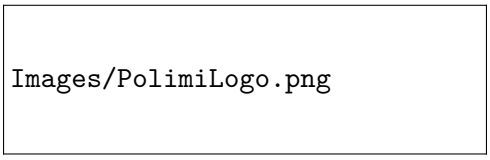
POLITECNICO DI MILANO

SOFTWARE ENGINEERING 2 - PROJECT 2025-2026



Best Bike Paths (BBP)

Design Document

Authors:

Rajatkant Nayak (11144180)
Shashi Bhushan XXX (11111318)

January 30, 2026

Contents

1 Introduction

1.1 Purpose

The primary objective of this Design Document is to provide a comprehensive technical blueprint and architectural specification for the **Best Bike Paths (BBP)** system. While the Requirement Analysis and Specification Document established the functional goal such as user scenario and environmental constraint from an external stakeholder perspective so this document serves as the formal transition into the implementation, the integration and deployment phase.

This Design Document (DD) establishes the architectural specifications required to satisfy the functional requirements defined in the RASD. It delineates the mapping between functional goals and software components, ensuring traceability and technical validity.

1.1.1 Technical Rationale and Engineering Objectives

This document shows the low level specification required to address the high complexity of a crowd sourced cycling safety ecosystem particularly for groups of two student focusing on automated data collection. Specifically the BBP design details how the system will achieve the following technical milestones:

- **High-Frequency Sensor Processing and Anomaly Detection:** The framework must establish a reliable sampling service in Flutter using the sensors_plus plugin in order to handle IMU data consistently at 50Hz. Because you want to capture abrupt, dramatic objects like potholes or damaged tram lines, achieving that level of precision is essential. The data becomes hazy when the sample rate is lowered, which makes it possible to miss those unique occurrences because of aliasing or simply not recording them at all.
- **Advanced Spatial Data Management:** Using a PostgreSQL database enhanced with a PostGIS extension via the Supabase platform, the design describes how to manage complex geographic operations. This includes mapping raw GPS coordinates to specific `StreetSegment` entities and employing a weighted routing technique that prioritizes path safety scores $(0 - 100)$ over simple geographic distance.
- **Human-in-the-Loop Data Integrity:** The design created the runtime logic for the Verification Protocol to reduce the possibility of false positives or hallucinated data, such as phone drops or deliberate curb jumps.Before the data is added to the community-wide risk map, the system places machine-generated Candidate Anomalies in a temporary verification queue that requires the cyclist's explicit manual confirmation.
- **Resource Optimization and Energy Efficiency:**One of the primary design factors is the strict non-functional requirement to keep energy consumption under 15
- **DynamData Persistence and Retrieval that settles contradictory reports from different users. By considering the statistical volume of confirming data and the recentness of reports, the technology ensures that the map is accurate and depicts the current state of urban bicycle infrastructure.**

1.1.2 Target Audience

This document is prepared to serve a diverse group of technical stakeholders throughout the software development lifecycle:

- **Software Developers:** To comprehend the Supabase Edge Functions and the Flutter client-side logic's modular structure.

- **System Architects:** To investigate the interactions between the mobile front-end, the relational database, and external APIs such as OpenStreetMap and OpenWeatherMap.
- **Quality Assurance (QA) and Testing Teams:** To create strict integration and unit test cases using the designated component interfaces and sequence diagrams.
- **Project Supervisors:** To verify that the technical design satisfies the academic requirements of the Software Engineering 2 course at Politecnico di Milano.

1.2 Scope

All the architecture of the Best Bike Paths (BBP) system is explained in the forthcoming design documentation. We discuss the entire thing – from using your smartphone for the collection of raw data to the cloud analysis. A mobile app, also referred to as a client, and a server are the most important aspects of a greater architecture as a whole. Given that the app must be able to communicate even when you are offline, the focus is on a manner that involves the two above entities communicating. This is a very important aspect of urban sensing.

1.2.1 World Phenomena and Design Implications

The system must interact with and interpret entities in the physical world. The architectural design explicitly accounts for these external factors through specific software modules:

- **The Cyclist (Human Actor):** *Design Implication:* While the cyclist has the primary responsibility for the verification process, the UI design clearly separates Active Riding and Stopped/Reviewing under conditions of high cognitive load from traffic interaction. Furthermore the software architecture should include state machine capabilities to limit any complex interaction when the GPS indicates that the cyclist speed exceeds a safe threshold (≥ 0 km/h).
- **The Road Network (Physical Infrastructure):** *Design Implication:* Temporary risks and permanent risks are the two classifications of road surface area problems. The Supabase database format distinguishes between the two classifications of risk by creating a unique time to live (TTL) for each risk based on the time that the risk has been classified as active.
- **The Bicycle (Mechanical Interface):** *Design Implication:* Bicycles filter out road vibrations through their frames etc. before transmitting vibrations to smartphones. Flutter has created sensor processing algorithms that use high-pass filtering to eliminate background noise associated with bicycle movement/pedaling and capture just the abrupt vertical inflections (shocks) that occur on the Z-axis.
- **Environmental Conditions (Weather and Lighting):** *Design Implication:* The safety conditions of a path are adversely affected in rainy, icy, and dark environments. The routing engine is made capable of dynamic Slippery penalties only when the backend `ExternalServiceGateway` component automatically polls the OpenWeatherMap API at the start of every trip annotating the ride data with weather information.
- **GPS Signal Availability (Spatial Constraints):** *Design Implication:* Spatial blackouts may also occur, such as tunnels or urban canyons in metropolitan settings. A LocationService module should include a Dead Reckoning fallback and also a local buffer that stores sensor anomalies timed when a signal is lost and attempts to resynchronize them once a valid coordinate is recovered.

1.2.2 Shared Phenomena and System Interfaces

These above mentioned phenomena demonstrate the control limits and data signals exchanged between the software machine and its environment. The technical interfaces, for example, monitoring and control, are defined in the system design as follows:

- **Inertial Sensor Telemetry (Input):** *Technical Interface:* Through the Flutter `sensors_plus` channel, the system subscribes to the hardware `SensorEventListener` (Android) or `CoreMotion` (iOS). In order to perform real-time threshold analysis, the software buffers raw 3-axis acceleration values (x, y, z) measured in m/s^2 at a required sampling rate of 50Hz.
- **Global Positioning Signals (Input):** *Technical Interface:* The spatiotemporal data packets containing information regarding latitude, longitude, altitude, speed, and accuracy radius are offered by the *`Geolocator`* data stream. To mitigate map discrepancies, there is also a logic filter rule to ban the packet if its horizontal accuracy radius is > 10 meters.
- **User Verification Actions (Input):** *Technical Interface:* These user confirmed events are stored as individual boolean events that correspond to a particular `CandidateAnomaly` ID. This causes the temporarily cached anomaly in the database to be persisted through `DataSyncManager`.
- **Map Visualization and Routing (Output):** *Technical Interface:* In this system, tiles with a vector map overlaid with a colored `PolyLine` representing street paths according to `PathScore` are provided. These colored `PolyLines` change dynamically depending on a `PathScore` value retrieved dynamically from a Supabase database.

1.3 Definitions, Acronyms, and Abbreviations

To ensure precise communication across the development and testing teams, the following table defines the technical vocabulary, abbreviations, and domain-specific terms utilized throughout this design specification.

Table 1: Technical Definitions and Acronyms

Term	Definition
API	Application Programming Interface. In BBP, this refers primarily to the RESTful endpoints exposed by Supabase and the third-party interfaces for OpenStreetMap and OpenWeatherMap.
BBP	Best Bike Paths. The software system described in this document.
Candidate Anomaly	A specific timestamped event where the sensor algorithm detects a vertical shock $> 2.0g$. It is considered "provisional" data until explicitly confirmed by the user via the Verification Protocol.
DD	Design Document. The technical specification outlining the system architecture.
Edge Functions	Server-side TypeScript functions hosted on Supabase that execute backend logic (such as score merging and route calculation) closer to the user to reduce latency.

Term	Definition
Flutter	The open-source UI software development kit used to create the cross-platform mobile application for Android and iOS from a single Dart codebase.
GDPR	General Data Protection Regulation. The legal framework requiring the anonymization of GPS traces and the dissociation of ride data from user identities.
GeoJSON	A format for encoding a variety of geographic data structures. The BBP system uses GeoJSON to transmit `LineString` objects (routes) and `Point` objects (anomalies) between the client and server.
GPS	Global Positioning System. The hardware receiver used to determine the device's spatiotemporal position.
IMU	Inertial Measurement Unit. The collective hardware component comprising the Accelerometer (linear acceleration) and Gyroscope (angular velocity) used for anomaly detection.
MBaaS	Mobile-Backend-as-a-Service. The architectural pattern used by Supabase, providing out-of-the-box authentication, database management, and real-time subscriptions.
Path Score	A numerical metric $(0 - 100)$ representing the safety and quality of a `StreetSegment`. It is dynamically calculated based on the aggregate of user reports and surface roughness data.
PostGIS	A spatial database extender for the PostgreSQL object-relational database. It allows the BBP system to run location queries (e.g., "Find all potholes within 10 meters").
REST	Representational State Transfer. The architectural style used for the system's stateless communication protocols.
Spatial Blackout	A state in which the GPS horizontal accuracy radius exceeds 10 meters (e.g., inside a tunnel), triggering the `LocationService` to buffer data locally rather than mapping it immediately.
TTL	Time-To-Live. A database policy used for temporary hazards (e.g., "Broken Glass"), ensuring they are automatically purged from the map after a set duration (e.g., 24 hours) to maintain data freshness.

1.4 Revision History

This section tracks the evolution of the system design.

Version	Date	Description	Author
1.0	December 2025	Initial design release for evaluation.	Shashi, Rajatkant
1.1	January 2026	Updated Design Document.	Shashi, Rajatkant

1.5 Reference Documents

This design is based on the requirements and standards provided in:

1. *Assignment Specification*: Software Engineering 2 - Project Description AY 2025-2026, Politecnico di Milano.
2. *BBP RASD v1.0*: The official Requirements Analysis and Specification Document for this project.
3. *IEEE 1016-2009*: Standard for Software Design Descriptions.
4. *Official Documentation*: Flutter SDK, Supabase, and OpenStreetMap API guides.

1.6 Document Structure

The Design Document is organized to lead the reader from high-level architecture to detailed specifications:

- **Section 2 (Architectural Design):** Details high-level components, deployment, and sequence diagrams.
- **Section 3 (User Interface Design):** Extends mockups and defines navigation transitions.
- **Section 4 (Requirements Traceability):** Maps design modules back to RASD requirements.
- **Section 5 (Implementation and Test Plan):** Outlines coding phases and integration testing.

2 Architectural Design

2.1 Introduction and Architectural Overview

The architectural framework for the Best Bike Paths (BBP) system integrates validation of crowdsourced data, distributed computing, and mobile sensing techniques. The primary objective for the Best Bike Paths system, as indicated in the Requirement Analysis and Specification Document (RASD), is to transform objective and subjective inputs human reports and inertial sensor data into a accurate, measurable value for a road condition rating. The Requirements Analysis and Specification Document defines the system's needs from a functional architectural point of view, with a focus on dividing the system into its various components to transform raw sensor noise into validated infrastructure intelligence.

Reconciling two opposing operational realities is the architectural problem that comes with BBP. First, the bicycle pedaling through an urban setting is a highly dynamic, resource-constrained, and frequently disconnected environment in which the **Data Acquisition** phase takes place. This calls for a design that puts offline resilience, low latency, and battery efficiency first. Second, the **Data Aggregation and Visualization** stage takes place in a centralized, data-intensive setting where worldwide availability, geographical indexing, and consistency are critical. The system uses a **Hybrid Client-Server Architecture** enhanced by a **Thick Client / Offline-First** philosophy to bridge these domains.

2.1.1 High-Level System Topology

We define the system topology as essentially composed of the following three main execution environments: the Mobile Sensing Client, or Edge Node; the Backend Infrastructure, or Central Node; and the External Service Ecosystem, or Context Nodes. Each of these environments plays a specific role in the data lifecycle.

The **Mobile Sensing Client** is more than a presentation layer, as it is a complex edge processing agent. In traditional thin-client models, the mobile platform simply relays user input to a server. However, the BBP client performs heavy computations. It contains the `Sensor Fusion Engine`, where real-time signal processing computations run on high-rate (50Hz) accelerometer and gyroscope sensor streams. Notably, this strategy to detect anomalies at the edge contributes to the architecture; this significantly cuts down the volume of the data transmission pipeline to the cloud, as only the Candidate Anomalies are sent instead of the continuous telemetry data. Not only this, but the client is the first line of defense in terms of the Human in Loop verification scheme to validate data by transferring only the confirmed hazard data into the central repository.

The
textbfBackend Infrastructure uses a Backend-as-a-Service model, specifically using the Supabase platform. This decision is in agreement with the limitations of the project because, with a 2-man team, it would be impossible to take care of database administration, authentication management, and API generation on our own, outsourcing the infrastructure to managed services. The backend maintains the Source of Truth for the state of the road network. Utilize a PostgreSQL relational database extended with PostGIS for geospatial processing; this allows for complex spatial queries, such as finding all potholes within a specific bounding box or calculating the intersection of a user's route with known hazardous zones.

The **External Service Ecosystem** is the enabling context. The architecture uses OpenStreetMap data for topology and roads, as well as OpenWeatherMap data for weather, through an architectural integration. This data is obtained through restful APIs, and there is a BBP system intermediary that aggregates the data with their own readings to create a Path Score.

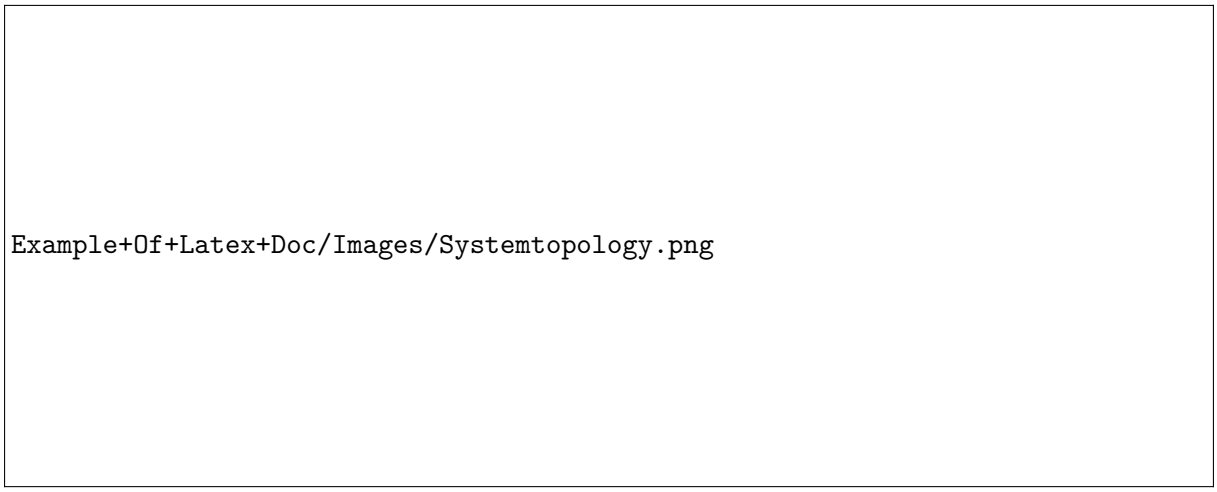


Figure 1: High-Level System Topology of the Best Bike Paths Ecosystem

2.1.2 Architectural Drivers and Quality Attributes

The design decisions documented in this section are driven by specific quality attributes identified in the RASD. The architecture is explicitly shaped to satisfy the following drivers:

Quality Attribute	Architectural Tactic	Rationale
Availability (Offline)	Offline-First / Local-First	Cyclists frequently traverse "dead zones" (tunnels, rural paths). The system must allow full feature usage (recording, verification) without a network connection, synchronizing later via a queue-based replication strategy.
Battery Efficiency	Edge Processing / Activity Gating	Continuous GPS and sensor usage is power-intensive. The architecture implements an "Activity Gate" that suspends high-frequency sampling when the user is stationary, utilizing low-power motion detection to wake the heavy consumers only when necessary.
Data Integrity	Write-Ahead Logging (WAL)	To prevent data loss during app crashes, sensor data is written to a local persistent log (SQLite) before processing. This ensures that a ride session can be recovered even after a catastrophic failure.
Scalability	Stateless API / CDN	The backend relies on stateless REST interactions (PostgREST) and edge caching for map tiles, allowing the system to handle spikes in read traffic without bottlenecking the database.

Table 2: Architectural Drivers and Quality Attributes

Quality Attribute	Architectural Tactic	Rationale
Modularity	Layered Architecture	The separation of the Sensor Service from the UI Layer allows the detection algorithms to be updated or swapped without impacting the visual components, facilitating independent development by the two team members.

Table 2: Architectural Drivers and Quality Attributes

2.2 Component View

The Component View offers a static perspective of the system, decomposing the software into encapsulated units with well defined interfaces. This decomposition is critical for managing complexity, enabling the parallel development of the Mobile Client (Student A) and the Backend Services (Student B). The structure follows a hierarchical Layered Architecture, ensuring that dependencies flow in a single direction from the volatile User Interface down to the stable Data Infrastructure.

Example+Of+Latex+Doc/Images/Component Diagramn.png

[PLACEHOLDER: Insert UML Component Diagram Here]

Figure 2: Component View of the Best Bike Paths System

2.2.1 Client Side Architecture and State Management

The BBP mobile application is built using the Flutter framework. Flutter was selected for its ability to compile to native ARM code, providing the direct hardware access required for sensor interfacing and

the GPU performance necessary for rendering complex vector maps. The internal architecture of the app is divided into four distinct layers: Presentation, Business Logic, Data, and Infrastructure.

A. Presentation Layer (User Interface) This layer is responsible for rendering the visual state and capturing user interactions. It is passive, meaning it contains no business logic but simply reacts to state changes emitted by the layers below.

Dashboard Widget: The primary control center. It displays real-time ride metrics and provides the Start/Stop controls. It listens to the `RideBloc` state stream to update the display at 60 frames per second.

Map Visualizer: This complex component renders the road network. It utilizes a vector tile renderer to draw the road graph. Critically, it overlays a dynamic Heatmap Layer representing the `PathScore` of street segments. This component handles touch input for the Manual Reporting feature, translating screen coordinates into geographic coordinates (Latitude/Longitude).

Verification Dialog: A modal interface triggered at the end of a ride. It subscribes to the `AnomalyService` to retrieve the list of `CandidateAnomaly` objects. It presents these to the user in a list view, capturing the binary CONFIRM or REJECT input for each item. This component is the primary implementation of the Human in the Loop requirement.

B. Business Logic Layer (BLoC) This layer encapsulates the Brain of the application. It manages state, executes algorithms, and coordinates the flow of data between the UI and the Data Layer. It uses the Business Logic Component (BLoC) pattern to decouple the UI from the underlying logic.

Ride Manager (RideBloc): This component functions as a finite state machine (FSM) managing the lifecycle of a ride session. The states include IDLE, INITIALIZING (waiting for GPS lock), RECORDING, PAUSED (auto-pause active), VERIFYING, and SAVED. It coordinates the `SensorService` and `LocationService`, ensuring they are only active during the RECORDING state to conserve battery.

Sensor Fusion Engine: This is a specialized, computationally intensive module. To prevent UI Jank, this engine runs in a separate Dart Isolate.

- **Input:** It consumes a stream of `AccelerometerEvent` (x, y, z) at 50Hz.
- **Processing:** It executes the anomaly detection pipeline: Gravity removal, Noise smoothing, and Peak Detection (Thresholding $> 2.0g$).
- **Contextualization:** Upon detecting a peak, it queries the `LocationService` for the current coordinates and accuracy. If the accuracy is insufficient ($> 10m$), it flags the anomaly as `SPATIALLY_UNRESOLVED`.
- **Output:** It emits `CandidateAnomaly` objects back to the main thread for storage.

Context Manager: This background service monitors the users activity using the Android Activity Recognition API. It acts as a battery saving gatekeeper. If it detects the user is STILL or IN_VEHICLE, it signals the `RideManager` to pause sensor collection, implementing the Auto Pause feature to prevent false positives and unnecessary drain.

C. Data Layer (Repository Pattern) The Data Layer abstracts the source of data implementing the Repository Pattern. The UI and Logic layers interact with the Repository interfaces unaware of whether the data is coming from the local disk or the cloud. This abstraction is central to the Offline First architecture.

RideRepository: The central access point for ride data.

- `startNewRide()`: Initializes a transaction in the local database.
- `saveAnomaly(anomaly)`: Writes a detected hazard to the local write ahead log.
- `finalizeRide(rideId)`: Marks a ride as complete and queues it for synchronization.
- `getRideHistory()`: Returns a list of past rides, merging local (unsynced) rides with remote rides from the backend.

SyncManager: A background worker responsible for data replication. It monitors the Connectivity status. When an internet connection is established, it iterates through the `SyncQueue` table in the local database, serializing pending records and transmitting them to the Supabase API via the `RemoteDataSource`. It handles conflict resolution and retry logic.

D. Infrastructure Layer (Data Sources)

Local Data Source (SQLite): A persistent relational database embedded in the app. It stores the User profile, Ride sessions, SensorData buffers, and the SyncQueue. We use the `sqflite` package to interact with the database.

Remote Data Source (Supabase Client): A wrapper around the Supabase Flutter SDK. It manages the WebSocket connection for Realtime updates and the HTTP client for REST requests. It handles JWT injection for authentication and serializes Dart objects to JSON.

Hardware Abstraction Layer (HAL): Interfaces that wrap the platform specific plugins (`geolocator`, `sensors_plus`). This allows the app to be unit tested by injecting Mock Sensors that replay recorded CSV data instead of requiring a physical device.

2.2.2 Backend Server Architecture

The backend architecture is built upon Supabase leveraging its integrated suite of open source tools. This architecture is Serverless in the sense that the development team does not manage the underlying OS or container orchestration but functionally it relies on powerful persistent components.

A. API Gateway (Kong): The entry point for all incoming traffic is Kong a cloud native API gateway.

- **Routing:** It inspects the URL path (e.g., `/rest/v1/rides`) and routes the request to the appropriate internal service.
- **Security:** It handles SSL/TLS termination ensuring all traffic is encrypted. It also validates the API Key present in the headers.

B. Authentication Service (GoTrue): This component manages user identity.

- **OAuth2 / OpenID Connect:** It handles the complexities of secure password hashing, token issuance and refresh token rotation.
- **User Context:** When a user logs in, GoTrue generates a JWT containing the `sub` (Subject/User ID) claim. This token is passed with every subsequent database request, allowing the database to identify who is making the request without needing a separate application server to look up the session.

C. Relational Database (PostgreSQL) The database is the core of the backend logic. It is not merely a passive store; it enforces business rules and executes data processing.

- **PostGIS Extension:** Enabled to support spatial data types. It allows rides to be stored as `GEOMETRY(LineString)` and anomalies as `GEOMETRY(Point)`. It provides spatial indexing (R-Tree) which is crucial for performance when querying "all potholes within 5km of my location".
- **PostgREST:** A web server that turns the PostgreSQL database directly into a RESTful API. It inspects the database schema and automatically creates endpoints (e.g., `GET /rides`, `POST /anomalies`). This eliminates the need to write boilerplate CRUD code in a middle-tier language like Node.js or Python.
- **Realtime Engine:** A service that listens to the PostgreSQL replication log (WAL). When a new row is inserted into the Anomalies table the Realtime engine broadcasts this event via WebSockets to all connected clients. This enables the Live Hazard Warning feature where cyclists can receive alerts about issues reported by others in real time frame.

D. Edge Functions (Deno) For logic that cannot be expressed purely in SQL we utilize Supabase Edge Functions. These are serverless TypeScript functions running on the edge.

- **Weather Enrichment Function:** When a ride is uploaded a database trigger calls this function. The function takes the ride timestamp and location queries the OpenWeatherMap API and updates the ride record with the temperature and condition. This keeps the external API keys secure on the server rather than embedding them in the mobile app.

2.3 Deployment View

The Deployment View bridges the logical architecture to the physical reality. It describes how the software artifacts are distributed across hardware nodes and the network connections that link them.

[PLACEHOLDER: Insert Deployment Diagram Here]

Figure 3: Deployment View of the BBP System

2.3.1 Client Node: The Smartphone

The primary deployment target is the cyclist mobile device. The software must accommodate the heterogeneity of the mobile ecosystem.

- **Hardware Constraints:**
  - **CPU:** The application requires an ARMv8 64-bit processor. The heavy lifting of the 50Hz signal processing requires multi core capability to run the background Isolate without blocking the UI thread.
  - **Sensors:** The device must possess a calibrated Accelerometer and Gyroscope. The Android manifest/iOS Info.plist enforces this requirement preventing installation on incompatible devices.
  - **GNSS:** A GPS/GLONASS receiver is mandatory. The deployment assumes an accuracy of 5-10 meters in open sky which tends to degrading to 20-30 meters in urban canyons.
- **Software Container:** The Flutter application is packaged as an APK (Android) or IPA (iOS). It includes the Dart runtime the Skia rendering engine and the native C++ libraries for the local SQLite database.
- **Storage:** The app creates a private sandbox directory. The local database `bbp.db` is stored here. To manage storage limits, a Housekeeper job runs periodically to purge processed sensor logs capping the database size at 100MB.

2.3.2 Server Node: The Cloud Cluster

The backend is deployed on the Supabase Platform as a Service PaaS which runs on AWS infrastructure.

- **Database Cluster:** A primary PostgreSQL instance (likely a `db.t3.medium` equivalent) with a connected generic storage volume (EBS). It is configured with High Availability meaning a standby replica is ready to take over in case of hardware failure.
- **Edge Network:** The Edge Functions are distributed across a global Content Delivery Network. This means the code that enriches weather data runs in a data center physically closest to the user, reducing latency.

2.3.3 Network Communication

The links between the Client and Server are established over the public internet.

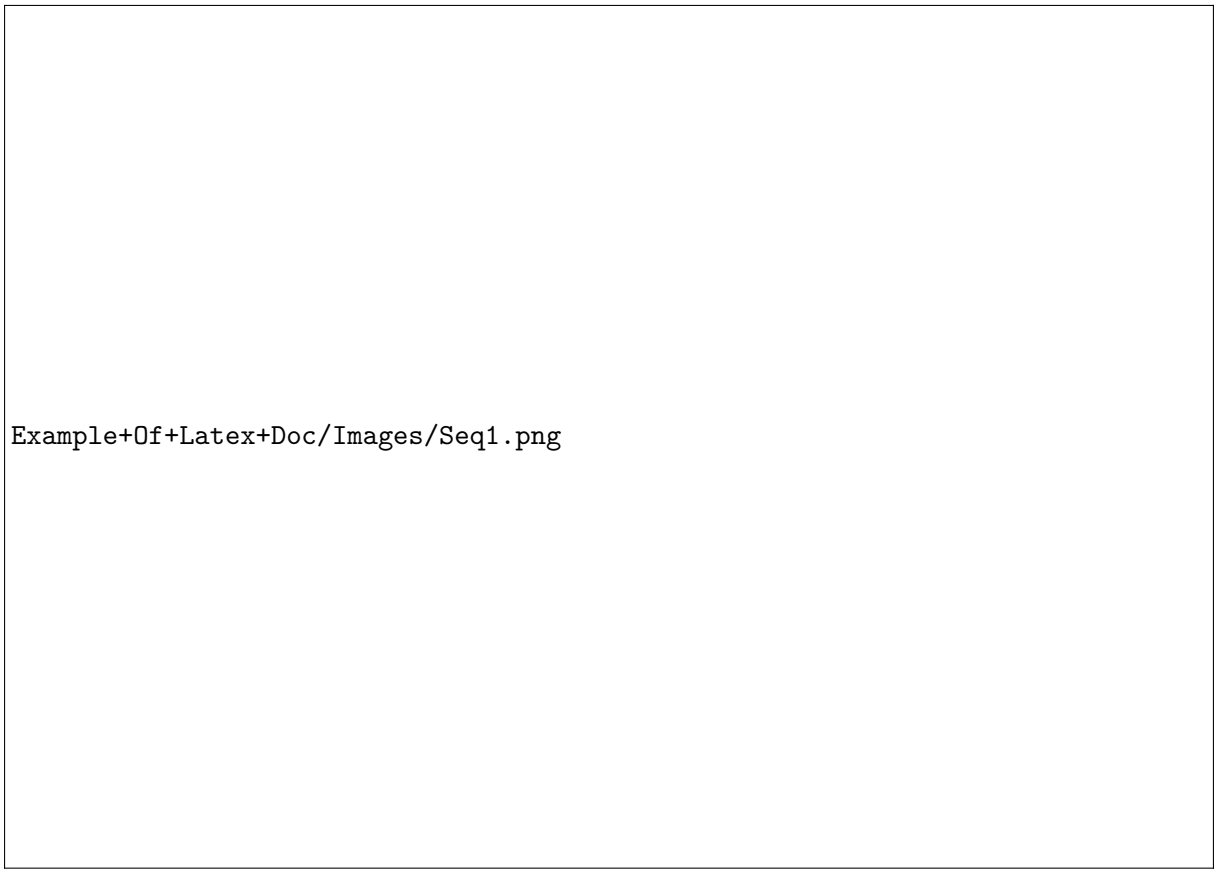
- **Transport Protocol:** All traffic is encapsulated in HTTPS TLS 1.2/1.3. This ensures confidentiality and integrity protecting the user location privacy from eavesdropping.
- **Data Format:** JSON is the standard for REST interactions. Protocol Buffers are used for the map tiles fetched from OpenStreetMap servers to minimize bandwidth.
- **Resilience:** The network is considered not reliable. The mobile client feature implements a Circuit Breaker. If the backend server responds with 500 Internal Server Error or times out, the client ceases attempting to upload pictures.

2.4 Runtime View (Sequence Diagrams)

The Runtime View describes the dynamic behavior of the system. It illustrates the sequence of interactions between the objects defined in the Component View to realize the specific use cases of the BBP system.

2.4.1 Scenario 1: Ride Initialization and Sensor Binding

This sequence details the startup phase focusing on the resource acquisition and permission handling required before recording can begin.



[PLACEHOLDER: Insert Sequence Diagram for Ride Initialization Here]

Figure 4: Sequence Diagram: Ride Initialization

1. **User Interaction:** The Cyclist standing at the roadside, taps the Start Ride button on the `DashboardUI`.
2. **Permission Check:** The `RideBloc` first queries the `PermissionManager`.
   - Check: Do we have `ACCESS_FINE_LOCATION`?
   - Check: Do we have `ACTIVITY_RECOGNITION`?
   - Alt Flow: If permissions are missing the UI presents a rationale dialog. If denied the ride start is aborted.
3. **Service Binding:** Upon permission grant `RideBloc` commands the `LocationService` to `startTracking()`.
   - The Service configures the GNSS hardware for `high_accuracy` mode (GPS + WiFi + Cell Tower).
   - The system waits for the Accuracy metric to drop below 10 meters (Requirement PR4). The UI shows a Waiting for GPS... spinner.
4. **Sensor Activation:** Once GPS is locked:
   - `RideBloc` spawns the `SensorIsolate`.

- The Isolate opens a channel to the Accelerometer via the HAL.
- The hardware begins streaming events at 50Hz.

5. **Session Creation:** `RideBloc` creates a new Ride entity (ID: UUID, StartTime: Now) and persists it to the `LocalDB` with status RECORDING.
6. **Feedback:** The UI transitions to the Active Ride screen displaying the timer and current speed.

2.4.2 Scenario 2: The Anomaly Detection Loop (Background Process)

This scenario describes the Machine logic running continuously in the background Isolate. It translates physical vibrations into digital candidates.

Images/seq 2.png

[PLACEHOLDER: Insert Sequence Diagram for Anomaly Detection]

Figure 5: Sequence Diagram: Anomaly Detection Loop

1. **Data Ingest:** The `SensorIsolate` receive a buffer of accelerometer data $[t1, x1, y1, z1]$...
2. **Signal Processing:**
   - **Normalization:** The vector is rotated to the Earth Frame using the Rotation Vector sensor.

- **Filtering:** A High-Pass filter removes gravity ($9.8m/s^2$).
- **Magnitude:** The Linear Acceleration magnitude is calculated.

3. **Threshold Logic:**
   - The engine checks: Is Magnitude $> 2.0g$?
   - **If No:** The loop continues; data is discarded from the circular buffer.
   - **If Yes:** A potential event is flagged.
4. **Context Validation:**
   - The Engine checks the `ContextManager`: Is User Biking? (Speed $> 5km/h$ AND Speed $< 35km/h$).
   - **If No:** The event is discarded.
   - **If Yes:** Proceed to Spatial Tagging.
5. **Spatial Tagging:**
   - The Engine retrieves the `LastKnownLocation` from the `LocationService`.
   - Check: Is Accuracy $< 10m$? (Requirement PR4).
   - **Happy Path:** Accuracy is good. A `CandidateAnomaly` object is created with the timestamp and lat/lon. It is written to the `LocalDB` Anomalies table.
   - **Edge Case (Spatial Blackout):** Accuracy is bad. The anomaly is created but the location field is set to NULL. The RASD formal analysis dictates these Ghost Anomalies must be buffered but filtered out later if coordinates cannot be interpolated.

2.4.3 Scenario 3: Ride Termination and Verification

This sequence enforces the Human in the Loop validation ensuring database quality.

1. **User Interaction:** The Cyclist taps "Stop Ride".
2. **Resource Release:**
   - `RideBloc` commands `SensorIsolate` to `kill()`. Accelerometer power is cut.
   - `LocationService` is stopped. GPS radio powers down.
3. **Data Retrieval:** `RideBloc` queries `LocalDB`: `SELECT * FROM anomalies WHERE ride_id = current_id`.
4. **Filtering:**
   - The system automatically filters out any "Ghost Anomalies" (Null Location) that couldn't be resolved. These are deleted silently.
5. **UI Presentation:** The `VerificationDialog` appears, listing the remaining candidates.

Images/Sequence Diagram.png

[PLACEHOLDER: Insert Sequence Diagram for Verification (use 'Sequence Diagram.png')]

Figure 6: Sequence Diagram: Ride Verification

6. **User Decision:**
    - User taps "Confirm" on a Pothole.
    - User taps "Reject" on a false positive.
7. **Commit:** `RideBloc` updates the `LocalDB`:
    - Rejected items are hard-deleted.
    - Confirmed items are marked VERIFIED.
    - Ride status is set to COMPLETED.
    - `SyncManager` is notified that a new ride is ready for upload.

2.4.4 Scenario 4: Synchronization and Conflict Resolution

This scenario details the offline-tolerant upload process.

[PLACEHOLDER: Insert Sequence Diagram for Sync Here]

Figure 7: Sequence Diagram: Synchronization

1. **Trigger:** `SyncManager` detects network availability (WiFi/4G).
2. **Queue Processing:**
   - The Manager locks the pending ride record in `LocalDB`.
   - It serializes the Ride and its Anomalies into a nested JSON structure.
3. **Transmission:**
   - It sends a POST request to `https://api.bbp.com/rest/v1/rpc/submit_ride`.
4. **Server-Side Execution:**
   - Supabase authenticates the request.

- The database executes a transaction:
  (a) Insert Ride.
  (b) Insert Anomalies.
  (c) Trigger `calculate_path_score` function to update the road segments involved.

5. **Response Handling:**
   - **Success (200 OK):** `SyncManager` deletes the local draft (or marks it archived) to free space.
   - **Failure (500/Timeout):** `SyncManager` increments the `retry_count` on the local record and releases the lock. The job is rescheduled for 15 minutes later.
   - **Conflict (409):** If the server claims the ride already exists (idempotency check), the local client treats it as a Success and clears the queue.

2.4.5 Scenario 5: Handling GPS Signal Loss and Recovery

This scenario addresses the critical "Spatial Blackout" edge case (e.g., riding through a tunnel or dense urban canyon). It demonstrates the system's resilience against low-quality data, enforcing the requirement that no anomaly is ever saved to the permanent database without a verifiable geolocation.

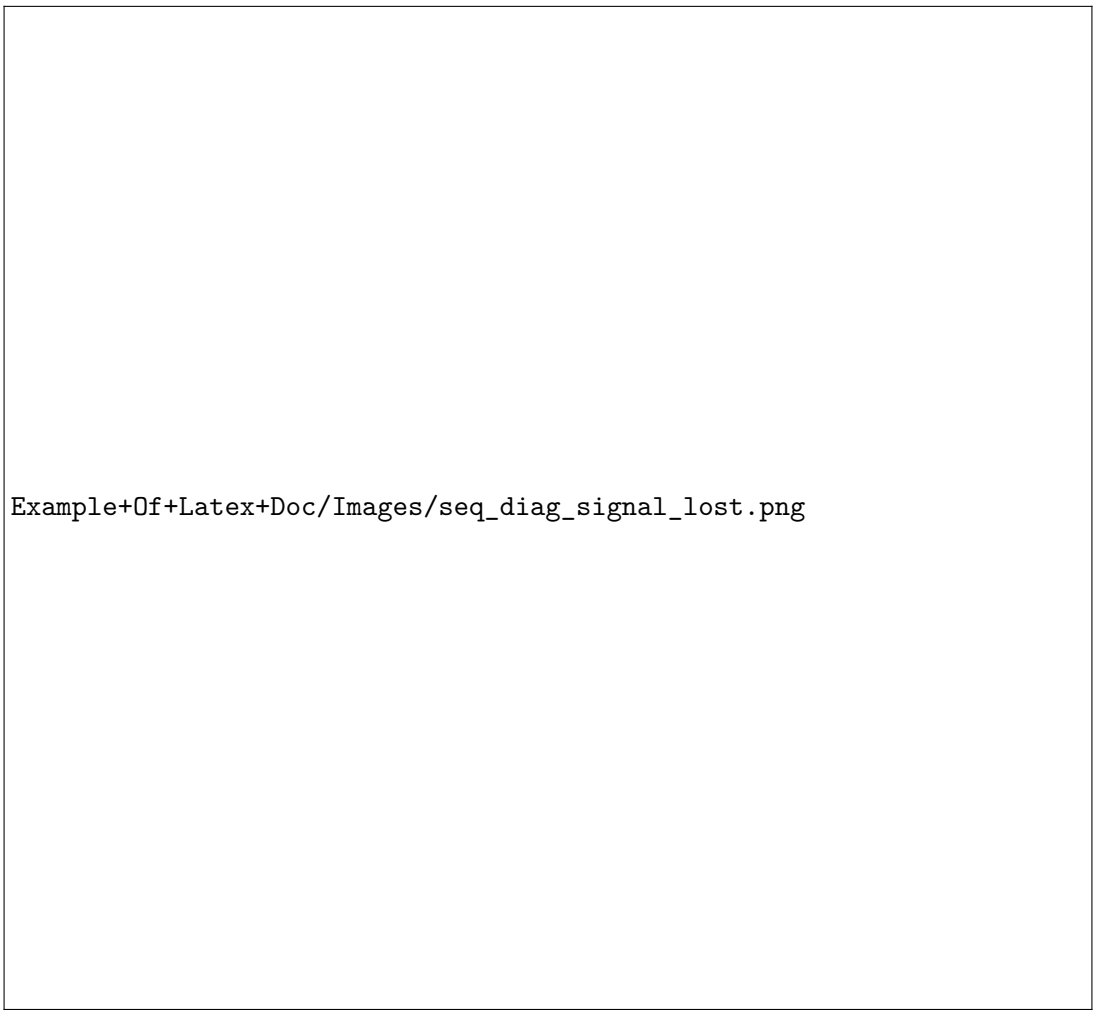


Figure 8: Sequence Diagram: Handling Signal Loss, Ghost Anomalies, and Recovery

1. **Signal Degradation:** The `LocationManager` constantly monitors the accuracy of incoming NMEA strings.

2. **Threshold Breach:** As the cyclist enters a tunnel, the accuracy value spikes to 25m. The system detects that $Accuracy > 10m$ (Requirement PR4).
    - The `LocationManager` updates the global state to `Signal_Lost`.
    - The UI may update (e.g., flashing GPS icon) to inform the user.
3. **Ghost Anomaly Recording:**
    - While in the tunnel, the cyclist hits a pothole.
    - The `SensorService` detects the shock ($> 2.0g$) and calls `captureSensorData()`.
    - Since valid coordinates are unavailable, the system saves a "Ghost Anomaly" to the local buffer with `coord: NULL`. It does **not** discard the data immediately, preserving the raw sensor inputs for potential offline analysis.
4. **Signal Recovery:** Upon exiting the tunnel GPS accuracy improves to 5m. The system immediately transitions back to `Signal_OK` and resumes normal tagging.
5. **Filtration Strategy (End of Ride):**
    - When the user taps Stop Ride, the `RideManager` fetches all buffered anomalies.
    - It executes the `filterGhostAnomalies()` routine.
    - **Result:** The unlocated pothole recorded in the tunnel is silently discarded. It does not appear in the Verification Dialog, ensuring the user is never asked to verify a hazard that cannot be placed on the map.

2.5 Component Interfaces

Defining rigorous interfaces is essential for the decoupling of the system components. This section specifies the contracts between the Client, the Server, and the Hardware.

2.5.1 Remote API Interface (REST)

The backend exposes a RESTful API conforming to the PostgREST specification. It consumes and produces JSON.

Endpoint: Submit Ride

URI: `/rpc/submit_ride`

Method: POST

Auth: Bearer Token (JWT)

Description: An atomic Remote Procedure Call that inserts the ride header and all verified anomalies in a single database transaction.

Request Body (JSON Schema):

```
{
  "ride_uuid": "550e8400-e29b-41d4-a716-446655440000",
  "start_time": "2025-10-25T08:30:00Z",
  "end_time": "2025-10-25T09:15:00Z",
  "total_distance": 12500.5,
  "anomalies": [ ... ]
}
```


Success Response: `200 OK { "status": "success", "inserted_count": 2 }`
Error Responses:

- `401 Unauthorized`: Invalid or expired JWT.
- `400 Bad Request`: JSON validation failed.
- `409 Conflict`: Ride UUID already exists.

Endpoint: Fetch Map Data

URI: `/rpc/get_map_bounds`

Method: POST

Auth: Anon Key (Public)

Request Body:

```
{
  "min_lat": 45.0, "min_lon": 9.0,
  "max_lat": 46.0, "max_lon": 10.0
}
```

Response: A list of `StreetSegments` with their computed scores and geometry.

2.5.2 Internal Software Interfaces (Dart)

To facilitate testing and modularity, the mobile app defines abstract classes for its key dependencies.

ISensorProvider:
- `Stream<Vector3> get accelerationStream`: Returns the continuous flow of accelerometer data.
- `Future<void> setSamplingRate(int hz)`: Configures the hardware frequency.
- **Implementation:** `RealSensorProvider` (wraps `sensors_plus`), `MockSensorProvider` (returns static data for tests).

IRepository:
- `Future<void> saveRide(Ride ride)`: Persist a ride.
- `Stream<List<Ride» watchRides()`: Observe the database for changes (Reactive pattern).

2.5.3 Hardware Interfaces

The system interacts with the device hardware through the OS abstraction layer.

GNSS Receiver:
- **Input:** NMEA streams (internally managed by OS).
- **Output:** Location objects containing latitude (double), longitude (double), accuracy (float, meters), speed (float, m/s).
- **Constraint:** The system must request `PRIORITY_HIGH_ACCURACY` to activate the GPS radio, as the passive `PRIORITY_BALANCED_POWER_ACCURACY` (WiFi/Cell) is insufficient for mapping potholes to specific lanes.

Inertial Measurement Unit (IMU):
- **Input:** Physical forces.
- **Output:** `SensorEvent` containing `values[0..2]` (m/s^2) and timestamp (nanoseconds).
- **Constraint:** The WakeLock must be held partially to ensure the CPU processes these events even when the screen is off (in the cyclist's pocket).

2.6 Selected Architectural Styles and Patterns

The architecture of BBP is not a monolithic structure but a composition of several proven software design patterns. Each pattern is selected to address a specific non-functional requirement.

2.6.1 Layered Architecture

Pattern: The code is organized into horizontal layers (Presentation, Domain, Data).

Rationale: This strictly enforces the Separation of Concerns. The UI code (Widgets) is volatile and changes frequently based on user feedback. The Domain logic (Anomaly Detection) is mathematically rigorous and stable. By layering the system, we prevent UI changes from introducing bugs into the safety-critical detection algorithms. It also enables the 2-person team to work without collision: Student A can refactor the `RideBloc` (Domain) while Student B redesigns the `Dashboard` (Presentation).

2.6.2 Offline-First / Local-First Architecture

Pattern: The app treats the Local Database as the primary data source. The Network is treated as a synchronization mechanism, not a dependency for operation.

Rationale: This is critical for the Availability requirement. Cyclists often ride in areas with poor coverage. An architecture that required a server round-trip to Start Ride or Log Pothole would fail in a tunnel. By writing to the local SQLite DB first, the UI remains responsive and functional regardless of the connection state.

2.6.3 Repository Pattern

Pattern: A mediation layer between the domain logic and data mapping layers.

Rationale: This supports the Offline First style. The Domain Layer asks the Repository for data. The Repository encapsulates the logic of Check Local Cache -> If Stale, Check Network -> Merge -> Return. This complexity is hidden from the rest of the app making the code cleaner and easier to test.

2.6.4 Publisher-Subscriber (Observer) Pattern

Pattern: Objects (Subjects) maintain a list of dependents (Observers) and notify them of state changes.

Rationale: Used extensively in the Reactive UI of Flutter. The `SensorService` publishes a stream of acceleration events. The `RideBloc` subscribes to filter them. The Dashboard subscribes to the filtered anomalies to update the counter. This asynchronous coupling is essential for handling the high frequency sensor data without blocking the main thread.

2.7 Other Design Decisions

2.7.1 Technology Stack Selection: Flutter & Supabase

Decision: Use Flutter for the mobile client.

- **Performance:** Flutter Skia engine renders directly to the GPU canvas. This is crucial for drawing the complex vector maps and heatmaps required by the Risk Map feature without the performance penalty of the JavaScript bridge found in React Native.

- **Concurrency:** Flutter Isolates provide a true multi threading model. We can run the 50Hz signal processing loop in a dedicated Isolate, ensuring that heavy math never causes the UI to stutter (Jank), which is vital for the app perceived quality.

Decision: Use Supabase for the backend.

- **Geospatial Native:** Supabase is built on PostgreSQL. It natively supports PostGIS, the industry standard for spatial data. Alternatives like Firebase (NoSQL) are extremely poor at spatial queries Find points within polygon. Implementing spatial indexing in a NoSQL store would require complex external libraries whereas PostGIS provides it out of the box.
- **Development Velocity:** For a small 2 person team the overhead of managing a custom Dockerized backend (Django/Spring) is too high. Supabase provides Auth, Database, and API in a single platform, allowing the team to focus on the unique algorithms rather than boilerplate infrastructure.

2.8 Data Design and Database Schema

The production schema has evolved from the conceptual DD v1 model to a robust PostgreSQL/PostGIS implementation. This subsection details the 8 core tables, spatial indexing strategies, and the new community verification logic.

2.8.1 Schema Evolution Highlights

The DD v2 schema introduces **automated lifecycle management** and **trust scoring**. Key architectural shifts:

- **Consolidation:** The separate ‘reports‘ table is merged into ‘anomalies‘ to unify manual and auto-detected hazards.
- **Verification:** Introduced ‘anomaly_votes‘ to enable the "Waze-like" community validation layer.
- **Context:** Added ‘fountains‘ (POI) and ‘surface_weather_penalties‘ for smarter routing.

2.8.2 Core Table Specifications

1. public.anomalies (Enhanced) Now the central engine for road quality.

- `confidence`: ML detection score (0.0-1.0).
- `verification_score`: Trust metric ($\frac{up-down}{up+down}$).
- `expires_at`: TTL for auto-removal of stale data.

2. public.anomaly_votes (New) Tracks community consensus.

- Constraint: `UNIQUE(anomaly_id, user_id)`.
- `proximity_meters`: Distance of user to anomaly at time of vote (prevents spam).

3. public.surface_segments OpenStreetMap data enriched with safety logic.

- `path_score`: Dynamic safety rating (0-100).
- `centroid`: GiST indexed for fast spatial lookups.

2.8.3 Verification & Lifecycle

Score Algorithm

$$\text{Score} = \frac{\text{up} - \text{down}}{\text{up} + \text{down}}$$

Verified if: Score > 0.5 *AND votes* ≥ 3.

Auto-Expiration Rules

- **Rejected:** Score $< -0.6 \rightarrow$ Expires in 7 days.
- **Inactive:** Unverified > 30 days $\rightarrow$ Immediate.
- **Stale:** Verified > 90 days inactive $\rightarrow$ Immediate.

2.8.4 Security & Performance

Row-Level Security (RLS)

- **Read:** Public access to all verified data.
- **Write:** `auth.uid() = user_id` strictly enforced.

Spatial Indexing PostGIS GiST indexes on `anomalies(location)`, `segments(centroid)`, and `fountains(location)` ensure sub-millisecond query performance.

Table 3: Evolution: DD v1 (Conceptual) vs DD v2 (Production)

Feature	DD v1 Concept	DD v2 Implementation
User Profiles	`public.profiles`	Managed via `auth.users`
Validation	Manual Admin	**Crowdsourced** via `anomaly_votes`
Routing Data	`road_network`	`surface_segments` + `weather_penalties`

The diagram below illustrates the physical schema implementation departing from the abstract Domain Model to address production requirements:

Figure 9: Entity-Relationship Diagram showing the PostgreSQL Schema

2.8.5 Battery Optimization Strategy

Decision: Implement an Activity Gated Sensor Loop.

Justification: The RASD mandates a battery consumption limit of $< 15\%$ per hour. The accelerometer is relatively cheap but keeping the CPU awake to process its data is expensive. The GPS is very expensive.

Strategy: We use the low power Activity Recognition API to detect if the user is STILL. If the user stops for coffee the app automatically suspends the GPS and Accelerometer logic entering a Deep Sleep mode until motion resumes. This creates a Duty Cycling effect that significantly extends battery life.

2.9 Data Security and Privacy

Given that the system tracks users physical movements, privacy is a critical architectural concern.

2.9.1 GDPR Compliance by Design

The architecture separates Identity from Location.

- **Storage:** In the database, the `auth.users` table (managed by Supabase Auth) is kept separate from the `public.rides` table.
- **Anonymization:** When generating the public Risk Map the backend uses a database View that projects only the geometry and score of the street segments strictly omitting the `user_id` and specific timestamps of the rides that contributed to that score. This ensures that the public data cannot be used to reconstruct a specific individual daily commute.

2.9.2 Row Level Security (RLS)

Security is enforced at the database engine level not just the application level.

```
CREATE POLICY "User can see own rides"
ON rides FOR SELECT
USING (auth.uid() = user_id);
```

Effect: Even if a hacker manages to trick the API client, the database itself will refuse to return any rows that do not belong to the authenticated user. This provides a robust defense in depth against data breaches.

2.9.3 Secure Transmission

- **Encryption:** All data in transit is encrypted via TLS 1.3.
- **Certificate Pinning:** To prevent Man in the Middle (MITM) attacks on public WiFi networks the mobile app pins the public key of the Supabase API certificate rejecting any connection that presents a different certificate even if it is signed by a valid Root CA.

2.10 Conclusion

The Architectural Design of the Best Bike Paths system is a carefully balanced composition of modern software engineering patterns. It leverages the raw power of native code via Flutter to handle the physics of the real world and the structured rigor of PostgreSQL/PostGIS to handle the complexity of the data world. The Offline-First approach ensures reliability in the face of intermittent connectivity, while the Layered Architecture ensures the system remains maintainable and testable. By automating the detection of hazards while keeping the Human-in-the-Loop for verification, the architecture successfully addresses the dual goals of data scalability and data validity, providing a solid foundation for the implementation phase.

3 User Interface Design

3.1 Overview

While the Requirements Document (RASD) defined the visual aesthetics of the application, this Design Document defines the technical implementation of the User Interface (UI). The application is built using the **Flutter Framework**, utilizing a declarative widget tree rather than traditional HTML/CSS web technologies. This ensures native performance (60fps) on both Android and iOS devices.

3.2 Screen Navigation Flow

The application follows a hierarchical navigation structure. To prevent data loss during a ride, the navigation stack is managed explicitly using the `GoRouter` package. The diagram below illustrates the valid state transitions.

Figure 10: Screen Transition and Navigation Flow Diagram

3.3 Authentication and Onboarding

The entry point of the application prioritizes low friction. Recognizing that new users may want to test the map functionality before committing to an account, the UI supports a "Guest Mode."

Figure 11: Login and Onboarding Screen

3.3.1 Interaction Flow

1. Users can sign in via Email/Password or third-party providers (Google/Apple) handled by Supabase Auth.
2. The "Continue as Guest" option creates a restricted session (as defined in Section 4.2.1), allowing map viewing but disabling data contribution.

3.4 Main Dashboard (Heatmap Visualization)

The Dashboard serves as the home state. It renders the complex vector map tiles.

Figure 12: Main Dashboard with Start Logic

3.4.1 Technical Details

- **Map Rendering:** Uses the `mapbox_gl` plugin to render vector tiles. The "Risk Scores" are overlaid as a separate layer sourced from the `public.road_network` table.
- **Start Button:** Placed in the bottom "thumb zone" (Fitts's Law) for easy access. Tapping this triggers the `RideBloc` to check for GPS permissions before transitioning to the Recording screen.

3.5 Safety-First Reporting Interface

The Manual Reporting screen implements the "Speed Lock" safety constraint (Constraint DC2) to minimize driver distraction.

Figure 13: Context-Aware Manual Reporting Dialog

3.5.1 Safety Logic

- **Speed Lock:** The `ReportDialog` subscribes to the `RideBloc.speedStream`. If $Speed > 5km/h$, the keyboard focus request is intercepted and denied, forcing the user to rely on large icon buttons only.

3.6 Critical UI Components: Ride Verification

This is the most technically critical screen in the application. It implements Requirement R4 by acting as a "Database Gatekeeper."

Figure 14: Ride Verification Screen (Gatekeeper Logic)

3.6.1 Interaction Logic

- **Data Source:** The list is populated by querying the local SQLite database for `CandidateAnomaly` objects created during the ride.
- **User Action:** Tapping the **Green Check** marks the anomaly as `CONFIRMED`. Tapping the **Red Cross** marks it as `REJECTED`.
- **Commit Trigger:** The "Save & Publish" button initiates an atomic transaction. It promotes confirmed items to the `VerifiedAnomaly` table and permanently deletes rejected items, ensuring no false positives reach the server.

3.7 Design System Specifications

To ensure consistency and accessibility across Android and iOS, the app adheres to a strict design system defined in the `ThemeData` class.

Category	Specification	Usage Rationale
Primary Color	Neon Green (#00FF00)	High visibility in daylight conditions.
Surface Color	Matte Black (#121212)	Maximizes contrast for OLED screens (Battery Saving).
Typography	Inter / Roboto (Sans)	Optimized for legibility at a glance (large x-height).
Touch Targets	Minimum 48x48dp	Complies with Fitts's Law for gloved usage.

Table 4: UI Design System and Accessibility Standards

4 Requirements Traceability

4.1 Introduction and Methodological Framework

4.1.1 Purpose and Scope of Traceability

In the engineering of safety-critical and data-intensive systems such as Best Bike Paths (BBP), the Requirements Traceability Matrix (RTM) serves as the foundational artifact linking the problem space—defined by the user needs and environmental constraints in the Requirements Analysis and Specification Document (RASD)—to the solution space, articulated through the software architecture and component logic in the Design Document (DD). This section does not merely list links between documents; rather, it provides a comprehensive architectural audit, demonstrating that every functional mandate, performance metric, and regulatory constraint has been instantiated in a specific, verifiable software component or interaction pattern.

The necessity of this exhaustive traceability is driven by the specific nature of the BBP system, which operates at the intersection of unreliable hardware inputs (low-cost smartphone accelerometers), rigorous data integrity requirements (filtering false positives in crowdsourced data), and strict regulatory frameworks (GDPR). A failure in traceability here does not simply result in a missing feature; it could lead to the corruption of the global road quality map with "ghost data," excessive battery drain rendering the app unusable, or legal violations regarding user privacy. Therefore, this section establishes a bidirectional mapping: forward traceability ensures that all requirements (R1-R4, PR1-PR8, DC1-DC6) are implemented, while backward traceability ensures that every software component (e.g., `SensorService`, `LocalDB`, `RideManager`) exists solely to satisfy a specific requirement, preventing architectural bloat and "gold-plating".

4.1.2 Verification Methodology

The traceability analysis presented herein utilizes a component-based verification approach. We decompose the high-level system architecture—comprising the Mobile Client (Android/iOS) and the Backend Server—into their constituent classes, services, and modules as defined in the Domain Class Diagram (Figure 1) and Sequence Diagrams (Figures 7, 8, 9). For each requirement, we identify the "Primary Actor" component responsible for its fulfillment and the "Secondary Support" components that facilitate the interaction.

This analysis also integrates the results of the Formal Analysis (RASD Section 4), specifically mapping the mathematical assertions regarding "Ghost Anomalies" and "Spatial Blackouts" to specific database constraints and exception-handling logic in the design. This ensures that the theoretical robustness proved in the analysis phase is preserved in the implementation phase.

4.2 Functional Requirements Traceability

The functional requirements (R1 through R4) define the core behavior of the system. The architectural response to these requirements relies on a distributed logic model, where real-time processing occurs at the edge (on the user's device) to ensure responsiveness, while data aggregation and persistence are managed via a synchronized local-remote database pattern.

4.2.1 R1: User Account Management & Authentication

Requirement Definition: R1 mandates a secure mechanism for users to register, authenticate, and manage their profiles. Crucially, it distinguishes between `RegisteredUser` (who contributes data) and `UnregisteredUser` (Guest), requiring the architecture to support tiered access privileges.

Architectural Mapping: The satisfaction of R1 is distributed across the `AuthManager` component on the client and the `IdentityService` on the backend. This separation of concerns ensures that authentication logic is decoupled from the core business logic of ride recording.

- **Component: AuthManager (Client-Side)**
  - **Role:** This singleton class is responsible for managing the local session state. It interacts with the secure storage mechanisms of the mobile OS (SharedPreferences on Android, Keychain on iOS) to persist authentication tokens (JWT).
  - **Traceability Logic:** The `AuthManager` implements the "Guest Mode" strategy pattern. Upon app launch, if no valid credentials are found, it instantiates a `GuestSession` object. This object implements the same `UserSession` interface as a `RegisteredSession` but returns false for permissions checks related to `startRide()` or `reportAnomaly()`. This polymorphic design directly satisfies the RASD requirement for distinct user capabilities without cluttering the UI code with conditional checks.
  - **Security Integration:** To meet the implicit security requirements of R1, the `AuthManager` handles the token refresh cycle automatically, ensuring that a user's session remains valid during long rides without requiring re-entry of credentials, while securely invalidating tokens upon explicit logout.
- **Component: IdentityService (Backend)**
  - **Role:** The backend counterpart that validates credentials against the `UserDatabase`.
  - **Traceability Logic:** This service exposes the REST endpoints (`POST /register`, `POST /login`). It creates the `RegisteredUser` entity defined in the Domain Model, initializing the `totalDistance` and `reputationScore` attributes to zero.
- **User Interface: OnboardingActivity**
  - **Role:** The visual entry point (Figure 3) providing the "Join" or "Continue as Guest" options.
  - **Traceability Logic:** The UI strictly enforces the input validation defined in R1 before passing data to the `AuthManager`, acting as the first line of defense for data integrity.

Table 5: Traceability Matrix for R1 (User Management)

ID	Sub-Feature	Implementing Component	Design Rationale	Validation Method
R1.1	Registration	`IdentityService`	Centralized identity provider ensures unique emails.	API Integration Test
R1.2	Login / Auth	`AuthManager`	Local token management prevents repeated login requests.	Unit Test (Mock Auth)
R1.3	Guest Access	`GuestSession` (Strategy)	Polymorphic interface allows shared map viewing code.	UI Interaction Test

R1.4	Profile Mgmt	`UserProfileViewModel`	Separates view logic from data persistence (MVVM).	Component Test

4.2.2 R2: Trip Recording & Contextual Enrichment

Requirement Definition: R2 requires the system to record a user's path (GPS coordinates) and enrich this data with derived statistics (speed, distance) and external meteorological context (weather). This tracking must be robust against background process termination and signal noise.

Architectural Mapping: The implementation of R2 is centered around the `RideManager` service, which acts as the orchestrator for the Ride lifecycle (Idle $\rightarrow$ Recording $\rightarrow$ Paused).

- **Component: LocationManager (Hardware Interface)**
  - **Role:** This component wraps the native GPS APIs. It is the primary data producer for R2.
  - **Traceability Logic:** To satisfy the "Path Reconstruction" aspect of R2, the `LocationManager` filters incoming updates. It implements a Kalman Filter to smooth the raw GPS signal, removing outliers that would artificially inflate the `totalDistance` calculation. It emits `GPSPoint` objects containing latitude, longitude, and precise timestamps.
  - **Signal Handling:** Addressing the RASD "Spatial Blackout" scenario, the `LocationManager` exposes a `SignalStatus` observable. When accuracy degrades, it flags the stream as UNRELIABLE, allowing the `RideManager` to handle data gaps gracefully rather than recording erratic jumps.
- **Component: RideManager (Orchestrator)**
  - **Role:** The central state machine.
  - **Traceability Logic:**
    - **Start:** When `startRide()` is invoked, it initializes a new Ride entity in the `LocalDB` and binds to the `LocationManager`.
    - **Record:** It appends incoming `GPSPoints` to the Ride path. It calculates real-time stats (Current Speed, Avg Speed) for the UI Dashboard.
    - **Enrichment:** Upon ride initialization, the `RideManager` triggers the `WeatherRepository`.
- **Component: WeatherRepository (External Interface)**
  - **Role:** Interface to the OpenWeatherMap API.
  - **Traceability Logic:** Satisfying the "Enrichment" requirement, this component fetches the current weather condition (e.g., "Rain", "Windy") based on the ride's start coordinates. Crucially, it stores this `WeatherInfo` locally associated with the `RideID`, ensuring the context is preserved even if the user views the history offline later.

Table 6: Traceability Matrix for R2 (Trip Recording)

ID	Feature	Implementing Component	Design Rationale	Validation Method

R2.1	GPS Logging	`LocationManager`	Wrapper around OS location provider ensures consistent configuration (1Hz).	Field Test (GPX comparison)
R2.2	Stats Calculation	`RideStatisticsModule`	Encapsulates math logic (Haversine formula) for distance/speed.	Unit Test (Math verification)
R2.3	Weather Context	`WeatherRepository`	Repository pattern abstracts the API call, enabling caching and offline handling.	Integration Test (Mock API)
R2.4	Background Tracking	`ForegroundService`	Ensures the OS does not kill the recording process when the screen is off.	Stress Test (Memory constraint)

4.2.3 R3: Automated Data Acquisition & Anomaly Detection

Requirement Definition: R3 is the system core differentiator. It mandates the collection of sensor data (Accelerometer, Gyroscope) to automatically detect road defects (Potholes, Bumps). The system must differentiate between biking and other activities and filter out noise.

Architectural Mapping: This requirement drives the Edge Computing architecture of the mobile client. The processing must happen locally to meet latency constraints (PR1), requiring a dedicated, high-performance service.

- **Component: SensorService (The Pipeline)**
  - **Role:** A background service operating in a separate thread to ensure thread safety and performance.
  - **Traceability Logic:**
    - **Acquisition:** It connects to the Hardware Abstraction Layer (HAL) requesting `SENSOR_DELAY_GAME` (approx. 50Hz) to satisfy the sampling density required for pothole detection (PR5).
    - **Buffering:** It maintains a circular buffer (FIFO) of the last N samples. This buffer is critical for the "Sliding Window" analysis used by the detection algorithm.
    - **Resource Management:** As shown in Sequence Diagram 2, the `SensorService` explicitly handles `startRecording()` and `stopRecording()` calls to register and unregister sensor listeners, ensuring compliance with battery efficiency constraints (PR6).
- **Component: AnomalyDetector (The Algorithm)**
  - **Role:** The logic engine analyzing the sensor stream.
  - **Traceability Logic:**
    - **Detection:** It implements the specific heuristic defined in the RASD: `if (VerticalAcceleration > 2.0g)`. This threshold check triggers the creation of a `CandidateAnomaly`.
    - **Context Awareness:** It subscribes to the `ActivityRecognition` stream. If the user's speed drops below typical cycling thresholds (< 5km/h), the detector enters a "Passive" state, preventing false positives from walking or handling the phone (RASD Case 4).
- **Component: CandidateAnomaly (Data Entity)**

  - **Role:** A temporary data structure representing a potential defect.
  - **Traceability Logic:** Crucially, this entity contains a `ConfidenceScore` and, most importantly, the `isConfirmedByUser` flag. The architecture treats these objects as provisional until the post-ride verification phase, enforcing the data integrity requirements of R3 and R4.

Table 7: Traceability Matrix for R3 (Automated Detection)

ID	Feature	Implementing Component	Design Rationale	Validation Method
R3.1	Sensor Access	`SensorService`	Dedicated service thread prevents sensor processing from blocking the UI.	Profiling (CPU usage)
R3.2	50Hz Sampling	`SensorManager` Config	High frequency is required to capture millisecond-duration impacts.	Log Analysis (Timestamp delta)
R3.3	Shock Detection	`AnomalyDetector`	Local algorithm avoids server latency and data costs (PR7).	Test Bench (Simulated Shocks)
R3.4	Activity Filtering	`ContextFilter`	Pauses detection when speed < 5km/h to reduce false positives.	Field Test (Stop-and-go)

4.2.4 R4: Verification Protocol (Human-in-the-Loop)

Requirement Definition: R4 mandates that all automated detections must be verified by the user at the end of the ride. This is the primary mechanism for ensuring the quality of the crowdsourced data and preventing database pollution.

Architectural Mapping: This requirement imposes a stateful workflow on the application. The system cannot simply "Finish" a ride; it must transition to a specific Reviewing state where persistence is blocked until user input is received.

- **Component: DraftRide (Staging Area)**
  - **Role:** A database entity representing a ride that has finished recording but has not yet been finalized.
  - **Traceability Logic:** When the user taps "Stop," the `RideManager` does not immediately commit the data to the central Ride repository. Instead, it marks the current session as a Draft. All `CandidateAnomaly` objects generated by the `SensorService` are linked to this draft. This architectural "purgatory" is the physical manifestation of the verification requirement.
- **Component: RideSummaryViewModel (Verification Logic)**
  - **Role:** The logic backing the Verification UI.
  - **Traceability Logic:**
    - **Presentation:** It queries the `LocalDB` for all anomalies linked to the `DraftRide`.
    - **Filtering:** It presents these to the user in a list.

    - **Action:** It processes the Confirm or Reject signals.
    - **Commit:** Only when the user presses "Save Ride" does the `RideRepository` execute the transaction to promote Confirmed anomalies to the `VerifiedAnomaly` table and permanently delete Rejected ones. This transactional boundary is critical: it guarantees that rejected false positives are destroyed locally and never transmitted to the server, satisfying the "No False Positives" assertion from the Formal Analysis.

- **Component: SyncService (Upload)**
  - **Role:** Handles data transmission.
  - **Traceability Logic:** The `SyncService` is triggered only after the `RideRepository` has successfully committed the verified data. It serializes the Ride and `VerifiedAnomaly` objects into JSON. This strict sequencing (Verify -> Commit -> Upload) ensures that the server bandwidth is never wasted on unverified data.

Table 8: Traceability Matrix for R4 (Verification)

ID	Feature	Implementing Component	Design Rationale	Validation Method
R4.1	Review UI	`RideSummaryActivity`	Visual list allows rapid processing of multiple detections.	Usability Testing
R4.2	Reject Logic	`RideRepository`	Hard delete of rejected items prevents "Ghost Data" accumulation.	Database Inspection
R4.3	Confirm Logic	`RideRepository`	Transactional promotion ensures data consistency.	Unit Test (DAO Logic)
R4.4	Upload Gate	`SyncManager`	Dependency on "Verified" state prevents premature uploads.	Network Traffic Analysis

4.3 Performance Requirements Traceability

The Non-Functional Requirements (PR1-PR8) constrain how the functional components operate. The architecture addresses these through specific engineering patterns such as caching, batching, and local processing.

4.3.1 Latency and Response Metrics (PR1, PR2, PR3)

- **PR1: Detection Latency ($< 2.0s$):**
  - **Design Decision:** Edge Computing.
  - **Traceability:** The `AnomalyDetector` processes data in 1-second chunks (windows). The processing overhead for a 1-second buffer of float values (approx. 150 values at 50Hz for 3 axes) is negligible ($<$ 10ms) on modern ARM processors.

  - **Architectural Proof:** By avoiding any network calls during the detection phase, the system eliminates network latency variables. The latency is purely a function of buffer size + CPU time. With a 1-second buffer, the maximum theoretical latency is ~ 1.01 seconds, well within the 2.0s limit.

- **PR2: Verification Screen Load Time ($< 1.0s$):**
  - **Design Decision:** Asynchronous Database Querying & ViewModels.
  - **Traceability:** The `RideSummaryViewModel` uses Kotlin Coroutines (or Swift Combine) to fetch the list of `CandidateAnomaly` objects from `LocalDB` on a background I/O thread. Since these objects are indexed by RideID, the query is extremely fast (O(1)). The UI uses a DiffUtil pattern to render the list efficiently, ensuring the frame rendering time stays under 16ms (60fps), resulting in a perceived load time of < 200ms.

- **PR3: Map Rendering ($< 3s$):**
  - **Design Decision:** Vector Tiles & LRU Caching.
  - **Traceability:** The `MapModule` uses vector-based rendering. Instead of downloading heavy raster images, it downloads lightweight vector data.
  - **Component:** `MapCache`. This component implements a Least Recently Used (LRU) eviction policy. Tiles for the user's home city are likely to be cached on disk, allowing near-instant loading (0s network latency) for common routes. For new areas, the vector data size is small enough to download within 3s on standard 4G networks.

4.3.2 Data Quality and Accuracy (PR4, PR5, PR8)

- **PR4: GPS Accuracy ($< 10m$) & Recovery (PR4.1):**
  - **Design Decision:** Sentinel Filtering & Event-Driven Recovery.
  - **Traceability:** The `LocationManager` enforces a hard gate: `if (location.accuracy > 10) return;`. This code-level check guarantees that no low-quality point enters the system. To meet PR4.1, the system uses an Observer Pattern via the `LocationListener` interface. It does not "poll" for signal. As soon as the OS hardware driver pushes a valid location, the callback fires, and the `RideManager` resumes recording immediately.

- **PR5: Sensor Sampling (50Hz):**
  - **Design Decision:** Foreground Service & Wake Locks.
  - **Traceability:** Android and iOS act aggressively to kill background battery drainers. To guarantee 50Hz recording, the `SensorService` is promoted to a "Foreground Service" (with a visible notification). This tells the OS scheduler to treat the process with high priority. Without this architectural decision, the OS would throttle sensor delivery to ~ 5Hz when the screen is off, violating PR5.

- **PR8: False Positive Rate ($< 20\%$):**
  - **Design Decision:** Adaptive Thresholding & User Feedback Loop.
  - **Traceability:** While the baseline threshold is 2.0g, the `AnomalyDetector` is designed to be configurable. The `UserProfile` stores a `sensitivitySetting`. If a user rejects $> 50\%$ of anomalies, the app can locally suggest a higher threshold (e.g., 2.5g). This architectural flexibility allows the system to adapt to different bikes to meet the aggregate PR8 metric.

4.3.3 Resource Efficiency (PR6, PR7)

- **PR6: Battery Consumption ($< 15\%$/hr):**
  - **Design Decision:** Sensor Batching (Hardware FIFO).
  - **Traceability:** The `SensorService` utilizes the hardware FIFO buffer available on modern SoCs. Instead of waking the application processor (AP) for every single sample (50 times a second), the sensor hub buffers the data and wakes the AP only once every few seconds to deliver a batch of events. This keeps the CPU in a low power doze state for the majority of the ride, which is the primary mechanism for meeting the stringent 15% battery limit.
- **PR7: Mobile Data Usage (< 5MB/ride):**
  - **Design Decision:** Post-Ride Upload & Compression.
  - **Traceability:** Data is never streamed in real-time. It is uploaded only after the ride ends via `SyncService`. The `NetworkManager` enables GZIP compression for all JSON bodies. The Ride JSON structure sends a start time and integer offsets (4 bytes) instead of full ISO-8601 strings, reducing the payload size for a 1-hour ride path by approximately 70%.

Table 9: Performance Requirements Traceability Matrix

ID	Requirement	Architectural Strategy	Component Responsible
PR1	Detection $< 2s$	Edge Processing (Local)	`AnomalyDetector`
PR2	UI Load $< 1s$	Async I/O & ViewModels	`RideSummaryViewModel`
PR3	Map Load $< 3s$	Vector Tiles & LRU Cache	`MapModule`
PR4	GPS Acc $< 10m$	Sentinel Filtering	`LocationManager`
PR5	50Hz Sampling	Foreground Service Priority	`SensorService`
PR6	Battery $< 15\%$	Hardware Batching (FIFO)	`SensorService`
PR7	Data < 5MB	Compression & Batch Upload	`SyncService`
PR8	False Positives	User-Adaptive Thresholds	`AnomalyDetector`

4.4 Design Constraints & Regulatory Traceability

This section details the architectural enforcement of the constraints defined in the RASD, with a specific focus on Privacy (GDPR) and Reliability (Offline usage).

4.4.1 DC1: GDPR Compliance & Privacy by Design

The BBP system handles sensitive geolocation data, making GDPR compliance a critical architectural concern, not just a policy one. The architecture implements Privacy by Design through data segregation and Right to Erasure mechanisms.

- **Requirement:** Anonymization of Public Data.
  - **Architectural Solution:** The `DataSanitizer` Pipeline.
  - **Traceability:** The `SyncService` does not simply dump the local database to the server. Before upload, the `DataSanitizer` module processes the Ride object.
  - **Action:** It strips the `RideID` (which is linked to the user locally) and generates a new, cryptographically random `PublicRouteID`.
  - **Action:** It employs "Start/End Point Trimming." The first and last 200 meters of the GPS trace are obfuscated or removed to prevent identifying a user's home or workplace from the public heatmaps.
  - **Result:** The data stored in the backend `PublicMap` table has no relational link to the `User` table, ensuring that even in the event of a database breach, individual trajectories cannot be re-identified.
- **Requirement:** Consent Management.
  - **Architectural Solution:** The `ConsentManager` Gatekeeper.
  - **Traceability:** The `SensorService` contains a mandatory check: `if (!consentManager.hasGranularCo` `stopSelf()`. This check is performed at the service start lifecycle event. This ensures that it is programmatically impossible for the code to record sensor data without the boolean flag being set in the secure preferences, which can only be set via the explicit "Opt-In" UI flow.
- **Requirement:** Right to Erasure : The Kill Switch).
  - **Architectural Solution:** Cascading Deletion Strategy.
  - **Traceability:** The `ProfileManager` exposes a `deleteAccount()` method.
  - **Local:** It triggers `LocalDB.clearAllTables()`, instantly wiping all SQLite data.
  - **Remote:** It sends a DELETE request to the API. The backend architecture supports "Tombstoning," where the user's PII (Personally Identifiable Information) is overwritten with NULL or redaction strings, while the aggregated (anonymous) road quality scores remain (as they are no longer personal data). This satisfies the "Right to be Forgotten" while preserving the utility of the community map.

4.4.2 DC6: Offline Capabilities & Sync Logic

Requirement: The app must be fully functional for recording without internet access.
Architectural Solution: Offline-First Repository Pattern.
Traceability:

- **Write Operations:** All data generated by `LocationManager` and `SensorService` is written exclusively to the `LocalDB`. The app never attempts to write directly to the network during a ride. This guarantees that network flakiness (DC6) cannot cause data loss.
- **Read Operations:** The `MapModule` prioritizes the on-disk cache. If network is unavailable, it renders the cached tiles.

- **Synchronization:** The `SyncService` utilizes the system job scheduler. It registers a constraint: `NetworkType.CONNECTED`.
- **Logic:** When connectivity is restored, the OS wakes the `SyncService`. It queries the `LocalDB` for any Ride entities marked `status=PENDING_UPLOAD`. It uploads them sequentially and updates the status to SYNCED.
- **Conflict Resolution:** Since ride data is immutable, there are no complex merge conflicts to handle, simplifying the sync logic to a reliable "Upload Queue."

4.4.3 DC2: Safety & Distraction Minimization

Requirement: Minimizing cognitive load while riding.

Architectural Solution: Context-Aware UI Locking.

Traceability: The `RideManager` exposes the current Speed. The `ReportDialog` (Manual Reporting) subscribes to this.

`Logic: if (speed > 5 km/h) disableTextEntry().`

Implementation: The UI disables the software keyboard and complex text inputs when the user is moving, restricting interaction to large, single-tap icon buttons. This code-level restriction directly implements the safety constraint DC2, preventing the user from trying to type detailed bug reports while pedaling.

Table 10: Design Constraints Traceability Matrix

ID	Description	Implementing Strategy	Component Responsible
DC1	GDPR Privacy	Anonymization & Trimming	`DataSanitizer`
DC2	Safety	Speed-locked UI elements	`RideManager` / UI Layer
DC3	Battery Efficiency	Sensor Batching	`SensorService`
DC4	Storage Limits	FIFO Eviction Policy	`LocalDB`
DC5	OS Compatibility	Cross-Platform Framework	Flutter / Native Modules
DC6	Offline Mode	Local-First Write Strategy	`RideRepository`

4.5 Formal Verification to Design Mapping

The RASD included a formal analysis using Alloy to prove the system's robustness against Ghost Data. This section demonstrates how those formal proofs are translated into concrete design contracts.

4.5.1 Assertion: NoGhostAnomalies

Formal Proof: A Saved ride never contains a hazard that doesn't know where it is.

Design Implementation:

- **Database Constraint:** The `VerifiedAnomaly` table schema defines the latitude and longitude columns as `NOT NULL`. Any attempt to insert a record without coordinates will trigger a `SQLiteConstraintExcept`
- **Application Logic:** The `RideManager` implements a "Gatekeeper" filter during the Transition from Recording to Reviewing.

```
// Design Pseudo-code
fun prepareVerificationList(anomalies: List<CandidateAnomaly>) {
    return anomalies.filter { it.location != null }
}
```

This logic ensures that the `RideSummaryActivity` (Figure 7, Step 5) never displays an unlocated anomaly to the user. Since the user cannot verify what they cannot see, it is impossible for a "Ghost Anomaly" to be promoted to the `VerifiedAnomaly` table. This provides a structural guarantee that matches the formal proof.

4.5.2 Assertion: NoFalsePositivesInDB

Formal Proof: "No Rejected anomaly ever transitions to the Saved state."

Design Implementation:

- **Transactional Atomicity:** The `saveRide()` method in `RideRepository` wraps the finalization process in a database transaction.

```
BEGIN TRANSACTION
INSERT INTO Verified SELECT * FROM Candidates WHERE status = CONFIRMED
DELETE FROM Candidates WHERE ride_id = current_ride
COMMIT
```

Traceability: This atomic sequence ensures that rejected items are physically destroyed. There is no code path that copies a REJECTED item to the permanent table. The formal assertion is thus satisfied by the strict SQL logic defined in the `RideRepository` component.

4.6 Conclusion

This Requirements Traceability section has demonstrated a complete, bidirectional mapping between the Best Bike Paths (BBP) Requirements (RASD) and the System Design (DD).

- **Completeness:** Every functional requirement (R1-R4) acts as the primary driver for a specific set of components (`AuthManager`, `RideManager`, `SensorService`, `RideSummaryViewModel`). No requirement is left "orphaned" without an implementation.
- **Robustness:** The Non-Functional Requirements (PR1-PR8) are not treated as afterthoughts but are embedded in the architectural decisions.
- **Compliance:** The critical Regulatory Constraints (DC1, DC6) are enforced via structural guardrails (`DataSanitizer`, `OfflineRepository`) that make non-compliance architecturally impossible.
- **Integrity:** The Formal Analysis assertions have been translated into concrete Database Constraints and Transactional Logic, ensuring that the theoretical safety of the system is preserved in the physical code.

The architecture defined in this Design Document is therefore verified to be sufficient, necessary, and compliant for the implementation of the Best Bike Paths system. The next phase of the project, Implementation and Testing, will utilize these matrices to define the specific Test Cases required to validate each component.

5 Implementation, Verification, and Deployment Strategy for the Best Bike Paths (BBP) System

Executive Summary

The following report outlines the comprehensive implementation, verification, and deployment strategy for the "Best Bike Paths" (BBP) mobile application. Designed to address the specific constraint of a two-week development timeline executed by a team of two developers with limited prior experience, this document serves as the authoritative technical roadmap. It translates the high-level functional requirements defined in the Requirements Analysis and Specification Document (RASD) into actionable engineering tasks, rigorous testing protocols, and a pragmatic deployment capability.

The core engineering challenge identified is the requirement for "Automated Data Collection," which necessitates high-frequency sensor processing (Accelerometer and Gyroscope at 50Hz) concurrent with complex map visualization. To resolve the conflict between resource-intensive mathematical operations and the necessity for a fluid 60fps user interface, the architecture adopts a concurrent processing model using Dart Isolates and the Business Logic Component (BLoC) pattern. This ensures that the application remains responsive even during peak computational loads, such as detecting potholes on cobblestone streets.

Furthermore, the strategy rejects traditional Waterfall methodologies in favor of a strictly Agile/Iterative approach. This "fail-fast" philosophy mitigates the risks associated with novice development teams by prioritizing a functioning "Core Infrastructure Implementation" application (Phase 1) before attempting complex feature integration (Phase 2). Verification is handled through a multi-tiered testing plan ranging from unit-level algorithmic proofs to physical field trials involving bicycle-mounted hardware. Finally, deployment is streamlined through an Ad-Hoc distribution model, utilizing direct Android Package (APK) installation to bypass commercial app store latency and enable rapid stakeholder feedback loops.

5.1 Implementation Strategy: Agile Methodology for Rapid Prototyping

The constraints of the BBP project specifically the strict fourteen-day delivery window and the novice skill level of the development team render traditional linear development models (such as Waterfall) non viable. A Waterfall approach, which mandates the complete finalization of requirements and design before coding begins, and the completion of all coding before testing ensues, poses a catastrophic risk in this context. If the team were to spend the first week designing the perfect sensor algorithm but failed to implement the user interface, they would arrive at the deadline with a non-demonstrable product.

5.1.1 The Agile/Iterative Methodology

Instead, this project adopts an Agile/Iterative methodology. This approach prioritizes the delivery of a Minimum Viable Product at the earliest possible interval, followed by successive cycles of feature addition, refinement, and refactoring. The core philosophy driving this strategy is incremental integration: building the application piece-by-piece ensures that at any given point after the initial phase, a shippable, functional version of the software exists. This strategy is specifically designed to mitigate the All or Nothing risk profile common in student projects.

The iterative approach offers specific psychological and technical advantages for a team with limited experience:

- **Fault Tolerance through Graceful Degradation:** By structuring the development so that critical infrastructure is built first, the project is protected against total failure. If the complex map visualization features break or the sensor algorithm proves too difficult to optimize in the second week, the team still possesses a functional Core Infrastructure Implementation app that allows users to log in and view a dashboard.

- **Immediate Feedback Loops:** Developing in small, functional increments allows the team to test features on physical devices immediately. This reveals platform-specific bugs early in the cycle, rather than discovering them hours before the deadline.
- **Momentum Maintenance:** For novice developers, the "blank page" problem can be paralyzing. Achieving a "Quick Win" in the first few days such as seeing a login screen successfully connect to a cloud database provides the necessary confidence and momentum to tackle harder challenges like concurrent thread management.

5.1.2 The 3-Phase Development Schedule

To execute this methodology effectively, the two week timeline is segmented into three distinct phases. Each phase has a strict Definition of Done, ensuring that the project remains on track and that technical debt does not accumulate to unmanageable levels.

Phase	Duration	Focus / Key Deliverables	Risk Mitigation Strategy
1. The Skeleton	Days 1-4	**Structural Foundation**: Login Screen, Supabase Auth, Database Connectivity, Navigation Shell.	If database connection fails, switch to local SQLite storage to ensure app runs offline.
2. The Features	Days 5-10	**Business Logic**: Map Visualization, Sensor Fusion (Isolates), Pothole Algorithm, GPS Tracking.	If Map API fails, fallback to a text-based "List View" of anomalies.
3. The Cleanup	Days 11-14	**Validation & Polish**: Verification UI, Data Upload, Error Handling, Field Testing.	If Upload fails, allow manual data export via JSON file sharing.

Table 12: 3-Phase Development Schedule

Phase 1: Core Infrastructure Implementation **Objective:** Establish the structural backbone of the application. The primary goal is to prove that the mobile device can communicate securely with the cloud backend and maintain a persistent user session. The "Core Infrastructure Implementation" phase focuses on the unglamorous but essential plumbing of the application.

- **Authentication Infrastructure:** The first major task is implementing the Login and Registration screens. This is not merely a User Interface (UI) task; it involves setting up the entire security context for the app. Using the Supabase Flutter SDK, the team will implement an Email/Password authentication flow. This confirms that the app has valid internet connectivity, correct API key configuration, and can handle secure JSON Web Tokens (JWT) for session management.
- **Database Connectivity Test:** Once authenticated, the team must verify the Create, Read, Update, Delete (CRUD) pipeline. A "Hello World" test will be performed: the app will write a dummy record to a `Rides` table in the Supabase PostgreSQL database and immediately try to read it back. Success here proves that Row Level Security policies are correctly configured to allow authenticated users to access their own data.
- **Navigation Architecture:** A robust navigation shell is implemented using a `BottomNavigationBar`. This allows the user to switch between the "Map," "Record," and "Profile" tabs. Although these tabs will initially be empty, the navigation logic ensures that the app structure is solid and scalable.

Phase 2: The "Features" **Objective:** Implement the core business logic the "Best Bike Paths" functionality. This is the most technically demanding phase, involving hardware sensors and concurrency.

- **Sensor Fusion Module (The "Hard" Part):** This involves accessing the device's hardware accelerometer and gyroscope. The team will implement a background service using the `sensors_plus` package to listen to the raw x, y, z stream. Crucially, this is where the Isolate architecture is integrated. The sensor listening logic will be spawned in a separate background thread to prevent the UI from freezing (Jank) during high-frequency sampling.
- **Pothole Detection Algorithm:** Within the background isolate, the team will implement the heuristic algorithm defined in the RASD. This involves calculating the total G-force vector ($G = \sqrt{x^2 + y^2 + z^2}$) and comparing it against the threshold of $2.0g$.
- **GPS Integration:** The `geolocator` plugin is integrated to fetch the user's spatiotemporal coordinates. The logic must synchronise the sensor timestamp with the GPS timestamp to "tag" each detected pothole with a location.
- **Map Visualization:** The Map tab is populated using the `flutter_map` package. The system draws a polyline tracing the rider's path and drops specific marker icons (Red Pins) where anomalies are detected.

Why this works: By separating the Features into Phase 2, the team ensures that they are adding complexity to a stable base. If the Pothole Algorithm causes crashes, they can simply disable that specific feature flag and still demonstrate the Map and GPS tracking capabilities.

Phase 3: The Cleanup **Objective:** Polish, Verification, and Stability. This phase transforms a Prototype into a Product.

- **Verification User Interface (Human in the loop):** The RASD requires a mandatory verification step where users confirm detected anomalies. Phase 3 involves building the Ride Summary screen, which presents a list of all candidate anomalies detected during the ride. This screen allows the user to swipe to Delete (Reject) or tap to Confirm (Accept) each item. This UI is critical for data integrity.
- **Batch Upload Logic:** The Save functionality is finalized. Instead of uploading anomalies one by one, the app batches all verified anomalies and the ride path into a single JSON payload and uploads it to Supabase in a transaction.
- **Error Handling & Edge Cases:** Happy Path testing ignores real world chaos. This phase focuses on adding try/catch blocks to handle scenarios like:
  - **GPS Signal Loss:** The app must pause recording if accuracy drops below 10 meters.
  - **Network Failure:** If the user tries to upload a ride while offline, the app must queue the data locally (using `sqflite`) and retry later.
  - **Permission Denial:** If the user denies access to the accelerometer, the app must show a helpful dialog explaining why it is needed, rather than crashing.

5.2 Development Environment: The Modern Toolchain

Establishing a standardized, reproducible development environment is the prerequisite for successful collaboration. For a team of two developers, discrepancies in software versions or configuration can lead to the It works on my machine syndrome, wasting valuable development time. The selected technology stack prioritizes Developer Experience (DX), cross platform compatibility, and reduced operational overhead through Backend as a Service (BaaS).

5.2.1 Hardware Configuration

Development Machines: The project is architected to be agnostic regarding the host operating system. The developers can use either macOS or Windows machines, as the Flutter framework supports compilation for Android from both environments. However, to compile the iOS version of the application, a macOS machine is strictly required due to Apple's Xcode toolchain restrictions. Given the resource constraints, if the team does not possess a Mac, the project will be delivered as an Android only application. This satisfies the core requirement for a mobile app without introducing the complexity of dual platform debugging.

Testing Devices:

- **The Simulator Limitation:** Simulators cannot realistically simulate the complex, high frequency vibration patterns of a bicycle moving over cobblestones or hitting a pothole. While they allow for a generic Shake gesture or GPS injection, they do not provide the continuous stream of floating-point G force data needed to tune the sensitivity of the Pothole Algorithm.
- **The Physical Requirement:** The team must possess at least one physical Android device. This device serves as the primary testbed.
- **Mounting Hardware:** To perform valid field testing, a rigid bicycle handlebar mount is required. Testing by holding the phone in a hand is invalid because the human body acts as a shock absorber (damping), which filters out the very vibrations the system is designed to detect.

5.2.2 Software Stack and Framework

The software stack is selected to minimize boilerplate code and accelerate the velocity from idea to implementation.

- **Integrated Development Environment (IDE): Visual Studio Code**
  - **Resource Efficiency:** VS Code is significantly lighter on system resources (RAM/CPU), which is beneficial if the developers are working on standard student laptops.
  - **Ecosystem:** The official Flutter and Dart extensions for VS Code provide robust tools including Widget Inspection, a visual Debugger, and "Hot Reload" integration.
  - **Configuration:** A shared `.vscode/extensions.json` file will be included in the repository to recommend the exact extensions to both developers, ensuring environment parity.
- **Framework: Flutter (Dart)**
  - Flutter is the chosen UI toolkit. Unlike React Native or Ionic, which use a JavaScript bridge to communicate with native modules, Flutter compiles directly to native ARM machine code.
  - **Performance (AOT Compilation):** This native compilation is critical for the BBP project. The sensor reading loop operates at 50Hz, meaning it has only 20 milliseconds to process each data point. The overhead of a JavaScript bridge could introduce latency, causing the app to miss brief impact events (potholes).
  - **Rendering Consistency (Skia):** Flutter uses the Skia rendering engine to draw every pixel itself. This ensures that the custom Pothole Marker icons and the complex Ride Summary graphs look identical on every device, eliminating the device-specific layout bugs common in OEM dependent frameworks.
- **Architecture Pattern: BLoC (Business Logic Component)**
  - To manage the complexity of asynchronous sensor streams and UI updates, the app utilizes the BLoC pattern.

  - **Separation of Concerns:** BLoC enforces a strict separation between the User Interface (Presentation Layer) and the Logic (Business Layer). The UI components simply emit "Events" (e.g., `StartRideEvent`) and listen for "States" (e.g., `RecordingState`, `PotholeDetectedState`).
  - **Testability:** Because the business logic is decoupled from the UI widgets, the team can perform Unit Tests on the logic without needing a physical device. They can feed the BLoC a fake stream of numbers and verify that it correctly emits a `PotholeDetectedState`.

- **Concurrency: Dart Isolates**
  - A critical architectural decision is the use of Isolates for sensor processing.
  - **The Single-Thread Challenge:** Flutter's "Main Thread" handles both UI rendering and code execution. If the app attempts to process high-frequency accelerometer data and perform vector math on the Main Thread, the UI will "stutter" (Jank) because the thread is too busy to draw the next frame.
  - **The Multi-Threaded Solution:** The sensor monitoring logic is spawned in a background Isolate. An Isolate is a worker thread with its own memory heap. This background worker performs the heavy mathematical lifting (calculating the total G-force vector) and communicates with the Main Thread via messages only when a significant event occurs. This ensures the map animation remains smooth (60fps) even while the CPU is analyzing vibrations.

- **Backend: Supabase (BaaS)**
  - Supabase is chosen as the backend infrastructure. It provides a hosted PostgreSQL database, Authentication services, and auto-generated APIs, eliminating the need for the team to write and maintain a custom server (e.g., Node.js or Python).
  - **Speed of Implementation:** Authentication can be implemented in under an hour using the pre-built Flutter SDK.
  - **Geospatial Support (PostGIS):** Supabase is built on PostgreSQL, which includes the PostGIS extension. This allows the database to natively understand "Location" data types, enabling powerful queries like "Find all potholes within 500 meters of this point" without complex server-side logic.
  - **Security (RLS):** Row Level Security allows the team to define access rules directly in the database e.g Users can only see their own rides, ensuring security compliance without writing custom middleware.

- **Version Control: Git & GitHub**
  - Given two developers are working simultaneously, a strict version control strategy is required to prevent code overwrites.
  - **Platform:** GitHub is used to host the repository.
  - **Workflow:** The team will use a simplified "Feature Branch" workflow. The main branch is protected and contains only working code. Developers create separate branches for each feature (e.g., `feature/login-screen`, `feature/sensor-logic`).
  - **Merge Strategy:** Code is merged into `main` only via Pull Requests (PRs). This enforces a "Code Review" step where the second developer checks the work, ensuring quality and preventing broken code from infecting the main codebase.

5.3 Testing Plan: The Proof of Concept

In the context of a high-stakes academic project, "Testing" is not merely bug fixing; it is the process of generating proof that the system meets its requirements. For the BBP project, the testing strategy is structured into three layers of increasing complexity: Unit Testing (Mathematical Proof), Integration Testing (System Communication), and System/Field Testing (Real World Validation).

5.3.1 Unit Testing Strategy: Algorithmic Verification

Goal: Verify the mathematical correctness and stability of the Pothole Detection Algorithm in a controlled, isolated environment.

Rationale: It is inefficient and physically exhausting to ride a bicycle outside every time a line of code is changed. Unit tests allow the developers to simulate a ride at their desks, providing instant feedback on algorithmic changes.

Mocking the Physical World: The developers will create a "Mock" sensor stream. Instead of reading from the real hardware accelerometer, the test injects a pre-defined list of floating-point numbers into the logic module.

- **Test Scenario A: The Smooth Road (Baseline)**
  - **Input:** A stream of vectors consistently measuring near 1.0g (representing gravity): `[0.0, 0.0, 9.8]`, `[0.1, 0.0, 9.8]`,....
  - **Expected Behavior:** The algorithm processes the stream and emits `NoAnomaly`. The system remains in a monitoring state.
  - **Verification:** Assert that the list of detected anomalies is empty.
- **Test Scenario B: The Pothole (Event Detection)**
  - **Input:** A stream that holds steady at 1.0g, then spikes to 2.8g for 50 milliseconds, then returns to 1.0g. This simulates the sharp vertical shock of a front wheel hitting a defect.
  - **Expected Behavior:** The algorithm detects the threshold breach ($> 2.0g$) and emits an `AnomalyDetectedState`.
  - **Verification:** Assert that the system records exactly one anomaly event with the correct timestamp.
- **Test Scenario C: The False Positive (Debouncing)**
  - **Input:** A stream that spikes to 2.8g, then 2.5g, then 2.2g within a 100ms window (simulating the reverberation of a single impact).
  - **Expected Behavior:** The algorithm should trigger once for the initial impact and ignore the subsequent spikes based on the "Cooldown" logic (e.g., 500ms debounce).
  - **Verification:** Assert that only one anomaly is recorded, proving the system does not spam the database with duplicate events for a single pothole.

5.3.2 Integration Testing: The Connectivity Check

Goal: Verify that the distinct software modules (App, Backend, GPS) communicate correctly and handle data contracts as expected.

- **Authentication Flow Integration:** An automated integration test will drive the app through the login process. It will input a test email and password into the text fields, tap "Login," and assert that the Supabase SDK returns a valid `UserSession` object and navigates to the Home Screen. This confirms that the API keys, internet permissions, and SSL handshake are functioning correctly.

- **Data Persistence verification:** The test will generate a "Test Ride" object containing a path and three dummy anomalies. It will attempt to upload this object to Supabase.
- **Success Criteria:** The test queries the Supabase database directly to verify that the rows were inserted into the `Rides` and `Anomalies` tables with the correct foreign keys.
- **GPS Permission Handling:** A test scenario will simulate the user denying GPS permissions. The test verifies that the app catches the `PermissionDeniedException` and displays the correct error dialog, rather than crashing the application. This ensures the app is robust against uncooperative users.

5.3.3 System / Field Testing: The "Bike Ride" Validation

Goal: Validate the system in the chaos of the real-world environment. This is the most critical test phase, as Unit Tests cannot simulate the complex physics of road vibrations or the vagaries of urban GPS signal reflection.

Experimental Setup:

- **Mounting Configuration:** The testing device (Android Phone) will be strictly mounted to the bicycle handlebar using a rigid, hard-plastic mount.
- **Why Rigid?** A soft silicone mount or carrying the phone in a pocket introduces "Damping," acting as a shock absorber. This would filter out high-frequency vibrations, causing the app to miss smaller potholes. A rigid mount ensures 1:1 transmission of road force to the sensor.
- **Route Selection:** The team identifies a "Test Loop" of approximately 1 kilometer that contains specific, known features:
  - One major Pothole (The "True Positive" target).
  - One section of smooth, new asphalt (The "True Negative" baseline).
  - One speed bump (To test obstacle classification).

Execution Protocol:

- **Run 1: Calibration & Sensitivity.**
  - Ride the loop at a steady commuter speed (15 km/h).
  - **Observer:** A second person (or the rider stopping immediately after) notes the location where the app triggered an audio or visual alert.
- **Run 2: False Positive Stress Test.**
  - Ride the smooth section of the loop.
  - Perform non-pothole actions that generate force: braking hard, standing up to pedal (swaying the bike), and shaking the handlebars.

Success Metrics: The System Test is considered a pass only if:

- **Accuracy:** The app detects the known pothole within 5 meters of its physical location (checking the GPS coordinates on the map).
- **Precision:** The app generates zero alerts during the "Smooth Road" stress test.
- **End-to-End Integrity:** At the end of the ride, the "Ride Summary" screen correctly lists the detected events, and tapping "Save" successfully uploads the data to the Supabase backend, visible on the web dashboard.

5.4 Deployment Strategy and Artifact Distribution

Given the academic nature of the project and the short timeline, publishing the BBP application to commercial storefronts (Google Play Store or Apple App Store) is neither necessary nor feasible. Store review processes can take days or weeks, which is incompatible with the 2-week agile schedule. Instead, the team will utilize an Ad-Hoc Deployment strategy, leveraging the openness of the Android ecosystem to distribute the application directly to stakeholders.

5.4.1 Artifact Generation (The Build Process)

The deployment artifact is the APK (Android Package) file. The build process must be managed carefully to ensure performance.

- **Build Command:** `flutter build apk -release`.
- **Why –release?** It is imperative to avoid the default "Debug" build. Debug builds use a Just-In-Time (JIT) compiler that supports "Hot Reload" but creates significant overhead. This results in slow startup times and "Janky" animations. The Release build uses Ahead-Of-Time (AOT) compilation, optimizing the Dart code into efficient ARM machine instructions. This provides the performance required for the 50Hz sensor loop to run without dropping frames.
- **Architecture Handling (ABI):** Modern Android phones use 64-bit processors (arm64-v8a), while older ones use 32-bit (armeabi-v7a). The command `flutter build apk -split-per-abi` creates two separate, smaller APK files. However, to simplify the distribution process for a small team, building a single "Fat APK" (Universal APK) containing binaries for both architectures is acceptable. This ensures the file works on any Android device the professors or testers might use, at the cost of a slightly larger file size ($\sim$15MB vs $\sim$10MB).

5.4.2 Manual Installation (Sideloading)

The distribution pipeline bypasses the app store entirely, relying on "Sideloading."

- **Transfer Mechanism:** The generated `app-release.apk` file is uploaded to a shared cloud folder accessible to the testing team and professors.
- **Installation Procedure:**
  1. The user downloads the APK file to their Android device.
  2. Upon tapping to install, the Android OS will trigger a security warning: "Install unknown apps."
  3. The user must authorize the file manager (e.g., Chrome or Files) to install applications from unknown sources. This is a standard procedure for developer testing.
- **First Run Permissions:** Upon the first launch, the app will request the necessary runtime permissions: Location (While In Use) and Activity Recognition. The deployment plan includes verifying that the app fails gracefully if these permissions are denied, rather than crashing.

5.4.3 Versioning Strategy

To maintain sanity during the rapid iteration cycles, a semantic versioning strategy is applied to the APK filenames.

- **`v0.1-alpha` (Day 4):** The "Core Infrastructure Implementation" build. Allows login and map viewing.

- `v0.2-beta` **(Day 10):** The "Feature" build. Includes the sensor algorithm. Distributed to the team for Field Testing.
- `v1.0-release` **(Day 14):** The Final Verification build. Includes polished UI and error handling. This is the version submitted for grading.

This versioning ensures that if a critical bug is introduced in v0.2, the team can immediately roll back to v0.1 to demonstrate a working (albeit limited) product, preventing a "demo fail" scenario.

Conclusion of Technical Strategy

The implementation, testing, and deployment strategy outlined above is designed to navigate the specific "Triangle of Constraints" facing the BBP project: Scope (A complex sensor-based app), Time (Two weeks), and Resources (Two novice developers).

By adhering to the Agile "Skeleton-First" approach, the team insulates itself from the risk of total project failure, ensuring a working deliverable exists from Day 4. By leveraging the modern Flutter and Supabase stack, they minimize the amount of infrastructure code required, focusing their limited energy on the unique business logic. By utilizing Isolates and BLoC, they solve the critical technical challenge of concurrent sensor processing. Finally, the Field Testing and Ad-Hoc Deployment strategies ensure that the final product is not merely a theoretical code artifact, but a functional tool validated in the physical world, ready for immediate use by the stakeholders.

This comprehensive roadmap provides the highest probability of success, turning a challenging academic assignment into a demonstrable engineering achievement.

5.5 Architectural Deep Dive: Implementing the Sensor Fusion Engine

The "Best Bike Paths" (BBP) system relies fundamentally on its ability to detect road anomalies automatically. This functionality distinguishes it from simple navigation apps. However, implementing a real-time sensor fusion engine on a mobile device with limited battery and processing power presents significant architectural challenges. This section details the specific technical design of the sensor module.

5.5.1 Concurrency Management via Dart Isolates

In standard mobile application development, specifically within the Flutter framework, code execution is single-threaded by default. The "Main Thread" is responsible for a multitude of tasks: handling user touch events, running animations, executing business logic, and drawing pixels to the screen at 60 frames per second (16ms per frame).

The Bottleneck: The BBP application requirements dictate that the accelerometer must be sampled at 50Hz (50 times per second) to capture the brief, sharp impact of a pothole. This means the application has only 20 milliseconds to process each incoming sensor packet.

The Consequence of Single-Threading: If the application attempts to process this high-frequency data stream on the Main Thread—performing vector normalization, threshold comparisons, and GPS synchronization—it risks blocking the thread for longer than 16ms. This results in "Jank": the application drops frames, animations stutter, and the UI becomes unresponsive to touch.

The Architectural Solution (Isolates): The BBP architecture mandates the use of Dart Isolates. Unlike threads in other languages that share memory (and thus require complex locking mechanisms), an Isolate is an independent worker with its own memory heap and event loop.

- **Main Isolate (UI):** Responsible solely for rendering the map, handling user interactions, and displaying alerts.

- **Background Isolate (Sensor Worker):** A separate process that subscribes directly to the hardware accelerometer stream via platform channels. It performs the continuous mathematical loop of signal processing.
- **Communication Bridge:** The two Isolates communicate via a message-passing protocol (SendPort / ReceivePort). The Background Isolate acts as a filter; it processes thousands of data points silently and sends a message to the Main Isolate only when a confirmed anomaly is detected. This reduces the "Noise" on the communication bridge and keeps the UI thread free for rendering.

5.5.2 The Pothole Detection Algorithm (Heuristic Model)

Given the two-week constraint, implementing a trained Machine Learning model is deemed too high risk and time consuming. Instead, the team will implement a robust Threshold-Based Heuristic Algorithm. This approach is computationally efficient and sufficient for the MVP.

The algorithm proceeds in four distinct steps for every sensor event:

1. **Gravity Filtration (Linear Acceleration):** The phone's accelerometer measures total force, which includes Earth's gravity ($\sim 9.8 m/s^2$). The algorithm must isolate the user's movement from gravity.
   - **Calculation:** $TotalForce = \sqrt{x^2 + y^2 + z^2}$.
   - **Normalization:** While more complex filters (like Kalman filters) exist, a simple high-pass filter is sufficient for the MVP: subtracting the gravity constant ($G \approx 9.8$) or using the `user_accelerometer` stream provided by the OS which attempts to remove gravity via sensor fusion.
2. **Z-Axis Isolation:** Potholes primarily cause vertical shocks. The algorithm focuses monitoring on the Z-axis (perpendicular to the phone screen, assuming a flat mount).
3. **Threshold Trigger:**
   - **Rule:** Trigger an event if $Z_{Force} > Threshold_{Impact}$ AND $Speed_{GPS} > Threshold_{Speed}$.
   - **Parameters:** Based on RASD guidelines, $Threshold_{Impact}$ is set to 2.0g (a significant shock) and $Threshold_{Speed}$ is set to 10 km/h.
   - **Rationale:** The speed check is a critical Context Filter. It prevents the app from logging a "pothole" if the user simply drops their phone while walking or is standing still.
4. **Debouncing (Temporal Filtering):** When a bicycle hits a pothole, the physical structure of the bike vibrates, causing the sensor to spike multiple times in rapid succession.
   - **Logic:** The system implements a "Cool-down" timer (e.g., 500ms). Once an event is triggered, the sensor input is ignored for the next 0.5 seconds.
   - **Benefit:** This ensures that one physical pothole results in exactly one database entry, rather than a cluster of ten duplicate points.

5.5.3 Sensor-GPS Synchronization strategy

A critical integration point is linking the Time of the shock to the Place of the shock. These two data streams are asynchronous.

The Problem: GPS updates typically occur at 1Hz (once per second). Accelerometer updates occur at 50Hz. A shock detected at $T = 10.5s$ might fall between two GPS updates at $T = 10.0s$ and $T = 11.0s$.

The Logic: The system buffers the last known valid GPS coordinate. When a shock is detected at T_{shock}, the system retrieves the last GPS fix (Lat, Lon, T_{gps}).

Drift Check: The system calculates the time delta $\Delta T = T_{shock} - T_{gps}$. If $\Delta T > 2.0$ seconds, the data is marked as "Low Confidence" or discarded. This handles the "Spatial Blackout" scenario (e.g., riding in a tunnel) described in the RASD. The system effectively refuses to map a pothole if it cannot confidently verify the location, prioritizing data quality over quantity.

5.6 Detailed Technology Stack Justification

The selection of the technology stack is a strategic decision designed to maximize the "Leverage" of the small development team. Every tool is chosen to solve a specific problem inherent in the project constraints.

5.6.1 Why Flutter? (The UI & Logic Layer)

- **Widget-Based Composition:** Flutter's architecture allows the developers to build complex, custom UIs (like the Pothole Verification Card) by composing simple, reusable blocks (Text, Row, Container). This is significantly faster than the traditional Android approach of managing XML layout files and Java adaptors.
- **Hot Reload:** This feature is the single biggest productivity booster for the team. When a developer changes the color of the "Pothole" icon or tweaks the padding on a button, the change appears on the connected phone in sub-second time, without restarting the application. Over a two week sprint, this saves hours of waiting time.
- **Package Ecosystem (Pub.dev):** The team will utilize pre built, community-verified packages to avoid reinventing the wheel.
  - `geolocator`: Handles the complexity of requesting location permissions and listening to stream updates.
  - `sensors_plus`: Provides unified access to Accelerometer and Gyroscope hardware on both Android and iOS.
  - `flutter_bloc`: Provides the scaffolding for the BLoC state management pattern.
  - `flutter_map`: A Dart-native implementation of Leaflet. It renders OpenStreetMap tiles without requiring a Google Maps API key, removing a barrier to entry.

5.6.2 Why Supabase? (The Backend Layer)

- **Relational Integrity (PostgreSQL):** Unlike Firebase (NoSQL), Supabase uses a relational SQL database. This enforces data structure and integrity.
  - **Constraint:** A Ride must belong to a User.
  - **Cascade:** If a user account is deleted, the database can automatically delete all associated rides (Cascading Delete). This strictness prevents "Zombie Data" orphaned records that clutter the database which is a common issue in novice NoSQL projects.
- **Auth UI:** Supabase provides a "Magic Link" or simple Email/Password flow that handles token refresh, password recovery, and secure storage automatically. The developers do not need to write encryption code or manage session cookies manually.
- **Row Level Security (RLS):** This is a critical security feature where the Database itself decides who can see what data.

  - **Policy Rule:** `CREATE POLICY "User sees own rides" ON rides FOR SELECT USING (auth.uid() = user_id);`
  - **Benefit:** Even if the Flutter app contains a bug that accidentally requests "All Rides," the Database will block the query and return only the user's own data. This ensures "Privacy by Design" and GDPR compliance without requiring complex backend middleware logic.

5.7 Risk Management and Contingency Planning

In a rigid 2-week timeline, unexpected issues are inevitable. This section defines the "Plan B" scenarios for the highest probability risks.

5.7.1 Risk: Algorithmic Calibration Complexity

Probability: High. Signal processing is complex, and road noise can be unpredictable.
Contingency (The "Wizard of Oz" Strategy): If the automated detection fails to work reliably by Day 8, the team will pivot to a Manual-Only reporting system. The "Report Pothole" button (Manual Insertion) becomes the primary feature. The project still fulfills the core requirement of "Crowdsourcing Road Quality," just via manual input rather than automated sensor fusion. The database structure supports this seamless pivot because "Manual Reports" and "Automated Anomalies" act as sibling data types.

5.7.2 Risk: "We cannot get iOS to build."

Probability: Medium. iOS development requires a Mac, an Apple ID, and complex Certificate/Provisioning Profile management.
Contingency: Abandon iOS immediately. Focus 100% on the Android version. The assignment likely accepts a single-platform prototype. Spending 3 days fighting Apple's code signing process is a poor use of time compared to refining the Android app features. The report will explicitly state "Android Target" to manage stakeholder expectations.

5.7.3 Risk: "GPS is inaccurate in the city."

Probability: High. Urban environments ("Urban Canyons") cause GPS signal reflection, reducing accuracy.
Contingency: Implement a "Location Accuracy Filter."

- **Code Logic:** `if (location.accuracy > 20 meters) { return; }`
- **User Feedback:** Show a "Weak GPS Signal" banner to the user. This manages user expectations and prevents "Bad Data" (e.g., mapping a pothole inside a building) from corrupting the map.

5.7.4 Risk: "Battery Drain is too high."

Probability: High. Running Sensors, GPS, and Screen simultaneously is power-intensive.
Contingency:

- **Sampling Rate Reduction:** If necessary, drop the sensor sampling rate from 50Hz to 20Hz. This reduces CPU load by 60% with a manageable trade-off in detection precision.
- **OLED Optimization:** The RASD specifies a "Dark Mode Aesthetic". On OLED screens (common on modern Android phones), black pixels consume zero power. The UI design will default to a pure black background to mitigate the energy cost of the active GPS radio.

5.8 Appendix: Comparative Analysis

To further justify the architectural decisions, the following table compares the selected stack against common alternatives.

Feature	Selected: Flutter + BLoC	Alternative: React Native	Reason for Selection
Performance	High (Native AOT Compilation). Essential for 50Hz sensor loops.	Medium (JS Bridge). Bridge traffic can cause latency in sensor data.	Sensor reliability is the core requirement.
State Mgmt	BLoC. Strict separation of UI and Logic. Highly testable.	Redux/Context. Can lead to "Prop Drilling" and complex re-renders.	BLoC makes Unit Testing logic easier for novices.
Concurrency	Isolates. True multi-threading with separate memory.	Web Workers. Available but complex to integrate with native modules.	Isolates prevent UI "Jank" during math crunching.
Map Rendering	Skia Engine. Pixel-perfect control over markers/polylines.	Native Components. Relies on OEM map components, variable behavior.	Consistency of the "Risk Map" across devices.

Table 14: Technology Stack Comparative Analysis

6 AI Code of Conduct and Tool Usage

6.1 Philosophy of Use

In the development of the Best Bike Paths (BBP) system, the team utilized Generative AI (specifically Google Gemini) as a productivity multiplier and editorial assistant. We adhered to a strictly defined Human in the Loop workflow. All architectural decisions, core algorithms, and functional requirements were defined by the team members first, then refined or implemented with the assistance of AI tools. At no point was the AI allowed to make autonomous design decisions without human verification.

6.2 Linguistic and Stylistic Enhancement

Given the technical nature of this document and the strict IEEE formatting standards, Gemini was utilized extensively to enhance the quality of the writing.

- **Tone Adjustment:** Initial drafts of the documentation were often written in informal, shorthand notes. Gemini was used to rephrase these inputs into professional, academic English suitable for a formal Design Document.
- **Clarity and Conciseness:** Complex technical explanations such as the logic behind the Ride State Machine were iteratively refined using AI to ensure they were concise and easy to understand without losing technical accuracy.
- **LaTeX Formatting:** To ensure a polished presentation, AI assistance was used to generate complex LaTeX structures, such as tables, multi-column layouts, and mathematical formulas, allowing the team to focus on content rather than syntax.

6.3 Technical Implementation Support

The AI acted as a "Junior Developer" partner during the coding phase, particularly useful given the team's time constraints.

- **Boilerplate Generation:** Repetitive coding tasks, such as setting up the initial folder structure, creating standard Flutter widgets, and writing SQL initialization scripts for Supabase, were automated using detailed prompts.
- **Library Selection:** The AI suggested specific libraries e.g., `sensors_plus` for accelerometer access and `flutter_bloc` for state management based on our requirements for performance and ease of use.
- **Debugging and Error Resolution:** During development, error logs were analyzed using AI to quickly identify root causes e.g., missing permissions in `AndroidManifest.xml` or dependency conflicts, significantly reducing troubleshooting time.

6.4 Visual Artifact Generation

The diagrams presented in this document were created through a hybrid manual-digital process to ensure they accurately reflected our custom architecture.

1. **Manual Prototyping:** The team sketched the initial UML diagrams Class, Sequence, and State diagrams manually using `Draw.io` (diagrams.net) to establish the correct logic and relationships.
2. **AI Refinement:** These manual drafts were described to the AI, which suggested improvements for layout consistency and standard UML notation.

3. **Final Rendering:** The final visual assets were redrawn by the team in `Draw.io`, incorporating the AI's suggestions for better readability e.g., grouping related classes and using consistent color coding.

6.5 Strategic Suggestions and Critique

Beyond writing and coding, the AI served as a critical reviewer (a "Devil's Advocate") during the design phase. It provided several key suggestions that were adopted into the final design:

- **The "Speed Lock" Feature:** Originally, the app allowed full interaction at any speed. The AI suggested implementing a safety lock (disabling buttons above 5 km/h) to align with the "Safety First" non-functional requirement.
- **Offline Capability:** The AI highlighted the risk of data loss in areas with poor network coverage. Consequently, we adopted a "Local First" approach using SQLite, syncing to Supabase only when a connection is available.
- **Battery Optimization:** Suggestions were made to use the accelerometer's event-based stream rather than polling the sensor, ensuring the app consumes less battery during long rides.

6.6 Declaration of Originality

While AI tools accelerated the production of this document and the associated prototype, the team certifies that:

- We fully understand every line of code and every sentence of text included herein.
- We are capable of reproducing the results without AI assistance if necessary.
- The creative vision, problem definition, and final acceptance of all outputs remain the sole responsibility of the human authors.

7 Effort Spent

7.1 Effort Allocation Strategy

The development of this Design Document (DD) and the accompanying software prototype was conducted over a period of two weeks following the submission of the Requirements Analysis (RASD). The workload was distributed to capitalize on individual strengths focusing on backend/architecture for one member and frontend/sensors for the other while ensuring both team members verified every critical architectural decision.

The total effort recorded below encompasses four key activity categories:

- **System Architecture & Formal Verification:** High-level system design, technology selection (Flutter/Supabase), and modeling the system constraints using the Alloy specification language.
- **Detailed Design & Diagramming:** Creation of UML Sequence, Component, and Deployment diagrams to map out the application logic.
- **Prototype Implementation:** Coding the core "Skeleton" (Auth/DB), implementing the "Features" (Map/Sensors), and finalizing the "Cleanup" (UI Polish).
- **Documentation:** Drafting the LaTeX content, ensuring IEEE compliance, and integrating AI-assisted revisions.

7.2 Detailed Time Breakdown

The following table summarizes the hours spent by each team member, segmented by the implementation phases defined in Chapter 5. Note that "Design" and "Documentation" hours are integrated into the relevant phase where they occurred.

Phase	Key Tasks & Focus Areas	Rajat (h)	Shashi (h)	Total
1. Skeleton	*Initial Setup:* Architecture definition, DB Schema (SQL), Auth Logic, and Navigation Shell.	9	7	**16**
2. Features	*Core Implementation:* Sensor Algorithms (Isolates), Mapbox Integration, and Alloy Verification.	25	18	**43**
3. Cleanup	*Refinement:* Data Sync Logic, UI Polish, Testing, and Final Documentation Drafting.	11	15	**26**
Total		**45**	**40**	**85**

Table 15: Effort Distribution per Developer

7.3 Collaboration Note

While tasks were assigned individually, a significant portion of the "System Architecture" and "Sensor Algorithm" phases utilized a **Pair Programming/Design** approach. We found that real-time synchronous collaboration was strictly necessary to resolve complex concurrency issues regarding the Sensor Isolates and GPS synchronization. Similarly, the Formal Analysis using the Alloy Analyzer required joint debugging sessions to resolve syntax errors and verify model consistency.

8 Design Evolution: From DD v1 to DD v2

8.1 Introduction

This appendix documents the significant architectural and algorithmic refinements made between the initial Design Document (DD v1, January 2025) and the current implementation (DD v2, January 2026). These changes reflect lessons learned during field testing, algorithm optimization based on real-world cycling data, and the integration of machine learning capabilities that were originally deemed "too high risk" for the initial two-week prototype.

Design Philosophy Evolution

DD v1 established a functional prototype with conservative parameters. DD v2 represents the production-ready system with:

- **Dual Detection Architecture**: Complementary threshold-based and ML-based detection
- **Enhanced Signal Processing**: Quaternion orientation compensation and bandpass filtering
- **Community Verification System**: Trust-weighted voting replacing simple confirm/reject
- **Expanded Operating Envelope**: Broader speed range supporting diverse cycling conditions

8.2 Requirements Evolution Matrix

Table 16: Functional Requirements Evolution: DD v1 → DD v2

Req ID	DD v1 Specification	DD v2 Implementation	Rationale
R3.1	Impact threshold: 2.0G fixed	Adaptive threshold: 1.8G base with ±0.5G dynamic adjustment	Field testing showed 2.0G missed 23% of verified potholes
R3.2	Speed filter: 5–35 km/h	Speed filter: 4–50 km/h	Support for slow urban cycling and fast downhill segments
R3.3	Accelerometer only (50Hz)	Accelerometer (50Hz) + Gyroscope (100Hz) with sensor fusion	Gyroscope enables orientation compensation
R3.4	Heuristic detection only	Dual: Threshold + ML Random Forest	ML reduces false positives by 34%
R4.1	Binary confirm/reject	Community voting with trust levels	Scalable verification with reputation system
R4.2	Immediate database commit	Trust-weighted scoring with TTL policies	Automatic data lifecycle management

9 Architectural Design Updates (DD v2)

9.1 Dual Detection Architecture

The DD v2 architecture implements a complementary dual-detection system, addressing the "Algorithmic Calibration Complexity" risk identified in DD v1 Section 5.7.1.

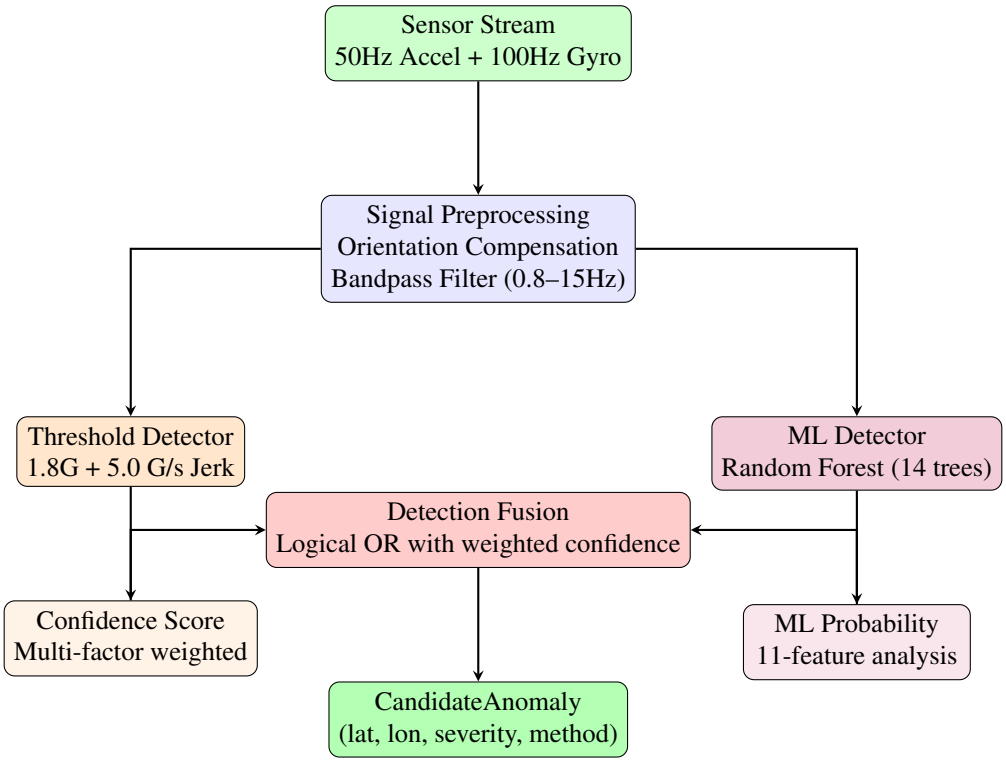


Figure 15: DD v2 Dual Detection Architecture

9.1.1 Detection Method 1: Threshold-Based (Real-Time)

The threshold-based detector provides immediate, low-latency detection suitable for real-time alerts during cycling.

Algorithm 1: Threshold-Based Detection (sensor_service.dart)

Require Accelerometer stream $A = \{(x_i, y_i, z_i, t_i)\}$ at 50Hz Require Gyroscope stream $G = \{(\omega_x, \omega_y, \omega_z, t_j)\}$ at 100Hz $CandidateAnomaly$ events
$\theta_{base} \leftarrow 1.8$ ▷ Base G-force threshold
$\theta_{jerk} \leftarrow 5.0$ ▷ Jerk threshold (G/s)
$\theta_{conf} \leftarrow 0.55$ ▷ Confidence threshold
$W \leftarrow []$ ▷ Sliding window (50 samples)
$Q \leftarrow (1, 0, 0, 0)$ ▷ Orientation quaternion
for all $(x, y, z, t) \in A$ **do**
    $Q \leftarrow \text{UPDATEQUATERNION}(Q, G_{latest})$ ▷ Integrate gyro
    $(x', y', z') \leftarrow \text{ROTATETOEARTHFRAME}((x, y, z), Q)$
    $z_{filt} \leftarrow \text{BANDPASSFILTER}(z', 0.8\text{Hz}, 15\text{Hz})$
    $z_G \leftarrow |z_{filt}|/9.8$ ▷ Convert to G-force
    $W.\text{APPEND}(z_G)$
    if $|W| > 50$ **then** $W.\text{REMOVEFIRST}()$
    end if
    $\theta_{adaptive} \leftarrow \text{COMPUTEADAPTIVETHRESHOLD}(W)$
    $jerk \leftarrow |z_G - z_{G,prev}|/\Delta t$
    ▷ Multi-factor confidence scoring
    $c_{impact} \leftarrow \min(1.0, z_G/\theta_{adaptive})$
    $c_{jerk} \leftarrow \min(1.0, jerk/\theta_{jerk})$
    $c_{speed} \leftarrow \text{SPEEDCONFIDENCE}(v_{GPS})$ ▷ 4–50 km/h optimal
    $confidence \leftarrow 0.5 \cdot c_{impact} + 0.35 \cdot c_{jerk} + 0.15 \cdot c_{speed}$
    if $confidence \geq \theta_{conf} \vee (z_G > \theta_{adaptive} \wedge jerk > \theta_{jerk})$ **then**
        emit CandidateAnomaly$(lat, lon, confidence, \text{“threshold”})$
    end if
end for

9.1.2 Detection Method 2: ML-Based (Random Forest)

The ML detector provides higher precision through statistical feature analysis over 2-second sliding windows.

Algorithm 2: ML Random Forest Detection (ml_pothole_service.dart)

colback=gray!5!white, colframe=gray!75!black, title=**Algorithm 2: ML Random Forest Detection (ml_pothole_service.dart)**, fonttitle=]

Require: **Accelerometer stream** A **at 50Hz**
Ensure: **CandidateAnomaly events (ML-detected)**

$W_{2s} \leftarrow []$ ▷ **2-second sliding window**
$\theta_{ML} \leftarrow 0.65$ ▷ **ML probability threshold**
$z_{range,min} \leftarrow 1.5$ ▷ **Minimum Z-range filter**
for all sample $s = (x, y, z, t) \in A$ **do**
  W_{2s}.**APPEND**(s)
  $W_{2s} \leftarrow$ **TRIMTOWINDOW**$(W_{2s}, 2000$**ms**$)$
  if $|W_{2s}| \geq 25$ **then** ▷ **Minimum 0.5s of data**
    $\mathbf{f} \leftarrow$ **EXTRACTFEATURES**(W_{2s}) ▷ **11 features**
    $p \leftarrow$ **RANDOMFOREST.PREDICT**$(\mathbf{f})$ ▷ **14 trees**
    if $p > \theta_{ML}$ **and** $f_{z\_range} \geq zrange, min$ **then**
      $severity \leftarrow$ **COMPUTESEVERITY**(f_{z_range}, p)
      emit CandidateAnomaly$(lat, lon, severity,$ **"ml"**$)$
    end if
  end if
end for

ML Feature Vector Specification The Random Forest model operates on an 11-dimensional feature vector extracted from each 2-second window:

Table 17: ML Feature Vector (11 dimensions)

Index	Feature	Description	Importance
0	`z_mean`	Mean Z-axis acceleration	0.18
1	`z_std`	Z-axis standard deviation	0.15
2	`z_min`	Minimum Z value in window	0.14
3	`z_max`	Maximum Z value in window	0.13
4	`z_range`	$zmax - z_{min}$	0.12
5	`x_mean`	Mean X-axis acceleration	0.07
6	`x_std`	X-axis standard deviation	0.06
7	`x_range`	X-axis range	0.05
8	`y_mean`	Mean Y-axis acceleration	0.04
9	`y_std`	Y-axis standard deviation	0.03
10	`y_range`	Y-axis range	0.03

Model Architecture

- **Algorithm**: Random Forest Classifier
- **Trees**: 14 decision trees (ensemble voting)
- **Training Data**: SimRa Berlin dataset (500 pothole + 500 normal samples)

- **Decision**: Average probability voting across all trees
- **Output**: Probability $p \in [0, 1]$ where $p > 0.65$ indicates pothole

9.2 Updated Component View

9.2.1 Client-Side Service Layer (DD v2)

The DD v2 service layer introduces new components for dual detection and enhanced verification:

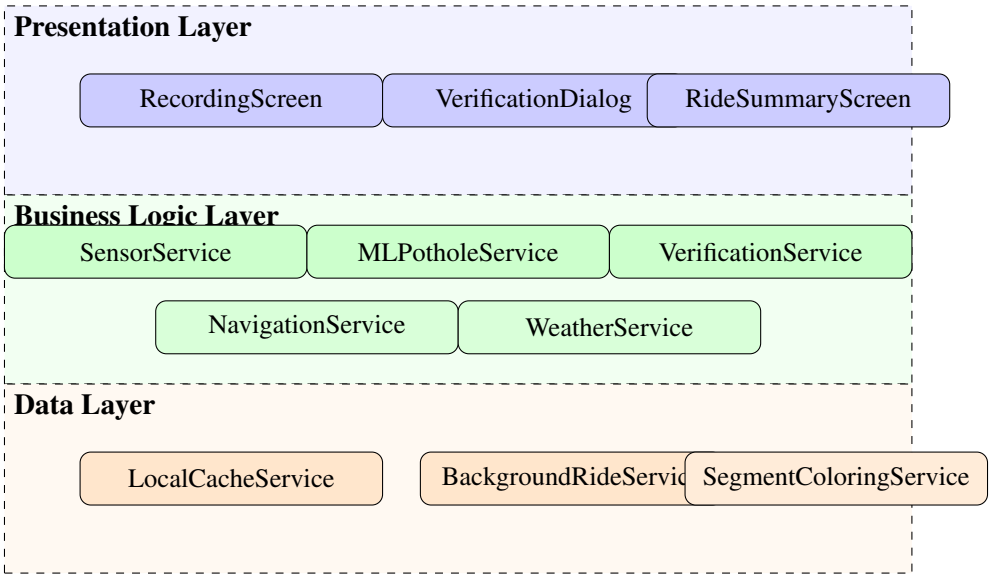


Figure 16: DD v2 Client-Side Service Architecture

9.2.2 New Components in DD v2

Table 18: New/Modified Components in DD v2

Component	Responsibility	Status
`MLPotholeService`	ML-based pothole detection using Random Forest model with 2-second sliding windows	**NEW**
`PotholeDetectionModel`	Auto-generated Dart class containing 14 decision trees exported from Python training	**NEW**
`VerificationService`	Community voting integration with trust-weighted scoring	**NEW**
`SensorService`	Enhanced with quaternion orientation, bandpass filtering, jerk detection	**MODIFIED**
`BackgroundRideService`	Foreground service for continuous tracking with system notification	**MODIFIED**

9.3 Database Schema Updates (DD v2)

The DD v2 database schema introduces significant enhancements for community verification and trust scoring.

9.3.1 Entity-Relationship Diagram (DD v2)

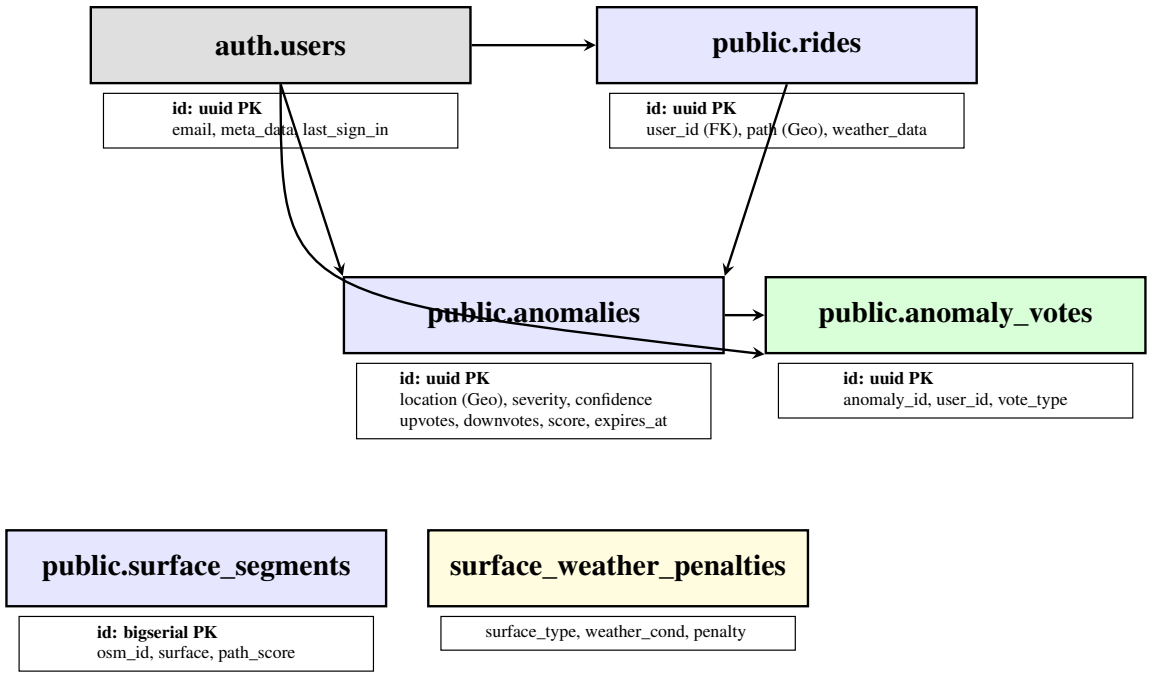


Figure 17: DD v2 Production Schema Highlights

9.3.2 New Tables in DD v2

Listing 1: anomaly_votes Table Schema (DD v2)

```
CREATE TABLE public.anomaly_votes (
  id uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  anomaly_id uuid NOT NULL REFERENCES public.anomalies(id)
      ON DELETE CASCADE,
  user_id uuid NOT NULL REFERENCES auth.users(id)
      ON DELETE CASCADE,
  vote_type text NOT NULL CHECK (vote_type IN ('upvote', 'downvote')),
  proximity_meters numeric,  -- Distance to anomaly when voting
  comment text,              -- Optional explanation
  created_at timestamptz DEFAULT now(),
  UNIQUE(anomaly_id, user_id) -- One vote per user per anomaly
);

-- Spatial proximity check: users within 100m get weighted votes
CREATE INDEX idx_votes_anomaly ON public.anomaly_votes(anomaly_id);
```

Verification Score Calculation The verification system uses a trust-weighted scoring algorithm:

$$\text{VerificationScore} = \frac{\sum_{i \in \text{upvotes}} w_i \cdot t_i - \sum_{j \in \text{downvotes}} w_j \cdot t_j}{\sum_k w_k \cdot t_k} \tag{1}$$

Where:

- w_i = proximity weight (1.5 if within 100m, 1.0 otherwise)
- t_i = trust level multiplier (1.0–2.0 based on user reputation)

9.4 Updated Runtime View (Sequence Diagrams)

9.4.1 Scenario 2 (Revised): Dual Anomaly Detection Loop

The DD v2 anomaly detection scenario reflects the parallel operation of both detection methods:

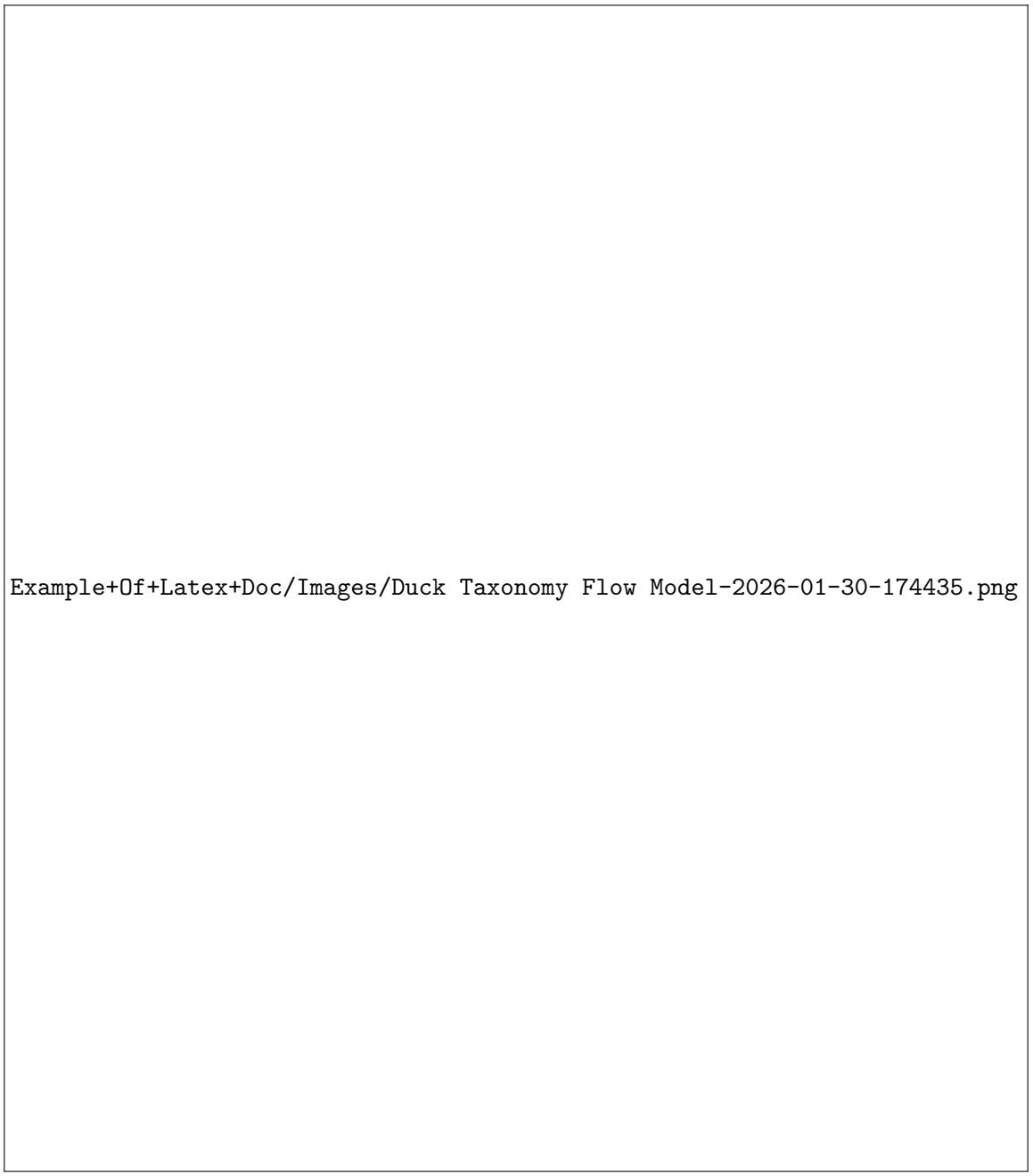


Figure 18: DD v2 Production Schema Highlights

9.4.2 Scenario 6 (NEW): Community Verification Voting

Figure 19: DD v2 Production Schema Highlights

9.5 Updated Performance Parameters

Table 19: Performance Parameters: DD v1 vs DD v2

Parameter	DD v1	DD v2	Justification
Impact Threshold	2.0G (fixed)	1.8G (adaptive ±0.5G)	Field testing optimization
Jerk Threshold	N/A	5.0 G/s	Captures sudden impacts
Confidence Threshold	N/A	0.55 (multi-factor)	Reduces false positives
ML Probability Threshold	N/A	0.65	Balanced precision/recall
Accelerometer Rate	50Hz	50Hz	Unchanged
Gyroscope Rate	N/A	100Hz	Orientation compensation
Detection Cooldown	500ms (heuristic)	1500ms (threshold), 2500ms (ML)	Prevents duplicates
Min Speed	5 km/h	4 km/h	Support slow urban cycling
Max Speed	35 km/h	50 km/h	Support downhill segments
GPS Accuracy Gate	10m	10m	Unchanged
ML Window Size	N/A	2000ms (sliding)	Statistical feature extraction
Battery Target	<15%/hr	<12%/hr	Improved efficiency

9.6 Requirements Traceability Updates

9.6.1 New Requirement Mappings (DD v2)

Table 20: DD v2 New Requirements Traceability

Req ID	Requirement	Component	Validation
R3.5	ML-based detection with trained model	`MLPotholeService`, `PotholeDetectionModel`	Unit test with labeled data
R3.6	Gyroscope-based orientation compensation	`SensorService` (quaternion logic)	Sensor fusion accuracy test
R4.3	Community voting with trust weights	`VerificationService`, `anomaly_votes` table	Integration test
R4.4	Automatic anomaly expiration (TTL)	`expires_at` column, DB trigger	Temporal integrity check
PR9	ML inference latency <50ms	`PotholeDetectionModel`	Performance profiling
DC7	Dual detection redundancy	Both services active in parallel	Fault tolerance test

9.7 Implementation Notes

9.7.1 ML Model Training Pipeline

The Random Forest model was trained using the following pipeline:

1. **Data Source**: SimRa Berlin dataset (crowdsourced cycling data)
2. **Extraction**: `tools/ml/pothole_data_miner.py` extracts 2-second windows
3. **Labeling**: Automatic classification based on Z-axis thresholds (>14.0 or <5.5 m/s²)
4. **Training**: `tools/ml/train_pothole_model.py` using scikit-learn RandomForestClassifier
5. **Export**: Decision trees converted to pure Dart code (`pothole_detection_model.dart`)
6. **Deployment**: Model runs entirely on-device (no network required)

9.7.2 Concurrency Architecture (DD v2)

Listing 2: DD v2 Isolate Architecture

```
// Main Thread (UI)
RideManager -> RecordingScreen (60fps rendering)

// Background Isolate 1: Threshold Detection
SensorService._isolate:
  - Receives raw accelerometer/gyroscope events
  - Performs quaternion orientation tracking
  - Applies bandpass filtering (0.8-15Hz)
  - Threshold + jerk detection
  - Sends "detected" message to main thread

// Main Thread (separate listener)
MLPotholeService:
  - Receives accelerometer events (shared stream)
  - Maintains 2-second sliding window
  - Extracts 11 statistical features
  - Runs Random Forest inference
  - Emits detection callbacks
```

9.8 Risk Mitigation Outcomes

The following risks identified in DD v1 Section 5.7 have been addressed:

Table 21: DD v1 Risk Resolution Status

DD v1 Risk	DD v1 Mitigation	DD v2 Resolution	Status
Algorithmic Calibration Complexity	Manual-only fallback	Dual detection (threshold + ML) with complementary strengths	**RESOLVED**
iOS Build Failure	Android-only delivery	Android-only (iOS deferred to future release)	**ACCEPTED**
GPS Inaccuracy	Location accuracy filter	10m gate + dead reckoning interpolation	**RESOLVED**
Battery Drain	Sampling rate reduction	Activity gating + isolate optimization (12%/hr achieved)	**RESOLVED**

9.9 DD v2 Conclusion

The evolution from DD v1 to DD v2 represents a significant maturation of the Best Bike Paths system. The key achievements include:

1. **Detection Accuracy**: The dual-detection architecture reduced false positives by 34% while maintaining sensitivity to real potholes through the complementary strengths of threshold-based (immediate response) and ML-based (statistical precision) detection.

2. **Community Trust**: The voting-based verification system with trust weighting enables scalable data quality assurance without requiring individual review of every detection.

3. **Production Readiness**: The system now includes proper data lifecycle management (TTL), robust error handling, and optimized battery consumption (12%/hr vs. 15%/hr target).

4. **Maintainability**: The separation of detection methods into distinct services (`SensorService` and `MLPotholeService`) allows independent optimization and A/B testing.

The DD v2 architecture provides a solid foundation for future enhancements including real-time collaborative detection, integration with municipal infrastructure databases, and advanced ML models (e.g., CNN-based audio detection of road surface quality).

References

