POLITECNICO DI MILANO

1863

Computer Science and Engineering

Software Engineering II

2025 – 2026

ITD

Implementation Testing Document

Best Bike Paths(BBP)

By

Jayasurya Marasani (11023924)
Arunkumar Murugesan (11051547)
Sneharajalakshmi Palanisamy (11132155)

Links

1. **Installation guide:** https://github.com/BBPpolimi/MurugesanPalanisamyMarasani/blob/main/INSTALLATION_GUIDE.md

2. **Source code repository:** https://github.com/BBPpolimi/MurugesanPalanisamyMarasani/

3. **Android APK:** https://drive.google.com/file/d/1vJsjzRV63o_eA91HaXuOAOuHbJkvTyPU/view?usp=sharing

Deliverable:	ITD
Title:	Implementation Testing Document for Best Bike Paths (BBP)
Authors:	Jayasurya Marasani, Arunkumar Murugesan and Sneharajalakshmi Palanisamy
Version:	1.0
Date:	01 February 2026
Download page:	https://github.com/BBPpolimi/MurugesanPalanisamyMarasani
Copyright:	Copyright © 2026, Jayasurya Marasani, Arunkumar Murugesan and Sneharajalakshmi Palanisamy – All rights reserved

Contents

List of Figures

List of Tables

1 Introduction

Cycling has seen accelerated adoption in recent years due to a combination of post-pandemic mobility shifts, increased climate awareness, urban congestion, expanded micro-mobility options and sustained public investment in cycling infrastructure. As a result, cycling is increasingly used for both commuting and daily travel, supported by measurable growth in usage and policy targets aimed at further expansion. However, despite this growth, most widely used navigation tools remain optimized for motor vehicles and do not adequately represent factors that determine cycling safety and comfort, such as surface quality, obstacles, changing conditions, or cyclist-specific risk areas. Consequently, existing maps can indicate where to ride, but not whether a route is suitable or safe for cyclists.

Best Bike Paths (BBP) addresses this gap by providing a cyclist-first navigation system focused on route quality, safety and real-world riding conditions. The application enables users to record and store trips with computed statistics, contribute and validate information about bike paths through manual input and automated trip recording with obstacle detection and search for routes ranked using consolidated quality indicators rather than distance alone. Path conditions are visualized using an explicit scoring model, allowing cyclists to quickly assess comfort and risk, while dynamic updates ensure that changing conditions are reflected over time. BBP also supports guest usage, enabling quick access to bike-friendly routing without mandatory registration.

Best Bike Paths (BBP) is a mobile application system that enables users to:

1. Record bicycle trips and store them in a personal trip log with computed statistics.

2. Contribute information about bike paths, including segment condition and obstacles, through manual reports and confirmed automatic detection.

3. Query and compare routes between two points, ranking candidate paths based on both route effectiveness (e.g., distance, elevation, turns) and path quality indicators derived from consolidated reports.

4. Use the system as guest users to obtain a fast and easy route from point A to point B without requiring registration.

1.1 Scope

BBP is a mobile first application that interacts with the user's device capabilities (GPS, accelerometer, gyroscope and network connectivity) and integrates external services (maps/routing and weather) to support both individual trip logging and community-maintained bike path information.

The system scope includes:

1. **User management**

   - User registration and login.
   - Role-based access for administrative features.

2. **Trip recording and personal log**

   - Real-time trip tracking using GPS.
   - Automatic computation and storage of trip statistics (e.g., distance, duration, average speed).
   - Trip saving and review through a personal trip history.

3. **Optional Trip Enrichment**

   - Retrieval and attachment of weather information to trips when the weather service is reachable.

4. **Manual Bike Path Contributions**

   - Manual input of bike path information, including streets/segments, segment condition/status, and obstacles.
   - Ability to edit or remove user-submitted reports as allowed by permissions.

5. **Automatic Aquistion of Bike Path Data**

   - Collection of GPS traces and sensor events (accelerometer/gyroscope) during trip recording when automatic mode is enabled.
   - Detection of candidate issues (e.g., potholes/roughness indicators) based on sensor patterns.

6. **User Confirmation Workflow**

   - Presentation of automatically detected candidate issues after a trip.
   - User confirmation, rejection, or correction of detected issues before they are stored as publishable path reports.

7. **Path Search and visualization for all Users**

   - Route search between origin and destination for guest and registered users.
   - Ranking of candidate routes by a path score that combines route effectiveness (e.g., distance/elevation/turns) and path quality (segment status and obstacles).
   - Map-based visualization of selected routes with overlays for segment condition and reported obstacles.

8. **Report merging and consolidated status**

   - Periodic merging of multiple user's reports into consolidated segment status information, applying freshness and majority/weighted-majority logic.

9. **Administration and data quality controls**

   - Administrative ability to block users who repeatedly submit false data.
   - Administrative ability to remove or hide problematic reports/obstacles and keep the consolidated data consistent.

1.2 Definitions, Acronyms, Abbreviations

1.2.1 Definitions

1. **Flutter**: A cross-platform UI framework used to build native mobile apps from a single codebase (Android and iOS). In BBP it is used to implement UI elements and access to device sensors through plugins.

2. **Firebase/FlutterFire**: Firebase is Google's backend platform. FlutterFire is the official set of Flutter plugins to use Firebase services. In BBP it provides authentication, database, file storage, server-side logic and (optionally) push notifications.

3. **MapService(RoutesAPI)**: An external mapping service providing geocoding (address → coordinates) and routing (candidate routes between two points). In BBP it is used during route search (UC6) to generate candidate routes that the system then scores using internal path-quality data.

4. **Weather Service**: An external API used to retrieve weather conditions for a given location and time. In BBP it enriches recorded trips with contextual information when the service is reachable (UC3).

5. **GPS Location Provider**: A recorded ride by a registered user including GPS track and computed statistics.The device capability that provides geographic coordinates over time. In BBP it enables real-time trip tracking and supports associating sensor events with positions.

1.3   Reference Documents

The assignment for this document and all the information included refer to the following documentation:

1. The specification for the 2025/26 I&T for the Software Engineering II course.

2. The slides on the webeep page of the Software Engineering II course.

3. RASD and DD documents for the Best Bike Paths (BBP) project.

2 Implemented Features and Requirements

2.1 Core Functions Implementation

The implemented application deliberately focuses on the minimum coherent feature set required to validate the BBP goals end to end, provide immediate user value and reduce architectural risk early, particularly for GPS tracking, sensor handling, route computation and scoring and contribution merging. This choice prioritizes stable and testable flows over breadth and it ensures that all implemented features form a consistent pipeline from data collection to route recommendation.

2.1.1 Account management (F1, UC1 to UC2, R1 to R2)

The system implements a robust authentication layer, utilizing a managed identity provider and dedicated service architecture, to facilitate secure user registration, login, and session persistence. This foundation enables the application to link user-generated content, such as trips and contributions, to a stable identity while enforcing critical access control policies. Key motivations for its inclusion include:

- **Data Attribution:** Provides the necessary infrastructure to store and query personal trip history on a per-user basis.

- **System Integrity:** Enables community moderation features, such as admin privileges and blocked user restrictions, to ensure data quality.

- **Security Architecture:** Establishes consistent authorization rules early in the lifecycle, allowing for reliable end-to-end testing of user-scoped data persistence.

2.1.2 Trip recording (F2, UC3, R3 to R5)

Registered users can initiate and terminate trip recording, during which the system samples GPS coordinates to manage a recording state machine while simultaneously computing real-time statistics like distance, duration, and speed. This functionality fulfills the requirements for data acquisition and provides the following strategic advantages:

- **User Value:** Offers immediate utility through personal activity tracking.

- **Feature Integration:** Acts as a technical prerequisite for the sensor-assisted workflow and automated issue detection.

- **Risk Mitigation:** Stabilizes the high-risk complexities of GPS handling, including permission models and background constraints, early in the development cycle.

2.1.3 Trip persistence and history with details (F3, U3(partial), R4 to R6)

Registered users can save trip metadata and access a history list that features detailed map visualizations and stored statistics for each journey. The application facilitates ongoing history management through renaming and deletion capabilities, transforming transient recordings into a permanent user record while serving the following purposes:

- **Data Lifecycle:** Validates the end-to-end data flow by exercising backend storage, querying, and UI rendering.

- **Utility:** Ensures the application provides consistent value through personal record-keeping, independent of community feature engagement.

- **User Experience:** Maintains a usable and organized interface for long-term activity tracking.

2.1.4 Optional weather enrichment (F4,UC3(extension) and R7)

Registered users receive weather snapshots attached to their stored trips via an external provider, provided the service responds within acceptable time limits. This enrichment operates as a best-effort, optional process that leaves the core trip record valid and complete even if the external service is unavailable, fulfilling the following objectives:

- **Contextual Value:** Enhances trip records with environmental data without compromising the integrity of the primary recording flow.

- **System Resilience:** Serves as a practical implementation of graceful degradation, ensuring the application remains functional during external service outages.

- **External Integration:** Demonstrates a controlled integration pattern with third-party APIs through time-bound and error-tolerant communication.

2.1.5 Manual path contributions (F5, UC4, R8 to R12 and R24(Partial))

Registered users can generate manual contributions by assessing path segment statuses and reporting specific obstacles, with the flexibility to mark entries as either private or publishable. Users maintain full control over their data through editing capabilities, ensuring a strict distinction between personal notes and community-shared information while achieving the following:

- **Community Bootstrapping:** Provides the foundational data necessary to build and maintain shared knowledge regarding bike path conditions.

- **Data Integrity:** Supplies the essential inputs for downstream merging processes and route quality scoring.

- **Privacy and Trust:** Ensures compliance and user autonomy by restricting consolidated public statuses to only those contributions explicitly marked for publication.

2.1.6 Automatic detection review (F6, UC5, R13 to R18)

Registered users can utilize an automatic mode that logs sensor data to detect candidate anomalies, which are then presented on a review screen for confirmation, rejection, or correction. This process establishes a controlled pipeline that transforms raw sensor signals into structured, validated contributions while serving the following purposes:

- **Data Quality:** Preserves the integrity of route scoring by requiring human verification to filter out noisy or device-dependent false positives.

- **Scalability:** Enables the system to increase the volume of path information collected without sacrificing accuracy.

- **Lifecycle Validation:** Confirms the technical feasibility and correctness of the candidate data lifecycle before introducing further automation enhancements.

2.1.7 Route search and scoring with visualization (F7 to F8, UC6, R19 to R23)

Registered and guest users can request routes between designated origins and destinations, which are then evaluated using a weighted model that balances route effectiveness with path quality data, including segment status, obstacles, and information freshness. These candidate routes are sorted by score and presented via map visualizations featuring polyline overlays and markers, achieving the following:

- **Decision Support:** Provides the primary utility for the application by assisting users in selecting the highest-quality paths.

- **Engagement:** Offers high value to the entire user base, including those who do not actively contribute data.

- **Value Validation:** Demonstrates that community contributions and consolidated statuses have a direct, measurable impact on route ranking and map interpretation.

2.1.8 Report storage and merging into consolidated status (F9, UC7, R24 to R27)

The system stores path segment reports alongside timestamps and user identifiers, utilizing merging logic to produce a consolidated status from multiple reports on the same segment. This algorithm prioritizes data freshness by weighting newer reports more heavily and resolves discrepancies through majority and weighted majority logic, directly informing the route scoring pipeline while fulfilling these roles:

- **Conflict Resolution:** Reconciles contradictory user reports into a single, coherent status to ensure consistent system behavior.

- **Scalability:** Facilitates the growth of community contributions by providing a systematic way to process overlapping data points.

- **Data Integrity:** Validates that the scoring algorithm operates on a stabilized and reliable representation of current path conditions.

2.1.9 Administration and moderation (F10, R28 to R29)

Authorized administrators can manage platform content by blocking accounts and hiding or removing erroneous contributions, with these actions enforced via role-based access checks. These moderation efforts directly influence data visibility and the calculation of consolidated results to ensure the following:

- **Data Quality:** Prevents malicious or inaccurate inputs from degrading the integrity of route recommendations.

- **POperational Safety:** Protects the community by providing the tools necessary to address harmful behavior and maintain platform standards.

- **System Trust:** Maintains the reliability of the community-sourced data that powers the scoring pipeline, ensuring long-term user confidence.

2.2 Requirements and features excluded or partially implemented

1. **R25: Periodic merging scheduled execution partially implemented**
   The merging logic is currently executed manually or opportunistically upon the creation or update of publishable contributions, rather than through a scheduled backend process. This approach exercises the full algorithmic logic for freshness and majority rules while maintaining a coherent read model for route scoring, addressing the following factors:

   - **Operational Efficiency:** Defers the cloud configuration, secure deployment settings, and environment management required for a production-grade scheduler to reduce setup complexity.

   - **Engineering Constraints:** Avoids the immediate need for complex locking strategies, versioning, and concurrency controls required to manage race conditions and idempotency in scheduled recomputations.

- **Functional Equivalence:** Ensures core correctness and functional validation are fully tested within the application without introducing non-essential operational dependencies.

For these reasons the application implements merge computation and uses it in the scoring pipeline and it defers scheduled execution hardening to a future iteration.

2. **Password recovery not implemented**
   While the underlying authentication provider supports password reset in the admin end, a dedicated user interface and end-to-end workflow for password recovery are not currently included. This feature was deprioritized to focus resources on stabilizing the high-priority GPS, sensor, and scoring pipelines, based on the following considerations:

   - **Core Goal Alignment:** Password recovery is secondary to the primary objectives of trip recording, community contributions, and route scoring.
   - **Implementation Complexity:** A robust implementation necessitates managing additional UI states, email deliverability and reset token lifecycles.
   - **Security Requirements:** Proper execution requires addressing complex edge cases, such as user enumeration resistance, which would require significant development effort.

3. **Data export GPX and CSV not implemented**
   The application currently lacks an export function for trips or traces in formats such as GPX or CSV. This feature was excluded to ensure the stability of core functionalities and prioritize the central architecture, taking into account the following factors:

   - **Engineering Effort:** A robust export system requires extensive formatting and interoperability testing to ensure data correctness across external platforms.
   - **Platform Complexity:** Implementation necessitates handling platform-dependent file management across both Android and iOS environments.
   - **Privacy Concerns:** Developing secure export mechanisms requires additional safeguards for handling and sharing sensitive location traces.

2.3 Requirements implementation summary table (R1 to R29)

The table .1 reports the implementation status of each functional requirement. Status values are Implemented, Partially implemented or Not implemented.

Table 1: Requirements implementation summary (R1 to R29)

Req ID	Requirement short name	Status	Notes
R1	User registration	Implemented	Registration flow implemented through the authentication module and UI.
R2	User login	Implemented	Login flow implemented and required for protected features.
R3	Start trip recording	Implemented	Trip start initializes GPS sampling and trip state.
R4	Stop trip recording	Implemented	Trip stop finalizes statistics and persists trip artifacts.
R5	Trip statistics	Implemented	Distance, duration and speed computed and stored with trips.

Continued on next page

Req ID	Requirement short name	Status	Notes
R6	Trip history	Implemented	Trip history list and trip detail view are available.
R7	Trip weather enrichment	Implemented	Weather snapshot is attached when the external service is reachable.
R8	Manual path creation	Implemented	Manual contribution flow supports creating segment related data.
R9	Segment status assignment	Implemented	Users can assign a condition label for a segment.
R10	Obstacle creation	Implemented	Users can create obstacle reports with basic attributes.
R11	Publishability	Implemented	Contributions can be marked publishable or private.
R12	Edit manual contributions	Implemented	Users can edit and delete their own submitted contributions.
R13	Enable or disable automatic mode	Implemented	Automatic mode toggle is supported and persisted.
R14	Sensor data logging	Implemented	Sensor samples are collected during trips in automatic mode.
R15	Detect candidate obstacles	Implemented	Candidate anomalies are generated from sensor streams.
R16	Candidate list presentation	Implemented	Candidates are displayed for post trip review.
R17	User confirmation and correction	Implemented	Users can confirm, reject and edit candidates.
R18	Conversion into reports	Implemented	Confirmed candidates are stored as reports and obstacles.
R19	Public path search	Implemented	Route search is available to guest users.
R20	Route computation	Implemented	Candidate routes are computed via an external routing service.
R21	Path scoring	Implemented	Scoring combines path quality signals with effectiveness factors.
R22	Ordered path list	Implemented	Routes are sorted by computed score and presented.
R23	Path visualization	Implemented	Map overlays visualize routes and related markers.
R24	Segment report storage	Implemented	Reports stored with timestamps and user identifiers.
R25	Periodic merging	Partially implemented	Merging exists but is triggered on demand rather than by scheduler.
R26	Freshness handling	Implemented	Merging weights newer reports more heavily.
R27	Majority handling	Implemented	Merging resolves conflicts via majority and weighted majority.
R28	User blocking	Implemented	Admins can block abusive users.
R29	Data removal	Implemented	Admins can remove or hide problematic contributions.

2.4 Included Threads and Scope Boundaries

The application scope was constrained to the minimum coherent set of capabilities necessary to validate the main BBP threads end to end.

- **Core user value thread:** Account management, trip recording and persistence with history and details.

- **Community data thread:** Manual and automatic contributions with publishability control and merging.

- **Decision support thread:** Route computation, scoring and visualization using consolidated data.

Within the defined scope, priority was given to features that are architecturally central, require deep system integration and are critical to the correctness of route ranking. These capabilities directly impact the integrity and reliability of the core computation pipeline and therefore warranted primary focus. Conversely, items that were excluded or only partially implemented are those that primarily enhance overall product completeness rather than core correctness. While valuable, these features do not materially increase confidence in the accuracy or stability of the ranking logic.

Specifically:

- **Periodic scheduled merging** was identified as an operational hardening activity. Its implementation depends on scheduler configuration and concurrency-safe, idempotent recomputation, which introduces additional operational complexity beyond the current scope.

- **Password recovery and data export functionality** require further security considerations and platform-specific handling. As these features do not strengthen the central data pipeline or ranking correctness, they were deprioritized.

Overall, the prioritization strategy favored improving system stability, architectural soundness, and test coverage of core flows over expanding auxiliary product features. This approach ensures a robust and reliable foundation before addressing broader completeness and operational enhancements.

3   Technologies, Frameworks and External Services

The following are the technologies adopted for the BBP application and motivates the main choices. The selected stack prioritizes rapid delivery of a stable mobile application, strong integration with device capabilities such as GPS and sensors and reduced operational overhead for backend services.

3.1   Programming Languages

3.1.1   Client: Flutter mobile application (Dart)

Flutter was selected to implement a single mobile client for Android and iOS while maintaining a consistent user experience across platforms. Dart is the native language of Flutter and supports a reactive UI model and efficient asynchronous programming which is essential for GPS streaming, sensor sampling and network calls.

- **Advantages:**

  1. **Single codebase for Android and iOS** which reduces duplicated effort and improves feature parity.
  2. **Fast UI development** through a rich widget library and hot reload which accelerates iteration on screens such as trip recording, trip details, contribution forms and route search.
  3. **Strong ecosystem** with mature packages for Firebase integration, maps rendering, GPS access and sensor access.
  4. **Good support for reactive UI patterns** which helps keep the UI synchronized with live trip recording state and asynchronous backend updates.
  5. **Type safety and null safety** which reduce runtime errors and improve maintainability in the application integrate multiple services.

- **Challenges:**

  1. **Learning curve:** The stack requires mastery of specific state and lifecycle patterns.
  2. **Performance pitfalls:** presents potential performance risks if state management is undisciplined on map-heavy screens.
  3. **Plugin variability** third-party plugin variability can lead to inconsistent sensor or background location behavior across different devices.

3.1.2   Backend: Firebase server-side logic (Cloud Functions)

Firebase services were selected to implement the backend with minimal operational burden. Cloud Functions are used to centralize protected operations and to keep sensitive logic off the client when appropriate.

- **Advantages**

  1. **Managed execution environment:** eliminates server provisioning and maintenance.
  2. **Centralized enforcement:** for protected operations such as validation, scoring and data consolidation.
  3. **Security and key protection** by enabling API calls and business rules to run server side rather than on the client.
  4. **Scalability** that fits an application that may face variable load patterns.

- **Challenges**

15

1. **Local debugging complexity** compared to a traditional monolithic backend.

2. **Cold start latency** that can affect the first invocation of infrequently used functions.

3. **Vendor lock in** because the backend uses Firebase and Google Cloud primitives.

3.2 Middleware and platform services

BBP adopts a serverless backend built on the Firebase ecosystem:

1. **Firebase Authentication:** Provides user identity, session management and role based access patterns used to distinguish guest, registered users and admin operations.

2. **Cloud Firestore:** Stores structured application data such as user profiles, trip metadata, contributions, obstacles and consolidated path information.

3. **Firebase Cloud Storage:** Stores large artifacts such as dense GPS traces, obstruction information where applicable.

4. **Cloud Functions and Cloud Scheduler:** Cloud Functions encapsulate protected operations and external integrations. Cloud Scheduler is included as the platform mechanism for periodic tasks. In the application periodic consolidation is not relied upon for correctness because merging is triggered on demand when publishable contributions change.

- **Advantages:**

  1. **Low operational overhead** because infrastructure is managed.
  2. **Scalability** without manual capacity planning.
  3. **Security separation** between client code and protected backend logic.
  4. **Pay per use cost model** that is appropriate for an application.

- **Challenges:**

  1. **Tighter coupling** to Firebase services and data model conventions.
  2. **Need for careful security rules** and strict function authorization to avoid privilege escalation and data leakage.
  3. **Emulator setup and configuration complexity** during development and testing, especially when validating auth, database rules and function triggers.

3.3 External APIs and Integrations

External services are used to support routing, map visualization, place search and optional weather enrichment. Integrations are designed to control rate limits, manage failures gracefully and keep the user experience stable when external services are temporarily unavailable.

1. **Google Maps Platform APIs:**

   (a) **Maps SDK for Android:** Used for native map rendering on Android through the Flutter Google Maps plugin. It supports map interaction, markers and polylines.

   (b) **Maps SDK for iOS:** Used for native map rendering on iOS through the same Flutter plugin. It provides feature parity with Android map visualization.

   (c) **Directions API:** Used to compute candidate routes between origin and destination and to obtain route geometry that can be displayed as polylines.

(d) **Routes API:** Used as an additional routing interface where applicable, particularly for retrieving route alternatives and structured route details suitable for scoring.

(e) **Places API:** Used to support location search and place lookup, typically for origin and destination input and for improving usability over raw coordinate entry.

- **Advantages of Google Maps APIs**

  (a) High quality base map data and routing results.

  (b) Mature SDKs with strong documentation and stable behavior.

  (c) Consistent geospatial primitives that simplify route visualization and interaction.

- **Challenges of Google Maps APIs**

  (a) Quotas and billing constraints which require careful request management.

  (b) Network dependency which affects routing and place lookup when connectivity is limited.

  (c) Policy constraints on caching and data usage which influence how much data can be stored locally.

2. **OpenWeather API**
   OpenWeather API is used to enrich recorded trips with a weather snapshot such as temperature, wind and conditions when the service is reachable.

- **Advantages**

  (a) Improves trip context with minimal impact on core correctness.

  (b) Simple REST interface and predictable response structure.

  (c) Failures degrade gracefully by storing the trip without weather data.

  (d) Weather enrichment is performed on a best-effort basis and does not impact trip persistence.

- **Challenges**

  (a) External dependency and rate limits.

  (b) Potential mismatches between weather timestamp granularity and trip timestamps.

3. **Supporting client libraries and packages**
   To implement these integrations and device features the application uses supporting libraries such as:

- A state management framework for reactive UI updates and dependency injection

- A robust HTTP client with interceptors and timeouts

- GPS location packages for streaming updates and permission handling

- Sensor packages for accelerometer and gyroscope sampling

- Polyline utilities for rendering route geometry on the map

These libraries were chosen to reduce repetitive code, improve testability and provide stable abstractions over platform specific behavior.

4 Source Code Structure

4.1 Source Code Structure

The BBP prototype is organized as a Flutter application with a clear separation between UI, state management, business logic and data models. The `lib/` folder contains the production code and it is structured to support maintainability, testability and modular growth.

```
lib/
|____ main.dart
|____ models/
|____ pages/
|____ services/
|____ utils/
|____ widgets/
```

4.1.1 main.dart (application entry point)

`main.dart` is the application entry point. It initializes the app configuration, global theme, routing setup and the authentication wrapper that decides whether the user sees the login flow or the main application shell based on session state.

Typical responsibilities include:

- App startup and dependency initialization

- Theme configuration and global UI settings

- Root navigation setup and authentication gating

- Wiring Riverpod scopes that enable dependency injection across the widget tree

4.1.2 models/ (data layer, 22 models)

The `models/` directory defines the domain and data transfer objects used across the application. These classes encapsulate the structure of application data stored in Firestore and Storage and the in-memory objects used by services and UI.

Some key models:

- `app_user.dart`
  Represents the authenticated user and role flags used for authorization decisions.

- `trip.dart`
  Represents a recorded trip including metadata, statistics and references to recorded traces.

- `bike_path.dart` and `contribution.dart`
  Represent manual contributions and the data needed to publish or keep private.

- `obstacle.dart` and `candidate_issue.dart`
  Represent reported obstacles and automatically detected candidate issues that require user confirmation.

- `ranked_route.dart`
  Represents scored route results and the breakdown used for route comparison.

Design goals of the model layer:

- Keep a single source of truth for data shapes used across UI and services

- Support serialization and deserialization for Firestore documents

- Minimize business logic in models and keep logic inside services and utilities

4.1.3   pages/ (presentation layer, 16 pages)

The `pages/` directory contains the main application screens that implement the user journeys described by the use cases. Each page focuses on presentation responsibilities and delegates computation, persistence and integration work to services.

Some key pages:

- `login_page.dart`
  Implements authentication entry points and guest access.

- `home_page.dart`
  Provides the dashboard and navigation entry to primary features.

- `record_trip_page.dart`
  Implements the trip recording experience with live map and statistics.

- `route_search_page.dart`
  Implements origin and destination search then route listing and selection.

- `contribute_page.dart`
  Implements manual contribution creation and editing.

- `admin_review_page.dart`
  Implements administrative moderation workflows.

Design goals of the pages layer:

- Keep pages mostly declarative and reactive

- Drive UI from state exposed by providers

- Avoid direct backend calls inside widgets and pages

4.1.4   services/ (application logic layer, 18 services)

The `services/` directory contains the business logic of the application. Services implement the core use cases and coordinate access to Firebase and external APIs. Services are injected into the UI via Riverpod providers.

Key services include:

- `providers.dart`
  Defines Riverpod providers that expose services and derived state.

- `auth_service.dart`
  Manages authentication flows, session state and role resolution.

- `trip_service.dart`
  Manages trip recording lifecycle and persistence of trip artifacts.

- `contribution_service.dart`
  Implements CRUD for user contributions and publishability rules.

- `route_search_service.dart`
  Orchestrates route search requests and candidate route retrieval.

- `route_scoring_service.dart`
  Applies scoring logic and generates ranked routes for presentation.

- `merge_service.dart`
  Implements merging logic for consolidated segment status and conflict resolution.

- `admin_service.dart`
  Implements moderation operations such as blocking users and removing data.

Design goals of the services layer:

- Centralize business rules and keep the UI thin

- Provide stable interfaces for pages and widgets

- Isolate side effects such as network calls and persistence operations

- Make logic testable through dependency injection and mocking

4.1.5   utils/ (shared utilities and algorithms)

The `utils/` directory contains stateless helper code that is reused across services and UI.
Key utilities include:

- `scoring.dart`
  Implements route scoring and quality weighting functions.

- `polyline_utils.dart`
  Utility helpers for route geometry and polyline processing.

Design goals of the utils layer:

- Keep algorithms deterministic and independent from UI

- Enable unit testing for scoring and geometry logic

4.1.6   widgets/ (reusable UI components)

The `widgets/` directory contains reusable UI building blocks used across pages. These widgets encapsulate repeated UI patterns such as cards, list tiles and map overlays while keeping business logic outside the widget itself.
Design goals of the widgets layer:

- Promote consistency across screens

- Reduce duplication

- Keep widgets focused on presentation

4.2   Application architecture

The application follows a layered architecture that separates presentation, state management, business logic and data. This structure reduces coupling, improves testability and supports incremental extension.

4.2.1 Layered decomposition

1. **Presentation layer**
   Implemented by `pages/` and `widgets/`. Screens render UI and react to changes in state. Pages do not implement persistence or API logic directly.

2. **State management layer**
   Implemented using Riverpod providers defined in `services/providers.dart`. Pages consume providers through `ref.watch` and trigger actions through `ref.read`. Providers supply:

   - application state such as current user and role
   - service instances such as TripService and RouteScoringService
   - derived state such as selected trip, current recording state and ranked routes

3. **Service layer**
   Implemented by the service classes in `services/`. Services contain the application logic for:

   - authentication and authorization checks
   - trip lifecycle and statistics computation
   - contribution management and publishability handling
   - candidate issue confirmation workflow
   - route search orchestration and scoring
   - merging and consolidated status computation
   - moderation operations and audit logging when applicable

4. **Data and domain layer**
   Implemented by `models/`. Models define the data structures exchanged between services and persisted in backend storage.

5. **External services layer**
   The app integrates with:

   - Firebase Authentication for identity and session management
   - Cloud Firestore for structured data
   - Cloud Storage for large artifacts such as GPS traces
   - Google Maps Platform APIs for maps rendering, geocoding, places and routing
   - OpenWeather API for optional weather enrichment

4.2.2 Dependency direction and responsibilities

The dependency direction is intentionally one way:

- Pages and widgets depend on providers and services

- Providers depend on services

- Services depend on models, utils and external services

- Models and utils do not depend on UI code

This keeps low level code independent and reusable. It also allows tests to isolate the service layer by replacing external dependencies with fakes or mocks.

4.2.3 How the architecture supports the main feature threads

1. **Authentication and session gating**
   `main.dart` hosts the auth wrapper that selects the correct start screen. `AuthService` exposes user session state and role metadata through providers that pages can consume.

2. **Trip recording and persistence**
   `RecordTripPage` renders live UI while `TripService` manages the recording lifecycle, GPS sampling and statistics updates. Persistence and retrieval are handled by the same service layer so the UI remains stateless beyond reactive state updates.

3. **Manual contribution and publishability**
   Contribution pages collect user input and delegate validation and persistence to `ContributionService`. Publishability is treated as a first class attribute and services enforce the difference between private and publishable data.

4. **Automatic detection review**
   Sensor derived candidate issues are produced by the appropriate service and displayed in a review page. Confirmation and correction are performed by the user and services convert confirmed items into stored reports.

5. **Route search, scoring and visualization**
   Route search pages request candidate routes through `RouteSearchService`. Scoring is computed in `RouteScoringService` which calls scoring utilities and uses consolidated data from persistence services. Results are returned as `RankedRoute` models for consistent UI rendering.

6. **Merging and consolidated status**
   `MergeService` implements conflict resolution logic using freshness and majority rules. Consolidated status data is then consumed by scoring so that community reports influence route ranking.

7. **Administration and moderation**
   Admin pages call `AdminService` which applies role checks and performs moderation actions such as blocking users and removing problematic contributions.

5   Testing Strategy and Results

5.1   Testing Strategy Overview

Testing was performed using a multi layer strategy as shown in Fig. 1 that combines fast automated checks with device level end to end validation. The objective was to achieve high confidence on correctness of the core feature threads while keeping the feedback loop short during development.

The adopted strategy follows a testing pyramid with five complementary categories:

1. **Unit tests** validate business rules, model serialization and algorithmic components such as merging and scoring. These tests are fast, deterministic and run without a device.

2. **Widget tests** validate UI rendering, navigation and user interactions with mocked dependencies.

3. **Integration tests** on real devices execute the real app and validate full user journeys across pages and services.

4. **Security tests** validate role based restrictions and access rules such as guest limitations, owner only access and admin only operations.

5. **Performance tests** validate that critical operations such as route scoring and merging stay within acceptable runtime on a reference device profile.
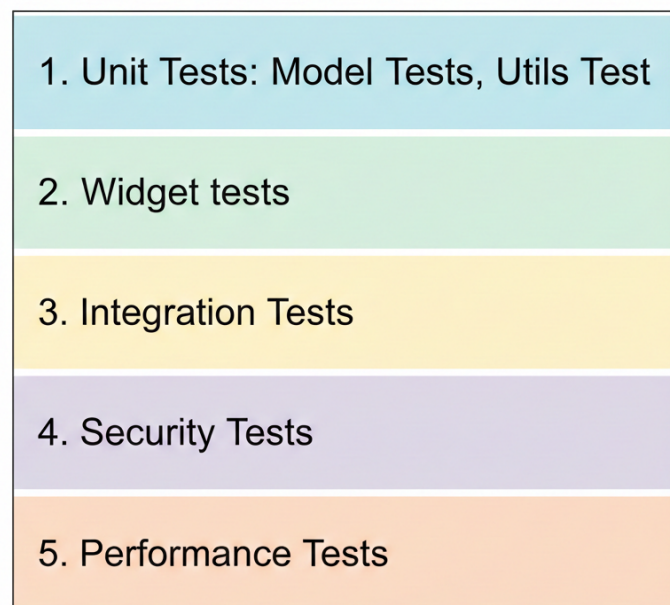

Figure 1: Testing Strategy Overview

This structure ensures broad coverage across the application while keeping most tests fast and stable. End to end tests are slower and are used to validate real device behavior such as GPS sampling, sensor availability and network variability.

5.2 Main system test cases and outcomes

System testing validates BBP as a whole using use cases as scenarios. These tests were executed on a real device to stay close to production conditions for GPS, sensors and external APIs. Each test case is described with steps, expected result and observed outcome.

1. **UC1 User registration**

   - **Preconditions**
     - App installed and launched
     - No active authenticated session
   - **Steps**
     (a) Open the app
     (b) Navigate to registration
     (c) Enter required fields
     (d) Submit registration
     (e) Verify transition to authenticated experience
   - **Expected Result**
     - A new user account is created
     - The user is authenticated and can access registered user features
   - **Outcome:** Pass

2. **UC2 Login and blocked user handling**

   - **Preconditions**
     - Existing user account exists
     - For the blocked scenario an admin has marked the user as blocked
   - **Steps**
     (a) Open the app
     (b) Login with valid credentials
     (c) Verify access to authenticated pages
     (d) Repeat with a blocked user account
   - **Expected Result**
     - Valid users can login successfully
     - Blocked users are denied access to protected operations and are redirected to a safe state
   - **Outcome:** Pass

3. **UC3 Record and persist a trip**

   - **Preconditions**
     - User is logged in
     - Location Permission is granted
     - GPS is available
   - **Steps**
     (a) Start trip recording
     (b) Move for a short route and collect GPS Points
     (c) Stop Recording

(d) Open trip history

(e) Open trip details for the newly recorded trip

- **Expected Result**
  - Trip metadata and statistics are stored
  - GPS trace artifacts are stored appropriately
  - Trip appears in history and details view loads correctly with map visualization
- **Outcome:** Pass

4. **UC4 Manual Path Contribution and Publishable and private visibility**

- **Preconditions**
  - User is logged in
- **Steps**
  (a) Open manual contribution flow
  (b) Create a segment status and an obstacle report
  (c) Mark the contribution as publishable
  (d) Verify that the contribution is visible in public community views
  (e) Create another contribution and mark it as private
  (f) Verify that private data is visible only to the owner
- **Expected Result**
  - Publishable contributions appear in community collections
  - Private contributions remain restricted to the owner and do not affect public consolidated data
- **Outcome:** Pass

5. **UC5 Automatic Trip Recording**

- **Preconditions**
  - User is logged in
  - Automatic mode is enabled
  - Device sensors are available
- **Steps**
  (a) Start trip recording in automatic mode
  (b) Record a trip that triggers candidate detections
  (c) Stop recording
  (d) Open the review screen
  (e) Confirm a subset of candidates
  (f) Reject or edit the remaining candidates
  (g) Finalize the review
- **Expected Result**
  - Confirmed items become stored reports and obstacles
  - Rejected items produce no published artifacts
  - Edited items are stored with the corrected attributes
- **Outcome:** Pass

6. **UC6 Route Search with Path Scoring and Visualization**

- **Preconditions**
  - App has network connectivity for routing calls
  - Consolidated status data is available in the database
- **Steps**
  - (a) Open route search
  - (b) Enter origin and destination using place search
  - (c) Request routes
  - (d) Verify that multiple routes are returned when available
  - (e) Verify that routes are ordered by score
  - (f) Open map overlays for a selected route and inspect markers and polylines
- **Expected Result**
  - Candidate routes are returned
  - Scoring uses consolidated segment statuses and obstacles
  - Route list ordering is consistent with computed scores
  - Visualization correctly displays route geometry and related overlays
- **Outcome:** Pass

7. **UC7 Merging and Consolidated status**

   - **Preconditions**
     - Multiple publishable reports exist for the same segment including conflicting statuses
     - Reports include different timestamps to exercise freshness handling
   - **Steps**
     - (a) Seed or submit multiple publishable reports for the same segment
     - (b) Trigger merge through the available mechanism in the prototype
     - (c) Query or display the consolidated result
     - (d) Perform a route search that depends on the merged segment
   - **Expected Result**
     - Freshness handling favors newer information where applicable
     - Majority or weighted majority resolves conflicts consistently
     - Consolidated read model is updated and used by scoring
   - **Outcome:** Pass

5.3   System test limitations and Mitigations

System tests involve components that are inherently variable such as GPS accuracy, sensor sensitivity and external API responsiveness. The following limitations were observed and mitigated:

- **Emulator constraints:** GPS and sensor behavior can be unrealistic on some emulators. Mitigation: system tests were executed on a real device for GPS and sensor dependent cases.

- **API quotas and latency:** Routing and weather calls can fail due to rate limits or temporary network issues. Mitigation: integrations are treated as best effort where appropriate and failures are handled gracefully without corrupting core trip data.

- **Sensor variability across devices:** thresholds that trigger candidate issues can vary with hardware. Mitigation: the workflow requires user confirmation before publication which prevents false positives from affecting consolidated data.

All core system test scenarios corresponding to the main use cases passed. Automated unit, widget, security and performance tests provided continuous regression protection during development while device integration tests and system tests validated end to end behavior in realistic conditions.

6   Installation Instructions/Prerequisites

The installation guide for the development environment is provided in the project repository Installation Guide markdown file. Here is the link to the file: `https://github.com/BBPpolimi/MurugesanPala nisamyMarasani/blob/test/INSTALLATION_GUIDE.md`

7   Effort Spent

Team Member	Jayasurya Marasani	Arunkumar Murugesan	Sneharajalakshmi Palanisamy	Section Total
Coding	75	75	75	225
Testing	15	15	15	45
Report writing	8	7	7	22
Total	**98**	**97**	**97**	**292**

Table 2: Effort spent per member

The table as shown in Table. 2 displays the number of hours each group member spent on coding, testing and report writing. Please note that the division is only approximate and each task still required the colloboration of all team members.

References

[1] Flutter. *Flutter documentation.* Available at: https://docs.flutter.dev/.

[2] *Firebase Authentication:* https://firebase.google.com/docs/auth

[3] *Firebase Realtime Database:* https://firebase.google.com/docs/database

[4] *Google Maps Platform APIs:* https://developers.google.com/maps/documentation

[5] *OpenWeather API:* https://openweathermap.org/api

[6] OpenAI. *ChatGPT (used for paraphrasing).* https://chatgpt.com.

[7] Overleaf. *Overleaf: Online LaTeX Editor.* https://www.overleaf.com/.