Informe Técnico Final: Prototipo Colaborativo LIA

Proyecto: LIA (Localization Intelligent Automated) **Tipo:** Prototipo Funcional de Aplicación Web Colaborativa (MVP Avanzado) **Arquitectura:** Mobile-First, Serverless (BaaS) **Tecnologías Clave:** React (Next.js), Firebase (Firestore, Realtime Database, Auth), Tailwind CSS

1. Resumen Ejecutivo

LIA es un prototipo funcional de alta fidelidad que valida una solución colaborativa para el mapeo y la gestión de información en la Universidad del Cauca. El sistema resuelve el problema de la obsolescencia de los mapas estáticos implementando un ciclo de colaboración en tiempo real, donde la propia comunidad es responsable de mantener los datos actualizados.

El prototipo se centra en tres pilares funcionales:

1. **Colaboración Basada en Roles:** Un flujo de trabajo estricto (Comunidad -> Reportero -> Validador) que asegura la integridad de la información desde su creación hasta su publicación.
2. **Gamificación (Motivación):** Un sistema de puntos, rankings con medallas (🏅) y premios que incentiva la participación activa.
3. **Awareness (Conciencia Colectiva):** Múltiples mecanismos (feeds de actividad, estado "resuelta" en búsquedas, chat general y lista de usuarios online/offline) que mantienen a la comunidad conectada e informada.

La arquitectura es completamente *serverless* (sin servidor), con toda la lógica de negocio y estado en tiempo real gestionada por los servicios de Firebase y ejecutada en el lado del cliente (React).

2. Arquitectura Técnica y Herramientas

La arquitectura fue diseñada para ser *mobile-first* y altamente escalable, priorizando la sincronización instantánea de datos con un coste de infraestructura nulo (Plan Spark de Firebase).

- **Frontend: Next.js (React)**. Se utilizó para el renderizado del lado del cliente (`"use client"`), el enrutamiento basado en archivos (`/login` , `/`) y la gestión del estado global a través de `Context API` (`AuthContext.js`).
- **Estilos: Tailwind CSS**. Se empleó para un diseño *utility-first* rápido y responsivo, permitiendo la creación de los complejos paneles deslizables.
- **Animación: Framer Motion**. Se usó para todas las transiciones de UI (paneles, menús), proporcionando una experiencia de usuario fluida y profesional.
- **Autenticación: Firebase Authentication**. Gestiona el registro, inicio de sesión y el estado de la sesión del usuario.
- **Base de Datos (Datos Persistentes): Cloud Firestore (Firestore)**. Es la base de datos principal para *todo* el contenido persistente (usuarios, reportes, espacios, mensajes de chat). Su función clave es `onSnapshot` , que permite a la UI suscribirse a cambios en tiempo real.

- **Base de Datos (Estado Efímero): Firebase Realtime Database (RTDB)**. Se utiliza *específicamente* para una función que Firestore no puede manejar: la **gestión de presencia (online/offline)**. Su función clave `onDisconnect` garantiza que el estado de un usuario se elimine automáticamente si cierra el navegador.

3. Estructura de Datos (Bases de Datos)

3.1. Cloud Firestore (db)

1. `users` : Almacena el perfil de cada usuario.
   - *Campos Clave:* `displayName` , `email` , `role` (Comunidad, Reportero, Validador), `points` .
2. `spaces` : Almacena los puntos *validados* y visibles en el mapa.
   - *Campos Clave:* `name` , `type` , `location` (coordenadas), `status: "Validado"` .
3. `reports` : "Bandeja de entrada" de reportes pendientes de validación.
   - *Campos Clave:* `name` , `type` , `location` , `status: "Pendiente"` , `reportedBy` .
4. `searchRequests` : Solicitudes de búsqueda publicadas por la comunidad.
   - *Campos Clave:* `searchQuery` , `searchType` , `status` ("pendiente" o "resuelta"), `requestedBy` .
5. `activity_log` : Historial de acciones para los feeds de actividad.
   - *Campos Clave:* `message` , `timestamp` , `userId` .
6. `chat_messages` : Almacena el historial del chat general.
   - *Campos Clave:* `text` , `senderName` , `senderId` , `createdAt` .

3.2. Realtime Database (db_rtdb)

1. `status/{userId}` :
   - **Propósito:** Almacena el estado de conexión *actual* de un usuario. Es una ruta efímera.
   - *Campos Clave:* `online: true` , `role` , `name` , `last_changed` .

4. Funcionalidades Clave y Flujos de Colaboración

4.1. Gestión de Identidad y Presencia (AuthContext.js)

El `AuthContext.js` es el componente de nivel superior que gestiona el estado del usuario.

- **Autenticación:** Utiliza `onAuthStateChanged` para detectar el inicio/cierre de sesión.
- **Roles:** Al iniciar sesión, consulta el documento del usuario en Firestore (`users`) para obtener su `role` y `points` , proveyéndolos al resto de la app.
- **Presencia (RTDB):** Inmediatamente después de obtener los datos del usuario, escribe en la ruta `status/{userId}` de la RTDB que el usuario está `online: true` .

- **Desconexión:** Utiliza `onDisconnect().remove()` para instruir a Firebase que borre ese registro de RTDB si el usuario se desconecta abruptamente. La función `logOut` también borra este registro manualmente para una desconexión limpia.

4.2. Flujo de Colaboración: Reporte y Validación

Este es el ciclo central de la aplicación, implementado de forma segura en el cliente mediante **Escrituras por Lote (Batched Writes)**.

1. **Reporte (Rol: Reportero):** Un Reportero hace clic en el mapa. El `PanelInteractivo.jsx` se abre. Al enviar, un `writeBatch` crea atómicamente dos documentos: uno en `reports` (status: "Pendiente") y otro en `activity_log`.
2. **Validación (Rol: Validador):** El `PanelValidacion.jsx` escucha (`onSnapshot`) los reportes pendientes. Al hacer clic en "Aprobar", la función `handleApprove` ejecuta un `writeBatch` que realiza **6 operaciones atómicas**:
    1. **Actualiza** el `report` a `status: "Validado"`.
    2. **Crea** el nuevo `space` (haciéndolo visible en el mapa para todos).
    3. **Consulta** (`getDocs`) y **Actualiza** las `searchRequests` coincidentes a `status: "resuelta"`.
    4. **Suma** +5 puntos al Reportero (usando `increment()`).
    5. **Suma** +3 puntos al Validador (usando `increment()`).
    6. **Crea** un `activity_log` (mensaje: "X aprobó Y").

4.3. Flujo de Awareness: Búsqueda Colaborativa

Este flujo conecta la necesidad de la comunidad con la acción de los contribuidores.

1. **Comunidad:** En `PanelComunidad.jsx`, un usuario busca un lugar. Si la búsqueda local (en `spaces`) falla, puede "Publicar Búsqueda", creando una `searchRequest` ("pendiente").
2. **Reportero/Validador:** Sus paneles (`PanelValidacion.jsx`, `PanelBusquedasPendientes.jsx`) escuchan las `searchRequests` pendientes y les notifican.
3. **Cierre de Ciclo:** El `writeBatch` del Validador (Paso 4.2) cierra automáticamente la solicitud.
4. **Notificación:** La pestaña "Mis Búsquedas" del usuario Comunidad se actualiza en tiempo real, mostrando la solicitud como "Resuelta".

4.4. Componentes de UI Principales

1. `MenuPrincipal.jsx` **(Navegación Global):**
    - Es un menú desplegable (hamburguesa) con `z-index: 50` que maneja funciones globales.
    - Utiliza un estado interno (`modalContent`) para renderizar "sub-páginas" (Acerca de, Trofeos, Chat, Usuarios) dentro de un modal animado, sin perder el contexto de la app.
    - **Trofeos:** Muestra los premios para el podio (🥇 1er, 🥈 2do, 🥉 3ro).

- **Cerrar Sesión:** Redirige a `/login` (no requiere lógica de `AuthContext`).

2. `ChatTab.jsx` y `UsuariosTab.jsx` **(Componentes Sociales):**
   - **Chat:** Componente reutilizable que escucha la colección `chat_messages` de Firestore y permite enviar nuevos mensajes.
   - **Usuarios Online/Offline:** Componente avanzado que **combina dos fuentes de datos**:
     1. Obtiene (`getDocs`) la lista de *todos* los usuarios de **Firestore** una sola vez.
     2. Escucha (`onValue`) los cambios de estado en la ruta `status/` de la **RTDB** en tiempo real.
     3. Usa `useMemo` para calcular y renderizar una lista combinada, categorizada por rol y estado (Online/Offline).
3. **Paneles de Rol (Paneles Deslizables):**
   - `PanelComunidad.jsx`, `PanelValidacion.jsx`, `PanelBusquedasPendientes.jsx`: Son los paneles principales de interacción, diseñados como "Bottom Sheets" (`framer-motion`) para una UX móvil nativa.
   - Todos incluyen interfaces de pestañas para organizar la información (Reportes, Búsquedas, Actividad, Ranking, Chat, Online).
4. `PuntoInteres.jsx` **(Pines del Mapa):**
   - Se rediseñó para usar **Iconos SVG** en lugar de emojis, mejorando la legibilidad.
   - Muestra una etiqueta de texto siempre visible.
   - Cambia de color y estilo según el tipo de lugar.
   - Acepta un prop `isHighlighted` que (gestionado por `page.js`) permite que la búsqueda resalte un pin específico con una animación y `z-index` elevado.