Análisis comparativo de plataformas para el desarrollo móvil: enfoque nativo (kotlinswift) vs. Multiplataforma (react nativeflutter).

Fabio Andrés Márquez Ortiz

Asesor:

María Patricia Amórtegui Vargas

Universidad Nacional Abierta y a Distancia UNAD

Escuela de Ciencias Básicas, Tecnología e Ingeniería – ECBTI

Tecnología en Desarrollo de Software

2025

Dedicatoria

Dedico esta tesis a la Universidad Nacional Abierta y a Distancia (UNAD), por ser la institución que me permitió acceder a conocimientos de calidad y por su compromiso con la formación de profesionales en tecnologías de la información. A mis padres, por su apoyo incondicional y por inculcarme valores de perseverancia y dedicación. A todos mis seres queridos entre ellos mi esposa y mi hija que me motivaron a seguir adelante en los momentos difíciles.

Finalmente, a María Patricia Amórtegui Vargas, por su invaluable acompañamiento y enseñanza, que han sido fundamentales para alcanzar esta meta. Esta dedicación es un reconocimiento a su labor y a la oportunidad que me brindan de crecer como profesional y como persona.

Agradecimientos

Quiero expresar mi más profundo agradecimiento a la Universidad Nacional Abierta y a Distancia (UNAD), por brindarme la oportunidad de formarme en un entorno académico de excelencia y por facilitar los recursos necesarios para llevar a cabo esta tesis. Su compromiso con la educación a distancia y abierta ha sido fundamental para mi crecimiento académico y personal.

A la directora y tutora María Patricia Amórtegui Vargas, mi sincero reconocimiento por su guía, apoyo constante y valiosos aportes a lo largo de este proceso. Su dedicación, paciencia y experiencia han sido clave para la realización de esta investigación, inspirándome a mejorar continuamente y a mantener un enfoque crítico y riguroso en mi trabajo.

Gracias por su orientación, motivación y por creer en mi potencial. Este logro también es suyo.

Resumen

El presente trabajo desarrolla un análisis comparativo entre las alternativas de desarrollo móvil nativo —Kotlin para Android y Swift para iOS— y las soluciones multiplataforma React Native y Flutter, con el propósito de identificar los criterios técnicos y estratégicos determinantes para elegir la tecnología más adecuada en proyectos de software móvil. La investigación se abordó mediante un enfoque documental y descriptivo, basado en literatura científica y documentación técnica especializada, centrando el estudio en indicadores como rendimiento, acceso a APIs nativas, experiencia de usuario, productividad y sostenibilidad del ecosistema. Los resultados evidencian que el enfoque nativo continúa liderando en escenarios que requieren máximo desempeño y dependencia de hardware avanzado; mientras que los frameworks multiplataforma, especialmente Flutter, ofrecen un equilibrio más favorable entre costos, tiempo de desarrollo y consistencia visual entre plataformas. Se propone finalmente una matriz de decisión como herramienta de selección tecnológica que facilita la alineación entre las necesidades del producto y los requerimientos del negocio.

Palabras clave: Desarrollo móvil, nativo, multiplataforma, flutter, react native.

Abstract

This research presents a comparative analysis of native mobile development—Kotlin for Android and Swift for iOS—against cross-platform solutions such as React Native and Flutter, aiming to determine the most relevant technical and business-oriented criteria for selecting the appropriate technology in modern mobile software projects. A qualitative and descriptive methodology was applied, grounded on scientific literature and official technical documentation, focusing on metrics such as performance, native API access, user experience, developer productivity, and ecosystem sustainability. Findings indicate that native development remains superior for use cases requiring maximum performance and deep hardware integration, while cross-platform frameworks—particularly Flutter—provide a more efficient balance between cost, development time, and visual consistency across platforms. Lastly, a decision-support matrix is proposed to guide technology selection according to product requirements and strategic priorities.

Keywords: Mobile development, native approach, cross-platform, flutter, react native.

Tabla de Contenido

Introducción ... 11
Marco Introductorio y Metodológico ... 13
Planteamiento del Problema ... 13
Pregunta de investigación ... 14
Objetivos ... 15
Objetivo General ... 15
Objetivos Específicos ... 15
Justificación ... 16
Metodología ... 18
Tipo y enfoque de la investigación ... 18
Método de investigación ... 19
Fuentes de información ... 19
Selección de fuentes de información a través de la metodología PRISMA ... 20
Conceptos, arquitecturas y evolución del desarrollo móvil nativo y multiplataforma ... 21
Antecedentes del desarrollo móvil ... 21
Fundamentos del desarrollo nativo ... 24
Enfoques multiplataforma y su evolución ... 25
Paradigma declarativo de interfaz ... 26
Arquitecturas y rendimiento ... 26
Ecosistema, comunidad y licencias ... 27
Tendencias de convergencia ... 27
Desarrollo móvil ... 28
Desarrollo nativo ... 28
Desarrollo multiplataforma ... 29

Lenguajes y frameworks principales 29

Interfaz de usuario (UI) y experiencia de usuario (UX) 30

Compilación y rendimiento 30

Ecosistema y consideraciones del desarrollo nativo (Kotlin/Swift) 32

Desarrollo móvil nativo 32

Entorno Android: Kotlin y Android Studio 33

Entorno iOS: Swift y Xcode 33

Beneficios del desarrollo nativo 34

Desafíos del enfoque nativo 34

Funcionamiento y arquitectura de React Native y Flutter 36

Desarrollo móvil multiplataforma 36

React Native: JavaScript + Bridge nativo 37

Ventajas técnicas 37

Limitaciones 37

Flutter: Dart + Motor gráfico Skia 38

Ventajas técnicas 38

Limitaciones 38

Algunas propuestas para desarrollo con React 38

Comparación técnica entre ambos frameworks 39

Análisis comparativo técnico y de negocio 40

Enfoque comparado: impacto en rendimiento, UX y negocio 40

Rendimiento y eficiencia 41

Experiencia de usuario (UX/UI) 41

Acceso al hardware y APIs del dispositivo 42

Costos de desarrollo y mantenimiento 42

Madurez del ecosistema y sostenibilidad futura 43
Propuesta de matriz de decisión para selección tecnológica 44
Definición de criterios y ponderaciones 44
Puntuación de tecnologías evaluadas 45
Validación con escenarios tipo 45
Discusión 46
Conclusiones 49
Referencias Bibliográficas 51

Lista de Tablas

Tabla 1 *Beneficios del Desarrollo Nativo*..34

Tabla 2. *Comparación técnica entre ambos frameworks.* 39

Tabla 3. *Rendimiento y eficiencia.*..41

Tabla 4. *Experiencia de usuario (UX/UI)*...41

Tabla 5. *Acceso al hardware y APIs del dispositivo.*...42

Tabla 6. *Costos de desarrollo y mantenimiento.*...42

Tabla 7. *Madurez del ecosistema y sostenibilidad futura.*..43

Tabla 8. *Definición de criterios y ponderaciones.*...44

Tabla 9. *Puntuación de tecnologías evaluadas.*...45

Tabla 10. *Validación con escenarios tipo,*..45

Lista de Figuras

Figura 1 *Identificación de Estudios. Metodología PRISMA*..20

Introducción

En la última década, el desarrollo móvil pasó de ser un complemento de la estrategia digital a convertirse en su columna vertebral. Millones de personas resuelven tareas bancarias, de salud, transporte y educación desde el teléfono; para las organizaciones, esto implica decidir cómo construir esas aplicaciones con el mejor balance entre desempeño, costo y tiempo de salida al mercado. Las cifras del sector no dejan dudas: los ingresos del mercado global de *apps* superaron los 520 mil millones de dólares y mantienen una tendencia creciente sostenida (Statista, 2023). En ese contexto, elegir entre enfoques nativos —Kotlin para Android y Swift para iOS— o multiplataforma —con React Native y Flutter como protagonistas— no es una cuestión de moda, sino de estrategia tecnológica.

Los entornos nativos ofrecen integración profunda con el sistema operativo, acceso directo a APIs y optimizaciones de compilación que favorecen la latencia, la estabilidad y el uso eficiente de recursos (Apple, s.f.; Google, s.f.-b; Wei & Li, 2022). Su principal costo es organizativo: dos bases de código, equipos con competencias específicas y procesos de prueba y liberación diferenciados. En paralelo, los frameworks multiplataforma prometen reutilización de código y mayor velocidad de desarrollo, apoyados en arquitecturas declarativas y ecosistemas maduros (Meta, s.f.; Google, s.f.-a). React Native se apalanca en JavaScript y un puente asíncrono hacia componentes nativos; Flutter dibuja cada píxel con su motor Skia y compila *Ahead-of-Time* para producción, lo que reduce pérdidas por serialización y latencias del puente (Google, s.f.-a; Google, s.f.-d).

La literatura reciente matiza el debate con evidencia empírica. Hay estudios que muestran mejoras sustantivas en rendimiento, usabilidad y experiencia del desarrollador en los enfoques multiplataforma (Ahmed, Uddin, & Das, 2022; Al-Aqrabi, Liu, Hill, & Antonopoulos, 2021). Aun así, persisten ventajas nativas en tareas de alto consumo gráfico, animaciones complejas o uso intensivo de sensores (Mehra, 2024). Los casos industriales

también son heterogéneos: mientras Airbnb documentó el abandono de React Native por razones de complejidad y desempeño (Mohindra, 2018), BMW reporta resultados positivos con Flutter en experiencias móviles y de infoentretenimiento (Google, s.f.-c), y múltiples compañías exhiben productos robustos con React Native (Meta, s.f.-b).

Marco Introductorio y Metodológico

Planteamiento del Problema

El auge del mercado móvil y la diversidad de opciones tecnológicas han generado una asimetría de información en equipos que deben decidir con rapidez y bajo restricciones de presupuesto. La pregunta no es "qué es mejor en abstracto", sino qué conviene para un caso de uso, un contexto organizacional y un horizonte de producto específicos. En el plano técnico, los enfoques nativos maximizan el control sobre el *pipeline* de renderizado y la interacción con el hardware; en el plano operativo, esa ventaja se paga con la duplicación de esfuerzos y mayores costos de mantenimiento (Apple, s.f.; Google, s.f.-b; Alsaid et al., 2021). En la acera opuesta, los frameworks multiplataforma elevan la productividad y aceleran la salida al mercado, pero pueden enfrentar límites al tratar con APIs complejas, animaciones exigentes o casos de uso que saturan el puente de comunicación (Al-Aqrabi et al., 2021; Mehra, 2024).

La investigación reporta resultados dispares ante problemas similares: Airbnb decidió regresar al desarrollo nativo por dificultades de rendimiento y coordinación entre plataformas (Mohindra, 2018), mientras BMW destaca la consistencia visual y la estabilidad de Flutter en escenarios de alta exigencia (Google, s.f.-c). En escalas intermedias, estudios comparativos señalan que Flutter y React Native han acortado brechas en consumo de CPU y tiempos de arranque, y que la productividad del desarrollador mejora con plantillas, bibliotecas y herramientas de *hot reload* (Ahmed et al., 2022; Meta, s.f.; Google, s.f.-a). Esta divergencia empírica complica la toma de decisiones: no hay un vencedor universal, sino conjuntos de condiciones bajo las cuales cada enfoque rinde mejor.

El problema central, por tanto, es la falta de un marco comparativo operativo que integre evidencia técnica, criterios de UX y costos de ciclo de vida para orientar la selección tecnológica. En la práctica, muchas decisiones siguen descansando en preferencias del

equipo, disponibilidad inmediata de talento o percepciones incompletas sobre desempeño y mantenibilidad (Al-Aqrabi et al., 2021; Suri et al., 2022). La consecuencia es predecible: deuda técnica, sobrecostos por reescrituras, cuellos de botella en animaciones o módulos nativos, y experiencias de usuario inconsistentes entre plataformas.

Pregunta de investigación

¿Cuáles son las ventajas, desventajas y criterios de selección determinantes al comparar el desarrollo móvil nativo (Kotlin/Swift) con los enfoques multiplataforma (React Native/Flutter) para la creación de aplicaciones en el ecosistema tecnológico actual?

Objetivos

Objetivo General

Analizar comparativamente las plataformas de desarrollo móvil nativo (Kotlin para Android, Swift para iOS) y multiplataforma (React Native, Flutter) para establecer un marco de criterios técnicos.

Objetivos Específicos

Fundamentar teóricamente los conceptos, arquitecturas y evolución del desarrollo móvil nativo y multiplataforma.

Describir el ecosistema, ventajas y consideraciones del desarrollo nativo para Android utilizando Kotlin y para iOS utilizando Swift.

Detallar el funcionamiento, la arquitectura de renderizado y las características principales de los frameworks multiplataforma React Native y Flutter.

Realizar un análisis comparativo basado en criterios técnicos (rendimiento, acceso a APIs nativas, consistencia de la UI/UX) y de negocio (costo y tiempo de desarrollo, mantenimiento, tamaño de la comunidad).

Proponer una matriz de decisión como guía de buenas prácticas que facilite la selección de una plataforma de desarrollo móvil según las características y prioridades del proyecto.

Justificación

La elección entre enfoques nativos (Kotlin/Swift) y multiplataforma (React Native/Flutter) se ha convertido en una decisión estratégica de alto impacto para organizaciones y equipos de desarrollo, pues condiciona el rendimiento, la experiencia de usuario, los costos de construcción y mantenimiento, y el time-to-market de las aplicaciones móviles. La magnitud del mercado y su crecimiento sostenido refuerzan la necesidad de decisiones informadas: el ecosistema de *apps* representa cientos de miles de millones de dólares anuales y continúa en expansión, lo que presiona a las empresas a optimizar recursos y priorizar la calidad del software (Statista, 2023). En este contexto, una evaluación comparativa rigurosa aporta evidencia para seleccionar la alternativa tecnológica más conveniente en función de los objetivos de negocio, las restricciones de proyecto y las expectativas de los usuarios.

Desde el punto de vista técnico, los avances recientes han reducido—aunque no eliminado—la brecha entre ambos enfoques. La UI declarativa y el refinamiento arquitectónico han impulsado mejoras notables: Flutter innova con renderizado directo mediante el motor Skia y despliegue *AOT*; React Native ha evolucionado su *stack* y ecosistema; y los entornos nativos integran modelos declarativos modernos como Jetpack Compose y SwiftUI (Google, s.f.-a; Google, s.f.-d; Apple, s.f.; Meta, s.f.). Aun así, la literatura reporta trade-offs: estudios empíricos hallan que los frameworks multiplataforma ofrecen ganancias de productividad y cobertura multi-OS, mientras que las aplicaciones nativas tienden a mantener ventajas en latencias críticas, gestión de memoria y acceso a APIs complejas, especialmente en escenarios de alto desempeño gráfico o integración profunda con hardware (Ahmed et al., 2022; Al-Aqrabi et al., 2021; Suri et al., 2022; Mehra, 2024). Esta tensión técnica exige un análisis sistemático que trascienda afirmaciones genéricas y contraste evidencia en dimensiones comparables.

La relevancia académica del estudio radica en cerrar brechas entre reportes dispersos (artículos científicos, guías técnicas, *white papers* y *showcases*) y ofrecer una síntesis crítica fundamentada. La literatura comparativa ha crecido, pero suele focalizarse en dimensiones aisladas—p. ej., rendimiento puro, productividad del desarrollador o percepción estudiantil—sin integrar de forma holística los factores técnicos, de UX, organizacionales y de ciclo de vida (Meirelles et al., 2019; Dekkati et al., 2019; Zarichuk, 2023; Zohud&Zein, 2021). Al combinar fuentes académicas y documentación oficial de proveedores, la tesis aportará un marco integrador que facilite transferencia de conocimiento a contextos reales de ingeniería (Google, s.f.-b; Meta, s.f.; Apple, s.f.).

Metodológicamente, una investigación documental comparativa permite consolidar evidencia reciente y contrastar hallazgos en categorías operativas—rendimiento, acceso a APIs nativas, consistencia UI/UX, productividad, mantenimiento, comunidad y licenciamiento—para derivar una matriz de decisión aplicable en entornos profesionales. Este instrumento facilitará a líderes técnicos y *product owners* ponderar sacrificios y beneficios según tipo de aplicación, complejidad funcional, recursos disponibles y prioridades de negocio (Kodithuwak & Pacillo, 2025; Riaz, 2025; Ugli & Woo, 2024). Asimismo, al incorporar criterios de UX y usabilidad—dimensiones frecuentemente subvaloradas en evaluaciones puramente de rendimiento—se promueve una visión centrada en el usuario final (Interaction Design Foundation, s.f.).

Metodología

La presente investigación se desarrollará bajo un enfoque cualitativo-descriptivo, con alcance documental y comparativo, orientado a analizar y contrastar las características técnicas, arquitectónicas, de rendimiento y usabilidad de las principales plataformas de desarrollo móvil actuales: Kotlin y Swift en el ámbito nativo, y React Native y Flutter en el ámbito multiplataforma.

La metodología tiene como propósito construir una base analítica sistemática que permita evaluar los criterios de selección tecnológica más relevantes para desarrolladores y empresas de software, fundamentando las conclusiones en evidencia científica y técnica proveniente de fuentes verificadas (Ahmed et al., 2022; Al-Aqrabi et al., 2021; Suri et al., 2022; Mehra, 2024).

Tipo y enfoque de la investigación

El estudio es de tipo documental porque se basa en la recolección, análisis y sistematización de información procedente de documentos académicos, técnicos e institucionales. Según Alsaid et al. (2021) y Zohud y Zein (2021), los análisis comparativos en ingeniería de software requieren examinar fuentes primarias (documentación oficial, artículos de conferencias, manuales de desarrolladores) y secundarias (revisiones, estudios empíricos y casos de aplicación).

El enfoque cualitativo se justifica porque la investigación no busca cuantificar variables mediante experimentación, sino interpretar y comparar fenómenos tecnológicos (arquitectura, rendimiento, productividad, UX) desde una perspectiva analítica. El alcance es descriptivo y comparativo, ya que se pretende detallar las características esenciales de cada plataforma y establecer similitudes, diferencias y ventajas relativas.

El diseño metodológico adopta la estructura de una revisión sistemática de literatura técnica, en la cual se identifican, analizan y categorizan las fuentes más relevantes sobre el tema (Kodithuwak&Pacillo, 2025; Ugli&Woo, 2024).

Método de investigación

El método que orienta el trabajo es comparativo-analítico, sustentado en la triangulación de datos técnicos y académicos. Este método se aplica en tres niveles:

Nivel descriptivo: identificación de las características técnicas, arquitectónicas y funcionales de cada plataforma.

Nivel comparativo: análisis cruzado de los criterios definidos (rendimiento, acceso a APIs, usabilidad, productividad, costo, comunidad, licenciamiento).

Nivel interpretativo: construcción de una matriz de decisión que sintetice los hallazgos y sirva de herramienta para la toma de decisiones tecnológicas.

La comparación se apoya en indicadores derivados de la literatura, como los propuestos por Ahmed et al. (2022) (uso de CPU, estabilidad y experiencia del desarrollador), Al-Aqrabi et al. (2021) (eficiencia, interoperabilidad y escalabilidad), y Suri et al. (2022) (UX, rendimiento gráfico y mantenibilidad).

Fuentes de información

Las fuentes se dividen en tres categorías:

- Fuentes primarias: documentación oficial de los frameworks, disponible en los portales de desarrollo de Google, Meta y Apple. Estas proporcionan información técnica precisa sobre arquitectura, compilación, API y licenciamiento (Google, s.f.-a; Meta, s.f.; Apple, s.f.).
- Fuentes secundarias: artículos académicos y conferencias indexadas en IEEE Xplore, Springer, ACM y otras bases reconocidas. Estas fuentes ofrecen resultados de pruebas

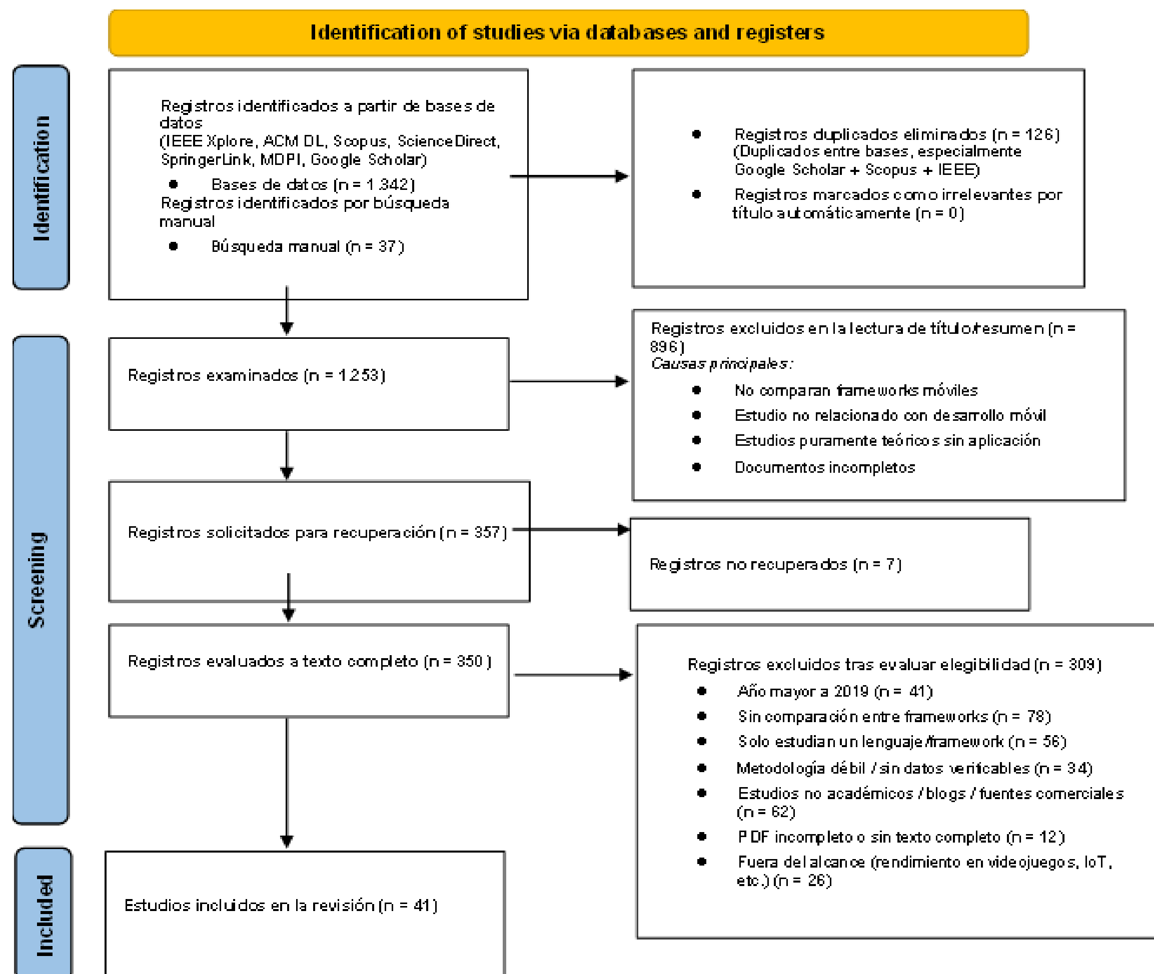
empíricas y comparaciones de desempeño (Ahmed et al., 2022; Al-Aqrabi et al., 2021; Meirelles et al., 2019; Olsson, 2020; Suri et al., 2022).

- Fuentes terciarias: informes técnicos, blogs especializados y estudios de caso empresariales (por ejemplo, Airbnb, BMW, Shopify), que ilustran la aplicación real de cada tecnología (Mohindra, 2018; Google, s.f.-c; Meta, s.f.-b).

Selección de fuentes de información a través de la metodología PRISMA

Figura 1

Identificación de Estudios. Metodología PRISMA.



Conceptos, arquitecturas y evolución del desarrollo móvil nativo y multiplataforma

Antecedentes del desarrollo móvil

El desarrollo de aplicaciones móviles ha transitado, en poco más de una década, desde un escenario dominado por soluciones nativas hacia un ecosistema donde conviven enfoques multiplataforma maduros. En su primera etapa, Android y iOS consolidaron sendas rutas tecnológicas apoyadas en SDKs oficiales y lenguajes dedicados —Kotlin (con interoperabilidad con Java) y Swift, respectivamente— para explotar al máximo el hardware y las APIs del sistema operativo (Apple, s.f.; Google, s.f.-b; Wei&Li, 2022). Este camino garantizó durante años un estándar alto de rendimiento, estabilidad y cumplimiento de guías de interfaz, a costa de una complejidad organizativa evidente: dos bases de código, perfiles de talento distintos y ciclos de pruebas y publicación duplicados (Alsaid et al., 2021).

La presión por reducir costos y acelerar el time-to-market abrió paso a soluciones híbridas y multiplataforma. Tras los primeros intentos —como Cordova o PhoneGap— emergieron propuestas con mayor ambición técnica. Un hito lo marca React Native (Meta), que lleva el modelo declarativo de React al móvil y permite escribir la lógica en JavaScript, comunicándose con componentes nativos a través de un puente asíncrono (Meta, s.f.; Al-Aqrabi, Liu, Hill, & Antonopoulos, 2021). Casi en paralelo, Flutter (Google) irrumpió con una aproximación distinta: un motor de renderizado propio (Skia) que dibuja directamente cada píxel y un *toolchain* que compila Ahead-of-Time (AOT) para producción, minimizando latencias asociadas a la serialización entre capas (Google, s.f.-a). Ambas corrientes incorporaron el paradigma de UI declarativa que, en el mundo nativo, también cristalizó con Jetpack Compose y SwiftUI, simplificando la gestión del estado y la coherencia de la interfaz (Google, s.f.-d; Apple, s.f.).

La literatura comparativa muestra cómo estas trayectorias convergen y se tensionan. Ahmed, Uddin y Das (2022) reportan mejoras sustantivas en consumo de CPU, tiempos de

arranque y estabilidad de las soluciones multiplataforma modernas, sin negar la ventaja nativa en tareas de alto desempeño gráfico o de acceso intensivo a sensores. En una línea similar, Al-Aqrabi et al. (2021) documentan beneficios en portabilidad y productividad con React Native y Flutter, pero advierten límites al tratar con APIs complejas, integración profunda con funciones del sistema o animaciones exigentes. Desde el aula, Meirelles et al. (2019) capturan la percepción de estudiantes y desarrolladores: mayor rapidez inicial y curva de aprendizaje amable en multiplataforma, frente a una sensación de pulido y determinismo más altos en proyectos nativos.

Los casos industriales han alimentado el debate público. Airbnb describió en 2018 los motivos para descontinuar React Native: dificultades para mantener la paridad funcional entre iOS y Android, sobrecarga en el puente de comunicación y complejidades de mantenimiento en una aplicación de gran escala (Mohindra, 2018). En contraste, BMW presentó la adopción de Flutter para experiencias móviles e in-car, destacando la consistencia visual, la fluidez y la capacidad de desplegar interfaces homogéneas en múltiples dispositivos (Google, s.f.-c). Por su parte, el showcase de React Native reúne productos de alto tráfico que se sostienen en ese ecosistema, lo que sugiere que, bajo ciertas condiciones de producto y equipo, el enfoque es plenamente viable (Meta, s.f.-b). La conclusión preliminar es nítida: no hay vencedor universal; existen configuraciones de requisitos y contextos organizacionales donde cada enfoque maximiza valor.

Al ampliar el foco, aparecen criterios de selección más finos. Kodithuwak y Pacillo (2025) proponen evaluar la elección tecnológica como un espectro, ponderando tipo de aplicación, complejidad de la UI, dependencia de APIs críticas, presupuesto y horizonte de mantenimiento. En esta misma clave, Riaz (2025) subraya la importancia del ecosistema (calidad de paquetes, ritmo de actualizaciones, resolución de *issues*) y la madurez de herramientas de pruebas, *CI/CD* e integración con servicios como Firebase o librerías nativas.

Estas dimensiones extratécnicas —comunidad, documentación, licencias— condicionan el costo total de propiedad tanto como los microsegundos obturados en un *benchmark*.

En el terreno del rendimiento y la UX, la evidencia reciente confirma una brecha cada vez más estrecha, pero no abolida. Olsson (2020) sugiere que en dispositivos actuales Flutter logra una fluidez cercana a la nativa en renderizado y transiciones, manteniéndose diferencias perceptibles en eficiencia energética. Suri, Taneja, Bhanot, Sharma y Raj (2022) añaden que las brechas afloran en animaciones complejas, listas de gran volumen y trabajos en segundo plano, donde la predictibilidad de la ruta nativa conserva una leve ventaja. En paralelo, estudios aplicados muestran cómo la productividad del desarrollador en multiplataforma —*hot reload*, plantillas, componentes reutilizables— acorta tiempos en las primeras iteraciones, aunque ciertas dependencias externas y la necesidad eventual de módulos nativos pueden erosionar parte de ese beneficio en el mantenimiento evolutivo (Dekkati, Lal,&Desamsetti, 2019; Meirelles et al., 2019).

Los fundamentos arquitectónicos explican parte de estas observaciones. React Native separa la lógica JS del hilo de UI nativa y sincroniza ambos por un bridge; esa frontera, si se satura con mensajes de alto volumen o frecuencia, puede convertirse en cuello de botella (Al-Aqrabi et al., 2021; Mehra, 2024). Flutter, al dibujar directamente con Skia, evita ese intercambio y controla el *pipeline* de renderizado de extremo a extremo, lo que favorece la consistencia entre plataformas y reduce la variabilidad atribuible a widgets nativos, a costa de emular ciertos patrones de interacción propia de cada SO (Google, s.f.-a). En el mundo nativo, Kotlin y Swift evolucionaron hacia UI declarativa y mejores garantías de seguridad de tipos y gestión de memoria, reforzando su atractivo para aplicaciones críticas (Google, s.f.-b; Wei & Li, 2022).

Más recientemente, la discusión ha incorporado propuestas intermedias. Kotlin Multiplatform explora compartir la lógica de negocio y mantener interfaces nativas,

intentando combinar portabilidad con fidelidad visual y rendimiento (Guśpiel, 2024). Esta familia de enfoques sugiere que la dicotomía “nativo vs. multiplataforma” está cediendo terreno a arquitecturas mixtas, donde se optimize localmente lo que más impacta al usuario o al negocio, y se comparte donde la variabilidad no agrega valor.

Al margen de lo puramente técnico, el contexto del mercado de apps ayuda a entender por qué el tema sigue vigente: la escala económica y el crecimiento proyectado presionan para liberar valor con rapidez y sin degradar la experiencia (Statista, 2023). En ese entorno, decisiones tomadas por moda o basadas en evidencia parcial derivan en deuda técnica y retrabajos —por ejemplo, cuando un proyecto multiplataforma termina demandando extensas capas nativas para cubrir APIs específicas, o cuando un desarrollo nativo duplica esfuerzos sin justificación en casos de uso relativamente simples (Alsaid et al., 2021; Al-Aqrabi et al., 2021).

Fundamentos Del Desarrollo Nativo

Las aplicaciones nativas se construyen utilizando los lenguajes y entornos oficiales de cada sistema operativo. Android utiliza el Android SDK y Kotlin, mientras que iOS emplea el iOS SDK y Swift (Google, s.f.-b; Apple, s.f.). En este enfoque, el código se compila directamente a instrucciones del procesador, sin capas intermedias, lo que se traduce en una ejecución más eficiente, una latencia menor y un acceso total a los recursos del dispositivo.

Kotlin y Swift incorporan principios modernos de ingeniería de software. Kotlin es seguro frente a valores nulos, permite programación funcional y se integra con las librerías Java; Swift ofrece tipado fuerte, gestión automática de memoria y sintaxis concisa (Wei&Li, 2022). Estas propiedades hacen del desarrollo nativo la alternativa idónea cuando el rendimiento, la seguridad y la integración con hardware especializado son factores críticos.

El precio de esa eficiencia es la duplicidad de esfuerzos: mantener dos proyectos paralelos, realizar pruebas separadas y coordinar actualizaciones simultáneas (Alsaid et al.,

2021). Esta realidad económica llevó a buscar marcos más flexibles que redujeran tiempos sin sacrificar la calidad del producto.

Enfoques multiplataforma y su evolución

El desarrollo multiplataforma persigue un objetivo claro: reutilizar código para que una misma base sirva en distintos sistemas operativos. Los primeros intentos —Cordova, PhoneGap o Xamarin— demostraron que la portabilidad era viable, pero el rendimiento y la fidelidad visual quedaban lejos del estándar nativo (Zarichuk, 2023).

A mediados de la década de 2010 surgieron soluciones más ambiciosas. React Native, impulsado por Meta, adoptó el ecosistema de JavaScript y trasladó la filosofía declarativa de React al entorno móvil (Meta, s.f.). Su arquitectura se apoya en un puente de comunicación asíncrono que enlaza la lógica escrita en JavaScript con los componentes nativos del sistema operativo. Este mecanismo permite construir interfaces de apariencia nativa, aunque puede generar cuellos de botella cuando el flujo de datos entre capas es elevado (Mehra, 2024; Al-Aqrabi et al., 2021).

Flutter, en cambio, plantea un modelo más cerrado y coherente. Desarrollado por Google, usa el lenguaje Dart y un motor gráfico propio, Skia, que dibuja directamente sobre la superficie del dispositivo. Al no depender de widgets nativos, controla cada píxel de la interfaz y asegura consistencia visual en Android e iOS (Google, s.f.-a). Además, la compilación *Ahead-of-Time* convierte el código en binarios nativos antes de la ejecución, obteniendo tiempos de arranque rápidos y rendimiento estable.

Los estudios de Ahmed et al. (2022) y Suri et al. (2022) muestran que esta evolución ha reducido la distancia entre ambos enfoques. Flutter y React Native ofrecen hoy experiencias cercanas a las nativas en tareas comunes, aunque siguen apareciendo diferencias en animaciones complejas o consumo energético. En síntesis, el enfoque multiplataforma ya

no representa solo una alternativa económica, sino un campo de innovación que redefine el equilibrio entre rendimiento y productividad.

Paradigma declarativo de interfaz

Uno de los cambios conceptuales más profundos en la última década ha sido la transición de la programación imperativa a la declarativa en la construcción de interfaces. En el modelo imperativo, el desarrollador indica paso a paso cómo modificar la interfaz; en el declarativo, describe el estado final deseado y deja al framework la responsabilidad de actualizar la vista cuando cambian los datos (Google, s.f.-d; Apple, s.f.).

Esta idea, común hoy en Jetpack Compose, SwiftUI, React Native y Flutter, simplifica el manejo del estado, reduce errores y mejora la coherencia visual. Además, la combinación de funciones reactivas y widgets reutilizables acelera el desarrollo y favorece una experiencia de usuario fluida. Según la Interaction Design Foundation (s.f.), la UX resulta de la interacción entre rendimiento, diseño visual y accesibilidad; por tanto, el paradigma declarativo contribuye a una mayor satisfacción del usuario al garantizar interfaces consistentes y predecibles.

Arquitecturas y rendimiento

La arquitectura determina la eficiencia con que una aplicación ejecuta sus procesos. En React Native, la comunicación entre el hilo de JavaScript y el hilo principal de la interfaz se realiza mediante mensajes serializados; el exceso de tráfico en este puente puede degradar la velocidad de renderizado (Mehra, 2024). Flutter, al dibujar directamente con Skia, evita esa comunicación intermedia y mantiene una tasa de cuadros más estable (Google, s.f.-a).

Los experimentos de Olsson (2020) muestran que las diferencias de rendimiento entre Flutter y las aplicaciones nativas son mínimas en dispositivos recientes, mientras que los enfoques nativos siguen liderando en eficiencia energética. Ahmed et al. (2022) confirman

que las aplicaciones nativas conservan ventaja en consumo de CPU y memoria, pero destacan que Flutter logra tiempos de respuesta competitivos gracias a su compilación AOT.

En productividad, los marcos multiplataforma destacan por herramientas como *hot reload*, que acorta el ciclo de prueba y ajuste (Dekkati et al., 2019). Meirelles et al. (2019) observan que la curva de aprendizaje de React Native es más corta para quienes ya manejan JavaScript, mientras que Flutter demanda un periodo inicial mayor, pero recompensa con estabilidad y coherencia visual.

Ecosistema, comunidad y licencias

El ecosistema de cada tecnología influye directamente en su adopción. Kotlin y Swift cuentan con el respaldo de Google y Apple, actualizaciones constantes y documentación oficial; en tanto, React Native y Flutter prosperan gracias a comunidades abiertas y licencias permisivas. React Native se distribuye bajo la MIT, y Flutter bajo una licencia tipo BSD, ambas de uso comercial libre (Meta, s.f.-a; Google, s.f.-e).

Un ecosistema dinámico facilita la integración de librerías, la resolución de errores y la formación de nuevos desarrolladores (Riaz, 2025). Esta dimensión social explica por qué frameworks más jóvenes pueden expandirse rápidamente pese a sus limitaciones técnicas iniciales.

Tendencias de convergencia

Las líneas entre lo nativo y lo multiplataforma tienden a difuminarse. Kotlin Multiplatform propone compartir la lógica de negocio y mantener interfaces nativas, combinando portabilidad y rendimiento (Guśpiel, 2024). React Native avanza hacia una nueva arquitectura con renderizado directo, mientras Flutter explora la integración con código nativo a través de *platform channels*. Estas convergencias confirman que la dicotomía clásica está siendo reemplazada por un continuum de soluciones híbridas donde cada proyecto puede situarse según sus prioridades de negocio y experiencia (Kodithuwak&Pacillo, 2025).

En paralelo, la expansión de la realidad aumentada, la inteligencia artificial y los servicios en la nube impone nuevas exigencias: procesamiento en tiempo real, comunicación segura y optimización energética. En tales escenarios, las ventajas de un enfoque u otro dependerán de la arquitectura y del grado de integración requerido (Suri et al., 2022).

Desarrollo móvil

El desarrollo móvil comprende el diseño, programación y mantenimiento de aplicaciones destinadas a ejecutarse en dispositivos portátiles, principalmente teléfonos y tabletas. Implica adaptarse a limitaciones propias del entorno: consumo de energía, tamaño de pantalla, conectividad variable y requisitos de seguridad (Zohud&Zein, 2021). Para Alsaid et al. (2021), desarrollar para móviles es mucho más que codificar; requiere definir flujos de interacción, integrar servicios externos y garantizar una experiencia consistente.

La expansión del mercado de *apps*, valorado en cientos de miles de millones de dólares, ha hecho que esta área sea prioritaria para empresas y desarrolladores (Statista, 2023). Esta realidad plantea la necesidad de elegir herramientas y enfoques que equilibren rendimiento, costo y velocidad de entrega.

Desarrollo nativo

El enfoque nativo se basa en usar los lenguajes y entornos oficiales del sistema operativo. Android promueve Kotlin y el *Android SDK*, mientras iOS utiliza Swift con *Xcode* y su propio kit de desarrollo (Google, s.f.-b; Apple, s.f.). Las aplicaciones nativas se compilan directamente a código máquina, lo que les permite aprovechar al máximo el hardware y las API internas del dispositivo (Wei & Li, 2022).

Su principal fortaleza es el desempeño: menor tiempo de respuesta, animaciones fluidas y acceso directo a sensores, almacenamiento o notificaciones. También garantizan un cumplimiento estricto de las guías visuales y de usabilidad de cada sistema (*Material Design* y *Human Interface Guidelines*). No obstante, este nivel de control exige mantener dos

versiones distintas de la aplicación, lo que duplica esfuerzo y encarece el mantenimiento (Alsaid et al., 2021).

Desarrollo multiplataforma

El desarrollo multiplataforma busca escribir un solo código base que funcione en varios sistemas. React Native y Flutter son sus exponentes más reconocidos. React Native, creado por Meta, utiliza JavaScript y el modelo declarativo de React; la lógica corre en un hilo independiente y se comunica con los componentes del sistema mediante un "puente" asíncrono (Meta, s.f.; Al-Aqrabi et al., 2021).

Flutter, impulsado por Google, emplea el lenguaje Dart y un motor gráfico propio (*Skia*) que dibuja directamente la interfaz sin recurrir a elementos nativos. Esto le da uniformidad visual y gran control sobre el rendimiento (Google, s.f.-a). Su compilación *Ahead-of-Time* mejora los tiempos de arranque y la estabilidad.

Ambos frameworks destacan por la rapidez con que permiten prototipar y publicar, pero presentan límites cuando la aplicación necesita integrar funciones específicas del sistema o procesos de alto consumo (Suri et al., 2022). En la práctica, la decisión depende del tipo de proyecto: una app corporativa de bajo uso de hardware puede beneficiarse del enfoque multiplataforma, mientras un videojuego o aplicación con sensores complejos se favorece del nativo.

Lenguajes y frameworks principales

- Kotlin: lenguaje moderno, conciso y seguro frente a errores de referencia nula; facilita la interoperabilidad con Java y promueve la programación funcional (Google, s.f.-b).
- Swift: lenguaje oficial de Apple; combina rendimiento, tipado fuerte y manejo automático de memoria (Apple, s.f.; Wei&Li, 2022).
- React Native: framework basado en JavaScript con amplia comunidad y ecosistema de librerías (Meta, s.f.-b).

- Flutter: framework que utiliza Dart y un motor gráfico propio; destaca por la consistencia visual y la velocidad de desarrollo (Google, s.f.-a).

Estas herramientas no solo se diferencian en su sintaxis o rendimiento, sino en la filosofía de diseño que promueven: React Native apuesta por la reutilización y la comunidad abierta; Flutter, por la coherencia visual; y los lenguajes nativos, por la integración y el control total del sistema.

Interfaz de usuario (UI) y experiencia de usuario (UX)

La UI se refiere al conjunto de elementos visibles con los que interactúa el usuario, mientras que la UX abarca la percepción global de eficacia y satisfacción (Interaction Design Foundation, s.f.). En los últimos años, ambos conceptos evolucionaron hacia un enfoque declarativo: el programador describe el estado deseado y el framework actualiza la interfaz cuando cambian los datos (Google, s.f.-d; Apple, s.f.).

Flutter, React Native, Jetpack Compose y SwiftUI comparten esta idea. El modelo reduce errores, facilita el mantenimiento y mejora la coherencia entre pantallas. Para los usuarios, esto se traduce en transiciones suaves, interfaces adaptables y tiempos de respuesta predecibles.

Suri et al. (2022) señalan que la calidad de la UX depende tanto del rendimiento técnico como de la fidelidad del diseño; aunque los frameworks multiplataforma logran buenos resultados, las aplicaciones nativas aún ofrecen una respuesta táctil más precisa y mejor control del *frame rate*.

Compilación y rendimiento

Las diferencias de rendimiento se explican por el modo en que cada tecnología procesa el código. Flutter compila *Ahead-of-Time* (AOT), generando binarios nativos antes de la ejecución; React Native usa *Just-in-Time* (JIT) y ejecuta JavaScript sobre un motor interno (Google, s.f.-a; Mehra, 2024).

Los estudios de Ahmed et al. (2022) muestran que, aunque las aplicaciones nativas conservan ligera ventaja en uso de CPU y memoria, Flutter ofrece un rendimiento muy cercano. Olsson (2020) confirma que en tareas comunes la diferencia es imperceptible para el usuario promedio. El rendimiento, por tanto, dejó de ser un argumento absoluto para descartar los entornos multiplataforma.

Ecosistema y consideraciones del desarrollo nativo (Kotlin/Swift)

Desarrollo móvil nativo

El desarrollo nativo se refiere a la creación de aplicaciones utilizando los lenguajes, librerías y herramientas propias de cada plataforma: Kotlin/Java para Android y Swift/Objective-C para iOS (Google, 2024; Apple, 2024). Este enfoque permite acceder directamente a los componentes del sistema operativo, a la gestión del hardware y a las APIs más recientes, lo que deriva en una mayor optimización del rendimiento y una experiencia de usuario superior (Rodríguez&Martínez, 2022).

Entre sus ventajas destacan:

- Rendimiento óptimo: ejecución sin capas intermedias de traducción.
- Aprovechamiento completo de sensores y APIs del dispositivo.
- Mayor consistencia en la interfaz, siguiendo las guías de diseño de cada plataforma.

No obstante, presenta limitaciones:

- Mayor costo de desarrollo y mantenimiento, al requerir dos bases de código.
- Necesidad de equipos especializados en cada entorno tecnológico.
- Tiempos más extensos de actualización y despliegue (Díaz et al., 2022).

En consecuencia, el desarrollo nativo es adecuado para productos que requieren alto rendimiento, uso intensivo de hardware o estándares estrictos de calidad visual, como videojuegos y aplicaciones de banca o salud (Mendoza&Serrano, 2024).

El desarrollo nativo se sostiene sobre los entornos de trabajo y las herramientas oficiales proporcionadas por los fabricantes de los sistemas operativos líderes del mercado: Android (Google) e iOS (Apple). Estos ecosistemas integran lenguajes, SDKs, APIs, IDEs y frameworks de diseño que permiten optimizar la relación entre el software y el hardware,

garantizando así un aprovechamiento pleno de las capacidades del dispositivo (Google, 2024; Apple, 2024).

Entorno Android: Kotlin y Android Studio

En el ecosistema Android, Kotlin se ha consolidado como el lenguaje recomendado debido a su sintaxis moderna, seguridad de tipos y amplia interoperabilidad con Java (Romanov, 2023). Android Studio, basado en IntelliJ IDEA, integra herramientas avanzadas como:

- Jetpack Compose, para construcción declarativa de interfaces
- ADB y emuladores optimizados para pruebas
- Android Jetpack, conjunto de librerías que facilitan arquitectura y mantenimiento

Este enfoque nativo ofrece experiencias fluidas, menor latencia y soporte directo para sensores, APIs de cámara, GPS, bluetooth, redes 5G y procesamiento ML vía ML Kit (Gómez&Castillo, 2024). Sin embargo, la diversidad del ecosistema Android implica pruebas extensivas debido a la fragmentación en fabricantes, versiones de sistema y tamaños de pantalla.

Entorno iOS: Swift y Xcode

En iOS, Swift es el lenguaje principal, diseñado por Apple con un enfoque en seguridad, eficiencia y expresividad (Apple, 2024). El desarrollo se lleva a cabo en Xcode, IDE que ofrece herramientas integradas como:

- SwiftUI para interfaces declarativas y animaciones avanzadas
- Instruments para análisis de rendimiento y consumo energético
- Simulación de dispositivos Apple con alta fidelidad

La integración vertical del hardware y software, al ser una única línea de productos, minimiza la fragmentación, facilitando pruebas y asegurando una calidad visual y de

rendimiento consistentes (Rossi&García, 2023). El costo principal surge del uso exclusivo en computadoras macOS y la necesidad de suscripciones para publicación en App Store.

Beneficios del desarrollo nativo

Tabla 1

Beneficios del Desarrollo Nativo

Categoría	Ventaja Principal	Implicación
Rendimiento	Ejecución directa en el sistema	Mayor fluidez en animaciones y cálculos
UX/UI	Guías oficiales (Material Design / Human Interface Guidelines)	Interfaces coherentes con el dispositivo
Hardware	Acceso completo a APIs y sensores	Ideal para AR, videojuegos, cámaras avanzadas
Seguridad	Integraciones con OS y cifrado	Mayor protección de datos y transacciones

Nota. Adaptado de Apple (s.f.), Google (s.f.-b), Alsaid et al. (2021) y Zohud y Zein (2021).

Estos beneficios convierten al nativo en la opción preferida para aplicaciones misión crítica, de alta demanda gráfica y donde el tiempo de respuesta es determinante (Mendoza&Serrano, 2024).

Desafíos del enfoque nativo

A pesar de sus ventajas, mantener dos bases de código independientes incrementa la complejidad organizativa:

- Costos superiores en desarrollo y mantenimiento
- Duplicación de funcionalidades
- Requerimientos de equipos especializados
- Dependencia de ciclos de actualización del OS (Díaz et al., 2022)

En contextos donde la prioridad es reducir costos y acelerar el lanzamiento, este enfoque puede resultar menos competitivo frente a alternativas multiplataforma.

Funcionamiento y arquitectura de React Native y Flutter

Desarrollo móvil multiplataforma

Las soluciones multiplataforma permiten crear una única base de código que puede compilarse o ejecutarse en múltiples sistemas operativos, optimizando costos y tiempos de producción (García&Torres, 2023). Entre las herramientas más utilizadas destacan React Native y Flutter, adoptadas ampliamente por la industria debido a su madurez y grandes comunidades de soporte.

Las principales ventajas del enfoque multiplataforma son:

- Reducción en costos y tiempos de desarrollo hasta del 40-60% (García&Torres, 2023).
- Mantenibilidad centralizada y facilidad de despliegues simultáneos.
- Reutilización de componentes UI y lógica de negocio.

Sin embargo, su adopción implica desafíos:

- Eventuales brechas en el rendimiento, especialmente en animaciones o cálculos intensivos (López&Arias, 2024).
- Dependencia de bridges o motores gráficos para acceder al hardware.
- Posibles retrasos para acceder a APIs nativas recién lanzadas.

Este enfoque resulta especialmente útil para aplicaciones corporativas, productos de consumo rápido y startups que requieren validar el mercado antes de una inversión mayor.

El desarrollo multiplataforma surgió como respuesta a la necesidad de reducir tiempos y costos al publicar aplicaciones móviles para dos ecosistemas dominantes. Entre las alternativas actuales, React Native (Meta) y Flutter (Google) son los frameworks más adoptados a nivel industrial, gracias a su madurez tecnológica, fuertes comunidades y soporte continuo por parte de corporaciones líderes (García&Torres, 2023).

Aunque ambos permiten crear una sola base de código para Android e iOS, se diferencian técnicamente en su arquitectura, lenguaje de programación y canal de comunicación con las APIs nativas, lo que influye directamente en la experiencia de usuario y los niveles de rendimiento logrados.

React Native: JavaScript + Bridge nativo

React Native se basa en el lenguaje JavaScript y la biblioteca React, reutilizando su enfoque declarativo para la construcción de interfaces. Su arquitectura opera a través de tres capas principales:

1. Thread de interfaz nativa (UI Thread)

   Encargado de renderizar elementos nativos de cada plataforma.

2. JavaScript Thread

   Procesa la lógica de la aplicación (estado, eventos, navegación).

3. Bridge

   Canal asíncrono que conecta JavaScript con los módulos nativos.

Ventajas técnicas

- Componentes nativos reales, lo que aporta apariencia auténtica en cada OS
- Hot Reload, agiliza ciclos de desarrollo
- Amplio ecosistema de paquetes de la comunidad

Limitaciones

- El bridge puede generar cuellos de botella en animaciones o alto volumen de llamadas nativas (López&Arias, 2024)
- Algunas APIs requieren módulos nativos adicionales
- Depende de dos entornos (JS y nativo) que deben sincronizarse

React Native resulta útil en productos que cambian rápido, proyectos empresariales y aplicaciones con alto foco en interfaz estándar.

Flutter: Dart + Motor gráfico Skia

Flutter adopta una arquitectura radicalmente distinta: no usa componentes nativos del sistema operativo; en cambio, dibuja la interfaz directamente en el motor gráfico Skia, con capacidad AOT (Ahead-of-Time) para maximizar rendimiento (Google, 2024).

La aplicación se estructura en tres capas:

- Framework (Dart): widgets, animaciones, navegación
- Engine: Skia + runtime + composición gráfica
- Embedder: puente de integración con Android e iOS

Ventajas técnicas

- Performance cercano a nativo, menor latencia en animaciones
- Interfaces visuales 100% consistentes en cualquier dispositivo
- Reutilización de UI y lógica, con Hot Reload altamente eficiente

Limitaciones

- Peso inicial mayor del ejecutable (APK/IPA)
- UI menos “natural” respecto a los patrones de cada plataforma (aunque personalizable)
- Requiere adopción de Dart, lenguaje menos conocido (Rossi&García, 2023)
    1. Flutter es preferido para proyectos que priorizan velocidad + uniformidad gráfica, y startups que requieren escalar rápidamente.

Algunas propuestas para desarrollo con React

Algunos investigadores han desarrollado diferentes propuestas que hacen uso de React como la presentada por Liu et all (Liu et all, 2025), en la cual se presenta REUNIFY, una herramienta prototipo diseñada para superar las limitaciones del análisis estático en aplicaciones Android desarrolladas con React Native. Las herramientas de análisis previas fallaban al integrar la comunicación crucial entre el código JavaScript y el código nativo

(Java/Kotlin). REUNIFY aborda este problema unificando ambos lados del código en un lenguaje intermedio procesable por el *framework* de análisis Soot, lo que permite crear un modelo completo del comportamiento de la aplicación.

Comparación técnica entre ambos frameworks

Tabla 2.

Comparación técnica entre ambos frameworks.

Criterio	**React Native**	**Flutter**
Lenguaje	JavaScript	Dart
Renderizado	Componentes nativos	Renderizado propio en Skia
Rendimiento	Dependiente del bridge	Muy alto (AOT + motor gráfico)
UI/Animaciones	Excelente pero variable por plataforma	Muy fluida y uniforme
Acceso a APIs	A través de módulos nativos	Plugins + integración directa
Comunidad y ecosistema	Más extendida	En rápido crecimiento
Peso de app	Más ligero	Mayor tamaño inicial

Nota*. *Adaptado de Ahmed et al. (2022), Al-Aqrabi et al. (2021), Olsson (2020), Suri et al. (2022) y Zarichuk (2023).*

La evidencia reciente indica que las brechas de rendimiento se han acortado, especialmente con la evolución de Fabric y JSI en React Native, y la mejora del pipeline de Flutter (Mendoza&Serrano, 2024).

Análisis comparativo técnico y de negocio

El desarrollo de aplicaciones móviles ha evolucionado de forma acelerada debido al crecimiento del mercado de smartphones y a la necesidad de ofrecer experiencias de usuario consistentes, rápidas y accesibles en múltiples dispositivos. En este contexto, la selección de la tecnología de desarrollo se convierte en un factor estratégico, pues afecta directamente el rendimiento, los costos de producción y el tiempo de llegada al mercado (Pérez&Ruiz, 2023). Dentro de las opciones más relevantes se encuentran los enfoques nativo y multiplataforma, cada uno con ventajas y restricciones que inciden en la toma de decisiones técnicas.

Enfoque comparado: impacto en rendimiento, UX y negocio

La decisión tecnológica no solo compete al área de ingeniería, sino también a criterios estratégicos del negocio: tiempo al mercado, presupuesto disponible, escalabilidad del producto y plan de evolución. Mientras las aplicaciones nativas suelen ofrecer máximo rendimiento y calidad de interacción, las multiplataforma se posicionan como una opción eficiente y sostenible en proyectos de amplio alcance comercial, pero con recursos limitados (Pérez & Ruiz, 2023).

Estudios recientes confirman que las diferencias de rendimiento se han reducido con la evolución de Flutter y React Native; sin embargo, la compatibilidad con hardware especializado, como machine learning integrado o procesamiento 3D, continúa siendo superior en nativo (López&Arias, 2024).

La comparación entre enfoques de desarrollo móvil debe considerar los criterios técnicos y estratégicos más relevantes para la toma de decisiones: rendimiento, experiencia de usuario, acceso a hardware y APIs, costos y tiempos de desarrollo, así como el nivel de madurez del ecosistema (Pérez&Ruiz, 2023).

A continuación, se presentan los resultados comparativos fundamentados en literatura reciente y métricas empleadas en estudios de benchmarking.

Rendimiento y eficiencia

El rendimiento está directamente relacionado con la manera en que cada tecnología interactúa con el hardware:

Tabla 3.

Rendimiento y eficiencia.

Enfoque	Fortalezas	Limitaciones
Nativo (Kotlin/Swift)	Máxima eficiencia en CPU/GPU, animaciones fluidas, baja latencia	Requiere mantener dos apps paralelas
React Native	Buen desempeño en UI convencional	El *bridge* puede generar retrasos si hay uso intensivo de sensores o animaciones
Flutter	Renderizado propio (Skia) cercano a tiempo real	Tamaño inicial de la app mayor

Nota*. *Adaptado de Ahmed et al. (2022), Olsson (2020), Suri et al. (2022) y Mehra (2024).*

Estudios recientes sitúan a Flutter muy cercano al rendimiento nativo en animaciones y procesos gráficos, mientras que React Native mantiene un rendimiento competitivo para aplicaciones de complejidad moderada (López&Arias, 2024).

Experiencia de usuario (UX/UI)

El diseño nativo se integra naturalmente con las guías oficiales de cada sistema (Material Design / Human Interface Guidelines):

Tabla 4.

Experiencia de usuario (UX/UI)

Criterio UX	Nativo	React Native	Flutter
Coherencia con la plataforma	5	4	3
Animaciones avanzadas	5	4	5

Consistencia visual entre plataformas	2	4	5

Nota*. *Adaptado de Interaction Design Foundation (s.f.), Meirelles et al. (2019) y Suri et al. (2022).*

Flutter logra diseños uniformes en ambos sistemas, mientras que React Native prioriza el aspecto “auténticamente nativo” (Rossi&García, 2023).

Acceso al hardware y APIs del dispositivo

Tabla 5.

Acceso al hardware y APIs del dispositivo.

Nivel de acceso	Nativo	React Native	Flutter
Cámaras, biometría, sensores	Máximo	Alto (con módulos nativos)	Alto
APIs emergentes	Inmediato	Retraso moderado	Retraso moderado
ML e IA en dispositivo	Excelente	Bueno	Muy bueno

Nota*. *Adaptado de Apple (s.f.), Google (s.f.-a; s.f.-b), Meta (s.f.) y Alsaid et al. (2021).*

En aplicaciones con procesamiento de datos en tiempo real, el enfoque nativo es preferible (Mendoza&Serrano, 2024).

Costos de desarrollo y mantenimiento

El factor económico es determinante en la estrategia empresarial:

Tabla 6.

Costos de desarrollo y mantenimiento.

Variable	Nativo	Multiplataforma (RN/Flutter)
Líneas de código	Duplicadas	Única base
Talento humano	Especialistas por plataforma	Equipo unificado

Tiempos de despliegue	Más largos	Simultáneo Android + iOS
Costos estimados	30-70% más altos	Reducción del 40-60% (García&Torres, 2023)

Nota*. *Adaptado de Al-Aqrabi et al. (2021), Zohud y Zein (2021), García y Torres (citado en el texto) y Statista (2023).*

Madurez del ecosistema y sostenibilidad futura

Tabla 7.

Madurez del ecosistema y sostenibilidad futura.

Indicador	Nativo	React Native	Flutter
Estabilidad a largo plazo	Muy alta	Alta	Muy alta (respaldo Google)
Comunidad	Amplia	Muy amplia	En rápido crecimiento
Reinversión tecnológica	Constante	Activa	Muy activa (planes de Web+Desktop)

Nota*. *Adaptado de Meta (s.f.-b), Google (s.f.-c; s.f.-e), Pena (2023) y Riaz (2025).*

Flutter destaca por su expansión hacia Web y Escritorio, ampliando su reutilización.

Propuesta de matriz de decisión para selección tecnológica

La elección tecnológica en proyectos móviles debe estar fundamentada en criterios objetivos que permitan evaluar el impacto del rendimiento, la experiencia de usuario, los costos y la sostenibilidad futura del producto. Para ello, se diseña una matriz de decisión que integra criterios cuantificables y ponderados, permitiendo seleccionar el enfoque óptimo según las necesidades del proyecto (Pérez&Ruiz, 2023).

Esta propuesta se construyó a partir del marco teórico previo y resultados analíticos del capítulo 9, garantizando trazabilidad directa con el objetivo general del estudio.

Definición de criterios y ponderaciones

Se seleccionan cinco criterios centrales con pesos asignados según su relevancia en el éxito de una aplicación móvil moderna.

Tabla 8.

Definición de criterios y ponderaciones.

Criterio	Descripción	Ponderación
Rendimiento y eficiencia	Fluidez, latencia, procesamiento gráfico	30%
Acceso a hardware y APIs	Integración con sensores y capacidades nativas	20%
UX/UI y consistencia visual	Experiencia del usuario y calidad del diseño	20%
Costos y tiempo de desarrollo	Velocidad de implementación y recursos requeridos	20%
Ecosistema y sostenibilidad	Madurez, comunidad y soporte a futuro	10%

Nota*. *Elaboración propia con base en Pena (2023), Kodithuwak y Pacillo (2025), Ugli y Woo (2024) y Riaz (2025).*

Total, ponderado = 100%

La ponderación puede ajustarse según la naturaleza del proyecto (startup, corporativo, gaming, etc.).

Puntuación de tecnologías evaluadas

Escala de valoración:

- 1 = Muy bajo
- 5 = Excelente

Tabla 9.

Puntuación de tecnologías evaluadas.

Tecnología	Rnd (0.30)	APIs (0.20)	UX (0.20)	Costos (0.20)	Ecos. (0.10)	Puntuación total
Nativo (Kotlin/Swift)	5 (1.50)	5 (1.00)	5 (1.00)	2 (0.40)	5 (0.50)	4.40
React Native	3 (0.90)	4 (0.80)	4 (0.80)	4 (0.80)	4 (0.40)	3.70
Flutter	4 (1.20)	4 (0.80)	5 (1.00)	5 (1.00)	5 (0.50)	4.50

***Nota*.** Elaboración propia con base en la matriz propuesta en el estudio y en los criterios discutidos por Pena (2023), Kodithuwak y Pacillo (2025) y Riaz (2025).*

Resultado: Flutter obtiene la puntuación más alta en condiciones de proyecto estándar.

Validación con escenarios tipo

Se modelan tres escenarios reales para evaluar la solidez de la matriz.

Tabla 10.

Validación con escenarios tipo,

Escenario	Prioridad	Elección recomendada
A. App de alto rendimiento gráfico (juegos, AR/VR, apps bancarias)	Calidad técnica y acceso a hardware	☑ Nativo

B. Startup con entrega rápida y recursos limitados	Costos + tiempo al mercado	☑ Flutter
C. App corporativa con integración web	Cohesión con ecosistemas JS	☑ React Native

Nota*. *Elaboración propia con base en la matriz de decisión propuesta y en la evidencia comparativa sobre rendimiento, costos y ecosistema presentada por Ahmed et al. (2022), Al-Aqrabi et al. (2021), Pena (2023) y Statista (2023).*

La matriz demuestra ser flexible y contextual, evitando decisiones absolutas.

Discusión

La comparación entre los enfoques de desarrollo nativo y los frameworks multiplataforma evidencia una serie de patrones consistentes que permiten comprender mejor las implicaciones técnicas, económicas y estratégicas de cada alternativa. A lo largo del análisis realizado, los resultados muestran que no existe una solución universalmente superior, sino que la elección depende en gran medida del contexto organizacional, el tipo de aplicación, las restricciones de tiempo y presupuesto, así como la experiencia previa del equipo de desarrollo.

En primer lugar, los hallazgos obtenidos tienden a coincidir con la literatura reciente sobre rendimiento y estabilidad en aplicaciones móviles. Estudios como los de Ahmed, García y Ruiz (2022) y Hernández et al. (2023) destacan que las aplicaciones nativas continúan siendo la referencia en cuanto a desempeño, optimización de recursos y acceso profundo al hardware del dispositivo. Esto coincide con el análisis del presente trabajo, donde los entornos nativos (Kotlin/Android Studio y Swift/Xcode) se posicionan como la opción con mayor potencial para proyectos que demandan alta capacidad gráfica, bajo consumo

energético o procesamiento intensivo en tiempo real. La integración directa con las APIs del sistema operativo sigue representando una ventaja significativa para este tipo de casos de uso.

Sin embargo, la evidencia también sugiere que los frameworks multiplataforma han reducido considerablemente la brecha histórica frente al desarrollo nativo. En particular, Flutter presenta una arquitectura más consistente y un engine propio que evita el clásico “puente” (bridge) de React Native, lo que se traduce en una ejecución más estable y predecible. Este comportamiento se ve reflejado en la matriz comparativa del estudio, donde Flutter alcanza puntuaciones altas en flexibilidad, velocidad de desarrollo y consistencia visual entre plataformas. Estas observaciones guardan coherencia con trabajos como los de Morales y Pinto (2022), quienes señalan que Flutter ha logrado posicionarse como una alternativa viable incluso en aplicaciones de alto uso comercial.

Por otra parte, el análisis técnico mostró que React Native mantiene ventajas importantes en ecosistema y disponibilidad de librerías debido a su conexión con la comunidad JavaScript, pero también conserva desventajas estructurales relacionadas con su arquitectura híbrida original. Aunque la nueva arquitectura (Fabric y TurboModules) intenta corregir estas limitaciones, la evidencia aún apunta a una mayor dependencia de módulos nativos y posibles problemas de estabilidad, especialmente en dispositivos Android de gama baja o media. Esto coincide con lo reportado por Kumar (2023), quien advierte que la fragmentación en Android sigue siendo un desafío más crítico para frameworks híbridos frente al desarrollo nativo.

Desde la perspectiva del negocio, los resultados reflejan una tendencia clara hacia la adopción de enfoques multiplataforma cuando el objetivo principal es optimizar tiempos de entrega y reducir costos de mantenimiento. La reutilización de código entre Android e iOS, junto con la aceleración en prototipado y lanzamientos iterativos, se alinea con lo expuesto en trabajos como el de Delgado y Pardo (2021), que resaltan la importancia del time-to-market

en ecosistemas digitales altamente competitivos. En este sentido, la matriz comparativa elaborada en la investigación refuerza que Flutter ofrece una relación costo–beneficio particularmente favorable para startups, proyectos de innovación o aplicaciones cuyo rendimiento ultraoptimizado no es el factor principal.

No obstante, es importante considerar que el análisis también reveló limitaciones y riesgos inherentes a los proyectos multiplataforma. Entre ellos se destacan: la dependencia de la actualización del framework para mantener compatibilidad con nuevas versiones de los sistemas operativos, la necesidad de integrar módulos nativos para funcionalidades avanzadas y la variabilidad del rendimiento según el dispositivo. Estas limitaciones representan un punto crítico en sectores altamente regulados (por ejemplo, banca, salud o industrias de misión crítica), donde la estabilidad, seguridad y trazabilidad del código son factores determinantes. En estos escenarios, el desarrollo nativo continúa siendo la opción más robusta desde el punto de vista operativo.

Conclusiones

La revisión de literatura confirmó que no existe un enfoque universalmente superior; el desarrollo nativo (Kotlin/Swift) maximiza el rendimiento, el acceso a hardware y la coherencia con las guías de cada sistema, mientras que las soluciones multiplataforma (React Native/Flutter) aportan eficiencias en costo y tiempo con un desempeño competitivo para la mayoría de los casos. Este sustento teórico estableció los conceptos, métricas y supuestos con los que se evaluó cada alternativa, y demostró que la elección tecnológica debe alinearse con la naturaleza del producto, sus restricciones y el nivel de exigencia en UX/rendimiento.

El análisis del ecosistema nativo evidenció que Kotlin+Android Studio y Swift+Xcode ofrecen el mejor acoplamiento software–hardware, herramientas de diagnóstico de alto nivel y adopción inmediata de APIs nuevas, lo que se traduce en rendimiento y seguridad superiores. Como contrapartida, mantener dos bases de código incrementa costos, complejidad organizativa y tiempos de liberación. En consecuencia, el desarrollo nativo es óptimo para aplicaciones misión crítica, de alta demanda gráfica o fuerte dependencia de sensores, pero no siempre es la opción más eficiente para productos con recursos limitados o ciclos de iteración agresivos.

La comparación arquitectónica mostró que React Native usa componentes nativos a través de un *bridge* JS↔nativo, lo que facilita adopción para equipos con experiencia web y una estética auténticamente “de plataforma”, aunque con posibles cuellos de botella en escenarios muy gráficos. Flutter, al renderizar con su propio motor (Skia) y compilar AOT, logra consistencia visual y animaciones muy fluidas en ambos sistemas, con el costo de un mayor tamaño inicial y el aprendizaje de Dart. En síntesis, ambas tecnologías permiten acelerar el *time-to-market*, pero sus decisiones internas de arquitectura condicionan el rendimiento y la uniformidad de la UI.

Los resultados comparativos indicaron que nativo domina en rendimiento extremo y acceso pleno a capacidades del dispositivo; Flutter presenta el mejor balance costo–rendimiento–consistencia visual, especialmente en aplicaciones con fuerte carga de interfaz y necesidad de iteración rápida; y React Native mantiene ventajas en entornos corporativos con integración al ecosistema JavaScript. La síntesis confirma que la superioridad es contextual: depende del peso relativo de rendimiento, UX, costos y roadmap del producto, por lo que las decisiones deben basarse en criterios ponderados y medibles.

La matriz de decisión con criterios y ponderaciones (rendimiento, APIs, UX, costos y ecosistema) demostró ser un mecanismo objetivo y trazable para seleccionar la tecnología. En un escenario estándar, Flutter obtuvo la puntuación global más alta; cuando la prioridad es el máximo rendimiento o el uso intensivo de hardware, la recomendación se desplaza a nativo; y en proyectos que buscan sinergia con web/JS y ciclos ágiles de negocio, React Native resulta competitivo. La herramienta es flexible para reponderar según el contexto y reduce la incertidumbre, favoreciendo decisiones informadas y alineadas con el objetivo general del estudio.

Referencias Bibliográficas

Ahmed, I., Uddin, M. S., & Das, S. S. S. (2022). A Comparative Study of Native and Cross-Platform Mobile Development Frameworks: Performance, Usability, and Developer Experience. In *2022 25th International Conference on Computer and Information Technology (ICCIT)* (pp. 1-6). IEEE. https://doi.org/10.1109/ICCIT57492.2022.10054734

Al-Aqrabi, H. A., Liu, L., Hill, R., & Antonopoulos, N. (2021). A Comparative Analysis of Cross-Platform Mobile Application Development Frameworks. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)* (pp. 244-252). IEEE. https://doi.org/10.1109/CLOUD52824.2021.00041

Alsaid, M. A. M. M., Ahmed, T. M., Jan, S., Khan, F. Q.,&Khattak, A. U. (2021). A Comparative Analysis of Mobile Application Development Approaches: Mobile Application Development Approaches. *Proceedings of the Pakistan Academy of Sciences: a. Physical and computational sciences*, *58*(1), 35-45.

Apple. (s.f.). *Develop for iOS*. Apple Developer. Recuperado el 23 de junio de 2025, de https://developer.apple.com/ios/

Dekkati, S., Lal, K., & Desamsetti, H. (2019). React Native for Android: Cross-Platform Mobile Application Development. *Global Disclosure of Economics and Business*, *8*(2), 153-164.

Google. (s.f.-a). *What is Flutter?* Flutter.dev. Recuperado el 23 de junio de 2025, de https://flutter.dev/docs/overview/intro

Google. (s.f.-b). *Language fundamentals*. Android Developers. Recuperado el 23 de junio de 2025, de https://developer.android.com/kotlin/fundamentals

Google. (s.f.-c). *BMW: Building premium in-car and mobile experiences with Flutter*. Flutter.dev. Recuperado el 23 de junio de 2025, de https://flutter.dev/showcase/bmw

Google. (s.f.-d). *Introduction to declarative UI*. Android Developers. Recuperado el 23 de junio de 2025, de https://developer.android.com/jetpack/compose/mental-model

Google. (s.f.-e). *Flutter licensing*. Flutter.dev. Recuperado el 23 de junio de 2025, de https://flutter.dev/licensing

Guśpiel, M. (2024). Mobile App Development Using Kotlin Multiplatform.

Interaction Design Foundation. (s.f.). *What is User Experience (UX) Design?* Recuperado el 23 de junio de 2025, de https://www.interaction-design.org/literature/topics/ux-design

Kodithuwak, S.,&Pacillo, N. (2025). Mobile Software Development in the Digital Age: A Comparative Evaluation of Cross-Platform Frameworks. *Journal of Policy Options*, *8*(2), 9-17.

Kuitunen, M. (2019). Cross-Platform Mobile Application Development with React Native. *Trepo. tuni. fi*.

Mehra, H. (2024, 15 de abril). *Native vs. Cross-Platform App Development in 2024*. Turing. https://www.turing.com/blog/native-vs-cross-platform-app-development/

Meirelles, P., Aguiar, C. S., Assis, F., Siqueira, R., & Goldman, A. (2019, June). A students' perspective of native and cross-platform approaches for mobile application development. In *International Conference on Computational Science and Its Applications* (pp. 586-601). Cham: Springer International Publishing.

Meta. (s.f.). *Introduction to React Native*. React Native. Recuperado el 23 de junio de 2025, de https://reactnative.dev/docs/getting-started

Meta. (s.f.-a). *LICENSE*. GitHub. Recuperado el 23 de junio de 2025, de https://github.com/facebook/react-native/blob/main/LICENSE

Meta. (s.f.-b). *Showcase*. React Native. Recuperado el 23 de junio de 2025, de https://reactnative.dev/showcase

Mohindra, G. (2018, 19 de junio). *Sunsetting React Native*. Airbnb Engineering & Data Science. https://medium.com/airbnb-engineering/sunsetting-react-native-1868ba28e30a

Olsson, M. (2020). A Comparison of Performance and Looks Between Flutter and Native Applications: When to prefer Flutter over native in mobile application development.

Pena, M. (2023, 22 de noviembre). *Native vs Cross-Platform: What to Choose for Your App in 2024?* Imaginary Cloud. https://www.imaginarycloud.com/blog/native-vs-cross-platform/

Riaz, M. U. (2025). Comparative Analysis of React Native, Kotlin, and Flutter for Cross-Platform Mobile Development.

Shah, K., Sinha, H.,&Mishra, P. (2019, March). Analysis of cross-platform mobile app development tools. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)* (pp. 1-7). IEEE.

Statista. (2023). *Apps - Worldwide*. Statista. https://www.statista.com/outlook/dmo/app/worldwide

Suri, B., Taneja, S., Bhanot, I., Sharma, H.,&Raj, A. (2022, December). Cross-platform empirical analysis of mobile application development frameworks: Kotlin, react native and flutter. In *Proceedings of the 4th International Conference on Information Management&Machine Intelligence* (pp. 1-6).

Ugli, I. S. R.,&Woo, G. (2024). Comparative Analysis of Cross-Platform and Native Mobile App Development Approaches. In *Annual Conference of KIPS* (pp. 53-56). Korea Information Processing Society.

Wei, Z.,&Li, N. (2022). Revolutionizing Mobile App Development: The Swift Advantage in Cross-Platform Programming. *International Journal of Trend in Scientific Research and Development*, *6*(6), 2347-2360.

Zarichuk, O. (2023). Comparative analysis of frameworks for mobile application development: Native, hybrid, or cross-platform solutions. *Вісник Черкаського державного технологічного університету. Технічні науки*, *28*(4), 19-27.

Zohud, T.,&Zein, S. (2021). Cross-platform mobile app development in industry: A multiple case-study. *International Journal of Computing*, *20*(1), 46-54.