow users to “keep” apps they find most useful on their home screen.
* **Linkabl**e — Easily shared via a URL and does not require complex installation.

Following these criterions can be quite difficult, reason why frameworks like **Gatsby**, **Next** and **Nuxt** can be used in order to achieve these features.

**Key Architecture Enabler**

1. **Manifest File:**

All manifest files are JSON files, supporting how PWA is treated as an installed application, including the look and feel, basic behavior within the operating system.

1. **Service Worker:**

These are scripts that run separately from the website and are used to handle network request and caching. It enables PWA to deal with poor network connectivity and helps to increase the performance of the application. It also enables them to work offline and handle push notifications.

The final enabler is that of the core architecture where it comes down to use Modern JavaScript Framework like **React** and **Vue.**  Apps Framework like Next and Nuxt provide even more tooling for constructing web application core architecture, providing different rendering techniques such as **SSG/SSR** increasing application’s performance and SEO friendly ability.

**2. Native Applications:**

Native apps are built using the native development tools and languages for each platform — Java/Kotlin for Android, and Swift/Objective-C for iOS. The codebase is platform-specific, and apps have direct access to all native device capabilities through platform-specific APIs.

Native apps provide the highest performance since everything is natively compiled and optimized for the given device OS and hardware. The downside is you need to build the app separately for each platform, resulting in higher development costs.

## What are the challenges of native app development?

While native app development is considered the best, the perception is slowly changing due to the advancement of non-native technologies and the challenges of native methodologies. Some of those challenges are as listed below:

### Lack of code flexibility

In the case of native app development, the same codebase cannot be used or tailored to be used on different platforms. Therefore, the more platforms you are targeting, the more complicated the code writing and maintenance becomes. Alternatively, if you have limited means, you might not be able to target multiple platforms with the same kind of resources.

### Longer turnaround time

Since separate code writing and maintenance is required, building native apps can take longer as compared to non-native apps. It is also not as easy to find dedicated Android and iOS developers as it is to find web app developers. Dedicated development projects also take longer to accomplish, even though it is easier and more powerful in the long run.

### High-development cost

Lack of code-flexibility, need for separate development teams and longer project timelines ultimately increase the budget of app development. For those targeting multiple platforms like Android and iOS, the budget can get way too high. However, this depends on your choice of development tool, app type, and other factors.

**3. Hybrid Applications:**

Hybrid apps are a combination of native and web technologies, which allows them to have the best of both worlds. Hybrid apps are coded in HTML5, CSS3, and JavaScript and run on a web view. They can be deployed to the App Store or Google Play.

Hybrid apps provide a wide range of benefits: they’re easy to develop and maintain because they use web technologies; they don’t require compiling; they can be integrated with an existing website; they can be easily ported between platforms; they allow users to access offline functionality; they support multiple languages by using JSON files.

**Comparison**

# **Performance**

Performance is a crucial factor when choosing between PWAs, native apps, and hybrid apps. This includes both speed and responsiveness.

PWAs can offer near-native performance thanks to features like service workers for caching assets and background sync for network requests. Well-optimized PWAs load fast even on slow networks. They are also responsive since they rely on web technologies like JavaScript. However, native apps still have a slight edge when it comes to consistently smooth animations and transitions.

Native apps offer the best performance since they are built specifically for the target platform. They have direct access to native APIs and are compiled into machine code, allowing them to achieve 60fps animations and seamless UX. However, the performance depends on the skills of the developer. A poorly coded native app can still be slow and clunky.

Hybrid apps fall somewhere in between. They use web technologies but are wrapped in a native container, giving them access to some native APIs for performance boosts. However, they still cannot fully match the speed and fluidity of well-optimized native apps, especially for complex visuals and animations. The performance can also deteriorate over time as the app grows.

So in summary — PWAs can offer great performance that meets most user needs, but native apps edge them out for consistently flawless animations and transitions. Hybrid app performance depends on the complexity but will likely fall short of well-coded native apps, especially as the app grows over time.

# **User Experience**

The user experience can vary significantly between PWA, native, and hybrid apps.

PWAs aim to provide an app-like experience while still leveraging the benefits of websites. The key advantage is the ability to work cross-platform, delivering a consistent UI whether on mobile, desktop or tablet. PWAs also load faster than websites and can provide features like push notifications. However, they may lack some native capabilities and smoothness compared to native apps.

Native apps provide the best user experience as they utilize platform-specific UI components and have full access to device capabilities. The UI is fast, fluid and optimized for each platform. Native apps can fully leverage features like advanced animations, gestures, 3D touch etc. However, native apps need to be built separately for each platform which can be more work.

Hybrid apps use web technologies but are wrapped in a native container which allows accessing some device capabilities not available to websites. The UI is rendered using web technologies so it may not be as smooth or fast as pure native. Hybrid apps can work cross-platform but may have limitations in leveraging native features. The upside is faster cross-platform development though the UX tradeoffs need consideration.

Overall, native apps deliver the best and most optimized UX. PWAs aim for app-like UX but with web strengths while hybrid apps take a middle path. The choice depends on development priorities and target users. For consumer apps demanding high quality native UX, native development is recommended.

# **Development**

Developing each type of app comes with its own set of advantages and disadvantages from a developer’s perspective.

## **Progressive Web Apps**

PWAs can be built using standard web development skills like HTML, CSS and JavaScript so there is no need to learn new languages or frameworks. The service worker and manifest file do add some additional configuration but overall PWAs are fast and easy for web developers to build. The main benefits of developing PWAs include:

* Leverage existing web dev skills and avoid learning new languages
* Quick iterations since there is no compilation step
* Seamless integration with web analytics and other web-based tools
* Can be hosted and deployed just like a website

The downside is that PWAs 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**

Developing native apps requires learning platform-specific languages like Swift/Objective-C for iOS or Java/Kotlin for Android. There is a steeper learning curve compared to web development. However, native apps have full access to device capabilities and APIs for the richest user experience. Other benefits include:

* Maximize performance through direct access to native APIs
* Tighter integration with OS features like notifications or widgets
* Distribution through app stores for discoverability and updates

The cons are slower development speed due to compilation and more effort to build for multiple platforms.

## **Hybrid Apps**

Hybrid apps use web development skills combined with a native wrapper framework like React Native, Flutter, or Ionic. This allows access to some native features while reusing existing web skills. Benefits of the hybrid approach include:

* Faster cross-platform development using web skills
* Partial access to native APIs through the framework
* Ability to deploy as a website or native app package

The downsides are that performance may not be as fast as pure native and there are some device integration limitations. The framework also adds its own learning curve.

Overall, it depends on the specific app requirements and team skills when choosing the best development approach. PWAs offer the fastest web development while native maximizes performance and customization. Hybrid provides a middle ground for teams with web skills.

# **Accessibility**

Accessibility refers to how easily people with disabilities can use an app. This is an important consideration for any app development approach.

* PWAs have good accessibility support across modern browsers. They utilize web standards like WAI-ARIA for accessible components. However, PWAs may have limited accessibility on some older platforms.
* Native apps have the best accessibility support on their target platform. iOS and Android provide robust accessibility APIs to support screen readers, text-to-speech, captions, and more. However, a native app only targets one platform.
* Hybrid apps also have good accessibility support on mobile when built properly with web standards. The embedded web content can interact with native APIs for accessibility. But support will vary based on review implementation.

Overall, native apps tend to provide the most robust accessibility support, followed by hybrid apps and PWAs. However, PWAs have the advantage of a web codebase that can be accessible across multiple platforms. So there is no clear winner — it depends on the target platforms and user needs. Developers should aim for high accessibility regardless of approach.

# **Offline Usage**

Offline capabilities are an important consideration when choosing between PWAs, native apps, and hybrid apps.

PWAs offer robust offline support through service workers and cache APIs. Once a user visits a PWA, it gets cached locally and loads like a native app when offline. PWAs can also leverage background sync to submit data and receive push notifications when the connection is restored.

Native apps have full access to device features and can build offline workflows leveraging local storage. Data persists, and the app remains functional without an internet connection.

Hybrid apps can also leverage browser caching and local storage for offline support. However, the webview container may limit access to some native features that fully native apps can utilize offline. Performance could suffer compared to a truly native experience.

Overall, PWAs and native apps generally provide better offline capabilities than hybrid apps. PWAs match or exceed hybrid apps for offline use cases, while native remains optimal for full control. The choice depends on weighing offline needs with other factors like development costs and time to market.

# **Maintenance**

Maintenance needs vary greatly between PWAs, native apps, and hybrid apps. PWAs require very little maintenance since the app codebase lives on a server. As long as the server is maintained, the PWA will be updated automatically for all users when they load the app. Some additional maintenance may be required for the service worker and cache, but overall maintenance needs are minimal.

Native apps require releasing new versions of the app in the relevant app store(s) whenever new features or fixes need to be deployed. This makes native app maintenance more complicated than PWAs. Automated testing and CI/CD pipelines help streamline some of these processes.

Hybrid apps require app store deployments like native apps. But they also require maintenance of the web codebase that renders within the native wrapper. This makes maintenance more complicated than native or PWA. Any changes to UI, business logic, or data require repackaging and redeploying the hybrid app. Developers must also ensure the web and native layers remain in sync through ongoing maintenance.

So in summary, PWAs have the lowest maintenance overhead, followed by native apps. Hybrid apps require the most involved maintenance processes.

# **Security**

Native and hybrid apps have the advantage when it comes to security compared to PWAs.

Native apps are written specifically for the platform they are built for, allowing access to built-in security features provided by the OS and hardware. For example, iOS apps can take advantage of Apple’s App Transport Security feature which enforces secure connections. Android apps can integrate with the Keystore system to securely store cryptographic keys.

Hybrid apps also have access to native security features as they are essentially native apps running embedded web views. The native container handles permissions, certificates, and encryption while the web code accesses APIs through a bridge.

PWAs on the other hand rely on the security model of the browser, which sandboxes web resources to prevent access outside of it. Without native integration, PWAs cannot access native encryption, key stores, trusted hardware, etc. PWAs also do not have the walled garden approach of app stores for vetting.

However, PWAs can implement other measures like HTTPS, CORS, Content Security Policy, and authentication with protocols like OAuth to strengthen security. When hosted on HTTPS, PWAs can take advantage of TLS for secure communication. Proper implementation is key for PWAs to provide adequate security.

## When to use web apps vs. hybrid apps vs. native apps?

Large companies have to use a combination of native, hybrid, and web apps to reach the widest possible customer base. Before choosing the best app type for your use case, you can consider the following factors.

### **Time to market**

Early stage start-ups prefer using web apps to release a minimum viable product for customers as soon as possible. Native and hybrid apps require comparatively more time, planning, and effort to launch successfully.

### **Customer requirements**

Some products and services have large customer bases that use mobile apps regularly to complete tasks. In this case, native applications are preferred over hybrid and web apps.

### **Marketing strategy**

For some companies, their app development project is often tied closely to their marketing goals. They use a progressive web app to reach the largest possible audience and get initial sign-ups. The web app may have limited functionality or offer full-feature free trials for a limited period. The company then uses native or hybrid mobile apps to enhance the experience of paying customers.

### **Complexity**

In some instances, the mobile app functionality may be so complex that there is no option but to develop hybrid apps or native apps that support the requirements. For example, mobile banking apps require native features to keep fingerprint authorization features.

**MOBILE APP PROGRAMMING LANGUAGE**

The Different programming Languages used to develop mobile applications are categorized according to the type of Mobile application as follows

1. **Native Apps:**

These are a collection of frameworks allowing developers to create mobile device applications. Native apps are built using the programming language and development environment specific to each operating system, resulting in apps that can only be used on the device they were created for.

1. **Languages used to create IOS native applications**

#### **SWIFT AND OBJECTIVE-C**

Swift and Objective-C are two of the most common native app development languages.

The Swift app is a relatively new language created by Apple in 2014 to replace Objective-C. It is designed to be easier to use, more modern, and more reliable than Objective-C.

Brad Cox and Tom Love created Objective-C in the 1980s to make writing software easier on early personal computers when memory was limited.

1. **Languages used to create Android native applications**

#### **KOTLIN AND Java**

Kotlin is a JVM-based programming language that JetBrains created. It is an object-oriented, statically typed language with support for functional programming and a strong focus on interoperability with Java. Kotlin is completely interoperable with Java and can be compiled into Java bytecode or JavaScript source code.

Kotlin’s primary development is at JetBrains, which uses the language for its products, including the IntelliJ IDE.

Java has been the primary programming language for Android app development for many years. It offers a rich set of libraries and tools provided by the Android SDK.

1. **PWA apps:**

Progressive Web Apps (PWAs) are developed using web technologies, so the primary programming languages used for building PWAs are:

HTML (Hypertext Markup Language): HTML is the standard markup language for creating web pages and web applications. It defines the structure and content of a web page, including text, images, links, and other elements.

CSS (Cascading Style Sheets): CSS is used to style the appearance and layout of web pages. It allows developers to define colors, fonts, spacing, and other visual aspects of the user interface.

JavaScript: JavaScript is a programming language that enables dynamic and interactive behavior on web pages. It is used to add interactivity, handle user inputs, manipulate the DOM (Document Object Model), make HTTP requests, and perform other client-side tasks.

1. **Hybrid apps:**

Hybrid apps combine elements of both native and web apps. They are built using web technologies (HTML, CSS, JavaScript) and are wrapped in a native container that allows them to be distributed through app stores.

Hybrid app frameworks like Apache Cordova, Ionic, and Xamarin facilitate the development of cross-platform apps using web technologies.

While hybrid apps offer faster development and easier maintenance compared to native apps, they may not provide the same level of performance and user experience.

**MOBILE APP DEVELOPMENT FRAMEWORKS**

A mobile development framework is a software design to facilitate the creation of mobile applications, by providing developers with a set of tools, libraries and reusable components that streamline the process of building testing and deploying mobile applications for various platforms such as IOS, Android and sometimes cross-platform solutions.

**TYPES OF MOBILE DEVELOPMENT FRAMEWORK**

1. **Native Frameworks**

These are platform specific frameworks allowing developers to create applications exclusively for a single platform. Examples include;

**1. IOS (Swift with UIKit or SwiftUI)**

Programming Language: Swift

Performance: High performance as it directly accesses platform-specific APIs.

Cost and Time to Market: Generally higher cost and longer time to market due to separate development efforts for each platform.

UX and UI: Can offer the best native user experience and interface.

Complexity: Moderate to high, especially for complex apps.

Community Support: Strong community support with extensive documentation and resources.

Usage: Exclusive for iOS devices.

**2. Android (Java or Kotlin with Android SDK)**

Programming Language: Java or Kotlin

Performance: High performance with direct access to Android APIs.

Cost and Time to Market: Similar to iOS, higher cost and longer time to market for multiple platform support.

UX and UI: Provides a native Android user experience and interface.

Complexity: Moderate to high, depending on the app's complexity.

Community Support: Robust community with a vast array of libraries and resources.

Usage: Exclusive for Android devices.

**Hybrid Frameworks**

**1. React Native**

Programming Language: JavaScript

Performance: Near-native performance by leveraging native modules.

Cost and Time to Market: Reduced cost and faster time to market due to cross-platform compatibility.

UX and UI: Can achieve native-like UX/UI with reusable components.

Complexity: Moderate, especially for apps that require native modules.

Community Support: Strong community support with a large number of libraries and plugins.

Usage: Cross-platform (iOS, Android).

**2. Xamarin**

Programming Language: C# with .NET

Performance: Near-native performance as it compiles to native code.

Cost and Time to Market: Reduced cost and faster time to market with shared codebase.

UX and UI: Native-like UX/UI as it uses native components.

Complexity: Moderate, especially for apps with complex business logic.

Community Support: Solid community support with Microsoft backing.

Usage: Cross-platform (iOS, Android, Windows).

**Progressive Web Application (PWA) Frameworks**

**1. Angular (for PWAs)**

Programming Language: TypeScript

Performance: Good performance but may not match native app performance.

Cost and Time to Market: Lower cost and faster time to market as they are web-based.

UX and UI: Can offer a responsive and app-like user experience.

Complexity: Moderate, especially for complex web apps.

Community Support: Strong community support with Google's backing.

Usage: Web-based, accessible via browsers on various devices.

**2. Vue.js (for PWAs)**

Programming Language: JavaScript

Performance: Good performance with a lightweight footprint.

Cost and Time to Market: Lower cost and faster time to market due to web-based nature.

UX and UI: Can achieve a responsive and engaging user experience.

Complexity: Moderate, suitable for small to medium web apps.

Community Support: Growing community with a variety of plugins and extensions.

Usage: Web-based, accessible via browsers on various devices.

3. Flutter

Programming Language: Dart

Performance: High performance with a compiled codebase.

Cost and Time to Market: Similar to React Native; cost-effective and quick.

UX and UI: Rich set of customizable widgets for native interfaces.

Complexity: Moderate; single codebase for multiple platforms.

Community Support: Growing community backed by Google.

Where to Use: Ideal for apps requiring custom UI/UX designs and animations on multiple platforms.

**MOBILE ARCHITECTURE AND DESIGN PATTERNS**

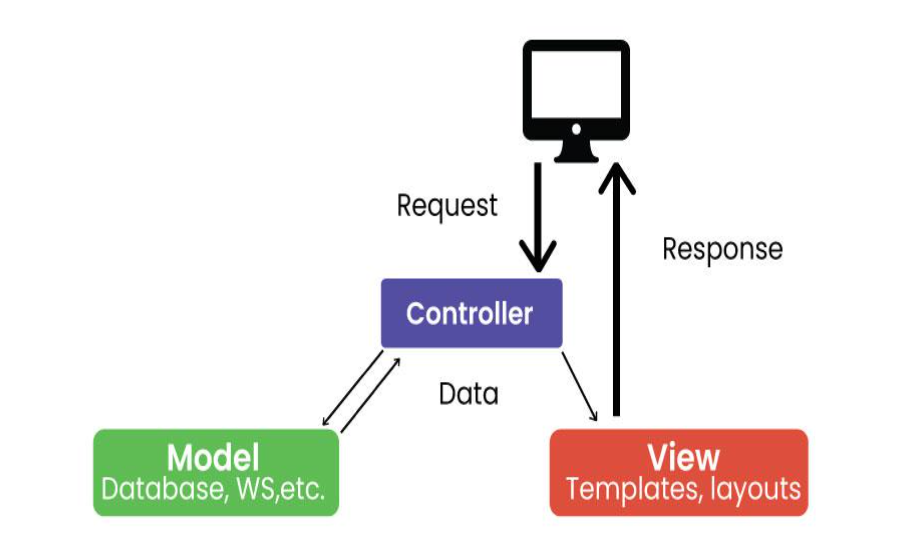
Mobile applications and design patterns are fundamental principles that guides the organization and structure of mobile apps. They help developers build apps that are scalable, maintainable and efficient.

Design Patterns are an integral part of modern software development, playing a pivotal role in enhancing the user experience (UX). These patterns represent well-established solutions to common design and programming challenges, providing a blueprint for creating software that not only functions efficiently but also delivers a superior user interface.

Important Topics for Enhancing User Experience Using Design Patterns are as follows

1. **Model View Controller (MVC) Architecture**

MVC is a design model that separates an application into three interacting parts: Model, View, and Controller. This separation allows for better code design and modularization.



**Model**: represents application data and business logic

**View**: displays data to the user

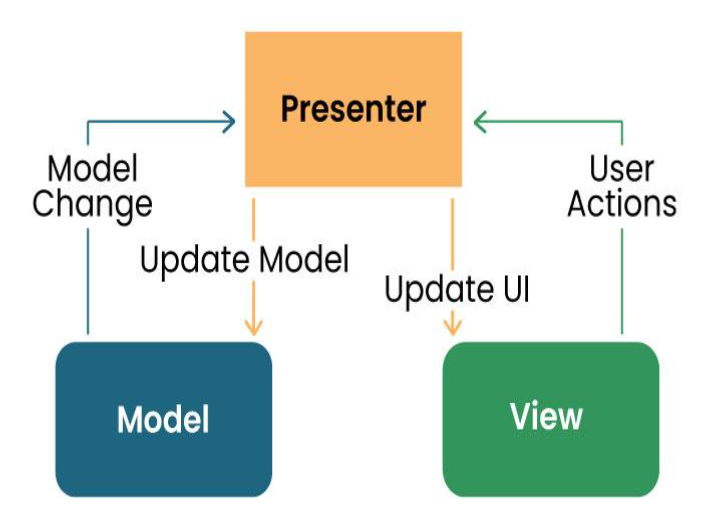
**Controller**: processes user inputs and controls data flow between model and view

For Example:

Imagine a mobile weather app. The model stores weather information, the View displays it to the user, and the controller handles user interactions such as updating the displayed location or converting units (e.g. from Celsius to Fahrenheit).

1. **Model View Presenter (MVP) Architecture**

MVP is a new architecture that separates an application into three parts: Model, View, and Presenter. This is similar to MVC but puts more responsibility on the Teacher to manage the interaction between Model and View.



Model: Manages data and business logic.

View: Represents the user interface.

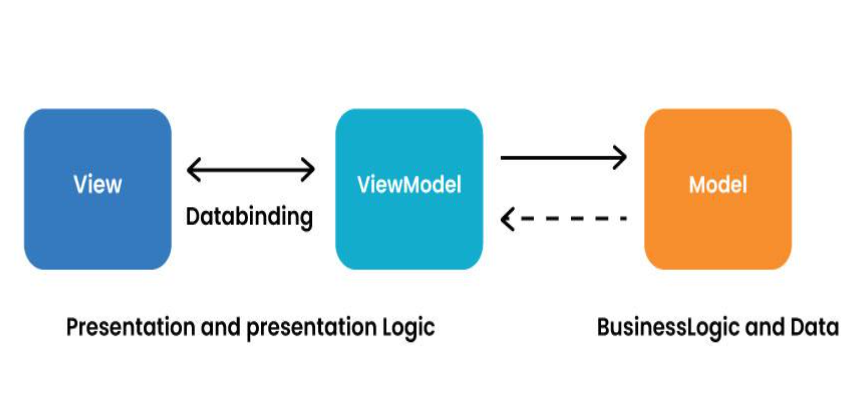
Designer: Acts as an intermediary processing user input and updating the View and Model.

For Example:

In a note-taking app, the Model would store the text, the View would display it, and the provider would handle user input such as typing, editing, or deleting the process.

1. **Mobile View View Model Architecture**

MVVM is a design model widely used in mobile development, especially in frameworks like Android’s Jetpack. Its purpose is to separate the application into three parts: Model, View, and ViewModel.



Model: Represents data and business logic.

View: Represents the user interface.

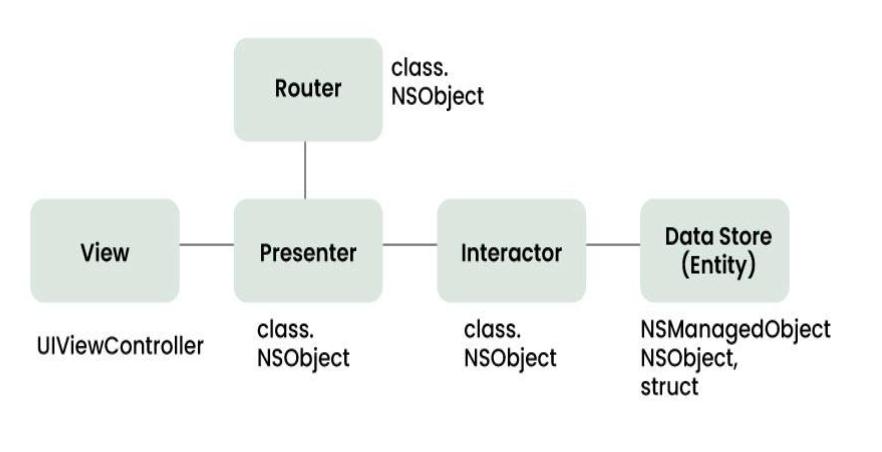
ViewModel: Acts as an interface between the Model and the View, which contains the reference logic.

For Example:

In an e-commerce application, the Model contains product data, the View displays product information, and the ViewModel manages interactions, such as adding items to a cart.

1. **VIPER Architecture**

VIPER stands for View, Interactor, Presenter, Entity, and Router. VIPER is primarily based at the clean architecture ideas, which purpose to separate the concerns of different layers of the utility. Each layer has a single duty and communicates with different layers through properly-defined interfaces.



Let’s briefly explain the function of every element:

**View**: This is the consumer interface layer, wherein the perspectives and look at controllers are defined. The view is chargeable for showing the information provided by way of the presenter and forwarding the person moves to the presenter.

**Presenter**: This is the presentation layer, where the good judgment for formatting and imparting the records is defined. The presenter is liable for fetching the records from the interactor, reworking it right into a suitable layout for the view, and updating the view hence. The presenter additionally handles the consumer movements acquired from the view and calls the router to navigate to other screens.

**Interactor**: This is the enterprise good judgment layer, where the common sense for manipulating the data and interacting with external services is described. The interactor is accountable for gaining access to the facts from the service layer, acting any vital operations on it, and returning it to the presenter. The interactor additionally communicates with the entity layer to store and retrieve the information fashions.

**Entity**: This is the information layer, wherein the data models and systems are described. The entity is responsible for representing the data in a constant and coherent manner throughout the software. The entity layer also can encompass records get entry to gadgets (DAOs) or repositories that summary the information of records patience and retrieval.

**Router**: This is the navigation layer, where the logic for routing and transitioning among different monitors is defined. The router is chargeable for developing and providing the view controllers, passing any vital facts to them, and coping with any dependencies or configurations. The router also communicates with the presenter to get hold of the navigation requests and execute them.

**DESIGN PATTERNS**

1. **Singleton Method Design Pattern**

The singleton policy ensures that there is only one instance of a class and provides global access. This is especially useful when you want to manage a single instance of an object or control access to a delayed object.

For Example:

Singleton can be used to manage player’s score in mobile game. There can only be one instance that is responsible for tracking scores and is updated throughout the game.

1. **Factory Method Design Pattern**

The Factory Method model defines an interface for creating an object but allows subclasses to modify the type of the created object. Especially useful when you need to create objects with a common interface but different functionality.

For Example:

in a mobile app that supports multiple payment gateways, payments can be made using the Factory Method. Each payment gateway (e.g., PayPal, Stripe) is a small business and provides its services.

1. **Observer Method Design Pattern**

The observer structure defines one to many dependencies between objects, so when one object changes its state, all its dependents are automatically notified and updated. This is useful for scheduling distributed events.

For Example: In the reports app, many features (Observers) such as the title widget, the report feed view, and the notification provider (Themes) can subscribe to updates when new information arrives. The observer model ensures that they are created all registered parts report, and accordingly You can update it.

1. **Dependency Injection (DI) Method Design Pattern**

Dependency Injection is a method of providing class dependencies from the outside, rather than creating them in the class. It improves code modularity and testability by making classes independent of their dependencies.

For Example:

In an Android app, instead of creating a single database connection object in the class, you can place the database object externally, allowing you to easily test and modify database operations.

1. **Adapter Method Design Pattern**

The adapter configuration allows you to use the interface of an existing class as a link to a new one. It is often used to work with others without modifying the source code of existing classes.

For Example: If you want to use a third-party library that provides data in a different way, you can create an adapter that will convert the library’s results to the format your app would expect and make sure that they are compatible meet without changing the library code.

1. **Strategy Method Design Pattern**

The strategy model defines a family of algorithms, contains each of them, and provides them with flexibility. It allows you to select the appropriate algorithm at runtime. This example is useful when you want to provide different options for a task.

For Example:

In a weather application, you can use various methods to retrieve weather information, such as using a REST API, WebSocket, or local storage. The user can change these options, and the app adapts to his preferences.

1. **Composite Method Design Pattern**

A composite pattern allows you to arrange objects in a tree structure to represent a part-of-the-whole structure. This is helpful when you have to deal with individual objects and sets of objects accurately.

For Example:

You can use Composite pattern to create complex shapes from simple shapes in the mobile drawing app. Complex designs can contain individual designs, allowing users to manipulate and categorize resources as needed.

**REQUIREMENT ENGINEERING**

What Are Mobile App Requirements?

Mobile app requirements document the business logic, technical specifications, and development guidelines for mobile app developers to design the application of your business dreams.

It includes the key app’s features, app user personas, and business goals to ensure that multiple team members are on the same page before the software development process commences.

Mobile App Requirements Benefits

At Requiment, we often see that requirements-gathering for mobile or other applications has benefits, including:

A better overall project development

Improved project management throughout the software development process

Enhanced efficiency to document the technical aspects required for an optimal product

A higher probability for the right target audience that meets the business requirements

Smoother app development with a better user persona and user story

Higher chance of success in developing the desired features for the target user

Target users help develop the right user interface to improve user experience

A competitive advantage over other apps

The ability to target different user groups in one mobile app requirements document

Better scalability with all the features designed for a specific market

App technology that enhances user loyalty and satisfaction

Cost optimisation by removing unnecessary features

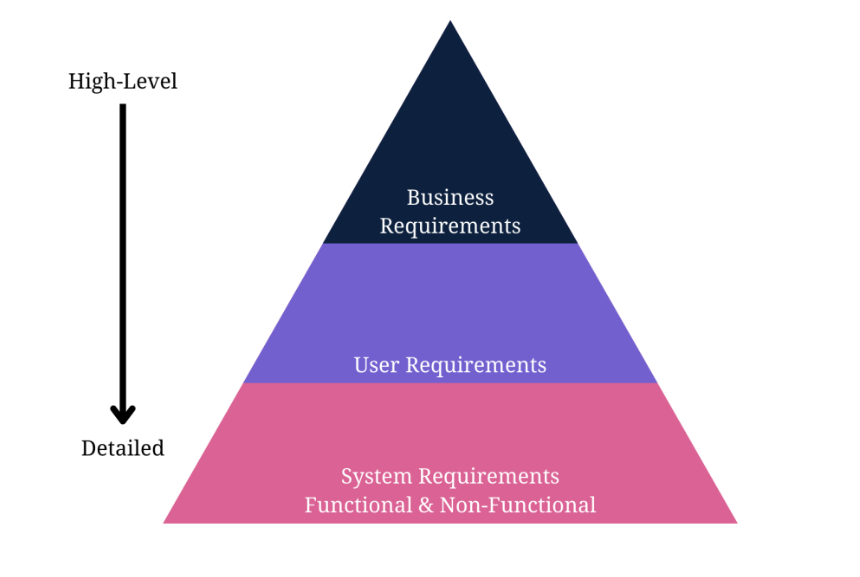
**Mobile App Requirement Types**

Mobile apps have different requirement types to collect, including:

**Business requirements** are high-level requirements that ensure the app will align with business objectives, and the project’s scope, and identify the key stakeholders.

**User requirements** are valuable insights into what your target audience needs and wants, how you can solve their problems, and what the audience experiences from your prototyping app.

**Product or system requirements** are non-functional requirements and functional requirements that include technical requirements and technical specifications for the engineering team.



Next, let’s show you the steps to gather requirements for a mobile application to ensure you meet the business requirements and key features necessary to develop a profitable product.

Some steps will involve sub-steps to help you gather the right information and requirements for documents.

### Step 1: Define Your App Idea and Purpose

Mobile development requirements-gathering starts with a business idea. The first information you need is the idea or purpose of the mobile app.

What purpose will it serve? Does it offer a solution to a potential problem?

You need to identify a problem the app will solve to recognise the idea or purpose behind it.

[Requirements gathering and management for mobile apps](https://www.requiment.com/requirements-gathering-and-management-for-mobile-apps/) require some effort with massive results.

### Step 2: Gather and Align the App and Business Objectives or Goals

An app idea is fruitless without understanding business needs, business goals, and business rules.

This step encourages you to gather business requirements to understand how the enterprise aligns with the idea from the first step.

Gathering business requirements to document involves these steps:

1. Identify the stakeholders for the right mobile application software development based on the business idea.
2. Define clear and concise business goals and objectives to understand the project’s scope.
3. Elicit stakeholder requirements and user requirements with elicitation techniques.
4. Document the requirements in a business requirements document.
5. Validate your requirements with stakeholders for a further transparent and opportunistic process.

So, what requirements-gathering elicitation techniques could you use to gather stakeholder requirements for the business requirements document?

Elicitation techniques work for any requirements-gathering type.

Here are some popular and successful techniques to use:

* Analyse similar documents
* Analyse similar external and internal interfaces
* Brainstorm use cases and user stories
* Create user stories and use cases
* Hold stakeholder focus groups
* Host requirements workshops
* Interview all the relevant stakeholders
* Observe documents and case studies
* Prototype visual examples for feedback
* Reverse engineer the processes
* Use online surveys/questionnaires
* Validate ideas with stakeholders

Use our [tips for conducting effective stakeholder interviews](https://www.requiment.com/tips-for-conducting-effective-stakeholder-interviews-for-requirements-gathering/) to enhance this step.

### Step 3: Run a Market Analysis and Competitor Analysis

Conduct a market or competitor analysis to truly understand the user’s perspective and design the appropriate user personas.

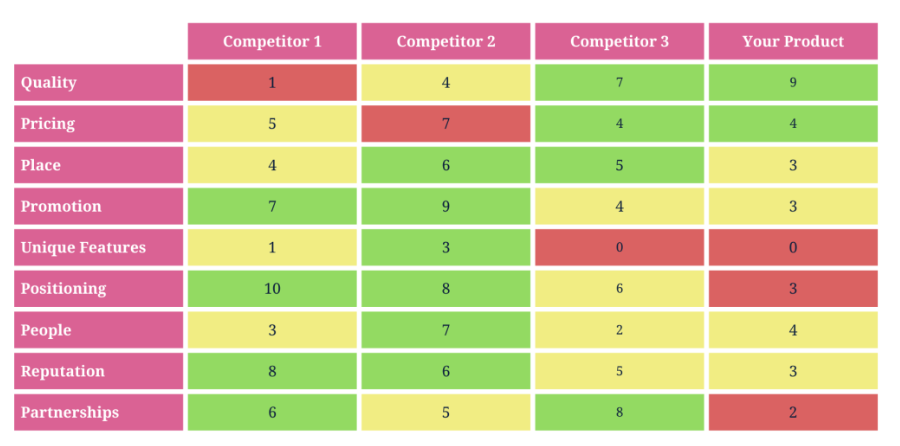
It also helps your team gather more user requirements for the development company.

The following steps explain the process of gathering user and competitor requirements:

1. Identify the direct, indirect, secondary, and substitute competitors for the mobile app. Remember to recognise any businesses offering similar mobile app services or products and those offering different products in a broader niche umbrella.
2. Gather competitor information, including products, descriptions, pricing structures, geographic reach, engaging promotions, target market positioning, business reputation, user profiles, and key partnerships to understand what your product needs to compete against.
3. Use a SWOT analysis to determine your competitor’s strengths, weaknesses, opportunities, and threats. You could learn from another app’s mistakes to improve your requirements and identify possibly unique features other apps don’t provide.

The SWOT analysis in a table will help you see what the new product needs to do to compete better with the top market competitors.

Rank each competitor from 1-10 on each key element.



1. Next, determine what competitive advantage you hold over other apps. For example, the sample analysis shows that your product quality is far superior to others. The pricing is also better than competitors. Choose at least three elements in which you wish to compete with other apps.

Fun fact: [Requirements gathering is the most undertaught area in software development](https://www.requiment.com/requirements-gathering-is-the-most-undertaught-area-in-software-development/).

### Step 4: Determine Scenarios and a User Persona

The next major step in mobile app requirements-gathering is to design user personas and scenarios to guide the requirements.

A user persona fictionalises the target users for the mobile app.

It should describe the ideal person who uses the app, with some flexible aspects for alternate users.

The ultimate user person could include the following details about target users:

* Age (also, typical generational qualities)
* Behavioural considerations
* Gender (including non-binary if relevant to the product)
* Geographic location
* Goal or problem the app addresses or solves
* Goal quotes or principles
* Goal-related frustrations
* Motivation to use the app
* Range of hobbies and daily activities
* Typical occupation range

How would you transform user personas into scenarios?

Create a persona scenario or storyboard by focusing on the goals, how their typical behaviours affect them, and how the persona’s background motivates them to respond differently.

Write the scenario as a short paragraph for starters, as you’ll design user stories later. Meanwhile, [identifying stakeholders for requirements gathering](https://www.requiment.com/identifying-stakeholders-for-requirements-gathering/) means creating personas.

### Step 5: Gather and Prioritise Functional and Non-Functional Requirements

Your user and business requirements are shaped through the initial steps of app requirements gathering.

You still need to document them, but you’ll do that soon enough.

Meanwhile, start prioritising the functional and non-functional requirements for the technical details, which also design use cases.

[Functional and non-functional requirements](https://www.requiment.com/what-are-functional-and-non-functional-requirements/) differ.

First, determine which functional app requirements the project needs.

Here are some examples of functional mobile requirements:

* A complete description of a feature the app offers or software interfaces.
* How the app allows users to sign up, verify accounts, or subscribe to a newsletter.
* Buttons and dashboards users interact with to complete a specified task.
* External and internal interfaces users interact with on the app.
* The necessary administrative functions for different user classes.
* Transaction adjustment, correction, and cancellation functions.

Secondly, determine the non-functional requirements necessary to run your app.

Here are some examples of non-functional requirements in mobile development:

* How fast the app responds to user input.
* How the app protects user and business data.
* Whether the app can work on multiple platforms.
* How much data does the app store and is it scalable?
* How reliable and maintainable the app remains.
* Does the app comply with local laws and regulations?

Next, you’ll prioritise the non-functional requirements (NFRs) and functional requirements for an app.

Priorities determine the tech stack and importance of each function.

The MoSCow prioritisation technique helps with any requirements prioritisation before documenting the requirements.

The technique requires you to put every technical requirement into one of four categories:

* **Must Have** – The highest-level requirements are critical to the requirements document to ensure the project’s success.
* **Should Have** – The second highest-level specifications are necessary for the project but won’t delay the progress of development or success.
* **Could Have** – The medium-level specifications could enhance user experience but aren’t dealbreakers if you don’t develop them right away.
* **Won’t Have** – The low-level requirements aren’t important to stakeholders at the time of requirements documentation and won’t affect the development process.

Include requirements for user experience (UX) and user interface (UI) with your functional and non-functional requirements.

It helps to have these requirements in place before designing use cases and documenting a key user experience and user flow requirement for app development.

[Requirements prioritisation](https://www.requiment.com/requirements-prioritisation-making-informed-decisions/) simplifies decision-making for your app.

### Step 6: Design Use Cases and User Stories

A mobile system requirements document won’t be complete unless you add use cases and stories.

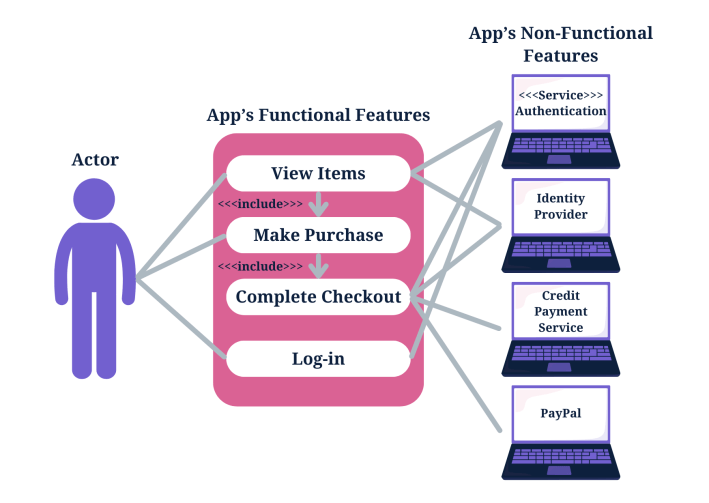
Design them before documenting the specifications for the development company or team.

Use stories and cases to add visual representation to your documents for mobile app development documentation.

#### How to Design a Use Case for App Requirements Documents

A use case diagram lets everyone at the development company visualise an overview of how users will interact with the app.

It’s an overview that includes actors, how actors interact with the app, and the sequence of interactions actors will deploy.

Here’s a simple example of a use-case diagram that shows an overview of how actors interact with functional features while non-functional features interact with the app from the back-end stack:

#### How to Design a User Story for App Requirements Documents

A story evolves the user persona you wrote earlier. Creating user stories lets the development company design an app that meets acceptance criteria.

Transforming a user persona into a story means you must follow the INVEST acronym to guide the criteria.

In other words, every user story must meet the following factors:

* **Independent** – stories must be independent from each other.
* **Negotiable** – The *what* and *why* of stories should be concrete, while the *how* shouldn’t be.
* **Valuable** – Stories should add value for all the stakeholders.
* **Estimable** – Each story should be capable of estimation.
* **Small** – Stories must be small enough for sprint completion.
* **Testable** – Stories must be testable with written acceptance criteria possible immediately.

Here’s a criteria example that depicts a good user story:

1. A business logo displays when users load the application.
2. Animation includes enlarging the logo until it fills the screen and a slow sideways transition to the next screen.
3. The user clicks on a dropdown menu on the top-right tab to search the categories, which include information about the company, shopping categories, FAQS about payment, a help guide, and a *Contact Us* page as some key features.
4. The user clicks on the shopping categories to open a new dropdown menu that includes hats, shirts, pants, and shoes.
5. Users click on hats, and the screen changes to show the products.
6. The animation includes an enlarging logo with a sideways screen transition.
7. Users click on the *add to cart* buttons when seeing a desired product.
8. The button fills with a tick, as the top screen shows a banner that the item was added to the cart.
9. The user clicks on the top-left dashboard button to enter the cart.

The criteria can go on but keep it short, as you’ll use different user stories, including one where the user checks an item out of their cart and makes payment.

### Step 7: Write an App Requirements Document

Delivering a proper app requirements document means you need to know how to write a mobile app requirements document.

Our [requirements-gathering template checklist](https://www.requiment.com/requirements-gathering-template-checklist/) can help you design a mobile app requirements template, as it works for web and app-based documentation.

Also, check out our [effective requirements documentation best practices for beginners](https://www.requiment.com/effective-requirements-documentation-best-practices-for-beginners/).

We guide you to know [how to write a software requirements document (SRD)](https://www.requiment.com/how-to-write-a-software-requirements-document-srd/), but let’s show you alternative steps.

Alternatively, follow our how-to write a mobile app requirements document steps to get the development process moving.

Mobile application development relies on the requirements document to design proper flow or the best app features and hit the right target audience.

#### Step 7a: Formulate the App’s Idea Statement

Every app requirements document should include an idea statement that lets every stakeholder and software developer understand the document before diving into the details.

Start your app requirements document with a simple single-sentence statement that aligns with the app’s idea.

#### Step 7b: Document All Relevant App Details

A detailed description of development plans in the requirements document is instrumental to completing the documentation.

A successful mobile app requirements document includes more than the details of an application and its functions for the development team.

##### Descriptions to Include in a Mobile Requirements Document

A mobile application requirements document should include a detailed description of functional and technical requirements and the app’s functionality to properly capture and represent the project’s scope.

* Business requirements
* User requirements
* Software requirements specification
* Technical specifications
* Functional specifications
* Non-functional requirements
* Hardware interfaces
* A list of must-have features
* Unique app features
* Internal and external interfaces
* Non-functional key metrics
* User stories
* Acceptance criteria

#### Step 7c: Prepare a Navigation Sequence

The development team requires a simple navigation sequence they can follow during software development.

The mobile app requirements document outlines the sequence in which the software development process flows.

Add the details from step 7b in a sequence that every developer can easily grasp.

#### Step 7d: Add Requirements Formats for Visuals

Add your user stories from a user’s perspective and the use case overviews you designed to the app requirements document to help stakeholders and the development team understand every aspect of the app requirements document.

Successful software development means knowing the intended users.

#### Step 7e: Add Cost Optimisation Details

The development team, stakeholders, and the client will appreciate a cost-benefit analysis to ensure cost optimisation throughout the software development process.

You’ll find guidelines for cost optimisation in your business objectives or budget.

Business analysts also insist on adding a cost-benefit analysis to an app requirements document to meet the business needs and have a greater chance of success against competing apps.

Did you know that [poor requirements management can lead to projects going over budget](https://www.requiment.com/why-poor-requirements-management-can-lead-to-projects-going-over-budget/)?

#### Step 7f: Add Communication Protocols

Ensure another key element is in your document before delivering the mobile app requirements document to a project manager, stakeholder, developer, or operating environment.

Add communication methods for a collaborative process. Collaboration relies on dependable communication.

The [importance of effective communication in requirements gathering](https://www.requiment.com/the-importance-of-effective-communication-in-requirements-gathering/) outlines why you need it.

#### Alternate App Requirements Document Process

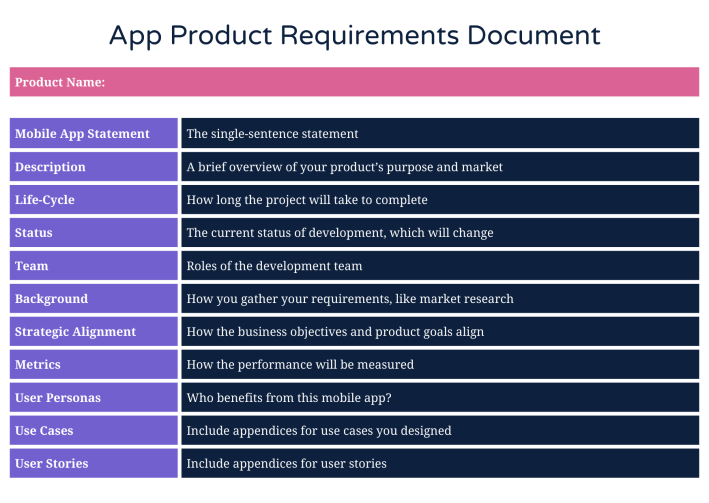
Alternatively, use a mobile app requirements document template to capture the requirements for app development.

A mobile app development requirements document template speeds up the process for non-business analysts or requirements analysts.

**Helpful related post on** [business analyst requirements gathering](https://www.requiment.com/how-to-gather-requirements-as-a-business-analyst/) how-to guide.

Our [product requirements document template](https://docs.google.com/document/d/1ZImLzeq96TrqafoChnl_bCfjK4yJX8O9ueoMhqdlELo/edit) for app development will help you document everything.

The requirements document focuses on product or system requirements to develop an app.

It’s a basic app requirements document template that app stores and development teams can follow.

### Step 8: Deploy Prototyping and Wireframing

Prototyping and wireframing let you design the user flow of user interfaces and basic app functions.

It also lets you test and validate layouts and transitions between app pages.

Here are the steps to wireframe your requirements for an app, which you will then validate in the next step:

1. Map the target user flow.
2. Sketch the flow’s core part.
3. Set a mobile wireframe.
4. Determine the layout with boxes.
5. Use design patterns.
6. Add intended copy.
7. Connect the app’s pages to design a flow.
8. Design a prototype.
9. Release the initial design to gather feedback.

Also, read about the [significance of wireframing in requirements gathering and validation](https://www.requiment.com/the-significance-of-wireframing-in-requirements-gathering-and-validation/).

### Step 9: Validate the App Requirements

Validation is a quality control process you use before launching the final product based on your requirements.

The prototype app collects feedback from stakeholders, and you can invite stakeholders to verify that the app meets the documented requirements. Use the feedback for the final step.

The [benefits of collaborative requirements gathering](https://www.requiment.com/the-benefits-of-collaborative-requirements-gathering/) share insights about feedback and collaboration.

### Step 10: Apply Agile Methodology

Agile methodology in requirements-gathering means you’ll always adapt the requirements document as per the feedback from stakeholders, testing, and initial product releases.

Agile methodology focuses on user experience and constant testing and validation to further improve your application.

You should know the [importance of updating requirement documentation throughout a project](https://www.requiment.com/the-importance-of-updating-requirement-documentation-throughout-a-project/).

## Summing Up Mobile App Requirements

Gathering the right requirements for a mobile development requirements document follows ten steps, some containing sub-steps with easy-to-follow guidelines.

Our Requiment product helps you through these steps, so don’t forget to [sign up](https://app.requiment.com/signUp) for the free trial.

However, our easy-to-access document template for mobile requirements-gathering also sets your project up for success.

Follow the steps in this guide, or check out our [complete guide to requirements-gathering in 2024](https://www.requiment.com/a-complete-guide-to-requirements-gathering-in-2023/) to ensure a smooth, successful project.

## Mobile Requirements FAQs

### What Are Android Mobile App Requirements?

An Android app requirements document will need the following additional technical details:

* APK file size
* App permissions
* App version
* Debug and test specifications
* Operating system and SDK version
* Restrictions

### What Are iOS Mobile App Requirements?

An iOS app requirements document will need the following additional technical specifications:

* A business model
* Accurate meta descriptions for all app features
* Additional documents for UIKit, AppKit, WatchKit, iOS data storage, and app extensions
* An innovative and unique design sample
* Extensive safety features
* Legal and regulatory requirements list
* Performance metrics
* SDK versions and permissions
* User experience design

**HOW TO ESTIMATE AN APP DEVELOPMENT COST**

Estimating the cost of mobile app development is a crucial step in the planning phase of any mobile app project. Several factors influence the overall cost, and understanding these factors can help businesses and developers make informed decisions. Below is a comprehensive study on how to estimate the cost of mobile app development.

**Factors Affecting Mobile App Development Cost**

**Scope and Complexity of the App:** The number of features, functionalities, and complexity of the app significantly impact the development cost.

**Platform:** Developing for iOS, Android, or both (cross-platform) can influence the cost. Native apps generally cost more due to platform-specific development.

**Design:** Custom UI/UX design vs. using standard templates or themes can affect the cost. High-quality, custom designs usually increase the development cost.

**Development Approach:** Native, hybrid, or progressive web app. Each approach has its cost implications, with hybrid and progressive web apps generally being more cost-effective.

**Backend Development:** Setting up servers, integrating databases, developing APIs, and other backend services add to the development cost.

**Testing:** Quality assurance, bug fixing, and device testing are essential stages that contribute to the overall cost.

**Maintenance and Updates:** Ongoing costs for updates, bug fixes, and customer support should be considered for long-term budgeting**.**

**Geographical Location of the Development Team:** Rates vary based on the location of the development team, with developers in North America and Western Europe generally charging higher rates compared to developers in Asia or Eastern Europe.

**Estimation Methods**

**1. Fixed Price**

**Description:** A fixed price is set based on the project scope, features, and requirements.

**Pros:** Predictable budget, clear scope of work, and timeline.

**Cons:** Less flexibility for changes and additions; may lead to scope creep.

**2. Time and Material**

**Description:** Costs are calculated based on hourly rates multiplied by estimated development hours.

**Pros:** Flexibility to make changes during the development process; suitable for agile development**.**

**Cons:** Uncertain final cost, potential for exceeding the budget.

**3. Value-Based Pricing**

**Description:** Pricing is based on the value provided to the client or business, considering factors like ROI, market demand, and competitive landscape.

**Pros:** Aligns with business goals and expected outcomes; potentially higher ROI.

**Cons:** Difficult to quantify value; requires a deep understanding of business objectives.