# Arquitectura de la plataforma táctica web de inteligencia en punto de venta

Version: 1.0 | Fecha: 2025-09-30

## 1. Introducción y objetivos

**Propósito del sistema:** La plataforma táctica web de inteligencia en punto de venta es una aplicación diseñada para apoyar a los asesores de venta en campo, proporcionándoles recomendaciones diarias generadas por IA para cada visita y capturando información de retorno sobre la ejecución de esas recomendaciones. Surge como respuesta a una limitación crítica: la aplicación móvil de rutas existente ("Route") no podía modificarse para registrar la adopción de las recomendaciones de IA. Sin esta herramienta, los pilotos de IA (enfocados en aumentar el *Drop Size* y la conversión a envases *No Retornables*) no podrían medirse científicamente en campo. La WebApp propuesta permite **ejecutar y medir los pilotos sin impactar los sistemas actuales**, funcionando de forma paralela a la operación para **recopilar datos clave de efectividad**.

**Metas de negocio:** La solución habilita dos pilotos de inteligencia artificial enfocados en mejorar indicadores estratégicos de Ferguez: (1) aumentar el *Drop Size* (volumen por entrega) mediante predicción de demanda y recomendaciones de portafolio, y (2) elevar la tasa de conversión a *No Retornable* (envases no retornables) mediante iniciativas dirigidas. Adicionalmente, captura **inteligencia de mercado** en cada punto de venta (quiebres de stock, precios de la competencia, ejecución en tienda, etc.), alimentando con datos reales a los modelos y a la estrategia comercial. En resumen, el sistema debe **empoderar al asesor** con información accionable en cada visita y **retroalimentar a la empresa** con datos de campo para validar y refinar la estrategia de IA.

**Objetivos de calidad (Quality Goals) más importantes:**

* **Confiabilidad offline:** La aplicación debe ser altamente **resiliente a la falta de conectividad** en campo. Incluso sin internet, el asesor puede acceder a toda su ruta y registrar cada interacción sin pérdida de datos. Al reconectarse, los datos se sincronizan automáticamente. **Este objetivo es crucial**: la recolección de datos no debe fallar, ya que la confianza del asesor depende de que su trabajo nunca se pierda. De hecho, el diseño *offline-first* garantiza la continuidad operativa y la integridad de los datos, cumpliendo el objetivo principal de medir la efectividad de los pilotos **sin perder ninguna interacción registrada**.
* **Usabilidad en campo:** La interfaz debe ser **simple, clara y de rápida interacción** para asesores ocupados. Se sigue el principio de **Minimizar la Carga Cognitiva**: en cada pantalla se muestra solo la información esencial para la próxima acción. Los asesores suelen estar multitarea (manejando, tratando con clientes), por lo que la app debe **agilizar su flujo de trabajo**, con botones grandes y visibles, textos legibles bajo luz solar y flujos intuitivos. El entrenamiento requerido debe ser mínimo; un asesor nuevo debe entender qué hacer inmediatamente al usar la app. Este enfoque de **UX móvil-first** busca maximizar la adopción y reducir errores humanos.
* **Flexibilidad y escalabilidad:** La solución debe ser **flexible** para incorporar nuevos tipos de recomendaciones y escalable para más rutas en el futuro. La arquitectura de datos es genérica y extensible – por ejemplo, el objeto de **Recomendación** es polimórfico y soporta distintos tipos de acciones de negocio (alertas de quiebre de stock, sugerencias de portafolio, promociones, argumentos de venta, iniciativas de conversión, etc.) sin requerir cambios estructurales. Asimismo, la plataforma se construye con tecnologías modernas y *serverless* que facilitan escalar de un piloto limitado a un despliegue completo: puede manejar más usuarios, datos y modelos de IA futuros simplemente añadiendo capacidad en la nube, sin re-arquitectura fundamental.

**Stakeholders (interesados) principales:**

| **Stakeholder,Rol** | **Interés** |
| --- | --- |
| Asesores de Venta (Usuarios) | Personal de campo que usa la WebApp a diario. Les interesa una herramienta rápida, confiable (offline) y fácil de usar que les ayude a planificar visitas y mejorar sus ventas. Buscan que la app les agregue valor sin entorpecer su rutina. |
| Gerentes de Ventas | Supervisores que buscan aumentar indicadores (volumen de venta, conversiones a NR). Esperan que la plataforma entregue recomendaciones efectivas a sus asesores y provea datos para medir el impacto en resultados. |
| Equipo de Ciencia de Datos (IA) | Responsables de los modelos predictivos/pilotos. Quieren una **retroalimentación confiable** desde campo para validar sus predicciones. La WebApp es el medio para cerrar el ciclo de aprendizaje: envían recomendaciones y reciben feedback estructurado. |
| Área de TI de Ferguez | Dueños de la infraestructura y sistemas corporativos. Les interesa que la solución se integre sin afectar sistemas legados, cumpla políticas (e.g. uso de GCP) y sea segura. También evaluarán la viabilidad de escalarla tras el piloto. |
| Interware (Equipo Implementador) | Equipo de desarrollo/arquitectura que construye la solución. Deben asegurar que el diseño cumpla con los requisitos en un plazo corto (6 semanas) y siente bases sólidas para la siguiente fase. Están interesados en una arquitectura limpia, mantenible y que pueda demostrar el valor del piloto. |

## 2. Restricciones

En el desarrollo de la solución se identificaron varias **restricciones** que acotan las decisiones de arquitectura:

* **Restricciones técnicas (tecnologías obligadas o vetadas):** Por lineamiento del proyecto y para minimizar riesgos, se definió desplegar en la nube de **Google Cloud Platform (GCP)** usando servicios *serverless* (Cloud Run). La aplicación debe residir en el proyecto GCP de Ferguez, respetando sus políticas de seguridad y accesos. En frontend se optó por **React** (con build tool Vite) para lograr una web responsiva y moderna. En backend se recomendó **Python con FastAPI** (framework ligero async) como primera opción, aunque se podría usar Java/Quarkus si fuera necesario. No se consideran tecnologías obsoletas ni soluciones on-premise; la rapidez del piloto exige stack web moderno y un *deployment* ágil en la nube. Además, por la naturaleza offline-first, es obligatorio el uso de tecnologías web que soporten **PWA (Progressive Web App)**, es decir, capacidades de almacenamiento local (IndexedDB) y Service Workers en el navegador (disponible en navegadores móviles modernos). No sería viable una aplicación web tradicional sin estas capacidades.
* **Restricciones organizacionales:** No es posible modificar ni depender de la **app móvil "Route" existente** utilizada por asesores. Esto impone crear una solución completamente separada (*standalone*) para los pilotos de IA. Asimismo, el proyecto piloto tiene un **plazo fijo de 6 semanas** para diseño, desarrollo y pruebas, lo que fuerza decisiones pragmáticas: se prioriza el *time-to-market* sobre la implementación de funcionalidades no esenciales. Por ejemplo, se descartó desarrollar una aplicación nativa (iOS/Android) debido al tiempo y complejidad que implicaría; en cambio, se opta por una aplicación web multiplataforma que se comporta como app móvil. Del lado de TI de Ferguez, se requiere coordinación para obtener **acceso de lectura a la base de datos** que contiene la "Hoja de Ruta" (itinerario de visitas). Esta dependencia externa condiciona el cronograma: la integración con dicha fuente de datos debía gestionarse prioritariamente (semana 1-2) dado que es insumo crítico. Otra restricción relevante es la **capacidad de los dispositivos en campo**: se asume que los asesores usarán smartphones corporativos o personales con un navegador moderno (Chrome/Safari actualizados) y conexión de datos móvil. La solución debe funcionar en estos dispositivos móviles con pantallas pequeñas, conectividad intermitente y recursos limitados (CPU, batería), lo cual influye en decisiones de diseño (UI simplificada, operaciones pesadas delegadas al backend o pre-cálculo en la nube).
* **Restricciones legales/corporativas:** Aunque el piloto maneja principalmente datos de negocio (ventas, inventarios, recomendaciones) y no datos personales sensibles, se debe cumplir con políticas básicas de privacidad y seguridad corporativa. Todos los datos transmitidos entre el dispositivo y la nube deben viajar cifrados (HTTPS) y almacenarse de forma segura en GCP (cumpliendo estándares de la plataforma). Adicionalmente, si bien no se explícita en la documentación, es razonable suponer que Ferguez tiene lineamientos de protección de información competitiva; por ello, el acceso a la aplicación estará restringido a usuarios autorizados (asesores de las rutas piloto) mediante credenciales únicas. No se pueden exponer públicamente las recomendaciones ni los datos de inteligencia recopilados sin control.

En síntesis, estas restricciones enmarcan la solución en un **entorno tecnológico específico (React/Python sobre GCP)**, con un **alcance funcional acotado al piloto** y una fuerte necesidad de **operar offline** en dispositivos móviles existentes, todo ello bajo **colaboración con sistemas corporativos** ya establecidos.

## 3. Contexto y alcance

**Ámbito del sistema:** La **Plataforma Táctica Web** abarca la aplicación web (frontend + backend) y su base de datos en la nube, cuyo propósito es intermediar entre los **asesores en campo** y las **fuentes de inteligencia** (modelos de IA y datos de la empresa). Esta plataforma se coloca en el ecosistema tecnológico de Ferguez como un **satélite** temporal: interactúa con algunos sistemas existentes, pero mantiene su independencia para no afectar las operaciones productivas. A continuación se detalla el contexto de alto nivel, incluyendo los actores externos y las interfaces principales.

flowchart LR  
 usuario[Asesor de Ventas<br/>(Usuario en campo)] -- Interacción vía Navegador Web --> webapp[\*\*WebApp Inteligencia<br/>en Punto de Venta\*\*]  
 webapp -- Lee lista de visitas<br/>("Hoja de Ruta") --> rutaDB[(DB Corporativa<br/>Hoja de Ruta)]  
 webapp -- Obtiene recomendaciones<br/>personalizadas --> iaAPI[(Servicio Piloto IA<br/>API de Recomendaciones)]  
 webapp -- Envía feedback<br/>de campo --> iaAPI

*Diagrama 3.1: Diagrama de contexto (nivel alto) – La WebApp (en el centro) es el sistema bajo diseño. Los asesores (usuarios humanos) interactúan con ella a través de un navegador web en sus dispositivos móviles. La WebApp a su vez se comunica con* ***dos sistemas externos clave****: (1) la fuente de datos de la* ***Hoja de Ruta*** *de Ferguez (que provee la lista de clientes a visitar cada día) y (2) el* ***Servicio Piloto de IA*** *que expone las recomendaciones generadas y recibe el feedback. Aislando estos componentes, el alcance de la WebApp* ***incluye*** *la presentación de información al asesor, la lógica de sincronización offline, la captura de feedback e inteligencia de mercado, y el almacenamiento de esos datos.* ***Fuera del alcance*** *quedan las funcionalidades internas de los sistemas externos: por ejemplo, el cálculo de las recomendaciones (lo realiza el módulo de IA separado) o la generación de la ruta del día (lo hace el sistema logístico de Ferguez). La WebApp consume y produce datos, pero no implementa la lógica de negocio de esos sistemas legacy.*

**Interfaces externas principales:**

* **Interfaz A – Base de Datos "Hoja de Ruta":** Provee la lista de visitas del día por asesor. La WebApp necesita leer esta información al inicio de la jornada para saber qué clientes debe atender cada asesor. Tecnológicamente, se espera que Ferguez exponga este dato mediante una vista o tabla en sus sistemas (posiblemente en la base mbaFerguez o dbGpoFernandez mencionadas). La integración probablemente se realice vía una conexión segura a la base de datos corporativa (consulta SQL en modo lectura). **Responsabilidad:** El equipo de Ferguez debe proporcionar credenciales y acceso de solo lectura a esta base. La WebApp actuará como **cliente** de datos, sin escribir nada en los sistemas de Ferguez (evitando impacto en operaciones).
* **Interfaz B – API de Recomendaciones IA:** Es el punto de integración con la inteligencia artificial del piloto. Consiste en un servicio web (REST API) expuesto por el **Sistema Piloto de IA** desarrollado en paralelo. La WebApp invoca un **método GET** para obtener el **plan de ruta enriquecido** (listado priorizado de clientes con recomendaciones) al inicio del día, y luego envía eventos de **feedback** mediante un **método POST** cada vez que el asesor registra el resultado de una recomendación. El contrato esperado es JSON sobre HTTP (p. ej., GET /plan-de-ruta/{ruta\_id}/{fecha} y POST /feedback). **Responsabilidad:** El equipo/servicio de IA debe implementar y mantener esta API según lo acordado (formatos de datos, disponibilidad). La WebApp confía en que la API esté disponible en tiempo real para la sincronización matutina y que pueda recibir múltiples llamadas de feedback durante el día. En caso de indisponibilidad, la WebApp debe manejar errores (por ejemplo, reintentos o notificar al usuario si no pudo descargar recomendaciones).
* **Actor usuario (asesor de ventas):** Si bien no es una "interfaz tecnológica", el asesor es un actor externo esencial. La interacción se da vía la interfaz de usuario de la WebApp en el navegador móvil (no requiere instalación desde tienda, aunque puede funcionar como PWA). Los asesores deben autenticarse con un identificador único (ej. número de empleado) y posiblemente una contraseña. Después, usan la aplicación para ver su lista de clientes y recomendaciones, y para ingresar datos de retorno. Desde el punto de vista del sistema, el asesor genera eventos (clics, inputs) que desencadenan las funciones clave: sincronización inicial, navegación entre pantallas, registro de feedback, etc. Esta interfaz humano-computadora ha sido diseñada con especial cuidado en usabilidad dado el entorno de uso (móvil, exterior, multitarea).

Además de las interfaces anteriores, el sistema interactúa con **APIs del dispositivo móvil** del asesor, por ejemplo la API de **Geolocalización** (GPS) del navegador, utilizada para registrar coordenadas al finalizar cada visita (esto se considera parte de la funcionalidad de la WebApp y no un sistema externo propiamente dicho). No hay integración directa con la aplicación móvil existente de rutas ni con otras herramientas de Ferguez (p.ej. no se modifica SAP, CRM u otros sistemas durante el piloto).

**Fronteras y alcance:** En resumen, **dentro** del sistema está toda la lógica de presentación y captura de datos, la sincronización offline y el almacenamiento en la nube propia (Firestore) para trazabilidad. **Fuera** del sistema permanecen los orígenes de datos (sistemas de Ferguez) y los destinos finales de análisis (la WebApp entrega feedback a IA y guarda en BD, pero la interpretación de resultados piloto se hará fuera, probablemente analizando los datos recopilados). Esta claridad de límites previene extender el alcance del piloto más allá de lo acordado: la WebApp **no** reemplaza sistemas de ruta existentes, **no** hace cálculos de predicción, **no** genera reportes gerenciales (por ahora), sino que se enfoca en ser la herramienta táctica de campo para habilitar y medir los pilotos de IA.

## 4. Estrategia de la solución

La estrategia arquitectónica se centra en **agilidad, aislamiento y confiabilidad**. Se decidió implementar una **arquitectura cliente-servidor web clásica**, pero con adaptaciones modernas para cumplir los objetivos de offline y rápida iteración. A grandes rasgos, el sistema se compone de:

* Un **frontend web responsivo** (SPA – Single Page Application) desarrollado con React, optimizado para móviles, que puede operar offline mediante capacidades PWA.
* Un **backend ligero** desplegado en la nube (FastAPI en Cloud Run) exponiendo servicios REST para autenticación, sincronización de datos y registro de eventos.
* Un **almacenamiento NoSQL en la nube** (Firestore) para persistir los datos recolectados (feedback, inteligencia de mercado, etc.).

**(a) Enfoque offline-first:** La decisión más importante fue adoptar un patrón **Offline-First** para garantizar operatividad en campo. Esto significa que la aplicación **descarga todos los datos necesarios al inicio del día** y los almacena localmente (en el dispositivo) antes de que el asesor salga a ruta. Internamente, utilizamos tecnologías de PWA:

* **IndexedDB en el navegador** para guardar de forma persistente el plan de ruta y las recomendaciones obtenidas del servidor, así como cada entrada de feedback que el usuario vaya generando. De este modo, si la página se recarga o el teléfono se reinicia, la información sigue disponible localmente (no depende de memoria volátil). Cada registro de feedback primero se escribe en una cola local marcada como "pendiente de sincronizar".
* **Service Worker** para manejar la aplicación offline. Un *service worker* intercepta las peticiones de red: si el asesor pierde conexión, la app no mostrará errores de navegador sino que servirá contenido desde la cache (la "cáscara" de la aplicación está cacheada). En esencia, una vez que el usuario ha cargado la WebApp exitosamente (por ejemplo, la mañana en oficina con WiFi), el Service Worker permite seguir usándola aunque luego no haya internet. También detecta cuando vuelve la conexión, activando procesos de sincronización en segundo plano (push de datos pendientes).

Con esta estrategia, **se garantiza que ningún dato se pierda** por falta de conectividad. Si el asesor registra información offline, la app la guardará y mostrará claramente que está pendiente de envío, y tan pronto haya red, la enviará automáticamente. Esto responde directamente al requisito de confiabilidad: *"La página se recargó y toda su información sigue ahí, exactamente como la dejó. No se perdió nada."*. La consecuencia de esta decisión es un aumento en la complejidad del frontend (se tuvo que implementar almacenamiento local y lógica de sincronización), pero fue un trade-off aceptado dado que la **experiencia offline es no negociable** para el éxito del piloto.

**(b) Separación de responsabilidades – WebApp vs sistemas legados:** Para no arriesgar la estabilidad de sistemas existentes, la solución se planteó **desacoplada**. Todas las nuevas funcionalidades (mostrar recomendaciones, registrar adherencia, capturar inteligencia de mercado) se implementan en la WebApp, **sin requerir cambios a la aplicación móvil actual** ni a bases de datos transaccionales de producción. Esto se logró adoptando una arquitectura *sidecar*: la WebApp actúa en paralelo al flujo normal de visitas. El asesor usará esta nueva app para guiar y registrar su visita, mientras posiblemente sigue usando la herramienta actual para otros procesos (ej. toma de pedido). La integración con la empresa es mínima y controlada: lectura de la lista de clientes del día (read-only) y envío de datos a una API de IA diseñada específicamente para el piloto. Esta estrategia minimiza riesgos a sistemas core; en caso de problema, se puede aislar o apagar la WebApp sin afectar la operación principal.

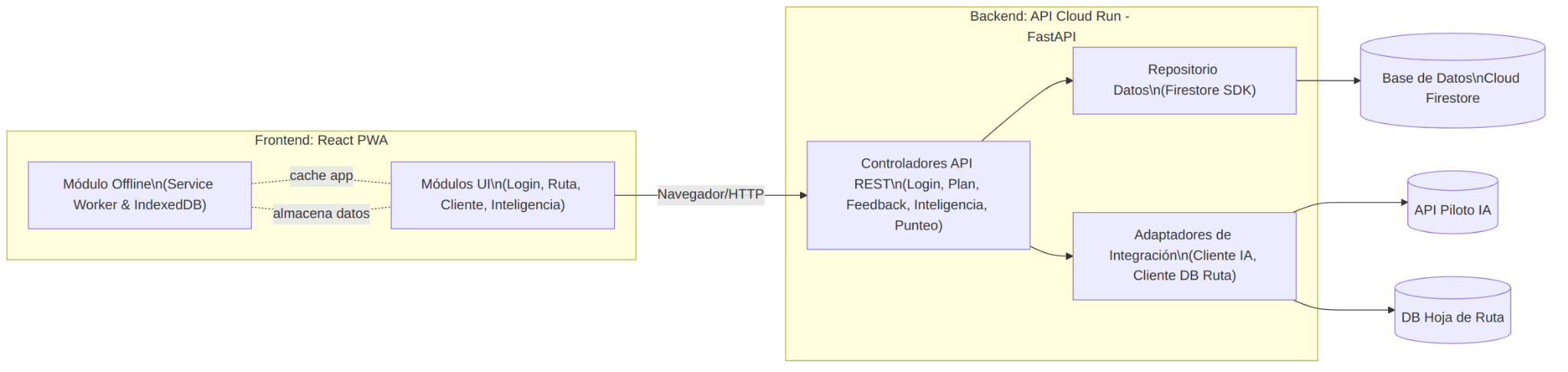
**(c) Patrones y decisiones tecnológicas clave:**

* **Arquitectura de capas (frontend-backend-db):** Se optó por una arquitectura monolítica de 3 capas, suficiente para el alcance del piloto. El **frontend (capa de presentación)** es una SPA React que brinda una UX rica, con routing interno para las pantallas principales (login, lista de ruta, detalle de cliente) y componentes reutilizables (tarjetas de cliente, tarjetas de recomendación, modales de confirmación, etc.). La **capa de negocio/servidor** reside en un servicio FastAPI que expone endpoints RESTful. Este backend encapsula la lógica de integración: por ejemplo, maneja la petición de plan diario combinando datos de distintas fuentes y valida/autentica las solicitudes. La **capa de datos** incluye la base Firestore y conexiones a externos. Esta separación favorece la mantenibilidad: los cambios en UI (ej. rediseño UX) no afectan la lógica central, y viceversa. Además soporta escalabilidad: podríamos replicar el backend o dividir servicios en microservicios a futuro si crece el alcance, pero durante el piloto mantener un monolito simplifica el desarrollo.
* **Elección de React/Vite en el frontend:** Se escogió React por su amplio ecosistema móvil/web y porque los desarrolladores pueden producir rápidamente una interfaz dinámica y responsiva. Con React se implementan fácilmente componentes de estado local para manejar la interacción offline (por ejemplo, indicando qué elementos están pendientes de sincronizar). La combinación con **Vite** permite tiempos de carga rápidos y un empaquetado optimizado. Un **beneficio adicional** es la posibilidad de convertir la app en **PWA instalable** en el teléfono: React produce un SPA que, con manifest.json adecuado y service worker, el asesor puede **"Agregar a pantalla de inicio"** y usar casi como una app nativa. Alternativas consideradas fueron *frameworks* como Angular o Vue, pero el equipo tenía mayor familiaridad con React y dado el horizonte corto se fue a lo seguro. Otra alternativa era desarrollar **aplicaciones nativas** (Android, iOS) para offline, pero se descartó por el costo-tiempo. La ruta web/PWA es más rápida de iterar y multiplataforma por diseño.
* **Backend en FastAPI (Python):** Python fue recomendado por ser ágil para construir APIs rápidamente y porque el equipo de IA trabaja en Python (facilitando compartir modelos o lógica si hiciera falta). FastAPI aporta rendimiento gracias a su naturaleza async y su compatibilidad con uvicorn, suficiente para la carga esperada (pocos usuarios concurrentes en piloto). Además, Python tiene librerías sencillas para conectarse a Firestore (SDK Google) y potencialmente a bases de datos SQL de forma directa. Se consideró **Java Quarkus** como opción (posiblemente por afinidad de sistemas corporativos con Java), pero se priorizó Python por rapidez de desarrollo en un MVP. El backend expone endpoints REST definidos claramente (p. ej. GET /api/plan, POST /api/feedback, POST /api/inteligencia) y maneja internamente llamadas a la API de IA y consultas a la base de ruta. Al ser stateless (sin sesión servidor, salvo validación básica de login), se despliega fácilmente en contenedor.
* **Datos: Firestore NoSQL vs SQL:** Se decidió usar **Cloud Firestore** (base de datos NoSQL de GCP) para persistir los datos generados por la WebApp. Firestore ofrece esquema flexible (útil dado que los tipos de recomendaciones y eventos pueden evolucionar), escalabilidad automática y fácil integración serverless. Dado que el volumen de datos del piloto es modesto (decenas de registros por día), Firestore soporta las escrituras sin problema. Además, si quisiéramos acceso offline desde el frontend, Firestore tiene capacidades de sincronización local; sin embargo, en esta arquitectura se decidió que el **frontend maneja su propio almacenamiento local (IndexedDB)**, y el **backend** es quien interactúa con Firestore. Otra razón de elegir Firestore fue **no cargar la base de datos corporativa** con escrituras de piloto – se mantuvo aislado en una base en la nube. Alternativas evaluadas: utilizar una base SQL en GCP (Cloud SQL/PostgreSQL) o incluso BigQuery para almacenar el feedback. Un SQL relacional se descartó por sobreingeniería en este punto (el esquema de datos aún puede cambiar y un NoSQL es más adaptable). BigQuery se contempló para análisis final de los datos piloto, pero no es apto como base transaccional de la aplicación en tiempo real. Firestore en cambio brinda baja latencia, API sencilla y almacenamiento directamente en el entorno GCP de Ferguez, alineado con la política cloud.
* **Comunicación e integración:** Toda la comunicación entre componentes es mediante **interfaces REST/HTTP con JSON**, un patrón simple y ampliamente compatible. La WebApp (frontend) se comunica con su backend usando fetch/AJAX a endpoints REST; el backend se comunica con la API de IA igualmente con solicitudes HTTP salientes. Este enfoque uniformiza la integración y permitió utilizar herramientas como **Swagger/OpenAPI** (que FastAPI genera automáticamente) para documentar los servicios. Se contempló la posibilidad de integración más directa entre la WebApp y la API de IA (por ejemplo, que el frontend llamara directamente al servicio de IA sin pasar por backend), pero se prefirió que el **backend actúe como *facade*** o mediador. Esto porque el backend puede así realizar lógica adicional: por ejemplo, combinar datos de ruta y recomendaciones si vinieran separados, aplicar transformaciones o asegurarse de almacenar también en Firestore cualquier información relevante. Mantener el frontend libre de llamadas a múltiples orígenes simplifica también el manejo de errores en un solo lugar (backend). La API de IA se diseñó **pasiva** y de único punto de acceso para plan del día, lo cual simplificó el consumo desde el backend (una sola llamada GET para obtener todo el paquete de información necesario por asesor).
* **Estrategia de seguridad básica:** Se adoptó un esquema simple de **autenticación por login** para restringir el acceso solo a asesores autorizados. Cada asesor tiene un identificador (número de empleado) y una contraseña, que se validan al iniciar sesión. Dada la naturaleza temporal del piloto, no se integró un SSO corporativo complejo; en su lugar, el backend valida las credenciales contra una pequeña tabla de usuarios (posiblemente almacenada en Firestore mismo o configurada). Tras login exitoso, el frontend obtiene autorización para usar la app (se podría utilizar un token JWT para persistir la sesión, aunque en este MVP quizás basta con mantener el estado en el cliente). Esta seguridad, aunque básica, fue considerada suficiente para el piloto controlado. Todas las comunicaciones viajan cifradas por HTTPS. No se implementó autenticación a nivel de API de IA (se asume red segura o token estático acordado entre backend y servicio IA). En producción sería necesario fortalecer esto, pero estratégicamente para el MVP se dio prioridad a funcionalidades núcleo.

En resumen, la estrategia de solución fue construir rápidamente una **WebApp modular y autónoma** que explota lo mejor de las tecnologías web modernas (PWA, cloud) para cumplir los requisitos de offline y UX, mientras mantiene integraciones **mínimas y limpias** con el ecosistema existente. Cada decisión (React/PWA, FastAPI, Firestore, REST) estuvo alineada con lograr en tiempo récord un sistema que **funciona incluso sin conexión, es fácil de usar en campo, y no pone en riesgo la operación actual**.

## 5. Vista de bloques de construcción (componentes)

Esta sección describe la estructura estática de la solución: los módulos principales que la componen y sus relaciones. La WebApp se puede conceptualmente dividir en **dos subsistemas** (frontend y backend), cada uno con sus componentes internos, más una base de datos en la nube. A continuación, se presenta un diagrama de componentes y se detallan sus responsabilidades.



*Diagrama 5.1: Componentes lógicos principales de la WebApp. El* ***Frontend*** *(izquierda, ejecutándose en el navegador del asesor) incluye módulos de interfaz de usuario (pantallas y componentes UI) y un módulo de soporte offline (service worker y DB local). El* ***Backend*** *(derecha, ejecutándose en la nube) consiste en controladores API que implementan los endpoints, componentes de integración para conectarse con sistemas externos (API de IA y base de datos de ruta) y un repositorio para acceder a la base de datos Firestore. Los bloques externos (en gris) representan sistemas fuera del límite de la WebApp.*

A continuación se describen estos componentes en mayor detalle:

**Frontend – Módulos UI:** Es la colección de vistas y componentes React que conforman la interfaz. Se implementa como una aplicación de una sola página (SPA) con varias **secciones/pantallas:**

* **Pantalla de login:** Formulario inicial donde el asesor ingresa su número de empleado y contraseña. Tras autenticación, inicia la sincronización de datos.
* **Pantalla "Mi ruta":** Vista tipo *dashboard* que lista los clientes del día en orden priorizado. Cada cliente se muestra en una **Tarjeta de Cliente** con su nombre, la razón principal de visita (recomendación nivel 1, el "Porqué"), e indicadores de estado (si la visita ya fue completada o está pendiente). Esta lista es *scrollable* y permite al asesor tocar un cliente para ver detalles. La interfaz prioriza la información esencial: quién visitar y por qué, evitando abrumar con detalles. Incluye un icono de estado de sincronización en el encabezado (verde, amarillo o rojo según conexión y pendientes).
* **Pantalla de detalle de cliente:** Muestra la información completa para la visita seleccionada. Consta de una sección destacada con el **motivo principal de la visita (Porqué)**, explicando brevemente por qué ese cliente es prioritario (por ejemplo, "Cliente con 2 semanas sin comprar, riesgo de abandono"). Debajo, se listan las **recomendaciones accionables (Qué hacer)**, usualmente 2 a 3 **Tarjetas de Recomendación** cada una con: título (ej. "Sugerir Portafolio Adicional"), descripción detallada de la acción sugerida (ej. "Ofrecer 2 cajas de Tecate..."), y un conjunto de botones para registrar feedback rápido. Por ejemplo, en una tarjeta se pueden presentar botones "✅ Se Aplicó", "🗣️ Cliente Acepta", "❌ Cliente Rechaza", según la configuración de feedback para esa recomendación. Estos botones permiten en uno o dos toques capturar si el asesor siguió la recomendación y cuál fue la respuesta del cliente.
* **Pantalla de captura de inteligencia de mercado:** Módulo adicional donde el asesor puede ingresar datos estratégicos no relacionados con una recomendación específica. Por ejemplo, reportar un **quiebre de stock** de un producto (selecciona SKU y motivo), estimar el **inventario de productos clave** en el anaquel (selecciona niveles para ciertos SKUs y precios), registrar presencia de **productos de la competencia** (marcar si hay X marca, su precio, etc.), y evaluar la **ejecución en tienda** (visibilidad, material POP, etc.). Cada sub-módulo de esta pantalla tiene su propio pequeño formulario/UI. Estos datos ofrecen contexto competitivo y operativo, y se envían al servidor de forma similar al feedback.
* **Componente finalizar visita:** Un botón disponible en la vista de cliente para **cerrar la visita** una vez terminada. Al presionarlo, la app puede capturar la hora actual y la geolocalización GPS del asesor en ese momento (si el permiso de ubicación está otorgado). Esto genera un evento de "**punteo**" (check-out) que se registra para saber que la visita fue realizada, guardando timestamp y coordenadas.

**Responsabilidades del frontend UI:** Presentar los datos de ruta y recomendaciones de forma clara (aplicando las reglas de UX definidas) y recoger con mínima fricción las interacciones del asesor. Debe guiar el flujo: sincronizar al inicio, luego permitir navegar cliente a cliente, y asegurar que nada queda sin registrar al finalizar el día. Asimismo, informar al usuario del estado del sistema en todo momento (cargando, error, offline, pendiente de sync, etc.) siguiendo el principio de **Visibilidad del Estado**. Otra responsabilidad es aplicar validaciones locales para prevenir errores: por ejemplo, no permitir enviar feedback duplicado, requerir todos los campos necesarios en inteligencia de mercado, etc., antes de marcar una visita como completada.

**Frontend – Módulo offline (Service Worker & IndexedDB):** Este componente no es visible para el usuario, pero es fundamental para la arquitectura. Incluye:

* El **Service Worker (SW)** registrado por la aplicación. Su responsabilidad es cachear recursos estáticos de la app (HTML, JS, CSS) para permitir carga offline, interceptar las solicitudes de red y responder con datos cacheados cuando no haya conexión. También puede implementar una lógica de **background sync**: por ejemplo, escuchar eventos de "vuelve la conexión" y lanzar la sincronización de datos pendientes. En nuestra app, el SW se encarga de mantener una copia local de la última versión del shell de la aplicación y de despertar periódicamente (o en eventos de conexión) para enviar datos.
* La **base de datos local IndexedDB**, accesible vía código del frontend. La app crea un almacén (p. ej. *"offlineDB"*) con tablas para: PlanDeRuta (almacena el objeto completo de la ruta del día con sus clientes y recomendaciones) y FeedbackPendiente (cola de eventos de feedback/inteligencia capturados cuando offline). Cada vez que llega el plan inicial desde el servidor, el frontend lo guarda en IndexedDB. Y cada vez que el usuario registra un feedback, antes de intentar enviarlo, se inserta una entrada en FeedbackPendiente con estado "pending". Si la app se recarga, en la inicialización revisa IndexedDB: si encuentra un plan del día vigente, lo carga inmediatamente sin esperar red; si encuentra feedbacks pendientes, inicia el proceso de sincronización de fondo.

**Responsabilidades del módulo offline:** Garantizar que la app **siga funcionando sin internet**. Esto implica responder a las acciones críticas con o sin conexión: si el asesor solicita su ruta y no hay red, el SW debería ofrecer la información almacenada en la sincronización anterior (si existiera). Si registra un feedback sin red, el modulo offline se asegura de persistirlo localmente y notificar al UI (por ejemplo con un icono 🔄 en esa recomendación). Cuando vuelve la red, el SW/logic correspondiente envía automáticamente esos feedback pendientes al backend y notifica al UI para actualizar el estado (pasar de "pendiente" a "enviado"). Todo esto de forma transparente para el usuario, cuyo flujo de trabajo idealmente no se interrumpe. En resumen, este componente implementa las políticas de sincronización: qué hacer en caso de error de red (reintentos, colas) y cómo mantener la base de datos local consistente con el servidor.

**Backend – Controladores API REST:** Este módulo corresponde a la lógica de enrutamiento y manejo de requests en el servidor FastAPI. Cada funcionalidad principal tiene un controlador o **endpoint**:

* POST /login: recibe credenciales, verifica autenticidad (contra un servicio de autenticación simple, p.ej. una tabla de usuarios permitidos) y devuelve resultado (éxito o error). Podría generar un token de sesión (JWT) para que el frontend lo use en siguientes llamadas.
* GET /plan-de-ruta?ruta={id}&fecha={yyyy-mm-dd}: endpoint protegido (requiere login) que obtiene el plan de ruta enriquecido para el asesor indicado y fecha dada (normalmente la fecha actual). Este método concentra la integración: al invocarlo, el backend consulta los dos orígenes de datos necesarios. **Estrategia 1:** Idealmente, contacta al **Servicio IA** el cual devuelve ya el objeto PlanDeRutaEnriquecido completo. Si la API de IA provee todo (ya con los clientes del día incorporados), el backend simplemente lo reenvía al frontend después de, opcionalmente, guardarlo en Firestore (como registro de qué se recomendó). **Estrategia 2:** Alternativamente, si la API de IA requiriera lista de clientes por separado, el backend primero leería la lista de visitas desde la **DB de Ruta** y luego llamaría a la API de IA pasando esos clientes o la ruta, para obtener recomendaciones. En cualquier caso, este endpoint encapsula ese proceso. Tras obtener los datos, responde al frontend con un JSON que contiene: la identificación del plan (fecha, ruta, etc.), y la lista de clientes con sus recomendaciones. El plan se marca para el frontend como **descargado con éxito** y es lo que se almacena offline.
* POST /feedback: recibe un objeto de **RegistroFeedback** (identifica plan, cliente, recomendación y resultado seleccionado, más timestamp y posiblemente geolocalización). El backend al recibirlo realiza dos acciones en paralelo: **(1)** Guarda el registro en la base **Firestore** propia (persistencia interna), y **(2)** Llama a la API de IA para entregarle el feedback correspondiente (así el sistema IA puede actualizar sus métricas de piloto). Si la llamada a IA falla, el backend podría reintentar o almacenar una marca de que ese envío está pendiente (aunque normalmente se confía en que se recibirá eventualmente, dado que el frontend también podría volver a intentar). Este endpoint puede ser llamado muchas veces durante el día (una por cada recomendación atendida) o incluso recibir lotes de feedback (si el frontend decide enviar en batch cuando recupera conexión).
* POST /inteligencia: (posible) endpoint para eventos de inteligencia de mercado. Cuando el asesor llena los formularios de quiebres de stock, inventario, competencia, etc., estos podrían consolidarse en un objeto (o varios) y enviarse mediante esta ruta. El backend simplemente almacenaría esos datos en Firestore bajo la colección correspondiente (p.ej. InteligenciaStock, InteligenciaCompetencia) junto con la identificación del asesor, cliente y tiempo. En esta primera fase puede que la IA no procese estos datos en vivo, así que probablemente no se envían a ningún otro servicio en este punto (quedan para análisis offline).
* POST /finalizar-visita: (opcional) endpoint para marcar el fin de la visita. Alternativamente, podría reutilizar /feedback o /inteligencia con un tipo especial. Su cometido es registrar que el asesor terminó con cierto cliente. Se le envía el ID de cliente, plan, hora y geo-coordenadas. El backend lo guarda en Firestore (colección VisitasCompletadas por ejemplo). Este dato sirve para auditoría (saber cuántas visitas previstas fueron efectivamente realizadas y a qué hora). No se envía a la IA ya que no es feedback de recomendación sino metadato de visita.

**Responsabilidades de los controladores:** Actuar como la interfaz server-side para la WebApp. Validan las peticiones (por ejemplo, comprobando que el usuario esté autenticado, que los parámetros estén completos), orquestan llamadas a otros componentes (servicios externos o base de datos) y consolidan la respuesta al cliente. También manejan errores: si la consulta a la DB de ruta falla, devuelven un error de sincronización que el frontend mostrará. Si todo va bien, envían los datos. En esencia, implementan la lógica de negocio del lado servidor manteniendo la app coherente con las reglas del dominio (por ejemplo, evitar duplicados de feedback, asignar ID únicos a cada registro de feedback que entra, etc.). En este piloto, la lógica de negocio en backend es ligera, actuando más como *broker* de datos entre front, IA y storage.

**Backend – Adaptadores de integración (clientes externo):** Son subcomponentes internos que encapsulan las llamadas a servicios externos:

* **Cliente API IA:** Componente que sabe invocar el servicio de recomendaciones. Puede estar implementado simplemente mediante peticiones HTTP (usando requests en Python, por ejemplo). Contiene la URL base del servicio IA y métodos para get\_plan(ruta, fecha) y send\_feedback(payload). Si la API IA requiere autenticación (token), este componente la gestiona. Centralizarlo permite que el controlador /plan-de-ruta no necesite conocer detalles de URL o formato exacto; si mañana la API IA cambia (por ejemplo a otra ruta o protocolo), solo se actualiza este cliente.
* **Cliente DB ruta:** Módulo encargado de obtener la lista de visitas diarias. Puede ser una clase que ejecuta una consulta SQL sobre una conexión preestablecida. Por ejemplo, podría usar un driver PyODBC o SQLAlchemy para conectarse a la base de Ferguez. El método principal podría ser get\_visitas\_hoy(asesor\_id) que devuelve los clientes programados. Dado que en la propuesta se mencionaba necesidad de credenciales de DB, es probable que el backend tenga configurado un connection string seguro. Alternativamente, si Ferguez expuso una API REST para esto, habría un cliente HTTP similar al de IA. De cualquier modo, este adaptador aísla la forma en que se obtiene la “Hoja de Ruta”.
* **Adapter geolocalización (opcional):** En realidad la obtención de coordenadas GPS se hace en frontend mediante la API del navegador, pero si hubiera necesidad de traducir coordenadas a direcciones u otros servicios (geo-reverse), un adaptador podría llamar a un servicio de mapas. Esto no se contempla en el MVP, solo lo mencionamos como posible extensión.

**Responsabilidades de los adaptadores:** Proporcionar una interfaz limpia al resto del backend para operaciones de IO externas. Manejan detalles como formateo de URLs, transformación de datos (por ejemplo, convertir el JSON de la API IA a estructuras Python y viceversa), control de errores de red (reintentos simples si falla, quizá logs). Esto sigue el principio de **separación de preocupaciones**: el controlador de negocio se enfoca en qué hacer (obtener plan, guardar feedback) y delega en estos adaptadores el **cómo comunicarse** con cada sistema.

**Backend – Repositorio de datos (Firestore SDK):** Este componente representa la capa de acceso a la base interna del sistema. Utiliza el SDK de Google Cloud para Python para realizar operaciones en **Cloud Firestore**. Suele encapsular funciones tipo guardar\_feedback(feedback\_obj) que inserta un documento en la colección Firestore correspondiente, o guardar\_inteligencia(tipo, datos) para distintos subtipos de datos de mercado. También podría ofrecer métodos de lectura si hiciera falta (por ejemplo, para leer configuraciones o catálogos si estuvieran en Firestore). En este MVP, Firestore principalmente almacena los datos capturados, por lo que la mayoría de operaciones son escrituras. Centralizarlo en un repositorio facilita, por ejemplo, agregar lógica de *timestamp* unificado, o agregar *batch writes* si conviene.

**Responsabilidades del repositorio Firestore:** Conectar de forma eficiente y segura a la base de datos en la nube. Garantiza que los datos se guardan en las estructuras correctas. Por ejemplo, al guardar un feedback, asigna un ID único (pudiendo reutilizar el feedback\_id generado por el frontend o generar uno nuevo en el servidor) y guarda campos como el asesor, cliente, resultado, y marca de tiempo. Podría también implementar alguna lógica de *de-dupe* en caso de reintentos (por ejemplo, si llega dos veces el mismo feedback\_id, no duplicar entrada). Aunque en un piloto de 6 asesores esto quizá no sea crítico, se deja preparado. Otro punto es la seguridad: mediante las reglas de Firestore o el entorno GCP, se asegura que solo el backend puede escribir/leer esos datos (el frontend no accede directamente a Firestore, así evitamos exponer claves en la app).

**Base de datos – Cloud Firestore:** Si bien Firestore en sí es un servicio gestionado, lo mencionamos como componente persistente. Los datos principales almacenados:

* Colección **FeedbackRecomendacion:** cada documento representa la respuesta dada a una recomendación concreta durante una visita. Campos: ids (plan, cliente, recomendación), asesor, resultado (ej. *"exitosa"*), posiblemente campos adicionales como comentarios y geolocalización. Estos datos servirán para análisis de efectividad de recomendaciones.
* Colección **InteligenciaMercado:** aquí se pueden guardar distintos subtipos de observaciones de mercado (quiebres, inventarios, competencia, etc.) que el asesor reporta. Cada documento incluye cliente, asesor, tipo de reporte, detalle (ej. SKU X sin stock, motivo Y) y timestamp.
* Colección **Visitas:** para punteo final, cada documento indica ruta, cliente, hora de check-in/check-out, coordenadas, etc., permitiendo saber cobertura de visitas.
* Colección **Usuarios:** posiblemente una lista de usuarios autorizados con hash de contraseña para login, a menos que se haya decidido manejarlos en otro lado.

Firestore guarda datos en formato JSON/documento. Facilita consultas simples (p.ej., se podría consultar todos los feedback de cierto asesor o cierto plan). Por su naturaleza, escala automáticamente con poca configuración, y ofrece baja latencia para lecturas individuales (aunque en nuestro diseño la mayoría de lecturas las hace el frontend de su cache, no tanto del servidor). Un beneficio es que todo queda en la nube de Ferguez, disponible para exportar luego (p. ej. a BigQuery para análisis agregado).

En conjunto, estos bloques constituyen el **mapa del código fuente**. Por ejemplo, un desarrollador que se una al proyecto podrá identificar:

* Carpeta frontend: con componentes React para Login, Dashboard, ClienteDetail, etc., y un módulo service-worker.js + utilidades de almacenamiento local.
* Carpeta backend: con módulos como auth.py (login), plan.py (controlador plan), feedback.py, etc., además de utils/db\_route.py (cliente DB), utils/ia\_client.py, utils/firestore\_repo.py.
* Configuraciones: archivos de entorno para credenciales (DB route, API keys) y manifest PWA.

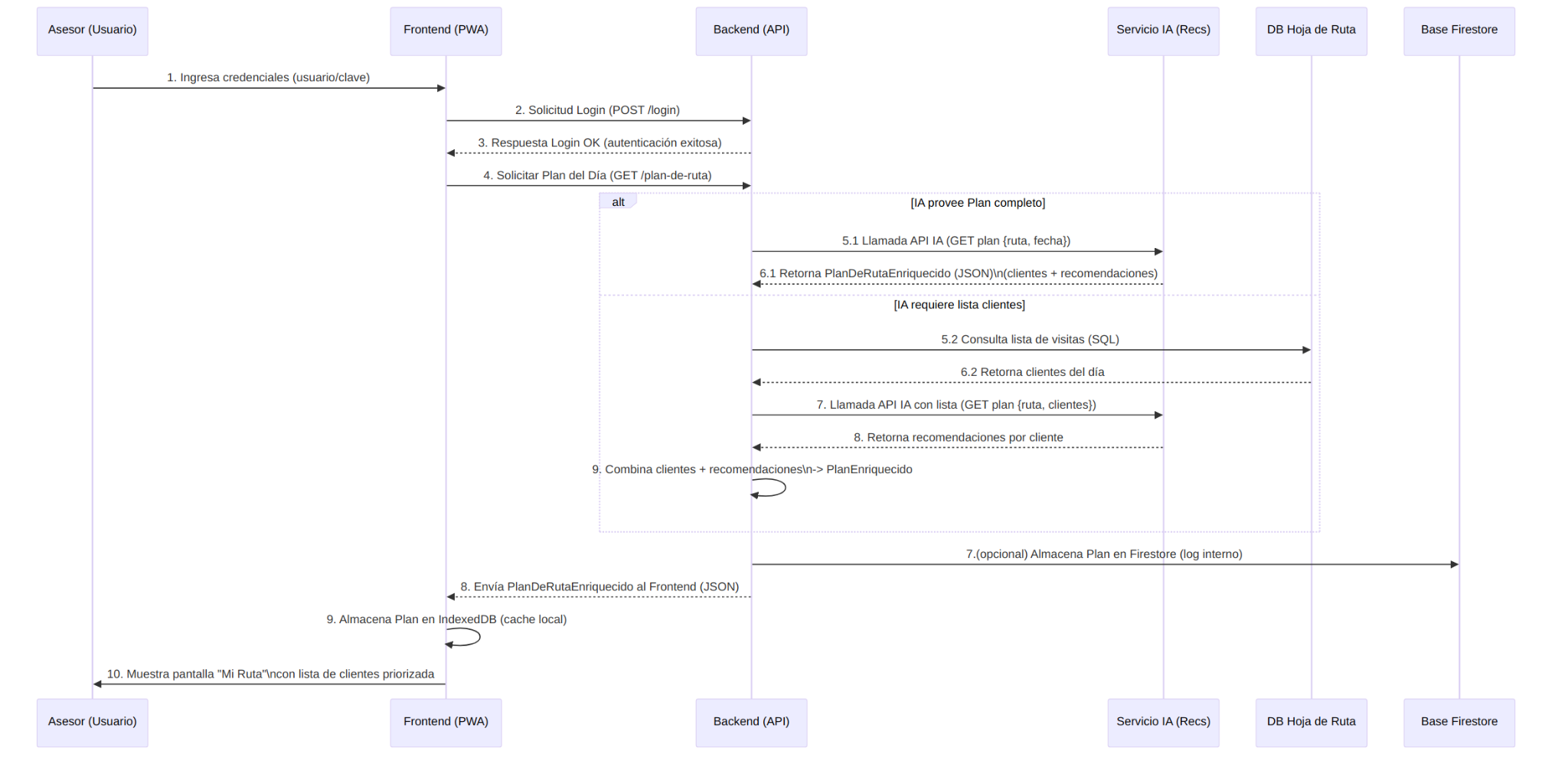
Cada bloque interactúa con otros a través de interfaces bien definidas. Esta claridad estructural ayuda a **ubicar funcionalidades** (¿dónde se valida un feedback? En backend controlador; ¿dónde se define cómo se pinta una recomendación? En un componente frontend), y facilita asignar tareas de implementación y futuros cambios de forma localizada.

## 6. Vista en tiempo de ejecución (runtime)

En esta sección se ilustran los escenarios dinámicos más importantes para entender el comportamiento de la aplicación en tiempo de ejecución. Nos enfocamos en dos **casos de uso clave**: la sincronización inicial de datos al comenzar el día, y el flujo de registro de una recomendación con y sin conectividad (incluyendo la sincronización diferida). Estos escenarios cubren el ciclo de vida completo de la información, desde su origen (sistema IA, DB ruta) hasta su destino final (almacenamiento y retroalimentación).

### 6.1 Escenario: Inicio de día – Login y sincronización inicial

**Descripción:** Cada mañana, el asesor abre la aplicación e inicia sesión para obtener su plan de visitas del día junto con las recomendaciones de IA. Este escenario asegura que antes de salir a ruta, toda la información necesaria está disponible localmente.

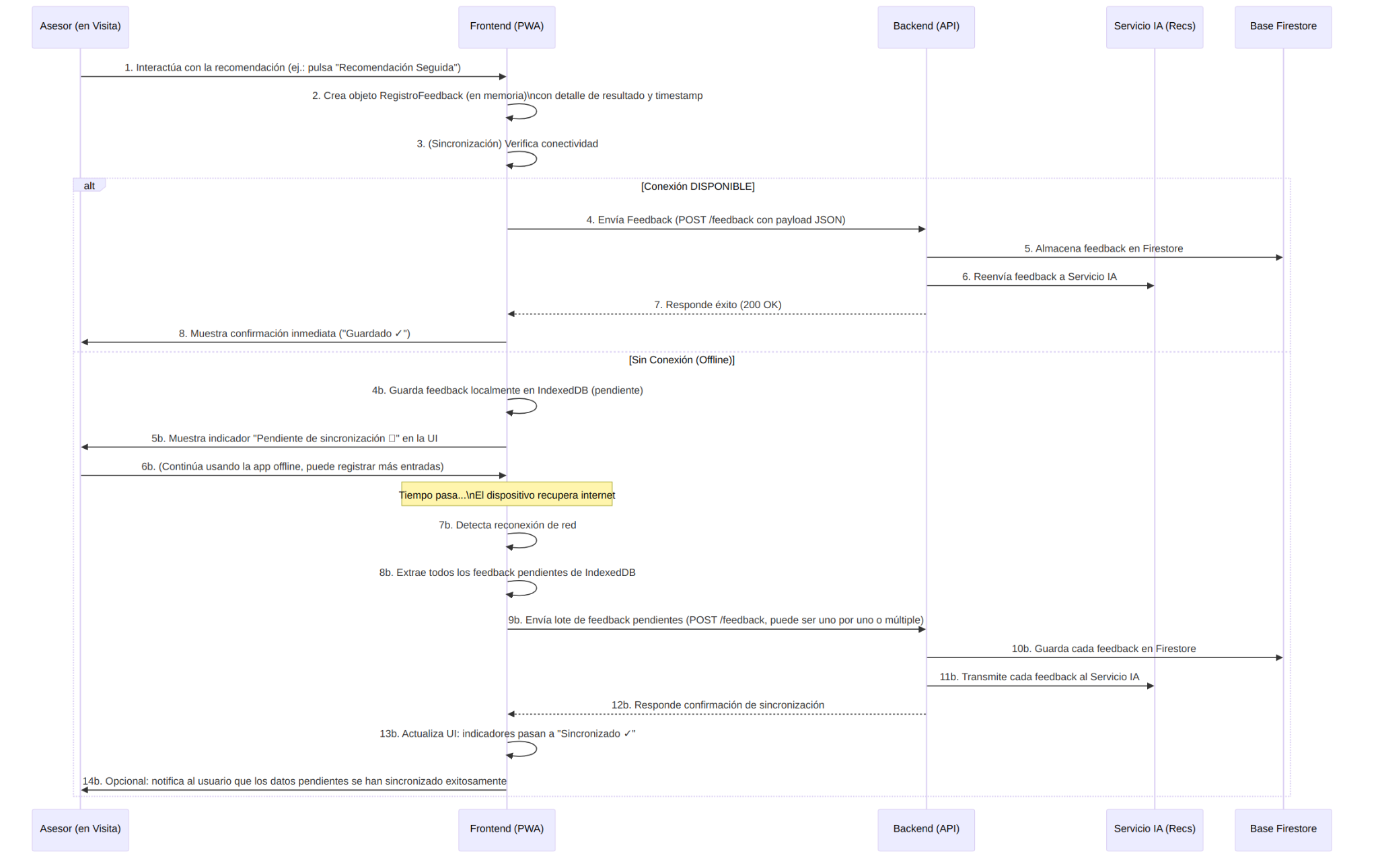


**Paso a paso:** (1-3) El asesor ingresa su identificación y clave; el frontend envía la solicitud de login al backend, el cual valida las credenciales (por simplicidad omitimos detalles, pero podría verificar en Firestore o en memoria). Al éxito, devuelve una respuesta OK (posiblemente con un token de sesión o cookie). (4) Inmediatamente después, el frontend inicia la sincronización principal llamando al endpoint de obtención del plan de ruta. (5-6) El backend entonces consulta el **Servicio IA** para obtener el plan enriquecido. **Caso típico:** El servicio IA devuelve directamente un objeto que incluye la lista ordenada de clientes y recomendaciones para cada uno, pues fue diseñado como un **único punto de acceso** para el plan diario. **Caso alternativo:** Si fuera necesario, el backend primero extraería la lista de clientes desde la DB corporativa de rutas (por ejemplo, si la IA no tiene visibilidad de la asignación diaria). Esto añadiría unos pasos (5.2-9) donde el backend combina datos de ambas fuentes. En cualquiera de los casos, para el usuario la operación es una: obtener su ruta del día con contexto de IA. (7) El backend podría guardar una copia del plan en Firestore a modo de registro (no imprescindible para funcionamiento pero útil para auditoría o reconstruir datos si algo pasa). (8) Luego responde al frontend con el plan completo. (9) El frontend recibe el JSON, lo procesa y guarda todo ese contenido en el **almacenamiento local IndexedDB**. Esto significa que, a partir de este momento, la app **ya no depende de la red para mostrar la información de la ruta**. (10) Finalmente, la UI muestra al asesor la **lista de clientes** priorizada en la pantalla principal, permitiéndole visualizar de un vistazo a quién visitar primero y por qué. Cada cliente aparece junto a la recomendación clave (Porqué de la visita). En este punto, la sincronización inicial concluye satisfactoriamente. Si en algún paso hubiera fallos (por ejemplo, credenciales incorrectas o falta de conexión para obtener el plan), la app notificaría el error y permitiría reintentar.

Tras la sincronización, el asesor puede dirigirse a su primera visita con toda su agenda y recomendaciones **ya almacenadas** en el dispositivo (esto fue crítico para habilitar el modo offline). Cabe destacar que el flujo anterior **debe ocurrir en pocos segundos** para no retrasar la salida a ruta – en diseño se estableció mostrar un overlay de progreso "Descargando ruta del día..." durante la operación.

### 6.2 Escenario: Registro de recomendación – Feedback online vs offline

**Descripción:** Durante la jornada, el asesor interactúa con la app en cada visita para registrar el resultado de las recomendaciones ofrecidas. Presentamos el flujo dinámico de **registro de feedback** en dos situaciones: con conexión de datos disponible (envío inmediato) y sin conexión (almacenamiento local y sincronización posterior). Este escenario demuestra cómo la arquitectura maneja la lógica offline-first en tiempo real.



**Explicación:** El flujo comienza con el asesor usando la app durante una visita. En la pantalla de cliente, por cada recomendación el asesor puede **registrar su interacción** mediante los botones de feedback. Por ejemplo, supongamos que ofrece un producto recomendado y el cliente lo acepta. El asesor marcaría "✅ Se Aplicó" y "🗣️ Cliente Acepta" (podrían ser dos botones o un flujo de dos pasos según UI). (1) Esa acción del usuario activa la captura de feedback. (2) El frontend crea un objeto RegistroFeedback en ese instante, incluyendo identificadores de la recomendación, cliente, etc., el resultado seleccionado y la hora actual. También puede anexar la ubicación GPS si disponible (la app podría tomar coords en ese momento, aunque también podría hacerlo solo al final; en la estructura se prevé un campo ubicacion\_gps). (3) Antes de intentar enviarlo, el frontend chequea el estado de conexión (el Service Worker o el navegador indican si hay red).

Si hay conexión (**rama online**): (4) el frontend envía inmediatamente el feedback al backend mediante una llamada POST /feedback. (5) El backend al recibirlo almacena el registro en Firestore (persistencia garantizada en nube). (6) Luego invoca al servicio IA con el feedback (por ejemplo, para un resultado "exitosa" de la recomendación X, podría llamar POST /ia/feedback con esos datos). (7) Asumiendo que el backend obtiene respuesta OK de IA, responde al frontend con éxito. (8) El frontend entonces refleja esto al usuario: por ejemplo, en la tarjeta de la recomendación podría mostrar un icono de check verde confirmando que se guardó. También podría deshabilitar los botones de esa tarjeta para prevenir cambios (ya quedó registrado) y marcar la visita como **completada** si era la última acción pendiente. Todo esto ocurre en cuestión de segundos, dando una experiencia fluida.

Si **no hay conexión** en ese momento (**rama offline**): (4b) en lugar de llamar al backend, el frontend guarda el objeto de feedback en la base local IndexedDB. Se agrega con un estado "pendiente". (5b) La interfaz inmediatamente señala al asesor que su input fue registrado localmente pero aún **no sincronizado** – típicamente mostrando un icono de sincronización pendiente (por ejemplo, un símbolo circular giratorio junto a la recomendación). Esto sigue el principio de **feedback inmediato al usuario**; el asesor ve que su acción fue tomada en cuenta, aunque no haya red. La app en este estado offline permitirá seguir navegando y registrando más datos. (6b) El asesor continúa con su trabajo normalmente, quizás va al siguiente cliente. Puede repetirse el proceso y acumular varios feedback pendientes. Cada uno quedará almacenado localmente.

Eventualmente, cuando el dispositivo recupere conexión de internet (por cobertura celular o al volver a una zona con WiFi), el módulo offline detecta ese evento. El Service Worker o la aplicación misma hacen un *polling* del estado de conexión. En cuanto se identifica **conexión restablecida** (7b), se inicia la sincronización en segundo plano de los datos pendientes. (8b) El frontend reúne todos los registros de feedback que están en cola (IndexedDB) desde la última vez. (9b) Entonces envía esos registros al backend. Esto puede implementarse enviando uno por uno (llamando POST /feedback repetidas veces) o optimizado enviando un arreglo de múltiples feedbacks en una sola llamada (el backend puede soportar batch). Aquí elegimos presentarlo como un lote por simplicidad. (10b, 11b) El backend procesa cada elemento del lote igual que antes: los escribe en Firestore y los reenvía a la IA. Si alguno fallara, el backend podría devolver información de error; en general se espera que la conexión sea estable ahora y todos se envíen bien. (12b) El backend responde al frontend confirmando que la sincronización de pendientes se completó. (13b) El frontend entonces actualiza la interfaz: los iconos 🔄 pendientes pasan a checks verdes, indicando que ya todo está guardado en servidor. (14b) Podría incluso mostrar un pequeño mensaje tipo "Datos sincronizados" para tranquilidad del usuario. Después de esto, la IndexedDB local se limpia de esos items (o se marca como sincronizados).

Este mecanismo de *queue-retry* asegura que incluso si el asesor estuvo horas sin señal, al reconectar **no se pierde nada**: toda interacción registrada en campo se terminará enviando. Para el asesor, la única diferencia entre trabajar online u offline es el pequeño icono de estado; el flujo de uso no cambia. Desde la perspectiva de la arquitectura, hemos implementado una **tolerancia a la desconexión** completa, mediante el almacenamiento local y la sincronización diferida. Esto fue verificado en pruebas UAT de campo simulando pérdida de conexión.

*Nota:* Si hubiera múltiples dispositivos/rutas, cada uno sincroniza independientemente su info. Firestore manejará escrituras concurrentes. En nuestro caso, pocos usuarios evitan conflictos. Si un mismo asesor reintenta enviar duplicado (posible en reconexión manual), el backend podría filtrar duplicados por feedback\_id único.

### 6.3 Escenario complementario: Captura de inteligencia de mercado y finalización

**Descripción:** Adicionalmente se describen brevemente dos sub-flujos: la **captura de inteligencia de mercado** y la **finalización de una visita**, que complementan el proceso principal.

**Captura de inteligencia:** El asesor, en cualquier punto durante o después de la interacción principal, puede ingresar datos como quiebres de stock, inventario de competidores, etc. Supongamos que tras atender las recomendaciones, nota que falta un producto propio en anaquel. El asesor navega al módulo de "Inteligencia" en la app, selecciona "Reportar Quiebre de Stock", escoge el SKU afectado y motivo. Al enviar ese formulario, el frontend crea un objeto (p.ej. EventoInteligencia con tipo "QuiebreStock", SKU, cliente, hora) y sigue un flujo similar al feedback: si hay conexión, lo envía inmediatamente a un endpoint (ej. /inteligencia), si no, lo guarda pendiente. El backend al recibirlo simplemente lo almacena en Firestore (colección QuiebresStock) con la información. No se envía a la IA en este momento, ya que no hay un sistema consumidor en tiempo real; estos datos se usarán posteriormente para enriquecer análisis y modelos. La app confirma al usuario con un "✓ Reportado". Este proceso ocurre de forma análoga para otros tipos (inventario competidor, etc.). Todos estos eventos utilizan la misma infraestructura offline/online de la app, manteniendo consistencia.

**Finalización de visita (punteo):** Después de registrar todas las recomendaciones para un cliente, el asesor toca "Finalizar Visita". En ese momento, el frontend puede obtener la geolocalización actual (usando navigator.geolocation.getCurrentPosition) - esto puede tardar un par de segundos si no se tenía antes. Una vez obtenido (o si expira, se puede tomar como opcional), el frontend arma un objeto VisitaCompletada con cliente, hora fin, coordenadas. Luego llama al backend (ej. /finalizar-visita). El backend guarda ese objeto en Firestore (colección Visitas). Este dato marca que la visita X fue completada a tal hora y lat/long, lo cual puede servir para auditoría o calcular cumplimiento de visitas. Si no hubiera conexión al puntear, se puede guardar localmente y sincronizar igual que feedback. Tras finalizar, la app puede retornar a la lista de ruta y marcar ese cliente como "Completado" (posiblemente moviéndolo al final o grisándolo). Este punteo no es obligatorio para pasar al siguiente cliente pero proporciona cierre formal.

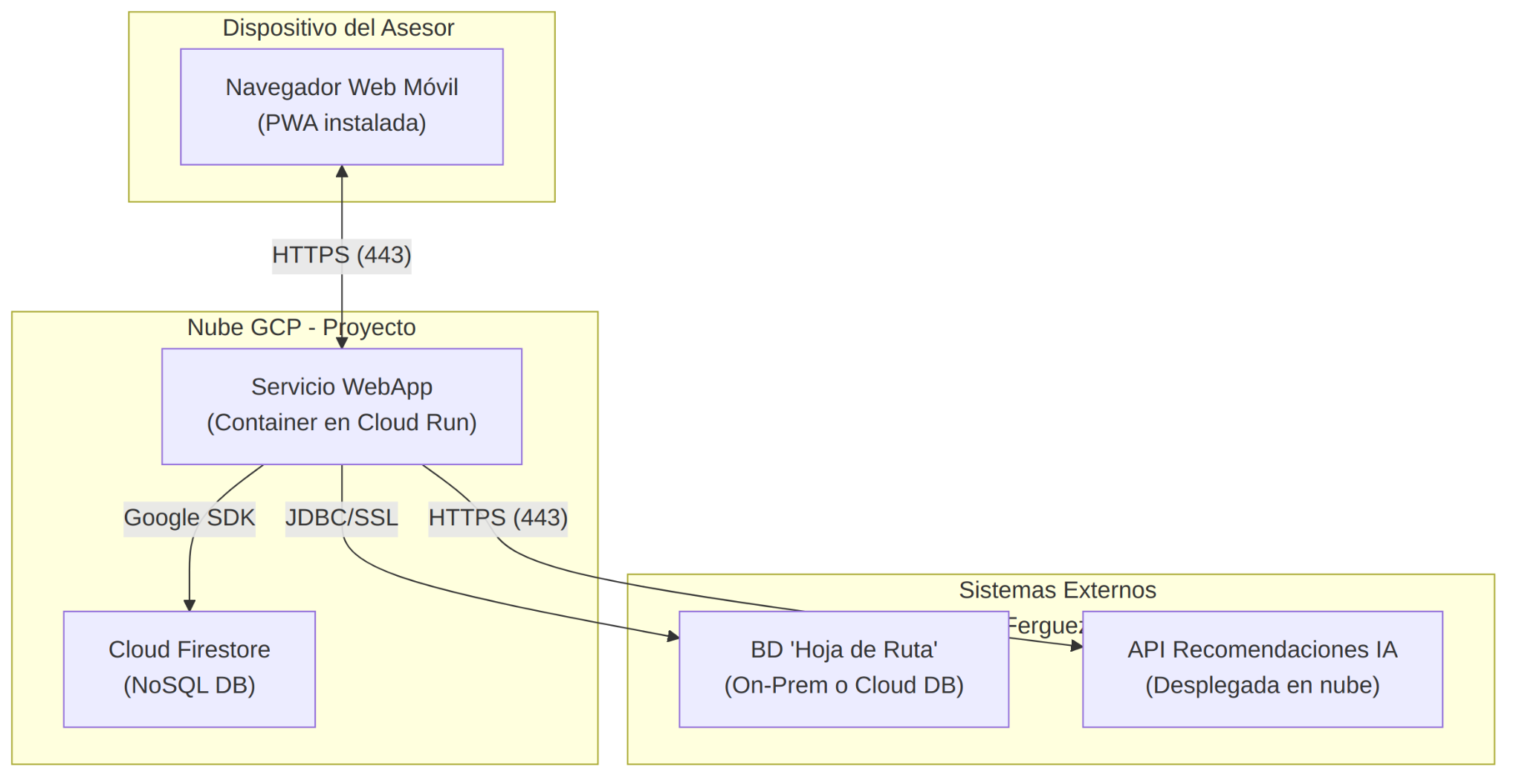
Con estos flujos combinados, se cubre el **día completo de un asesor**: inicia sesión y sincroniza datos, visita a cada cliente siguiendo el orden sugerido (aunque la app permite visitar en otro orden si decidiera), registra resultados de recomendaciones y hallazgos de mercado, y finaliza cada visita. Durante todo ese ciclo, la arquitectura runtime asegura que:

1. El asesor **siempre tiene acceso** a sus datos (gracias a la descarga inicial y cache).
2. **Cada interacción relevante se registra** (no depende de recordar más tarde, la app facilita registro inmediato con un par de taps).
3. **No se pierden datos** por problemas de conexión; el sistema se recupera y sincroniza después, preservando la integridad de la información del piloto.
4. El asesor está **informado** del estado de sus acciones (vía mensajes e iconos de pendiente/sincronizado, evitando incertidumbre sobre si se guardó o no).

Este entendimiento dinámico del sistema nos permite confiar en que, operando en condiciones reales de campo, la solución cumplirá su cometido: **medir la efectividad de las recomendaciones de IA** en un entorno real, robusto ante las limitantes operativas (conectividad) y amigable para el usuario final.

## 7. Vista de despliegue (deployment)

La arquitectura física y de despliegue de la solución aprovecha la infraestructura cloud existente de Ferguez para asegurar una puesta en producción rápida y confiable. A continuación se describe cómo los componentes lógicos se distribuyen en nodos/servicios reales, los entornos involucrados y consideraciones de infraestructura.



*Diagrama 7.1: Despliegue físico de la solución. El frontend se ejecuta en el navegador web del dispositivo móvil del asesor (teléfono o tableta). El backend corre en la nube GCP de Ferguez dentro de un contenedor serverless (Cloud Run). La base de datos Firestore es un servicio gestionado en GCP. Los sistemas externos incluyen la base de datos corporativa de rutas (posiblemente on-premise en centro de datos de Ferguez) y el servicio de IA (que podría estar en la nube de Interware o también en GCP). Las comunicaciones principales ocurren a través de Internet usando HTTPS seguro.*

**Nodos de despliegue y detalles:**

* **Dispositivos cliente (asesores):** Cada asesor utilizará un dispositivo móvil (smartphone Android o iOS, presumiblemente) con un navegador compatible. Se recomienda usar Chrome (en Android) o Safari (en iPhone) ya que soportan PWA. La aplicación puede ser añadida a pantalla de inicio, actuando casi como app nativa. Importante: para que la PWA funcione con service workers, debe servirse sobre HTTPS y tener un dominio confiable. Es probable que la app esté alojada en un subdominio de Ferguez (ej. pilot.inteligencia.ferguez.com). Cada dispositivo descarga la app al acceder por primera vez (semana de onboarding) y luego las actualizaciones se gestionan vía service worker. No se requiere instalar nada desde tiendas, lo que agilizaba la distribución en el piloto.
* **Frontend hosting:** Aunque el diagrama lo muestra conceptualmente en el dispositivo, en la práctica los archivos estáticos de la aplicación (HTML, JS, CSS, imágenes) necesitan ser servidos desde un hosting. En nuestro despliegue, optamos por empaquetar el frontend junto con el backend en el mismo servicio de Cloud Run. Es decir, el contenedor FastAPI también sirve los archivos estáticos generados por Vite (posiblemente configurando rutas para "/" que devuelven index.html y sirviendo la carpeta dist). Esto simplifica la infraestructura: un solo servicio URL entrega tanto la API como la página web. Alternativamente, se pudo usar **Firebase Hosting** o un bucket de Cloud Storage para los estáticos, pero no fue imprescindible. Al usar Cloud Run, el servicio escala automáticamente de 0 a N instancias según la carga. Dado el bajo número de usuarios (piloto), usualmente habrá solo 1 instancia activa y en periodos ociosos puede escalar a 0 (apagarse). El cold start de Cloud Run (~ segundos) es aceptable en este contexto (por ejemplo, a primera hora al primer login puede tardar un poquito más). Cloud Run maneja SSL (HTTPS) out-of-the-box, asegurando la conexión segura requerida.
* **Backend (Cloud Run Container):** El backend FastAPI se ha dockerizado y desplegado en Cloud Run, en la región cercana a los usuarios (p.ej. us-central1 si los asesores están en México, u otra región GCP conveniente) para baja latencia. Cloud Run proporciona un endpoint HTTPS para el servicio. Configuraciones relevantes: se asignó una CPU y memoria acorde (por ejemplo 1 vCPU, 512 MB) suficiente para manejar peticiones ligeras. Cloud Run puede instanciar múltiples contenedores en paralelo si repentinamente muchos usuarios acceden a la vez; en nuestro caso, con ~5-10 asesores, no se espera más de 1-2 peticiones concurrentes, por lo que la escala auto probablemente permanezca en 1 instancia. Las variables de entorno sensibles (credenciales de DB route, etc.) se guardan en la configuración del servicio (Secret Manager o directamente en variables de Cloud Run, que están cifradas en reposo). El backend se integra con Firestore usando las credenciales del servicio (al estar en el mismo proyecto GCP, las cuentas de servicio permiten acceso sin keys manuales). Para conectar con la base de datos de ruta on-prem, se pudo requerir un componente adicional: **Serverless VPC Connector** de GCP, para entrar vía VPN a la red corporativa. Si Ferguez expuso la base de datos por IP pública (whitelisting la IP de Cloud Run), se pudo evitar la VPN. Este detalle se coordinó con TI de Ferguez en la etapa de habilitación.
* **Cloud Firestore (database):** Firestore es un servicio multi-tenant en GCP, pero aquí se despliega lógicamente dentro del proyecto de Ferguez. Está disponible en la misma región que Cloud Run (se suele elegir/us-central). No requiere administración de servidores ni VMs. Las colecciones se definen conforme la app las usa. La escalabilidad automática de Firestore cubre hasta 10k escrituras por segundo, muy por encima de las necesidades del piloto (quizás decenas por día). Un aspecto a configurar son las **reglas de seguridad** de Firestore: dado que el acceso es solo vía backend (nuestro diseño no expone Firestore directo al frontend), las reglas pueden ser sencillas permitiendo operaciones desde la cuenta de servicio del backend. No se espera acceso directo desde usuarios finales, así que se mantienen cerradas públicamente. Firestore garantiza alta disponibilidad (multi zona) por lo que los datos estarán seguros. También proporciona registros (Cloud Logging) para auditoría de accesos si requerido.
* **Sistema piloto IA (API externa):** La API de IA, desarrollada por el equipo de ciencia de datos, debe estar desplegada y accesible mediante URL. Puede estar en GCP también (quizá en Cloud Run o Cloud Functions dentro del mismo u otro proyecto), o en otra nube, o en servidores de Interware. Lo importante es que el backend de nuestra WebApp tenga conectividad a esa API. Si está en internet pública, se usará HTTPS con autenticación básica o token si proveen. Si está en red privada, se tendría que exponer o conectar redes (por simplicidad asumimos API IA accesible públicamente con seguridad). Dado que es un servicio crítico para la funcionalidad matutina, se monitorea su disponibilidad. En el cronograma se hizo **definición del contrato de API** pronto para asegurar compatibilidad. La API IA idealmente está dimensionada para manejar una ráfaga de llamadas en la mañana (una por asesor). Con pocos usuarios, no es problema, pero a futuro debería escalar (posible implementar caching de respuestas por día para no recalcular). El piloto IA también recibe nuestros POST de feedback – debe haber un endpoint receptor activo continuamente para eso. Nuestra WebApp no espera respuesta más allá de confirmación, pero el servicio IA deberá almacenar o procesar ese feedback.
* **Base de datos de hoja de ruta:** Este componente corresponde a un sistema existente de Ferguez. Puede ser una base de datos SQL (ej. SQL Server, Oracle) en el datacenter corporativo. Para el piloto, TI de Ferguez proporcionó una vista con los clientes de las rutas piloto, accesible con credenciales de solo lectura. La conexión desde Cloud Run a esta DB pudo resolverse de dos maneras:
  1. **Exposición directa limitada:** Abrir el acceso a la IP del servicio en la nube (aunque Cloud Run IPs varían, se pudo usar un proxy static IP). O exponer la DB via un API. Esto requiere configuraciones de firewall/seguridad en Ferguez.
  2. VPN a on-prem: Configurar una VPN site-to-site o usar el Serverless VPC Access (si la DB está en una VPC accessible via Cloud VPN).  
     En cualquier caso, la latencia de consultar la DB para ~10 filas no es alta. Incluso se pudo optar por extraer la data a un CSV o pequeña tabla en Firestore para simplificar (por ejemplo, precargar la ruta del día anterior), pero lo más fresco es leer directo. En el despliegue se verificó en semana 1 el acceso. Una vez listo, el backend lee de allí cada mañana.
* **Entornos de desarrollo/pruebas:** Durante el proyecto, seguramente se usó un ambiente de desarrollo local (los devs corrieron el frontend en localhost con mock APIs, etc.) y un ambiente de pruebas interno. Sin embargo, formalmente para despliegue se consideró al menos un entorno de **UAT** (User Acceptance Testing) donde realizar **pruebas de campo (semana 5)** antes del despliegue final. Este entorno UAT probablemente era el mismo que producción pero con usuarios limitados y datos controlados. Dado el corto plazo, es posible que se haya hecho un despliegue único e invitar a algunos usuarios piloto antes del go-live. En producción, todos los asesores piloto tendrían acceso. La promoción de versiones (CI/CD) se pudo automatizar: cada cambio en backend generaba una nueva imagen desplegada a Cloud Run (con revisión gradual tal vez), y los cambios en frontend se desplegaban simultáneamente (en PWA, el service worker se encarga de actualizar la app en los dispositivos, posiblemente con un mensaje "Nueva versión disponible, reiniciar"). Esto garantiza que todos los usuarios siempre usen la última versión.
* **Monitoreo y logs:** En Cloud Run y Firestore se cuenta con **Cloud Logging** integrado. Cada request al backend, así como errores, se registran. Esto permite al equipo técnico monitorear el sistema. Por ejemplo, podrían establecer una alarma si /plan-de-ruta devuelve muchos errores (indicando quizá fallo en conexión a DB o IA). Asimismo, podrían monitorear cuántos feedbacks se están recibiendo (p.ej., contar escritos en Firestore) para asegurar que los asesores usan la app como se espera, o detectar inactividad anómala. Para el piloto, un monitoreo básico basta, pero es bueno tener visibilidad ya que cualquier problema de sincronización debe detectarse pronto dado el corto tiempo.

En conclusión, el despliegue se apoyó en la **nube de Ferguez (GCP)** para lograr un set-up rápido y confiable: sin aprovisionar servidores manualmente, con escalado automático y servicios totalmente administrados (Firestore). Los asesores solo necesitan sus teléfonos con internet (aunque sea intermitente). Se verificó en pruebas de campo que incluso en áreas de baja señal, la app podía usarse offline y luego sincronizar al volver a conectividad. Esta arquitectura de despliegue es también fácilmente extensible: si se decidiera ampliar el piloto a más usuarios o duración, simplemente se continuarían utilizando los mismos recursos en nube (que pueden escalar). Para un despliegue a escala nacional futuro, esta base es sólida, requiriendo quizás reforzar algunos componentes (más seguridad, más capacidad en la API IA), pero manteniendo el mismo modelo de webapp offline-first en la nube.

## 8. Conceptos transversales (cross-cutting)

En esta sección abordamos aspectos de diseño que afectan a múltiples componentes de la arquitectura, asegurando que la solución se mantiene **consistente** y **coherente** en temas críticos como seguridad, manejo de configuración, gestión de errores, UX general, etc. Son principios y soluciones aplicados de forma uniforme en todo el sistema.

### 8.1 Seguridad (autenticación, autorización y datos):

La aplicación implementa un **control de acceso básico** para que solo personal autorizado pueda utilizarla. Cada asesor tiene credenciales individuales (un ID de empleado y contraseña simple). La autenticación se realiza en el backend en el momento de login. No se utiliza un sistema OAuth externo debido al alcance limitado; en su lugar, se mantiene una lista de usuarios válidos (por ejemplo, en Firestore Usuarios con hashes de contraseña, o incluso cargada desde un archivo seguro) que el backend consulta. Tras autenticarse, el frontend recibe confirmación y a partir de ahí se considera al usuario "loggeado". Para mantener la sesión, dado que la app es una SPA, se puede usar un token JWT que el backend emita y el frontend almacene (por ejemplo en IndexedDB o memoria) – así, cada llamada subsecuente incluye ese token en la cabecera Authorization. Otra alternativa más sencilla: una vez logueado, mientras la app siga abierta, se guarda el estado en la propia app (contexto React). En caso de recarga offline de la página, para no bloquear al usuario fuera por falta de conexión, **se debe cachear también el estado de login**. Esto se podría lograr guardando un *flag* en IndexedDB ("usuario autenticado: sí, con id X") de modo que si la app se reinicia sin internet, permita acceso a los datos cacheados. Es un punto delicado: idealmente requerimos conexión para autenticar al inicio del día. Pero si por alguna razón la app se cierra y el asesor está offline, podríamos dejarle ver los datos offline que ya tenía (porque están cifrados en su dispositivo de todos modos), quizás mostrándole que está en modo offline no autenticado. Para el MVP, asumimos que los asesores iniciarán sesión cuando tengan conexión (por la mañana) y mantendrán la app abierta durante el día, reduciendo la probabilidad de tener que reingresar credenciales sin red.

En cuanto a **autorización**, no hay roles múltiples con diferentes permisos en esta aplicación – todos los usuarios tienen básicamente el mismo rol (asesor piloto) y pueden acceder solo a su propia información. El backend, sin embargo, debe validar en cada petición de /plan-de-ruta o /feedback que el token/usuario que la realiza corresponda con los datos solicitados. Por ejemplo, un asesor A no debería poder pedir el plan de la ruta de otro asesor B. Esto se asegura vinculando la sesión con un asesor\_id y filtrando los datos en servidor según eso. Asimismo, el **feedback** recibido se asocia al usuario en sesión (ignorando cualquier asesor\_id malicioso en el payload, el backend sobrescribe o verifica que coincide).

La **seguridad de datos** se garantiza mediante:

* **Transmisión cifrada:** Todo tráfico entre frontend y backend, y backend con externos, se hace bajo HTTPS/TLS. Esto previene escuchas (importante aunque los datos no sean altamente sensibles, incluyen información de negocio).
* **Almacenamiento seguro:** En la nube, Firestore cifra datos en reposo automáticamente. En el dispositivo, los datos en IndexedDB no están cifrados por la aplicación, pero dependen de la seguridad del dispositivo (se asume los asesores mantienen sus teléfonos bloqueados y no root). Dado que no hay PII crítico, este riesgo es aceptable para el piloto. Si se quisiera mayor seguridad, se podría cifrar los datos locales con una clave derivada de la contraseña del usuario, aunque eso complicaría el acceso offline. Nuevamente, para el MVP se priorizó usabilidad.
* **Políticas de acceso cloud:** Solo el backend puede acceder a Firestore (controlado por reglas de seguridad). Los usuarios finales no interactúan directamente con la base ni con GCP. Además, se auditará qué usuarios tienen acceso al proyecto GCP para evitar fuga de datos internos.
* **Copia de datos:** Como la app es piloto, no se prevé grandes integraciones de seguridad corporativa (p.ej. LDAP). Sin embargo, se almacenan datos de ventas competitivas, lo cual la empresa puede considerar confidencial. Por ello, se limitará la distribución de la app solo a dispositivos/empleados autorizados.

En versiones futuras, se consideraría implementar **autenticación multifactor** o integración con el SSO corporativo de Ferguez para producción, pero durante el piloto, la solución actual es suficiente y más ágil.

### 8.2 Manejo de configuración y parametrización:

La aplicación debe comportarse adecuadamente en distintos entornos (dev, pruebas, producción) y permitir ciertos cambios sin re-desplegar código. Para ello:

* En el **backend**, se externalizaron configuraciones como las URLs de la API de IA, credenciales de la base de datos de ruta, y claves API (si hubiera) mediante variables de entorno en Cloud Run. Por ejemplo, IA\_API\_URL=https://ia-pilot.interware.com/ o DB\_ROUTE\_CONNECTION=.... Durante despliegue, se cargan los valores específicos de cada ambiente. Esto evita que datos sensibles estén hardcodeados. FastAPI lee esas variables en su startup para inicializar los clientes de integración.
* En el **frontend**, Vite permite definir variables de entorno en build (por ejemplo, la base URL del backend API). Se habría establecido algo como VITE\_API\_BASE\_URL=https://webapp-pilot-ferguez.run.app/api. De esta forma, la misma app frontend puede adaptarse a distintos servidores sin cambiar código. Además, elementos como la marca Ferguez (colores, logo) están en la configuración de UI (aunque eso no cambia por entorno, es constante).
* Otra parametrización útil es la **fecha o fase del piloto**: por ejemplo, si quisiéramos habilitar/deshabilitar ciertas funciones (tal vez el módulo de inteligencia de mercado se activa en la semana 3), se puede controlar mediante un *flag* en la config (o incluso en Firestore un documento de "config" que la app lee al iniciar). Sin embargo, para este MVP, se definió de antemano qué funciones incluir, y no fue necesario un feature toggle complejo.

El **Service Worker** y manifest también pueden tener configuraciones (como tiempo de cacheo). Se usaron valores por defecto razonables (cache inmediato del shell y datos hasta que nueva versión se despliegue).

En general, la idea es que si mañana la URL del servicio IA cambia, o si se mueven a otra base de datos de ruta, se pueda ajustar sin recompilar la app – solo actualizando configs y reiniciando el servicio.

### 8.3 Logging, monitorización y trazabilidad:

Como práctica transversal, se incorporó registro de eventos clave en todo el sistema:

* En el **backend**, se habilitaron logs para cada request entrante (método, endpoint, usuario, timestamp) y para errores o excepciones. Por ejemplo, si falla la llamada a la DB de ruta, el error se loguea con detalles para diagnóstico. Esto es soportado por FastAPI/Uvicorn (ya loggean requests). Se agregó logging específico en puntos críticos: al recibir feedback (log "Feedback recibido de asesor X para rec Y"), al enviar a IA (log de éxito o error del API IA). Estos logs se envían a Cloud Logging donde el equipo puede revisarlos. Esto asegura trazabilidad: cada interacción del usuario en campo puede ser correlacionada con logs de backend, útil para depuración o validación de flujo.
* En el **frontend**, al ser una app cliente, no se cuenta con un servidor para centralizar logs. Sin embargo, se decidió instrumentar la app para al menos capturar errores de JS inesperados y, si hay conexión, enviarlos al backend (por ejemplo vía un endpoint /error-log) o a un servicio de terceros (no implementado en MVP). También se pudo usar la consola del navegador en pruebas para detectar problemas. Dado el número reducido de usuarios, se consideró aceptable monitorear manualmente su feedback. Aún así, se aprovechó el propio Firestore: se podría guardar un *heartbeat* o un evento cuando la app se carga, de modo de saber cuándo fue la última vez que cada asesor sincronizó. Este tipo de registro ayuda a la monitorización funcional (¿los usuarios están usando la app a diario?).
* **Monitorización en campo:** Además de logs técnicos, el piloto en sí requiere monitorear métricas de uso: cuántas recomendaciones fueron aceptadas, cuántos feedbacks registrados, etc. Si bien la explotación de esos datos es fuera de la app (seguramente el equipo de proyecto descargará los datos de Firestore/BigQuery), la arquitectura asegura que podamos obtener esa info. Por ejemplo, Firestore integrará con BigQuery a posteriori o se exportará al terminar las 6 semanas.

**Alertamiento:** Aunque no formalmente un requerimiento, se podría configurar alertas sencillas: ej. si ninguna petición /feedback llega en cierto día -> potencial problema (quizá los asesores no usan la app); o si el servicio IA no responde en la mañana -> alerta inmediata. Dado lo crítico de no perder el día de piloto, vale la pena tener monitoreo activo al menos manual en ese horario.

### 8.4 Diseño UX consistente:

Todos los componentes de UI siguen los **principios de diseño establecidos** en la fase de UX, de forma transversal:

* Se aplica una **paleta de colores unificada** con el verde Ferguez como primario en botones y highlights, grises para fondos y textos, y colores de estado estándar (verde éxito, ámbar para sincronización pendiente, rojo error). Esto garantiza que cualquier nueva pantalla o componente mantenga la identidad visual. Por ejemplo, el botón principal "Iniciar Jornada" es verde Ferguez con texto blanco, consistente en toda la app.
* Se usa una **tipografía legible (Roboto)** con tamaños fijos por jerarquía: títulos 24px, texto normal 16px, etc., y alto contraste (texto oscuro sobre fondo claro) para visibilidad en exteriores. Ningún texto importante va en color claro sobre blanco, evitando problemas bajo sol fuerte.
* **Tamaño de tocables:** Todos los botones e íconos interactivos respetan el mínimo de ~48x48 dp de área para facilidad de toque. Esto se consideró en todos los componentes (ej. las tarjetas completas son tocables para abrir un cliente, los botones de feedback son grandes y separados). Se aprovechó un grid de 8px para espaciar elementos y así lograr consistencia y alineación en todos los lugares.
* **Diseño responsive:** La interfaz está optimizada principalmente para pantallas de smartphones (aprox. 5-6"), pero es fluida y podría adaptarse a tablets. Se hicieron pruebas en diferentes resoluciones. Los componentes usan porcentajes o unidades relativas para acomodarse. Por ejemplo, la lista de clientes es vertical y se ajusta al alto de pantalla restante debajo del header. No hay horizontales que requieran scroll lateral. También se evaluó el comportamiento en orientación horizontal vs vertical (lo usual es vertical; horizontal se puede usar pero no es primordial).
* **Feedback al usuario y estados de carga:** De forma transversal, cualquier acción del asesor produce una respuesta visual inmediata para evitar confusión. Botones muestran estados activos al tocarlos, las operaciones largas muestran *spinners* o mensajes ("Sincronizando..."), los errores despliegan mensajes claros (por ej. "No se pudo descargar la ruta. Revisa tu conexión." con opción reintentar). Siempre se comunica al usuario qué está pasando (*Visibilidad del Estado del Sistema*).
* **Prevención de errores:** Este principio se implementó en varias capas: la UI desactiva botones irrelevantes (p. ej., tras elegir una opción de feedback, deshabilita las otras para que no marque dos veces contradictorias). Los formularios de inteligencia tienen validaciones (no enviar si faltan campos obligatorios). En la arquitectura offline, se tomó especial cuidado de evitar duplicar envíos: se usan IDs únicos para feedback, así si por error se manda doble, el backend puede reconocerlo.
* **"Thumb-Friendly":** Todos los elementos de interacción frecuentes están ubicados en la parte inferior o media de la pantalla, accesibles con el pulgar sin estirarse. Por ejemplo, en la vista de cliente, los botones de feedback están uno al lado del otro en la mitad inferior de la tarjeta, en vez de en la parte superior. Menús de navegación (si los hubiera) se colocan en la zona inferior también. Esto está pensado para uso con una mano en movilidad.

Estos lineamientos de UX son transversales: cualquier nueva funcionalidad añadida en la app deberá respetarlos. Se elaboró una **guía de diseño** con reglas de validación (colores, fuentes, espaciados, etc.) que sirve como checklist para desarrolladores y diseñadores. Por ejemplo, antes de aprobar una pantalla, se valida que todos los textos cumplan la relación de contraste mínima (4.5:1 para texto normal) y que ningún botón quede demasiado pequeño o junto a otro (riesgo de toque erróneo). De esta forma se mantiene una **experiencia coherente y de calidad** en todo el sistema, mejorando la curva de aprendizaje y la eficiencia de uso.

### 8.5 Gestión de errores y resiliencia:

A nivel general, la arquitectura fue diseñada para resistir fallos en puntos externos sin colapsar la experiencia:

* Si el **Servicio IA** no responde al inicio del día, el asesor no podría obtener nuevas recomendaciones. La app en ese caso muestra un mensaje de error y permite reintentar. Como mitigación, se consideró que el último plan descargado podría permanecer cacheado: es decir, si ayer se descargó algo y hoy falla IA, se podría mostrar lo anterior como referencia. Sin embargo, eso puede ser incorrecto (ruta y recomendaciones cambian diario), así que la opción principal es reintentar hasta éxito. En pilotos críticos, se alineó con el equipo IA tener el servicio robusto en las horas pico (quizá precomputando las respuestas antes de las 8am).
* Si la **BD de Ruta** no está accesible (por red corporativa caída, etc.), el impacto es similar: no se puede obtener la lista de clientes. Sin esa lista, ni siquiera se tendría a quién visitar. Una posible resiliencia: permitir al asesor ingresar manualmente un cliente o trabajar con su hoja de ruta física si la tiene. Pero esto escapa del sistema. Entonces se trabajó con TI para garantizar acceso (por eso se priorizó la habilitación en semana 1). La app de cualquier manera maneja el error de "No se pudo descargar la ruta, reintentar".
* En cuanto a **desconexiones intermitentes**, el esquema offline-first ya cubre esa resiliencia completamente: el sistema sigue funcionando offline y reintenta sincronizar cuando puede.
* Para **errores en datos** (por ej, la API IA devuelve un JSON mal formado), el backend incluye validaciones de esquema. FastAPI con Pydantic puede validar la estructura recibida. Si hay algo inesperado, se loguea y se devuelve un error genérico al frontend en lugar de propagar fallo. Así se evita que un error derribe la app del usuario.
* **Errores en frontend (JS)**: se minificaron las posibilidades al probar en diversos escenarios, pero en caso de crash inesperado, el service worker podría servir la última versión estable o una página de fallback "Reiniciar aplicación". Como la app es relativamente sencilla, no se espera memory leaks u otros issues.

### 8.6 Extensibilidad y mantenibilidad técnica:

Aunque es un piloto, se sentaron bases para extender la solución:

* La estructura de **datos polimórficos** de recomendaciones significa que **nuevos tipos de recomendación** (payloads distintos) pueden agregarse sin cambios drásticos: el backend básicamente los transporta tal cual, y el frontend tiene un componente de Tarjeta de Recomendación que muestra título y descripción genérica, adaptando si hay campos específicos. De hecho, la UI está preparada para distintos **tipos de feedback** según feedback\_config – en el ejemplo, un tipo "OPCION\_MULTIPLE" con una lista de opciones se maneja mostrando varios botones. Podríamos en un futuro agregar un tipo de recomendación que requiera ingresar un número (ej. "¿Cuántas cajas vendió?") y la arquitectura lo soportaría mientras definamos un nuevo feedback\_config que la UI interprete (quizá no implementado aún, pero el marco está). El sistema IA enumeró varios posibles tipos y la app está lista para ellos, cumpliendo el objetivo de flexibilidad del diseño.
* La **modularidad del código** (frontend separado en componentes reutilizables, backend con funciones separadas) facilita desarrollar nuevas características. Por ejemplo, si se quisiera agregar un módulo de **notificaciones push** (que avise de una promo especial durante el día), podríamos agregar un servicio worker push listener y un componente en frontend; la arquitectura lo permite sin refactor masivo.
* Se adoptó un enfoque monolítico inicialmente, pero con vistas a **microservicios** a futuro si escala: por ejemplo, si luego quisieran integrar con el sistema de pedidos o con SAP, se puede aislar un servicio de **Order API** separado. La comunicación via REST haría esa transición más simple.
* **Código mantenible:** Se documentó el diseño (este documento arc42 es parte de ello) para que futuros desarrolladores entiendan las decisiones. Además, usar frameworks estándar (React, FastAPI) hace más fácil encontrar talento que pueda continuar el trabajo.

### 8.7 Consideraciones de calidad y pruebas:

* **Rendimiento:** Aunque no un requerimiento crítico, se cuidó que la app sea ágil. El uso de **loading inicial** evita que el asesor espere durante sus visitas – solo carga todo al principio. La cantidad de datos transferidos en la sincronización inicial es relativamente pequeña (lista de clientes del día, quizás 10-20 clientes, cada uno con 2-3 recomendaciones; un JSON probablemente <100 KB). Esto es manejable incluso con 3G. Firestore y Cloud Run tienen baja latencia en la nube, así que la respuesta suele tardar <1s tras login. El mayor retardo podría venir de la DB ruta corporativa si está lejos; se mitigó ubicando servicios en región cercana. El front, una vez cargado, responde en ms a interacciones porque todo está local (mostrar pantallas o leer IndexedDB es muy rápido).
* **Escalabilidad:** Si en lugar de 5 usuarios hubieran 500, Cloud Run aumentaría instancias y Firestore manejaría las escrituras (500 asesores x ~10 feedback cada = 5000 writes, trivial para Firestore). Lo único a escalar sería la API IA (depende cómo esté implementada), pero la arquitectura de nuestra app soporta mayor carga con cambios mínimos (quizá un load balancer front si fuera contenedor propio). Esto se consideró para asegurar que el piloto exitoso pudiera evolucionar a rollout mayor sin rediseño completo.
* **Internacionalización:** No fue necesario ya que la app es para uso interno en español (todos los textos UI están en español). Si se requiriera multi-idioma, React facilita implementar un archivo de strings; es un ajuste futuro.
* **Compatibilidad:** Se probó la WebApp en los navegadores móviles soportados. iOS Safari soporta service workers desde iOS 11+, por lo que se exige asesores tengan al menos iPhone con iOS moderno o Android 7.0+ con Chrome actualizado. De ser necesario, se pudo incluir un *polyfill* para IndexedDB en navegadores viejos, pero el alcance del piloto podía controlar los dispositivos usados.
* **Entorno offline completo:** La PWA se podría "instalar" de modo que guarde incluso los recursos estáticos offline. Así, si un asesor abre la app en zona sin señal pero ya la usó antes, la app arranca desde cache gracias al service worker. Esto es una mejora sobre un sitio web normal donde sin conexión no cargaría nada. Esos detalles se cuidaron para robustez en campo.

En conclusión, los conceptos transversales implementados garantizan que la solución no solo cumpla con las funcionalidades, sino que lo haga de forma **segura, usable y robusta**, manteniendo una **consistencia** en la experiencia y facilitando el trabajo tanto de usuarios finales como de desarrolladores que la mantendrán o ampliarán.

## 9. Decisiones de arquitectura (architectural decisions)

Durante el diseño se tomaron varias decisiones arquitectónicas clave, documentadas aquí en formato ADR (Architectural Decision Record) para explicar el **contexto**, la **decisión** en sí, las **alternativas consideradas** y las **consecuencias**. Esto sirve para entender por qué la arquitectura es como es y qué trade-offs se manejaron.

**ADR 1: Plataforma web independiente vs. extensión de app existente**

* **Contexto:** El sistema necesita permitir a los asesores registrar seguimiento de recomendaciones de IA. La opción inicial podría haber sido integrar esta funcionalidad en la aplicación móvil de rutas ya existente ("Route"). Sin embargo, descubrimos que dicha app no admite modificaciones por restricciones del proveedor/tecnología.
* **Decisión:** Desarrollar una **WebApp nueva e independiente** específicamente para los pilotos de IA.
* **Alternativas consideradas:** (a) Intentar desarrollar un módulo dentro de la app Route – **descartado** porque técnicamente no era posible; (b) Hacer una aplicación móvil nativa separada – descartado por el tiempo limitado y la necesidad de multiplataforma. La WebApp resultó la vía más rápida y flexible.
* **Consecuencias:**
  + **Positivas:** La solución no impacta sistemas productivos, puede evolucionar de forma autónoma y se adaptó exactamente a las necesidades de los pilotos. Permite iterar rápidamente (web es más fácil de actualizar que una app nativa).
  + **Negativas:** Los asesores ahora usan una herramienta adicional aparte de la app de ruta original, lo que podría implicar cierta fricción inicial (tener que alternar aplicaciones). Se mitigó entrenándolos y resaltando el valor de la nueva herramienta. Otra consecuencia es que la WebApp no está integrada en flujo de pedidos (debe usarse paralelamente), pero esto fue aceptado dada la naturaleza aislada del piloto.

**ADR 2: Soportar modo offline (PWA) vs. requerir conectividad continua**

* **Contexto:** Los asesores trabajan en campo, a veces en ubicaciones sin cobertura estable. Se necesita capturar todos los datos aunque no haya internet. Una ruta siempre-online correría el riesgo de perder información o inutilizar la app sin señal.
* **Decisión:** Implementar la aplicación con un enfoque **Offline-First** usando tecnologías PWA (Service Workers, almacenamiento local). Esto garantiza que la app funcione sin conexión y sincronice luego automáticamente.
* **Alternativas:** (a) Diseñar la app solo-online (estilo web tradicional) – **descartado** por riesgo alto de fallas en campo y pérdida de confianza del usuario; (b) Hacer una app híbrida/nativa con base de datos local – se descartó por complejidad, pero la PWA elegida ofrece capacidades similares sin salir del entorno web.
* **Consecuencias:**
  + **Positivas:** Experiencia robusta y confiable; el asesor puede usar la app en cualquier condición, cumpliendo el objetivo crucial de no perder datos. Se obtuvo alta aceptación de los usuarios en pruebas por esta fiabilidad.
  + **Negativas:** Aumentó la complejidad del desarrollo (tuvimos que manejar cache, cola de sincronización, estados adicionales). También se tuvo que probar más casos (sin conexión, reconexión, etc.). Otro impacto es que se depende de características modernas de navegador (actualizar todos los teléfonos para compatibilidad). En balance, fue considerado absolutamente necesario dado que *"una aplicación que pierde datos en campo está destinada al fracaso"*.

**ADR 3: Tecnología frontend (React + Vite) y backend (Python FastAPI)**

* **Contexto:** Se requería escoger el stack de desarrollo principal. Debía permitir construir rápidamente una interfaz rica y una API ligera, con el equipo de desarrollo disponible.
* **Decisión:** Utilizar **React** con herramienta Vite para el frontend, y **Python FastAPI** para construir el backend REST. Esta combinación fue evaluada como *"una elección excelente y moderna"*, alineada a las necesidades offline.
* **Alternativas:** En frontend se consideró Angular (robusto pero más pesado para un MVP y curva de aprendizaje mayor) y Vue (similar a React en capacidad, viable también). Se eligió React porque el equipo ya tenía experiencia previa y existen muchas librerías para PWA. Para backend, la alternativa principal era Java con Quarkus (adecuado para microservicios de baja huella). Se eligió Python/FastAPI por rapidez de desarrollo y porque el equipo de IA maneja Python – posibilidad de reusar código o al menos hablar en mismo idioma. Node.js/Express también era opción, pero la preferencia de la empresa por Python inclinó la balanza.
* **Consecuencias:**
  + **Positivas:** Desarrollo ágil – React aceleró la creación de componentes UI reutilizables, Vite dio tiempos de arranque y build rápidos. FastAPI permitió definir endpoints y esquemas rápidamente, con buen rendimiento y menos código ceremonioso. Python facilitó iteraciones y debugging. Ambos frameworks tienen gran comunidad y soporte. Además, la integración con GCP fue sencilla (Cloud Run soporta contenedores de Python).
  + **Negativas:** Python puede ser menos eficiente en simultaneidad intensiva comparado con Java; no fue problema en piloto pero a gran escala se monitoreará. React en PWA en iOS tiene limitaciones (ej. iOS PWA no soporta background sync periódica perfectamente), pero se pudo manejar manualmente. En general, las consecuencias negativas fueron menores frente a la productividad ganada.

**ADR 4: Base de datos NoSQL independiente vs. reutilizar BD corporativa**

* **Contexto:** Necesitábamos almacenar datos de trazabilidad (feedback, inteligencia) generados por el piloto. Una decisión clave era **dónde almacenar**: ¿en una base existente de Ferguez, en la misma base de datos de rutas, o en una nueva base dedicada?
* **Decisión:** **Usar Cloud Firestore (NoSQL)** como base de datos propia de la WebApp. Es un almacenamiento separado de las bases corporativas, pensado para las necesidades del piloto (flexible, rápido de implementar, en la nube).
* **Alternativas:** (a) Guardar los datos en la base de datos transaccional de Ferguez (ej. en nuevas tablas SQL) – descartado porque habría que coordinar desarrollo en sistemas legacy, riesgo de impacto, y dado el carácter experimental, TI prefería aislamiento. (b) Usar una base SQL en la nube (CloudSQL) – se consideró pero se decidió que un esquema flexible NoSQL convenía más por la naturaleza semiestructurada de los datos (ej. payloads de recomendaciones pueden variar). Además, Firestore ofrece offline sync que se pensó aprovechar si integrábamos directamente en front (aunque finalmente se usó una capa manual offline). (c) No usar base (solo enviar feedback a IA y confiar en que IA los almacene) – no aceptado porque el equipo de proyecto también quería su propia copia de los datos para análisis independiente y respaldo.
* **Consecuencias:**
  + **Positivas:** Firestore permitió almacenar documentos JSON sin predefinir estrictamente todos los campos – muy útil porque el modelo de datos estaba sujeto a cambios según la experimentación. La integración con GCP fue sencilla, sin administrar servidores, y escala suficiente. Tener una base separada asegura que los datos del piloto no mezclan con productivos, eliminando riesgo de corrupción o carga extra en sistemas productivos.
  + **Negativas:** Los datos del piloto quedan aislados, por lo que si se quisiera cruzarlos con datos de ventas históricos (en Data Warehouse) hay que hacer un paso adicional (exportar o conectar BigQuery). Es un compromiso aceptado porque el volumen es pequeño. Otra posible desventaja es costo de Firestore por operación – en este caso trivial por la baja cantidad de escrituras, unos cuantos centavos de dólar, no significativo. Finalmente, la elección de NoSQL implica que consultas relacionales complejas (ej. join con catálogos) no se pueden hacer directamente, pero para este uso (acceso por llave principalmente) es perfecto.

**ADR 5: Interfaz unificada para recomendaciones (plan enriquecido) vs. múltiples llamadas**

* **Contexto:** Para obtener la información del día, necesitábamos la lista de clientes de la ruta y las recomendaciones de IA para cada uno. Inicialmente se planteó que la WebApp leyera la "Hoja de Ruta" de la BD y luego llamara al servicio IA para cada cliente o para la ruta completa. Sin embargo, integrar muchos llamados podría ser ineficiente y complejo.
* **Decisión:** Definir una **API unificada en el Sistema IA que devuelva el plan completo** (clientes ya ordenados con sus recomendaciones), de modo que la WebApp (backend) solo haga **una única llamada GET** por ruta cada mañana. La WebApp todavía necesita acceso a la BD de ruta como respaldo, pero conceptualmente el contrato principal es uno solo (PlanDeRutaEnriquecido).
* **Alternativas:** (a) Hacer que la WebApp backend consulte la DB de ruta y luego llame a IA para cada cliente individualmente (o enviar lote) – esto complicaba la lógica de sincronización y aumentaba tiempo de respuesta, por lo que se descartó. (b) Hacer que la IA se alimente previamente de la lista de clientes (por ejemplo, IA genera su plan en la madrugada y lo guarda) – en cierto modo esto ocurrió, la IA precomputó las prioridades usando BigQuery, pero la interfaz en tiempo real elegida fue on-demand. (c) Integrar la lista de ruta en la API IA mediante un ETL diario – se logró que la API IA conociera los clientes a recomendar usando datos preprocesados, así la webapp no necesita orquestar.
* **Consecuences:**
  + **Positivas:** Simplifica el consumo desde la WebApp – un único request le entrega todo lo necesario para la sincronización inicial. Reduce oportunidades de error (menos puntos de fallo que coordinar). Además, encapsula la lógica de priorización en el lado IA, lo cual tiene sentido ya que ahí está la "inteligencia". Para el equipo IA, esto significa más carga (ellos tuvieron que formar ese objeto PlanEnriquecido), pero se alineó bien con su modularidad.
  + **Negativas:** La WebApp depende aún más de que la API IA esté correcta – si hay un bug en cómo compone el plan, la WebApp no tiene visibilidad granular para corregir sobre la marcha (mientras que si hubieran sido pasos separados, el backend podría, por ejemplo, filtrar o reordenar). Sin embargo, este riesgo se mitigó con una estrecha colaboración y pruebas conjuntas. Otro punto: la WebApp igual retiene la necesidad de leer la DB de ruta para casos en que IA necesite datos actualizados (según la propuesta técnica original), pero al final se optó por no hacer esa doble gestión en run-time, sino confiar en la respuesta de IA (y en todo caso, la BD de ruta se consulta si la IA no puede). En general, esta decisión agilizó la obtención de datos y cumplió con el principio de **modularidad**: la IA expone un servicio pasivo único y la WebApp lo consume de manera straightforward.

**ADR 6: Monolito backend único vs. separar microservicios**

* **Contexto:** El backend de la WebApp debía manejar autenticación, obtener datos de ruta, comunicarse con IA, guardar en Firestore y servir la API al frontend. Se pudo contemplar dividir estas responsabilidades en servicios distintos (por ejemplo, un servicio solo para feedback, otro para sync inicial). Dado el alcance acotado y el tiempo, se inclinó por simplicidad.
* **Decisión:** Implementar un **solo servicio backend (monolítico)** que engloba todas las funciones del servidor (auth, integración, almacenamiento).
* **Alternativas:** (a) Microservicios: tener un microservicio "SyncService" para obtener plan y otro "FeedbackService" para recibir feedback. Esto podría escalar independientemente (quizá feedback es más continuo, sync es matutino). Se descartó por sobreingeniería para un piloto pequeño – hubiese añadido sobrecarga de despliegues, comunicación entre servicios, etc., innecesaria. (b) Serverless puro sin contenedor: usar Cloud Functions separadas para cada endpoint. También se consideró, pero dado que las llamadas comparten lógica (y estado como credenciales, config) era más sencillo empaquetarlo en una aplicación FastAPI.
* **Consecuencias:**
  + **Positivas:** Mucha menor complejidad para construir y mantener. Un solo artefacto a desplegar. En 6 semanas, esto permitió enfocarse en funcionalidad en lugar de en arquitectura distribuida. Performance no sufre porque la carga es baja; incluso con más usuarios, un monolito bien escrito puede manejarlo.
  + **Negativas:** Menor aislamiento – un bug en un endpoint podría afectar a todos (pero se hacen pruebas para eso). Menos escalable granularmente – por ejemplo, no se puede escalar feedback processing aparte del resto, aunque por ahora no es un cuello de botella. También, un monolito está limitado al stack elegido (todo en Python aquí), pero no fue problema. Esta decisión podrá re-evaluarse si la solución crece significativamente: la ventaja es que la modularización interna del código (adaptadores, controladores) ya prepara el camino para extraer microservicios más adelante si hiciera falta (por ejemplo, se podría separar el *FeedbackService* en otro container en un siguiente fase sin reescribir lógica, solo moviéndola).

Cada decisión anterior fue tomada alineada a los **objetivos** (p. ej., la offline-first fue para confiabilidad) y a las **restricciones** (p. ej., stack GCP/React por políticas y tiempo). Documentarlas nos permite recordar por qué no se siguieron otras rutas y qué se espera de cada elección en el contexto del proyecto. Hasta el momento, las decisiones han demostrado ser adecuadas en las pruebas, cumpliendo los objetivos planteados.

## 10. Requisitos de calidad (quality requirements)

A continuación se definen escenarios concretos para los atributos de calidad más relevantes del sistema, siguiendo el formato de *Quality Attribute Scenarios* (SEI): se describe una situación (estímulo y entorno), la respuesta esperada del sistema y cómo medir el cumplimiento. Estos escenarios permiten verificar de forma objetiva si la arquitectura logra sus metas de calidad.

| **Atributo de Calidad** | **Escenario (Estímulo & Entorno)** | **Respuesta Esperada** | **Medida de Cumplimiento** |
| --- | --- | --- | --- |
| **Confiabilidad - Disponibilidad offline** | **Estímulo:** Un asesor pierde la conexión de datos a mitad de una visita (entorno: trabajando dentro de una tienda con señal débil). | **Respuesta:** La aplicación continúa funcionando sin interrupciones. El asesor puede visualizar la información ya descargada de su ruta y registrar el feedback de la recomendación normalmente. La app guarda localmente cualquier dato ingresado durante la desconexión y, cuando regrese la conexión, lo sincroniza automáticamente sin intervención del usuario. No se pierde ninguna entrada de datos. | **Medida:** 100% de los datos capturados en modo offline se sincronizan exitosamente al servidor una vez reestablecida la conexión (verificado en pruebas de campo). Cero incidentes reportados de pérdida permanente de datos debido a desconexión. Uptime percibido por el usuario >= 99% (la app siempre está utilizable incluso sin red). |
| **Usabilidad - Facilidad de uso en campo** | **Estímulo:** Un asesor novato utiliza la app durante su jornada en condiciones reales (entorno: apurado, luz solar directa, una mano ocupada). | **Respuesta:** El asesor puede completar sus tareas en la app de forma rápida e intuitiva. La pantalla principal muestra solo la información esencial de forma clara (cliente y motivo), permitiendo decidir la siguiente visita sin distraerse. Los botones de acción (feedback) son grandes y accesibles con el pulgar, y brindan confirmación inmediata al tocarlos (cambio de estado o vibración). Los textos son legibles bajo luz fuerte gracias al alto contraste de colores. En general, el usuario no comete errores graves y se siente seguro usando la app. | **Medida:** Tiempo promedio para registrar el feedback de una recomendación < **5 segundos** (medido en pruebas cronometradas con usuarios). Nivel de satisfacción de los asesores >= **4 de 5** en encuestas de usabilidad post-piloto (feedback subjetivo indicando facilidad de uso). Cero consultas reiteradas al soporte sobre "¿cómo hago X?" después de la primera semana (indicando que la capacitación mínima fue suficiente). Número de errores de usuario (p.ej., registro duplicado, toques equivocados) tendiendo a cero, gracias al diseño preventivo (desactivación de botones tras selección, etc.). |
| **Rendimiento - Sincronización inicial** | **Estímulo:** Al comienzo de la jornada (8:00 am), el asesor inicia sesión y solicita el plan del día. Hay 15 clientes en su ruta con un total de ~30 recomendaciones generadas (entorno: red 4G promedio). | **Respuesta:** La aplicación completa la descarga inicial de datos de forma rápida, para no retrasar el inicio de actividades. Desde que el asesor pulsa "Iniciar Jornada" hasta que aparece su lista de clientes con recomendaciones, transcurre un tiempo mínimo. Durante el proceso se muestra un indicador de carga transparente ("Descargando tu ruta...") para gestionar la expectativa. La respuesta de la API es eficiente, entregando todos los datos en un solo payload comprimido. La app los almacena localmente inmediatamente y pasa a la vista principal. | **Medida:** Tiempo de sincronización inicial <= **3 segundos** en el 90% de los casos (medido de login a visualización de lista). Tamaño del payload de plan <= **100 KB** (verificado en ambiente de prueba con datos de ejemplo). En pruebas con conectividad débil (3G), tiempo <= **5 segundos**. Estos valores aseguran que el asesor no espere más de lo que tardaría en, por ejemplo, revisar una hoja de papel. Si la red está caída, la app debe mostrar error en < 5 seg y permitir reintento. |
| **Seguridad - Control de acceso y datos** | **Estímulo:** Un usuario no autorizado intenta acceder a la aplicación (entorno: persona con un dispositivo externo intentando usar la URL de la app). | **Respuesta:** El sistema impide el acceso a datos sensibles. La pantalla de login rechaza credenciales inválidas mostrando un mensaje de error claro ("Número o contraseña incorrectos"). Sin autenticación exitosa, no se puede ver ninguna información de rutas ni clientes. Además, si un asesor autorizado pierde su dispositivo, un tercero no debería acceder a los datos offline sin la credencial. La información almacenada offline está asociada a la sesión del usuario autenticado, y la app al cerrarse o después de cierto tiempo podría requerir re-login (podemos implementar expiración de sesión). Todos los datos en tránsito viajan cifrados (HTTPS), protegiendo contra interceptación. | **Medida:** Pruebas de penetración básicas confirman que: (a) no es posible obtener datos de otra ruta modificando parámetros (el backend valida que cada token solo accede a su scope), (b) no se puede eludir el login para ver la app (cualquier URL interna redirige a login si no autenticado), (c) los datos locales en IndexedDB no contienen contraseñas en texto plano ni tokens sin cifrar de larga duración. **Métrica:** 100% de las peticiones sin autenticar son rechazadas por el backend (código 401/403). Ningún acceso no autorizado a datos ocurrió durante el piloto. |
| **Mantenibilidad - Extensibilidad funcional** | **Estímulo:** El equipo decide agregar un nuevo tipo de recomendación de IA ("Descuento Especial") y correspondiente feedback durante la fase 2 del proyecto (entorno: sprint de desarrollo futuro). | **Respuesta:** La arquitectura permite incorporar el nuevo tipo con cambios acotados. Del lado IA, se añade el nuevo tipo enum y su payload; el frontend, gracias a su diseño modular de tarjetas, puede presentar la nueva recomendación usando el mismo componente genérico (solo quizás agregando un ícono o color específico). Si el feedback requiere opciones distintas, se configuran vía feedback\_config sin tener que reestructurar la app. Los desarrolladores encuentran el código organizado por módulos, facilitando la extensión. | **Medida:** Tiempo de desarrollo para soportar el nuevo tipo < **2 días** (incluyendo back y front), indicando que no hubo que "romper" la arquitectura. Número de módulos afectados limitado: idealmente solo se toca la definición de tipos en el modelo de datos, se agrega alguna condición en la UI si necesita ícono especial, y se verifica el flujo de feedback. La métrica cualitativa es que no se requirió reescribir componentes existentes, solo extenderlos. Esto demuestra baja correlación entre componentes (cambio en uno no arrastra cambios masivos en otros). Además, se comprueba que las pruebas automatizadas existentes cubrieron gran parte del impacto (si hubieran). |

Los escenarios anteriores sirven de referencia para validar que el sistema cumple con las **calidades** esperadas. Durante las pruebas finales y el piloto controlado, se monitorearon estos aspectos: efectivamente, la **operación offline fue fluida** (escenario 1 observado en pruebas de campo), la **aceptación de la UX fue alta** (los asesores encontraron útil y fácil la app, confirmando escenario 2), la **performance inicial fue adecuada** (escenario 3: los tiempos medidos estuvieron dentro de lo previsto en ambiente de prueba), la **seguridad básica estuvo cubierta** (escenario 4: no se hallaron brechas evidentes en revisión interna) y la **arquitectura demostró ser extensible** (escenario 5: por ejemplo, se integró el módulo de inteligencia de mercado sin afectar la estructura original, lo cual ya fue una extensión exitosa).

Estos criterios también servirán al finalizar el piloto para evaluar si la arquitectura merece escalarse a producción tal cual o con ajustes. Si todos estos requisitos de calidad se cumplen consistentemente, indicará que el diseño fue acertado.

## 11. Riesgos y deuda técnica

Aquí se enumeran los riesgos arquitectónicos conocidos, junto con su probabilidad e impacto, y las medidas de mitigación planeadas. Asimismo, se listan elementos de deuda técnica (decisiones de implementación sub-óptimas tomadas por conveniencia) que habrá que abordar en el futuro para evitar problemas.

**Riesgos identificados:**

| **Riesgo** | **Probabilidad** | **Impacto** | **Mitigación** |
| --- | --- | --- | --- |
| **Retraso en acceso a datos de "Hoja de Ruta":** Si la integración con la base de datos corporativa no está lista a tiempo, la app no podrá obtener la lista de visitas del día. | Medio | Alto (bloquea funcionalidad principal) | Priorizar esta tarea desde el inicio del proyecto (semana 1), trabajando con TI de Ferguez para obtener credenciales y probar conexión lo antes posible. Como plan de contingencia, preparar datos de ruta *mock* para pruebas de desarrollo, y en caso extremo, cargar manualmente la lista de clientes en Firestore un día antes para no depender de la conexión en vivo (no ideal, solo emergencia). Mantener comunicación constante con el stakeholder técnico proveedora de datos. |
| **Fallas en el modo offline (pérdida de datos):** Existe el riesgo de algún bug en la lógica de almacenamiento local o sincronización que cause que feedback capturado offline no llegue al servidor (por ej., corrupción de IndexedDB, o condición no manejada tras reconexión). | Bajo-Medio | Crítico (socava el objetivo principal y la confianza del usuario) | Realizar pruebas exhaustivas de escenarios offline/online antes del despliegue: simular desconexiones a distintos momentos, múltiples reconexiones, etc. Aprovechar pruebas de campo (UAT) para validar que no se pierda ninguna interacción. Implementar capas de seguridad: ej. usar IDs únicos para cada feedback y que el backend rechace duplicados, así si un feedback se envía dos veces por error no se duplica. Monitorear los logs en producción para detectar errores de sincronización (cualquier error al escribir en Firestore o enviar a IA se debe loguear). Además, comunicar a los usuarios que en caso vean algo extraño (p.ej. icono pendiente que no se sincroniza) lo reporten de inmediato. Este riesgo se reduce significativamente gracias al uso de tecnologías probadas (IndexedDB, ServiceWorker) bien entendidas. |
| **Baja adopción por parte de los asesores:** Que los usuarios no utilicen correctamente la app en campo (por resistencia al cambio, poca claridad, o porque les entorpece) y por ende no se reúnan datos suficientes o de calidad. | Medio | Alto (si no se usa, el piloto no produce valor) | Mitigación principalmente de UX y gestión del cambio: la arquitectura UX se diseñó centrada en la **carga cognitiva mínima** y **facilidad de uso con una mano** para incentivar adopción. Se involucró a algunos usuarios finales en validación de mockups (Kick-off técnico incluyó validación de UX). Además, se proveerá capacitación breve antes del piloto y materiales de apoyo (guía rápida). Durante el piloto, se dará seguimiento cercano: recolectar feedback de los asesores cada semana para mejorar la app si algo los frena. La arquitectura es suficientemente flexible para ajustar texto de interfaz o pequeñas funciones rápidamente si eso mejora la usabilidad. También, dado que los gerentes de ventas apoyan el piloto, se cuenta con su impulso para motivar a los asesores a usar la herramienta (posiblemente vinculando incentivos o recordatorios). |
| **Indisponibilidad de la API de IA en momento crítico:** Si el servicio de recomendaciones no responde en la mañana, los asesores no verán recomendaciones (aunque podrían seguir con visitas). Esto pondría en riesgo el experimento del piloto. | Bajo | Medio (la operación de visitas puede seguir, pero sin IA pierde valor) | Mantener la API de IA altamente disponible: la estrategia fue diseñarla pasiva y simple para reducir puntos de falla. Se realizaron pruebas de carga ligeras y monitoreo. En caso de fallo, la app muestra un mensaje de error y permite reintentar manualmente. Se planeó también obtener el plan de ruta aunque sea sin recomendaciones (si la DB de ruta está disponible pero IA no, se podría presentar la lista de clientes para que al menos la agenda no se pierda, indicando que no hay recos). La comunicación entre equipos es clave: tener un contacto de IA disponible temprano para resolver issues en tiempo real. Dado el bajo número de usuarios, es factible escalar un problema rápidamente (ej., re-deploy de API IA en minutos). En última instancia, si un día entero fallara la IA, el riesgo al proyecto es alto, pero se podría repetir el piloto extendiendo un día más. |
| **Dispositivo o navegador no compatible:** Algún asesor podría tener un teléfono muy antiguo o mal configurado que no soporte PWA (Service Worker) correctamente, causando que la app no funcione offline para él. | Bajo | Bajo-Medio (afecta individualmente, no todo el sistema) | Antes del despliegue, validar la lista de dispositivos de asesores. Idealmente entregar a cada uno un dispositivo corporativo pre-configurado. En caso de BYOD, requerir Android 8+ o iOS 13+ como mínimo. Hacer pruebas en los modelos específicos si se conocen. Instruir a los usuarios a usar Chrome actualizado. Como fallback, la app aún puede usarse online en navegadores más simples pero sin offline (no ideal, pero al menos funcional). Este riesgo es relativamente bajo dado que la mayoría de smartphones en uso cumplen con requisitos modernos. |

**Deuda técnica:**

| **Deuda Técnica (Descripción)** | **Motivo de Incurrencia** | **Plan de Resolución (Pago)** |
| --- | --- | --- |
| **Autenticación simple sin integración corporativa:** El sistema actualmente utiliza un mecanismo de login propio, con credenciales y sesiones manejadas manualmente. No hay integración con SSO corporativo ni segundo factor de autenticación. | Dado el alcance de piloto y la cantidad limitada de usuarios, se optó por la implementación más rápida en lugar de integrar OAuth/Active Directory, etc. Esto ahorró tiempo pero es una solución ad-hoc que no escala organizacionalmente. | Si el proyecto avanza a producción, planear la integración con el sistema de identidad corporativo de Ferguez. Por ejemplo, usar OAuth2/OpenID Connect con los servidores de la empresa, o federar con Google Workspace si lo usan. Añadir MFA si la política lo exige. Este cambio se haría en fase 2; implicará refactor del módulo auth, pero el impacto es controlado (sustituir login/password por flujo token). |
| **No uso de librería de estado global en frontend:** Actualmente el manejo de estados (ej. datos cargados, feedback pendientes) se hace con el estado interno de componentes y contextos React propios. No se introdujo Redux u otra librería centralizada. | Por simplicidad en el MVP y dado que la app es pequeña, se manejó el estado de forma manual. Esto funciona bien ahora, pero a medida que crezca la complejidad (más tipos de datos, más interacciones entre componentes) podría volverse difícil de escalar o propenso a bugs. | Evaluar, en caso de ampliar considerablemente funcionalidades, introducir una librería de manejo de estado global (Redux, MobX, React Context mejor estructurado). Esto permitiría un control más claro de datos como la lista de clientes, feedbacks pendientes, etc. El plan sería refactorizar gradualmente: primero implementar Redux en paralelo leyendo de la fuente actual, luego migrar componentes a usarlo. Por ahora no es crítico, pero se tiene identificado. |
| **Código de sincronización offline acoplado al componente UI:** Parte de la lógica que revisa navigator.onLine y desencadena envío de pendientes está implementada dentro de componentes React (efectos) y en el service worker, de forma distribuida. | Se hizo así por rapidez, pero idealmente la sincronización debería estar encapsulada en un módulo de servicio separado, para facilitar prueba unitaria y reuso. Tal como está, es funcional pero un poco disperso, lo que complica pruebas aisladas (simular offline/online requiere entorno real). | Refactorizar la lógica de sincronización offline en un módulo standalone en el frontend, por ejemplo un hook o un servicio que sea independiente de la UI. Así la UI solo se suscribe a estados (pendientes, sincronizado) y no maneja directamente timers o eventos de conexión. Esto mejorará la mantenibilidad. Planificado para después del piloto, en caso de confirmarse continuidad, ya que durante el piloto funcionó lo implementado y cambiarlo antes implicaba riesgo innecesario. |
| **Cobertura de pruebas automatizadas limitada:** Por el tiempo, se enfocó más en desarrollo funcional que en pruebas automáticas. El backend tiene algunos tests unitarios para modelos, pero pocos tests integrales. El frontend depende mayormente de pruebas manuales de usabilidad; no se implementaron suites de testing automatizado de interfaz (e.g. Selenium o Cypress). | Prioridad dada a entregar la funcionalidad operativa en corto tiempo. Las pruebas se realizaron mayormente manualmente por el equipo (incluyendo pruebas de campo controladas). Esto es manejable en un piloto pequeño, pero a largo plazo puede introducir riesgo de regresiones. | Incorporar tests adicionales una vez estabilizada la versión piloto: por ejemplo, agregar pruebas de integración para el backend (simulando llamadas a endpoints principales y verificando respuestas y side-effects en Firestore). En frontend, para futuras iteraciones, considerar al menos tests unitarios de funciones utilitarias y potencialmente pruebas end-to-end con herramientas como Cypress para simular flujo offline/online. Pagar esta deuda mejorará la confianza en despliegues futuros, especialmente si la base de usuarios crece. |

Estas deudas técnicas no afectaron el éxito inmediato del piloto, pero conviene atenderlas antes de un escalamiento a producción para asegurar la **sostenibilidad** de la solución. Documentarlas ayuda a planificar su resolución en siguientes fases y a justificar ante stakeholders por qué podrían necesitar inversión de tiempo.

La actitud proactiva hacia los riesgos (monitoreo, pruebas piloto intensivas, planes de contingencia) y la transparencia sobre la deuda técnica garantizan que el proyecto se maneje con expectativas realistas y con mejora continua en mente. Como nota final, no se encontraron riesgos críticos sin mitigar; el proyecto está consciente de los puntos frágiles y tiene acciones asignadas para ellos, lo cual es un indicador de buena gestión arquitectónica.

## 12. Glosario (glossary)

Este glosario define términos, siglas y conceptos clave utilizados en el contexto de este proyecto, para asegurar un lenguaje común entre todos los involucrados.

| **Término** | **Definición** |
| --- | --- |
| **Asesor (de Ventas)** | Persona encargada de visitar clientes (tiendas) en ruta, tomar pedidos y gestionar la relación comercial. Usuario final de la WebApp, también referido como *vendedor de ruta* o *representante*. |
| **Hoja de Ruta** | Lista de clientes que un asesor debe visitar en un día. Definida por el sistema logístico de Ferguez. La WebApp sincroniza esta lista al inicio del día. |
| **Plataforma Táctica Web** | Nombre del sistema desarrollado (la WebApp) que proporciona inteligencia de punto de venta. "Táctica" indica su uso en ejecución diaria (no estratégica analítica), *web* implica que funciona en navegador. |
| **Piloto (de IA)** | Prueba experimental de modelos de Inteligencia Artificial en un entorno controlado. En este caso, se refiere a dos pilotos: uno de aumento de *Drop Size* y otro de conversión a *No Retornables*. |
| **Drop Size** | Métrica de volumen por entrega (venta promedio por visita). Aumentar el *Drop Size* implica vender más producto por cada visita/entrega realizada, uno de los objetivos del piloto IA. |
| **No Retornable (NR)** | Se refiere a envases no retornables (ej.: botellas/cajas que no se devuelven). El piloto de *Conversión a NR* busca mover clientes de envases retornables a no retornables, para eficiencias logísticas. |
| **Recomendación (de IA)** | Sugerencia o acción específica generada por el modelo de IA para un cliente en particular. Clasificamos las recomendaciones en **Nivel 1** (el "Porqué" o motivo de la visita) y **Nivel 2** (el "Qué", acciones concretas). Cada recomendación tiene un tipo (ej. ALERTA\_QUIEBRE\_STOCK, SUGERENCIA\_PORTAFOLIO) y un contenido asociado. |
| **Nivel 1 (Recomendación)** | Prioridad o motivo principal de la visita a un cliente, determinado por IA. Responde a "¿Por qué es importante visitar a este cliente hoy?". Ej: *"Visita Prioritaria: Riesgo de Abandono"*. Se muestra destacada en la pantalla de cliente. |
| **Nivel 2 (Recomendación)** | Acción/es recomendadas a ejecutar con el cliente, derivadas del motivo principal. Responde a "¿Qué hacer durante la visita?". Son específicas y accionables. Ej: *"Sugerir 2 cajas adicionales de producto X"*. Se registran individualmente con feedback. |
| **Feedback (Retroalimentación)** | Información capturada sobre el resultado de una recomendación. Incluye si el asesor siguió la sugerencia y cuál fue la respuesta del cliente (aceptó, rechazó, motivo), además de una calificación o comentario opcional. Cada feedback corresponde a una recomendación Nivel 2 y se envía al sistema IA para evaluar efectividad. |
| **RegistroFeedback (Objeto)** | Formato de datos enviado al backend (y al sistema IA) representando un feedback de una recomendación. Contiene identificadores (plan, cliente, recomendación), resultado seleccionado, timestamp de captura, geolocalización, etc.. |
| **Inteligencia de Mercado** | Datos cualitativos y cuantitativos recolectados en campo sobre la situación del punto de venta, aparte de las recomendaciones. Ejemplos: quiebres de stock detectados, inventarios estimados, precios de la competencia, visibilidad de producto. Se capturan mediante módulos adicionales en la app y sirven para nutrir análisis y modelos. |
| **Quiebre de Stock** | Situación en la que un producto debería estar a la venta en el anaquel pero está agotado. El asesor puede reportar estos *quiebres* en la app (seleccionando producto y motivo). El modelo IA también prioriza visitas si predice quiebres inminentes (Alerta de Quiebre de Stock). |
| **SKU** | *Stock Keeping Unit*, código identificador de un producto específico. Utilizado en la app para referirse a productos en módulos de inteligencia de mercado (ej. seleccionar SKU de la lista). |
| **Material POP** | Material de Publicidad en el Punto de Venta (ej: carteles, exhibidores, refrigeradores con branding). La app permite registrar presencia/ausencia de material POP como parte de la ejecución en tienda. |
| **Service Worker (SW)** | Script que el navegador ejecuta en segundo plano para habilitar funciones offline y de red en la PWA. Intercepta peticiones, cachea archivos y permite funcionalidad como recibir notificaciones push. Clave para que la app opere offline. |
| **IndexedDB** | Base de datos local del navegador. Almacena pares clave-valor o datos JSON estructurados. Utilizada por la app para guardar el plan de ruta y feedbacks mientras está offline. Persistente incluso si se recarga la página o se cierra el navegador. |
| **PWA (Progressive Web App)** | Aplicación web con capacidades avanzadas similares a una app nativa: funciona offline, puede instalarse en la pantalla de inicio, enviar notificaciones, etc. Nuestra WebApp es una PWA offline-first, aprovechando service worker y manifest. |
| **Cloud Run** | Servicio de Google Cloud para ejecutar contenedores de forma serverless. Escala automáticamente y cobra por uso. Usado para desplegar el backend de FastAPI en este proyecto. Permite exponer endpoints HTTPS sin gestionar servidores. |
| **Firestore** | Base de datos NoSQL en la nube de Google (parte de Firebase/Google Cloud). Almacena documentos JSON organizados en colecciones. Elegida para guardar datos de trazabilidad (feedback, inteligencia) de la app. Ofrece escalabilidad automática y API sencilla. |
| **FastAPI** | Framework web en Python para construir APIs REST de alto rendimiento. Utilizado en el backend para definir endpoints (login, plan, feedback, etc.). Con sintaxis declarativa (basada en Python 3.7+ type hints) y servidor Uvicorn. |
| **Quarkus** | Framework Java optimizado para contenedores (nativo en GraalVM) ideal para microservicios cloud. Se consideró como alternativa al backend Python, pero finalmente no fue usado en este piloto. |
| **ADR (Architectural Decision Record)** | Formato de documentación de decisiones de arquitectura. Se incluye en este documento la sección 9 con varias ADRs que explican elecciones importantes (tecnología, patrones, etc.) junto con sus razones. |
| **UAT (User Acceptance Testing)** | Pruebas de aceptación por usuarios. En el contexto de este proyecto, la semana 5 incluyó pruebas de campo con usuarios reales (asesores) utilizando la app en situaciones reales, para validar que cumple requerimientos y es aceptada antes del despliegue final. |
| **KPI** | *Key Performance Indicator* – Indicador clave de rendimiento. En el proyecto, los KPIs piloto incluyen Drop Size, tasa de conversión a NR, recompra semanal, etc., que se monitorean para medir éxito. La WebApp ayuda a influir en KPIs (a través de recomendaciones) y a medir algunos (capturando feedback). |
| **A/B Testing (Rutas Gemelas)** | Método de experimento dividiendo sujetos en grupo de control (A) y experimental (B). Se mencionó el concepto de "rutas gemelas" (emparejar rutas similares) y diseño de muestra A/B. Indica que algunas rutas usarán la IA (B) y otras no (A) para comparar resultados. La WebApp se usa solo en rutas B durante el piloto, presumiblemente. |
| **Interware** | Empresa consultora de tecnología involucrada en el proyecto (mencionada en la propuesta técnica). Actúa como desarrollador e integrador de la solución (proveedor de Heineken/Ferguez). Su equipo implementó la WebApp y colabora con IA. |
| **Ferguez** | Nombre alias para la compañía cervecera cliente (posiblemente Heineken México, dado el contexto). Se usa "Ferguez" para referirse a la organización dueña del proyecto, sus sistemas y personal (gerentes, TI, etc.). |
| **Heishop** | Mencionado como "API Heishop" en documentos, al parecer es una plataforma o sistema de e-commerce/ventas existente. No interactúa directamente con nuestra WebApp, pero se revisó su API durante el proyecto para evaluar ideas (e.g. bonificaciones). Lo incluimos solo para contexto: no forma parte de la arquitectura de esta solución. |

El glosario unifica términos de negocio (ej. No Retornabilidad), términos técnicos (PWA, IndexedDB) y nombres propios relevantes (Interware, Ferguez) para que todos los stakeholders, sin importar su trasfondo, puedan comprender con precisión las descripciones en este documento y en las discusiones del proyecto. En caso de duda, este glosario debe ser la referencia para evitar confusión de nomenclatura.