POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Software Engineering II

Implementation Document

Project: Best Bike Paths

Authors: Ianosel Bianca Roberta, Gholami Vajihe, Errigo Simone
Version: 1.0
Date: 01.02.2026
Project Link: Errigo-Gholami-Ianosel (github.com)

Contents

1 | Introduction

1.1. Purpose

1.2. Scope

1.3. Definitions, Acronyms, Abbreviations

1.3.1. Definitions

1.3.2. Acronyms

- **API:** Application Programming Interface.
- **APK:** Android Package
- **DBMS:** DataBase Management System.
- **DD:** Design Document.
- **DOM:** Document Object Model.
- **DTO:** Data Transfer Object, represents a link between the user input and a Java Object.
- **HTTP:** HyperText Transfer Protocol.
- **IPA:** iOS App Store Package.
- **JPA:** Java Persistence API.
- **JS:** JavaScript.
- **QR Code:** Quick Response Code.
- **REST:** REpresentational State Transfer (see DD).
- **RASD:** Requirements Analysis and Specification Document.

- **S2B:** Software To Be.
- **UI:** User Interface.
- **URL:** Uniform Resource Locator.
- **UX:** User eXperience.
- **ORM:** Object-Relational Mapping.

1.3.3. Abbreviations

- **something:**

1.4. Revision History

- Version 1.0 (01 February 2026);

1.5. Document Structure

Mainly the current document is divided into six chapters:

1. **Introduction:** provides an overview of the document, outlining its purpose, scope, and relevance to the project.
2. **Implemented Functionalities and Requirements:** details the functionalities and requirements that have been implemented in the project.
3. **Adopted Development Frameworks:** describes the development frameworks utilized in the project, explaining their roles and benefits.
4. **Source Code Structure:** outlines the organization and structure of the source code, facilitating understanding and navigation.
5. **Testing Strategy:** presents the testing methodologies and strategies employed to ensure the quality and reliability of the software.
6. **Installation Instructions:** provides step-by-step guidance on how to install and set up the software.
7. **References:** lists the references and resources used in the creation of the document and the project.

8. **Effort Spent:** details the distribution of work and time spent by each team member throughout the project.

2 | Implemented Functionalities and Requirements

2.1. Product Functions

This section describes the core functionalities of the app, organized to support the main goals and requirements defined for the project.

2.1.1. User Access and Identity Management

The app allows users to access the system either as guests or as authenticated users, with different levels of interaction. Guest users can explore bike paths, visualize ranked routes between two locations, start trips, and access general information. Their interaction is limited, as they cannot access restricted areas of the app, submit or validate reports, and any data generated during their usage is not saved.
Guest users can register at any time by providing basic identification information. After logging in, users gain access to the full set of functionalities, including viewing previously recorded trips, accessing personal statistics, managing created bike paths, and contributing to the system by submitting or validating reports. Authenticated users can also access and modify their personal profile information.

2.1.2. Trip Recording and Management

Users can start cycling trips both as guests and as authenticated users, but any data generated during the activity as a guest user is not saved once the trip ends.
Logged-in users can record and manage their cycling trips with full data persistence. A trip represents a single cycling activity and includes information such as duration, distance, and the route followed. Additional contextual information, such as weather conditions, may also be retrieved to provide further context to the recorded activity. When the trip ends, all collected data is saved and becomes part of the user's trip history.

2.1.3. Path Discovery, Creation and Selection

The app allows users to search and explore bike paths. Users can select an origin and a destination, and the app suggests one or more bike paths connecting the two locations. Suggested paths are ordered according to their overall quality and suitability, based on the information available in the system. Users can select a specific path and view its details, including its condition, length, and any associated reports.
Logged-in users can also create new bike paths to extend the set of available routes. Paths can be created by drawing the route directly on the map. When creating a path, users can choose whether it should remain private or be shared with the community. Either way, the created path is stored in the system and can be accessed by the user.

2.1.4. Map-based Visualization and Navigation Support

The app uses an interactive map as the main interface to display bike paths and trips. Suggested paths, selected routes, and recorded trips are shown directly on the map, allowing users to easily understand the layout and characteristics of each route.
While navigating, the app shows the user's current position along the selected path, making it easy to track progress in real time. The map may also display reports, providing useful contextual information while the user is moving.

2.1.5. Report Submission and Confirmation

Logged-in users can contribute information about bike paths conditions by submitting reports. Reports describe obstacles, anomalies, or the overall condition of a path segment and can be created manually by the user during an active trip. In all cases, reports are linked to a specific path.
To improve the reliability of the collected information, the app allows users to confirm or reject existing reports, while cycling. User feedback helps reduce false positives and outdated data, making the reported information more accurate over time.

2.1.6. Path Condition Evaluation and Ranking

The app evaluates the condition of bike paths by combining information collected from user reports and recorded trips. Each path is associated with indicators that describe its current condition, taking into account reported issues, detected obstacles, confirmations provided by multiple users over time and their freshness. This allows the app to keep an updated and reliable view of path quality.

Path Status Scoring Model

Each report submitted by a user refers to a specific path condition, represented as a status. To enable aggregation and quantitative reasoning, each status is mapped to a numerical score as follows:

Path Status	Numerical Score
Optimal	5
Medium	4
Sufficient	3
Requires Maintenance	2
Closed	1

Table 2.1: Mapping of path status to numerical scores

Report Freshness Model

Since real-world path conditions evolve over time, the algorithm assigns a freshness weight to each report in order to prioritize recent information. For a report i, freshness is computed as:

$$fresh_i = 2^{-\frac{ageMin_i}{H_{min}}}$$

where $ageMin_i$ represents the age of the report in minutes, computed as the difference between the current time and the report confirmation timestamp. This exponential decay ensures that older reports progressively lose influence over the aggregation process.

Report Validation Contribution

Reports may undergo validation by other users and can be confirmed or rejected. Only validated interactions contribute to the reliability of a report. For each report, two partial scores are computed:

$$confirmedScore = \sum fresh_i \quad \text{for each confirmation}$$

$$rejectedScore = \sum fresh_i \quad \text{for each rejection}$$

Reports in the *IGNORED* state do not contribute to either score, while those in the *CREATED* state contribute to the confirmed score.

Report Reliability Computation

The overall reliability of a report is computed by combining confirmation and rejection scores through a weighted difference:

$$reportReliability = clamp(1 + \alpha \cdot confirmedScore - \beta \cdot rejectedScore,\ min,\ max)$$

where α and β are weighting parameters controlling the impact of confirmations and rejections respectively, and min and max define lower and upper bounds for reliability. In the current configuration, $\alpha = 0.6$, $\beta = 0.8$, $min = 0.1$, and $max = 2.5$.
Reports whose reliability falls below a minimum threshold are excluded from further aggregation, as they are considered no longer representative of the current path condition.

Path Status Aggregation

The overall status score of a path is computed as a weighted average of the scores of all associated reports:

$$PathStatusScore = \frac{\sum(statusScore_i \cdot reportReliability_i)}{\sum reportReliability_i}$$

Path Status Determination

The final numerical score is mapped back to a discrete path status according to predefined thresholds:

Path Status	**Score Range**
Optimal	$[4.5, 5]$
Medium	$[3.5, 4.5)$
Sufficient	$[2.5, 3.5)$
Requires Maintenance	$[1.5, 2.5)$
Closed	$[1, 1.5)$

Table 2.2: Mapping of numerical scores to discrete path statuses

Impact on Path Ranking

The computed path status directly influences the ranking of paths during route discovery. Paths with higher evaluated conditions are prioritized when suggesting routes between an origin and a destination. Since the merging algorithm is continuously updated as

new reports and validations are received, path rankings dynamically adapt to evolving real-world conditions.

2.1.7. Statistics Computation and Caching

The app computes statistics to help users better understand their cycling activities and overall performance. Statistics are generated from recorded trip data.
To keep the system efficient, the app avoids unnecessary recomputations. Aggregated statistics are updated only when new trips are recorded, while previously computed results are reused whenever possible. Per-trip statistics are generated when needed and then stored, so they do not have to be recalculated every time they are accessed. As a result, users can access up-to-date statistics without affecting the overall performance of the app.

2.1.8. Weather Data Acquisition and Enrichment

The app retrieves weather information from external meteorological services to provide additional context for cycling activities. Weather data, such as temperature, wind, and other relevant conditions, is associated with recorded trips based on the time and location of the activity.
Weather information is collected when a trip is completed, ensuring that the recorded conditions accurately reflect the environment in which the activity took place. This allows users to better understand their trips and interpret performance data in relation to external factors.

2.2. Requirements

Rx	Description	Implemented
R1	The system shall allow guest users to create an account by providing personal information and credentials.	Yes
R2	The system shall allow registered users to log into the application using valid credentials.	Yes
R3	The system shall allow logged-in users to view their profile and account settings.	Yes
R4	The system shall allow logged-in users to update their profile and account settings.	Yes

Continued on next page

Rx	Description	Implemented
R5	The system shall allow logged-in users to log out of the application, ending their current session.	Yes
R6	The system shall allow guest users to start a cycling trip.	Yes
R7	The system shall allow guest users to stop a currently active trip, but shall not store any trip data after the trip ends.	Yes
R8	The system shall allow the user to start a trip only when their GPS position matches the path origin.	Yes
R9	The system shall display a pop-up suggesting to start a trip when cycling is detected while no trip is active and the app is open.	TODO
R10	The system shall set the current GPS position as trip origin when starting from auto-detection.	TODO
R11	The system shall automatically stop the active trip when the user's GPS position deviates from the selected path within a certain threshold.	TODO
R12	The system shall allow logged-in users to start a cycling trip in manual or automatic mode.	Partial - Only Manual Mode was required
R13	The system shall allow logged-in users to stop a currently active trip and save the recorded data.	Yes
R14	The system shall collect GPS data during trip recording.	Yes
R15	The system shall collect motion sensor data (accelerometer, gyroscope) during trip recording only when Automatic Mode is enabled.	No - Not required
R16	The system shall allow logged-in users to view the list of their recorded trips.	Yes
R17	The system shall allow logged-in users to view a summary of their overall cycling statistics (total distance, total time, average speed, etc.).	Yes
R18	The system shall allow logged-in users to view statistics for each trip (distance, speed, duration, etc.).	Yes
R19	The system shall display the route and reported obstacles associated with a recorded trip.	Yes

Continued on next page

Rx	Description	Implemented
R20	The system shall allow logged-in users to delete a recorded trip.	Yes
R21	The system shall communicate with external weather services to retrieve meteorological data related to the time and location of a trip.	Yes
R22	The system shall detect when a user is cycling based on speed and acceleration patterns.	TODO
R23	The system shall detect irregular movements from sensor data that may suggest potholes or surface defects when Automatic Mode is enabled.	No - Not required
R24	The system shall present automatically detected path and obstacle data to the logged-in user for manual confirmation before publishing.	No - Not required
R25	The system shall allow logged-in users to manually create a new bike path by drawing segments.	Yes
R26	The system shall allow logged-in users to manually report obstacles or problems on a bike path while performing an active trip.	Yes
R27	The system shall allow logged-in users to manually confirm or reject the presence of obstacles reported by other users.	Yes
R28	The system shall allow logged-in users to create a new bike path in automatic mode using GPS tracking.	No - Not required (but I think it wont be difficult actually to implement, so let's think about this one)
R29	The system shall allow logged-in users to delete their previously created paths.	Yes
R30	The system shall allow logged-in users to set the visibility of their created paths as public or private.	Yes
R31	The system shall aggregate multiple user reports referring to the same path segment.	Yes

Continued on next page

Rx	Description	Implemented
R32	The system shall evaluate the reliability of each path segment based on the number of confirmations and report freshness.	Yes
R33	The system shall determine the current status of a path (optimal, medium, sufficient, requires maintenance, closed).	Yes
R34	The system shall allow any user (guest or logged-in) to view the detailed status and latest reports of a selected bike path.	Yes
R35	The system shall allow any user (guest or logged-in) to browse available public bike paths on a map.	Yes
R36	The system shall allow any user to search for bike paths connecting two locations.	Yes
R37	The system shall compute suggested routes based on path quality and distance.	Yes
R38	The system shall rank suggested routes according to their safety and quality.	Yes
R39	The system shall display the user's current GPS position during navigation along a selected path.	Yes
R40	The system shall send pop-ups to warn users about nearby obstacles or closed path segments during an active trip.	Yes
R41	The system shall interface with map and geocoding services to translate addresses into coordinates and render paths.	Yes
R42	The system shall ensure that communication with all external services (map, weather) handles temporary unavailability gracefully.	Yes

Table 2.3: Mapping between BBP Requirements and implemented functionalities

3 | Adopted Development Frameworks

3.1. Adopted Frameworks

3.1.1. Frontend

React Native Expo React Paper -> Theming Lucide Icons -> Icons React Native Maps -> Maps Axios -> Api Calls Zod -> Data Validation Zustand -> State Management Expo Router -> Navigation

3.1.2. Backend

NestJS Openmeteo service

3.1.3. Data Layer

PostgreSQL Prisma For data storage and management, the BBP platform utilizes PostgreSQL as its primary DBMS. To bridge the gap between the modular NestJS backend and the relational data, Prisma is employed as the Object-Relational Mapping (ORM) tool.

- **Relational Data Integrity:** PostgreSQL was selected for its reliability and its ability to handle complex relational queries, such as merging path information from multiple users or calculating scores based on freshness and confirmation counts.
- **Type-Safe Persistence:** Prisma provides a type-safe database client that is automatically generated from a central schema file. This ensures that the DTOs used in the application layer are perfectly synchronized with the database layer, significantly reducing runtime errors during data insertion (e.g., manual vs. automated mode data).
- **Schema Management:** Using Prisma Migrations, the technical team can track

and version-control changes to the database structure, such as new tables for "Reports" or "Meteorological Snapshots," ensuring consistency across development and production environments.

3.2. Adopted Programming Languages

TypeScript

3.3. Development Tools

Docker + Shared Nginx + Cloudflare + Docker

3.4. API Calls

Any API not included in the DD should be mentioned here.

4 | Source Code Structure

4.1. Frontend

The following directory tree provides an overview of the frontend source code structure of the BestBikePaths mobile app. The structure reflects the organization adopted by Expo Router and the separation of concerns between routing, UI components, and utilities.

```
src/
|-- api/                          # Backend API communication
|   |-- client.ts                 # Axios client with interceptors
|   |-- auth.ts                   # Authentication API wrappers
|   |__ ...                       # Other API modules
|-- app/                          # Expo Router navigation structure
|   |-- (auth)/                   # Authentication-related screens
|   |   |-- _layout.tsx           # Auth flow layout
|   |   |-- welcome.tsx           # Welcome screen
|   |   |__ ...                   # Other auth screens
|   |-- (main)/                   # Main application screens
|   |   |-- _layout.tsx           # Bottom navigation and access guards
|   |   |-- home.tsx              # Map-based path search and navigation
|   |   |__ ...                   # Other main screens
|   |-- _layout.tsx               # Root layout with global providers
|   |__ +not-found.tsx            # Fallback screen for unknown routes
|-- android/                      # Android native project
|-- assets/                       # Static assets
|   |-- images/                   # Icons and images
|   |__ fonts/                    # Custom fonts
|-- auth/                         # Authentication and session management
|   |-- authSession.ts            # In-memory session handling
|   |__ storage.ts                # Zustand store + SecureStore integration
|-- components/                   # Reusable UI components
|   |-- ui/                       # UI primitives (buttons, inputs, popups)
```

```
|   |   |-- AppButton.tsx         # Custom button component
|   |   |__ ...                   # Other UI primitives
|   |-- icons/                    # Icon wrappers and helpers
|   |   |__ LucideIcon.tsx        # Lucide icon set integration
|   |__ ...                       # Other components
|-- constants/                    # Static configuration values
|   |-- Colors.ts                 # Color palette definitions
|   |__ Privacy.ts                # Privacy options
|-- hooks/                        # Custom React hooks
|   |-- useBottomNavVisibility.tsx # Bottom navigation visibility
|   |__ ...                       # Other hooks
|-- ios/                          # iOS native project
|-- tests/                        # Automated tests
|   |-- integration/              # Integration tests
|   |-- mocks/                    # Module mocks
|   |-- unit/                     # Unit tests
|   |__ utils/                    # Test helpers
|-- theme/                        # Theming configuration
|   |-- layout.ts                 # Layout helpers
|   |-- mapStyles.ts              # Map style definitions
|   |-- paperTheme.ts             # React Native Paper theme
|   |__ typography.ts             # Typography settings
|-- utils/                        # Utility functions
|   |-- geo.ts                    # Distance and route helpers
|   |-- apiError.ts               # API error normalization
|   |__ ...                       # Other utilities
|-- validation/                   # Zod validation schemas
|   |-- auth.ts                   # Login, signup, profile schemas
|   |__ ...                       # Other validation schemas
|-- .expo/                        # Expo local state
|-- .env                          # Environment variables
|-- .gitignore                    # Git ignore rules
|-- .npmrc                        # NPM configuration
|-- app.json                      # Expo app configuration
|-- babel.config.js               # Babel configuration
|-- expo-env.d.ts                 # Expo TypeScript env definitions
|-- jest.config.js                # Jest configuration
```

```
|-- jest.setup.ts                # Global test setup
|-- node_modules/                # Installed dependencies
|-- package.json                 # Dependencies and scripts
|-- package-lock.json            # Locked dependency versions
|__ tsconfig.json                # TypeScript configuration
```

4.2. Backend

The following directory tree outlines the structure of the backend source code of the Best-BikePaths system. The backend follows a modular and layered architecture, separating concerns between routing, middleware, business logic, data access, and external service integrations.

```
backend/
|-- prisma/                                # Prisma ORM configuration
|   |-- schema.prisma                      # Database schema definition
|   |-- migrations/                        # Database migrations
|   |__ json.types.d.ts                    # Custom Prisma JSON type definitions
|-- src/                                   # Source code
|   |-- errors/                            # Custom error classes
|   |   |-- app.errors.ts                  # Application-specific errors
|   |   |__ index.ts                       # Export all error classes
|   |-- managers/                          # Business logic
|   |   |-- auth/                          # Authentication logic
|   |   |-- path/                          #  Path management logic
|   |   |__ ...                            # Other managers
|   |-- middleware/                        # Middlewares
|   |   |-- jwt.auth.ts                    # JWT authentication middleware
|   |   |-- http.logger.ts                 # HTTP request logging middleware
|   |   |__ ...                            # Other middlewares
|   |-- routes/                            # API route definitions
|   |   |-- v1/                            # Version 1 of the API
|   |       |-- index.ts                   # API version entry point
|   |       |-- auth.routes.ts             # Authentication routes
|   |       |-- user.routes.ts             # User routes
|   |       |__ ...                        # Other routes
|   |-- schemas/                           # Validation schemas
|   |   |-- auth.schema.ts                 # Auth-related schemas
```

```
|   |   |-- user.schema.ts              # User-related schemas
|   |   |__ ...                         # Other schemas
|   |-- services/                       # External service integrations
|   |   |-- openmeteo.service.ts        # OpenMeteo API integration
|   |   |__ ...                         # Other services
|   |-- tests/                          # Test files
|   |   |-- integration/                # Integration tests
|   |   |   |-- auth.integration.test.ts # Auth integration tests
|   |   |   |__ ...                     # Other integration tests
|   |   |-- unit/                       # Unit tests
|   |       |--  auth.manager.test.ts   # Auth manager tests
|   |       |__ ...                     # Other unit tests
|   |-- types/                          # TypeScript type definitions
|   |   |-- express/                    # Express-related types
|   |   |   |-- index.d.ts              # Custom Express types
|   |   |__ coordinates.types.ts        # Coordinate types
|   |   |__ ...                         # Other type definitions
|   |-- utils/                          # Utility functions
|   |   |-- prismaclient.ts             # Prisma client instance
|   |   |-- geo.ts                      # Geospatial utility functions
|   |   |__ ...                         # Other utility functions
|   |__ server.ts                       # Server entry point
|-- .env                                # Environment variables
|-- .gitignore                          # Git ignore rules
|-- docker-compose.yml                  # Docker Compose configuration
|-- Dockerfile                          # Dockerfile for backend service
|-- jest.config.mjs                     # Jest configuration
|-- jest-storage/                       # Jest storage
|-- node_modules/                       # Installed dependencies
|-- notes.md                            # Project notes
|-- package.json                        # Dependencies and scripts
|-- package-lock.json                   # Locked dependency versions
|-- prisma.config.ts                    # Prisma configuration
|-- setup.test.ts                       # Global test setup
|__ tsconfig.json                       # TypeScript configuration
```

4.3. Server

The following directory structure provides a high-level overview of the server-side deployment environment used to host the BestBikePaths system. It illustrates the organization of reverse proxy configuration, SSL certificates, and Dockerized application services on the target server.

```
/opt/
|-- nginx/                          # NGINX reverse proxy configuration
|   |-- conf.d/                     # Virtual host and routing configuration
|   |   |-- site.conf               # Main site configuration
|   |   |__ api.conf                # Backend API proxy configuration
|   |-- ssl/                        # TLS certificates and private keys
|   |   |-- bia3iaorigin.crt
|   |   |-- bia3iaorigin.key
|   |   |__ ...                     # Other certificate files
|   |-- log/                        # NGINX log files
|       |-- access.log              # Access logs
|       |__ error.log               # Error logs
|-- residenzaclasmarina/            # Additional hosted service
|   |-- docker-compose.yml          # Docker Compose configuration
|   |__ ...                         # Service source code and configuration
|__ bbpbackend/                     # BestBikePaths backend service
    |-- Dockerfile                  # Backend container definition
    |-- docker-compose.yml          # Backend service orchestration
    |__ ...                         # Backend source code and configuration
```

5 | Testing Strategy

5.1. Unit Testing

5.2. Integration Testing

In the notes, both of APP and BACKEND there is explanation.

6 | Installation Instructions

The BBP platform can be used in different configurations, depending on the intended use case and on whether the user aims to simply run the application or to actively develop and test it.
In particular, three main usage scenarios are supported:

- **Remote backend with prebuilt mobile application**: the user installs a prebuilt APK and interacts with an already deployed backend instance.
- **Fully local execution**: both the backend services and the mobile application are executed locally for development and testing purposes.
- **Hybrid configuration**: the mobile application is run locally while relying on a remotely deployed backend.

6.1. Prerequisites

To run the BBP platform locally, a set of prerequisites must be satisfied to ensure a consistent and reproducible development environment.

- **Node.js and npm**: required to manage dependencies and execute project scripts for both the backend services and the mobile application. All build, test, and development workflows rely on the Node.js runtime.
- **Docker and Docker Compose**: required to run backend services and auxiliary components in isolated containers. In the local setup, Docker Compose is used to start the BBP backend, the PostgreSQL database, and the OSRM routing service, ensuring a configuration that closely mirrors the production environment without requiring manual installation of these components on the host system.
- **Expo CLI**: required to run the mobile application locally during development. It is used to start the Metro bundler and to enable live previews of the application on emulators or physical devices.

- **Expo Go**: required on a physical mobile device to preview the application during development by connecting to the local Metro bundler started via Expo CLI. May be replaced with an emulator.

6.2. Backend Setup

6.2.1. Production Deployment

The BBP backend is deployed on a **VPS** using a **container-based architecture** built on Docker and Docker Compose. The service is designed to coexist with other applications hosted on the same server while remaining isolated and not directly exposed to the public network.
The backend application runs as a stateless service inside **Docker containers** and listens on port 3000, which is reachable only within the container network. No backend container exposes public ports on the host system. All incoming traffic is handled by a **shared NGINX reverse proxy**, which is the only component exposing ports 80 and 443.
NGINX acts as the **single entry point** for the system and is responsible for HTTPS termination, request routing, load balancing, and basic traffic filtering. TLS is managed using **Cloudflare Origin Certificates**, with Cloudflare acting as **DNS provider and upstream security layer**. Requests validated by Cloudflare are forwarded to the reverse proxy, which then routes them to the backend service.
The backend is horizontally scalable and is deployed using **multiple container replicas**. In the current configuration, three backend instances are executed in parallel. NGINX resolves backend containers dynamically using **Docker's embedded DNS** and **balances incoming requests** across replicas through an upstream definition. This approach allows the system to scale without introducing a dedicated load balancer and ensures high availability of the API layer.
The deployment process starts by copying the backend project files to the server. This can be performed from a local machine using a secure file transfer mechanism, for example:

```
rsync -avz ./BACKEND/ user@<SERVER_IP>:/opt/bbp-backend/
```

Once the project is available on the server, environment-specific configuration values are provided through a dedicated **.env** file. This file defines runtime parameters such as authentication secrets, database connection details, and service timeouts. Sensitive values are **never committed** to version control and are configured directly on the server.
The backend is built and started using **Docker Compose**. A shared **Docker network** is created once and is used to connect the reverse proxy, backend replicas, and auxiliary

services. The backend containers join this network and expose port 3000 only internally. The services can be built and started with the following commands:

```
docker compose build --no-cache
docker compose --profile tools run --rm migrate
docker compose up -d --scale api=3
```

Database **migrations** are executed explicitly through a dedicated migration container and are only run when schema changes are introduced. This avoids unintended schema modifications during routine restarts.
Persistent data storage is handled through **PostgreSQL** accessed via **Prisma Accelerate**. This configuration allows the backend to rely on a **managed database connection layer** without hosting a database instance directly on the server, while still preserving transactional guarantees and schema consistency.
In addition to the core backend service, snapping-related functionalities are delegated to a separate **OSRM service**, which runs in its own container. OSRM is responsible for snapping user-defined paths to the OpenStreetMap road network using a cycling profile. The backend communicates with OSRM through an internal HTTP interface and exposes a dedicated API endpoint to the mobile application.
After deployment, the backend can be verified by inspecting container status and logs:

```
docker compose ps
docker logs -f bbp_api --tail=50
```

A dedicated **health endpoint** is exposed and can be used by the reverse proxy to verify service availability. If correctly deployed, the backend responds to authenticated API requests through the public API endpoint exposed by the reverse proxy.

6.2.2. Local Backend Execution

For local development and testing, the BBP backend can be executed using a Docker-based setup that mirrors the production environment, while enabling development-specific features such as live reload. To keep the production configuration unchanged, the local development setup is defined through an additional Docker Compose file and a dedicated development Dockerfile.
Before starting the backend locally, developers must ensure they are operating from the backend project directory, which contains the production **docker-compose.yml**, the development override **docker-compose.dev.yml**, and the related **Dockerfile.dev**. All commands must be executed from this directory.

The development configuration builds the backend image using **Dockerfile.dev** and runs the service with **NODE_ENV=development**. Project source files are mounted into the container through bind mounts, so changes to the codebase are immediately reflected without rebuilding the image. Node.js dependencies are installed inside the container during the image build and are not required on the host system.
Environment-specific configuration values are provided through a dedicated **.env** file located in the backend project directory. This file defines authentication secrets, database connection parameters, and service endpoints. Additional local-only variables can be added when needed without affecting the production configuration.
To build and start the backend locally, the development override is applied on top of the base configuration using Docker Compose:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml up -d --build
```

Once the backend container is running, application logs can be inspected in real time to monitor startup and runtime behavior:

```
docker logs -f bbp_api --tail=50
```

A basic health check endpoint is exposed by the backend and can be used to verify that the service is running correctly:

```
curl -i http://localhost:3000/health
```

In the development configuration, port 3000 is published on the host system. As a result, the backend API is available at **http://localhost:3000** and can be accessed by the mobile application or by API testing tools.
Automated tests can be executed directly inside the running backend container. This allows tests to run in the same environment as the application, without requiring local dependency installation. Tests can be started with:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml exec api npm run test:wat
```

When the local backend is no longer needed, all running services can be stopped and removed using Docker Compose:

```
docker compose -f docker-compose.yml -f docker-compose.dev.yml down
```

This command stops and removes the containers associated with the local development environment without affecting images or configuration files. All backend services can be stopped and restarted at any time without affecting the host system, as the entire runtime environment is isolated within containers.

6.3. Frontend Setup

Use Expo CLI to run the app locally on an emulator or a physical device. You can also build an APK for Android and install it directly.
If you want to use a local server you should change the API URL in the .env file and build it or run it on the emulator, since the built app points to the production server.

7 References

7.1. Reference Documents

The preparation of this document was supported by the following reference materials:

- IEEE Standard for Software Requirement Specifications [2];
- Assignment specification for the ITD of the Software Engineering II course, held by professors Matteo Rossi, Elisabetta Di Nitto, and Matteo Camilli at the Politecnico di Milano, Academic Year 2025/2026 [6];
- Slides of the Software Engineering II course available on WeBeep [7].

7.2. Software Used

The following software tools have been used to support the development of this project:

- **Visual Studio Code**: editing of source code and documentation (LaTeX), with project-wide search and formatting support [5].
- **LaTeX**: typesetting system used to produce the final RASD document in a consistent format [3].
- **Git**: version control used to track changes and support collaborative development [8].
- **GitHub**: remote repository hosting and collaboration platform used for versioning, reviews, and issue tracking [1].
- **Lucidchart**: creation of UML diagrams (use case diagrams, state diagrams, domain class diagram) [4].

7.3. Use of AI Tools

AI tools were used during the project in the same way as other supporting software tools. Their role was not to autonomously generate content, but to assist in improving the presentation of the document, supporting the organisation of ideas and enhancing overall textual coherence.
Their use was mainly limited to the drafting phase, where they helped compare different ways of explaining scenarios, simplify long paragraphs, and check whether certain sentences could be misunderstood. In several cases, interacting with an AI assistant helped clarify the underlying concepts before writing the final version of the text.

7.3.1. Tools Used

The AI tools employed during the project were:

- Gemini
- ChatGPT

7.3.2. Typical Prompts

AI tools were queried using prompts such as:

- "Rephrase this design description to make the interaction flow clearer."
- "Does this explanation of the component interaction sound ambiguous?"
- "Help restructure this paragraph describing a UI flow to improve readability."
- "Format this design description or table using LaTeX"
- "Help debug formatting or build issues related to VS Code or LaTeX"

7.3.3. Input Provided

The input given to AI tools consisted mainly of:

- Early drafts of paragraphs.
- Short text fragments requiring clarity checks.
- Sections with repeated structure where consistent wording was needed.

7.3.4. Constraints Applied

When using AI tools, the following constraints were strictly enforced:

- Preserve the intended meaning of the original text.
- Avoid introducing new design decisions or assumptions.
- Maintain terminology aligned with the definitions provided in this document.

7.3.5. Outputs Obtained

The interaction with AI tools resulted in:

- Clearer or more concise formulations of existing statements.
- Identification of potentially ambiguous sentences.
- Terminology suggestions to improve internal coherence.
- LaTeX formatting assistance for tables and code snippets.

7.3.6. Refinement Process

All AI-generated outputs were subject to a manual refinement process that included:

- Critical review of all suggestions.
- Verification against the original intent to avoid unintended changes.
- Manual integration to ensure consistency with the overall writing style.
- Alignment checks with established terminology and definitions.

8 | Effort Spent

This section provides a breakdown of the number of hours each group member dedicated to completing this document. The work distribution is tracked per section and task.

Section	Ianosel Bianca	Simone Errigo	Vajihe Gholami	Total Hours
Introduction	4 hours	4 hours	5 hours	13 hours
Overall Description	11 hours	7 hours	10 hours	28 hours
Specific Requirements	19 hours	8 hours	12 hours	39 hours
Formal Analysis	7 hours	21 hours	11 hours	39 hours
Final Review & Editing	3 hours	3 hours	3 hours	9 hours
Total Hours	**44 hours**	**43 hours**	**41 hours**	**128 hours**

Table 8.1: Time spent on document preparation

Bibliography

[1] GitHub Inc. Github. Online platform, 2025. https://github.com/.

[2] ISO/IEC/IEEE. Systems and software engineering - life cycle processes - requirements engineering, 2018.

[3] LaTeX Project Team. Latex: A document preparation system. Document preparation system, 2025. https://www.latex-project.org/.

[4] Lucid Software Inc. Lucidchart: Diagramming and visualization tool. Online platform, 2025. https://www.lucidchart.com/.

[5] Microsoft. Visual studio code. Source code editor, 2025. https://code.visualstudio.com/.

[6] M. Rossi, E. Di Nitto, and M. Camilli. Software engineering 2 itd assignment specification, Academic Year 2025/2026.

[7] M. Rossi, E. Di Nitto, and M. Camilli. Slides of the software engineering 2 course. WeBeep platform, Academic Year 2025/2026.

[8] Software Freedom Conservancy. Git. Version control system, 2025. https://git-scm.com/.

List of Figures

List of Tables