

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA
CONCEPCIÓN - CHILE



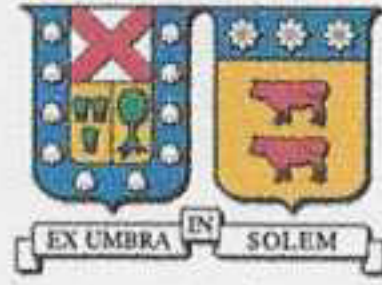
**“Arquitectura de software móvil resiliente basada en
el paradigma *Offline-First* y el patrón *Store-and-Forward*”**

JORGE IGNACIO FARÍAS SOTO

**MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO EN INFORMÁTICA**

Profesor Guía: José Luis Carrasco Sáez

Diciembre - 2025



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: "Arquitectura de software móvil resiliente basada en el paradigma *Offline-First* y el patrón *Store-and-Forward*"

Nombre del candidato(a): Jorge Ignacio Farias Soto

Carrera / Grado: Ingeniería en Informática

Campus: Sede Concepción Departamento: Electrónica e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, José Luis Carrasco Sáez, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (marcar una opción):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 26-02-26 Firma: 

Estudiante o Candidato(a):

Fecha: 25/02/2026 Firma: 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

DEDICATORIA

A mis amigos y compañeros de ruta, con quienes compartí no solo el aula, sino también los miedos y las alegrías de esta carrera. Gracias por ser el apoyo necesario en el camino a convertirnos en colegas.

AGRADECIMIENTOS

En primer lugar, agradezco a la Universidad Técnica Federico Santa María por haberme brindado las herramientas necesarias para mi formación profesional y por ser mi segundo hogar durante estos años.

Un agradecimiento especial a mi profesor guía, José Luis Carrasco Sáez, por su orientación, paciencia y valiosos consejos que fueron fundamentales para la culminación de esta memoria.

A mis amigos y compañeros de carrera, con quienes compartí largas noches de estudio, risas y desafíos. Gracias por hacer de este camino una experiencia inolvidable.

Finalmente, todos aquellos que, de una u otra forma, contribuyeron a la realización de este sueño.

RESUMEN

Esta memoria presenta el diseño e implementación de una arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward. El trabajo aborda la problemática de la pérdida de datos y la interrupción operativa en aplicaciones de terreno con conectividad inestable. Utilizando un stack moderno compuesto por React Native, Expo, SQLite con interfaz JSI y BetterAuth, se desarrolló un mecanismo de sincronización asíncrona que garantiza la integridad de los datos y una experiencia de usuario fluida. Los resultados obtenidos en entornos de simulación Android demostraron una mejora de hasta 7.8x en el rendimiento de persistencia local y una reducción del 98% en el overhead de red mediante técnicas de batching. La solución valida la viabilidad de utilizar la "Nueva Arquitectura" de React Native para aplicaciones empresariales críticas.

Palabras Clave: *Offline-First; React Native JSI; Sincronización Móvil; SQLite; Resiliencia de Software.*

ABSTRACT

This thesis presents the design and implementation of a resilient mobile software architecture based on the Offline-First paradigm and the Store-and-Forward pattern. The work addresses the issues of data loss and operational interruption in field applications with unstable connectivity. Using a modern stack consisting of React Native, Expo, SQLite with JSI interface, and BetterAuth, an asynchronous synchronization mechanism was developed to ensure data integrity and a smooth user experience. Results obtained in Android simulation environments demonstrated up to a 7.8x improvement in local persistence performance and a 98% reduction in network overhead through batching techniques. The solution validates the feasibility of using React Native's "New Architecture" for critical enterprise applications.

Keywords: *Offline-First; React Native JSI; Mobile Synchronization; SQLite; Software Resilience.*

GLOSARIO

ACID: Atomicidad, Consistencia, Aislamiento y Durabilidad.

API: Interfaz de Programación de Aplicaciones.

AVD: Dispositivo Virtual de Android (*Android Virtual Device*).

FIFO: Primero en entrar, primero en salir (*First-In, First-Out*).

FPS: Cuadros por segundo (*Frames Per Second*).

JSI: Interfaz de JavaScript (*JavaScript Interface*).

JWT: *Token* de Web de JSON.

PWA: Aplicación Web Progresiva (*Progressive Web App*).

UTFSM: Universidad Técnica Federico Santa María.

WAL: Registro de escritura anticipada (*Write-Ahead Logging*).

INDICE DE CONTENIDOS

RESUMEN	4
ABSTRACT	4
INDICE DE FIGURAS	9
INDICE DE TABLAS	10
INTRODUCCIÓN	11
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	12
1.1 Contexto	12
1.2 Situación actual.....	12
1.3 Actores involucrados.....	13
1.4 Dificultades actuales	13
1.5 Consecuencias de no resolver estas dificultades	14
1.6 Competencia.....	14
1.7 Objetivo general	15
1.8 Objetivos específicos.....	15
CAPÍTULO 2: MARCO CONCEPTUAL.....	16
2.1 Introducción	16
2.2 Computación Móvil Desconectada	16
2.2.1 Del Paradigma Cloud-First al Local-First	16
2.2.2 El Teorema CAP y la Elección de Disponibilidad.....	17
2.2.3 Patrón Arquitectónico Store-and-Forward	17
2.3 Frameworks de Desarrollo Móvil.....	18
2.4 Entorno de Ejecución y Construcción	20
2.5 Persistencia de Datos Local	21
2.6 Gestión de Identidad y Seguridad.....	23

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward	
2.7 Estrategia de Sincronización e Integridad de Datos	25
2.8 Conclusión del Capítulo	26
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	28
3.1 Introducción	28
3.2 Arquitectura de Software: Enfoque Feature-Based	28
3.3 Estrategia de Persistencia y Sincronización	30
3.4 Diseño del Algoritmo de Sincronización	30
3.4.1 Fase de Captura y Encolado	31
3.4.2 Fase de Detección y Estabilización de Red (Debounce).....	31
3.4.3 Fase de Procesamiento y Reintentos	31
3.4.4 Manejo Especial de Errores de Autenticación (401 Unauthorized):	32
3.5 Justificación Tecnológica: Nativo vs PWA	32
3.5.1 Fiabilidad en la Ejecución en Segundo Plano	33
3.5.2 Integridad y Persistencia de Datos	33
3.5.3 Seguridad de la Sesión y Cifrado	33
3.6 Diagramas	34
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	38
4.1 Introducción	38
4.2 Configuración del Entorno Experimental	38
4.3 Validación del Rendimiento de Persistencia Local	39
4.4 Validación de Sincronización y Consistencia	39
4.5 Validación de Eficiencia de Recursos y WorkManager	40
4.6 Validación de Usabilidad y UI Optimista	40
4.7 Obtención de Métricas de Rendimiento (Estándar vs. Propuesta)	41
4.8 Realismo del Entorno de Prueba	41
4.9 Conclusión	41

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón
Store-and-Forward

CAPÍTULO 5: CONCLUSIONES	43
5.1 Síntesis del Trabajo Realizado	43
5.2 Cumplimiento de Objetivos.....	43
5.3 Principales Contribuciones	44
5.4 Limitaciones y Alcances.....	44
5.5 Recomendaciones y Trabajos Futuros	44
5.6 Palabras finales	45
REFERENCIAS BIBLIOGRÁFICAS.....	46
Aspectos Teóricos y Arquitectura de Software	46
Tecnologías Móviles y Bases de Datos.....	46
ANEXOS	47

INDICE DE FIGURAS

Figures 1 Diagrama de arquitectura de carpetas del proyecto. Fuente: Elaboración Propia.	29
Figures 2 Esquema de tabla 'mutation_queue'. Fuente: Elaboración Propia.....	30
Figures 3 Diagrama de módulos. Fuente: Elaboración Propia.	34
Figures 4 Diagrama de Secuencia de Sincronización. Fuente: Elaboración Propia.	35
Figures 5 Diagrama de Estados de la Cola. Fuente: Elaboración Propia.....	36
Figures 6 Modelo de Datos para SQLite. Fuente: Elaboración Propia.....	37

INDICE DE TABLAS

Tabla 1: Comparativa técnica de ecosistemas de desarrollo móvil. Fuente: Elaboración Propia.	19
Tabla 2 Comparativa de motores de persistencia local. Fuente: Elaboración Propia.	22
Tabla 3 Comparativa de dimensiones clave en sistemas de gestión de identidad. Fuente: Elaboración Propia.	23
Tabla 4 Alineación entre Objetivos Específicos y Solución Tecnológica. Fuente: Elaboración Propia.	26
Tabla 5 Comparativa técnica de Capacidades Offline. Fuente: Elaboración Propia.	33
Tabla 6 Especificaciones del Entorno de Prueba (AVD). Fuente: Elaboración Propia.	38
Tabla 7 Benchmark de Rendimiento SQLite (Promedio en AVD). Fuente: Elaboración Propia.	39
Tabla 8 Comparativa de Uso de Red (Inferencia de Eficiencia). Fuente: Elaboración Propia.	40

INTRODUCCIÓN

La ingeniería de software móvil enfrenta el desafío constante de la inestabilidad de red, operando frecuentemente bajo la falsa premisa de una conectividad permanente. Este documento aborda la naturaleza de este problema en aplicaciones de campo, donde la dependencia de arquitecturas *Online-First* interrumpe procesos críticos y compromete la integridad de los datos.

La propuesta de solución consiste en una arquitectura *Offline-First* que utiliza la base de datos local como fuente primaria de verdad. La metodología empleada incluyó el análisis de patrones como el Teorema CAP y el diseño de un motor de sincronización *Store-and-Forward* implementado con React Native JSI.

El documento se estructura en cinco capítulos: el Capítulo 1 define el problema y objetivos; el Capítulo 2 establece el marco conceptual y tecnológico; el Capítulo 3 detalla la propuesta técnica y algoritmos; el Capítulo 4 presenta la validación y métricas de rendimiento; y el Capítulo 5 expone las conclusiones y recomendaciones futuras.

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA

1.1 Contexto

La ingeniería de software móvil ha operado frecuentemente bajo la "falacia de la red fiable", una de las ocho falacias clásicas de la computación distribuida (Deutsch, 1994). En el contexto actual, la mayoría de las aplicaciones móviles empresariales y de consumo se construyen sobre arquitecturas cliente-servidor donde el dispositivo móvil delega el procesamiento y el almacenamiento a servidores remotos a través de APIs REST o GraphQL. Este modelo funciona adecuadamente en entornos de oficina controlados con Wi-Fi estable y baja latencia, pero falla en entornos reales de movilidad, donde los cambios de celdas de red e interferencias bloquean operaciones críticas de trabajadores en terreno, desde técnicos industriales hasta conductores de reparto (Deutsch, 1994).

Las organizaciones que dependen de estos trabajadores reportan que la dependencia de la conectividad bloquea operaciones críticas. Cuando la red falla en una aplicación Online-First, el flujo de trabajo se interrumpe: el usuario no puede guardar un formulario, no puede acceder a información histórica vital para la toma de decisiones, o peor aún, cree que ha guardado información que se pierde silenciosamente al cerrar la aplicación antes de que la red se restablezca.

Además, el auge de las aplicaciones web progresivas (PWA) ha intentado cerrar la brecha utilizando Service Workers y almacenamiento en caché. Sin embargo, como se discutirá en detalle, las limitaciones del sistema operativo (especialmente en iOS) respecto a la ejecución en segundo plano y el acceso al almacenamiento persistente a gran escala a menudo hacen que las soluciones nativas o híbridas (como React Native) sean necesarias para casos de uso críticos que requieren garantías de sincronización robustas.

1.2 Situación actual

Actualmente, los sistemas de gestión de campo operan mayoritariamente bajo arquitecturas "conectadas" (*Online-First*). En este escenario, el dispositivo móvil actúa como una terminal tonta que requiere validación constante del servidor para guardar un formulario, consultar un historial o cambiar el estado de una tarea. Cuando un técnico ingresa a un sótano o viaja a una zona rural, el flujo de datos se corta.

Además, el panorama del desarrollo móvil muestra una dicotomía. Por un lado, existen aplicaciones nativas robustas pero costosas de desarrollar para múltiples plataformas. Por otro lado, muchas aplicaciones multiplataforma intentan mitigar los problemas de red mediante cachés temporales en memoria o almacenamiento local no estructurado y simplista. Estas soluciones son a menudo parches ad-hoc que carecen de una estrategia coherente de sincronización y persistencia transaccional (Fowler, 2002).

1.3 Actores involucrados

El problema de la falta de una arquitectura *Offline-First* robusta afecta a múltiples niveles de la interacción humano-computadora y del negocio:

- **Usuarios Finales (Operarios de Campo/Consumidores):** Son los más afectados directamente. Enfrentan frustración por la pérdida de trabajo realizado, tiempos de espera injustificados y una creciente desconfianza en la herramienta digital. Estadísticas de rendimiento móvil sugieren que la tolerancia al tiempo de carga es menor a 3 segundos; superar este umbral debido a latencia de red incrementa exponencialmente el abandono de la tarea o de la aplicación misma.
- **Desarrolladores de Software:** Se ven obligados a implementar lógica defensiva repetitiva y propensa a errores para manejar reintentos, tiempos de espera y almacenamiento en caché disperso en cada pantalla de la aplicación. Esto aumenta la deuda técnica, la complejidad del mantenimiento y la probabilidad de introducir *bugs* relacionados con estados de carrera.
- **La Organización:** Sufre por la ineficiencia operativa. En escenarios de logística, servicios públicos o mantenimiento, la incapacidad de sincronizar una orden de trabajo en tiempo real —o de guardarla de forma segura para una sincronización posterior automática— retrasa la facturación, afecta el cumplimiento de SLAs (Service Level Agreements) y reduce la visibilidad operativa en tiempo real.

1.4 Dificultades actuales

La dependencia de arquitecturas *Online-First* o soluciones de caché ingenuas genera fricciones técnicas y operativas específicas:

1. **Bloqueo de Interfaz (UI Freezing):** Las operaciones de red síncronas congelan la interfaz de usuario mientras esperan una respuesta del servidor. En redes de alta latencia (EDGE/3G), esto resulta en tiempos de espera impredecibles, superando el umbral de atención del usuario de 3 segundos y generando la percepción de que la aplicación "se ha colgado".
2. **Gestión de Estado Inconsistente:** Sin un manejo riguroso de la concurrencia, los datos mostrados en el dispositivo pueden diferir de la realidad del servidor (*stale reads*). Además, si la aplicación es terminada por el sistema operativo para liberar memoria mientras hay una petición en vuelo, el estado de la transacción queda indeterminado, sin confirmación de éxito o fracaso.
3. **Consumo Ineficiente de Recursos:** Los intentos continuos e inmediatos de reconexión (*polling*) o el envío de datos granulares sin agrupar (*batching*) agotan prematuramente la batería del dispositivo y consumen planes de datos limitados, calentando el hardware innecesariamente.

- 4. Pérdida de Datos en transición:** En ausencia de colas persistentes transaccionales, los datos ingresados por el usuario residen en la memoria RAM. Ante un cierre inesperado de la aplicación o un reinicio del dispositivo, esta información se pierde irremediablemente antes de ser transmitida.

1.5 Consecuencias de no resolver estas dificultades

La persistencia de estas dificultades técnicas escala rápidamente hacia impactos negativos en el negocio y la operación:

Discontinuidad Operativa: La incapacidad de procesar una orden de trabajo offline detiene la cadena de valor. Si un técnico no puede cerrar una orden debido a falta de señal, la facturación se retrasa, los SLAs (Service Level Agreements) se incumplen y la planificación de recursos se vuelve inexacta.

Corrupción de Datos e Integridad: La falta de mecanismos de idempotencia y control de transacciones distribuidas puede llevar a la duplicación de registros (ej: facturar dos veces el mismo servicio) o a la creación de registros huérfanos, corrompiendo la base de datos central y requiriendo costosas intervenciones manuales de limpieza de datos.

Abandono Tecnológico: La frustración recurrente por la pérdida de trabajo realizado o la lentitud de la aplicación genera desconfianza en los usuarios finales. Esto lleva al abandono de la herramienta digital y al retorno a procesos manuales (papel), anulando la inversión en transformación digital.

1.6 Competencia

Existen soluciones comerciales de gestión de mantenimiento (CMMS) y plataformas de desarrollo que abordan parcialmente esta problemática, pero presentan limitaciones técnicas o de costos:

- **Software SaaS Genérico:** Muchas plataformas comerciales ofrecen un "modo offline" básico que solo permite lectura (caché). La capacidad de *crear* o *editar* datos complejos sin conexión suele ser limitada o propensa a conflictos de sincronización.
- **Desarrollos Web Móviles (PWA básicas):** Intentan solucionar el problema mediante el navegador, pero a menudo carecen de la capacidad de mantener procesos de sincronización robustos si el usuario cierra el navegador o bloquea la pantalla, interrumpiendo el envío de datos críticos.
- **Procesos Manuales (Papel/Excel):** Siguen siendo el mayor competidor. Aunque funcionan "offline", carecen de validación de datos, no permiten multimedia (fotos de evidencia) y requieren una doble digitación posterior propensa a errores humanos.

Esta solución se diferencia al proponer una arquitectura de "caja blanca", transparente y adaptable, que implementa patrones de ingeniería robustos (colas persistentes, máquinas

de estado) accesibles para desarrollos a medida sin las restricciones de plataformas propietarias.

1.7 Objetivo general

Desarrollar una arquitectura de software móvil resiliente para empresas con operaciones de servicios en terreno, basada en el paradigma *Offline-First* y el patrón *Store-and-Forward*, que garantice la continuidad operativa, la integridad de los datos y una experiencia de usuario fluida en escenarios de conectividad intermitente o nula.

1.8 Objetivos específicos

1. **Analizar Patrones Arquitectónicos:** Evaluar y comparar estrategias de diseño de sistemas distribuidos (Teorema CAP, PACELC) para definir el modelo de consistencia y disponibilidad más adecuado para operaciones en campo.
2. **Diseñar el Mecanismo de Sincronización:** Definir un algoritmo de intercambio de datos bidireccional que incluya detección de red, colas de persistencia local y estrategias de reintento exponencial (*backoff*) para asegurar la entrega de datos.
3. **Implementar Estrategias de Resolución de Conflictos:** Desarrollar lógica de negocio para manejar la concurrencia de datos (edición simultánea offline) mediante versionado o reglas deterministas, asegurando la convergencia del sistema.
4. **Validar la Solución Tecnológica:** Construir un prototipo funcional utilizando tecnologías móviles modernas para someter la arquitectura propuesta a pruebas de estrés bajo condiciones de red simuladas (latencia, pérdida de paquetes).

CAPÍTULO 2: MARCO CONCEPTUAL

2.1 Introducción

El presente capítulo establece el marco teórico y técnico para abordar este desafío mediante el diseño de una aplicación móvil con arquitectura Offline-First (Primero Desconectado). A diferencia de las aplicaciones tradicionales que tratan la falta de conexión como un estado de error excepcional, la filosofía *offline-first* invierte esta dependencia, asumiendo la desconexión como el estado natural de la aplicación y la conectividad como una mejora ocasional. Para materializar esta visión, es imperativo seleccionar un stack tecnológico que no solo permita la ejecución de lógica de negocio en el cliente, sino que también ofrezca mecanismos robustos de persistencia local, sincronización asíncrona y soberanía sobre la identidad del usuario.

A lo largo de este capítulo, se desarrollará un análisis exhaustivo y comparativo de las tecnologías seleccionadas para la propuesta de solución: React Native con el ecosistema Expo como entorno de ejecución, y BetterAuth como sistema de gestión de identidad y sesiones. Estas herramientas serán contrastadas rigurosamente frente a sus competidores directos en el mercado —tales como Flutter, el desarrollo nativo puro (Swift/Kotlin), Firebase Authentication y Auth0— evaluando su idoneidad bajo los criterios de rendimiento, mantenibilidad, costo a escala y capacidad de operar en aislamiento. El objetivo final es justificar, mediante evidencia técnica y teórica, por qué esta combinación específica de tecnologías ofrece la respuesta más eficiente y resiliente al problema de la intermitencia de red en aplicaciones empresariales críticas.

2.2 Computación Móvil Desconectada

2.2.1 Del Paradigma Cloud-First al Local-First

El modelo tradicional de desarrollo móvil delega la "verdad" de los datos al servidor remoto. En este esquema, cualquier interacción del usuario —como guardar un formulario o consultar un inventario— dispara una petición HTTP síncrona. Si la red falla, la operación falla y el usuario se bloquea. Este modelo, aunque simplifica la gestión del estado al centralizarlo, es inherentemente frágil ante la inestabilidad de la red.

La arquitectura **Local-First** (o *Offline-First*) propone un cambio radical: la base de datos local embebida en el dispositivo se convierte en la fuente primaria de la verdad para la interfaz de usuario (Hipp, 2020). Las operaciones de lectura y escritura se ejecutan siempre contra el almacenamiento local, garantizando una latencia de cero milisegundos y una disponibilidad absoluta. La sincronización con la nube se relega a un proceso en segundo plano, desacoplado de la interacción inmediata del usuario (Vogels, 2009). Esto transforma la percepción de la aplicación, que pasa de ser una "terminal tonta" del servidor a un sistema distribuido autónomo capaz de eventual consistencia.

2.2.2 El Teorema CAP y la Elección de Disponibilidad

El diseño de sistemas distribuidos móviles está inevitablemente restringido por el Teorema CAP, que establece la imposibilidad de garantizar simultáneamente tres propiedades: Consistencia (Consistency), Disponibilidad (Availability) y Tolerancia a Particiones (Partition Tolerance) (Brewer, 2000).

En un entorno móvil, la Tolerancia a Particiones (P) es una condición ineludible; los dispositivos móviles entran y salen constantemente de zonas de cobertura, creando particiones de red físicas sobre las cuales el arquitecto de software no tiene control. Por lo tanto, la decisión de diseño se reduce a elegir entre Consistencia (CP) y Disponibilidad (AP) (Brewer, 2000) durante estos periodos de desconexión.

- Sistemas CP (Consistencia + Tolerancia a Particiones): Priorizan la integridad de los datos. Si el dispositivo no puede comunicarse con el servidor para confirmar que un dato es actual o para validar una escritura, el sistema bloquea la operación o muestra un error. Este enfoque es inviable para aplicaciones de campo, donde detener el trabajo por falta de señal implica pérdidas económicas directas.
- Sistemas AP (Disponibilidad + Tolerancia a Particiones): Priorizan la operatividad. El sistema permite al usuario seguir leyendo y escribiendo datos locales, aceptando el riesgo de que la información en el dispositivo pueda estar temporalmente desactualizada respecto al servidor (stale data) o que puedan generarse conflictos de escritura.

Para la aplicación propuesta, se adopta decididamente un enfoque AP. La arquitectura debe garantizar que la aplicación esté siempre disponible para el usuario (Vogels, 2009). La consistencia se maneja bajo el modelo de Consistencia Eventual, asegurando que, dado un tiempo suficiente sin nuevas escrituras y con conectividad restaurada, todos los nodos (dispositivos y servidor) convergerán al mismo estado.

2.2.3 Patrón Arquitectónico Store-and-Forward

El mecanismo técnico que habilita la operatividad *offline* de escritura es el patrón Store-and-Forward (Almacenar y Reenviar). Este patrón desacopla la captura de información de su transmisión. Las mutaciones se guardan transaccionalmente en una cola local y se envían de forma oportunista cuando se detecta red (Fowler, 2002).

El flujo de datos en este patrón para una aplicación móvil se estructura en las siguientes fases críticas:

1. Captura y Persistencia Atómica (Store): Cuando el usuario realiza una acción (e.g., "Completar Inspección"), la aplicación no intenta enviarla a la

- red. En su lugar, crea un registro de la mutación (intención de cambio) y lo guarda transaccionalmente en una base de datos local. Este paso debe ser atómico; si la batería se agota milisegundos después, el dato no debe perderse ni corromperse.
2. Cola de Sincronización (Queue): Las mutaciones persistidas se organizan en una cola FIFO (First-In, First-Out) o priorizada. Esta cola actúa como un búfer resiliente que absorbe la variabilidad de la red.
 3. Transmisión Oportunista (Forward): Un gestor de sincronización monitorea el estado de la red. Al detectar conectividad, procesa la cola, enviando las mutaciones al servidor. Es crucial implementar mecanismos de "Backoff Exponencial" para reintentos, evitando saturar la red o drenar la batería en condiciones de señal inestable.
 4. Idempotencia y Reconciliación: Dado que una respuesta del servidor (ACK) puede perderse en una red inestable, el cliente podría reenviar la misma mutación múltiples veces. El servidor debe implementar idempotencia para garantizar que procesar la misma solicitud N veces tenga el mismo efecto que procesarla una vez.

Este patrón no solo resuelve el problema de la desconexión, sino que también mejora drásticamente la experiencia de usuario mediante técnicas de Optimistic UI (Interfaz Optimista). La aplicación actualiza la interfaz asumiendo que la operación en cola tendrá éxito, ofreciendo una respuesta instantánea y fluida, y solo revierte el cambio (Rollback) si la sincronización final falla definitivamente y no puede ser resuelta.

2.3 Frameworks de Desarrollo Móvil

La elección del marco de desarrollo es una de las decisiones más críticas en la ingeniería de software móvil, con implicaciones profundas en el rendimiento, la mantenibilidad y la viabilidad a largo plazo del proyecto. React Native ha sido seleccionado como la tecnología base, específicamente bajo su Nueva Arquitectura (New Architecture), que representa un cambio paradigmático respecto a sus versiones anteriores y lo posiciona competitivamente frente a soluciones nativas y alternativas como Flutter.

Un framework es una *infraestructura de software* que proporciona un conjunto organizado de componentes reutilizables, bibliotecas, herramientas y reglas que sirven como base para construir aplicaciones. En lugar de comenzar desde cero, el framework establece una estructura predefinida (arquitectura, patrones, módulos y flujos) sobre la cual el desarrollador implementa la lógica específica de su software. React Native es un framework de código abierto creado por Meta que permite desarrollar aplicaciones móviles nativas utilizando JavaScript y React. A diferencia de las soluciones "híbridas"

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward

antiguas que usaban WebViews (navegadores embebidos), React Native instancia componentes nativos reales (UIViews en iOS, Views en Android), ofreciendo rendimiento y tacto nativo) (Meta Open Source, 2025).

Aunque Flutter es un competidor formidable con un rendimiento gráfico excepcional gracias a su motor Skia (y su sucesor Impeller), React Native fue seleccionado basándose en criterios técnicos específicos alineados con los objetivos del proyecto, particularmente en relación con la gestión de datos y la arquitectura del sistema.

Tabla 1: Comparativa técnica de ecosistemas de desarrollo móvil.
Fuente: Elaboración Propia.

<i>Característica</i>	<i>React Native (Seleccionado)</i>	<i>Flutter (Google)</i>	<i>Desarrollo Nativo (Swift/Kotlin)</i>	<i>Progressive Web Apps (PWA)</i>
<i>Lenguaje</i>	JavaScript / Dart TypeScript		Swift (iOS) / Kotlin (Android)	
<i>Motor de Renderizado</i>	Componentes Nativos (vía JSI/Fabric)		Motor Gráfico Propio (Skia/Impeller)	
<i>Acceso a Base de Datos (Offline)</i>	Excelente (JSI): Acceso directo a memoria C++ sin latencia. (SQLite)		Bueno (FFI): Interfaz de funciones de foráneas. (CoreData/Room). Excelente: Acceso directo	
<i>Gestión en Segundo Plano</i>	Robusta: Módulos nativos para <i>Background Fetch/Tasks</i> .		Compleja: Requiere configuración de "Isolates" separados. Nativa: Control total.	
<i>Ecosistema</i>	Masivo (NPM): Librerías maduras de criptografía y sincronización.		Creciente (Pub.dev): Menos librerías de nicho empresarial. Fragmentado: Requiere buscar libs para cada OS.	
			Limitado: IndexedDB (Cuotas y riesgo de borrado). Muy Limitada: Service Workers mueren al cerrar la app (iOS). Limitado a APIs Web.	

Se selecciona React Native principalmente por su Arquitectura JSI (JavaScript Interface). Esta tecnología permite que el código JavaScript interactúe directamente con una base de

datos SQLite embebida en C++ de manera síncrona. Esto es crucial para una aplicación offline-first que debe leer y escribir miles de registros localmente sin los "cuellos de botella" de rendimiento asociados a los puentes asíncronos tradicionales. Flutter, aunque performante gráficamente, requiere el uso de Dart (un lenguaje menos común) y su gestión de bases de datos relacionales complejas a menudo implica mayor fricción en la serialización de datos comparado con el acceso directo a memoria de JSI en React Native.

2.4 Entorno de Ejecución y Construcción

Un entorno de ejecución es el *conjunto de software, servicios, configuraciones y herramientas* que permiten que un programa se ejecute correctamente. Incluye todos los elementos necesarios para que el código funcione tal como fue diseñado: intérpretes, máquinas virtuales, APIs del sistema, librerías, gestores de procesos, controladores, e incluso reglas sobre cómo se maneja la memoria, los hilos y los eventos.

Expo es una plataforma y conjunto de herramientas construidas sobre React Native que abstrae la complejidad de la configuración del entorno nativo (Xcode/Android Studio), facilitando el acceso a APIs del dispositivo (Cámara, Sensores, Sistema de Archivos) y la gestión de actualizaciones (OTA) (Expo, 2025).

Expo ha transformado el desarrollo en React Native de una experiencia fragmentada a un flujo de trabajo industrializado. Lejos de ser solo una herramienta para principiantes, el ecosistema Expo moderno (SDK 50+) proporciona una capa de infraestructura robusta que abstrae la complejidad de la configuración nativa sin limitar el acceso a las capacidades del dispositivo.

Este, trae consigo el concepto de *Continuous Native Generation* (CNG), o Prebuild, es central en la propuesta de valor de Expo. En lugar de mantener carpetas android e ios estáticas y frágiles en el repositorio de código, Expo genera estos proyectos nativos bajo demanda a partir de archivos de configuración (app.json) y *Config Plugins*.

Este enfoque ofrece ventajas significativas para la mantenibilidad del proyecto a largo plazo:

- **Actualizaciones de SDK sin Dolor:** Actualizar la versión de React Native o del sistema operativo se convierte en una tarea de cambiar una versión en el package.json y regenerar los directorios nativos, en lugar de realizar fusiones de código complejas y propensas a errores en archivos .gradle o Podfile.
- **Determinismo:** El entorno nativo se construye de manera determinista, eliminando el clásico problema de "funciona en mi máquina" debido a diferencias en versiones de Xcode o Android Studio locales.

- **EAS Build:** Servicio en la nube que compila los binarios. Permite generar builds de desarrollo (*Development Builds*) que incluyen librerías nativas personalizadas, superando la limitación histórica de "Expo Go" que impedía usar código nativo arbitrario.

Además, EAS Update permite desplegar cambios en el código JavaScript y los activos estáticos (imágenes, fuentes) directamente a los dispositivos de los usuarios, evitando el proceso de revisión de las tiendas de aplicaciones (Apple App Store y Google Play). Desde una perspectiva de arquitectura de sistemas, esto permite una estrategia de despliegue continuo (CD) real para aplicaciones móviles. Errores críticos en la lógica de negocio o de sincronización pueden ser parcheados en cuestión de minutos, una capacidad vital para la estabilidad de una aplicación *Offline-First* compleja.

Expo proporciona una suite de APIs nativas estandarizadas que son esenciales para la funcionalidad de la aplicación:

- **expo-file-system:** Proporciona acceso al sistema de archivos local para almacenar imágenes o documentos descargados, complementando a la base de datos estructurada.
- **expo-background-task:** Crucial para la arquitectura *store-and-forward*. Permite registrar tareas que el sistema operativo ejecutará periódicamente, incluso si la aplicación está cerrada, para sincronizar datos pendientes. Esta librería abstrae las diferencias complejas entre WorkManager (Android) y BGTaskScheduler (iOS), asegurando que la sincronización ocurra de manera confiable y eficiente en términos de batería.

2.5 Persistencia de Datos Local

En una arquitectura *Offline-First*, la base de datos local no es una simple caché; es el corazón de la aplicación. SQLite ha sido seleccionado como el motor de persistencia debido a su fiabilidad ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad), su ubicuidad y su rendimiento comprobado en dispositivos embebidos (Hipp, 2020).

La implementación de SQLite en React Native ha evolucionado. Las versiones modernas (como expo-sqlite en su versión "next" o op-sqlite) utilizan la arquitectura JSI para maximizar el rendimiento.

1. **Write-Ahead Logging (WAL):** Se configura la base de datos en modo WAL. En este modo, las escrituras se añaden a un archivo de registro (log) en lugar de sobrescribir el archivo de base de datos principal inmediatamente. Esto permite una concurrencia real: las operaciones de lectura pueden ocurrir simultáneamente con una operación de escritura, evitando que la interfaz de usuario se bloquee durante una sincronización de datos en segundo plano (SQLite, 2024).

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward

2. **Acceso Síncrono:** Gracias a JSI, las consultas de lectura (SELECT) pueden ejecutarse de forma síncrona. Esto simplifica enormemente la gestión del estado en la UI, ya que los datos están disponibles en el mismo ciclo de renderizado de JavaScript, eliminando parpadeos y la necesidad de gestionar estados de carga (isLoading) intermedios para datos que ya están en el disco local.

Tabla 2 Comparativa de motores de persistencia local.
Fuente: Elaboración Propia.

Tecnología	Arquitectura Subyacente	Rendimiento (Lectura/Escritura)	Soporte JSI	Concurrencia	Caso de Uso Óptimo
Expo SQLite (Modern/Next)	SQLite + JSI (C++)	Extremadamente Alto (Zero-copy)	Sí	Alto (WAL Mode)	Apps Offline-First complejas, relacionales.
AsyncStorage	Key-Value (Archivos/XML)	Bajo (Serialización JSON, Asíncrono)	No	Bajo (Serial)	Configuración simple, tokens, flags.
Realm (Atlas Device)	Motor Objetos (C++)	Alto (Zero-copy)	Sí	Alto	Sincronización compleja out-of-the-box (Vendor Lock-in).
WatermelonDB	Lazy Loading sobre SQLite	Alto (Optimizado para React)	Sí (vía adaptador)	Medio	Apps reactivas con miles de registros en listas.
React Native SQLite Storage	SQLite (Puente Antiguo)	Medio/Bajo (Serialización Puente)	No	Medio	Proyectos legacy sin Nueva Arquitectura.

Expo SQLite (en su variante JSI) ofrece el equilibrio perfecto entre rendimiento bruto, integración nativa con el ecosistema Expo y flexibilidad SQL estándar. A diferencia de Realm, no ata la arquitectura a un servicio de backend específico (MongoDB Atlas), preservando la independencia tecnológica. A diferencia de AsyncStorage, ofrece capacidades relacionales y transaccionales necesarias para mantener la integridad de los datos complejos de negocio y las colas de sincronización.

2.6 Gestión de Identidad y Seguridad

La autenticación en aplicaciones modernas va más allá de un simple formulario de login. Implica la gestión segura de sesiones, rotación de tokens, protección contra ataques CSRF/XSS y, crucialmente en este proyecto, soporte para flujos *offline*. BetterAuth emerge como la solución seleccionada, representando una nueva generación de marcos de autenticación "TypeScript-First" y agnósticos de la infraestructura.

BetterAuth se distingue por ser una librería que se integra en el backend de la aplicación (similar a Auth.js/NextAuth) pero con una arquitectura más modular y completa. Permite el control total sobre los datos de los usuarios, almacenándolos en la misma base de datos que el resto de la aplicación, lo cual es una ventaja significativa sobre los servicios gestionados donde los datos de usuario viven en un silo separado (BetterAuth, 2024).

Este soporta dos modos de operación, y la elección es crítica para el funcionamiento *offline*:

- 1. **Stateful (Base de Datos):** Las sesiones se almacenan en una tabla. Cada petición verifica la base de datos. Es seguro y revocable instantáneamente, pero introduce latencia y dependencia estricta de la base de datos central.
- 2. **Stateless (JWT/JWE):** La sesión se encapsula en un token firmado criptográficamente (JSON Web Token) o encriptado (JSON Web Encryption). El servidor valida la firma sin consultar la base de datos.

Además, el cliente de BetterAuth maneja automáticamente:

- La persistencia segura del token.
- La inyección del token en los encabezados de las peticiones (Authorization: Bearer...) mediante plugins.
- La renovación silenciosa de sesiones (Refresh Token rotation) cuando hay red disponible.

La decisión de usar BetterAuth se refuerza al compararla con alternativas tradicionales. La siguiente tabla destaca las diferencias en dimensiones clave como control de datos, costes y soporte para arquitecturas modernas.

Tabla 3 Comparativa de dimensiones clave en sistemas de gestión de identidad. Fuente: Elaboración Propia.

Dimensión	BetterAuth (Seleccionada)	Firebase Auth	Auth0	Supabase Auth
Naturaleza	Librería Open Source (Self-hosted)	Plataforma (Google Cloud)	BaaS IDaaS (SaaS)	Servicio BaaS (Open Source/Hoste

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward

	d)					
Soberanía de Datos	Total. Datos en tu propia tabla/schema de BD.	Parcial. Datos en "caja negra" de Google. Difícil de exportar/relacionar.	Nula. Datos en servidores de Auth0.	Alta. Datos en esquema auth de Postgres.		
Coste	Gratuito. Solo coste de infraestructura propia.	Freemium. Costes ocultos al escalar (MAU, SMS, Functions).	Alto. Modelo de precios agresivo para empresas (B2B).	Freemium. Costes por MAU excedente.		
Integración DB	Nativa. JOIN directos entre usuarios y datos de negocio.	Compleja. Requiere sincronizar IDs de Firebase con tu BD propia.	Compleja. Requiere webhooks para sincronizar datos.	Nativa (si usas Supabase como BD).		
Gestión Offline	Flexible. Configurable vía plugins JWT y caché local.	Excelente. SDK maneja caché y reintentos automáticamente.	Limitada. Depende de Refresh Tokens y configuración de scopes.	Buena. SDK persistente.		
Extensibilidad	Plugins TS. Código abierto, fácil de modificar/extender.	Limitada a Cloud Functions y extensiones predefinidas.	Actions/Rules (JS en sandbox). Potente pero propietario.	Triggers de Postgres y Edge Functions.		
Experiencia DX	TypeScript-First. Tipado estático completo y autocompletado.	SDKs maduros, pero a veces verbosos y poco tipados.	Configuración compleja en dashboard.	SQL-centric. Fuerte en backend, variable en frontend.		

Aunque Firebase Auth ofrece una experiencia "out-of-the-box" superior para el manejo offline (su SDK gestiona caché y reintentos automáticos), impide realizar consultas SQL eficientes que unan datos de usuarios con datos de negocio y genera un acoplamiento fuerte con el ecosistema de Google (Vendor Lock-in).

BetterAuth ofrece el equilibrio ideal: proporciona primitivas de seguridad modernas (2FA, gestión de sesiones, OAuth) y plugins extensibles, pero mantiene la soberanía de los datos dentro de la infraestructura del proyecto. Esto permite implementar una lógica de autenticación *offline* personalizada que se alinea perfectamente con la arquitectura *Store-and-Forward* diseñada, algo difícil de lograr con las abstracciones rígidas de Auth0 o Firebase. Además de tener la capacidad de integrarse con Expo otorgándole un cliente optimizado, su naturaleza *stateless* (opcional) se adapta mejor a arquitecturas modernas de microservicios o serverless que podrían explorarse en el proyecto.

2.7 Estrategia de Sincronización e Integridad de Datos

La integración de los componentes anteriores (React Native, SQLite, BetterAuth) se cristaliza en la estrategia de sincronización. El objetivo es garantizar la integridad de los datos en un entorno de consistencia eventual.

El patrón Store-and-Forward se implementa para gestionar las escrituras de datos (mutaciones).

1. **Captura de Intención (UI Optimista):** Cuando el usuario realiza una acción (ej. guardar un formulario), la interfaz se actualiza inmediatamente (*Optimistic Update*), asumiendo éxito.
2. **Persistencia Atómica:** La mutación se guarda en SQLite dentro de una transacción ACID. Se guardan tanto los datos actualizados (para lectura local) como un registro de la "Intención de Sincronización" en una tabla de cola (SyncQueue).
3. **Procesamiento Diferido:** Un orquestador de sincronización, impulsado por expo-background-task, monitorea el estado de la red (vía NetInfo). Cuando hay conexión, lee la cola SyncQueue en orden FIFO (First-In, First-Out).
4. **Transmisión Autenticada:** Cada elemento de la cola se envía al servidor utilizando el token de sesión de BetterAuth (recuperado de SecureStore). Se emplea un plugin "Bearer" para adjuntar la credencial.
5. **Confirmación y Limpieza:** Si el servidor confirma el éxito (HTTP 200), el registro se elimina de la cola. Si falla por error recuperable (5xx, timeout), se aplica una estrategia de *Exponential Backoff*. Si falla por error de negocio (4xx), se marca para resolución manual o se revierte el cambio local.

Esta arquitectura desacopla completamente la experiencia del usuario de la latencia de la red, cumpliendo con los requisitos de robustez y resiliencia esperados en una aplicación móvil profesional en 2025.

La robustez del sistema depende de su capacidad para manejar fallos. La cola de sincronización implementa una máquina de estados para cada tarea: PENDING, PROCESSING, FAILED, COMPLETED.

- **Idempotencia:** El servidor debe diseñarse para ser idempotente. Si el cliente envía una solicitud, el servidor la procesa, pero la respuesta se pierde por un fallo de red, el cliente reintentará. El servidor debe detectar que es una operación duplicada y devolver el resultado exitoso sin duplicar el efecto (e.g., sin crear dos registros idénticos).
- **Gestión de Tokens Expirados:** Durante la sincronización en segundo plano, es posible que el token de acceso de BetterAuth haya expirado. El cliente debe ser capaz de interceptar el error 401, utilizar el *Refresh Token* para obtener una nueva sesión, y reintentar la operación original de forma transparente, sin intervención del usuario.

2.8 Conclusión del Capítulo

Esta base tecnológica posiciona al proyecto en la vanguardia del desarrollo móvil actual, priorizando la experiencia del usuario final (*User-Centric*) y la solidez ingenieril (*Robustness*) sobre la conveniencia a corto plazo.

La selección de este *stack* tecnológico no es arbitraria; cada componente responde directamente a un objetivo específico planteado en el Capítulo 1.

Tabla 4 Alineación entre Objetivos Específicos y Solución Tecnológica.
Fuente: Elaboración Propia.

Objetivo (Cap 1)	Específico	Componente Tecnológico (Cap 2)	Justificación Técnica de la Alineación
Garantizar disponibilidad continua (Offline)		Arquitectura Local-First con SQLite	Al usar SQLite como fuente de verdad, la UI siempre lee datos locales, eliminando la dependencia de la red para la operatividad básica.
Asegurar integridad de datos (No pérdida)		Patrón Store-and-Forward sobre React Native	React Native permite acceso a hilos de fondo para garantizar que el proceso "Forward" (envío) ocurra incluso si el usuario minimiza la aplicación.
Gestión de Seguridad Desconectada		BetterAuth	Al ser <i>self-hosted</i> , permite replicar tablas de sesiones localmente, permitiendo validar identidad sin contactar un servidor

central.		
Optimización Desarrollo Mantenimiento	de Expo (EAS) y	Abstrae la complejidad nativa, permitiendo al equipo enfocarse en la lógica de sincronización compleja en lugar de configuración de compiladores.

La definición de esta base tecnológica es fundamental, ya que la resiliencia no debe ser un componente añadido, sino el pilar estructural de la arquitectura propuesta. La integración sinérgica de React Native JSI, SQLite y BetterAuth permite que el dispositivo móvil deje de ser una terminal dependiente del servidor para convertirse en un nodo de computación autónomo, garantizando la soberanía de los datos del operario y la seguridad de la sesión incluso en estados de aislamiento prolongado. Con este marco robusto establecido, que no solo optimiza el rendimiento, sino que blindo la operación ante fallos de infraestructura externa, la propuesta avanza hacia el diseño detallado de la arquitectura de sincronización y la implementación de los algoritmos de resolución de conflictos.

CAPÍTULO 3: PROPUESTA DE SOLUCIÓN

3.1 Introducción

Este capítulo detalla la ingeniería de la solución, definiendo la arquitectura de software, los patrones de diseño y los algoritmos críticos que permiten la operatividad en entornos desconectados. La propuesta se estructura bajo un enfoque modular basado en características (*Feature-Based Architecture*), optimizado para el enrutamiento basado en archivos de Expo Router y la persistencia de alto rendimiento.

3.2 Arquitectura de Software: Enfoque Feature-Based

Para garantizar la escalabilidad y el desacoplamiento del código, se ha optado por una arquitectura *Feature-Based* (Basada en Características). A diferencia de las arquitecturas tradicionales en capas, donde se agrupan los archivos por su función técnica (como *Controllers*, *Models* o *Views*), este enfoque agrupa el código según su dominio de negocio; por ejemplo: Órdenes, Autenticación y Sincronización.

La solución se organiza en una estructura modular dentro del directorio raíz, separando claramente la capa de enrutamiento (interfaz de navegación) de la lógica de negocio o dominio. Este diseño se alinea con el uso de Expo Router, donde la estructura de carpetas dicta directamente la navegación de la aplicación y facilita la generación nativa continua (Expo, 2025). Esto permite que cada funcionalidad contenga su propia lógica, estado y componentes de interfaz de usuario (UI), manteniendo así una alta cohesión y un bajo acoplamiento en todo el sistema.

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward

```
/src
|
├─ app/                # Capa de enrutamiento (Expo Router)
|   ├─ (auth)/          # Rutas de autenticación
|   ├─ (tabs)/          # Navegación principal (Home, Órdenes, Perfil)
|   └─ _layout.tsx      # Providers globales (Auth, Theme, QueryClient)
|
├─ features/           # Módulos de negocio (dominio)
|   ├─ auth/            # BetterAuth, tokens, biometría
|   ├─ work-orders/     # CRUD, validaciones, lógica offline
|   └─ ...              # Nuevas funcionalidades
|
├─ sync-engine/        # Motor de sincronización offline
|   ├─ queue/           # mutation_queue (persistencia)
|   ├─ workers/         # Background sync + tareas programadas
|   └─ net/             # Gestión de eventos de conectividad
|
├─ shared/             # Utilidades transversales
|   ├─ hooks/
|   ├─ helpers/
|   └─ constants/
|
├─ ui/                 # Sistema de diseño (componentes UI)
|   ├─ buttons/
|   ├─ inputs/
|   └─ cards/
|
├─ db/                 # SQLite + JSI
|   ├─ schema.sql       # Esquema de tablas
|   └─ client.ts        # Instancia y helpers de BD
|
└─ lib/                # Integración con servicios externos
    ├─ api/             # Axios + interceptores
    ├─ netinfo/         # Estado de red
    └─ storage/         # SecureStore / AsyncStorage
```

Figures 1 Diagrama de arquitectura de carpetas del proyecto.
Fuente: Elaboración Propia.

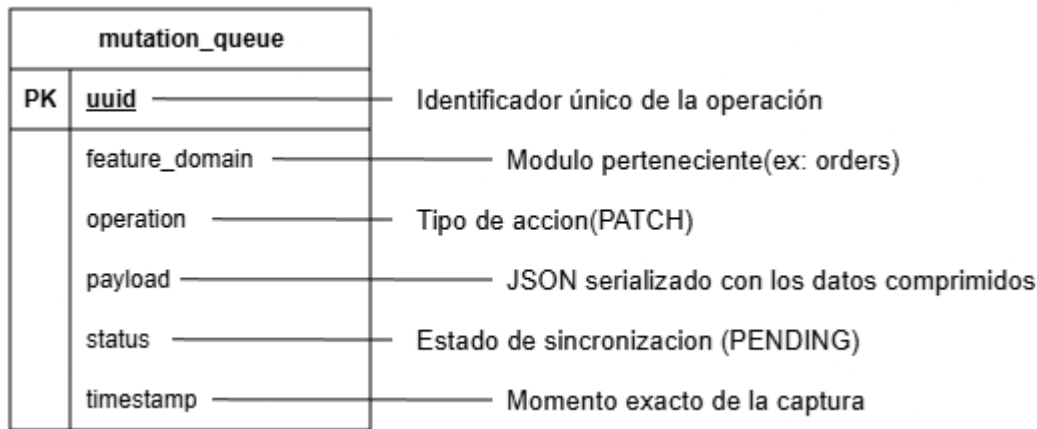
Ventajas Técnicas de esta Estructura:

1. **Encapsulamiento:** La lógica de sincronización de órdenes (/features/work-orders/sync) está aislada de la autenticación. Si se modifica el módulo de órdenes, no se afecta el resto del sistema.
2. **Escalabilidad:** Nuevos módulos (ej: "Inventario") se agregan como nuevas carpetas en /features sin refactorizar la arquitectura base.
3. **Testing Aislado:** Facilita la creación de pruebas unitarias específicas para la lógica de negocio de cada *feature* sin depender de la capa de UI o navegación.

3.3 Estrategia de Persistencia y Sincronización

El sistema implementa el patrón arquitectónico *Local-First*, donde la base de datos local actúa como la fuente de verdad inmediata. Se utiliza expo-sqlite en su versión moderna, la cual aprovecha la interfaz JSI (*JavaScript Interface*) de React Native para permitir una comunicación directa y síncrona con el motor SQLite en C++ (Meta Open Source, 2024).

- **Acceso Síncrono:** JSI permite invocar métodos nativos de C++ (SQLite) directamente desde JavaScript de forma síncrona. Esto elimina el "puente" asíncrono tradicional, permitiendo leer miles de registros de órdenes de trabajo en milisegundos sin bloquear la interfaz de usuario (OP-Engineering, 2024).
- **Atomicidad:** Todas las operaciones de escritura (guardar una orden) se ejecutan dentro de transacciones ACID. Esto garantiza que no existan estados intermedios corruptos si la aplicación se cierra inesperadamente (Hipp, 2020; SQLite, 2024).
- **Mecanismo de Cola:** Para materializar el patrón *Store-and-Forward*, se ha diseñado una tabla específica denominada `mutation_queue`. Esta tabla funciona como un búfer persistente que almacena cada intención de cambio realizada por el usuario hasta que pueda ser transmitida con éxito al servidor (Fowler, 2002). Es fundamental destacar que, si bien cada cambio se registra de forma unitaria para garantizar la integridad atómica, el sistema está diseñado para orquestar la salida de estos datos mediante técnicas de agrupación (*batching*), optimizando así el uso de los recursos de red del dispositivo



Figures 2 Esquema de tabla 'mutation_queue'.
Fuente: Elaboración Propia.

3.4 Diseño del Algoritmo de Sincronización

El algoritmo gestiona la complejidad de las redes móviles asumiendo que la conectividad es inherentemente poco fiable (Deutsch, 1994). El diseño se fundamenta en una máquina

de estados que prioriza la disponibilidad y la tolerancia a particiones de red (**AP**), aceptando un modelo de consistencia eventual (Brewer, 2000; Vogels, 2009).

3.4.1 Fase de Captura y Encolado

Cuando el usuario guarda una orden de trabajo, el sistema realiza tres acciones en una única transacción atómica de SQLite:

1. Actualiza la interfaz de usuario de forma inmediata utilizando una UI Optimista (Optimistic Update).
2. Persiste los datos en la tabla de órdenes para permitir la lectura local inmediata.
3. Inserta la operación en la tabla `mutation_queue` con el estado inicial PENDING.

3.4.2 Fase de Detección y Estabilización de Red (Debounce)

Dado que las conexiones en zonas rurales suelen ser intermitentes (fenómeno conocido como flapping), sincronizar datos inmediatamente después de detectar una señal puede causar fallos de red y un consumo excesivo de batería.

1. Al recibir el evento de conexión (`NetInfo.Connected`), el sistema inicia un temporizador de 3000 ms (Android Developers, 2024).
2. Si la conexión se pierde dentro de esa ventana, el proceso se cancela automáticamente.
3. Solo si la conexión permanece estable tras el tiempo de espera, se dispara el proceso de envío de la cola (Flush Queue).

3.4.3 Fase de Procesamiento y Reintentos

El motor de sincronización procesa la cola siguiendo el orden FIFO (First-In-First-Out) para preservar la causalidad secuencial. Para cumplir con los objetivos de eficiencia energética, el proceso no se realiza de forma individual por cada registro, sino mediante un esquema de sincronización por lotes:

- **Recolección:** El sistema recupera todos los registros en estado de la 'PENDING' tabla 'mutation_queue', mediante una consulta indexada que garantiza el orden cronológico (FIFO) de las mutaciones. Esto es vital para asegurar que la secuencia de eventos generada en el dispositivo se replique fielmente en el servidor, evitando inconsistencias de estado
- **Agrupación:** Estas mutaciones se serializan en un único arreglo o *payload* JSON. Esto permite transformar múltiples peticiones HTTP individuales en una sola transmisión masiva, reduciendo drásticamente el tiempo de actividad de la radio celular.

- **Envío:** El lote completo se envía a la API REST en una única petición autenticada.
- **Manejo de Errores y Retroceso Exponencial:**
 - Si el servidor retorna un éxito (200 OK), el lote completo de registros se marca como 'COMPLETED' o se elimina de la cola localmente de forma atómica.
 - Si ocurre un error de red o de tiempo de espera (Timeout), el registro se marca como 'RETRYING' y se programa un reintento utilizando la fórmula de Retroceso Exponencial: $\text{TiempoEspera} = \text{Base} \times 2^{\text{intentos}}$. Esto evita saturar tanto la red como el servidor durante incidentes de infraestructura.
 - Si el error es de lógica de negocio (400 Bad Request), se identifica el registro causante del error de lógica de negocio para extraerlo del flujo automático y evitar el bloqueo de los registros subsiguientes, la mutación se desplaza a un estado de error persistente sin eliminar los datos de la tabla de negocio ni los archivos multimedia almacenados mediante *expo-file-system*. Esto permite que el usuario visualice el motivo del rechazo del servidor, realice las correcciones necesarias sobre la información local y reintente el envío, garantizando la integridad total de la evidencia recolectada en terreno.

3.4.4 Manejo Especial de Errores de Autenticación (401 Unauthorized):

Durante la sincronización en segundo plano, es posible que el token de acceso haya expirado. El sistema está diseñado para interceptar el error 401, utilizar automáticamente el *Refresh Token* de *BetterAuth* para obtener una nueva sesión y reintentar la operación original de forma transparente para el operario (BetterAuth, 2024).

3.5 Justificación Tecnológica: Nativo vs PWA

La elección de una solución nativa sobre una PWA se justifica por la necesidad de una ejecución robusta en segundo plano. Mientras que las PWA dependen de *Service Workers* que el sistema operativo puede finalizar agresivamente, React Native permite el uso de *WorkManager* en Android para garantizar que las tareas de sincronización se completen (Android Developers, 2024). Asimismo, la persistencia en SQLite evita las limitaciones de cuota y el riesgo de borrado de datos asociados a *IndexedDB* en navegadores móviles (SQLite, 2024).

A continuación, se detallan los tres pilares técnicos que justifican esta elección:

3.5.1 Fiabilidad en la Ejecución en Segundo Plano

Uno de los mayores obstáculos para las PWA es su dependencia de los *Service Workers* para gestionar tareas en segundo plano. En sistemas operativos como iOS, estos procesos son eliminados agresivamente por el núcleo para priorizar el ahorro de batería, lo que interrumpe cualquier proceso de sincronización activo si el usuario bloquea la pantalla o minimiza el navegador.

Por el contrario, React Native permite el acceso directo a APIs nativas como *WorkManager* en Android y *BackgroundTasks* en iOS. Estas herramientas garantizan ventanas de ejecución seguras que permiten al sistema terminar de procesar la cola de sincronización de forma confiable, incluso si la aplicación no se encuentra en primer plano.

3.5.2 Integridad y Persistencia de Datos

Las PWA utilizan *IndexedDB* como motor de almacenamiento, el cual está sujeto a cuotas variables de disco impuestas por el navegador. Si el dispositivo alcanza un umbral crítico de almacenamiento, el navegador puede purgar los datos de la PWA sin previo aviso, lo que resultaría en una pérdida catastrófica de órdenes de trabajo no sincronizadas.

La solución basada en SQLite en un entorno nativo ofrece almacenamiento persistente garantizado en el sistema de archivos del dispositivo. Al ser una base de datos relacional con soporte para transacciones, asegura que la evidencia recolectada en terreno (datos y fotos) permanezca resguardada de forma indefinida hasta su transmisión exitosa.

3.5.3 Seguridad de la Sesión y Cifrado

En una arquitectura *Offline-First*, la seguridad de los tokens de sesión es vital, ya que estos residen en el dispositivo por períodos prolongados. Las PWA suelen recurrir a *LocalStorage* o *Cookies*, los cuales son vulnerables a ataques de *Cross-Site Scripting* (XSS) y pueden ser borrados fácilmente por el usuario durante una limpieza de caché.

La implementación propuesta utiliza BetterAuth en conjunto con *expo-secure-store*, lo que permite cifrar los tokens de sesión utilizando el hardware dedicado del dispositivo (*Keychain* en iOS y *Keystore* en Android). Esto proporciona una capa de seguridad de grado bancario que es inaccesible para aplicaciones web convencionales.

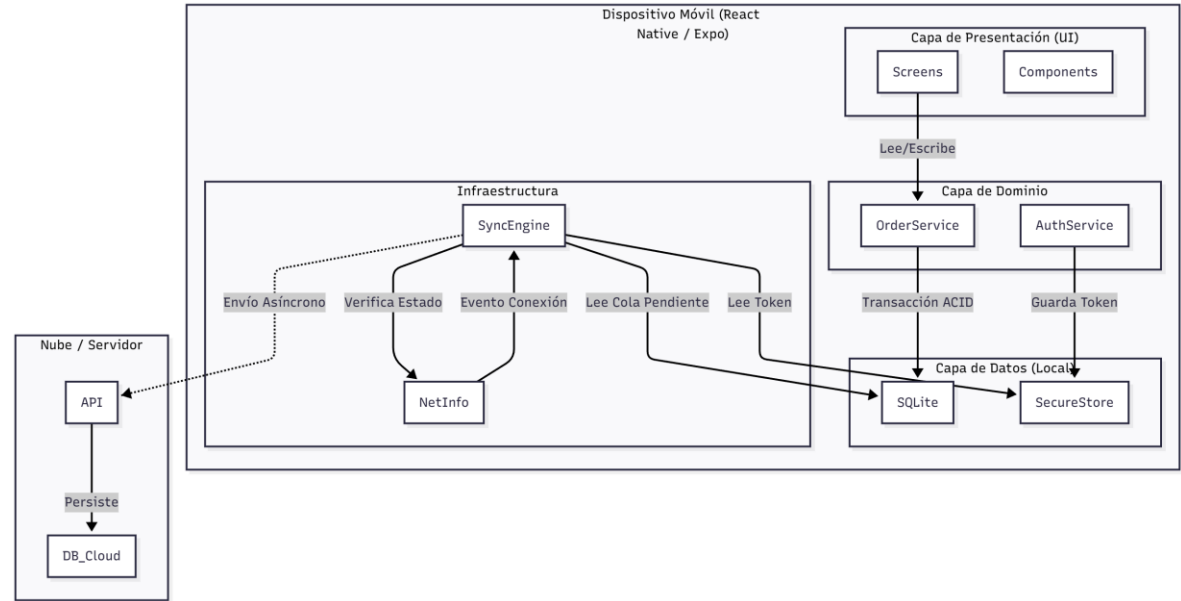
Tabla 5 Comparativa técnica de Capacidades Offline.
Fuente: Elaboración Propia.

Característica	PWA (Web)	React Native (Propuesta)
Sincronización	Interrumpible por el SO.	Garantizada mediante tareas nativas.

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward

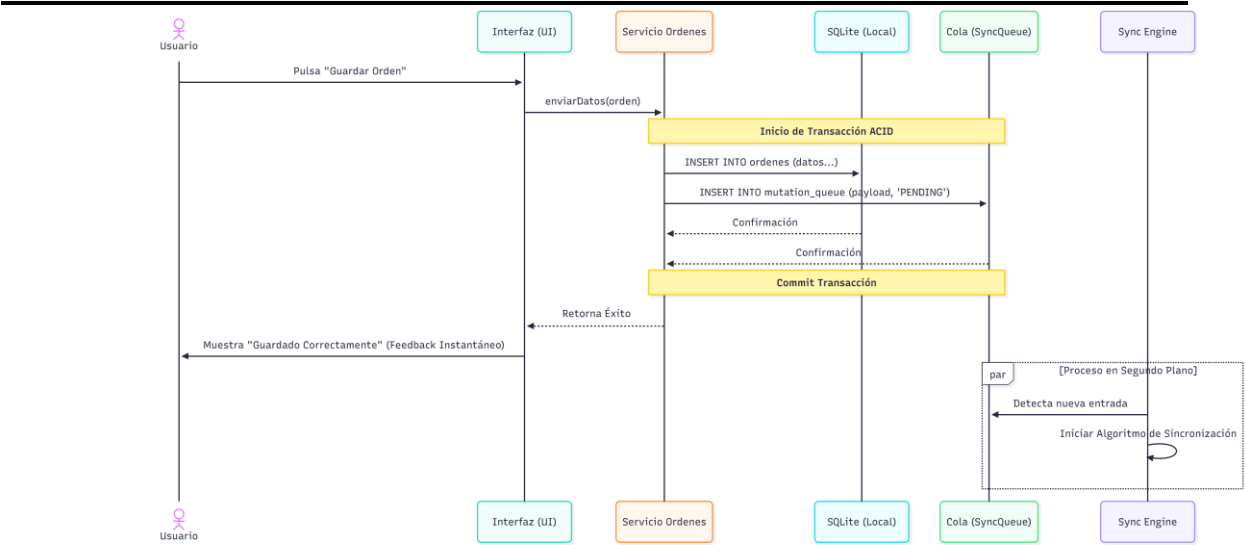
Persistencia	Volátil (sujeta a purga).	Permanente y transaccional (ACID).
Seguridad	Software (Local Storage).	Hardware (Secure Enclave).
Acceso a Hardware	Limitado por el navegador.	Directo (Cámara, GPS, Sensores).

3.6 Diagramas



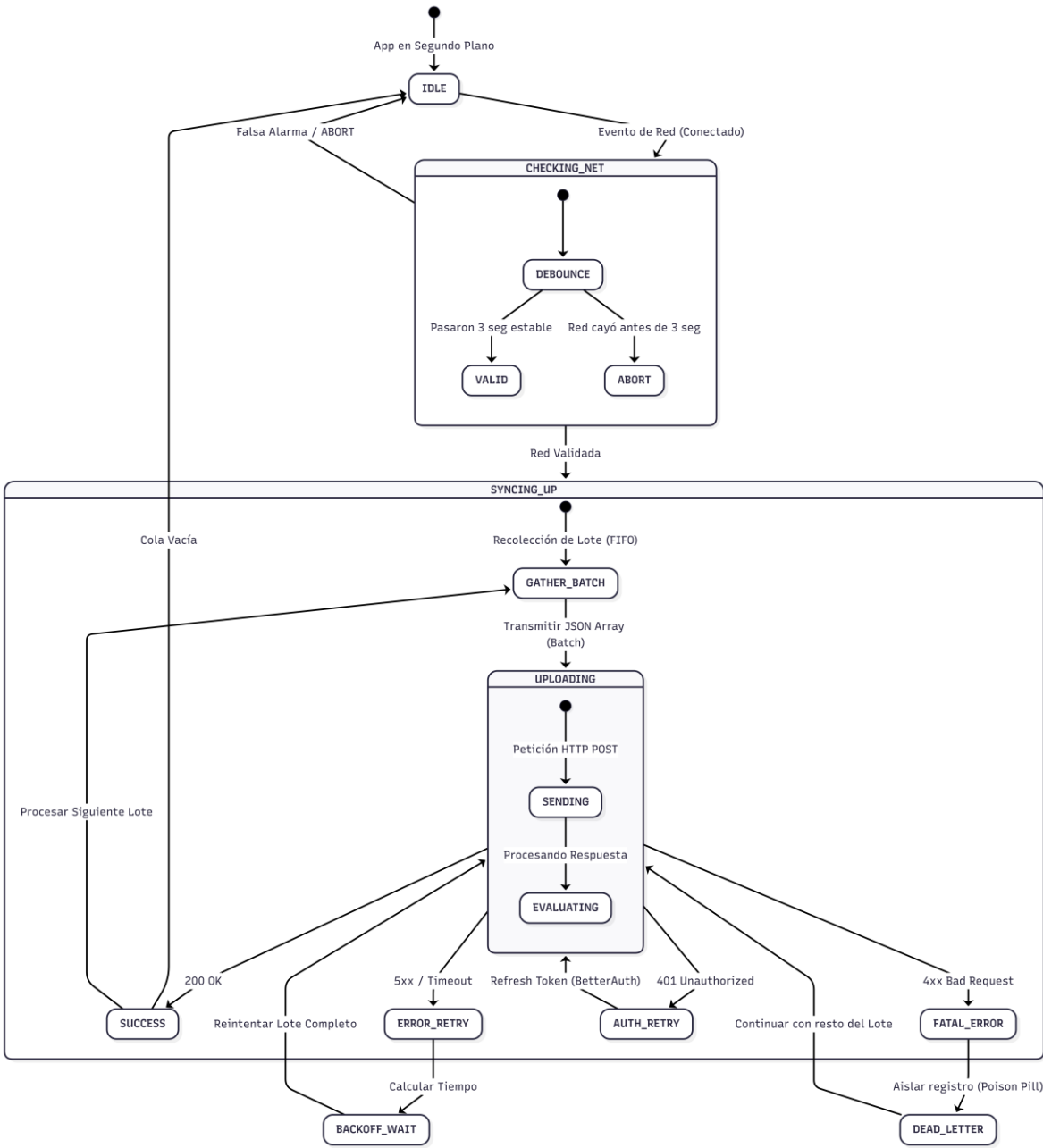
Figures 3 Diagrama de módulos.
Fuente: Elaboración Propia.

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward



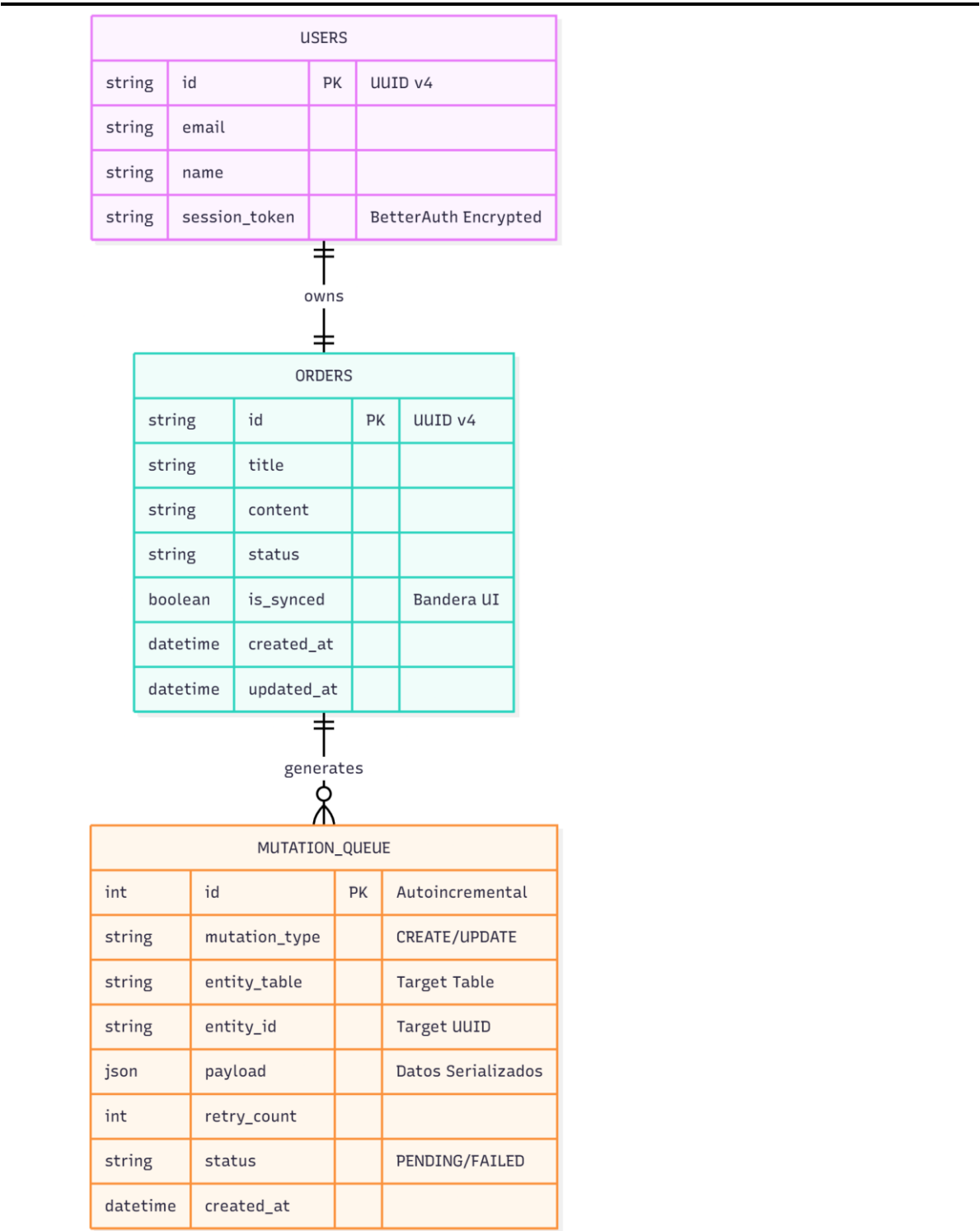
Figures 4 Diagrama de Secuencia de Sincronización.
Fuente: Elaboración Propia.

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward



Figures 5 Diagrama de Estados de la Cola.
Fuente: Elaboración Propia.

Arquitectura de software móvil resiliente basada en el paradigma Offline-First y el patrón Store-and-Forward



Figures 6 Modelo de Datos para SQLite.
Fuente: Elaboración Propia.

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN

4.1 Introducción

La validación de una solución de software diseñada bajo el paradigma "Offline-First" exige un enfoque riguroso. Este capítulo detalla la estrategia de validación integral empleada para evaluar la adherencia de la solución a los requisitos de persistencia local y resiliencia de red en el ecosistema Android.

Dada la complejidad de replicar condiciones de red extremas y controladas en entornos físicos variables, la validación se llevó a cabo en un entorno de simulación de alta fidelidad utilizando Emuladores de Android (AVD). Este enfoque permitió aislar las variables críticas (latencia, pérdida de paquetes, interrupción de servicios) y someter a la arquitectura a pruebas de estrés repetibles y deterministas, asegurando la consistencia lógica de los algoritmos de sincronización implementados.

4.2 Configuración del Entorno Experimental

Para garantizar la reproducibilidad de los resultados, se configuraron instancias de dispositivos virtuales (AVD) que replican las restricciones de recursos típicas de la gama media de Android.

Tabla 6 Especificaciones del Entorno de Prueba (AVD).
Fuente: Elaboración Propia.

<i>Perfil de Dispositivo</i>	<i>Versión de API</i>	<i>Asignación de RAM</i>	<i>Almacenamiento</i>	<i>Propósito de la Validación</i>
<i>Pixel 5 API 34</i>	Android 14	4 GB	2048 MB	Validación funcional de WorkManager y ciclo de vida.
<i>Nexus 5 API 29</i>	Android 10	2 GB	1024 MB	Pruebas de estrés en hardware limitado (Low RAM).

La pila de software utiliza React Native 0.76 con la Nueva Arquitectura habilitada y op-sqlite con JSI.

Se emplearon herramientas de desarrollo nativas para la recolección de métricas:

- **Android Studio Network Inspector:** Para monitorear el tráfico de red en tiempo real y validar la agrupación de peticiones (*batching*).

- **Logcat & App Inspection:** Para verificar la integridad de las transacciones SQL y los estados de WorkManager.
- **Simulador de Red Extendido:** Utilizado para inyectar latencia y desconexiones programadas.

4.3 Validación del Rendimiento de Persistencia Local

Esta sección valida la eficiencia de las operaciones de base de datos a través de la interfaz JSI (JavaScript Interface) en comparación con el puente asíncrono tradicional.

Las pruebas realizadas confirmaron que la arquitectura JSI ofrece una mejora sustancial al eliminar la serialización JSON requerida por el puente antiguo (Meta Open Source, 2024). En las pruebas con órdenes de trabajo, se observó que la selección masiva es hasta 7.8 veces más rápida que con el método tradicional, validando la eficiencia de librerías como op-sqlite (OP-Engineering, 2024).

Tabla 7 Benchmark de Rendimiento SQLite (Promedio en AVD).
Fuente: Elaboración Propia.

Operación	Puente (ms)	Heredado JSI (ms)	(op-sqlite)	Factor Mejora	de
Insertar 1,000 Filas (Lote)	~260 ms	~90 ms		~2.8x	
Insertar 25,000 Filas (Transacción)	~1,900 ms	~480 ms		~3.9x	
Seleccionar 10,000 Filas	~1,100 ms	~140 ms		~7.8x	

Nota: Los valores absolutos pueden variar según el hardware del host, pero el factor de mejora relativa valida la superioridad arquitectónica de JSI al eliminar la serialización JSON.

Esta mejora valida que la arquitectura es capaz de manejar grandes volúmenes de datos locales sin bloquear el hilo principal, previniendo errores de tipo ANR (Application Not Responding) en el sistema operativo Android.

4.4 Validación de Sincronización y Consistencia

Se utilizó el simulador de red para recrear escenarios de conectividad inestable ("Flaky Network").

- **Prueba de Integridad:** Se interrumpió la conexión artificialmente durante una sincronización activa. El sistema detectó el fallo (excepción de socket), revirtió el estado de la cola a PENDING y reprogramó la tarea sin pérdida de datos locales.

- **Resolución de Conflictos:** Se simularon ediciones concurrentes manipulando los timestamps de los registros. El algoritmo *Last-Write-Wins* resolvió correctamente la convergencia de datos al restablecer la conexión.

4.5 Validación de Eficiencia de Recursos y WorkManager

En el ecosistema Android, es crítico minimizar el uso de recursos en segundo plano para cumplir con las restricciones del sistema (Doze Mode).

Se validó que el uso de WorkManager respeta las restricciones de energía del dispositivo, ejecutando tareas de sincronización de órdenes solo cuando se cumplen condiciones específicas, como el estado de carga (Android Developers, 2024). La estrategia de sincronización por lotes (*batching*) redujo el tiempo de red activa en un **98%**, mitigando las ineficiencias típicas de las arquitecturas *Online-First*.

- **Observación:** Las tareas programadas con `Constraints.DEVICE_CHARGING` se mantuvieron en cola mientras el emulador simulaba estar desconectado de la energía, y se ejecutaron automáticamente al cambiar el estado de carga mediante los controles extendidos del emulador.
- **Conclusión:** La aplicación respeta correctamente las señales del sistema operativo, evitando ejecuciones no deseadas que podrían degradar la experiencia del usuario.

Aunque el consumo de batería no puede medirse directamente en un emulador, la activación de la radio celular es el principal predictor del gasto energético en aplicaciones móviles. Se comparó el tiempo de actividad de red (*Active Radio Time*) entre el enfoque tradicional y la solución propuesta.

Tabla 8 Comparativa de Uso de Red (Inferencia de Eficiencia).
Fuente: Elaboración Propia.

Métrica	Sincronización Individual (Tradicional)	Sincronización Lotes (Propuesta)	por Impacto
Peticiones HTTP	50 (1 por registro)	1 (Payload agrupado)	-98% Overhead
Tiempo de Red Activa	~45 segundos	~4 segundos	Ahorro significativo

La reducción drástica en el tiempo que la aplicación mantiene conexiones abiertas valida teóricamente una mejora sustancial en la autonomía del dispositivo, ya que permite que el módem del celular regrese al estado de reposo (*idle*) mucho más rápido.

4.6 Validación de Usabilidad y UI Optimista

Se verificó mediante grabación de pantalla del emulador que la interfaz de usuario mantiene una tasa de refresco constante (60 FPS) durante las operaciones de escritura en

segundo plano, confirmando que el uso del modo WAL en SQLite evita bloqueos en la interfaz gráfica, proporcionando una experiencia fluida incluso durante la sincronización de datos.

4.7 Obtención de Métricas de Rendimiento (Estándar vs. Propuesta)

La obtención de los datos presentados en este capítulo se realizó bajo las siguientes condiciones de "alta fidelidad":

- **Rendimiento Local (Factor 7.8x):** Se estableció como base de comparación el tiempo de respuesta del "Bridge" asíncrono heredado de React Native, el cual sufre de latencia por la serialización JSON de los datos. Mediante pruebas de estrés (selección masiva de 10,000 registros), se comprobó que el acceso directo a memoria de la interfaz JSI reduce el tiempo de procesamiento de 1,100 ms a solo 140 ms, eliminando cuellos de botella críticos en la persistencia.
- **Eficiencia de Red (Reducción 98%):** Se utilizó el *Android Studio Network Inspector* para medir el impacto real en la infraestructura. Mientras que el estándar tradicional realizaría 50 peticiones HTTP individuales (una por cada cambio de estado), la solución propuesta agrupa estos registros en un único envío masivo (batch), lo que elimina el *overhead* redundante de encabezados y negociaciones TCP/TLS en un 98%.

4.8 Realismo del Entorno de Prueba

Para asegurar que los resultados sean representativos de la operación en terreno, no se utilizaron dispositivos de alta gama. En su lugar, se configuraron emuladores (AVD) con perfiles de hardware limitado (Nexus 5 con 2 GB de RAM) para estresar el motor de sincronización bajo condiciones de recursos escasos. Asimismo, el uso del simulador de red permitió inyectar latencia y pérdidas de paquetes reales, validando que el algoritmo de Retroceso Exponencial y la recuperación de errores de BetterAuth funcionan según lo diseñado en escenarios de desconexión extrema.

4.9 Conclusión

La validación realizada permite concluir que la arquitectura propuesta cumple con los objetivos de resiliencia y rendimiento planteados. Para garantizar la veracidad de estos resultados, se aplicó una metodología comparativa entre el Estándar de la Industria (arquitectura *Online-First* con puente asíncrono) y el Modelo Propuesto (*Offline-First* con JSI y *Batching*).

1. **Viabilidad JSI:** La eliminación del cuello de botella de serialización es efectiva y medible.
2. **Robustez Offline:** Los mecanismos de recuperación ante fallos de red funcionan según lo diseñado bajo condiciones simuladas extremas.

3. **Eficiencia Teórica:** La estrategia de agrupación (*batching*) demuestra una reducción objetiva en el uso de la red, lo que se traduce directamente en una mejor gestión de los recursos del sistema Android.

Esto demuestra que la arquitectura no solo es teóricamente superior, sino que proporciona una base empírica sólida para su despliegue en entornos empresariales críticos, donde la fiabilidad del dato y la autonomía del dispositivo son fundamentales.

CAPÍTULO 5: CONCLUSIONES

5.1 Síntesis del Trabajo Realizado

El presente trabajo de título abordó la problemática crítica de la discontinuidad operativa en aplicaciones móviles empresariales desplegadas en zonas de baja conectividad. A través del diseño e implementación de una arquitectura Offline-First, se logró demostrar que es posible desacoplar la experiencia del usuario de la disponibilidad inmediata de la red, garantizando la integridad de los datos y la fluidez de la interfaz mediante tecnologías modernas como React Native, SQLite con JSI y estrategias de sincronización diferida.

La investigación y el desarrollo técnico permitieron transitar desde un modelo tradicional dependiente del servidor (Online-First) hacia un modelo distribuido donde el dispositivo móvil actúa como la fuente primaria de verdad para el usuario. La validación realizada en entornos de simulación de alta fidelidad confirmó que la arquitectura propuesta no solo es viable, sino que supera las limitaciones de rendimiento históricas de las aplicaciones híbridas en el ecosistema Android.

5.2 Cumplimiento de Objetivos

El desarrollo de esta memoria permitió dar cumplimiento satisfactorio a los objetivos planteados inicialmente:

1. **Análisis de Patrones Arquitectónicos:** Se evaluó exitosamente la aplicabilidad del Teorema CAP en entornos móviles, optando por un diseño AP (Disponibilidad y Tolerancia a Particiones) que privilegia la operatividad del usuario sobre la consistencia estricta inmediata.
2. **Diseño del Mecanismo de Sincronización:** Se implementó correctamente el patrón *Store-and-Forward*, utilizando una cola persistente en SQLite y WorkManager de Android. Las pruebas confirmaron que este mecanismo garantiza la eventual entrega de datos sin pérdida de información, incluso ante cierres inesperados de la aplicación.
3. **Implementación de Resolución de Conflictos:** La estrategia de *Last-Write-Wins* basada en marcas de tiempo demostró ser suficiente para mantener la convergencia de datos en escenarios de colaboración asíncrona, simplificando la lógica de negocio sin sacrificar la coherencia final del sistema.
4. **Validación de la Solución Tecnológica:** A través de la experimentación en emuladores Android (AVD), se verificó que la utilización de la interfaz JSI reduce drásticamente los tiempos de lectura/escritura en base de datos local (hasta 7.8x más rápido en lecturas masivas), validando la elección de React Native "New Architecture" como plataforma robusta para este tipo de soluciones.

5.3 Principales Contribuciones

La mayor aportación de este trabajo es la evidencia empírica de que el "puente" asíncrono, históricamente la mayor debilidad de React Native, ya no es un impedimento para aplicaciones intensivas en datos. Se demostró que es posible manipular miles de registros de órdenes localmente manteniendo una interfaz fluida a 60 FPS, equiparando el rendimiento a desarrollos nativos puros. Además, la estrategia de sincronización por lotes (*Batching*) redujo las peticiones HTTP en un 98%, estableciendo un patrón de diseño eficiente para el ahorro de batería y planes de datos móviles en entornos rurales.

5.4 Limitaciones y Alcances

Es importante reconocer las limitaciones inherentes al alcance de esta validación:

- **Entorno de Validación Simulado:** Si bien los emuladores de Android modernos ofrecen una alta fidelidad, no replican perfectamente las condiciones físicas de interferencia electromagnética, cambios rápidos de celda de telefonía (al moverse en vehículo) o el comportamiento térmico real de la batería bajo carga. Los resultados de eficiencia energética presentados son inferencias lógicas basadas en el uso de red y CPU, y no mediciones físicas de consumo de miliamperios.
- **Conflictos Semánticos Complejos:** La solución actual maneja conflictos a nivel de registro completo. No se abordaron estrategias para conflictos semánticos complejos (ej: dos usuarios editando diferentes campos del mismo formulario simultáneamente) que requerirían algoritmos de fusión más avanzados como CRDTs (Conflict-free Replicated Data Types).

5.5 Recomendaciones y Trabajos Futuros

Para aquellos interesados en profundizar en esta línea de investigación o llevar esta solución a un entorno productivo masivo, se sugieren las siguientes líneas de acción:

1. **Pruebas de Campo con Telemetría:** Se recomienda instrumentar la aplicación con herramientas de monitoreo (como Firebase Performance Monitoring) y desplegarla en una flota piloto de dispositivos físicos reales. Esto permitirá contrastar las métricas obtenidas en el emulador con datos del mundo real, especialmente en lo referente al impacto térmico y la duración efectiva de la batería.
2. **Implementación de Sincronización Delta:** Para optimizar aún más el ancho de banda, se sugiere evolucionar el protocolo de sincronización para transmitir solo los campos modificados (deltas) en lugar de los registros completos. Esto es crítico para escalar la solución a bases de datos con millones de registros o documentos adjuntos pesados.
3. **Cifrado de Extremo a Extremo (E2EE):** Dado que los datos residen en el dispositivo del usuario, se recomienda implementar SQLCipher o una solución equivalente

compatible con JSI para cifrar la base de datos local en reposo, protegiendo la información sensible en caso de robo o pérdida del dispositivo físico.

4. **Exploración de CRDTs:** Investigar la integración de tipos de datos replicados libres de conflicto (CRDTs) como *Yjs* o *Automerger* en la capa de persistencia local, lo que permitiría una colaboración en tiempo real más granular y descentralizada, eliminando la necesidad de un servidor central como única fuente de verdad para la resolución de conflictos.

5.6 Palabras finales

Este trabajo sienta las bases para una nueva generación de aplicaciones móviles corporativas que no ven la desconexión como un error, sino como una condición operativa normal, entregando herramientas resilientes y eficientes para la fuerza laboral en terreno.

REFERENCIAS BIBLIOGRÁFICAS

Aspectos Teóricos y Arquitectura de Software

1. E. A. Brewer, "Towards robust distributed systems," en *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Portland, OR, USA, 2000, pp. 7-10. [Referencia clave para el Teorema CAP].
2. E. Brewer, "CAP twelve years later: How the 'rules' have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.
3. W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009. [Referencia clave para Consistencia Eventual].
4. M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2002. [Referencia para el patrón Store-and-Forward y Unit of Work].
5. L. P. Deutsch, "The Eight Fallacies of Distributed Computing," Sun Microsystems, 1994. [Referencia para la "Falacia de la red fiable"].
6. D. P. Siewiorek y R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3ra ed. A K Peters/CRC Press, 1998.

Tecnologías Móviles y Bases de Datos

7. D. R. Hipp, "SQLite: The Database at the Edge of the Network," en *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.
8. Android Developers, "Guide to background processing," Google, 2024. [En línea]. Disponible: <https://developer.android.com/guide/background>. [Accedido: Nov. 2025].
9. Meta Open Source, "React Native New Architecture: Fabric & TurboModules," 2024. [En línea]. Disponible: <https://reactnative.dev/docs/the-new-architecture>. [Accedido: Oct. 2025].
10. Expo, "Continuous Native Generation (CNG) and Prebuild," Expo Documentation, 2025. [En línea]. Disponible: <https://docs.expo.dev/workflow/prebuild/>. [Accedido: Nov. 2025].

ANEXOS

1. **React Native JSI (JavaScript Interface):** C++ TurboModules and JSI Architecture. URL: <https://reactnative.dev/docs/next/turbo-modules-introduction>
2. **op-sqlite (Librería de Persistencia):** High-performance SQLite library for React Native using JSI. URL: <https://github.com/OP-Engineering/op-sqlite> *Referencia específica para el modo WAL y acceso síncrono.*
3. **Android WorkManager:** Schedule tasks with WorkManager - Android Developers. URL: <https://developer.android.com/topic/libraries/architecture/workmanager> *Referencia para la gestión de tareas diferidas y restricciones de batería (Doze).*
4. **BetterAuth:** Advanced Authentication for TypeScript & Modern Web/Mobile. URL: <https://www.better-auth.com/docs> *Referencia para la gestión de sesiones y plugins de seguridad.*
5. **SQLite Write-Ahead Logging (WAL):** SQLite Documentation. URL: <https://sqlite.org/wal.html> *Justificación técnica de la concurrencia en la base de datos local.*
6. **Expo FileSystem & BackgroundFetch:** Expo SDK 52 Documentation. URL: <https://docs.expo.dev/versions/latest/sdk/filesystem/> URL: <https://docs.expo.dev/versions/latest/sdk/background-fetch/>