REPUBLIQUE DU CAMEROUN

PAIX – TRAVAIL – PATRIE

MINISTERE DE L'ENSEIGNEMENT SUPERIEURE

UNIVERSITE DE BUEA

REPUBLIC OF CAMEROON

PEACE – WORK – FATHERLAND

MINISTRY OF HIGHER EDUCATION

UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY
PO BOX 63
BUEA, SOUTH WEST REGION

COURSE TITLE/CODE:

INTERNET PROGRAMMING(J2EE) AND MOBILE PROGRAMMING CEF440

REPORT:
MOBILE APP DEVELOPMENT PROCESS

Presented by:

BEH CHU NELSON	FE22A170
CHEMBOLI ROYWINFEEL	FE22A180
NFOR RINGDAH BRADFORD	FE22A257
SONE BILLE MILTON	FE22A197
TCHOUANI TODJIEU EMMANUEL	FE22A313

COURSE INSTRUCTOR: Dr. Eng. Nkemeni Valery

ACADEMIC YEAR: 2024/2025

Table Of Content

1. Introduction

a. Definition of Internet Programming (J2EE) and Mobile Programming

Internet Programming:

Refers to the development of web-based applications that operate over the internet. One of the key technologies for internet programming is **J2EE(Java 2 Platform, Enterprise Edition),** which is a framework for building scalable, secure, and robust enterprise application. J2EE provides tools such as **Servlets, JSP(JavaServerPages), EJB(Enterprise JavaBeans), and Web Service**, making it suitable for large-scale distributed application.

Mobile Programming:

Focuses on developing application specifically for mobile devices such as smartphones and tablets. It involves various platform, technologies, and frameworks for building mobile applications. Mobile programming plays a crucial role in modern software development, enabling businesses to reach users across different devices.

b. Purpose of the Report

The mobile application industry has evolved significantly, providing multiple development approaches, frameworks, and architectures. This report aims to analyse the different types of mobile applications, programming languages, and frameworks, along with key aspect such as **architecture design, requirement engineering, and cost estimation.** By understanding these concepts, developers can make informed decisions when creating mobile application.

c. Scope and Objectives

This report will cover:

- A **comparative analysis** of different mobile app types.
- A review of **mobile programming languages** and their use cases.
- A comparison of **mobile development frameworks** based on performance, cost, and complexity.
- An overview of **mobile app architectures and design patterns.**
- An exploration of **requirement engineering techniques** for mobile development.
- An analysis of **mobile app development cost estimation** and influencing factors.

2. Types of Mobile Apps

Mobile applications can be categorized based on their development approach, platform compatibility, and performance. **The three main types of mobile applications are Native Apps, Progressive Web Apps (PWAs), and Hybrid Apps.**

a. Native Apps

Native applications are developed specifically for a single platform (Android or iOS) using platform-specific programming languages and frameworks.

Key Features:

- Developed using Swift (iOS) or Kotlin/Java (Android).
- Optimized for high performance and responsiveness.
- Provides full access to device features such as camera, GPS, and push notifications.
- Requires installation from app stores (Google Play Store, Apple App Store).

Pros:

- ✓ High performance and efficiency.
- ✓ Better user experience (smooth animations, native UI components).
- ✓ Full access to hardware and software capabilities.

Cons:

- ✖ Higher development cost (separate codebases for iOS and Android).
- ✖ Longer development time compared to other approaches.
- ✖ Requires approval from app stores before deployment.

Example Applications:

- WhatsApp (Android & iOS)
- Instagram (iOS & Android versions)
- Google Maps

b. Progressive Web Apps (PWAs):

PWAs are web applications that behave like native apps by utilizing modern web technologies. They run in a browser but can be installed on a device for offline use.

Key Features:

- Built using HTML, CSS, JavaScript.
- Can work offline using service workers.
- Accessible via a web browser without installation.

- Provides push notifications and background sync.

Pros:

- ✓ Lower development and maintenance cost (single codebase for all platforms).
- ✓ No need for app store approval.
- ✓ Faster deployment and updates.

Cons:

- ✖ Limited access to device hardware features.
- ✖ Performance may not match native apps.
- ✖ User experience may vary across devices.

Example Applications:

- Twitter Lite(**X**)
- Uber Web App
- Pinterest PWA

c. Hybrid Apps

Hybrid applications combine elements of both native and web applications. They are built using web technologies but wrapped in a native container that allows them to be deployed as native apps.

Key Features:

- Developed using frameworks like **Ionic, React Native, or Flutter.**
- Runs on multiple platforms with a single codebase.
- Can access some native features through plugins.
- Installed via app stores like native apps.

Pros:

- ✓ Faster development compared to native apps.
- ✓ Single codebase for multiple platforms.
- ✓ Easier maintenance and updates.

Cons:

- ✖ Performance is lower than fully native apps.
- ✖ Limited access to device features.
- ✖ UI may not feel completely native.

Example Applications:

- Instagram
- Facebook (previously hybrid before moving to native)
- Uber

d. Comparative Analysis Differences, Pros, and Cons

Features	Native App	PWAs	Hybrid Apps
Performance	High	Medium	Moderate
Development Cost	High	Low	Medium
Platform Dependency	Single(Andriod/iOS)	Cross-Platform	Cross-Platform
Access to Device Features	Full	Limited	Partial
Installation	Requires App Store	No Installation needed	Require App Store
User Experience	Best	Good but broswer-dependent	Acceptable
Development Process	Long	Short	Medium
Example	WhatsApp, Instagram	Twitter(**X**) Lite, Uber Web	Instagram, Facebook

3. Mobile Apps Programming Languagues: An Overview and Comparative Analysis

In the dynamic realm of mobile app development, selecting the appropriate programming language is essential for creating efficient, robust, and user-friendly applications. As mobile platforms continue to evolve, developers face an array of choices, each offering unique features, benefits, and challenges. This report provides an overview of several prominent programming languages used in mobile app development, including Kotlin, Swift, JavaScript, and Dart, along with additional languages such as Java, Objective-C, C#, Ruby, and PHP. By examining these languages in terms of their performance, use cases, and popularity, this report aims to equip developers and stakeholders with the insights necessary to make informed decisions tailored to their specific project requirements.

a. Overview of Languages

I. Kotlin

Type: Statically typed, JVM language.

Platform: Primarily for Android development.

Features:

- Concise syntax.
- Interoperability with Java.
- Null safety to reduce crashes.
- Support for functional programming.

II. Swift

Type: Statically typed, compiled language.

Platform: Primarily for iOS and macOS development.

Features:

- Modern syntax and features like optionals.
- Strong performance due to compiled nature.
- Safety features to prevent common programming errors.
- Extensive libraries and frameworks (e.g., SwiftUI).

III. JavaScript

Type: Dynamically typed, interpreted language.

Platform: Primarily used for web development; can be used in mobile via frameworks like React Native or Ionic.

Features:

- Versatile and widely used for both front-end and back-end.
- Asynchronous programming support.
- Large ecosystem with numerous libraries and frameworks.

IV. Dart

Type: Statically typed, compiled language.

Platform: Primarily used with Flutter for cross-platform mobile app development.

Features:

- Hot reload feature for faster development.
- Strong support for reactive programming.
- Rich standard libraries and tooling.

V. Java

Type: Statically typed, object-oriented language.

Platform: Primarily for Android development.

Features:

- Strongly supported by Android SDK.
- Object-oriented, making it suitable for large applications.

- Extensive libraries and frameworks.

VI. C#

Type: Statically typed, object-oriented language.

Platform: Primarily used with Xamarin for cross-platform development.

Features:

- Strong integration with .NET framework.
- Rich libraries and tools.
- Good performance, especially for games using Unity.

VII. Ruby

Type: Dynamically typed, interpreted language.

Platform: Used in mobile development via RubyMotion or through web frameworks.

Features:

- Elegant syntax and dynamic features.
- Focus on simplicity and productivity.
- Strong community and gems for various functionalities.

VIII. PHP

Type: Dynamically typed, server-side scripting language.

Platform: Primarily for web development, but can be used in mobile through frameworks like PhoneGap.

Features:

- Server-side capabilities for mobile backends.
- Large ecosystem of libraries and frameworks.
- Easy to learn for beginners.

b. Comparative Analysis

Language	Performance	Use Case	Popularity	Learning Curve	Community Support
Kotlin	High(JVM Optimized)	Android Apps	Growing Rapidly	Moderate	Strong (Android devs)
Swift	Very high (compiled)	iOS/ macOS	Very popular	Moderate	Strong (iOS devs)
JavaScript	Moderate	Cross-	Extremely	Low	Massive

	(interpreted)	platform, web apps	popular		
Dart	High (compiled AOT)	Cross-platform (Flutter)	Increasing with Flutter	Moderate	Growing
Java	High (JVM optimized)	Android apps	Established	Moderate	Strong
C#	Moderate	Cross-platform (Xamarin)	Growing rapidly	Moderate	Strong
Ruby	Moderate	Mobile web apps	Moderate	Low	Active
PHP	Moderate	Mobile backends	Well-established	Low	Large

In conclusion, the choice of programming language for mobile app development significantly impacts the project's success. Each language offers distinct advantages and is suited to different platforms and use cases. Kotlin and Swift dominate their respective platforms, while JavaScript and Dart provide flexibility for cross-platform development. Additionally, languages like Java, Objective-C, C#, Ruby, and PHP continue to play important roles in specific contexts. Understanding these languages' strengths and weaknesses enables developers to make informed decisions that align with their project goals and technical requirements.

4. Mobile App Development Frameworks

The mobile app development landscape offers a variety of frameworks, each with distinct advantages and trade-offs in performance, cost, development speed, and user experience. Choosing the right framework depends on project requirements, team expertise, and business goals. Provided here is a structured comparison of leading mobile app development frameworks, evaluating their key features—programming language, performance, cost and time efficiency, user experience (UX) and user interface (UI) capabilities, complexity, and community support. Additionally, ideal use cases for each framework to guide decision-making are highlighted.

a. Frameworks

1. Native (Swift/Kotlin)

Native development uses platform-specific languages (Swift for iOS, Kotlin/Java for Android) to deliver the best performance, full access to OS features, and superior UX/UI. However, it requires maintaining separate codebases, increasing cost and development time. Ideal for high-performance apps like games, AR/VR, and applications needing deep hardware integration.

2. Flutter (Google)

Flutter is a cross-platform framework using Dart, offering near-native performance with a single codebase, hot reload for faster development, and highly customizable widgets. While it reduces costs and speeds up time-to-market, apps may have larger file sizes and limited native library support. Best for MVPs and apps requiring custom UI, like e-commerce platforms.

3. React Native (Meta)

Built on JavaScript/TypeScript, React Native enables cross-platform development with reusable code (70-90% shared) and a large community. It delivers near-native UX but can struggle with complex animations. Perfect for social/media apps and startups needing rapid deployment with moderate performance.

4. Xamarin (Microsoft)

Xamarin uses C# to build cross-platform apps with shared logic and native-like performance. While it integrates well with Microsoft tools, it has a steep learning curve and declining adoption due to .NET MAUI. Suited for enterprise solutions and teams already using .NET.

5. Ionic (Web-Based)

Ionic leverages web technologies (HTML/CSS/JS) to build hybrid apps quickly and cheaply. However, performance lags behind native frameworks, and apps often feel like websites. Best for PWAs, simple apps, and internal tools where speed of development outweighs performance needs.

6. Kotlin Multiplatform (JetBrains)

Kotlin Multiplatform allows sharing business logic across platforms while keeping UI native. It's efficient for Android-first teams but requires Kotlin expertise and has a smaller ecosystem. Ideal for apps where logic reuse (e.g., networking, databases) is critical.

b. Comparative Analysis

Framework	Performance	Cost & Time	UX/UI Quality	Complexity	Best For
Native	Very good	High	Best	High	High-performance, OS-specific apps
Flutter	good	Low	Custom	Moderate	Cross-platform, custom UI
React Native	fair	Low	Near-native	Low	Social apps, fast MVPs
Xamarin	fair	Moderate	Native-like	High	Enterprise, .NET teams
Ionic	bad	Lowest	Web-like	Very Low	PWAs, simple apps
Kotlin MPP	good	Moderate	Native	High	Android-first, shared logic

The choice of a mobile app development framework depends on project priorities:

- ✧ **For maximum performance & native experience**: **Swift/Kotlin (Native)**.
- ✧ **For fast cross-platform development**: **Flutter** or **React Native**.
- ✧ **For web developers**: **Ionic** (simple) or **React Native** (better performance).
- ✧ **For enterprise/C# teams**: **Xamarin** or **.NET MAUI**.
- ✧ **For Android-focused projects**: **Kotlin Multiplatform**.

5. Mobile Application Architecture and Design

a. Common Architectures

I. Model-View-Controller (MVC)

This architecture separates an application into three interconnected components:

- **Model**: Manages data and business logic.

- **View:** Displays the user interface and interacts with users.
- **Controller:** Acts as a mediator between the Model and View.

This separation allows for organized and maintainable code.

- **Pros:** Modular, easy to test, reusable components.
- **Cons:** As complexity increases, the Controller can become overloaded.

II. Model-View-ViewModel (MVVM)

This architecture improves upon MVC by introducing the ViewModel. The ViewModel acts as a data handler, reducing the direct connection between View and Model.

This is widely used in mobile frameworks like Flutter and Android Jetpack.

- **Pros:** Better separation of concerns, improved testability.
- **Cons:** Requires additional effort to implement effectively.

III. Clean Architecture

This architecture is designed to separate concerns into distinct layers:

- **Presentation Layer:** Manages the UI and user interaction.
- **Domain Layer:** Handles business logic and use cases.
- **Data Layer:** Manages repositories, APIs, and databases.

Clean architecture makes applications scalable and maintainable.

- **Pros:** Highly modular, testable, and maintainable.
- **Cons:** Requires a well-structured implementation, making initial setup complex.

b. Design Patterns

I. Singleton Pattern

- Ensures a class has only one instance and provides a global access point.

-This pattern is useful when only one object should control certain application states.

- **Example:** Managing shared resources such as database connections or configuration settings.

II. Observer Pattern

- Defines a dependency between objects, where changes in one trigger updates in others.

- Commonly used in event-driven programming.

- **Example:** In mobile apps, UI elements can observe changes in a data source and update automatically.

III. Factory Pattern

- Provides an interface for creating objects without specifying their exact class.
- Useful when multiple similar objects need to be instantiated dynamically.
- **Example:** Creating different types of UI components based on a user's theme selection.

6. Requirement Engineering for Mobile Applications

Requirement Engineering (RE) is a critical phase in the Software Development Life Cycle (SDLC) that ensures the successful development of mobile applications. It involves systematically gathering, analyzing, and validating user and system requirements to align the final product with stakeholder expectations. Given the dynamic nature of mobile app development, selecting appropriate requirement elicitation and analysis techniques is essential to mitigate risks such as scope creep, miscommunication, and project failure. This section explores the most effective methods for **collecting user requirements** and **analyzing requirements** in mobile app development, as identified through a systematic literature review. Additionally, it highlights common challenges and best practices to optimize the requirement engineering process.

a. Methods for Collecting User Requirements

Eliciting accurate and comprehensive user requirements is fundamental to developing a successful mobile application. Below are the most widely used techniques, along with their advantages and limitations.

I. Interviews

Description: Structured or semi-structured face-to-face discussions with stakeholders to gather detailed insights.

Advantages:

- Enables deep understanding through direct interaction.
- Allows for immediate clarification of ambiguities.

Disadvantages:

- Time-consuming, especially with multiple stakeholders.
- Potential bias based on interviewer framing.

II. Questionnaires and Surveys

Description: Standardized forms with open or closed-ended questions distributed to a broad audience.

Advantages:

- Cost-effective and scalable for large user groups.
- Facilitates quantitative data analysis.

Disadvantages:

- Limited ability to capture nuanced user needs.
- Risk of low response rates or misinterpretation.

III. Brainstorming

Description: Collaborative sessions where stakeholders generate ideas freely.

Advantages:

- Encourages creativity and diverse perspectives.
- Rapidly identifies potential features.

Disadvantages:

- May produce unfiltered or impractical suggestions.
- Requires skilled moderation to stay focused.

IV. Prototyping

Description: Developing an early functional or visual model of the app for stakeholder feedback.

Advantages:

- Helps stakeholders visualize the final product.
- Reduces misunderstandings early in development.

Disadvantages:

- Resource-intensive (time and cost).
- May lead to unrealistic expectations if not properly managed.

V. Joint Application Development (JAD) Sessions

Description: Intensive workshops involving developers, users, and business analysts.

Advantages:

- Accelerates decision-making through real-time collaboration.
- Aligns technical and business requirements effectively.

Disadvantages:

- Requires significant preparation and coordination.
- Can be expensive due to participant involvement.

VI. Use Cases and Scenarios

Description: Narrative descriptions of how users interact with the system in specific situations.

Advantages:

- Easy for non-technical stakeholders to understand.
- Clarifies functional requirements and workflows.

Disadvantages:

- May not cover all edge cases or non-functional needs.

VII. Observation and Ethnographic Studies

Description: Studying users in their natural environment to uncover implicit needs.

Advantages:

- Reveals unspoken user behaviors and pain points.
- Provides contextual insights that surveys may miss.

Disadvantages:

- Time-consuming and may require ethical approvals.
- Observer bias can influence findings.

b. Techniques for Analyzing Requirements

Once requirements are gathered, they must be systematically analyzed to ensure clarity, feasibility, and consistency. The following techniques are commonly used.

I. Requirement Modeling (UML, Data Flow Diagrams)

Purpose: Visually represents system processes and interactions.

Benefits:

- Improves stakeholder communication.
- Identifies gaps or contradictions early.

Limitations:

- May become overly complex for large-scale apps.

II. Laddering

Purpose: Hierarchically structures stakeholder goals to prioritize requirements.

Benefits:

- Clarifies dependencies between requirements.
- Helps in decision-making for feature prioritization.

Limitations:

- Difficult to manage with extensive requirement sets.

III. Card Sorting

Purpose: Organizes features into categories based on user input.

Benefits:

- Inexpensive and user-friendly.
- Reveals intuitive navigation structures.

Limitations:

- Less effective for highly complex systems.

IV. Requirements Reuse

Purpose: Adapts requirements from existing successful applications.

Benefits:

- Reduces development time and costs.
- Leverages proven best practices.

Limitations:

- May not fully align with unique project needs.

c. Common Challenges and Recommendations

Despite the availability of various techniques, mobile app development teams often encounter challenges in requirement engineering, including.

- **Ambiguity in Requirements**: Stakeholders may provide vague or conflicting inputs.
- ✓ **Solution**: Use prototyping and iterative feedback loops to refine requirements.
- **Changing Requirements**: Evolving user expectations can lead to scope creep.

- ✓ **Solution**: Implement agile methodologies to accommodate changes flexibly.
- **Communication Barriers**: Misalignment between developers and nontechnical stakeholders.
- ✓ **Solution**: Use visual aids (e.g., wireframes, flowcharts) to bridge understanding.
- **Stakeholder Conflicts**: Differing priorities among business, technical, and end-user groups.
- ✓ **Solution**: Conduct JAD sessions to facilitate consensus.

Conclusion

Effective requirement engineering is pivotal to the success of mobile applications. By employing a combination of elicitation techniques (e.g., interviews, prototyping, JAD) and analysis methods (e.g., modeling, laddering), development teams can ensure that requirements are accurately captured, well-structured, and aligned with stakeholder expectations. Addressing common challenges through iterative validation and clear communication further enhances the efficiency of the requirement engineering process.

7. Mobile App Development Cost Estimation

Mobile app development costs can vary significantly—from a few thousand dollars for a simple app to over $500,000 for complex enterprise solutions. According to industry research from **TMA Solutions**, understanding cost drivers and adopting structured estimation approaches is critical for budgeting accuracy. This section examines the **key factors influencing costs** and presents **proven estimation methodologies** to help businesses plan effectively.

a. Key Cost Factors in Mobile App Development

I. Platform Selection

a) **Native Apps (iOS/Android)**:
- Higher cost (separate codebases) but better performance.
  Example: A Swift-based iOS app + Kotlin-based Android app.

b) **Cross-Platform (Flutter/React Native)**:
- 30–40% cost savings with shared code (ideal for MVP).

c) **Publishing Fees**:
- Apple App Store ($99/year) vs. Google Play ($25 one-time).

II. Feature Complexity

Feature Tier	Examples	Cost Impact
Basic	Login, static content	$10K–$30K
Intermediate	API integrations, payments	$50K–$100K
Advanced	AI, AR/VR, real-time sync	$150K+

III. Design Requirements

UI/UX Complexity:

- Template-based ($5K–$15K) vs. custom animations ($30K+).

User Flows:

- More screens = **higher design/development time**.

IV. Development Team Structure

Team Type	Avg. Hourly Rate	Best For
Freelancers	$20–$80	Small projects
Agencies	$50–$150	Full-scale apps
In-House	Salaried	Long-term products

V. Backend & Third-Party Services

Infrastructure: AWS/Firebase ($1K–$10K/month).

APIs: Payment gateways (Stripe), maps (Google Maps).

VI. Post-Launch Costs

Maintenance: 15–20% of initial dev cost/year.

Updates: OS compatibility, feature additions.

b. Estimation Models

I. Feature-Based Breakdown

1. List all features (e.g., user profiles, push notifications).
2. Assign hours per feature (e.g., 20h for login, 50h for payments).
3. Multiply by hourly rate ($50–$150).

II. Agile Estimation (Scrum)

Sprint Planning: Estimate costs per 2-week sprint.

User Stories: Prioritize MVP vs. "nice-to-have" features.

III. Industry Benchmarks

App Type	Cost Range	Timeline
Simple MVP	$15K–$50K	2–4 months
E-commerce	$70K–$250K	6–9 months
Social Network	$100K–$500K+	8–12 months

IV. Contingency Planning

Reserve **15–20% extra** for scope changes or delays.

Cost Optimization Strategies

- ✓ **Start with MVP**: Validate ideas before scaling.
- ✓ **Use cross-platform frameworks**: Reduce duplicate efforts.
- ✓ **Outsource strategically**: Balance cost vs. expertise (e.g., Eastern Europe offers rates ~30% lower than the US).

Conclusion

As highlighted by **TMA Solutions**, app costs hinge on **platforms, features, team rates, and post-launch needs**. By combining **feature-based estimates** with **agile methodologies**, businesses can budget realistically. Always factor in a contingency buffer to accommodate unforeseen challenges.

For precise quotes, consult experienced developers and refine estimates iteratively.

8. Conclusion

a. Summary of Findings

This report analyzed mobile app development, covering app types, programming languages, frameworks, architectures, requirement engineering, and cost estimation. Key findings include:

- **App Types:** Native apps provide high performance but are platform-specific, while PWAs and hybrid apps offer cross-platform compatibility with some trade-offs.
- **Programming Languages:** Kotlin and Swift are best for native apps, while JavaScript and Dart power cross-platform frameworks like React Native and Flutter.
- **Frameworks:** Flutter and React Native simplify cross-platform development, while Xamarin is suited for enterprise applications.
- **Architecture & Design Patterns:** MVC, MVVM, and Clean Architecture improve maintainability, while Singleton and Observer patterns enhance scalability.
- **Requirement Engineering:** Effective user research ensures usability, while structured cost estimation helps manage budgets.

b. Recommendations

- Choose native apps for performance, hybrid/PWAs for wider reach.
- Select Flutter for rich UI, React Native for JavaScript-based development.
- Implement MVVM or Clean Architecture for scalability.
- Use structured requirement gathering and cost estimation for efficiency.
- Prioritize widely supported technologies for long-term sustainability.

9. Reference

a. Citation of Sources Used

- **ResearchGate. Requirements Elicitation Techniques in Mobile Applications: A Systematic Literature Review. Retrieved from** https://www.researchgate.net/publication/353063515_Requirements_Elicitation_Techniques_in_Mobile_Applications_A_Systematic_Literature_Review

- **TMA Solutions. Mobile App Development Cost. Retrieved from** https://www.tmasolutions.com/insights/mobile-app-development-cost