ChainLynx Bikepacking App – Architecture Overview

Pre-Release Technical Documentation — November 2025

1. Overview

The most viable approach for your requirements is a **hybrid architecture**.

A purely decentralized model (e.g., peer-to-peer) presents significant challenges for data discovery and the persistence of shared community data.

The hybrid model offers the best of both worlds:

- **User-Controlled Personal Data**
  All personal and sensitive user data (journals, private routes, settings, chat logs) is stored directly in the user's own cloud account (iCloud or Google Drive).
  Your backend never sees or stores this data.
- **Lightweight Central Service for Shared Data**
  A minimalist, privacy-respecting central server manages shared, persistent community data, primarily Points of Interest (POIs).
  This ensures that valuable community contributions remain available to everyone, even if the original contributor leaves the service.

This model makes a clear distinction between **"what is mine"** and **"what is ours"**, which is the cornerstone of its privacy-first design.

---

2. Data Storage Strategies

A. Personal Data — User-Owned Cloud Storage

This is the core of the app's privacy promise. The React Native application functions as a client for the user's personal cloud.

Mechanism

- Upon onboarding, the user grants the app permission via OAuth to access a sandboxed, app-specific folder in their Google Drive or iCloud Drive.
  The app's access is strictly limited to this folder.
- All personal data is written to and read from this folder.
  The app manages synchronization between the local device cache and the user's cloud.

Data Formats

- **GPS Tracks:** GeoJSON or GPX

- **Journals / Notes:** Markdown (.md)
- **App Settings:** JSON

React Native Implementation

- **Google Drive:**
  Use @react-native-google-signin/google-signin for authentication and the Google Drive REST API (via fetch or axios) to manage files.
- **iCloud:**
  Use native modules such as react-native-cloud-store or a custom native bridge to interact with Apple's CloudKit or iCloud Drive APIs.

Challenges

- **Sync Logic:**
  Implement robust synchronization logic, including conflict resolution (e.g., edits made on two devices while offline).
- **API Complexity:**
  Handle errors for rate limits, expired tokens, and insufficient storage gracefully.

B. Shared Data — Minimalist POI Service

To ensure community-submitted POIs are persistent and moderated, they must be managed by a lightweight central service.

Mechanism

- A user submits a new POI (e.g., water source, campsite) from the app.
- The app sends this data to the central backend API.
- The backend validates and stores the POI data in a geospatial database, with an **anonymized link** to the contributor (not publicly exposed).
- The app fetches and caches the global POI list from the API for offline use.

Persistence

If a user deletes their account, the anonymous link is severed, but their contributed POIs remain available for the community — a key privacy feature.

Alternative: Federation

Using a Mastodon/ActivityPub backend for true federation is theoretically possible but introduces extreme complexity in moderation, discovery, and synchronization.

It is **not recommended** for an initial build.

---

3. Privacy and Security by Design

- **No Central User Accounts:**
  The backend stores no user profiles, emails, or passwords. Authentication is handled via OAuth with cloud providers.
- **Anonymized Contributions:**
  The backend stores only POI data and a non-reversible hash of the user's unique ID.
  This allows banning malicious contributors without holding any personal data.
- **Minimal Data Collection:**
  The server only manages shared POIs and has zero knowledge of user location, private routes, or journals.
- **End-to-End Encryption (E2EE):**
  Any future messaging must use E2EE (e.g., the Signal protocol).
  The server only relays encrypted data and cannot access message contents.
- **Offline AI:**
  All AI assistants operate entirely on-device.
  No voice or other data is transmitted to external services.

---

4. Third-Party Data Integration

Direct integration with other community apps (e.g., iOverlander) is usually impossible, as these services lack public APIs.

Recommended Approach: User-Driven Import/Export

- Support open formats such as GPX, KML, and GeoJSON.
- Allow users to export data from other platforms and import it manually into your app.

This approach respects other platforms' terms of service while providing flexibility and data portability.

---

5. User Authentication

Authentication is intrinsically linked to the data storage strategy.

- **Primary Authentication:**
  Users sign in with Google or Apple via OAuth 2.0, granting access only to the app's sandboxed folder in their cloud account.
- **Backend Authentication:**
  When performing a shared action (e.g., submitting a POI), the app sends the OAuth identity token to the backend.
  The backend verifies it with the provider and issues a short-lived session token (JWT) tied to the user's unique ID (sub claim).

This confirms user legitimacy without ever storing personally identifiable information (PII).

---

6. Technology Stack Recommendations

Backend Framework

- **Node.js (Fastify / Express):** Ideal for lightweight I/O-bound APIs with strong ecosystem support.
- **Go (Gin):** High performance and resource-efficient; produces a single deployable binary.

Database

- **PostgreSQL + PostGIS:**
  Industry standard for geospatial applications. Enables efficient spatial queries (e.g., "find all POIs in map view").

Deployment

- **VPS (DigitalOcean, Linode):**
  Simple and cost-effective for running PostgreSQL with PostGIS.
- **Serverless (Vercel, AWS Lambda) + Managed Database (Neon, Supabase):**
  Pay-per-use scaling and built-in PostGIS support.

---

7. Alternative Architectural Models

(Ranked by degree of decentralization)

1. **Hybrid Model (Recommended)**
   - **Strengths:** Balances privacy, user control, and persistence. Proven, achievable.
   - **Weaknesses:** Relies on a small central server (a point of trust).

2. **Federated Model (ActivityPub)**
    - **Strengths:** Highly decentralized, censorship-resistant.
    - **Weaknesses:** Complex discovery, inconsistent moderation, slow cross-instance queries.
3. **Pure P2P / Local-First Model**
    - **Strengths:** Ultimate privacy and decentralization; no central server.
    - **Weaknesses:** Experimental and unreliable; data persistence and discovery remain unsolved challenges.

---

8. Mapping Architectural Solutions

Mapping is the functional heart of the ChainLynx app.

Because users rely on maps entirely offline, this subsystem must balance high performance, minimal bandwidth, and seamless integration with the hybrid privacy model.

A. Core Mapping Architecture

- **Offline-First Map Engine:**
  Choose a mobile-optimized SDK that supports vector tile rendering and caching:
    - MapLibre GL Native (open-source, privacy-friendly)
    - Mapbox SDK (commercial, telemetry considerations)
    - Self-hosted OpenStreetMap tiles for full control
- **Vector Tile Strategy:**
  Offline map regions are stored as compact .mbtiles or .pmtiles vector packages, downloadable per route.
  Each **Route Package** contains:
    - Route GPX/GeoJSON track
    - Map tiles (MBTiles/PMTiles)
    - POIs and elevation layers (GeoJSON)
    - Route metadata and guide text (Markdown)

This ensures that once downloaded, the full navigational experience — including layers and AI — is available offline.

B. Data Layers and Rendering

Layer Hierarchy

1. Base Map Layer — vector tiles (terrain, topo, satellite)
2. Route Layer — GPX/GeoJSON polyline for the active route
3. POI Layer — user and community POIs (water, shelter, food, hazards)
4. Dynamic Data Layer — trail reports and rider locations (online only)
5. Elevation Profile Overlay — synchronized with route position

All layers follow open standards to ensure full cross-platform compatibility.

C. Synchronization & Caching

- Personal POIs and notes → stored in the user's cloud as GeoJSON.
- Shared community POIs → fetched from the central API and cached locally in SQLite.
- Conflict resolution → use timestamps and hashes to detect duplicates.
- Tile cache → reuse regions and manage with an **LRU eviction** policy to save space.

D. Geospatial Backend

- **Database:** PostgreSQL + PostGIS for spatial indexing (ST_Within, ST_Distance, ST_Intersects).
- **API Endpoints:**
    - GET /pois?bbox=<bounds> → returns POIs in current map view
    - POST /poi → submits a new anonymized POI
    - GET /routes/:id → fetches route metadata and stats

This enables efficient, privacy-respecting map queries without exposing personal data.

E. Elevation Data Integration

- Use **SRTM** or **ALOS DEM** datasets (converted to .hgt or tiled GeoTIFF).
- Pre-process and bundle within route packages for offline use.
- Render elevation profiles using react-native-svg-charts or d3-shape, synchronized with the route cursor.

F. Privacy & Security Considerations

- No telemetry or third-party analytics in map or tile requests.

- No background location tracking; GPS access only with user consent.
- Encrypt cached map data and route packages.
- Use open data sources (OSM, open elevation) to avoid corporate tracking.

G. Scalability & Future Extensions

- **Custom Tile Server:**
  Optionally deploy **TileServer GL** or **Tegola** for curated OSM-based tiles (contours, paths, relief).
- **Federated POI Overlays:**
  Future support for importing overlays from trusted communities or partners.
- **AI Integration:**
  Offline AI models can process spatial context (“nearest water source within 10 km”) using local GeoJSON data.
