THE UNIVERSITY OF ZAMBIA

SCHOOL OF NATURAL AND APPLIED SCIENCES

DEPARTMENT OF COMPUTING AND INFORMATICS

FINAL YEAR PROJECT REPORT

ACADEMIC YEAR 2026

Implementation of a Web-Based Real-Time Public Bus Tracking System for Lusaka

The Driver Interface: Driver Sign-in, Bus Activation, Trip Start and Device GPS Tracking

A Final Year Project Report Submitted to the

Department of Computing and Informatics

in partial fulfilment of the requirements for the award of

Bachelor of Science in Computer Science

By

Madalitso Nyirenda

Computer Number: 2021502597

Group members:

Kenneth Kaunda Musonda (Computer Number: 2020056283)

Ngosa Muleba (Computer Number: 2022044371)

Supervisor: Dr. Lighton Phiri

Date of Submission: [date to be confirmed]

DECLARATION

I declare that this final year project report is my own work and has been prepared in accordance with the academic requirements of the University of Zambia. All sources of information, ideas, data, software, images, and other material that are not my own have been appropriately acknowledged and referenced.

This project was undertaken as part of a group. I have clearly identified my individual contributions in Chapter 5, Chapter 6 and Appendix A, and I have not presented the work of other group members as my own. Chapters 2 to 4 and parts of Chapters 1, 6, 7 and 8 describe material that was prepared or developed jointly by the group, or that was developed by another member; Section 1.8 states how that material is used in this report.

Student Name: Madalitso Nyirenda

Signature: _____

Computer Number: 2021502597

Date: _____

APPROVAL

This final year project report has been submitted for examination with the approval of the project supervisor.

Supervisor Name: Dr. Lighton Phiri

Signature: _____

Date: _____

Departmental/Examiner Approval (if applicable):
_____

ABSTRACT

Passengers who travel by bus in Lusaka often lack dependable, route-specific information about where a bus is and when it will reach their stop. This project, undertaken by a group of three students, set out to implement a web-based real-time public bus tracking system for Lusaka that gives passengers route-based journey information and an estimated arrival time, and gives drivers a view of the passengers waiting along their route. The project began from a proposal that assumed simulated GPS data. After supervisor feedback, the design was changed to use real coordinates and speed from a driver's GPS-enabled mobile device.

Development was iterative and incremental. One group member first built the initial backend skeleton. The author and a second member then developed the driver interface and the passenger interface respectively, and the backend was extended as the needs of the two interfaces emerged. The first member subsequently integrated the branches and deployed the complete system. The system is a web application with browser interfaces for passengers and drivers and a Django REST Framework server. It imports static GTFS route, stop, trip and stop-time data, authorises drivers to operate their assigned buses, maintains the trip, position, speed and status of each bus from its GPS updates, manages passengers' waiting requests, and estimates arrival times from road durations obtained from the OSRM routing service.

This individual report describes the whole system and evaluates it against the group's objectives. The author's principal individual contribution was the driver interface and the driver-side functionality: driver sign-in, bus selection and activation, trip start, device GPS tracking and the display of waiting passengers, together with the trip-lifecycle and stop-progression work on the Driver-Interface branch. The system was verified functionally by automated tests of the server-side logic, by tests run on the author's development branch and by manual checks of the deployed passenger and driver flows. The three intended features, Real-Time Bus Tracking, Journey Planning and Arrival Time Prediction, were implemented at prototype level. The evidence is limited. No operational bus fleet or operator data feed was available, so the GPS positions could not be verified as those of an operating bus, and no field trial, user study or performance benchmark was carried out. The system is therefore a verified prototype, not a proven reduction in waiting time.

Keywords: bus tracking; real-time GPS; driver interface; Django REST Framework; GTFS; estimated time of arrival

ACKNOWLEDGEMENTS

I thank my supervisor, Dr. Lighton Phiri, for his guidance and for the regular project reviews, whose feedback shaped the direction of the project, most notably the decision to replace simulated GPS data with real GPS data. I acknowledge the University of Zambia and the Department of Computing and Informatics for the academic environment in which this work was carried out.

I also acknowledge my group members. Kenneth Kaunda Musonda built the initial backend foundation, integrated the interface branches and deployed the complete system. Ngosa Muleba developed the passenger interface. The problem analysis, the requirements analysis and the integrated system were shared among the three of us, and this report would not describe a working system without their work.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION..... 14
1.1 Background and Context..... 14
1.2 Problem Statement..... 15
1.3 Project Aim..... 15
1.4 Project Objectives..... 15
1.5 Scope of the Project..... 17
1.6 Significance of the Project..... 18
1.7 Limitations..... 18
1.8 Organisation of the Report..... 19
CHAPTER 2: LITERATURE REVIEW..... 21
2.1 Introduction..... 21
2.2 Key Concepts and Theories..... 21
2.3 Existing Systems / Related Work..... 25
2.4 Comparative Analysis..... 26
2.5 Research / Knowledge Gap..... 27
2.6 Conceptual or Theoretical Framework..... 27
2.7 Chapter Summary..... 28
CHAPTER 3: METHODOLOGY..... 29
3.1 Introduction..... 29
3.2 Research / Development Approach..... 29
3.3 Requirements / Data Collection..... 35
3.4 System Architecture / Research Workflow..... 38
3.5 Tools and Technologies..... 38
3.6 Ethical and Legal Considerations..... 39
3.7 Evaluation Method..... 40
3.8 Chapter Summary..... 41
CHAPTER 4: SYSTEM ANALYSIS AND DESIGN..... 42
4.1 Introduction..... 42
4.2 Stakeholder and User Analysis..... 42
4.3 Functional Requirements..... 43
4.4 Non-Functional Requirements..... 46
4.5 Use Cases / User Stories..... 48
4.6 System / Solution Architecture..... 54
4.7 Database / Data Design..... 56
4.8 Interface / Interaction Design..... 61

4.9 Algorithms / Models / Technical Design....66
4.10 Chapter Summary....73
CHAPTER 5: IMPLEMENTATION / DEVELOPMENT....74
5.1 Introduction....74
5.2 Development Environment....75
5.3 System Components / Modules....75
5.4 Key Implementation Details....77
5.5 Integration....84
5.6 Challenges and Solutions....85
5.7 Individual Contribution (Group Projects)....87
5.8 Chapter Summary....89
CHAPTER 6: TESTING, RESULTS AND EVALUATION....90
6.1 Introduction....90
6.2 Testing Strategy....90
6.3 Test Cases and Results....91
6.4 Experimental Results / Performance Results....104
6.5 User Evaluation....105
6.6 Analysis of Results....105
6.7 Objective-by-Objective Evaluation....108
6.8 Individual Results....110
6.9 Chapter Summary....111
CHAPTER 7: DISCUSSION....112
7.1 Introduction....112
7.2 Interpretation of Findings....112
7.3 Comparison with Existing Work....114
7.4 Contributions of the Project....114
7.5 Limitations and Implications....115
7.6 Chapter Summary....116
CHAPTER 8: CONCLUSION AND RECOMMENDATIONS....117
8.1 Introduction....117
8.2 Summary of the Project....117
8.3 Conclusions....117
8.4 Achievement of Objectives....118
8.5 Recommendations....119
8.6 Future Work....119
REFERENCES....121
APPENDICES....123
APPENDIX A: INDIVIDUAL CONTRIBUTION STATEMENT (GROUP PROJECTS)124

APPENDIX B: USER MANUAL / INSTALLATION GUIDE.....129
APPENDIX C: ADDITIONAL SYSTEM DESIGNS / DIAGRAMS.....132
APPENDIX D: ADDITIONAL TEST CASES AND RESULTS.....134
APPENDIX E: DATA COLLECTION INSTRUMENTS.....140
APPENDIX F: ADDITIONAL CODE / CONFIGURATION.....141
APPENDIX G: OTHER SUPPORTING MATERIAL.....144

LIST OF FIGURES

Figure 2.2.1.1 Intelligent Transportation Systems and bus tracking (conceptual).....22
Figure 2.2.2.1 Generic GPS-based bus tracking information flow (conceptual).....23
Figure 2.2.4.1 Real-time passenger information flow (conceptual).....24
Figure 2.2.7.1 Relationship between bus, route, stop and ETA (conceptual).....25
Figure 2.6.1 Conceptual framework derived from the reviewed literature.....28
Figure 3.2.1.1 Iterative and incremental development process of the Bus Tracking System.....30
Figure 4.5.1.1 UML use-case diagram of the Bus Tracking System.....49
Figure 4.5.3.1 System sequence diagram: authorised driver starts a trip and updates the bus GPS position.....52
Figure 4.5.3.2 System sequence diagram: passenger plans a journey, creates a waiting request and retrieves the ETA.....53
Figure 4.6.1 System architecture of the Bus Tracking System.....55
Figure 4.7.1.1 Domain model of the Bus Tracking System (conceptual).....57
Figure 4.7.2.1 Entity-relationship diagram of the persisted data.....59
Figure 4.8.1 Passenger interface screens for route search, boarding-stop selection and waiting-request creation (passenger interface developed mainly by Ngosa Muleba).....65
Figure 4.8.2 A waiting request in the WAITING state (passenger interface developed mainly by Ngosa Muleba).....65
Figure 4.8.3 A waiting request in the ON_BOARD state (passenger interface developed mainly by Ngosa Muleba).....66
Figure 4.9.1.1 Design class diagram of the system modules and persistent classes.....67
Figure 4.9.1.2 Interaction diagram for the system operation updateBusLocation.....68
Figure 4.9.1.3 Interaction diagram for the system operation getWaitingRequestEta.....69
Figure 6.3.2 Landing page of the deployed system.....95
Figure 6.3.3 Driver sign-in (driver interface developed by the author).....96
Figure 6.3.4 Driver dashboard with the assigned bus selected and its status IDLE (driver interface developed by the author).....96
Figure 6.3.5 Bus activated and destination choice for the route Bauleni – Kulima Tower (driver interface developed by the author).....97
Figure 6.3.6 Browser permission request and waiting for a device GPS fix (driver interface developed by the author).....97
Figure 6.3.7 GPS update accepted, reported location, speed, status and live map (driver interface developed by the author).....98
Figure 6.3.8 Passenger map showing the bus marker (passenger interface developed mainly by Ngosa Muleba).....98
Figure 6.3.9 Passenger sign-in (passenger interface developed mainly by Ngosa Muleba).....99
Figure 6.3.10 Route search and destinations offered (passenger interface developed mainly by Ngosa Muleba).....99

Figure 6.3.11 Boarding-stop selection (passenger interface developed mainly by Ngosa Muleba) 100
Figure 6.3.12 Waiting request created, with the arrival estimate and the options to board or cancel (passenger interface developed mainly by Ngosa Muleba) 100
Figure 6.3.13 Passenger waiting request (left) and the driver dashboard showing one passenger waiting at Civic Center (right) 101
Figure 6.3.14 Django administration site of the deployed system 101
Figure C.1 Entity-relationship diagram of the persisted data 133

LIST OF TABLES

Table 1.4.1 Proposal objectives and their treatment in the final project.....15
Table 1.4.2 Objectives of the members' individual contributions.....16
Table 1.5.1 Scope of the project as implemented.....17
Table 2.4.1 Comparison of reviewed systems and standards with the Bus Tracking System.....26
Table 3.2.2.1 Stages of development and the work done in each stage.....31
Table 3.2.2.2 Stages of development and the passenger-interface work in each stage.....31
Table 3.2.2.3 Stages of development and the driver-interface work in each stage.....32
Table 3.2.4.1 Proposal compared with the implemented system.....33
Table 3.2.5.1 Proposal compared with the implemented passenger interface.....34
Table 3.2.6.1 Proposal compared with the implemented driver interface.....34
Table 3.3.3.1 Adaptation of ride-hailing interaction ideas to a public bus service.....36
Table 3.3.5.1 Contents of the GTFS files stored in the repository.....37
Table 3.3.6.1 Contents of the GTFS files that the passenger interface draws on.....37
Table 3.5.1 Principal tools and technologies.....38
Table 4.1.1 Analysis and design artefacts used in this report.....42
Table 4.2.1 Stakeholders and their needs.....43
Table 4.3.1 Functional requirements implemented by the system.....43
Table 4.3.2 Functional requirements of the passenger interface.....45
Table 4.3.3 Server requirements the passenger interface depends on.....45
Table 4.3.4 Functional requirements of the driver interface.....46
Table 4.4.1 Non-functional requirements and design response.....47
Table 4.4.2 Non-functional requirements and design response.....47
Table 4.4.3 Non-functional requirements and design response for the driver interface.....48
Table 4.5.2.1 Brief descriptions of the principal use cases.....50
Table 4.5.2.2 Principal passenger use cases.....50
Table 4.5.2.3 Principal driver use cases and the supporting part of the dashboard.....51
Table 4.5.3.1 System operations and the endpoints that implement them.....53
Table 4.7.2.1 Data dictionary of the principal entities.....60
Table 4.7.3.1 Data held about passengers and waiting requests.....60
Table 4.7.4.1 Data handled by the driver interface.....61
Table 4.8.1.1 Parts of the driver dashboard and what each one does.....63
Table 4.8.2.1 Parts of the passenger page and what each one does.....64
Table 4.9.3.1 Rules for next-stop update and bus status.....70
Table 4.9.8.1 Controls offered for each waiting-request status.....72
Table 5.1.1 Order of development and responsibility.....74
Table 5.3.1 System components, responsibilities and developers.....76

Table 5.5.1 Passenger-interface actions and the API calls they trigger....84
Table 5.7.1 Contribution summary by activity....88
Table 6.3.1 Behaviours verified by the integrated-system automated tests....92
Table 6.3.2 Driver-Interface branch test results (what is known)....94
Table 6.3.3 Manual checks of the passenger flows on the deployed system....102
Table 6.3.4 Automated server tests that support the passenger flows....103
Table 6.3.5 Manual checks of the driver flows on the deployed system....104
Table 6.6.1 Defects and technical limitations identified in the system....105
Table 6.6.2 Limitations of the passenger interface....107
Table 6.6.3 Limitations of the driver interface....107
Table 6.7.1 Evaluation of the group objectives....108
Table 6.7.2 Evaluation of the members' individual objectives....109
Table 8.4.1 Achievement of the group objectives....118
Table A.1 Project and group information....124
Table A.2 Nature of the individual contribution by activity....127
Table D.1 Test methods....134
Table D.2 Manual passenger test script....137
Table D.3 Manual driver test script....138
Table F.1 API endpoints of the integration branch....141
Table F.2 Environment variables read by the application....142

LIST OF ABBREVIATIONS AND ACRONYMS

Abbreviation	Meaning
API	Application Programming Interface
CSRF	Cross-Site Request Forgery
DRF	Django REST Framework
ER	Entity-Relationship
ETA	Estimated Time of Arrival
GPS	Global Positioning System
GTFS	General Transit Feed Specification
ITS	Intelligent Transportation Systems
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
OSRM	Open Source Routing Machine
SSD	System Sequence Diagram
UAT	User Acceptance Testing
WSGI	Web Server Gateway Interface

CHAPTER 1: INTRODUCTION

1.1 Background and Context

Public transport depends on information as much as on vehicles. A passenger who intends to travel by bus needs to know which route to use, where to board, whether a bus is operating on that route and roughly when it will arrive. The project proposal that started this work identified buses as the primary mode of transport in Lusaka and described passengers who rely on guesswork and informal information while waiting at stops. This report does not attempt to measure that situation. It takes the proposal's premise as the motivation for building a prototype and uses the literature to place the work in context.

Intelligent Transportation Systems (ITS) apply computing, positioning and communication technologies to transport, and recent reviews identify real-time monitoring and smart public transport as active areas [1]. GPS-based bus tracking is one application. Khalid, Salleh and Samsudin showed that consumer-grade GPS, mobile Internet and a central server can provide passengers with bus positions and arrival estimates [2], and Watkins et al. found that real-time information affects both perceived and actual waiting time for transit users [3]. Chapter 2 reviews this literature and its limits.

The project was carried out by a group of three students: Kenneth Kaunda Musonda (2020056283), Ngosa Muleba (2022044371) and the author, Madalitso Nyirenda (2021502597), under the supervision of Dr. Lighton Phiri. The group built one integrated system, the Bus Tracking System, made of browser interfaces for passengers and drivers, a Django server with an API, a database and a road-routing service. This is an individual report. It describes the shared project, its requirements, architecture and overall system, and evaluates the system as a whole. The author's own contribution, the driver interface and the driver-side functionality, is identified in Section 5.7, Section 6.8 and Appendix A. The work of the other members is acknowledged where it is discussed: Kenneth Kaunda Musonda built the initial backend foundation and later integrated the branches and deployed the system, and Ngosa Muleba developed the passenger interface.

A significant change occurred during the project. The proposal planned to simulate bus movement with a GPS simulation tool. Following supervisor feedback, the design was revised so that the system accepts real GPS coordinates and speed from a GPS-enabled device used by the bus driver. The final system, and therefore this report, is based on real GPS updates. Simulation appears in this report only as an earlier prototype stage that was replaced.

1.2 Problem Statement

Passengers who wait for a bus in Lusaka generally have no system-provided, route-specific information on where a given bus is and when it will reach the stop at which they are waiting. A raw coordinate would not solve this on its own. A coordinate is useful to a passenger only when the system can relate it to a route, a direction of travel, an ordered list of stops and the stop at which that passenger intends to board.

The specific problem addressed by the project is therefore how to provide a web-based system that (a) receives real position updates only from an authorised driver for a bus that the driver is permitted to operate, (b) keeps the current operational state of each bus, (c) connects that state to structured route, trip and stop data, (d) lets passengers choose a route and boarding stop and record that they are waiting, (e) shows drivers where passengers are waiting, and (f) returns an estimated arrival time only when enough valid information exists to calculate one. The reviewed literature does not allow a claim that no digital bus-tracking system exists anywhere in Zambia; the gap addressed is a localised implementation gap, developed in Section 2.5.

1.3 Project Aim

The aim of the group project is to implement a web-based real-time public bus tracking system for Lusaka that uses real driver-device GPS updates, structured transport data and a road-routing service to support three features: 1. Real-Time Bus Tracking, 2. Journey Planning and 3. Arrival Time Prediction.

This individual report documents the project as a whole, evaluates it honestly within the limits of the available evidence, and identifies the author's contribution to it (Section 5.7, Section 6.8 and Appendix A).

1.4 Project Objectives

The proposal listed five objectives. They were reviewed against what was built, and Table 1.4.1 shows how each one was treated in the final project. The first objective was changed because of the move from simulated to real GPS. These are the objectives of the group project and they apply to the system as a whole.

Table 1.4.1: Proposal objectives and their treatment in the final project

Proposal objective	Objective as implemented
1. Implement real-time bus tracking using generated GPS data	Real-Time Bus Tracking from real GPS coordinates and speed sent by an authorised driver device (changed after supervisor feedback)

Proposal objective	Objective as implemented
2. Build a passenger web UI to view bus locations and routes	A passenger web interface that shows active buses on a map and lets a passenger search a route, choose a destination and boarding stop, and place a waiting request (Journey Planning)
3. Provide estimated arrival times at specific stops	Arrival Time Prediction for a passenger's waiting request, from OSRM road-route duration
4. Enable operators to monitor and manage the fleet	Bus, driver, assignment and transport records can be managed through the Django administration site; no separate operator dashboard is claimed
5. Allow drivers to see active passengers at pickup points	The driver dashboard shows the number of waiting passengers for each stop of the bus's current trip

The group objectives are not replaced by individual objectives in this report. The author's individual work contributes mainly to objectives 1 and 5, through the driver interface and the driver-side trip and GPS functionality, and to the integrated system that realises all five. The specific individual contribution is set out in Section 5.7, evaluated in Section 6.8 and stated formally in Appendix A.

Individual contributions to the objectives. Each member's individual report concentrates on one part of the same system. Table 1.4.2 states the objectives that each member set for his or her own part, taken from the members' reports; the objectives for the driver interface are the ones formulated for this report. They are not additional project objectives. They show how the three parts combine to realise the five group objectives above.

Table 1.4.2: Objectives of the members' individual contributions

Member and part of the system	Individual objectives
Kenneth Kaunda Musonda: backend and deployment	(1) To design the persistent data model and implement GTFS import commands that load routes, stops, trips and stop times; (2) to authenticate drivers and authorise every driver operation through an active driver and an active driver-to-bus assignment; (3) to implement bus activation, deactivation, operational reset and trip start; (4) to process real GPS updates and maintain bus status and next-stop state; (5) to implement waiting-request services and OSRM-based arrival-time logic that declines to estimate when the data is invalid; and (6) to verify the arrival-time rules with automated tests and to prepare the deployment configuration.
Madalitso Nyirenda (the author): driver interface	(1) To let an authorised driver sign in, keep the session and see only the buses assigned to the account; (2) to let the driver activate, deactivate and reset a bus, choose a route and destination and start a trip; (3) to send the position and speed of the driver's device to the server while tracking and to show the position, speed and status that the server accepts; (4) to show the driver the number of passengers waiting at each stop of the trip; (5) to carry out the trip-lifecycle and stop-progression work that these screens need, on the Driver-Interface branch; and (6) to verify the driver flows on the deployed system, together with the passenger interface and the server.

Member and part of the system	Individual objectives
Ngosa Muleba: passenger interface	(1) To show the active buses on a live map, with their registration number, status and speed, and to refresh them automatically; (2) to let a passenger register, sign in and sign out, and reach the password-reset page; (3) to let a passenger search a route by name, choose one of its destinations and see the ordered stops of the trip; (4) to let a passenger choose a boarding stop and create a waiting request, within the rule that only one request may be active at a time; (5) to let a passenger see their own requests and mark a request as boarded, cancel it or complete it; (6) to show the arrival estimate for a waiting request when one is available; and (7) to verify these flows on the deployed system, together with the driver dashboard and the server.

All three members also contributed jointly to the problem analysis and the requirements analysis from which these objectives were derived.

1.5 Scope of the Project

Table 1.5.1 states the scope of the project as it was implemented in the final system. It lists the five items that the proposal placed in scope, all of which were implemented at prototype level, and the four items that remained out of scope. The last column states what was delivered for each item.

Table 1.5.1: Scope of the project as implemented

Scope item	Scope	What was delivered
Real-time GPS tracking of buses	In scope	Implemented. The driver dashboard sends the device's coordinates and speed, and the server accepts them only from an authorised driver (Section 5.4). The proposal's simulated GPS data was replaced after supervisor feedback.
Display of bus routes and stops on a map	In scope	Implemented. The passenger interface shows active buses as markers on a map, with their status and speed, and refreshes them every 15 seconds. Routes, destinations and the stops of a trip are presented as lists in the trip-planning panel.
Web application for passengers (browser-based)	In scope	Implemented. A passenger can register, sign in, search a route, choose a destination and a boarding stop, place a waiting request, mark it as boarded, cancel or complete it, and see its arrival estimate.
Driver interface: active passengers at pickup	In scope	Implemented. The driver dashboard offers sign-in, bus selection, activation, trip start, GPS tracking and the number of passengers waiting at each stop of the trip.
Admin dashboard for transport operators	In scope	Implemented through the Django administration site, where operators manage buses, drivers, assignments and transport records. No separate custom operator dashboard was built.
Online ticket payment system	Out of scope	Remains out of scope.

Scope item	Scope	What was delivered
Integration with government transport systems	Out of scope	Remains out of scope.
Physical GPS hardware installation in buses	Out of scope	Remains out of scope. Positions come from the driver's own device.
Native mobile app (iOS / Android)	Out of scope	Remains out of scope. The system is accessed through a web browser.

Within this scope, the author's individual work is the driver interface and the driver-side functionality (Section 5.7). The passenger interface was developed mainly by Ngosa Muleba, and the initial backend foundation, the integration of the branches and the deployment were carried out by Kenneth Kaunda Musonda. The statements about what was not done, such as the absence of city-wide deployment, user acceptance testing and performance testing, are limitations of the evidence and are given in Section 1.7.

1.6 Significance of the Project

The project shows how accessible components can be combined into one working pipeline: a GPS-enabled device used by the driver, structured GTFS-derived route data, a Django backend that keeps the state of each bus, and a road-routing service. It gives passengers a route-aware way to choose a bus, to register at a boarding stop and to obtain an arrival estimate, and it gives drivers a controlled way to announce their position and to see where passengers are waiting. For the university it provides a documented, tested base that later groups can extend. For the author it demonstrates the application of object-oriented analysis and design, using notation from Larman [4], to a real system in which the driver's device is the source of position data.

The significance should not be overstated. The project has not been evaluated by passengers or drivers, it has not run as a city-wide service, and its estimates have not been compared against observed arrival times. Its contribution is a verified prototype, not a proven reduction in waiting time.

1.7 Limitations

The limitations of the project are stated here as limitations of access, time, scope and evidence, data, technology, test coverage, security and deployment evidence. They affect how the results should be read and are discussed further in Sections 6.6 and 7.5.

- **Access to an operational fleet and operator data.** This was the principal limitation. The project had no access to an operational public-bus fleet or to an official bus-operator data feed. Although the system used real GPS coordinates from a GPS-enabled mobile device, the device location could not be independently verified as the location of an operating bus. Route, stop, trip and stop-time information was imported

from static GTFS files, so live service changes could not be received automatically. The evaluation therefore focused on functional verification of the implemented workflows, not on a long-term field trial involving operational buses and passengers. In short, the system could be verified against its design but not validated against an operating bus service.

- **Time.** The project had to be completed within a short academic year, while the members also carried the workload of other academic activities.
- **Scope and evidence.** No user study, user acceptance test, field trial or performance benchmark was carried out, although the proposal planned user testing and performance targets. The project therefore shows that the system works under test conditions. It does not show that the system improves passengers' waiting experience. The test evidence is the automated tests, the tests run on the author's development branch and manual functional and integration checks of the deployed flows (Section 6.3); none is exhaustive acceptance testing.
- **Data.** No observed arrival times were collected, and the automated tests use synthetic fixtures with a mocked routing function, so the accuracy of the ETA could not be measured. Journey planning uses 136 representative trips (68 routes, two directions each), not a full timetable.
- **Technology.** Arrival times use OSRM road durations and do not model live traffic or dwell time at stops. OSRM is an external service, and the report does not state its host. Only the current state of each bus is stored, so no location history is kept.
- **Test coverage.** The 35 automated tests of the integrated system exercise the server: the ETA, the waiting overview, access control, registration, login throttling and the debug default. There are no automated tests of the browser interfaces, which were checked manually, and these tests do not cover the GTFS import commands, the status and next-stop thresholds, validation of GPS values or a real OSRM instance. Tests run on the author's development branch are reported separately (Section 6.3).
- **Security.** Access control, password validation and login throttling were added in the final stage of development. Some weaknesses remain (Table 6.6.1), and no penetration test or independent security review was carried out.
- **Deployment evidence.** The report states the verified environment variables and the WSGI entry point. It gives the Render build and start commands. The project README gives the live address (https://bus-tracking6.onrender.com/) and the source repository (https://github.com/KennethKM/bus-tracking). The date on which the deployed flows were checked is not recorded in the supplied material.

1.8 Organisation of the Report

Chapter 2 reviews the literature, compares existing systems and states the research gap and conceptual framework. Chapter 3 presents the methodology, including the requirements

analysis and the iterative development approach. Chapter 4 gives the system analysis and design: requirements, use cases, system sequence diagrams, architecture, domain model, entity-relationship design and design model. Chapter 5 documents the implementation, the order in which the system was developed and the author's individual contribution. Chapter 6 reports testing and evaluation within the limits of the evidence. Chapter 7 discusses the findings, and Chapter 8 concludes and recommends future work. The references and appendices follow; Appendix A is the Individual Contribution Statement.

Use of shared group material. The problem statement, aim, objectives, scope, literature review, methodology, requirements, architecture, data design and algorithm design are shared project information. The group's final report, prepared by Kenneth Kaunda Musonda, and the group presentation are the primary sources for this report. Where this report reuses or adapts that material, the content is the product of joint group work or of the member named in the relevant section, and it has been edited for consistency with this individual report. Figures that were drawn for the group report are identified by their source figure numbers. The individual contribution statements in this report are the author's own and are not copied from the group report.

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

This chapter reviews the concepts and related work that informed the Bus Tracking System. It covers Intelligent Transportation Systems, GPS-based vehicle tracking, estimated time of arrival, real-time passenger information, existing systems, public-transport data standards and the software-engineering approach. Existing systems are compared, a research gap is identified and a conceptual framework is derived. The review is shared group work that was prepared for the group report and is reused here with minor edits for consistency.

The figures in this chapter (Figures 2.2.1.1 to 2.6.1 of the group report) are conceptual diagrams drawn from the literature to explain ideas. They are not diagrams of the deployed architecture of the Bus Tracking System; that architecture is presented in Chapter 4.

2.2 Key Concepts and Theories

2.2.1 Intelligent Transportation Systems

Intelligent Transportation Systems (ITS) apply computing, sensing, communication and information technologies to transport. In public transport they support vehicle monitoring, passenger information, fleet management and route planning. A recent systematic review of ITS in public and shared urban transport identifies smart infrastructure, data-driven mobility and smart public transportation among the prominent research areas [1]. The review treats ITS as an integrated technological environment rather than a single technology. A bus-tracking application is one component of that environment: positioning supplies the location of a vehicle, communication carries it to a server, and software processes it and presents it to passengers.

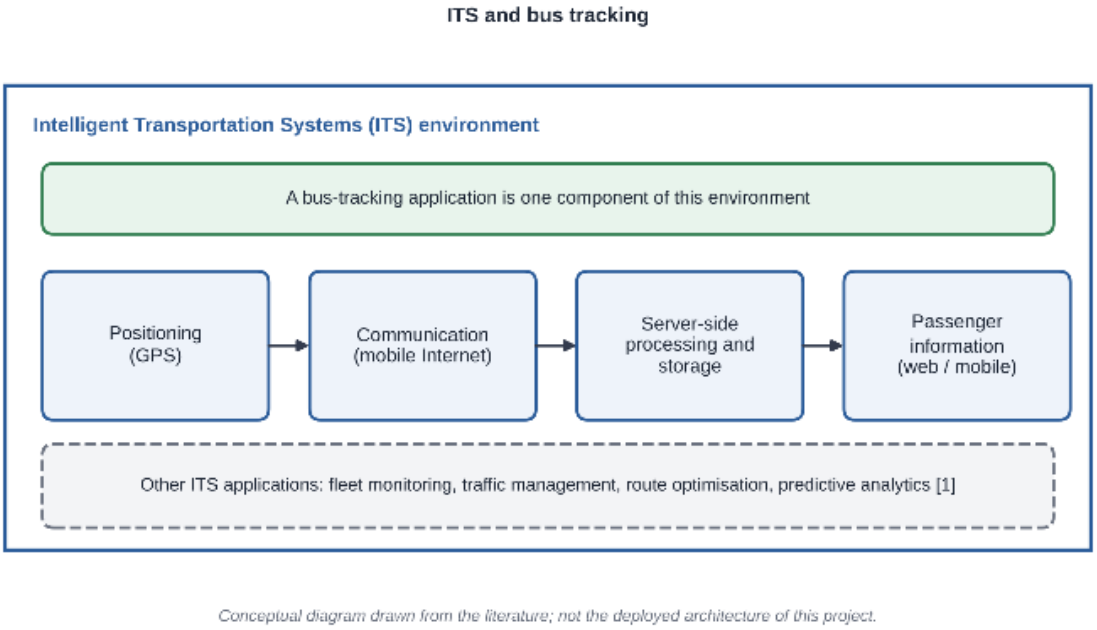


Figure 2.2.1.1: Intelligent Transportation Systems and bus tracking (conceptual)

2.2.2 GPS-Based Vehicle Tracking

GPS-based tracking determines the geographical position of a vehicle and communicates it to a central system. A typical process determines the position of the vehicle, transmits it, stores and processes it on a server, and presents the result to passengers. Khalid, Salleh and Samsudin built a Bus Tracking Information System on this pattern using consumer-grade GPS devices on buses, mobile Internet and a central server [2].

The usefulness of tracking depends on the freshness and accuracy of the data. Communication delays, outdated coordinates, positioning error, network availability and update frequency all influence what a passenger sees. A coordinate also needs context. It becomes useful to a passenger when it is associated with a route, a direction, a stop and ideally an arrival estimate. This is the reason the Bus Tracking System links bus position to structured route and stop data.

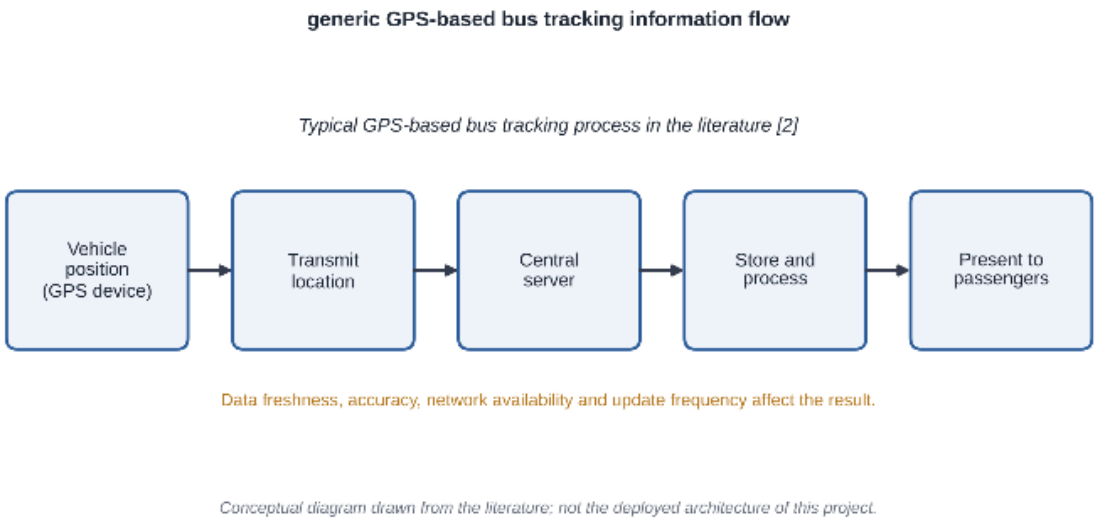


Figure 2.2.2.1: Generic GPS-based bus tracking information flow (conceptual)

2.2.3 Estimated Time of Arrival

Estimated Time of Arrival (ETA) is the expected time remaining before a vehicle reaches a stop. It tells a passenger more than a map position. An ETA can be derived in several ways: from a published schedule, from the vehicle position and an assumed or observed speed, or from a routing service that estimates travel time along a road. Khalid et al. update the arrival information using the bus location relative to the next stop [2]. Whatever the method, an ETA is an estimate rather than a guarantee. Liu and Miller found that real-time information does not always outperform simpler schedule-based strategies, partly because of delays and differences between reported and actual conditions [5]. The Bus Tracking System therefore treats its ETA as a conditional estimate and returns no value when its inputs are missing or invalid.

2.2.4 Real-Time Passenger Information

Real-time passenger information describes the current or expected state of a service. Unlike a static timetable it changes as vehicles move. The GTFS Realtime specification is an established example of how such dynamic information (vehicle positions, trip updates and service alerts) can be represented [6]. Passengers who do not know when the next bus will arrive experience uncertainty, and real-time information aims to reduce it. Watkins et al. studied the OneBusAway system and reported effects on both perceived and actual waiting time [3]. Liu and Miller later showed that the effect depends on how the information is delivered and used [5]. A cautious position follows: real-time information can reduce uncertainty, but a tracking system should not assume that it reduces actual waiting time in every setting.

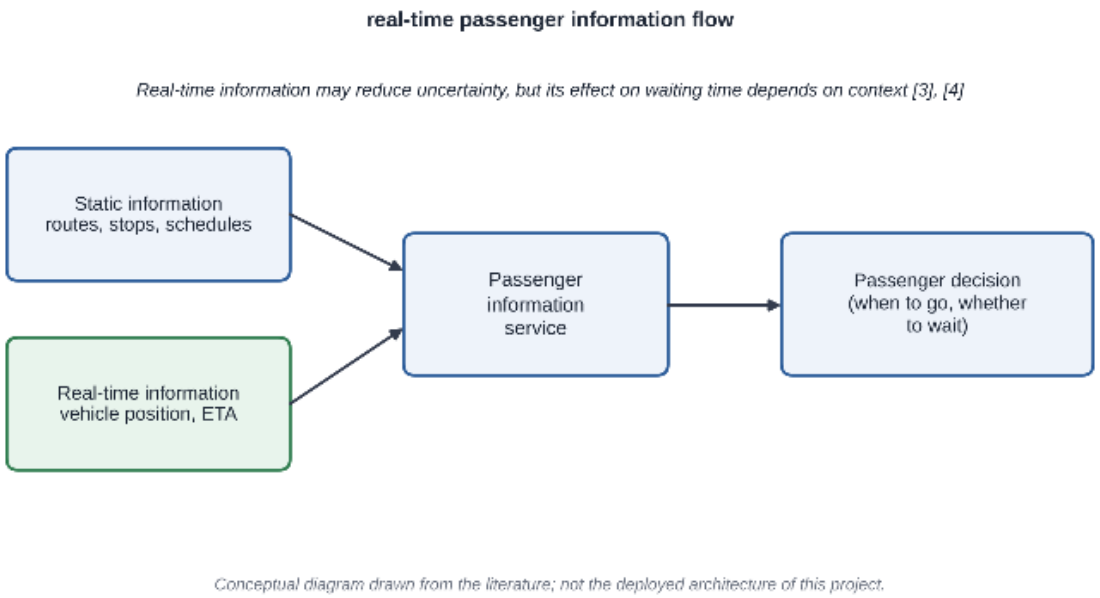


Figure 2.2.4.1: Real-time passenger information flow (conceptual)

2.2.5 Public Transport Data Standards

The General Transit Feed Specification (GTFS) is an open format for public transport information. Its static part describes agencies, routes, stops, trips and stop times in text files [7]. The Bus Tracking System imports four of these files: routes.txt, stops.txt, trips.txt and stop_times.txt, and it also persists the shape points from shapes.txt. GTFS Realtime extends the static model with trip updates, vehicle positions and service alerts [6]. The Bus Tracking System does not produce or consume a GTFS Realtime feed; bus positions arrive directly from the driver's device through the project's own API.

2.2.6 Maps and Road Routing

Bus tracking is geographical, so a passenger benefits from seeing buses, routes and stops on a map. The browser interfaces of the project use Leaflet with OpenStreetMap for this presentation. The system's use of routing technology is different. It asks the Open Source Routing Machine (OSRM) for the road route between two coordinates. The OSRM route service returns a distance in metres and a duration in seconds for a request made with longitude and latitude pairs [8]. The system uses these results to decide whether a bus is at a stop, whether it has left a stop and how long a bus will take to reach a stop.

2.2.7 Iterative Development and Object-Oriented Analysis

Transport information systems have several interacting parts and their requirements change as prototypes are tested. Iterative and incremental development supports gradual refinement of requirements, design and implementation. Larman describes iterative development and the Unified Process artefacts used to analyse and design object-oriented systems, including the domain model, system sequence diagrams and the design model [4]. Chapters 3 and 4 use

this approach. Larman's material establishes how to analyse and document a system; it does not itself establish what passengers need, which comes from the transport literature and the interface analysis described in Chapter 3.

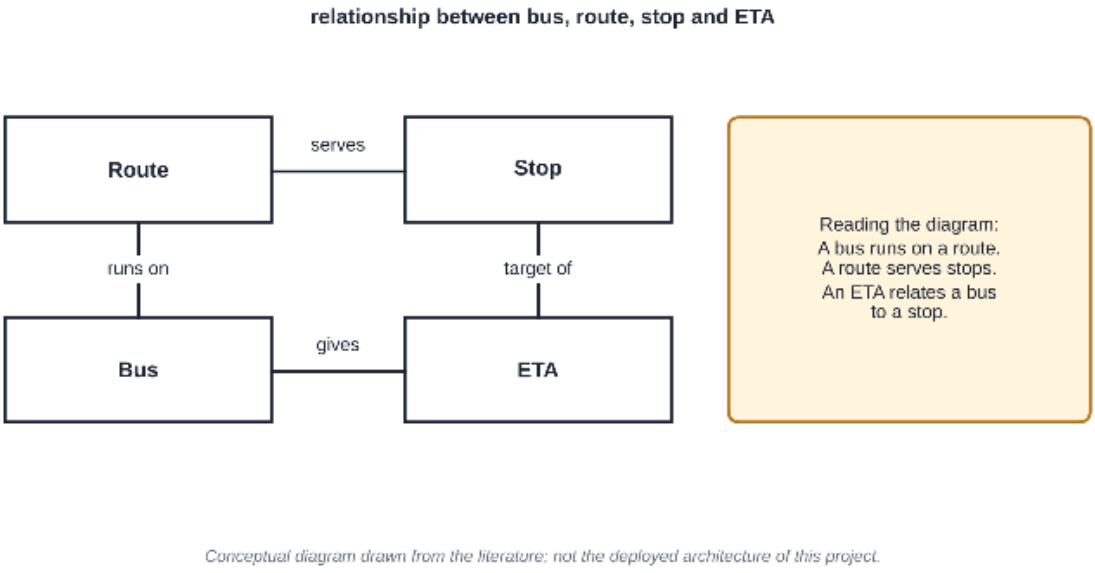


Figure 2.2.7.1: Relationship between bus, route, stop and ETA (conceptual)

2.3 Existing Systems / Related Work

2.3.1 Rea Vaya

Rea Vaya in Johannesburg is an African bus rapid transit (BRT) system that uses structured infrastructure and technology in an urban setting [9]. It is useful as African context. It is not a technical model for this project, because a BRT system has dedicated infrastructure, stations and institutional arrangements that a student prototype does not.

2.3.2 OneBusAway

OneBusAway is a passenger-facing real-time transit information system. Watkins et al. studied its effect on transit riders in the King County Metro service in the United States. It provided next-bus information through several channels, including a website, telephone, text messaging and smartphone applications [3]. Its relevance to this project is conceptual: it shows that tracking becomes more useful when location data is turned into passenger-oriented information. Its findings come from an established U.S. transit environment and should not be assumed to transfer to Lusaka.

2.3.3 A University Bus Tracking Information System

Khalid, Salleh and Samsudin developed a bus tracking information system for a university transport service using consumer-grade GPS and mobile Internet. Android devices on buses transmitted positions to a central server, and passenger clients displayed bus positions and

arrival information [2]. The study shows that tracking can be built from accessible technology. Its limitation for the present work is that it addresses a university service, not a public transport network with many routes. It is also the closest published precedent for this project's driver-side design, in which a device carried by the driver supplies the position.

2.3.4 Yango as an Interface Reference

Yango is a ride-hailing application. It was not a published source in this review. It was examined through the prior use of the application by the team members, as described in Section 3.3, and served as a reference for interaction concepts rather than for the transport data model.

2.3.5 Mobile and Browser Access

GSMA Intelligence describes the growth of mobile connectivity and mobile Internet in Sub-Saharan Africa [10]. This supports delivering transport information through the browser of a mobile device, which needs no application installation. The same source is regional, so it does not support a claim about the Internet access of Lusaka bus passengers, and the project makes none.

2.4 Comparative Analysis

Table 2.4.1 compares the reviewed systems and standards with the Bus Tracking System on the criteria that matter to this project. The comparison is based on the cited descriptions. It is not an evaluation of the systems.

Table 2.4.1: Comparison of reviewed systems and standards with the Bus Tracking System

System	Position source	Passenger information	Context	Limitation for this project
Rea Vaya [9]	Formal BRT infrastructure	Structured service information	Johannesburg, South Africa	BRT scale and infrastructure
OneBusAway [3]	Transit agency real-time data	Next-bus information by web, phone, text and apps	King County Metro, United States	Established agency environment
University bus tracking system [2]	Consumer-grade GPS on buses, mobile Internet	Bus positions and arrival information	University transport service	Not a multi-route public network
GTFS and GTFS Realtime [7], [6]	Data standards (feeds)	Static schedules; real-time positions, updates and alerts	International	A standard, not a tracking system; needs data feeds

System	Position source	Passenger information	Context	Limitation for this project
Bus Tracking System (this project)	Real GPS from the driver's device, validated against driver-bus assignment	Routes, destinations, boarding stops, waiting requests, OSRM-based ETA	Prototype for representative Lusaka routes	Prototype; not evaluated in the field

The comparison shows that the basic idea of GPS bus tracking is established. The contribution of this project is not the invention of tracking. It is the integration, in one working system, of driver-authorised real GPS updates, imported GTFS route data, a waiting-request model and a conditional OSRM-based ETA for Lusaka routes.

2.5 Research / Knowledge Gap

The literature supports five gaps, expressed cautiously because the reviewed sources cannot establish that no solution exists in Zambia.

- **Geographic.** Much of the cited empirical work originates outside Africa: Watkins et al. and Liu and Miller studied U.S. transit [3], [5], and Khalid et al. developed their system in Malaysia [2]. The ITS review also notes geographical concentration in the field [1].
- **Contextual.** Reported systems are often tied to a specific organisation, for example a university service [2]. This project considers representative public transport routes in Lusaka.
- **Implementation.** Many deployed systems rely on dedicated GPS equipment and agency data feeds. A student prototype can demonstrate the whole information flow with a GPS-enabled device used by the driver, without claiming an operational deployment.
- **Passenger information.** The effect of real-time information on waiting varies with context [3], [5]. The project therefore concentrates on providing information, not on claiming a behavioural outcome.
- **Interoperability.** GTFS and GTFS Realtime provide standard ways to exchange transport data [7], [6]. The project uses static GTFS for its data and leaves GTFS Realtime as future work.

The gap addressed by this project is therefore a localised implementation gap: a working, documented system that connects real driver-device GPS updates, GTFS-derived route and stop data, current bus state, waiting requests and a conditional ETA for Lusaka routes.

2.6 Conceptual or Theoretical Framework

The conceptual framework in Figure 2.6.1 links the inputs, processing and outputs of the system as suggested by the literature. Transport data (routes, stops and trips) and real GPS

location updates enter the system. The system validates who is allowed to act, maintains the state of the bus, stores data, and calls a road-routing service to estimate arrival. The results are presented to passengers by the browser interfaces. The three project features map onto this flow: Real-Time Bus Tracking relies on the GPS updates and stored bus state, Journey Planning relies on the transport data, and Arrival Time Prediction relies on the road-based ETA. The driver interface is the point at which the real GPS input enters the framework.

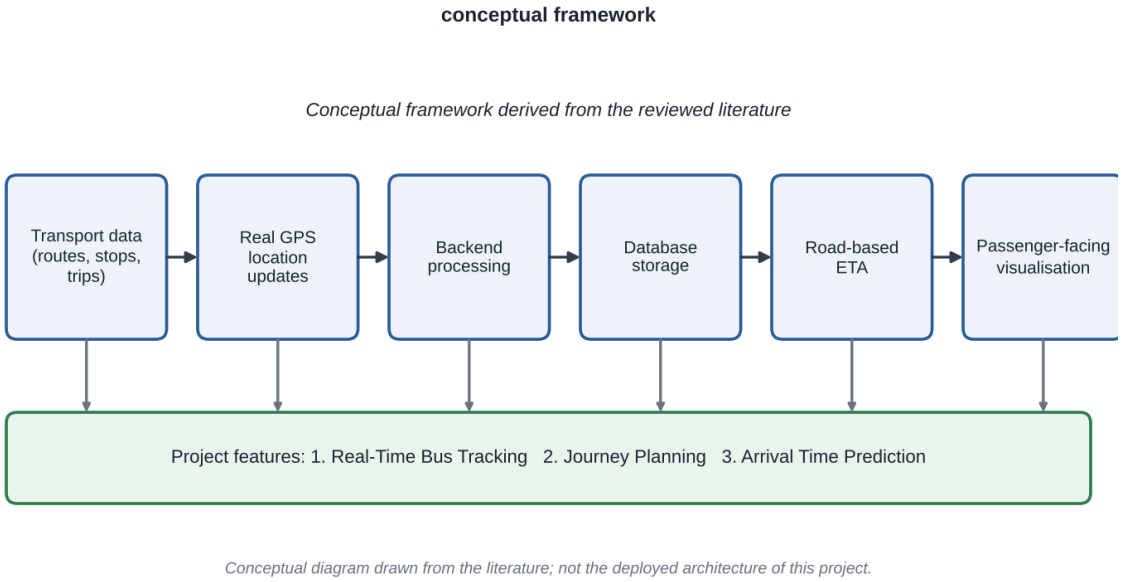


Figure 2.6.1: Conceptual framework derived from the reviewed literature

2.7 Chapter Summary

The literature establishes that GPS can supply vehicle position [2], that real-time passenger information can influence waiting experience [3] but not in every setting [5], and that ITS is an established area of work [1]. African examples such as Rea Vaya show that technology is used in formal urban transport [9]. GTFS provides a standard model for transport data [7]. The review supports building a localised prototype, treating ETA as a conditional estimate, and being cautious about claims on waiting time. The next chapter explains the methodology used.

CHAPTER 3: METHODOLOGY

3.1 Introduction

This chapter describes the methodology used to develop the Bus Tracking System for public transport in Lusaka. It explains the project's research design, the iterative and incremental approach used to design and implement the system, the architecture and principal components of the implemented solution, the database design, the user-interface design, and the software tools and technologies used.

The development process combined findings from the literature review with practical analysis of existing public-transport applications. The Rea Vaya bus system was considered during the literature review because it demonstrates the use of structured public-transport services in an urban context. The Yango application was also analysed through the team members' prior use of the application. Its interface and interaction patterns informed notes on potentially relevant functions, including location-based information, route and destination selection, vehicle information and passenger interaction.

The implemented system developed beyond the original proposal. Early work included simulated vehicle movement, but supervisor feedback led the team to replace simulation with real GPS coordinates obtained from a driver's GPS-enabled device. The final backend uses Django and Django REST Framework, GTFS-derived transport data, a PostgreSQL database, browser-based interfaces, OpenStreetMap and OSRM-supported road-distance processing. The methodology in this chapter therefore distinguishes between the original proposal, development prototypes and the functionality implemented in the official integration branch.

3.2 Research / Development Approach

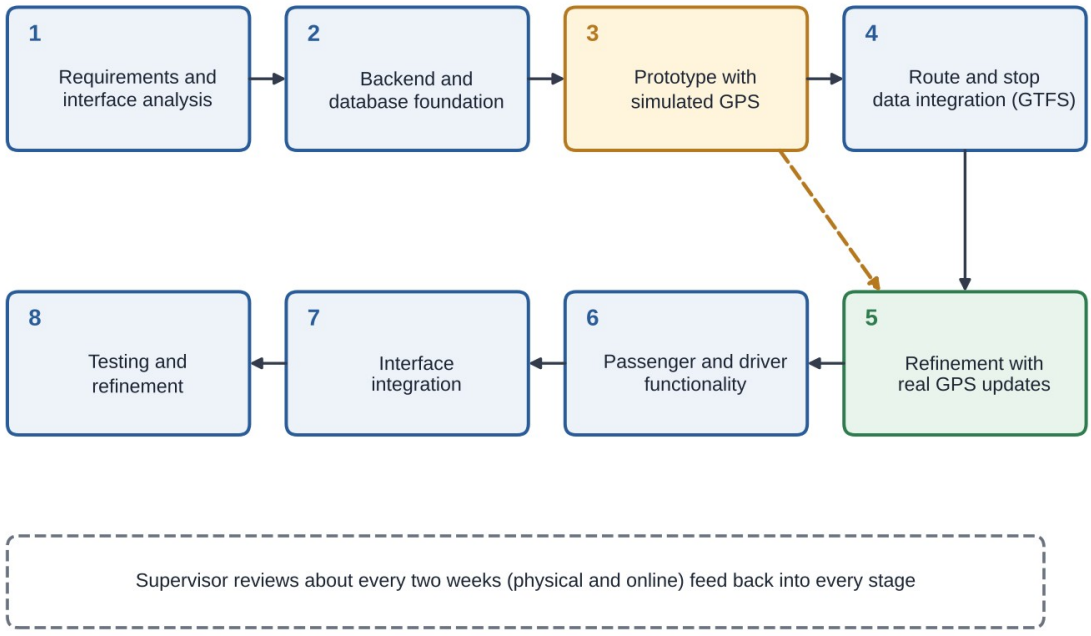
3.2.1 Iterative and Incremental Development

The project used an iterative and incremental approach. In an iterative process a limited subset of the requirements is designed, built and tested in each cycle, and early feedback is used to adapt later cycles; Larman presents this as a core idea of the Unified Process [4]. The approach suited this project because the system has several connected parts (data model, driver operations, GPS handling, waiting requests and arrival estimation) whose requirements became clearer as earlier parts were exercised.

The proposal had planned Agile Scrum with two-week sprints. In practice the group worked in iterations and reviewed progress with the supervisor roughly every two weeks, in person and online. The project did not keep formal sprint artefacts, so this report does not describe the process as Scrum. It describes it as iterative and incremental development with regular supervisor review.

The group divided the work by component. One member built the initial backend skeleton. The author and a second member then built the driver interface and the passenger interface on separate branches, and the backend was updated alongside them as their requirements emerged. The first member then integrated the branches and deployed the system. The stages are revisited in this report whenever the work of one component changed another. The prototype stage that used simulated GPS data is marked, because supervisor feedback led the group to replace it with real GPS updates (Stage 5). The simulation is not part of the final system.

Iterative and incremental development process followed by the project



Stage 3 → Stage 5: supervisor feedback replaced simulated GPS with real GPS updates (the simulation is not part of the final system).

Development was iterative: earlier stages were revisited as new requirements and feedback arrived.

Figure 3.2.1.1: Iterative and incremental development process of the Bus Tracking System

3.2.2 Work in Each Stage

Table 3.2.2.1 summarises the work done in each stage. The stage descriptions follow the group's own account of its process, and the final column adds the development sequence clarified for this report.

Table 3.2.2.1: Stages of development and the work done in each stage

No.	Stage	Work in the stage	Who
1	Requirements and interface analysis	Shared requirements analysis, including the study of the interaction concepts of the Yango application, from which the system's responsibilities were derived	Group
2	Server and database foundation	Django project, initial models and migrations, REST endpoints for routes, stops and buses	Kenneth Kaunda Musonda (initial backend skeleton)
3	Prototype with simulated GPS	Not part of the final system; replaced in Stage 5	Group
4	Route and stop data integration (GTFS)	Models for Route, Stop, Trip, StopTime and Shape; import commands that load the GTFS files	Not itemised in the available records
5	Refinement with real GPS updates	Authenticated GPS update endpoint, bus state, status and next-stop logic, and a driver interface that sends the device's location, replacing the simulation	Driver interface: author. Server-side changes: collaborative
6	Passenger and driver functionality	Passenger and driver accounts, driver-to-bus assignment checks, waiting requests and waiting counts, and the passenger and driver screens that use them	Driver screens: author. Passenger screens: Ngosa Muleba. Backend updates: collaborative
7	Interface integration	The browser interfaces connected to the API endpoints; branches merged into the integration branch	Kenneth Kaunda Musonda, with input from the interface developers
8	Testing and refinement	Automated tests of the server, tests on development branches, manual checks of the deployed flows, and the deployment configuration	Collaborative; deployment configuration: Kenneth Kaunda Musonda

Passenger-interface work in each stage. The passenger-interface report states the same stages with the passenger-interface work that each one contained.

Table 3.2.2.2 summarises the work done in each stage and identifies the passenger-interface work. The stage descriptions follow the group's own account of its process.

Table 3.2.2.2: Stages of development and the passenger-interface work in each stage

No.	Stage	Work in the stage	Passenger-interface work
1	Requirements and interface analysis	Shared requirements analysis, including the study of the interaction concepts of the Yango application	Joint contribution; derived the passenger flow of map, route, stop and request

No.	Stage	Work in the stage	Passenger-interface work
2	Server and database foundation	Django project, initial models and migrations, REST endpoints for routes, stops and buses	First map page showing buses returned by the bus endpoint
3	Prototype with simulated GPS	Not part of the final system; replaced in Stage 5	Map displayed simulated positions; replaced
4	Route and stop data integration (GTFS)	Models and import commands for the GTFS files	Route search, destination choice and listing of the stops of a trip
5	Refinement with real GPS updates	Authenticated GPS update endpoint, bus state and status logic; a driver interface that sends the device's location	Map shows real driver-device positions, with status and speed in the marker popup
6	Passenger and driver functionality	Passenger and driver accounts, assignment checks, waiting requests and waiting counts	Passenger sign-in and registration, boarding-stop selection, waiting-request creation and list, controls for board, cancel and complete
7	Interface integration	Browser interfaces connected to the API endpoints	Request list shows the arrival estimate; polling and refresh control
8	Testing and refinement	Automated server tests, manual checks of the deployed flows, deployment configuration	Manual verification of the passenger flows on the deployed system

The first three columns follow the group report. The right-hand column places the passenger-interface work within each stage.

Table 3.2.2.3: Stages of development and the driver-interface work in each stage

No.	Stage	Driver-interface work in the stage
1	Requirements and interface analysis	Shared analysis; the driver-side requirements (sign-in, assigned buses, bus operations, trip start, GPS reporting and waiting counts) were derived from it (Section 4.3)
2	Server and database foundation	No driver screen existed yet; the initial backend skeleton supplied the first endpoints that the interface developers could call
3	Prototype with simulated GPS	Not part of the final system; replaced in Stage 5
4	Route and stop data integration (GTFS)	The imported routes, destinations and ordered stops became the lists from which the driver chooses a route and destination to start a trip
5	Refinement with real GPS updates	The driver dashboard was built to send the device's position and speed to the authorised update endpoint, replacing the simulation
6	Passenger and driver functionality	Driver sign-in, assigned-bus selection, activation, deactivation and reset, trip start and the waiting counts by stop, with the supporting trip-lifecycle and stop-progression work on the Driver-Interface branch

No.	Stage	Driver-interface work in the stage
7	Interface integration	The dashboard was served by the Django project and connected to the endpoints in Table 4.5.3.1; the branches were merged by the integrating member
8	Testing and refinement	Manual checks of the driver flow on the deployed system (Section 6.3.3)

3.2.3 Evidence of Iteration in the Code

The migration history of the integration branch records some of the iterations. Migrations 0001 to 0009 show the data model being extended step by step: a Shape model and a bus status field were added in the second migration; a waiting request gained a link to a trip in the third and lost its earlier link to a route in the fourth; passengers were linked to Django users in the sixth and seventh; and the Driver and DriverBusAssignment models arrived in the eighth, followed by the removal of a driver name field in the ninth. The sequence shows the waiting-request design moving from a route-based to a trip-based model, which is the form documented in Chapter 4. The available records do not show which commits introduced migrations 0002 to 0009, or on which branch the Driver and DriverBusAssignment models were first added, so those migrations are not attributed to an individual member.

3.2.4 Proposal Compared with the Implemented System

The final system differs from the proposal in several respects. Table 3.2.4.1 records the differences so that the report does not describe the proposal as if it had been delivered.

Table 3.2.4.1: Proposal compared with the implemented system

Aspect	Proposal	Implemented system (integration branch)
Bus position source	Simulated GPS generated along routes	Real GPS latitude, longitude and speed sent by an authorised driver's device
Server framework	Django REST Framework	Django and Django REST Framework
Database	MySQL and Firebase	PostgreSQL, with its address read from DATABASE_URL
Passenger interface	React with Google Maps API	Browser interfaces using HTML, CSS and JavaScript with Leaflet and OpenStreetMap; no front-end framework
Real-time delivery	Firebase for real-time data	Standard HTTP requests, polled every 15 seconds; no WebSockets
Arrival estimate	ETA within ±2 minutes of expected	ETA from OSRM road duration; accuracy not evaluated
Operator dashboard	Admin dashboard for operators	Django administration site for records; no separate dashboard claimed

Aspect	Proposal	Implemented system (integration branch)
Process	Agile Scrum, two-week sprints	Iterative and incremental development with supervisor reviews about every two weeks
Evaluation	Unit tests, UAT with five or more users, SUS of at least 70, 10+ simulated buses, load times	Automated tests of the server and manual functional checks of the deployed flows; no UAT, SUS or performance results

3.2.5 Proposal Compared with the Implemented Passenger Interface

The final passenger interface differs from the proposal in several respects. Table 3.2.5.1 records the differences so that the report does not describe the proposal as if it had been delivered.

Table 3.2.5.1: Proposal compared with the implemented passenger interface

Aspect	Proposal	Implemented passenger interface
Bus position source	Simulated GPS generated along routes	Real GPS latitude, longitude and speed sent by an authorised driver's device and displayed on the map
Passenger interface technology	React with Google Maps API	Browser page using HTML, CSS and JavaScript with Leaflet and OpenStreetMap, delivered as a Django template, with no front-end framework
Real-time delivery	Firebase for real-time data	Standard HTTP requests; the page polls the API every 15 seconds; no WebSockets
Arrival estimate	ETA within ±2 minutes of expected	ETA from OSRM road duration shown with the waiting request; accuracy not evaluated
Route display	Routes shown on a map	Buses shown as markers on the map; routes, destinations and stops shown as lists
Evaluation	User acceptance testing with five or more users, SUS of at least 70, load times	Manual functional checks of the deployed passenger flows; no UAT, SUS or performance results

3.2.6 Proposal Compared with the Implemented Driver Interface

The proposal gave little detail about the driver side. Table 3.2.6.1 compares what the proposal assumed with what the driver interface implements.

Table 3.2.6.1: Proposal compared with the implemented driver interface

Aspect	Proposal	Implemented driver interface
Position source	Simulated GPS generated along routes	The browser's Geolocation API supplies the device's latitude, longitude and speed, which the dashboard sends to the server

Aspect	Proposal	Implemented driver interface
Delivery	Not specified for the driver; React and Firebase for real-time data	A Django template with inline script, the Fetch API and plain HTTP; no front-end framework, no WebSockets
Who may report a position	Not specified	An active driver with an active assignment to the bus; other updates are refused with HTTP 403
What the driver sees	Active passengers at pickup points	The number of passengers waiting at each stop of the bus's current trip, and the position, speed and status accepted by the server on a map
Evaluation	Unit tests and user acceptance testing	Manual checks of the deployed driver flow; no user study with drivers

3.3 Requirements / Data Collection

3.3.1 Requirements Analysis

Requirements analysis started from the transport problem described in Chapter 1 and from the interface analysis described below. The team did not conduct questionnaires, interviews, focus groups or any other direct requirements-gathering exercise with passengers, bus drivers or transport operators. The requirements in this report were therefore derived from the analysis of the transport problem, the examination of Yango, the review of related literature and systems, and technical refinement during implementation. They were treated as revisable. The change from simulated to real GPS data is the clearest case of a requirement revised after supervisor feedback. The three members contributed jointly to the problem analysis and requirements; the driver-side requirements were refined further by the author while the driver interface was developed.

3.3.2 Analysis of the Yango Application

A major source of functional and interface requirements was the analysis of the Yango application. Yango was the only existing transport application examined directly by the development team for this purpose. All members of the team had previously used the application as passengers, which gave them familiarity with its general workflow. During requirements analysis the team recorded notes on relevant features so that suitable concepts could be considered in the design of the Bus Tracking System.

The analysis concentrated on how a location-based transport application presents information and guides a user through a transport workflow. The features examined were the display of vehicle locations, the selection of destinations, the identification of pickup locations, the presentation of vehicle information, and the interaction between passengers and drivers.

3.3.3 Adapting Ride-Hailing Ideas to Predefined Bus Routes

The aim was never to reproduce Yango as a ride-hailing service. A ride-hailing passenger requests an individually assigned vehicle to travel between two freely chosen points. A bus passenger instead joins a service that follows a predefined route and stops at predefined stops. The interaction ideas were therefore adapted as follows.

Table 3.3.3.1: Adaptation of ride-hailing interaction ideas to a public bus service

Ride-hailing concept observed	Adaptation for public buses	Support in the system
Vehicle locations on a map	Active buses shown on a map	Bus records with current latitude, longitude, speed and status
Choosing a destination	Choosing a route and then one of its destinations	Route search and destination endpoints
Choosing a pickup point	Choosing a boarding stop from the ordered stops of a trip	Trip-stops endpoint using imported stop times
Requesting a ride	Creating a waiting request at a boarding stop	WaitingRequest model and lifecycle
Vehicle information and arrival time	Bus status and an arrival estimate for the waiting stop	Status logic and OSRM-based ETA
Driver side of the interaction	Driver activates a bus, starts a trip and sends GPS updates	Driver authorisation, trip and location endpoints, driver dashboard

The Yango analysis therefore influenced the functional requirements and the interaction concepts of the interface. It did not influence the underlying public transport data model, which comes from GTFS.

3.3.4 Related Systems

The interface analysis was supplemented by the review of related systems in Chapter 2. Rea Vaya was treated as a contextual reference for structured public transport in an urban African setting, not as a system to reproduce. Its operational and technological model is much larger than what the project could realistically implement.

3.3.5 Transport Data

Transport data came from a GTFS feed stored in the project's gtfs directory. The feed's own metadata names DigitalTransport4Africa as the publisher [11], gives the feed version as lusaka_city_pt, and lists three agency categories: minibuses of 14 to 18 seats, big buses and shared taxis. The system uses four of the files: routes.txt, stops.txt, trips.txt and stop_times.txt. The shapes.txt file is imported into a Shape table but is not used in route selection, trip progression or ETA. Counts computed directly from the files in the repository are given in Table 3.3.5.1.

Table 3.3.5.1: Contents of the GTFS files stored in the repository

Item	Count or observation
Routes (routes.txt)	68 routes: 55 minibus, 10 big bus and 3 shared taxi
Stops (stops.txt)	793 rows; one row has no coordinates and is skipped by the import
Trips (trips.txt)	408 trips: 136 for each of three start times (08:00:00, 11:00:00 and 15:00:00); 204 in each direction
Stop times (stop_times.txt)	9,219 rows covering 793 distinct stops
Shape points (shapes.txt)	13,431 points forming 136 shapes (imported; not used in route selection or ETA)
Representative trips	The 136 trips whose identifier ends in -08:00:00 (68 routes, two directions); they average about 23 stops, from 2 to 38

The three start-time variants of each trip carry the same stop sequence. The system therefore uses the 08:00:00 variant as a representative trip for each route and direction, both when a passenger plans a journey and when a driver starts a trip. A check of the files shows that the destinations reached by these 136 trips are the same as those of all 408 trips, so no destination is lost by using them. The consequence is that the system models route and direction, not a time-of-day schedule.

3.3.6 Transport Data as Used by the Passenger Interface

Transport data came from a GTFS feed stored in the project's gtfs directory. The feed's own metadata names DigitalTransport4Africa as the publisher [11], gives the feed version as lusaka_city_pt, and lists three agency categories: minibuses of 14 to 18 seats, big buses and shared taxis. Table 3.3.6.1 gives the counts that matter to the passenger interface, computed directly from the files in the repository.

Table 3.3.6.1: Contents of the GTFS files that the passenger interface draws on

Item	Count or observation
Routes (routes.txt)	68 routes: 55 minibus, 10 big bus and 3 shared taxi
Stops (stops.txt)	793 rows; one row has no coordinates and is skipped by the import
Trips (trips.txt)	408 trips: 136 for each of three start times (08:00:00, 11:00:00 and 15:00:00); 204 in each direction
Stop times (stop_times.txt)	9,219 rows covering 793 distinct stops
Representative trips	The 136 trips whose identifier ends in -08:00:00 (68 routes, two directions); they average about 23 stops, from 2 to 38

The three start-time variants of each trip carry the same stop sequence. The system therefore uses the 08:00:00 variant as a representative trip for each route and direction. For the passenger this means that choosing a route and a destination identifies exactly one trip, which

makes the choice unambiguous. The consequence is that the passenger interface models route and direction, not a time-of-day schedule.

3.4 System Architecture / Research Workflow

The system follows a client–server architecture. Browser interfaces send HTTP requests with JSON to the Django backend, which reads and writes its database and calls an external OSRM service for road routes. The architecture is described in Section 4.6 with its diagram, so it is not repeated here. In methodological terms the workflow was: analyse the problem and interface concepts; build the backend foundation and data foundation; import the GTFS data; replace the simulated movement with authorised real GPS updates; develop the driver and passenger interfaces while extending the backend to support them; integrate the branches; deploy; and test and refine. Figure 3.2.1.1 shows this sequence.

3.5 Tools and Technologies

Table 3.5.1 lists the principal technologies of the whole system, server and browser interfaces. The server versions are those pinned in requirements.txt of the integration branch; the Leaflet version is the one referenced by the page templates.

Table 3.5.1: Principal tools and technologies

Tool	Version	Use in the system
Python	—	Implementation language
Django [12]	6.0.6	Web framework, ORM, migrations, authentication, administration site, password-reset views
Django REST Framework	3.17.1	Serialisers and API views
psycopg[binary]	3.3.6	PostgreSQL driver used when DATABASE_URL is set
dj-database-url	3.1.2	Reads DATABASE_URL into the Django database setting
WhiteNoise	6.12.0	Serving static files
gunicorn	26.2.0	WSGI server (the WSGI application is bus_tracking.wsgi.application)
requests	2.34.0	HTTP calls to OSRM
django-anymail[brevo]	15.2	Sending password-reset e-mail through Brevo
PostgreSQL	—	Database of the deployed system
OSRM [8]	—	Road distance and duration, configured through OSRM_URL
HTML, CSS and JavaScript	—	Passenger interface, driver dashboard and landing page, written as Django templates with inline styles and scripts and no front-end framework; the scripts call the API with the browser's Fetch API and send the CSRF token

Tool	Version	Use in the system
Leaflet	1.9.4	Interactive map library for the passenger map and the driver's location map, loaded from the unpkg content-delivery network
OpenStreetMap tiles	—	Map background tiles requested by Leaflet in the browser
Browser Geolocation API	—	navigator.geolocation.watchPosition supplies the driver device's position, which the driver dashboard sends to the server
Visual Studio Code, Windows	—	Development tools and operating system used by the group
Git and GitHub	—	Source control; the integration branch is the authoritative branch for the integrated system; the author's work is on the Driver-Interface branch
Render and Brevo	—	Hosting of the web application and e-mail delivery for password reset

3.6 Ethical and Legal Considerations

No participants were recruited, so no consent procedure, questionnaire or interview data was involved. The system does store personal data: a username, an e-mail address and a hashed password for each account, and a display name for passengers. It also stores the position and speed sent from a driver's device for a bus. The following considerations apply.

- **Credentials and secrets.** Passwords are stored by Django's authentication system as hashes. The secret key, database address, e-mail service key and routing address are read from environment variables, not written in the source. Debugging is off unless the debug flag is explicitly set to true, and when it is off the settings require the secret key to be set.
- **Access control.** Driver operations check driver and assignment status. A passenger's waiting-request endpoints are scoped to that passenger. Section 6.6 records the remaining security limitations.
- **Location data.** The stored location is that of a bus, supplied by its driver, and only the latest value is kept; there is no history table. The driver's browser asks the device owner for permission before location is read. The screenshots in Chapter 6 show the location reported by the device used for the manual checks, which was a test device and not an operating bus.
- **Data licensing.** The GTFS feed names its publisher in its metadata. This report does not examine its licence terms, which should be checked before any redistribution or operational use.
- **Third-party software.** The libraries listed in Table 3.5.1 are used under their own licences, and the sources in Chapter 2 are cited.

3.6.1 Considerations for the Passenger Interface

No participants were recruited, so no consent procedure, questionnaire or interview data was involved. The system does store personal data: a username, an e-mail address and a hashed password for each account, and a display name for passengers. A passenger's waiting requests record the route, the boarding stop and times. The following considerations apply to the passenger interface.

- **Credentials.** The sign-in and registration forms of the passenger page send credentials to the server, which stores passwords as hashes. The page does not store a password itself.
- **Access control.** A passenger's waiting-request endpoints are scoped to that passenger, so the request list shows only the signed-in passenger's own requests.
- **Location data.** The location shown on the passenger map is that of a bus, supplied by its driver. In the checks reported in Chapter 6 the location came from a test device and not from an operating bus.
- **Data licensing.** The GTFS feed names its publisher in its metadata. This report does not examine its licence terms, which should be checked before any redistribution or operational use.
- **Third-party software and services.** Leaflet, OpenStreetMap tiles and the other libraries are used under their own licences and terms, and the sources in Chapter 2 are cited.

3.6.2 Considerations for the Driver Interface

- **Credentials.** The driver sign-in form sends the username and password to the server, which stores passwords as hashes and limits sign-in to five attempts per minute for each client address.
- **Location data.** The dashboard reads the position of the driver's device only after the browser has asked for, and received, the driver's permission, and only while the driver has started GPS tracking. The server keeps only the latest position of the bus.
- **Access.** The dashboard does not decide who may operate a bus: the server refuses a position or an operation unless the driver is active and actively assigned to the bus (Section 4.4).
- **Test device.** The screenshots in Chapter 6 show the position of a test device, not of an operating bus.

3.7 Evaluation Method

Evaluation was limited to functional verification at three levels. First, the integrated system's repository contains automated tests, written with Django's test framework, for the arrival-time and waiting-overview logic, access control, registration, login throttling and the debug

default; the command python manage.py test runs them [13]. Second, the author ran the test suite of the Driver-Interface development branch, which is reported separately because it is a different code state from the integrated system (Section 6.3). Third, the passenger interface, the driver dashboard and the server were checked together by hand on the deployed system (Section 6.3). The separate command python manage.py check only validates the project configuration and is not a test run. The method does not include user acceptance testing, usability scoring, field trials, load tests or an accuracy comparison of ETA values with observed arrival times, because none was carried out.

3.8 Chapter Summary

The methodology combined requirements analysis, an interface analysis of Yango adapted to predefined bus routes, a review of related systems, iterative development with supervisor review, and verification by automated tests and manual checks. The development sequence was an initial backend skeleton, parallel interface development with backend updates, then integration and deployment. The supervisor’s feedback replaced simulated GPS with real GPS. The next chapter sets out the analysis and design of the system.

CHAPTER 4: SYSTEM ANALYSIS AND DESIGN

4.1 Introduction

This chapter presents the analysis and design of the Bus Tracking System as a whole. It follows the Unified Process artefacts described by Larman [4]. Several diagrams are used, and each answers a different question, so they must not be confused with each other. The analysis and design are shared group work: the three members contributed to the problem analysis and requirements, and the diagrams in this chapter were prepared for the group report to match the implemented code. They are reused here with the source figure numbers identified. Where the design concerns the driver interface, which is the author's principal individual contribution, the design is described in more detail in Section 4.8. Table 4.1.1 states the purpose of each artefact and where it appears.

Table 4.1.1: Analysis and design artefacts used in this report

Artefact	What it shows	Level	Section
Use-case diagram	Goals of the human actors and the external system, and what the system offers them	Requirements	4.5
System sequence diagrams (SSDs)	For one scenario, the external events from an actor to the system treated as a black box, and its observable responses	Analysis	4.5
Architecture diagram	The parts of the implemented system and what it communicates with	Design	4.6
Domain model	Meaningful concepts of the problem domain and the associations between them, with no implementation detail	Analysis (conceptual)	4.7.1
Entity-relationship (ER) diagram	The persisted tables, keys and relationships actually implemented	Design (data)	4.7.2
Design model	Design classes (here, modules), their operations, dependencies and interactions that realise the system operations	Design (software)	4.9

4.2 Stakeholder and User Analysis

Table 4.2.1 identifies the people and external systems that interact with, or are affected by, the system. The requirements analysis relied on the problem analysis and interface study described in Chapter 3. No stakeholder interviews were conducted.

Table 4.2.1: Stakeholders and their needs

Stakeholder	Need	How the system serves it
Passenger	Find a route, choose a destination and boarding stop, register that they are waiting, and know when the bus will arrive	Passenger interface; route, destination and stop endpoints; waiting requests; ETA
Driver	Sign in, operate only the bus they are assigned, start a trip, report position, and see where passengers are waiting	Driver dashboard; driver login; assignment check; trip start; GPS update; waiting counts
Administrator	Maintain buses, drivers, assignments and transport records	Django administration site; import commands
OSRM routing service (external)	Receive route requests and return road distance and duration	The distance service calls the OSRM route API
Interface developers (group members)	Stable endpoints returning the data their screens need	JSON API under /api/
Supervisor and examiners	A documented, verifiable implementation	This report and the integration branch

4.3 Functional Requirements

The functional requirements are grouped under the three features of the system, in the order Real-Time Bus Tracking, Journey Planning and Arrival Time Prediction. Next-stop determination is not listed as a feature, because passengers never use it directly; it is internal logic that supports Real-Time Bus Tracking and Arrival Time Prediction. Table 4.3.1 lists the requirements implemented by the system.

Table 4.3.1: Functional requirements implemented by the system

ID	Feature	Requirement
FR1	Real-Time Bus Tracking	Accept latitude, longitude and speed from an authenticated driver for a bus to which the driver is actively assigned
FR2	Real-Time Bus Tracking	Store the current coordinates, speed, status and last-update time of each bus
FR3	Real-Time Bus Tracking	Derive the bus status (IDLE, IN_TRANSIT or AT_STOP) from the position relative to the current stop
FR4	Real-Time Bus Tracking	Keep the current trip and current stop of the bus and advance to the next stop when the bus has left the current one
FR5	Real-Time Bus Tracking	Allow an authorised driver to activate or deactivate a bus, to reset its operational state and to start a trip for a chosen route and destination
FR6	Real-Time Bus Tracking	Provide bus records (registration number, status, position, speed, last update) to clients

ID	Feature	Requirement
FR7	Driver access	Authenticate drivers through session login and refuse accounts that are not driver accounts or that are inactive
FR8	Driver access	Refuse any bus operation or GPS update unless there is an active driver-to-bus assignment (HTTP 403 otherwise)
FR9	Driver access	List the buses actively assigned to the signed-in driver
FR10	Journey Planning	Search routes by name
FR11	Journey Planning	List the destinations of a route, and the ordered stops of the trip for a route and destination
FR12	Journey Planning	Register, sign in and sign out passengers
FR13	Journey Planning	Create a waiting request for a trip and a stop served by that trip; allow one active request (WAITING or ON_BOARD) per passenger
FR14	Journey Planning	Support the waiting-request lifecycle: board, cancel, complete and expire
FR15	Journey Planning	Return the number of waiting passengers for a trip and stop, and for each stop of a bus's current trip
FR16	Arrival Time Prediction	Return an ETA in minutes for a passenger's own WAITING request, or null when no valid estimate exists
FR17	Arrival Time Prediction	Return the road-based duration from a given bus to a given stop
FR18	Data	Import routes, stops, trips, stop times and shapes from GTFS files repeatably
FR19	Account support	Support password reset by e-mail

Requirements FR1, FR5, FR7, FR8, FR9 and FR15 are the ones that the driver interface exercises directly. The interface requirements that the group derived for the driver dashboard were: sign in as a driver; see only the buses assigned to the account; activate, deactivate or reset a bus; choose a route and destination and start a trip; send the device's position and speed continuously while tracking; see the reported location, speed and status; and see the number of passengers waiting at each stop of the trip. These interface requirements are realised through the endpoints in Table 4.5.3.1.

4.3.1 Functional Requirements of the Passenger Interface

The functional requirements are grouped under the three features of the system: Real-Time Bus Tracking, Journey Planning and Arrival Time Prediction. Table 4.3.2 lists the requirements placed on the passenger interface. Each is linked to the group-level requirement (FR number) that it realises. Table 4.3.3 lists the server requirements on which they depend.

Table 4.3.2: Functional requirements of the passenger interface

ID	Feature	Requirement	Group FR
PR1	Real-Time Bus Tracking	Show the active buses as markers on a map, with a popup giving the registration number, status and speed of the bus	FR6
PR2	Real-Time Bus Tracking	Refresh the bus data automatically every 15 seconds and on the passenger's request	FR2, FR6
PR3	Journey Planning	Let a passenger register, sign in and sign out, and reach the password-reset page	FR12, FR19
PR4	Journey Planning	Let a passenger search routes by typing part of a name and select one of the matches	FR10
PR5	Journey Planning	List the destinations of the selected route and let the passenger choose one	FR11
PR6	Journey Planning	Show the ordered stops of the trip for the chosen destination and let the passenger choose a boarding stop, confirming the choice	FR11
PR7	Journey Planning	Create a waiting request for the chosen trip and stop, confirm that it was created, and respect the rule that a passenger has one active request at a time	FR13
PR8	Journey Planning	List the signed-in passenger's own requests with route, boarding stop, time and status	FR13
PR9	Journey Planning	Offer controls to mark a request as boarded, cancel it or complete it, according to its status	FR14
PR10	Arrival Time Prediction	Show the estimated arrival time for a request in the WAITING status when the server supplies one	FR16
PR11	All	Run in a standard web browser with no installation, as one page with overlay panels over the map	—

Table 4.3.3: Server requirements the passenger interface depends on

Group FR	Server requirement	Used by
FR2, FR6	Store and provide bus records (registration number, status, position, speed, last update)	PR1, PR2
FR10	Search routes by name	PR4
FR11	List the destinations of a route and the ordered stops of the trip for a route and destination	PR5, PR6
FR12	Register, sign in and sign out passengers	PR3
FR13	Create a waiting request for a trip and a stop served by that trip; allow one active request per passenger	PR7, PR8
FR14	Support the waiting-request lifecycle: board, cancel, complete and expire	PR9
FR16	Return an ETA in minutes for a passenger's own WAITING request, or null when no valid estimate exists	PR10

Group FR	Server requirement	Used by
FR19	Support password reset by e-mail	PR3

4.3.2 Functional Requirements of the Driver Interface

The requirements that the driver dashboard realises are listed in Table 4.3.4, together with the server requirement (Table 4.3.1) that each one relies on.

Table 4.3.4: Functional requirements of the driver interface

ID	Requirement	Server requirement relied on
DR1	The driver signs in with a username and password and sees a distinct message when the credentials are wrong or the account is not an active driver	FR7
DR2	A driver who reopens the page with a valid session is shown the dashboard without signing in again, and can log out	FR7
DR3	The dashboard lists the buses actively assigned to the account and shows the state of the selected bus	FR9
DR4	The driver can activate, deactivate and reset the operational state of the selected bus	FR5, FR8
DR5	The driver can choose a route and a destination from the route lists and start a trip	FR5, FR10, FR11
DR6	While tracking is on, the dashboard sends the latitude, longitude and speed of each position fix of the device and shows an error if the device cannot supply one	FR1, FR8
DR7	The dashboard shows the position, speed and status that the server accepted and marks the position on a map	FR2, FR3
DR8	The dashboard shows the number of passengers waiting at each stop of the bus's current trip	FR15
DR9	The driver can stop tracking at any time	None (browser only)

4.4 Non-Functional Requirements

No numerical performance or scale target is claimed, because none was evaluated. The non-functional requirements below are design requirements and the evidence for each is stated.

Table 4.4.1: Non-functional requirements and design response

Quality	Requirement	Design response and evidence
Security	Driver operations must be restricted to authorised drivers; passenger request data must be scoped to its owner	Session authentication; the authorisation helper get_authorized_bus_for_driver; waiting-request queryset filtered by the signed-in passenger. Gaps are listed in Section 6.6
Reliability	Failure of the routing service must not crash a GPS update or produce a false estimate	Services catch routing request failures; ETA returns null when no valid duration exists
Data integrity	Prevent duplicate assignments, stop sequences and shape points	Unique constraints on (driver, bus), (trip, stop_sequence) and (shape_id, shape_pt_sequence); atomic transactions for bus state changes
Maintainability	Business rules separated from request handling	Function-based service modules; models for operational state; management commands for imports
Portability and deployability	Run locally and on a hosting platform with environment-based settings	DATABASE_URL, DJANGO_SECRET_KEY and other settings read from the environment; WSGI application bus_tracking.wsgi.application
Simplicity of transport	Work over standard web infrastructure	Plain HTTP requests and JSON; no WebSockets
Usability on a driver's device	Run in a mobile browser without installation, using the device's location service	Browser-based dashboard using the Geolocation API; not evaluated with drivers
Performance	No target claimed	A GPS update can call OSRM up to twice (Section 4.9); not measured

4.4.1 Non-Functional Requirements as They Apply to the Passenger Interface

No numerical performance or scale target is claimed, because none was evaluated. The non-functional requirements below are design requirements and the evidence for each is stated.

Table 4.4.2: Non-functional requirements and design response

Quality	Requirement	Design response and evidence
Accessibility of access	The passenger must be able to use the system without installing anything	Browser page delivered by the server; no native application (Section 1.5)
Honesty of information	The interface must not present an invented arrival time	The estimate is shown only for a WAITING request and only when the server returns a value; the server returns null otherwise (Section 4.9.9)
Currency of information	The bus positions shown should be recent	Polling every 15 seconds and a manual refresh control; the age of the data also depends on the driver's device and the network

Quality	Requirement	Design response and evidence
Clarity of interaction	The route-to-request flow should be a short, ordered sequence	Four steps (search, destination, stop, request) in one panel, with a confirmation message after the stop and the request (Section 4.8.2.3)
Security	A passenger must see only their own requests; credentials must not be handled by the page beyond sending them	Session cookie and CSRF token on every call; the server filters requests by the signed-in passenger
Simplicity of transport	Work over standard web infrastructure	Plain HTTP requests and JSON; no WebSockets
Performance	No target claimed	Not measured

4.4.2 Non-Functional Requirements as They Apply to the Driver Interface

Table 4.4.3: Non-functional requirements and design response for the driver interface

Quality	Requirement	Design response and evidence
Security	Only an authorised driver may report the position of a bus	The dashboard relies on the server's assignment check; it shows the refusal (HTTP 403) and does not decide who may operate a bus
Currency of information	The position held by the server should follow the device	A position is sent for each fix that the browser delivers; the browser is asked for high accuracy with a maximum cached-position age of 1 second and a 15-second timeout; the age of the data also depends on the device and the network
Reliability	A slow reply must not pile up requests	Only one update request is outstanding at a time, and a fix that arrives while one is pending is skipped
Usability on a driver's device	Run in a mobile browser without installation	Browser-based dashboard using the device's location service; the browser asks for permission; not evaluated with drivers
Feedback	The driver must know whether an action worked	Messages for each operation, for GPS errors and for refused or failed requests
Performance and battery use	Not examined	No measurement was made

4.5 Use Cases / User Stories

4.5.1 Use-Case Diagram

Figure 4.5.1.1 shows the use cases of the system. The human actors, Passenger and Driver, are on the left. The OSRM routing service is a supporting external system outside the system boundary, on the right. Only goals understandable to the users are shown as use cases. The use case Retrieve ETA includes Calculate ETA, which in turn includes Obtain Road Duration; the last step is performed by OSRM. The diagram describes the system as a whole.

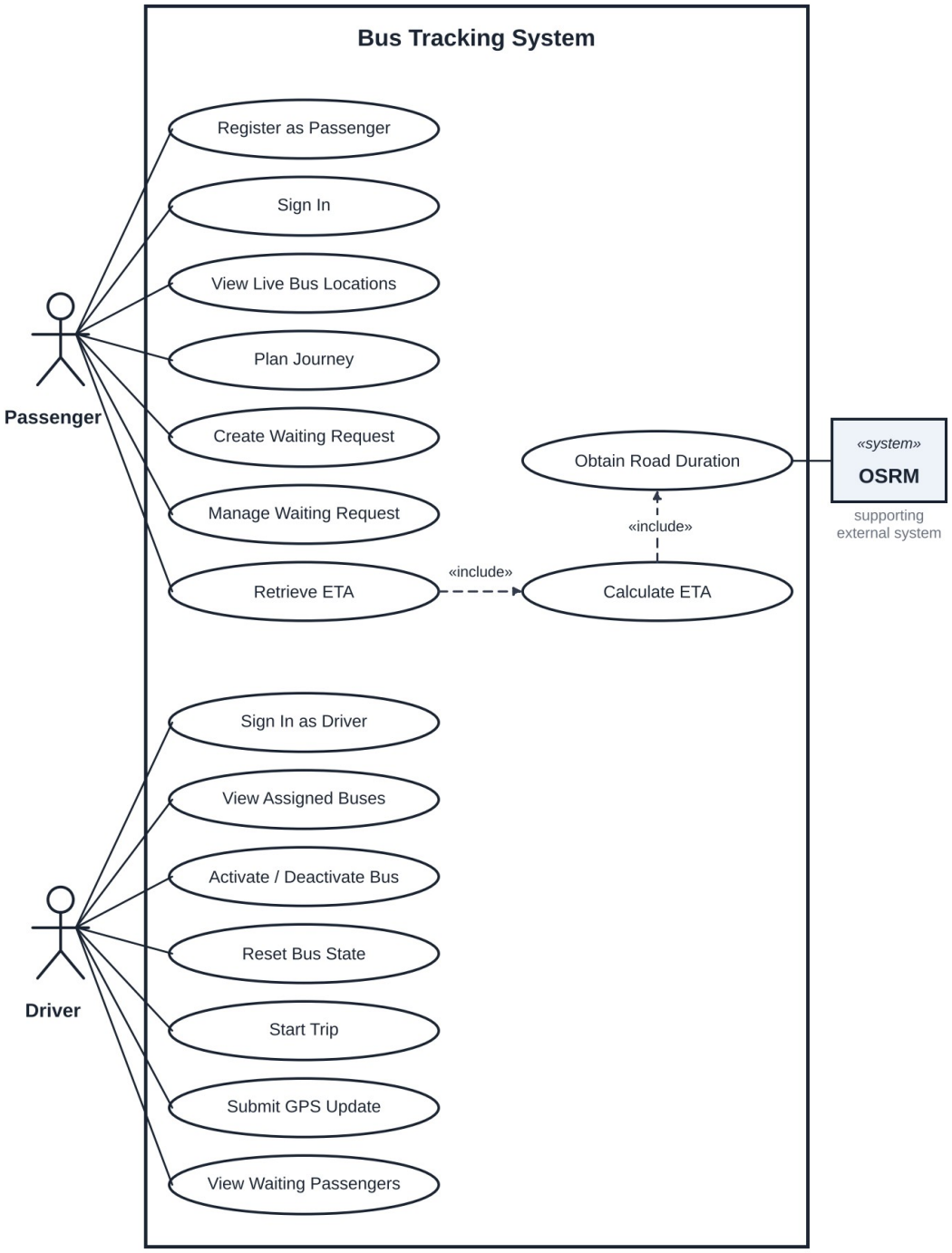


Figure 4.5.1.1: UML use-case diagram of the Bus Tracking System

4.5.2 Use-Case Summaries

Table 4.5.2.1 gives brief descriptions of the use cases that exercise the core logic of the system, in the casual format described by Larman [4].

Table 4.5.2.1: Brief descriptions of the principal use cases

Use case	Actor	Description
Start Trip	Driver	The driver chooses a route and destination for an activated bus. The system checks that the driver is active and assigned, selects the trip, sets the bus's current trip and first stop and reports the trip started
Submit GPS Update	Driver	While a trip is in progress, the driver's device sends latitude, longitude and speed. The system stores them, updates the next stop and status and returns the current state
View Waiting Passengers	Driver	The system returns the number of waiting passengers for each stop of the bus's current trip
Plan Journey	Passenger	The passenger searches a route, selects a destination and sees the ordered stops of the trip so that a boarding stop can be chosen
Create Waiting Request	Passenger	The passenger registers that they are waiting at a boarding stop of a trip. The system allows only one active request per passenger
Manage Waiting Request	Passenger	The passenger marks the request as boarded, cancels it, or completes the trip
Retrieve ETA	Passenger	For a WAITING request, the system calculates the arrival time of the nearest eligible bus using the road duration from OSRM, or reports that no estimate is available

Passenger use cases and the interface that supports them. Table 4.5.2.2 gives brief descriptions of the passenger use cases in the casual format described by Larman [4], with the part of the interface that supports each. The driver use cases (Start Trip, Submit GPS Update and View Waiting Passengers) belong to the driver dashboard and the server.

Table 4.5.2.2: Principal passenger use cases

Use case	Actor	Description	Interface part
View Active Buses	Passenger	The passenger opens the passenger page and sees the active buses as markers; a marker's popup gives its registration number, status and speed	Live map
Plan Journey	Passenger	The passenger searches a route, selects a destination and sees the ordered stops of the trip so that a boarding stop can be chosen	Plan Trip panel
Create Waiting Request	Passenger	The passenger registers that they are waiting at a boarding stop of a trip. The system allows only one active request per passenger	Plan Trip panel
Manage Waiting Request	Passenger	The passenger marks the request as boarded, cancels it, or completes the trip	My Requests sheet
Retrieve ETA	Passenger	For a WAITING request, the system calculates the arrival time of the nearest eligible bus using the road duration from OSRM, or reports that no estimate is available	My Requests sheet

Use case	Actor	Description	Interface part
Register and Sign In	Passenger	The passenger creates an account or signs in so that requests can be made and owned	Account area of the Plan Trip panel

Driver use cases and the dashboard that supports them. Table 4.5.2.3 relates the driver use cases of Table 4.5.2.1 to the parts of the driver dashboard that realise them.

Table 4.5.2.3: Principal driver use cases and the supporting part of the dashboard

Use case	Part of the driver dashboard	Operation
Sign In as Driver	Driver login panel	loginDriver (and the current-driver check on page load)
View Assigned Buses	Assigned bus selection	listAssignedBuses
Activate / Deactivate Bus; Reset Bus State	Activate, Deactivate and Reset operational state buttons	activateBus, deactivateBus, startFresh
Start Trip	Start a trip panel (Route, Destination, Start trip)	searchRoutes, getRouteDestinations, startTrip
Submit GPS Update	GPS tracking panel (Start GPS tracking)	updateBusLocation
View Waiting Passengers	Waiting requests by stop panel	getWaitingOverview

4.5.3 System Sequence Diagrams

A system sequence diagram (SSD) shows, for one scenario of a use case, the events that external actors generate and the system operations they trigger, with the system treated as a black box [4]. An event is something that happens and requires a system response, and events trigger system operations [4]. The SSDs therefore show only external actors, the events they cause and the observable responses. They do not show database tables, modules or internal calls, which belong to the design model in Section 4.9. Operation names are given at the level of intent and are derived from the endpoints of the implementation (Table 4.5.3.1).

Two separate SSDs are given. Figure 4.5.3.1 covers the scenario in which an authorised driver starts a trip and updates the GPS position of the bus. This is the scenario realised by the driver dashboard. OSRM does not appear in it, because the driver is not an actor of the routing exchange; it is an internal dependency of the system. Figure 4.5.3.2 covers the scenario in which a passenger plans a journey, creates a waiting request and retrieves the ETA. OSRM appears there because it provides the road duration that determines the response.

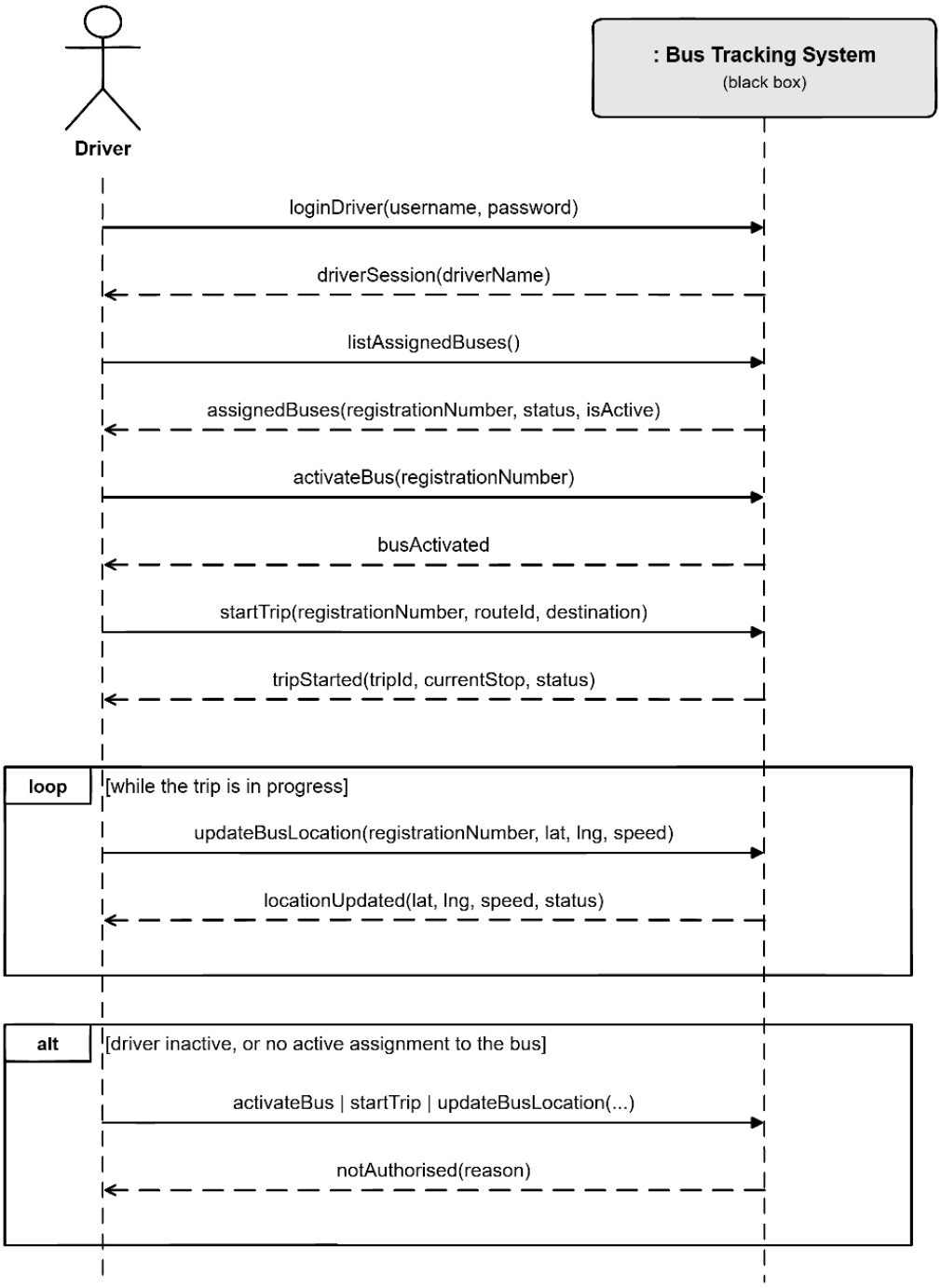


Figure 4.5.3.1: System sequence diagram: authorised driver starts a trip and updates the bus GPS position

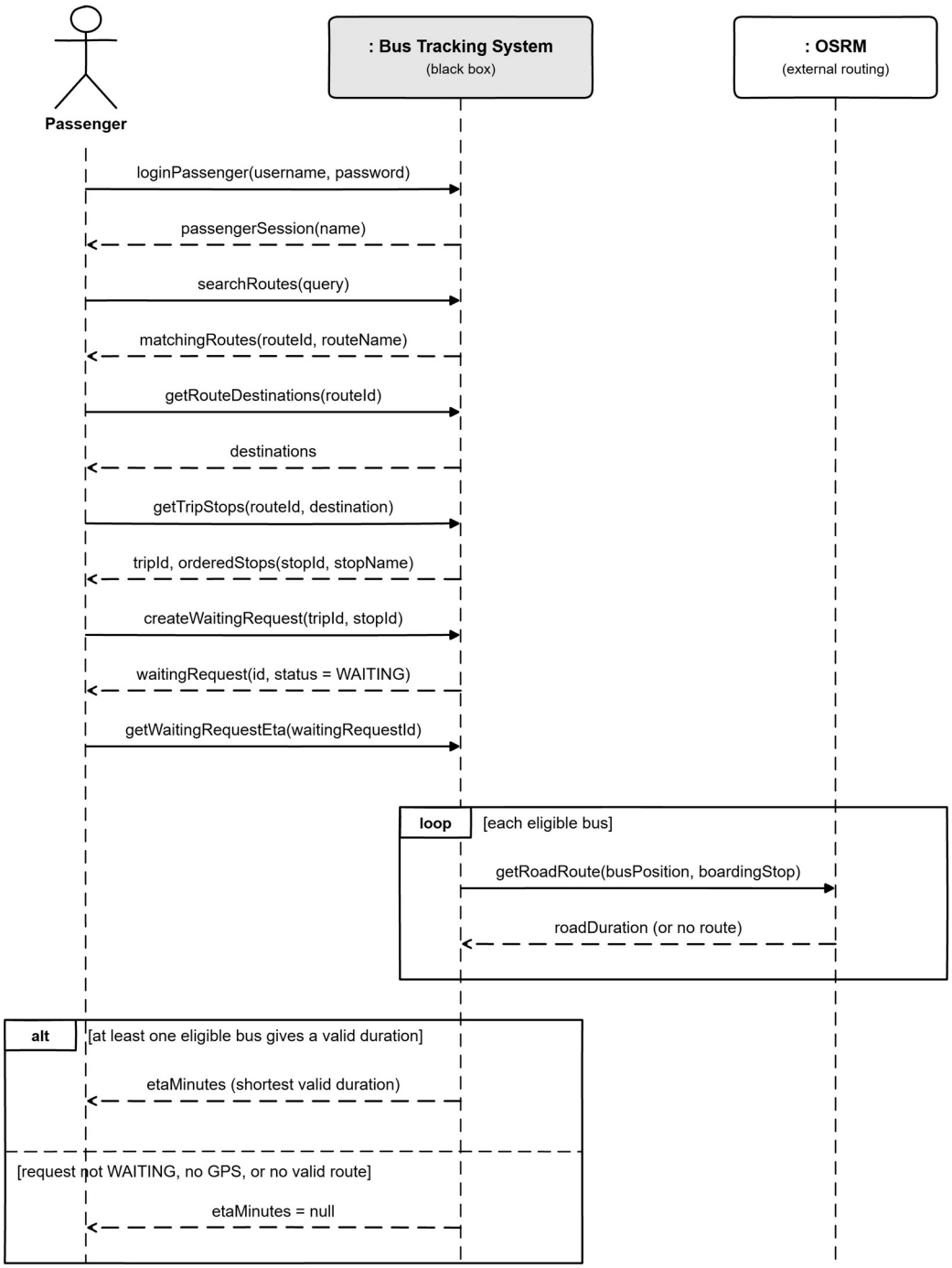


Figure 4.5.3.2: System sequence diagram: passenger plans a journey, creates a waiting request and retrieves the ETA

Table 4.5.3.1: System operations and the endpoints that implement them

System operation	Method	Endpoint (under /api/)	Used by
loginDriver	POST	auth/driver/login/	Driver dashboard
listAssignedBuses	GET	auth/driver/buses/	Driver dashboard

System operation	Method	Endpoint (under /api/)	Used by
activateBus	POST	buses/<registration_number>/activate/	Driver dashboard
startTrip	POST	buses/start-trip/	Driver dashboard
updateBusLocation	POST	buses/<registration_number>/location/	Driver dashboard
loginPassenger	POST	auth/login/	Passenger interface
searchRoutes	GET	routes/search/?q=	Passenger interface (and driver route choice)
getRouteDestinations	GET	routes/<route_id>/destinations/	Passenger interface (and driver destination choice)
getTripStops	GET	routes/<route_id>/stops/?destination=	Passenger interface
createWaitingRequest	POST	waiting-requests/	Passenger interface
getWaitingRequestEta	GET	waiting-requests/<id>/eta/	Passenger interface

The driver dashboard also calls auth/driver/me/ (restore the session), auth/driver/logout/, buses/<registration_number>/driver-session/ (trip and last-position record), buses/<registration_number>/deactivate/, buses/<registration_number>/start-fresh/ (reset operational state) and buses/<registration_number>/waiting-overview/ (waiting counts by stop). The available records do not show which of these driver-side endpoints were introduced or modified on the Driver-Interface branch.

4.6 System / Solution Architecture

Figure 4.6.1 shows the architecture of the implemented system. The browser clients communicate with the Django project over HTTP using JSON and a session cookie. The Django project bus_tracking handles URL routing, settings, session authentication and static files through WhiteNoise, and is served by gunicorn through the WSGI application. Inside it, the tracking app contains the API views and serialisers, an authorisation helper, the service modules, the models, and the management commands that import GTFS data. Data is stored in a PostgreSQL database whose address is given by DATABASE_URL. The GTFS text files are read only by the import commands. OSRM is an external service reached through HTTP. Password-reset e-mail is sent through Brevo using the Django built-in password-reset views.

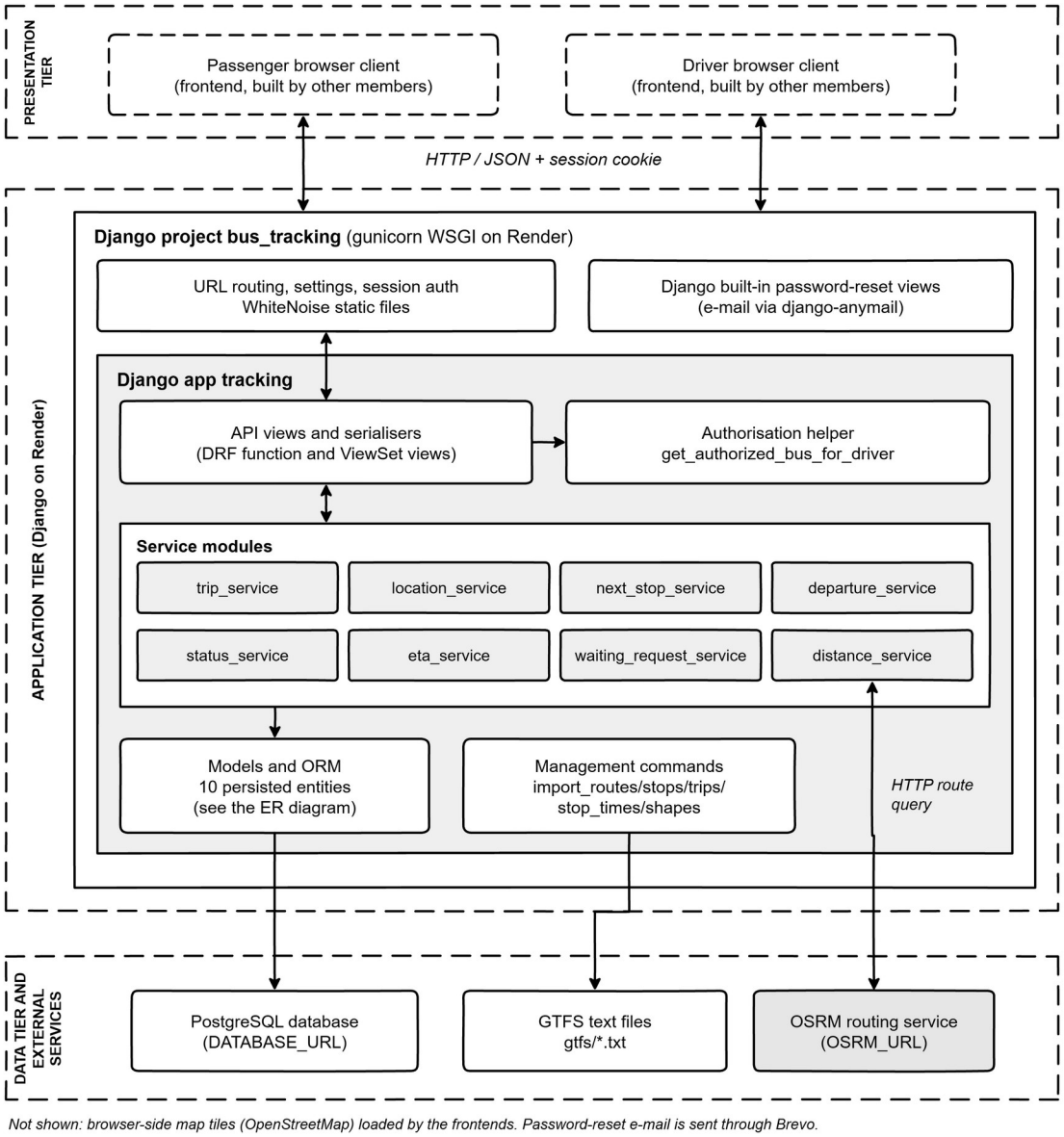


Figure 4.6.1: System architecture of the Bus Tracking System

The system is a client–server application with a three-tier (n-tier) structure. The presentation tier is the passenger interface and the driver dashboard, which run in the user's browser and are delivered as pages by the server. The application tier is the Django project, in which the API views call service modules that use the models and the ORM. The data tier is the PostgreSQL database, together with the GTFS files from which the data is imported. The system also calls an external road-routing service, OSRM. The passenger page polls the bus data over HTTP every 15 seconds. The driver dashboard does not poll on a timer: it posts the device's position each time the browser supplies a fix and then reloads the waiting overview. (The group architecture figure and the group presentation show 15-second polling for both interfaces; the code of the integration branch shows it only for the passenger page.) The

layering inside the application tier is not strict: some views, such as the read-only route, stop and bus endpoints, query the models directly (Section 4.9.1).

The map tiles are the one exception to the rule that the browser talks only to the Django project: Leaflet requests them directly from OpenStreetMap in the browser, and the Leaflet library itself is loaded from a content-delivery network. The service modules separate business rules from request handling: trip_service selects and starts trips, location_service stores GPS updates, next_stop_service and departure_service keep the stop state, status_service derives bus status, distance_service calls OSRM, eta_service estimates arrival, and waiting_request_service implements the waiting-request lifecycle and counts. A further module, route_service, defines a function that is not used by the current code. The environment variables read by the application are listed in Appendix F.

4.7 Database / Data Design

4.7.1 Domain Model

The domain model in Figure 4.7.1.1 is a conceptual model. Following Larman's notation it shows conceptual classes as boxes with names only, associations named in the form TypeName–VerbPhrase–TypeName, and multiplicities [4]. It includes need-to-know associations, that is, those whose memory the system must preserve, and omits associations that would only aid comprehension [4]. It deliberately contains no keys, field types, serialisers, services, APIs or framework tables, which belong to the ER diagram and the design model.

Conceptual classes and associations in Larman notation (no keys, fields or framework classes)

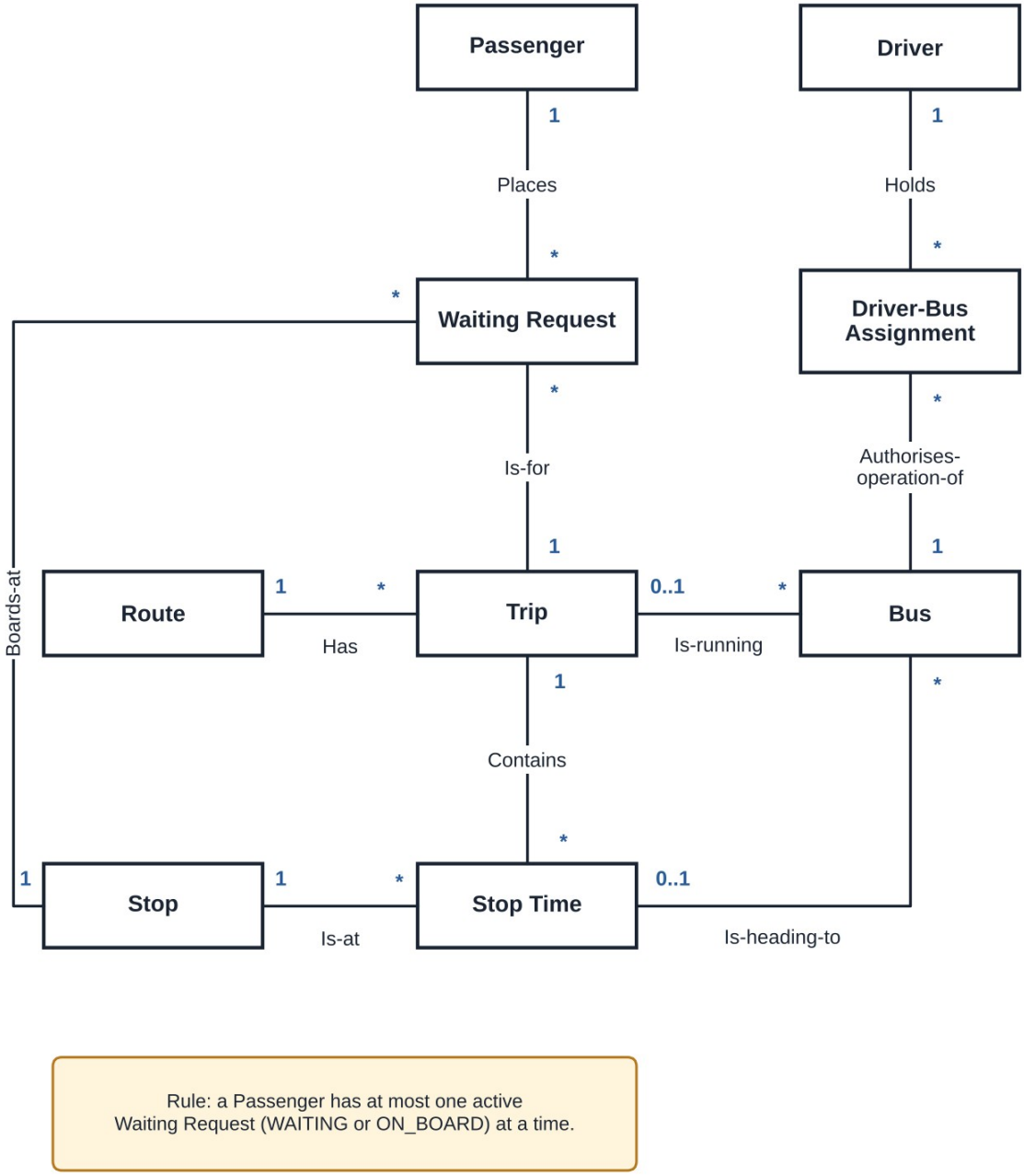


Figure 4.7.1.1: Domain model of the Bus Tracking System (conceptual)

A Passenger places Waiting Requests, each of which is for one Trip and boards at one Stop. A Route has Trips, and a trip contains an ordered sequence of Stop Times, each at one Stop. Multiplicities are the ones the database enforces, so a route or trip with no trips or stop times is not excluded by the schema. A Driver holds Driver–Bus Assignments, and each assignment authorises the operation of one Bus. A Bus may be running one Trip and heading to one Stop Time at a time. The rule that a passenger has at most one active waiting request at a time is a constraint on the model and is shown as a note.

4.7.2 Entity-Relationship Design

The ER diagram in Figure 4.7.2.1 shows the persisted design in crow's foot notation: each entity is a table with its primary key (PK) and foreign keys (FK), and the symbol at the end of each line shows whether one, zero or one, or zero or many instances of that entity take part in the relationship. It includes the User table of Django's authentication system, restricted to the fields used, and omits Django's internal tables. Driver and Passenger each extend User through a one-to-one relationship. Bus has two nullable foreign keys, current_trip and current_stop_time. WaitingRequest links to Passenger, Trip and Stop. The field Trip.shape_id is a plain identifier that matches Shape.shape_id; it is not a foreign key, and Shape is not used in route selection, trip progression or ETA.

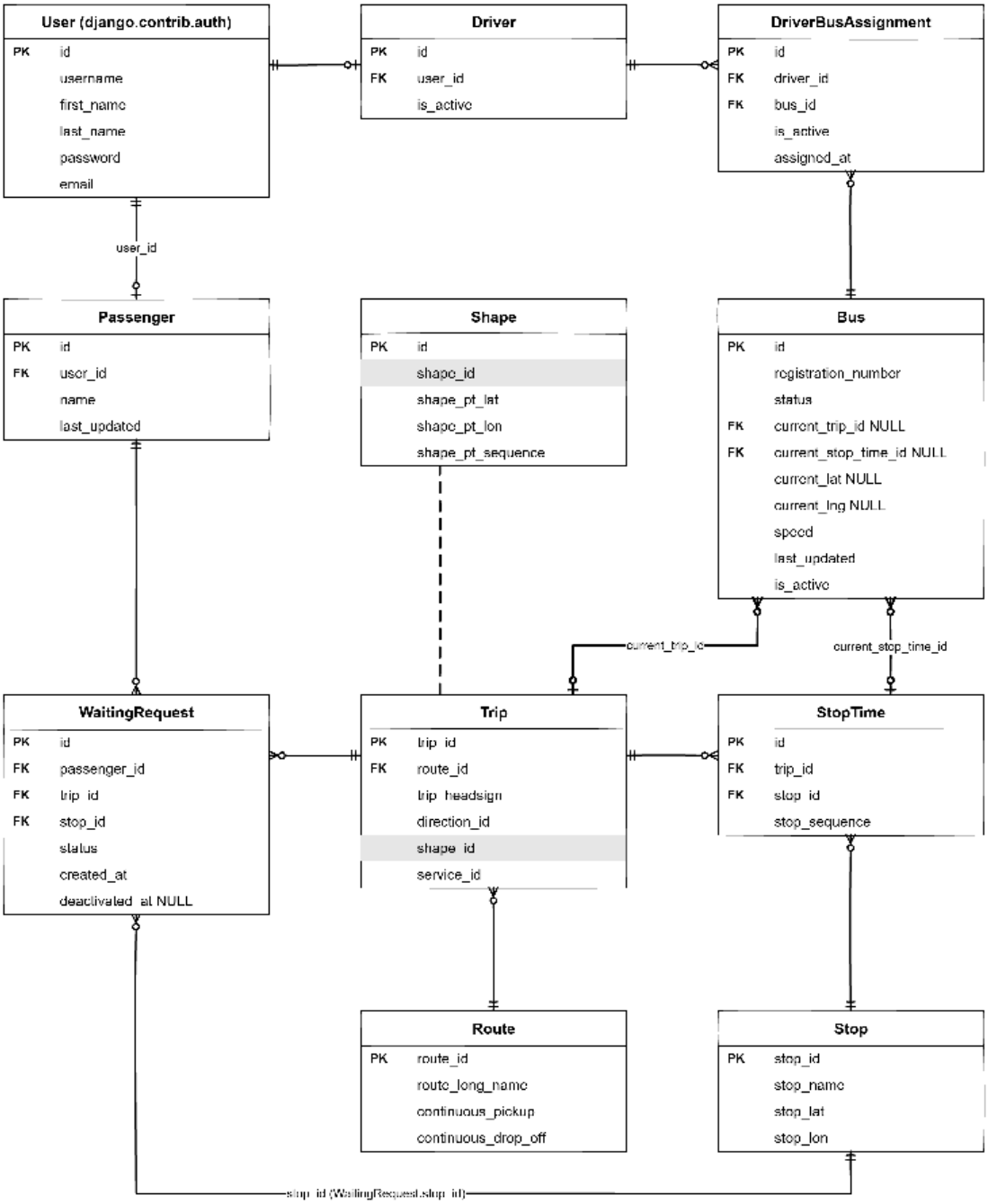


Figure 4.7.2.1: Entity-relationship diagram of the persisted data

The persisted data design has ten entities plus Django's built-in User model. The transport data (GTFS) group holds Route, Trip, StopTime, Stop and Shape; the data holds 68 routes, 793 stops, 9,219 stop times and 136 representative trips. The operations group holds Bus, Driver and DriverBusAssignment. The passenger group holds Passenger and WaitingRequest, whose status is one of WAITING, ON_BOARD, CANCELLED, COMPLETED or EXPIRED. Four key design decisions shape the data design. PostgreSQL is the database of the deployed system, managed through migrations. Only the current state

of each bus is stored, so there is no location history table. The 08:00:00 variant of each trip (136 of 408) is used as the representative trip, which makes route-and-direction selection unambiguous. A waiting request is trip-based: it names a route direction and a boarding stop. Table 4.7.2.1 describes the main entities and the rules the model enforces.

Table 4.7.2.1: Data dictionary of the principal entities

Entity	Principal fields	Rules
Route	route_id (key), route_long_name, continuous_pickup, continuous_drop_off	One route has many trips
Stop	stop_id (key), stop_name, stop_lat, stop_lon	Imported from stops.txt; a row without coordinates is skipped
Trip	trip_id (key), route, trip_headsign, direction_id, shape_id, service_id	Identifier ends in the start time; trips ending in -08:00:00 are the representative trips
StopTime	trip, stop, stop_sequence	Unique on (trip, stop_sequence)
Shape	shape_id, shape_pt_lat, shape_pt_lon, shape_pt_sequence	Unique on (shape_id, shape_pt_sequence); imported but unused in logic
Bus	registration_number (unique), status, current_trip, current_stop_time, current_lat, current_lng, speed, last_updated, is_active	Status is IDLE, IN_TRANSIT or AT_STOP; only the latest state is stored
Driver	user (one-to-one), is_active	Deleting the user deletes the driver
DriverBusAssignment	driver, bus, is_active, assigned_at	Unique on (driver, bus)
Passenger	user (one-to-one), name, last_updated	Deleting the user deletes the passenger
WaitingRequest	passenger, trip, stop, status, created_at, deactivated_at	Status is WAITING, ON_BOARD, CANCELLED, EXPIRED or COMPLETED

The application stores only the current state of each bus. It has no history table for locations. Past positions are therefore not available for analysis.

4.7.3 Data Presented by the Passenger Interface

The passenger interface reads and writes a small part of the persisted data. Table 4.7.3.1 lists the entities it touches and the fields that appear on the passenger's screen. The full entity-relationship design, prepared by another member, is in Appendix C.

Table 4.7.3.1: Data held about passengers and waiting requests

Entity	Fields the interface uses	Rules
Bus	registration_number, status, current_lat, current_lng, speed, is_active	Read-only for the passenger; status is IDLE, IN_TRANSIT or AT_STOP; only the latest state is stored
Route	route_long_name	Searched by name; one route has many trips

Entity	Fields the interface uses	Rules
Trip	trip_headsign, direction_id, route	The representative trip (identifier ending -08:00:00) identifies a route and direction
StopTime and Stop	stop_sequence, stop_name	Give the ordered stops of a trip that the passenger chooses among
Passenger	user (one-to-one), name	Created at registration; deleting the user deletes the passenger
WaitingRequest	passenger, trip, stop, status, created_at, deactivated_at	Status is WAITING, ON_BOARD, CANCELLED, EXPIRED or COMPLETED; the client cannot set its own status or timestamps

The application stores only the current state of each bus. It has no history table for locations, so the passenger map can show only where a bus is now, not where it has been.

4.7.4 Data Handled by the Driver Interface

The driver dashboard reads and writes a small part of the persisted data. Table 4.7.4.1 lists the entities it touches and the fields that appear on the driver's screen.

Table 4.7.4.1: Data handled by the driver interface

Entity	Fields the interface uses	Rules
Driver and DriverBusAssignment	is_active; the assigned bus	The dashboard lists only buses with an active assignment for the signed-in active driver
Bus	registration_number, status, current_trip, current_stop_time, current_lat, current_lng, speed, last_updated, is_active	Written by the driver's activation, reset and GPS updates; status is IDLE, IN_TRANSIT or AT_STOP and is derived by the server
Route, Trip, StopTime and Stop	route_long_name, trip_headsign, stop_sequence, stop_name	Read for the route and destination lists and the stops of the trip; the representative trip (identifier ending -08:00:00) identifies a route and direction
WaitingRequest	trip, stop, status	Read as counts of WAITING requests for each stop of the current trip; the driver cannot change a request

4.8 Interface / Interaction Design

The two browser interfaces are how drivers and passengers use the system. The driver dashboard was developed by the author and is described first and in more detail. The passenger interface was developed mainly by Ngosa Muleba and is described from the repository code, the supplied screenshots and the passenger-interface report. Both interfaces are Django templates with inline styles and scripts, use no front-end framework, call the JSON API with the browser's Fetch API and send the CSRF token with each state-changing request. A landing page (Figure 6.3.2) lets the user choose the passenger or the driver role.

4.8.1 Driver Interface (developed by the author)

The driver dashboard implements the sequence in Figure 4.5.3.1. Its design follows the interface requirements of Section 4.3 and the driver side of the adaptation in Table 3.3.3.1. Its screens and interactions are as follows.

- **Sign-in and session.** The driver signs in with a username and password through the driver login operation. The server refuses accounts that have no driver profile or whose driver record is inactive, and the dashboard shows a distinct message for wrong credentials (HTTP 401) and for an account that is not an active driver (HTTP 403). When the page is opened again, it asks the server for the current driver and, if a session exists, shows the dashboard without a new sign-in. A Log out button ends the session and stops tracking.
- **Bus selection.** After sign-in, the dashboard lists the buses actively assigned to the account (listAssignedBuses) and selects the first by default. For the selected bus it loads a driver-session record, which shows the route, destination, current target stop and last update of a trip already in progress, so a driver who reloads the page can see the state of a resumable trip. The status is IDLE before activation.
- **Bus operations.** The driver can activate the bus, deactivate it or reset its operational state. A message confirms the result of each operation.
- **Route and destination selection and trip start.** The driver chooses a route and a destination from the lists returned by the route endpoints and starts the trip. The server selects the representative trip, sets the bus's current trip and first stop, and the dashboard lists the number of passengers waiting at each stop of the trip.
- **GPS tracking.** When the driver presses Start GPS tracking, the browser asks for permission to read the device's location. The dashboard then calls navigator.geolocation.watchPosition with high accuracy enabled, a maximum cached-position age of 1 second and a 15-second timeout, and sends the latitude, longitude and speed of each fix to the location endpoint of the bus. The speed supplied by the browser in metres per second is multiplied by 3.6, so the speed sent to the server is in kilometres per hour. Only one update request is in flight at a time, and a fix that arrives while a request is pending is skipped. A GPS error is shown to the driver as a message. The same control stops tracking.
- **Feedback.** The dashboard shows a message that the GPS update was accepted by the backend, the reported location, the speed and the vehicle status (IDLE, IN_TRANSIT or AT_STOP), and marks the latest location on a Leaflet map. The latest location is also visible on the passenger map.

The design choices that follow from this are the use of one authoritative source of position (the driver's own device, tied to a bus the driver is assigned to), the display of the server's accepted state rather than a locally assumed one, and the use of plain HTTP so that the

dashboard needs no installation. The dashboard is a single Django template, driver_interface.html (489 lines, with its styles and script inline), that loads Leaflet 1.9.4 from the unpkg network and uses no front-end framework. Screenshots of the driver dashboard in use are in Figures 6.3.3 to 6.3.7 of Chapter 6.

4.8.1.1 Parts of the Driver Dashboard

Table 4.8.1.1 lists the parts of the driver dashboard and the requirements (Table 4.3.4) that each one realises.

Table 4.8.1.1: Parts of the driver dashboard and what each one does

Part	What the driver does or sees	Requirements
Header	The title Driver Dashboard	None
Driver login	Username and password fields and the Sign in button, with a message for a refused or failed sign-in	DR1, DR2
Assigned bus	A list of the assigned buses; the Activate, Deactivate, Reset operational state and Log out buttons; a message for each operation; the status of the selected bus	DR2, DR3, DR4
Start a trip	Route and Destination lists and the Start trip button	DR5
Waiting requests by stop	The route, destination and the number of passengers waiting at each stop of the current trip	DR8
GPS tracking	The Start GPS tracking and Stop GPS tracking button, a message that the update was accepted by the backend, and tiles for the location, the speed (km/h) and the vehicle status	DR6, DR7, DR9
Live bus location	A Leaflet map on OpenStreetMap tiles with a marker for the position that the server accepted, and a line stating the latest location returned by the backend	DR7

4.8.2 Passenger Interface (developed mainly by Ngosa Muleba)

The passenger interface is described in the passenger-interface report in more detail. It is presented here in full, as one of the two interfaces of the system, with the figure numbers of this report.

4.8.2.1 Overall Layout and Navigation

A landing page (Figure 6.3.2) lets the user choose the passenger or the driver role. The passenger page is organised around one idea: the map is always the base screen, and everything else opens over it. A blue header bar carries the title *Lusaka Bus Tracker* and four controls: **Active Buses**, **Plan Trip**, **My Requests** and a refresh control. A second, round refresh control floats at the lower right of the map. Table 4.8.2.1 lists the parts of the page.

Table 4.8.2.1: Parts of the passenger page and what each one does

Part	What the passenger does or sees	Requirements
Header bar	Title and the controls Active Buses, Plan Trip, My Requests and refresh	PR11
Live map	Leaflet map on OpenStreetMap tiles with zoom controls; a marker for each active bus; a popup with registration number, status and speed	PR1, PR2
Plan Trip panel	An overlay panel with the account area (register, login, forgot password, logout), the route search, the destination choice, the list of stops, and the Create Waiting Request and Cancel controls	PR3 to PR7
My Requests sheet	A sheet that rises from the bottom of the screen with the heading *My Waiting Requests* and one card per request	PR8 to PR10
Messages	Coloured banners at the top of the screen that confirm an action, for example that a boarding stop was selected or that a waiting request was created	PR6, PR7

4.8.2.2 The Live Map

Figure 6.3.8 shows the passenger map. An active bus appears as a green bus marker. The marker's popup gives the registration number, the status and the speed, which turns a bare position into information about the bus. The data is requested from the API when the page loads and again every 15 seconds. The refresh controls allow the passenger to request it at once. The map shows the buses only; routes and stops are presented as lists in the Plan Trip panel, not drawn on the map.

4.8.2.3 Trip Planning and Boarding Stop Selection

The Plan Trip panel guides the passenger through four steps, which correspond to the Journey Planning feature and to the ride-hailing concepts adapted in Table 3.3.3.1. First the passenger registers or signs in. Second, the passenger types part of a route name in the *Search route* field, and the matching routes appear in a selection list. Third, the panel displays the chosen route under *Trip and stop details* with the prompt *Select a destination*, and the passenger touches one of the destination buttons. Fourth, the panel lists the ordered stops of the trip, and the passenger touches the stop where they will board; a banner confirms the choice (*Selected boarding stop: …*). The *Create Waiting Request* button then submits the request. Figure 4.8.1 shows the route search, the boarding-stop confirmation and the submission control.

Two decisions shaped this design. First, a passenger chooses a *destination* and not a *direction*, because the destination is the form in which a passenger thinks of a journey; the server turns the route and destination into one representative trip (Section 3.3.4). Second, the stops are offered as a list in travel order, so that the passenger can see which stops lie ahead of the one they choose.

(a) Route search and selection



(b) Boarding-stop selection confirmed



(c) Waiting-request submission control

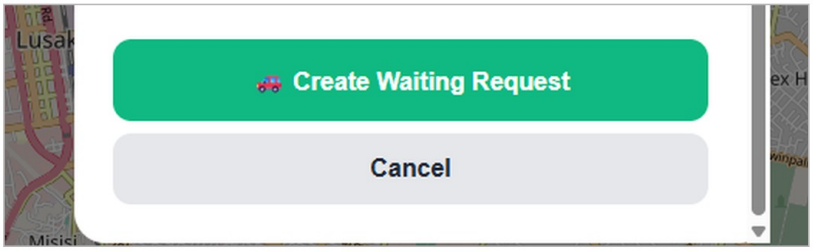


Figure 4.8.1: Passenger interface screens for route search, boarding-stop selection and waiting-request creation (passenger interface developed mainly by Ngosa Muleba)

4.8.2.4 Waiting Requests

The *My Waiting Requests* sheet lists the passenger's requests, newest at the bottom in the checks reported here. Each card gives the route name in blue, the status at the right in a colour that distinguishes it, and a second line with the boarding stop and the time of the request. The card of the request in progress is outlined. For a request in the WAITING state the card adds the line *Estimated arrival at …: about N minutes* and two controls, a green **Mark as Boarded** button and a grey **Cancel Trip** button (Figure 4.8.2). After boarding, the request is in the ON_BOARD state and the card offers the control that completes the trip (Figure 4.8.3). Requests that were cancelled stay in the list with their status and no controls.

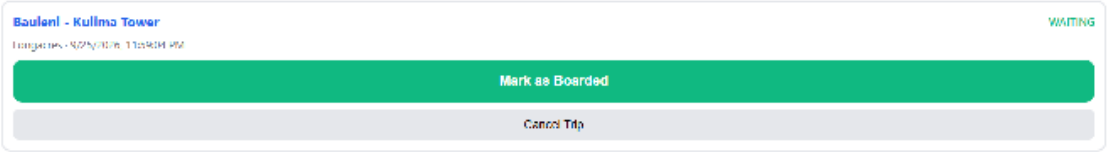


Figure 4.8.2: A waiting request in the WAITING state (passenger interface developed mainly by Ngosa Muleba)

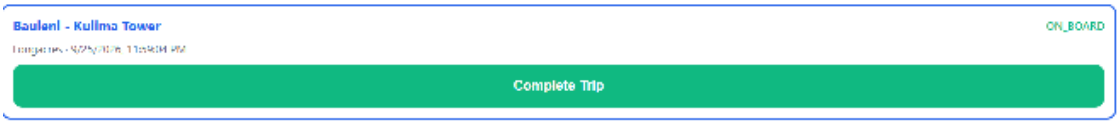


Figure 4.8.3: A waiting request in the ON_BOARD state (passenger interface developed mainly by Ngosa Muleba)

4.9 Algorithms / Models / Technical Design

4.9.1 Design Model

The design model describes how software objects realise the system operations. It consists of the design class diagram, interaction diagrams and package structure [4]. One caution is necessary: the implementation is written as Django views, function-based service modules and Django models, not as a hierarchy of classes with behaviour. The design class diagram therefore shows modules with the «module» stereotype, listing their functions as operations, and shows the Django models as the persistent classes. This keeps the diagram truthful to the code while still using Larman's notation for class diagrams, responsibilities and visibility. The diagram shows every module of the services package, the controller and authorisation functions of views.py that the two interaction diagrams below use, and the persistent classes that carry behaviour or are used by the services. The remaining models (Route, Driver, DriverBusAssignment, Passenger and Shape) have no behaviour of their own and appear in the ER diagram. The function get_active_buses in route_service is defined but not called by other code.

The two interaction diagrams realise the two operations with the most logic. Figure 4.9.1.2 shows updateBusLocation, the operation that the driver dashboard calls repeatedly. The view first calls the authorisation helper. It then calls save_bus_location, which loads the bus, sets the position and speed, and asks the stop services to update the next stop. The departure check calls get_distance_to_current_stop, which calls OSRM. get_bus_status then calls the distance service a second time, so one GPS update can involve two OSRM requests. The bus is finally saved and returned. Figure 4.9.1.3 shows getWaitingRequestEta. The view retrieves only the signed-in passenger's request, then calculate_waiting_request_eta queries the stop sequences and eligible buses. For each eligible bus it calls the distance service, which calls OSRM. The shortest valid duration, converted to minutes, is returned, or null when none is valid.

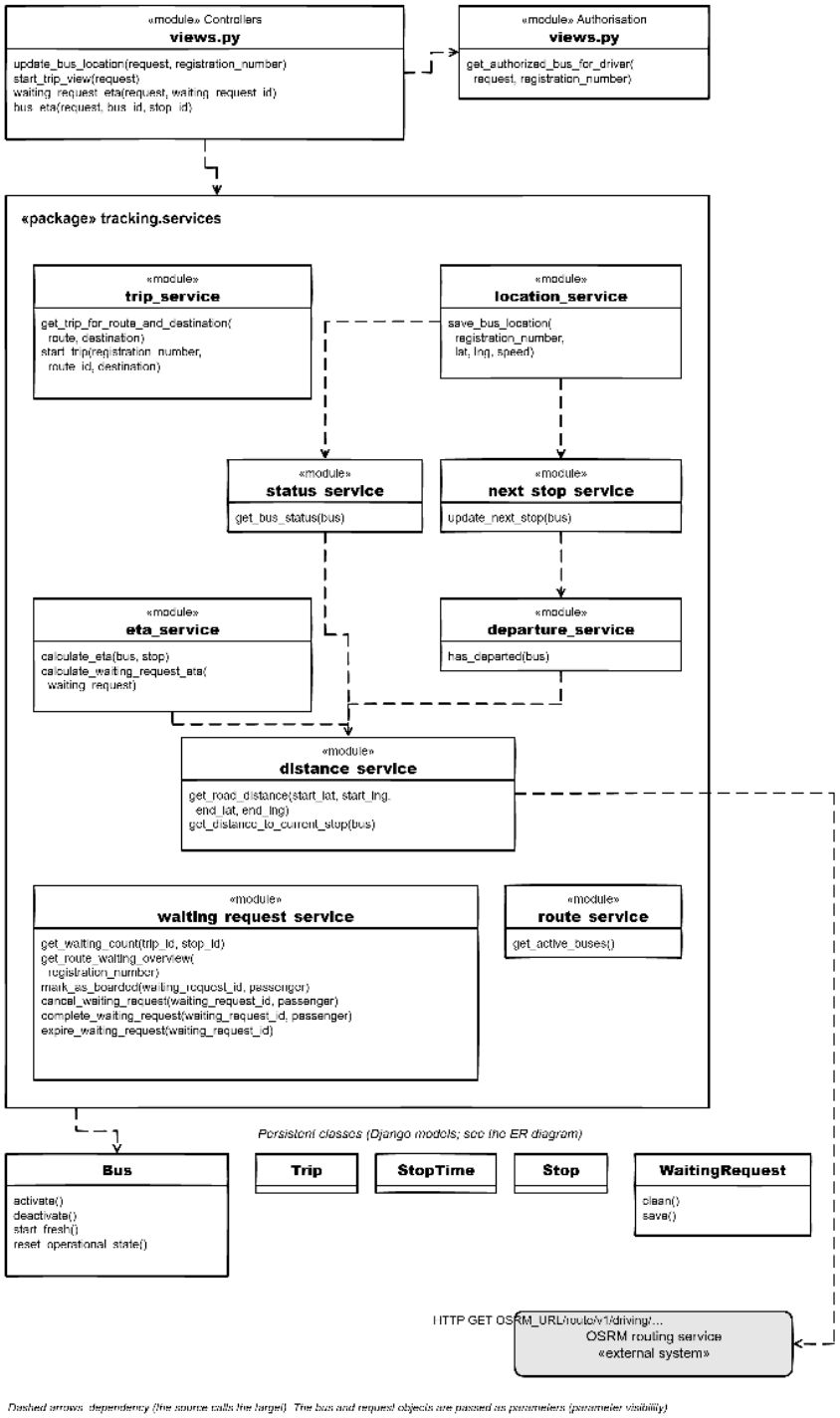


Figure 4.9.1.1: Design class diagram of the system modules and persistent classes

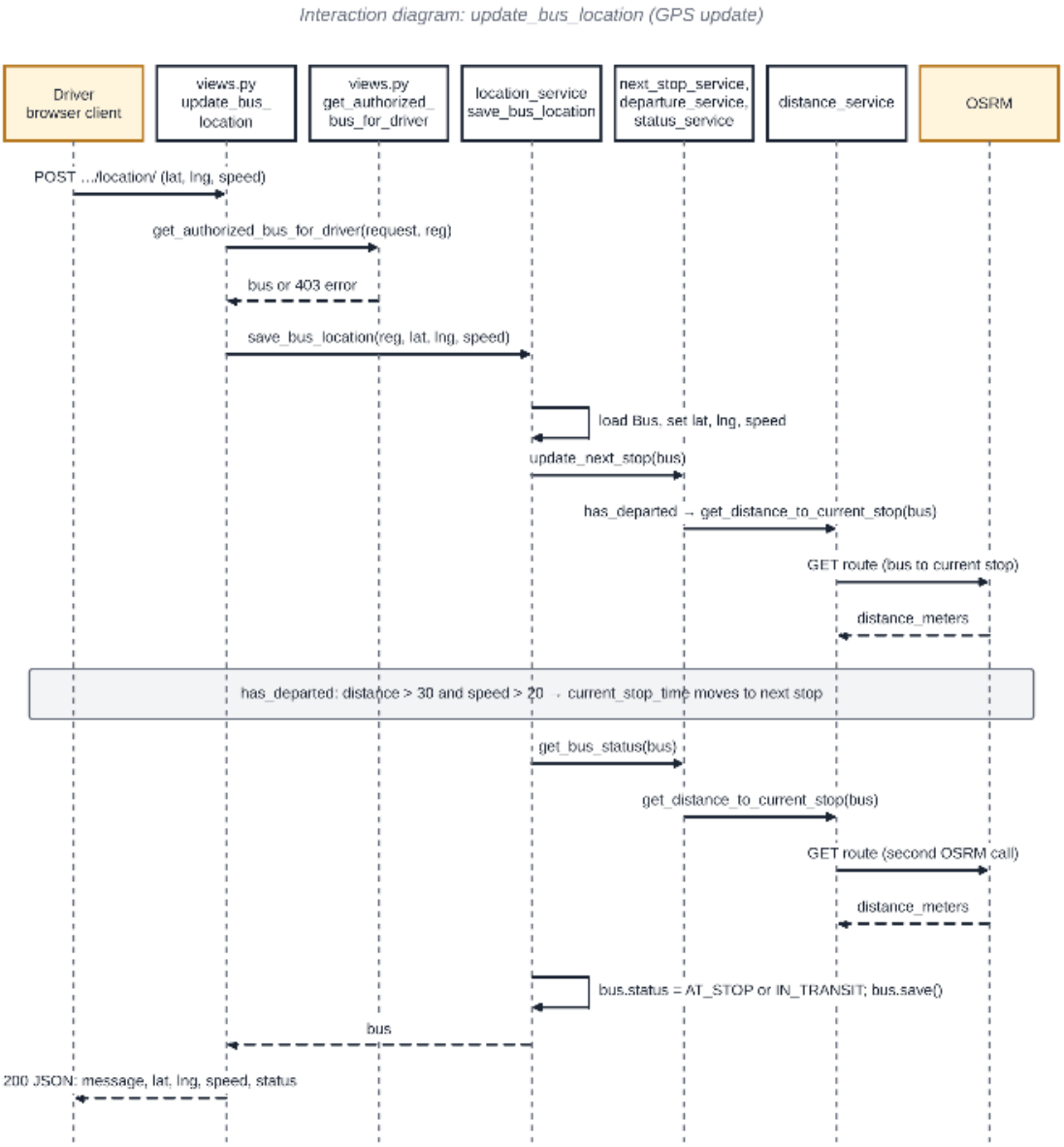


Figure 4.9.1.2: Interaction diagram for the system operation updateBusLocation

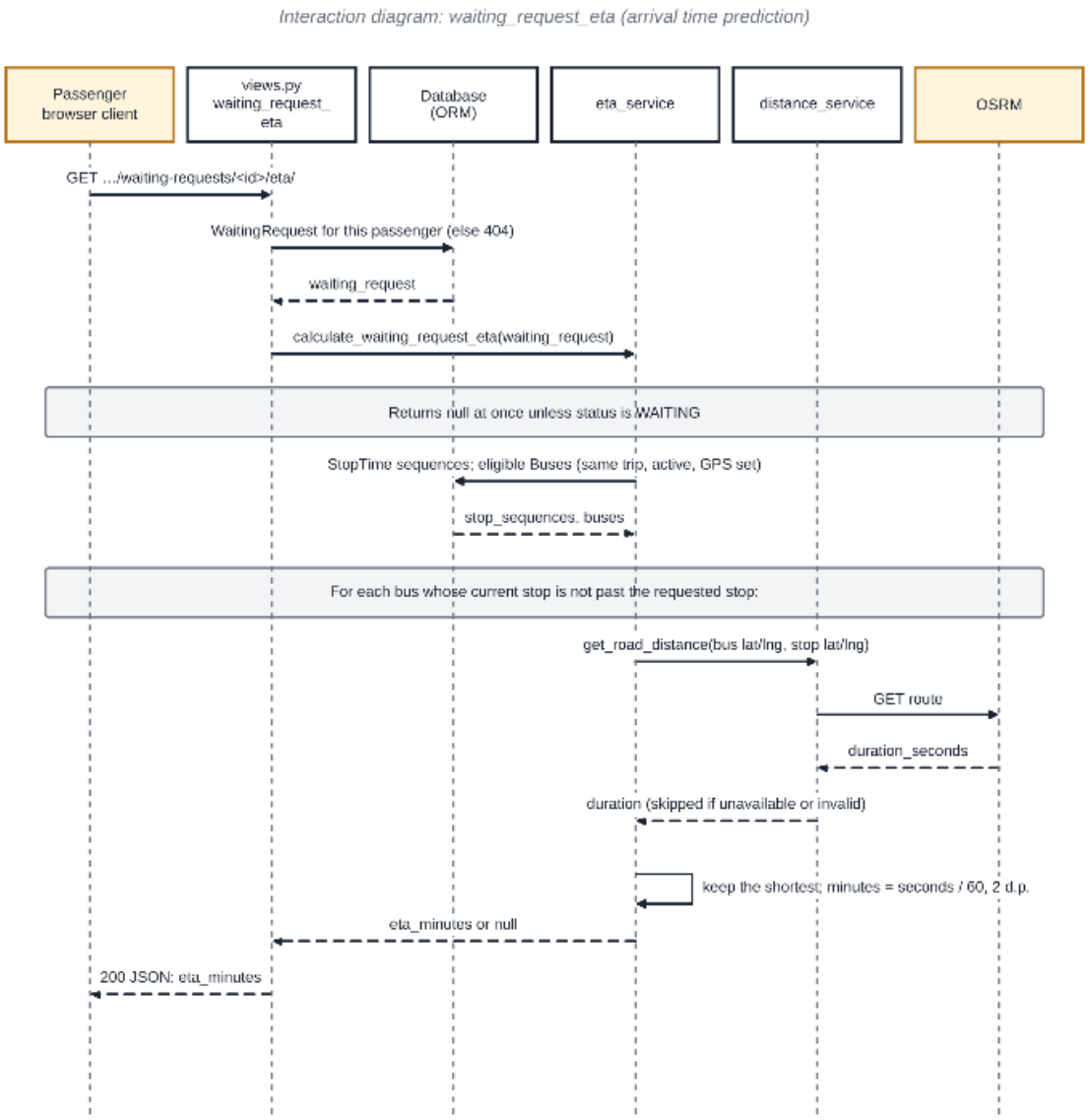


Figure 4.9.1.3: Interaction diagram for the system operation getWaitingRequestEta

4.9.2 Visibility

Visibility in the design is mostly parameter visibility: a service obtains the bus or request as a parameter, as in update_next_stop(bus) and calculate_waiting_request_eta(waiting_request). Module-level (global) visibility applies to the OSRM base URL, which is read once from the environment in the distance module. Local visibility arises where a service loads records itself, for example save_bus_location loads the bus by registration number.

4.9.3 Trip Selection and Bus State Logic

A driver chooses a route and a destination. get_trip_for_route_and_destination examines the representative trips (identifier ending in -08:00:00) of the route and returns the one whose last stop has the chosen destination name; if none is found it returns nothing and start_trip

raises an error. start_trip sets the bus's current trip and first stop time, sets the status to IDLE and the bus to active, inside one database transaction. On each GPS update the position and speed are stored without validation, the next stop is updated and the status is derived. Table 4.9.3.1 states the rules. Distances come from OSRM and are in metres [8]. The server does not state the unit of the speed it receives, but the driver dashboard converts the browser's speed from metres per second to kilometres per hour before sending it, so the 20 in the rules is a speed in km/h, as the group presentation states.

Table 4.9.3.1: Rules for next-stop update and bus status

Rule	Condition
Next stop advances (has_departed)	The bus has a current stop time, the road distance to the current stop is greater than 30 and the speed is greater than 20. If the routing request fails, the bus is treated as not departed
Status IDLE	The bus has no current stop time, or the distance to the current stop is unavailable
Status AT_STOP	The road distance to the current stop is at most 30 and the speed is at most 20
Status IN_TRANSIT	Any other case, including when the routing request fails

The thresholds are fixed values and have not been calibrated against field data.

4.9.4 Waiting-Request Lifecycle

A waiting request is created in the WAITING state. A passenger can mark it as boarded (ON_BOARD) or cancel it (CANCELLED). An on-board request can be completed (COMPLETED). A WAITING request can also be set to EXPIRED by an expiry endpoint. Cancelled, completed and expired requests record the time of deactivation. A passenger cannot create a request while another is WAITING or ON_BOARD.

4.9.5 ETA Algorithm

The ETA for a waiting request is calculated by the following steps. If any condition fails, the estimate is not available and null is returned instead of a guessed value.

1. Return null unless the request status is WAITING.
2. Find the stop sequences at which the requested stop appears in the requested trip; return null if there are none.
3. Select candidate buses: active, with status IN_TRANSIT or AT_STOP, on the same trip, with coordinates present and a current stop time on that trip.
4. Discard a candidate whose coordinates are not finite or are outside valid latitude and longitude ranges, or whose current stop sequence is greater than every sequence of the requested stop (the bus has passed it).

5. For each remaining candidate, request the road duration from the bus position to the stop from OSRM. Skip the candidate if the request fails, the response is malformed, or the duration is not finite or is negative.
6. Return the shortest remaining duration, converted to minutes and rounded to two decimal places, or null if no candidate remains.

The duration is that of the OSRM route profile and is returned in seconds [8]. It is not adjusted for live traffic or dwell time at stops. Section 6.6 discusses the consequences.

4.9.6 Passenger Interface: Map Refresh

The passenger interface contains little computation of its own: the server holds the rules. Its logic is the sequencing of calls and the choice of what to show.

- When the page loads, request the bus records from the API and draw a marker for each active bus that has a position.
- Attach a popup to each marker giving the registration number, status and speed.
- Repeat the request every 15 seconds and update the markers with the new data.
- Repeat the request at once when the passenger presses a refresh control.

The interval is a compromise between currency and load. It is not tuned, and the position the passenger sees can be older than 15 seconds because the driver's device also sends updates at its own rate.

4.9.7 Passenger Interface: Journey-Planning Sequence

- When the passenger types a route name, request the routes that match and offer them in the selection list.
- When the passenger selects a route, request its destinations and present them as buttons.
- When the passenger selects a destination, request the ordered stops of the trip for that route and destination, and present them as a list.
- When the passenger selects a stop, confirm the choice with a banner and keep the choice for the request.
- When the passenger presses Create Waiting Request, send the chosen trip and stop. On success, confirm with a banner and add the request to the list.

4.9.8 Passenger Interface: Waiting-Request Lifecycle and the Controls Shown

A waiting request is created in the WAITING state. A passenger can mark it as boarded (ON_BOARD) or cancel it (CANCELLED). An on-board request can be completed (COMPLETED). A WAITING request can also be set to EXPIRED, an operation limited to

administrators. A passenger cannot create a request while another is WAITING or ON_BOARD. The server enforces these rules, so a client cannot set a status directly. The interface offers only the controls that suit the current status, as Table 4.9.8.1 shows.

Table 4.9.8.1: Controls offered for each waiting-request status

Status	Meaning	Controls and information on the card
WAITING	The passenger is waiting at the boarding stop	Estimated arrival (when available); Mark as Boarded; Cancel Trip
ON_BOARD	The passenger has boarded	Control to complete the trip
COMPLETED	The journey is finished	None; kept in the list as a record
CANCELLED	The passenger cancelled	None; kept in the list as a record (Figure 6.3.12)
EXPIRED	An administrator ended a waiting request	None; kept in the list as a record

4.9.9 Passenger Interface: The Arrival Estimate as Presented

The estimate is calculated on the server. It is requested for a request in the WAITING state. The server selects the active buses on the same trip with valid coordinates that have not passed the stop, asks OSRM for the road duration of each to the stop, and returns the shortest valid duration in minutes, or null if no bus qualifies or routing fails [8]. The passenger interface shows the figure as *about N minutes* when the server returns one and has nothing to show when it returns null; it never substitutes a value of its own. This follows the principle of Section 2.2.2 that a wrong arrival time is worse than none. The duration is not adjusted for live traffic or for dwell time at stops.

4.9.10 Driver Interface: GPS Reporting Sequence

The driver dashboard contains little computation of its own either; the server decides who may report, stores the position and derives the status. The logic of the dashboard is the sequence in which it calls the server and what it shows. The reporting sequence follows.

- When the driver presses Start GPS tracking, check that a bus is selected and that the browser supports geolocation; otherwise show a message.
- Start watching the device position with high accuracy, a maximum cached-position age of 1 second and a 15-second timeout.
- For each position fix, skip it if an update request is still outstanding; otherwise convert the speed from metres per second to kilometres per hour (a missing speed is taken as zero) and send the latitude, longitude and speed to the location endpoint of the selected bus.

- When the server accepts the update, show the message that the update was accepted by the backend, the reported location, the speed and the status that the server derived, move the marker on the map, and reload the waiting counts by stop.
- When the server refuses or fails, or the device cannot supply a position, show the error to the driver and keep the dashboard usable.
- When the driver presses Stop GPS tracking, stop watching the device position.

The rate of updates is that of the browser's position fixes, so the position held by the server can be older than the latest movement of the bus when the device or the network is slow. This is one reason why the passenger map is described as showing the last position reported by the driver's device.

4.10 Chapter Summary

The analysis defines three features and nineteen functional requirements, supported by a use-case diagram and two system sequence diagrams that treat the system as a black box. The design is shown through the architecture, a conceptual domain model, an ER diagram and a design model made of modules, interaction diagrams and the rules for trip selection and bus state. The interface design is presented for both the driver dashboard and the passenger interface. The key technical rules are trip selection from representative trips, bus status and next-stop thresholds, the waiting-request lifecycle and the conditional ETA algorithm. Chapter 5 documents their implementation.

The passenger interface is the presentation tier of a three-tier system, organised around a live map with overlay panels for trip planning and for waiting requests, and its logic is a short sequence of calls to the API, a fixed refresh interval and a rule that the controls on a request card follow its status. The driver dashboard is the other presentation-tier page: its logic is the sequence of calls for sign-in, bus operations and trip start, and a reporting loop that sends the device position to the server and shows what the server accepted.

CHAPTER 5: IMPLEMENTATION / DEVELOPMENT

5.1 Introduction

This chapter describes how the Bus Tracking System was implemented. It gives an overview of the whole system and of the order in which it was developed, then documents the components, the key implementation details, the integration, the challenges met and the author's individual contribution. The implemented system has three parts. The first is the Django server, which exposes a JSON API, stores the data and calls the routing service. The second is the passenger interface, a browser page with a map of active buses and a trip-planning panel. The third is the driver dashboard, a browser page for bus operations and GPS tracking. The author developed the driver dashboard and driver-side functionality; the other parts are described as the work of the members who developed them, and shared work is identified as shared.

5.1.1 Development Sequence

The system was developed in the order shown in Table 5.1.1. The sequence matters for attribution: the backend was not completed before the interfaces were started, and it was not finished by one member alone. A first backend skeleton gave the interface developers an API to work against; the backend was then extended as the needs of the driver and passenger interfaces became clear; and the branches were integrated and deployed at the end.

Table 5.1.1: Order of development and responsibility

Step	Activity	Responsibility	Evidence status
1	Shared problem analysis, requirements and interface analysis	Group	Described in Chapter 3
2	Initial backend skeleton: Django project, initial models and migrations, first REST endpoints	Kenneth Kaunda Musonda	Group report
3	Interface development on separate branches: driver interface (Driver-Interface branch) and passenger interface	Author (driver interface); Ngosa Muleba (passenger interface)	Author's branch Driver-Interface, commit "Complete driver interface and trip lifecycle" (c816533); passenger branch (name not recorded)
4	Backend updates alongside the interfaces: driver authorisation and assignment, trip start, GPS handling, status and next-stop logic, waiting requests, ETA	Collaborative; some changes on the Driver-Interface branch, others on the backend line	Individual commits not itemised in the available records

Step	Activity	Responsibility	Evidence status
5	Integration of the branches into the integration branch, security hardening, deployment configuration and deployment on Render	Kenneth Kaunda Musonda	Group report; merge commits not itemised in the available records
6	Testing, manual checks of the deployed flows, documentation and presentation	Collaborative	Chapter 6

5.2 Development Environment

The group developed on Windows using Visual Studio Code, with Python for the server and HTML, CSS and JavaScript for the browser interfaces. The server uses Django 6.0.6 and Django REST Framework 3.17.1, the maps use Leaflet 1.9.4 with OpenStreetMap tiles, and routing uses OSRM through the OSRM_URL setting. The database of the deployed system is PostgreSQL. Source control used Git and GitHub, with each interface developed on its own branch and the integration branch as the final state. Deployment used Render for hosting and Brevo for password-reset e-mail, with secrets held in environment variables. The project is started locally with python manage.py migrate and python manage.py runserver. The verified environment variables are listed in Table F.2 in Appendix F.

The repository contains the Django project bus_tracking (settings, URL configuration and WSGI entry point), the application tracking, and the gtfs directory with the GTFS text files. Within tracking are the models, views, serialisers, URL configuration, administration registrations, the services package, the management/commands package, the migrations and the tests. The browser interfaces are Django templates.

The verified environment variables are listed in Table F.2 in Appendix F; the settings read each of them with os.environ, so no secret is stored in the source.

The passenger interface. The passenger interface was developed on Windows in Visual Studio Code, using HTML, CSS and JavaScript, and tested in a web browser against the Django development server. Git and GitHub were used for source control, and the integration branch is the final state of the system. The deployed system, on which the passenger flows were checked, runs on Render with a PostgreSQL database. The passenger page is a Django template served by the project; it uses Leaflet 1.9.4 for the map, loaded from the unpkg content-delivery network, and OpenStreetMap tiles for the map background.

5.3 System Components / Modules

Table 5.3.1 lists the components and states who developed each. Where the available evidence does not establish individual authorship of a server-side file, the entry says so and the work is treated as collaborative.

Table 5.3.1: System components, responsibilities and developers

Component	Responsibility	Developed by
Driver dashboard (template, scripts and styles)	Driver sign-in, bus selection, activation, deactivation and reset, route and destination choice, trip start, waiting counts by stop, device GPS tracking, status display and driver map	Author
Passenger interface (template, scripts and styles)	Leaflet map of active buses refreshed every 15 seconds, route search, destination and boarding-stop selection, waiting-request creation and the requests list	Ngosa Muleba
Landing page	Choice of the passenger or driver role	Group (shared page)
models.py and migrations (0001–0009)	Route, Stop, Trip, StopTime, Shape, Bus, Driver, DriverBusAssignment, Passenger and WaitingRequest; Bus methods activate, deactivate, start_fresh and reset_operational_state	Initial skeleton: Kenneth Kaunda Musonda. Later migrations: not itemised in the available records
management/commands (5 commands)	import_routes, import_stops, import_trips, import_stop_times and import_shapes, which read the GTFS files	Credited to Kenneth Kaunda Musonda in the group report
views.py and urls.py	Passenger and driver authentication, route and trip endpoints, bus operations, GPS update, waiting requests, ETA and waiting counts; the authorisation helper	Kenneth Kaunda Musonda (initial); collaborative updates
serializers.py	Serialisation and validation, including the trip and stop checks of a waiting request and the start-trip payload	Collaborative
services/trip_service.py	get_trip_for_route_and_destination and start_trip	Trip-lifecycle changes appear on the Driver-Interface branch (individual authorship of the server-side functions is not established, so they are treated as collaborative)
services/location_service.py, next_stop_service.py, departure_service.py, status_service.py	Store GPS updates, update the next stop, detect departure and derive bus status	Stop-progression changes appear on the Driver-Interface branch (individual authorship of the server-side functions is not established, so they are treated as collaborative)
services/distance_service.py, eta_service.py	OSRM call; ETA calculation	Credited to Kenneth Kaunda Musonda in the group report

Component	Responsibility	Developed by
services/waiting_request_service.py	Waiting counts and overview; board, cancel, complete and expire operations	Not itemised; treated as collaborative
admin.py	Registers the main models in the administration site	Not itemised; treated as collaborative
tests.py (35 tests in the integrated system)	Automated tests for ETA, waiting overview, access control, registration, throttling and debug default	Kenneth Kaunda Musonda (credited in the group report)
Tests on the Driver-Interface branch	Test run reporting 94 passing tests	Author's branch; authorship of individual tests not established
bus_tracking/settings.py	Environment-based settings for database, secrets, hosts, CSRF origins, static files, secure cookies and e-mail	Kenneth Kaunda Musonda
Deployment configuration (Render build and start commands, WhiteNoise, gunicorn, Brevo)	Hosting and e-mail	Kenneth Kaunda Musonda

The application stores only the current state of each bus. It does not have a historical location table. The route_service module defines a function get_active_buses that no other code calls.

5.4 Key Implementation Details

Sections 5.4.1 to 5.4.9 describe the server as implemented in the integration branch. The initial backend foundation was built by Kenneth Kaunda Musonda, and the backend was then extended while the two interfaces were developed (Section 5.1.1); changes that cannot be tied to one member are treated as collaborative (Section 5.7.1). Sections 5.4.10 and 5.4.11 describe the driver interface and the passenger interface, which are the parts that the server serves.

5.4.1 Data Model and Migrations

The models implement the design in Figure 4.7.2.1. Driver and Passenger each have a one-to-one relationship to Django's User model, with the related names driver_profile and passenger_profile, which the views use to decide whether an account is a driver or a passenger. Bus holds the current trip and current stop time as nullable foreign keys that are set to null if the referenced record is deleted. The Bus.deactivate, Bus.activate and Bus.start_fresh methods run inside database transactions. start_fresh and deactivate call reset_operational_state, which clears the trip, stop, coordinates and speed and sets the status to IDLE. The nine migrations record how the schema evolved (Section 3.2.3).

5.4.2 GTFS Import Commands

Each import command opens its file under the gtfs directory, reads it with Python's csv module and uses update_or_create, so running a command again updates existing rows instead of duplicating them. Trips are imported only for routes that exist, and stop times only for trips and stops that exist; other rows are counted as skipped. The stop import skips a row with no coordinates, which is the one stop in the supplied feed that has none. Each command prints the numbers imported, updated and skipped. The commands must therefore be run in dependency order: routes and stops, then trips, then stop times; shapes are independent.

5.4.3 Driver Authentication and Authorisation

Driver sign-in authenticates the credentials with Django, rejects accounts that have no driver profile or whose driver is inactive, and then starts a session. Every protected bus operation passes through the helper get_authorized_bus_for_driver, abridged below. It returns the bus, or an error response with HTTP status 403.

```
def get_authorized_bus_for_driver(request, registration_number):
    bus = get_object_or_404(Bus, registration_number=registration_number)
    if not hasattr(request.user, "driver_profile"):   # 403: not a driver account
        ...
    driver = request.user.driver_profile
    if not driver.is_active:                          # 403: inactive driver
        ...
    assignment_exists = DriverBusAssignment.objects.filter(
        driver=driver, bus=bus, is_active=True).exists()
    if not assignment_exists:                         # 403: not assigned to this bus
        ...
    return bus, None
```

In the final stage of development the access rules of the API were tightened, and automated tests were added for each rule. The settings define IsAuthenticated as the default permission class for Django REST Framework, so an endpoint is protected unless it explicitly allows anonymous access [14]. The routes, stops and buses viewsets became read-only and public, because passengers need them to plan a journey, and the passenger viewset was removed from the router. The waiting-request viewset allows only listing, retrieving and creating, requires a passenger account, and treats status, passenger and timestamps as read-only, so a client cannot set its own lifecycle state. The expire endpoint is restricted to administrators, and the waiting-overview endpoint uses the same assignment helper as the driver operations. Registration now applies Django's configured password validators. The passenger and driver sign-in views are limited to five attempts per minute for each client address by throttle classes in tracking/throttles.py.

The helper is called by the driver-session, activate, deactivate, start-fresh, start-trip and location views, so an unauthorised driver can neither operate nor report the position of a bus to which they are not actively assigned. The views that use it require an authenticated session.

5.4.4 Bus Operations and Trip Start

Activation, deactivation and reset are POST operations that call the Bus methods after authorisation. start_trip retrieves the bus and route (returning HTTP 404 if either does not exist), chooses the trip with get_trip_for_route_and_destination, finds the first stop time by stop sequence, and in one transaction sets the bus's current trip and current stop time, sets the status to IDLE and marks the bus active. If no trip ends at the chosen destination, start_trip raises a ValueError. The start-trip view does not catch this exception, so that case would produce a server error rather than a clear client error. This is recorded as a defect in Section 6.6.

5.4.5 Real GPS Update Handling

The location view authorises the driver and then calls save_bus_location with the latitude, longitude and speed from the request body; the speed defaults to zero if absent. save_bus_location loads the bus, stores the three values, calls update_next_stop, derives the status with get_bus_status and saves the bus. The response contains a confirmation message, the registration number, the latitude, the longitude, the speed and the status. The values are not validated before storage; validation of coordinates is applied only when an ETA is calculated.

update_next_stop does nothing if the bus has no trip or no current stop. Otherwise it asks has_departed whether the bus has left its current stop. If so, it moves the current stop time to the next stop time of the same trip. The status logic and thresholds are in Table 4.9.3.1. A failure of the routing request is caught in both services, so a GPS update is still saved when OSRM is unreachable.

5.4.6 Waiting Requests

The waiting-request viewset is restricted to authenticated users and returns only requests that belong to the signed-in passenger. When a request is created, the serialiser accepts only representative trips and checks that the chosen stop is served by the trip. The view then refuses creation while the passenger has a request that is ON_BOARD or WAITING, so a passenger cannot hold two active requests. The service functions mark_as_boarded, cancel_waiting_request, complete_waiting_request and expire_waiting_request change the status, filter on the expected current status and set the deactivation time where relevant. get_waiting_count counts WAITING requests for a trip and stop, and get_route_waiting_overview returns, for the current trip of a bus, each stop in sequence with its waiting count, or an empty list when the bus has no trip.

5.4.7 OSRM Integration

All routing calls pass through get_road_distance, which builds a request to the OSRM route service from the configured base URL and returns the distance in metres and the duration in

seconds of the first route [8]. Coordinates are sent as longitude then latitude, and the overview geometry is switched off because only distance and duration are needed. The base URL is read from OSRM_URL, with the default http://localhost:5000, and any trailing slash is removed.

```
OSRM_URL = os.getenv("OSRM_URL", "http://localhost:5000").rstrip("/")

def get_road_distance(start_lat, start_lng, end_lat, end_lng):
    url = (f"{OSRM_URL}/route/v1/driving/{start_lng},{start_lat};{end_lng},{end_lat}"
           "?overview=false")
    response = requests.get(url)
    response.raise_for_status()
    route = response.json()["routes"][0]
    return {"distance_meters": route["distance"], "duration_seconds":
route["duration"]}
```

The call has no explicit timeout. The status and departure services catch only RequestException, so a malformed OSRM response in those services is not handled in the way it is in the ETA service, which also catches index, key, type, value and overflow errors. This limitation is recorded in Section 6.6.

5.4.8 ETA Calculation

calculate_waiting_request_eta implements the steps in Section 4.9.5. It uses a database query to select candidate buses and then filters them in Python, checking that coordinates are finite and in range, that the bus has not passed the stop and that the OSRM duration is finite and non-negative. A second function, calculate_eta, returns the road duration in minutes from a bus to a stop for the bus-and-stop endpoint, and it does not apply these checks.

5.4.9 Password Reset and Deployment Configuration

Django’s built-in password-reset views are routed at password-reset/, with a reset timeout of 3600 seconds. E-mail is sent with the Brevo backend of django-anymail, using the credential and sender address from the environment. The settings read the secret key, debug flag, allowed hosts, trusted CSRF origins and database address from the environment, and refuse to start without a secret key when debugging is off. Debugging is off unless the debug flag is set to true. In that mode the settings also mark session and CSRF cookies as secure and trust the proxy’s forwarded-protocol header. The application is served by gunicorn through bus_tracking.wsgi.application, and WhiteNoise serves static files.

On Render the service is configured with the following commands, copied from its settings. The build command installs the pinned dependencies and collects the static files; the start command applies any pending migrations and then starts gunicorn, bound to all interfaces on the port that the platform supplies in the PORT variable. Migrations therefore run each time the service starts.

```
Build command:
pip install -r requirements.txt && python manage.py collectstatic --noinput
```

```
Start command:
python manage.py migrate --noinput && gunicorn bus_tracking.wsgi:application --bind
0.0.0.0:$PORT
```

5.4.10 Driver Interface (author)

The driver dashboard is the component that supplies real GPS data to the system, which is the change that supervisor feedback required. The author developed it on the Driver-Interface branch. Its implementation consists of a Django template that carries the page structure, styles and scripts, and that calls the API with the browser's Fetch API and the CSRF token. Its behaviour is described by the stages of a driver's session.

Sign-in and bus selection. The dashboard posts the driver's credentials to the driver login endpoint; on page load it first asks the server for the current driver, so an existing session is restored. On success it requests the buses assigned to the account and lists them, and for the selected bus it loads the driver-session record (route, destination, target stop and last update of a trip in progress). Messages distinguish a wrong password (401) from an account that is not an authorised driver or a bus that is not assigned (403). A Log out button ends the session. The server, not the page, decides which buses the driver may operate: an account without a driver profile, an inactive driver or a missing assignment results in HTTP 403, and the dashboard shows that the operation was refused.

Bus operations and trip start. For the selected bus the dashboard offers activation, deactivation and reset of the operational state, each a POST to the corresponding endpoint. The driver then chooses a route and a destination from the route search and destination endpoints and starts the trip with the start-trip endpoint. After the trip starts the dashboard lists the waiting passengers for each stop of the trip, taken from the waiting-overview endpoint that only the assigned driver may call.

Device GPS tracking. When the driver starts tracking, the page calls navigator.geolocation.watchPosition. The browser first asks the device owner for permission (Figure 6.3.6). The call uses high accuracy, a maximum cached-position age of 1 second and a 15-second timeout. Each position fix is sent as latitude, longitude and speed to the location endpoint of the bus, with the speed converted from metres per second to kilometres per hour (the browser value multiplied by 3.6); a flag ensures that only one update request is outstanding at a time. After each accepted update the dashboard reloads the waiting overview. The dashboard displays the server's reply: a message that the GPS update was accepted, the reported coordinates, the speed and the status that the server derived, and it moves a marker on a Leaflet map to the reported position (Figure 6.3.7). A control stops tracking.

Trip lifecycle and stop progression on the branch. The Driver-Interface branch, commit c816533 "Complete driver interface and trip lifecycle", also contains work on the trip lifecycle, on the location and stop-progression logic, on related server-side changes, on tests

and on supporting files. These server-side changes were needed so that the driver screens could start a trip, report a position and see the bus move through the stops of the trip. Some of the files that this branch touches are shared backend components. The author's ownership of a given server-side function cannot be established from the file's location alone; Section 5.7 states what is attributed to the author and what is treated as collaborative. The individual files and functions that the commit changed are not itemised in this report.

5.4.11 Passenger Interface (Ngosa Muleba)

The passenger interface was developed mainly by Ngosa Muleba. It is a Django template (index.html) that shows a Leaflet map with OpenStreetMap tiles, polls the bus data every 15 seconds and opens the trip-planning panel and the request list over the map. The detailed implementation follows, from the passenger-interface report.

5.4.11.1 Delivery and Structure of the Page

The Django project serves the passenger page as an HTML template. Its styles and scripts are written inline and no front-end framework is used. The page has one full-screen map beneath a fixed header, and the panels that the header controls open are layered over the map. The scripts call the JSON API with the browser's Fetch API and send the CSRF token with each request that changes data. The passenger's session is held in the session cookie that the server sets at sign-in. Because the page needs no build step and no installation, a passenger needs only a browser and a network connection.

5.4.11.2 The Live Bus Map

The map is created with Leaflet and OpenStreetMap tiles. For each active bus that has a position, the page draws a bus marker. The marker's popup gives the registration number, the status (IDLE, IN_TRANSIT or AT_STOP) and the speed, as supplied by the server. The page requests the bus data when it loads and repeats the request every 15 seconds; the header refresh control and the floating refresh control request it immediately. This is polling over ordinary HTTP. The proposal had planned Firebase for real-time delivery, but the final system uses no WebSockets (Table 3.2.4.1), and polling was sufficient for a prototype. The speed value is displayed as the server supplies it; the unit is not specified in the server code, so the report assigns none.

5.4.11.3 Account Area

At the top of the Plan Trip panel is the account area, with fields for the account details and the controls *Register*, *Login*, *Forgot password?* and *Logout*. Registering and signing in call the passenger authentication endpoints. The server applies the configured password validators at registration and limits sign-in to five attempts a minute for each client address, so the page does not implement those rules itself. The *Forgot password?* link leads to the password-reset page that the server provides. A waiting request needs a signed-in passenger, so the account area comes first in the panel.

5.4.11.4 Route Search and Destination Choice

Typing in the *Search route* field requests the routes whose names match the text, and the matches appear in a selection list (Figure 4.8.1 and Figure 6.3.10). In the test run, the text *bauleni* returned the route *Bauleni – Kulima Tower*. Selecting a route requests its destinations. Under the heading *Trip and stop details* the page then shows the route name and the prompt *Select a destination*, with one button for each destination; for this route the buttons were *Bauleni Junction* and *Kamwala Market – Independence Ave*.

5.4.11.5 Boarding Stop Selection

When the passenger selects a destination, the page requests the ordered stops of the trip and lists them, one button per stop, in the order the bus visits them (Figure 6.3.11). The passenger touches a stop, which is highlighted, and a blue banner confirms the choice, for example *Selected boarding stop: Civic Center*. The list is long on routes with many stops, up to 38 in the data (Table 3.3.4.1), so it scrolls inside the panel.

5.4.11.6 Creating and Managing Waiting Requests

The green *Create Waiting Request* button sends the chosen trip and stop to the server, which creates the request in the WAITING state for the signed-in passenger, and a green banner confirms it (Figure 6.3.12). The one-active-request rule and the checks that the stop is served by the trip are enforced by the server. The *My Requests* control opens the sheet that lists the passenger's requests. Each card shows the route name, the boarding stop, the time and the status. For a WAITING request the card shows the board and cancel controls; after boarding it shows the control that completes the trip (Table 4.9.8.1). Each control calls the matching server operation and the list is updated with the new status. Because the server filters the list by the signed-in passenger, the page shows no one else's requests.

5.4.11.7 Displaying the Arrival Estimate

For a request in the WAITING state the page asks the server for the estimate and shows it on the card as *Estimated arrival at …: about N minutes*. In the test run the estimate was about 10 minutes (Figure 6.3.12). This is the road-route duration of the nearest eligible bus as calculated by OSRM; it was not compared with an observed arrival time. When the server has no valid estimate it returns null, and the page has no figure to show.

5.4.11.8 Feedback to the Passenger

The page confirms the important steps with coloured banners at the top of the screen. A blue banner confirms the selected boarding stop and a green banner confirms the created waiting request. The status of each request is shown in the list in a distinguishing colour, so that waiting and cancelled requests can be told apart at a glance (Figure 6.3.12).

5.5 Integration

Integration was carried out by Kenneth Kaunda Musonda after the interface branches had been developed. It took place at four boundaries. First, the browser interfaces call the JSON API under /api/; the Django project also serves their HTML pages for the landing, passenger and driver screens. Second, the views call the service modules, which use the ORM to read and write the database. Third, the distance service calls OSRM over HTTP. Fourth, the settings read the database address from DATABASE_URL, and the e-mail backend sends through Brevo. The integration branch was used as the single source of truth for the integrated system, and the endpoints in Table 4.5.3.1 and Appendix F are those of that branch.

The author's Driver-Interface branch is a development branch. It differs from the integration branch, and the tests run on it are not the same as the tests of the integrated system (Section 6.3). It could not be established from the available records whether, or by which commit, the Driver-Interface branch was merged. The integration branch does contain the driver dashboard (driver_interface.html) and the driver, trip and GPS endpoints that the dashboard uses, so the driver-side functionality described in this report is present in the integrated system. Whether each individual change on the branch reached the integration branch was not established.

5.5.1 Integration as Seen from the Passenger Interface

Integration took place at three boundaries from the passenger interface's side. First, the page calls the JSON API under /api/ built on the server side of the project; Table 5.5.1 lists each passenger action and the call it triggers. Second, the passenger's waiting requests are consumed by the driver dashboard, which shows the number of passengers waiting at each stop of the bus's current trip, so a request created on the passenger page is what the driver sees (Figure 6.3.13). Third, the page relies on the driver dashboard indirectly, because the bus markers appear on the passenger map only after a driver has activated a bus, started a trip and sent a GPS update. The integration branch was used as the single source of truth.

Table 5.5.1: Passenger-interface actions and the API calls they trigger

Passenger action	Method	Endpoint (under /api/)
Register	POST	auth/register/
Login	POST	auth/login/
Logout	POST	auth/logout/
Open map, refresh (every 15 seconds)	GET	buses/
Type in Search route	GET	routes/search/?q=
Select a route	GET	routes/<route_id>/destinations/

Passenger action	Method	Endpoint (under /api/)
Select a destination	GET	routes/<route_id>/stops/?destination=
Create Waiting Request	POST	waiting-requests/
Open My Requests	GET	waiting-requests/
Mark as Boarded	POST	waiting-requests/<id>/board/
Cancel Trip	POST	waiting-requests/<id>/cancel/
Complete the trip	POST	waiting-requests/<id>/complete/
Show arrival estimate	GET	waiting-requests/<id>/eta/

5.5.2 Integration as Seen from the Driver Interface

From the driver dashboard's side, integration took place at three boundaries. First, the dashboard calls the JSON API under /api/ for sign-in, the assigned buses, bus operations, trip start, GPS updates and the waiting overview (Table 4.5.3.1 and Appendix F). Second, the positions it sends are what the passenger map displays: a bus marker appears on the passenger map only after the driver has activated the bus, started a trip and sent a position (Figure 6.3.8). Third, the passengers' waiting requests are what the driver dashboard counts for each stop of the trip, so a request created on the passenger page is the request the driver sees (Figure 6.3.13). The dashboard and the passenger page were developed in parallel against the same endpoints and were integrated by the integrating member into the final branch.

5.6 Challenges and Solutions

The challenges below are those of the group project. Items that concern the driver interface are marked.

- **From simulated to real GPS.** The proposal assumed simulated positions. After supervisor feedback, the system was changed to accept real coordinates and speed from a driver's device, protected by the assignment check, so that the source of position is an identified, authorised driver. The driver dashboard is the component that carries this change.
- **Giving a coordinate meaning.** A coordinate alone does not tell a passenger which stop is next. GTFS routes, trips and ordered stop times were imported and a bus keeps references to its current trip and stop, so position can be related to the route and the stops ahead.
- **Repeated trip variants.** The GTFS feed holds three start-time variants of each trip with identical stops. Using the 08:00:00 variant as the representative trip avoids ambiguity when a passenger or driver selects a route and direction, at the cost of not modelling the time of day.

- **Unreliable routing input.** OSRM may be unreachable or return unusable data. The ETA returns null in such cases instead of an unsupported number, and the status services fall back to IN_TRANSIT or not departed, so a driver's GPS update is still accepted.
- **Change of the waiting-request model.** The migration history shows the request moving from a route-based to a trip-based design so that a request refers to a specific route direction and stop.
- **Developing interfaces against a changing backend.** The interfaces were built while the backend was still being extended, so the endpoints that the screens call had to be kept consistent across branches until integration.
- **Testing driver features without an operating bus.** No operating bus was available, so driver GPS reporting was checked with a test device. The status shown after the first update in Figure 6.3.7 is IN_TRANSIT and not AT_STOP because the test device was not within 30 metres of the bus's current stop (Table 4.9.3.1).
- **Secrets and environments.** Secrets and addresses were moved into environment variables, and the settings fail safely when the secret key is missing outside debug mode. Debug mode was also made opt-in.
- **Open endpoints.** A code review found that the model viewsets and the expire endpoint allowed anonymous access. The default permission, read-only viewsets, the removal of the passenger endpoint and the administrator restriction closed these gaps, and tests now check that anonymous, passenger and driver users are refused.
- **Challenges in the passenger interface.** The following were recorded in the passenger-interface report.
  - **Giving a coordinate meaning.** A bus marker alone does not tell a passenger which bus it is or whether it is moving. The marker popup gives the registration number, status and speed, and the trip-planning panel relates buses to routes and stops through lists.
  - **Choosing among many routes.** The GTFS data has 68 routes in two directions. Searching by typed name and then choosing a destination narrows the choice to one representative trip without asking the passenger to understand directions.
  - **Keeping the map current without a push connection.** The page polls every 15 seconds and offers two refresh controls, so a passenger who doubts the display can update it at once.
  - **Not misleading with the arrival time.** The estimate is shown only for a waiting request and only when the server returns a value, because a wrong estimate is worse than none.

  - **Fitting several functions on one screen.** The map is the base screen and the trip planning and the request list open over it, so the passenger does not lose sight of the buses while planning.
  - **Working while the server was changing.** The endpoints were developed in parallel with the page. Ngosa Muleba and the backend developer (Kenneth Kaunda Musonda) worked from a shared list of endpoints (Table 4.5.3.1), and the page was integrated against the final branch.
  - **Change of the waiting-request model.** The request moved from a route-based to a trip-based design during development (Section 3.2). The page was adjusted so that a request names a route direction and a boarding stop.

- **Challenges in the driver interface.** The following follow from the design and the checks of the driver dashboard.
  - **Using the device's own position.** The position comes from the browser's geolocation service, which needs the driver's permission and may fail or be slow. The dashboard asks for permission, shows an error when no position can be supplied and keeps working.
  - **Avoiding a queue of requests.** A new fix can arrive while the previous update is still being sent. Only one update request is outstanding at a time and a fix that arrives meanwhile is skipped.
  - **Showing what the server accepted.** The dashboard displays the position, speed and status that the server returns, not a locally assumed state, so the driver sees the state that passengers see.
  - **Status that depends on the stop.** The status stays IN_TRANSIT until the device is within the stop distance and below the stop speed, so a test device away from the stop shows IN_TRANSIT (Figure 6.3.7).
  - **Testing without an operating bus.** No operating bus was available, so the driver flow was checked with a test device and a test bus.
  - **Working while the server was changing.** The dashboard was built while the endpoints it calls were still being extended, so it was integrated against the final branch.

5.7 Individual Contribution (Group Projects)

The author's individual contribution is the driver interface and the driver-side functionality of the Bus Tracking System. Table 5.7.1 distinguishes the author's work from shared work and from the work of the other members. Appendix A gives the formal Individual Contribution Statement. The attribution of server-side files rests on the development history described by the author and on the sources listed in Section 5.7.1.

Table 5.7.1: Contribution summary by activity

Activity	Contribution	Notes
Driver dashboard: sign-in, bus selection, activation, deactivation and reset, trip start, waiting counts by stop, status display	Primary (author)	Section 5.4.2; Driver-Interface branch, commit c816533 as recorded by the author; see Section 5.7.1
Device GPS tracking from the driver's browser and display of the accepted position on the driver map	Primary (author)	Section 5.4.2
Trip-lifecycle and stop-progression changes needed by the driver interface, including related server-side changes and tests	Collaborative; the author's own share is not itemised in the available records	Section 5.4.2; the branch touches shared backend files
Initial backend skeleton (Django project, initial models, migrations, first endpoints)	Kenneth Kaunda Musonda	Section 5.4.1; not claimed
Backend updates made while the interfaces were developed	Collaborative	Section 5.4.4
GTFS import commands	Kenneth Kaunda Musonda (per the group report)	Section 5.4.1; not claimed
OSRM integration and ETA logic	Kenneth Kaunda Musonda (per the group report)	Section 5.4.4; not claimed
Integration of the interface branches, security hardening and deployment configuration	Kenneth Kaunda Musonda	Sections 5.4.5 and 5.5; not claimed
Passenger interface	Ngosa Muleba	Section 5.4.3; not claimed
Problem analysis, requirements and Yango interface analysis	Joint (group)	Section 3.3
Literature review and group presentation	Joint (group)	Chapter 2; the review was prepared for the group report and is reused here
Tests on the Driver-Interface branch	Branch run by the author; authorship of individual tests not established	Section 6.3
Automated tests of the integrated system (35)	Kenneth Kaunda Musonda	Section 6.3; not claimed
Manual checks of the deployed flows	Collaborative	Section 6.3

5.7.1 Evidence and Limits of the Attribution

The attribution in this chapter rests on four sources: the group report and the group presentation, which assign the driver dashboard to the author; the individual report on the passenger interface, which lists the driver dashboard as the work of the author; the author's

account of the Driver-Interface development branch, which was created in the same repository as the integrated system; and the integrated code, which contains the dashboard described in Section 5.4.2. Because the interface branches were merged into the integration repository by the integrating member, the history of that repository lists the integrating member against most commits and does not by itself itemise who wrote each file. For that reason this report attributes work to the author only where the group documents and the author's account agree, and it treats server-side functions that were changed while the interfaces were developed as collaborative. The supervisor or the other group members can confirm the attribution if required.

The author does not claim the initial backend skeleton, the integration of the branches or the deployment, which were carried out by Kenneth Kaunda Musonda, and does not claim the passenger interface, which was developed mainly by Ngosa Muleba. Equally, the backend changes that were made to support the interfaces are not attributed to a single member.

5.8 Chapter Summary

The system was developed in sequence: an initial backend skeleton, then the driver and passenger interfaces on separate branches with backend updates alongside them, then integration, security hardening and deployment. The driver dashboard supplies real GPS position, speed and bus operations to the system, and the author developed it together with the related trip-lifecycle and stop-progression work on the Driver-Interface branch. The backend implements the data model, GTFS imports, driver authorisation, bus and trip operations, GPS handling, waiting requests, OSRM integration and conditional ETA in function-based service modules, configured for deployment through environment variables. Individual attribution of shared server-side files remains subject to verification against the Git history. Some implementation limits were identified and are recorded for Chapter 6.

CHAPTER 6: TESTING, RESULTS AND EVALUATION

6.1 Introduction

This chapter reports the testing evidence for the web application as a whole and evaluates the system against the group objectives of Section 1.4, within its limits. Three kinds of evidence exist and they are kept separate throughout the chapter because they come from different code states and different people.

1. **Integrated-system automated tests.** The 35 automated tests of the integration branch, run in the deployment working copy of the system (Section 6.3.1).
2. **Branch-specific tests.** The test run on the author's Driver-Interface development branch, which reported 94 passing tests (Section 6.3.2).
3. **Manual functional and integration checks** of the deployed passenger interface, driver dashboard and server working together (Section 6.3.3).

The chapter also draws on a review of the source code. Section 6.8 then separates the results attributable to the author. The testing was not exhaustive acceptance testing, and no performance benchmark, user study or field trial was carried out, so none is claimed. Where the evidence is absent the chapter says so.

6.2 Testing Strategy

The automated tests use Django's test framework with Django REST Framework's APITestCase. Each test builds its own data in a temporary test database, authenticates a test user and calls the real endpoint through the test client, so the tests exercise the view, serialiser, service and model together. In the integrated suite the routing function used by the ETA service is replaced by a mock, so the tests do not depend on a running OSRM server and do not test OSRM itself. The tests are run with python manage.py test, which discovers test modules in the project [13]. The command python manage.py check was also used. It is different: it validates the Django configuration and is not a test run.

The strategy of the integrated suite first concentrated on the most conditional piece of logic, the ETA, together with the waiting-passenger overview used by drivers. In the final stage of development, tests were added for the access rules of the API, registration, login throttling and the debug default. Other modules were verified by inspection of the source code but have no automated tests in the integrated suite. The branch tests were run during development of the driver interface and its trip-lifecycle and stop-progression logic (Section 6.3.2). The second level of testing was manual functional and integration testing. Key flows were checked through the deployed passenger interface and driver dashboard and the connected Django application, so that the parts were tested working together rather than as isolated

services. The browser interfaces have no automated tests, so manual checking is the only evidence for them.

Manual checks of the passenger flows. The manual checks followed the passenger's path through the system: sign in, view the map, search a route, choose a destination and a boarding stop, create a waiting request, read its arrival estimate, use the controls on the request, and confirm that the driver dashboard shows the request. They were made on the deployed system using a test passenger account, a test driver account assigned to a test bus, and the location reported by the device used for the checks, which was not an operating bus. The screenshots in this chapter were taken during the checks. Each shows the outcome of a flow at one moment; they are not a table of expected and actual results with pass or fail status.

The server's automated tests use Django's test framework with Django REST Framework's APITestCase. Each test builds its own data, authenticates a test user and calls the real endpoint, and the routing function used by the ETA service is replaced by a mock, so the tests do not depend on a running OSRM server and do not test OSRM itself. They were written by the backend developer (Kenneth Kaunda Musonda). They are reported here because they cover the behaviour behind what the passenger sees: the arrival estimate, the scoping of requests to their owner and the access rules.

Manual checks of the driver flows. The manual checks of the driver dashboard followed the driver's path through the system: sign in, select the assigned bus, activate it, choose a route and destination, start GPS tracking, allow the browser's location permission, and see the accepted position and the waiting passengers. They used a test driver account assigned to a test bus, and a test device. They are listed in Table 6.3.5.

6.3 Test Cases and Results

6.3.1 Integrated-System Automated Tests (35 tests)

Table 6.3.1 lists the behaviours of the server that the integrated suite verifies. The group report states that the run was made by Kenneth Kaunda Musonda in the deployment working copy of the integration branch, with the debug environment variable set to True because debugging is off by default. Django found 35 tests, ran all 35 in 19.248 s and reported OK, and the system check that precedes the run identified no issues. Rows 1 to 9 are the earlier tests of the ETA and waiting-overview classes, and rows 10 to 19 are the access-control, registration, throttling and debug-default tests added in the final stage. The result does not show that the untested modules are correct. A reader can reproduce it by running the same command on the integration branch. The copy of the integration branch supplied for this report (bus-tracking-integration.zip) contains tracking/tests.py with exactly 35 test methods in these five classes (6, 3, 20, 4 and 2), which agrees with the reported run. The copy has no Git history, so it cannot show who wrote which test.

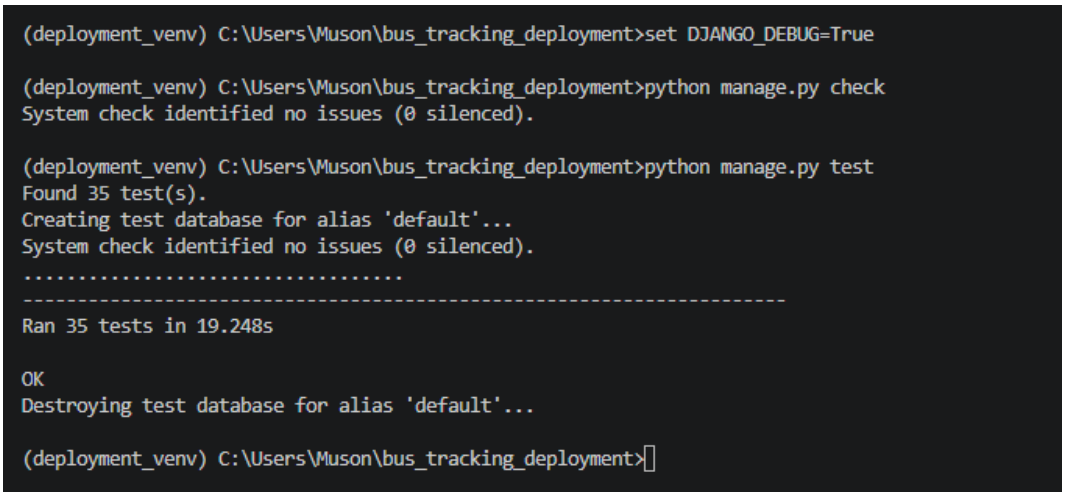
Table 6.3.1: Behaviours verified by the integrated-system automated tests

No.	Behaviour	Condition and asserted outcome	Test class
1	Shortest valid duration is chosen	Two eligible buses on the trip, OSRM mocked with different durations: the response contains the smaller duration in minutes and the routing function was called once per bus	WaitingRequestEtaTests
2	No ETA when GPS is unavailable	A bus without coordinates: the response has eta_minutes null and OSRM is not called	WaitingRequestEtaTests
3	Passed stop is excluded	A bus whose current stop is beyond the requested stop: ETA null and OSRM not called	WaitingRequestEtaTests
4	No ETA when not waiting	A request that is not in the WAITING state: ETA null	WaitingRequestEtaTests
5	Passenger scoping	A different passenger asks for the request: the response is HTTP 404 and OSRM is not called	WaitingRequestEtaTests
6	Unavailable routing response	The mocked routing function fails with an index error (no route): ETA null	WaitingRequestEtaTests
7	Waiting counts per stop	A boarding request is counted only at its own stop in the overview of the bus's current trip	WaitingOverviewTests
8	Non-waiting and other-trip requests excluded	Requests that are not WAITING, or belong to another trip, add nothing to the counts	WaitingOverviewTests
9	No active trip	A bus without a current trip returns a null trip and an empty stop list	WaitingOverviewTests
10	Transport writes disabled	POST, PUT, PATCH and DELETE on routes, stops and buses return HTTP 405 for anonymous, passenger and driver users, and the records are unchanged	ApiAccessSecurityTests
11	Passenger directory removed; public reads kept	The passenger endpoint returns 404; routes, stops, buses, route search, destinations, trip stops, bus ETA and waiting count return 200 without signing in	ApiAccessSecurityTests
12	Default permission	The default permission class is IsAuthenticated, and anonymous requests to the current-user and waiting-request endpoints return 403	ApiAccessSecurityTests
13	Driver authorisation	Passengers, unassigned drivers, inactive drivers and inactive assignments receive 403 on every bus operation and on the waiting overview, and the bus is unchanged; an assigned driver can update GPS and operate the bus	ApiAccessSecurityTests
14	Expiry restricted	Anonymous, passenger and driver users receive 403 and the request stays WAITING; an administrator receives 200 and the request becomes EXPIRED	ApiAccessSecurityTests
15	Password validation	Short, common and numeric passwords, and one equal to the username, are rejected with 400 and no account is created	ApiAccessSecurityTests

No.	Behaviour	Condition and asserted outcome	Test class
16	Waiting-request integrity	Update and delete return 405; status, passenger and timestamps supplied by the client are ignored on creation; another passenger cannot board, cancel or complete a request	ApiAccessSecurityTests
17	Passenger account required	Anonymous and non-passenger users receive 403 on every waiting-request path, and reads stay limited to the owner	ApiAccessSecurityTests
18	Login throttling	The sixth sign-in attempt in a minute from one address returns 429 for passenger and driver sign-in, other addresses are unaffected, the counters are separate and registration is not throttled	LoginThrottleTests
19	Debug default	The debug setting is false when the environment variable is absent and true when it is set to True	DebugDefaultTests

These tests cover valid behaviour, unavailable conditions and access restriction. Other modules were verified by inspection of the source code during development and during the preparation of this report, but they have no automated tests. The tests cover application and service-layer behaviour, not the visual layout of the frontend or every end-to-end user scenario.

The 35 tests are distributed over five classes: WaitingRequestEtaTests (6), WaitingOverviewTests (3), ApiAccessSecurityTests (20), LoginThrottleTests (4) and DebugDefaultTests (2). Row 13 is the test that bears most directly on the driver interface, because it verifies that an assigned driver can update GPS and operate the bus and that every other account is refused.



6.3.2 Driver-Interface Branch Tests (94 tests)

A test run on the author's Driver-Interface branch reported 94 passing tests. This result must be read carefully, and it is not combined with the integrated-system result for the following reasons.

- It is the result of a different code state. The branch is a development branch containing driver-interface, trip-lifecycle, location and stop-progression work and related server-side changes. It is not the integration branch from which the 35 tests were run, and it is not shown to be the deployed application.
- The figure of 94 is not the number of tests that the author personally wrote. The branch contains tests that belong to the shared backend as well as tests added for the driver and trip-lifecycle work, and the evidence supplied does not yet separate them.
- The figures of 94 and 35 are not additive and they are not directly comparable, because they come from different branches at different times.

What the result does establish is that, on that branch and at the time of the run, the test suite found no failure. The output of that run, its date and the list of its tests were not available when this report was written, so Table 6.3.2 records only what is known. The integration branch contains 35 tests, and the reason for the difference from 94 could not be established from the available records; no explanation is offered.

Table 6.3.2: Driver-Interface branch test results (what is known)

Item	Value
Branch and commit	Driver-Interface, c816533 “Complete driver interface and trip lifecycle”
Tests found and passed	94 passed (reported)
Date and environment of the run	Not recorded
Tests written by the author	Not established
Tests inherited from the shared backend	Not established
Tests concerning trip lifecycle, location and stop progression	Not itemised
Present in the integration branch	Not established; the integration branch contains 35 tests

6.3.3 Manual Functional and Integration Checks

The deployed system was also checked by hand through its passenger interface and driver dashboard. The checks covered access to the deployed application interfaces; the passenger-side and driver-side authentication and access flows; use of the driver dashboard by registered and authorised drivers; the driver’s trip and GPS-update workflow against the system; the GPS-related responses and the update of the bus’s current state; communication between the passenger-side functions and the deployed system; and the operation of the deployment configuration and the deployed Django application during the verification sessions.

These checks were made on the deployed system, using a test passenger account, a test driver account assigned to the bus BUS123, and the location reported by the device used for the checks, which was not an operating bus. Figures 6.3.2 to 6.3.14 are screenshots taken during the checks. Each shows the outcome of a flow at one moment; they are not a table of expected and actual results with pass or fail status. They demonstrate verification of key implemented functions. They do not constitute exhaustive acceptance, performance or large-scale field testing. The group report does not give the date of the checks; the deployed address is given in the project README (Section 1.7).

Driver flow (the author's interface). Figure 6.3.2 shows the landing page, where the user chooses the passenger or the driver sign-in. Figure 6.3.3 shows the driver sign-in. After signing in, the dashboard lists the buses assigned to the driver account and shows the status of the selected bus, IDLE before activation (Figure 6.3.4). Figure 6.3.5 shows the message that the bus was activated and the destinations offered for the route Bauleni – Kulima Tower. Figure 6.3.6 shows the browser asking permission to read the device's location, with the dashboard waiting for a fix. Figure 6.3.7 shows the result after the fix: the message that the GPS update was accepted, the reported coordinates, a speed of 0, the status IN_TRANSIT and the same position on the live map. The status is IN_TRANSIT and not AT_STOP because the device was not within 30 metres of the bus's current stop, which the status rule requires for AT_STOP (Table 4.9.3.1).

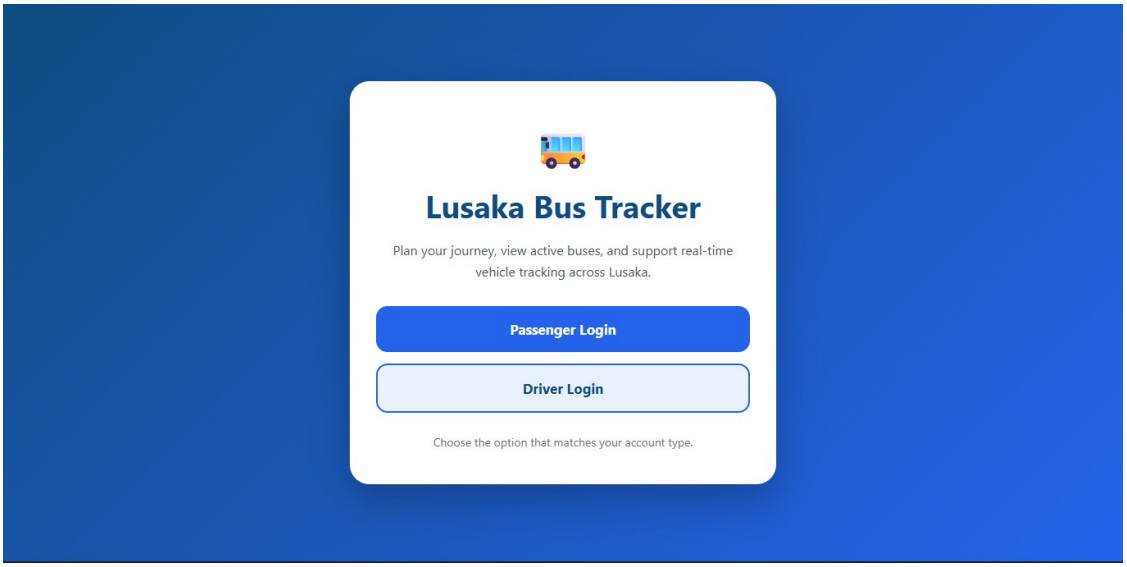


Figure 6.3.2: Landing page of the deployed system

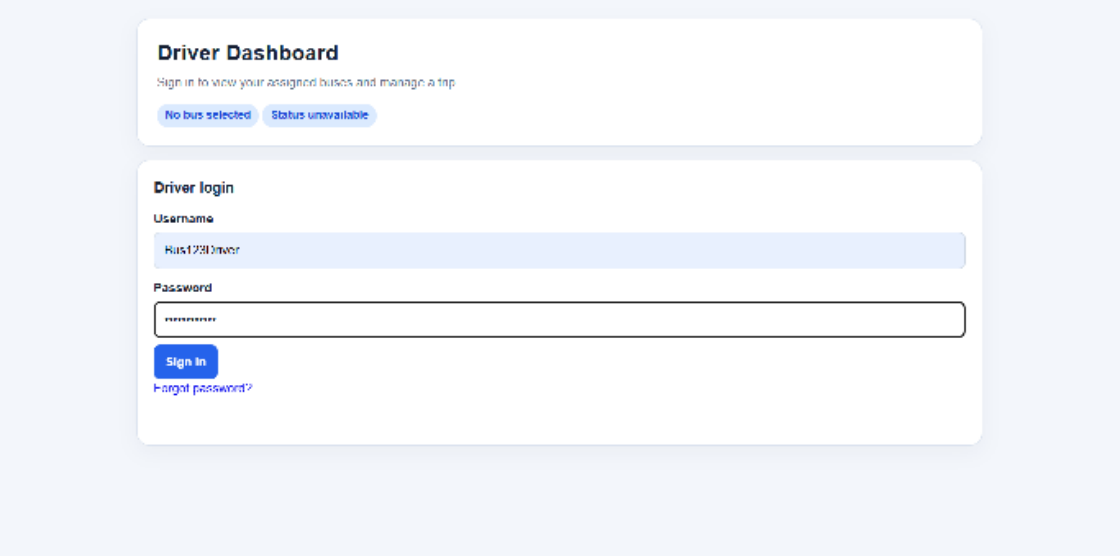


Figure 6.3.3: Driver sign-in (driver interface developed by the author)

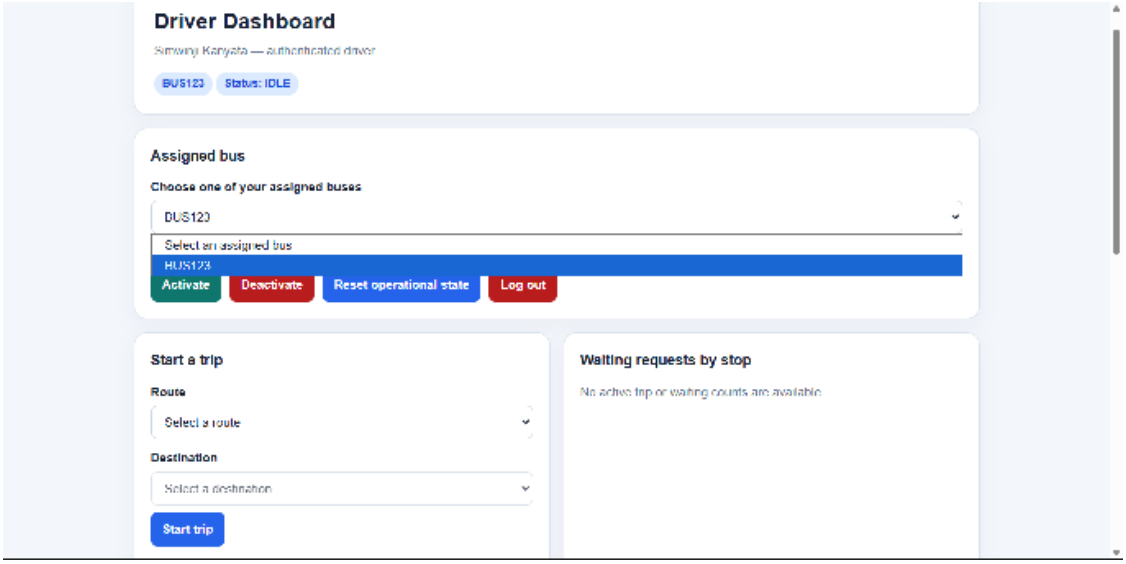


Figure 6.3.4: Driver dashboard with the assigned bus selected and its status IDLE (driver interface developed by the author)

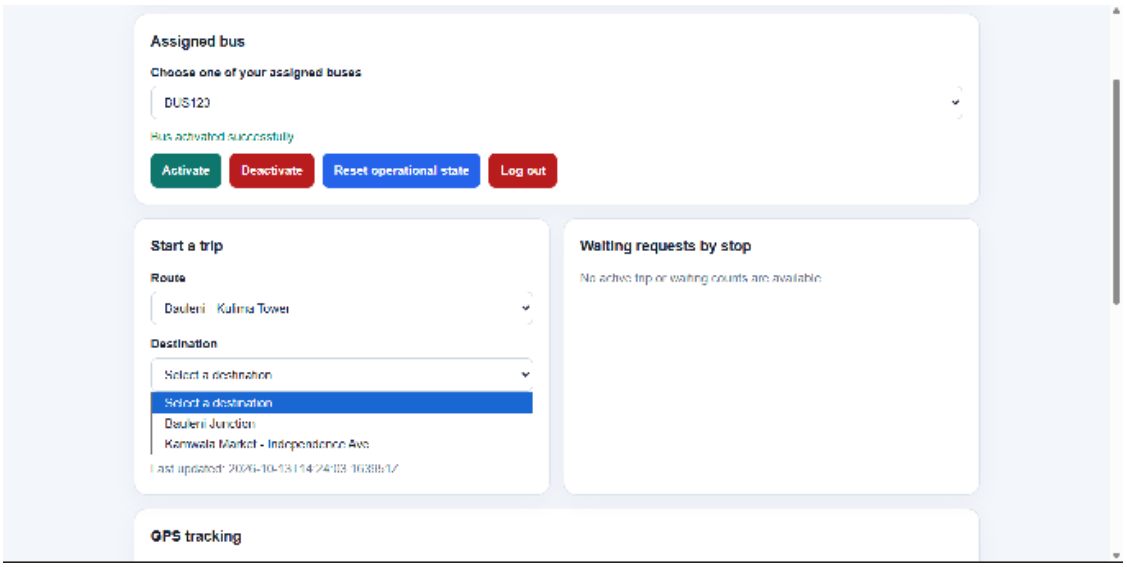


Figure 6.3.5: Bus activated and destination choice for the route Bauleni – Kulima Tower (driver interface developed by the author)

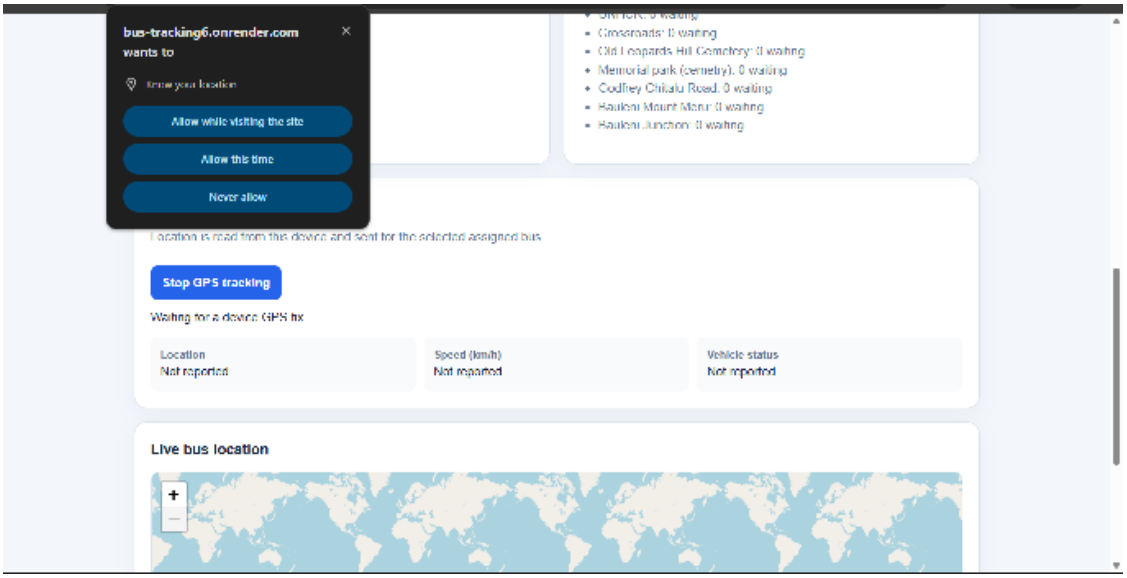


Figure 6.3.6: Browser permission request and waiting for a device GPS fix (driver interface developed by the author)

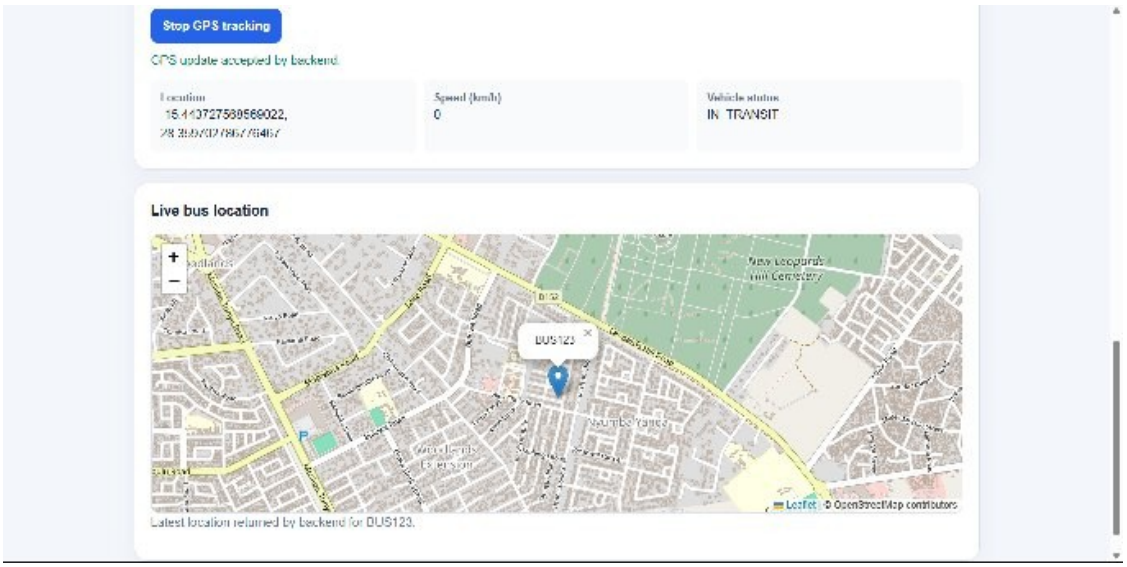


Figure 6.3.7: GPS update accepted, reported location, speed, status and live map (driver interface developed by the author)

Passenger flow (the passenger interface). The passenger flow is walked through in Sections 6.3.4 and 6.3.5, with the checks listed in Table 6.3.3 and the evidence in Figures 6.3.8 to 6.3.12.

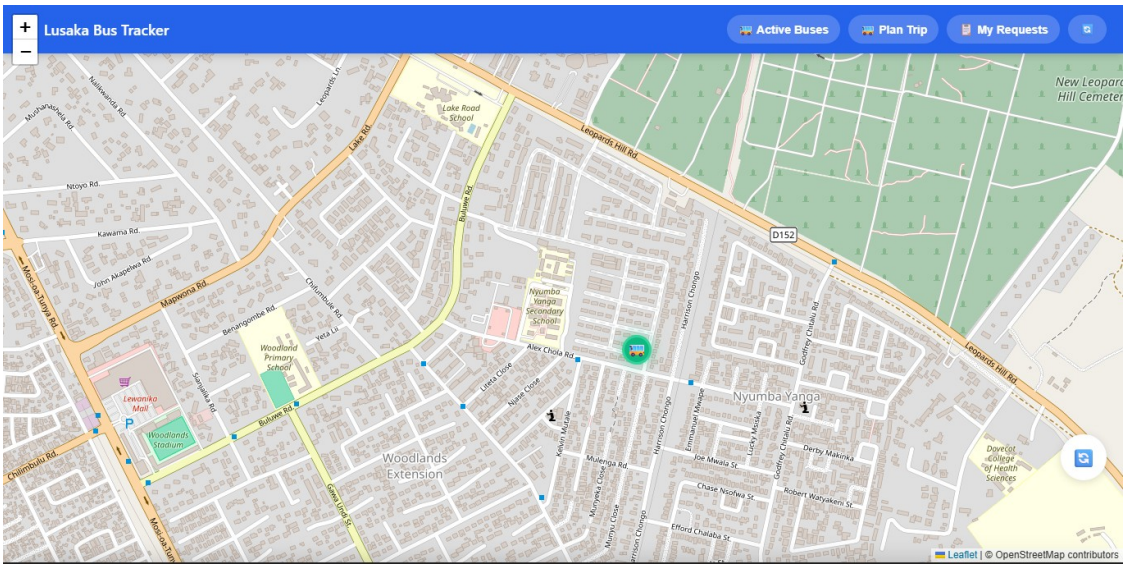


Figure 6.3.8: Passenger map showing the bus marker (passenger interface developed mainly by Ngosa Muleba)

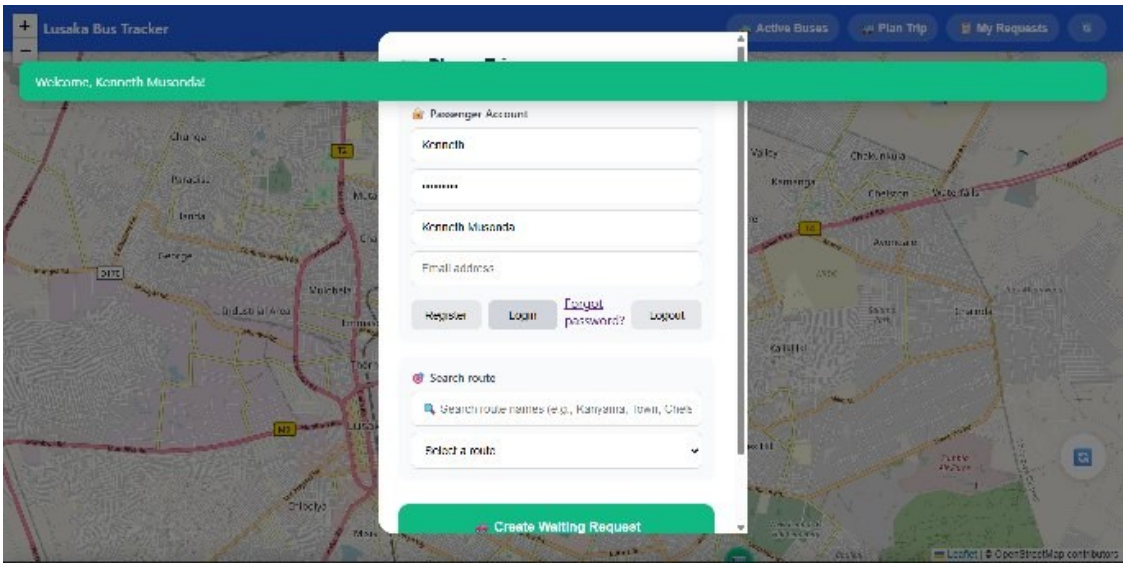


Figure 6.3.9: Passenger sign-in (passenger interface developed mainly by Ngosa Muleba)

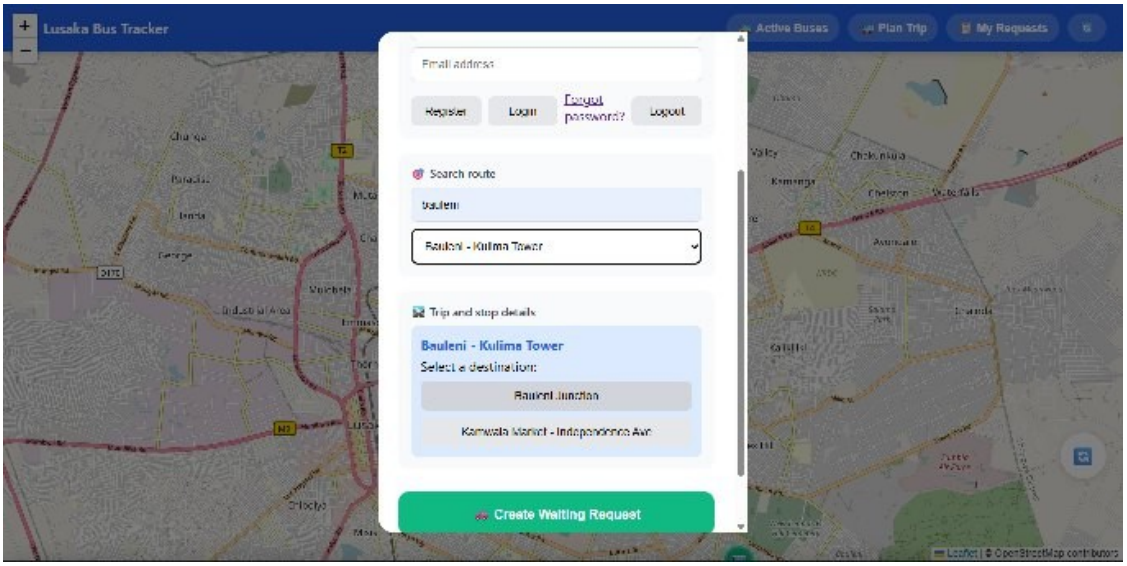


Figure 6.3.10: Route search and destinations offered (passenger interface developed mainly by Ngosa Muleba)

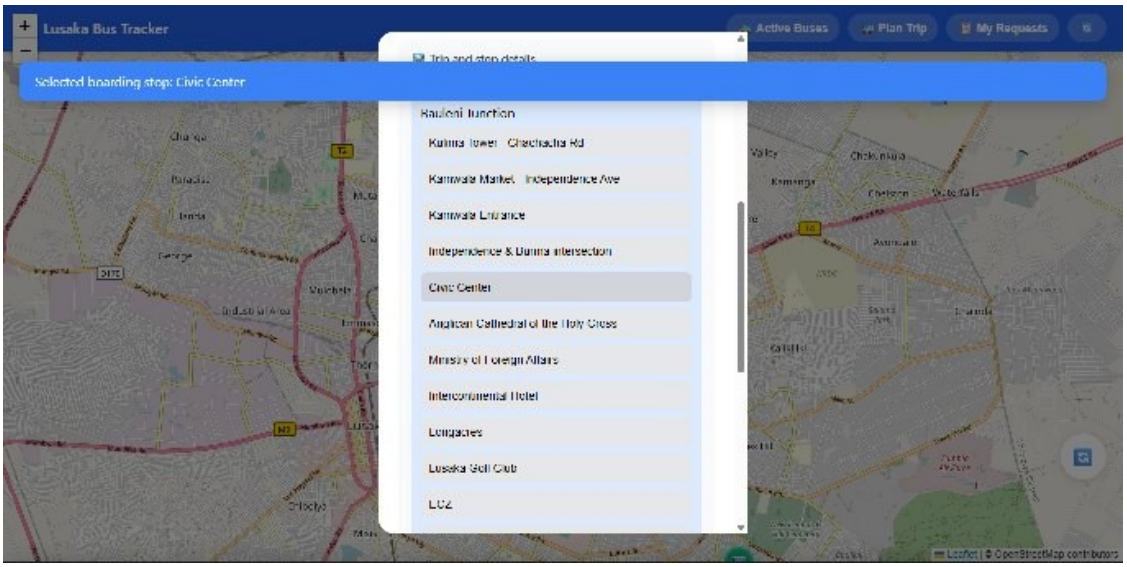


Figure 6.3.11: Boarding-stop selection (passenger interface developed mainly by Ngosa Muleba)

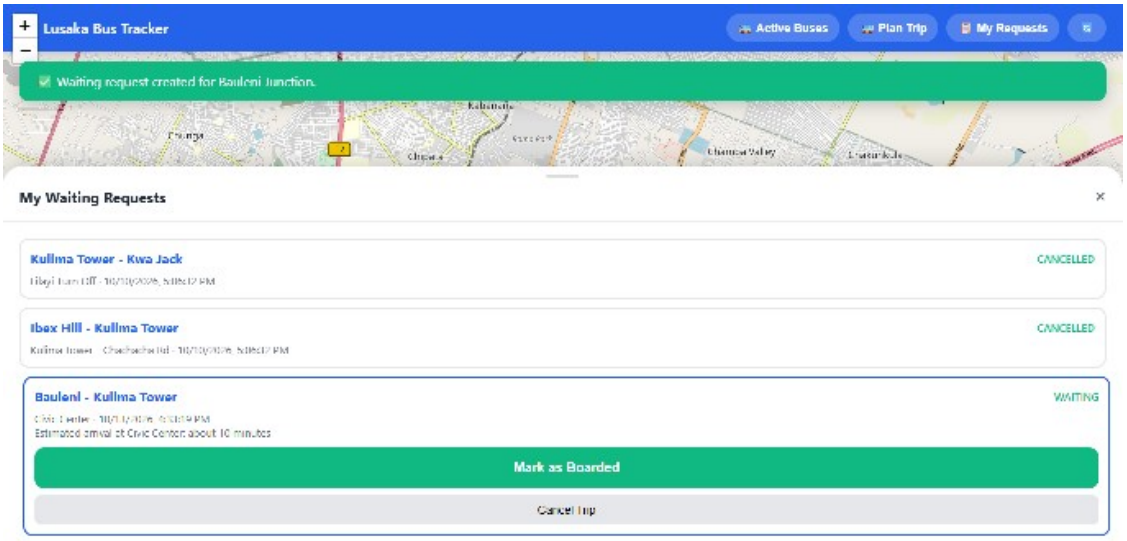


Figure 6.3.12: Waiting request created, with the arrival estimate and the options to board or cancel (passenger interface developed mainly by Ngosa Muleba)

Passenger request seen by the driver. Figure 6.3.13 shows both interfaces together: the passenger's waiting request and, on the right, the driver dashboard listing one passenger waiting at Civic Center, the stop chosen by the passenger. This is the end-to-end flow that connects the passenger interface, the server and the driver interface, and it is the check that bears most directly on group objective 5.

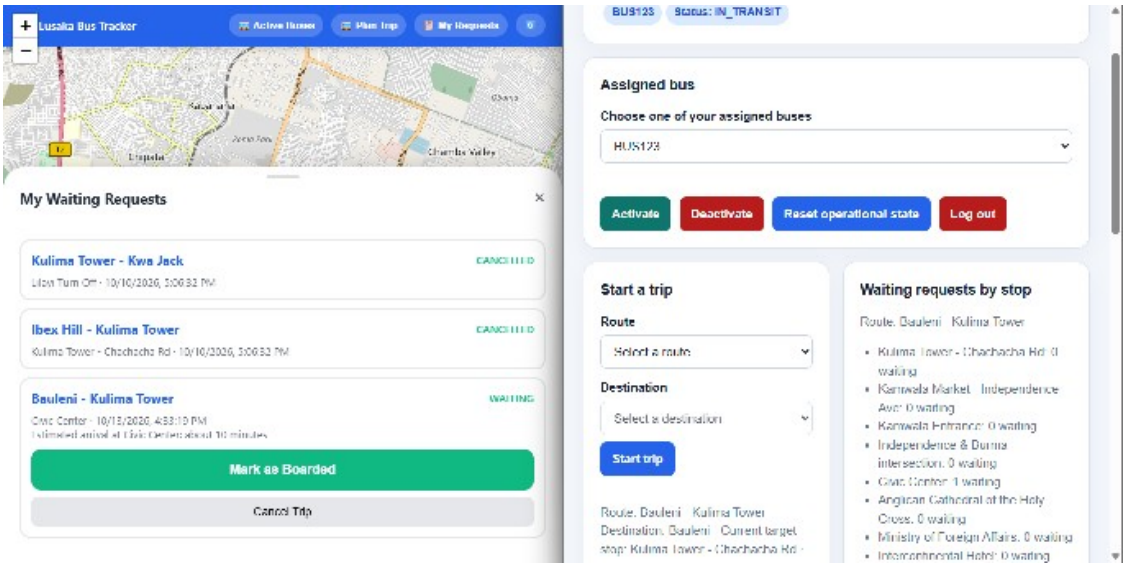


Figure 6.3.13: Passenger waiting request (left) and the driver dashboard showing one passenger waiting at Civic Center (right)

Operator management. Figure 6.3.14 shows the Django administration site of the deployed system, which lists the bus, driver assignment, driver, passenger, route, stop and waiting-request records that the operator can manage.

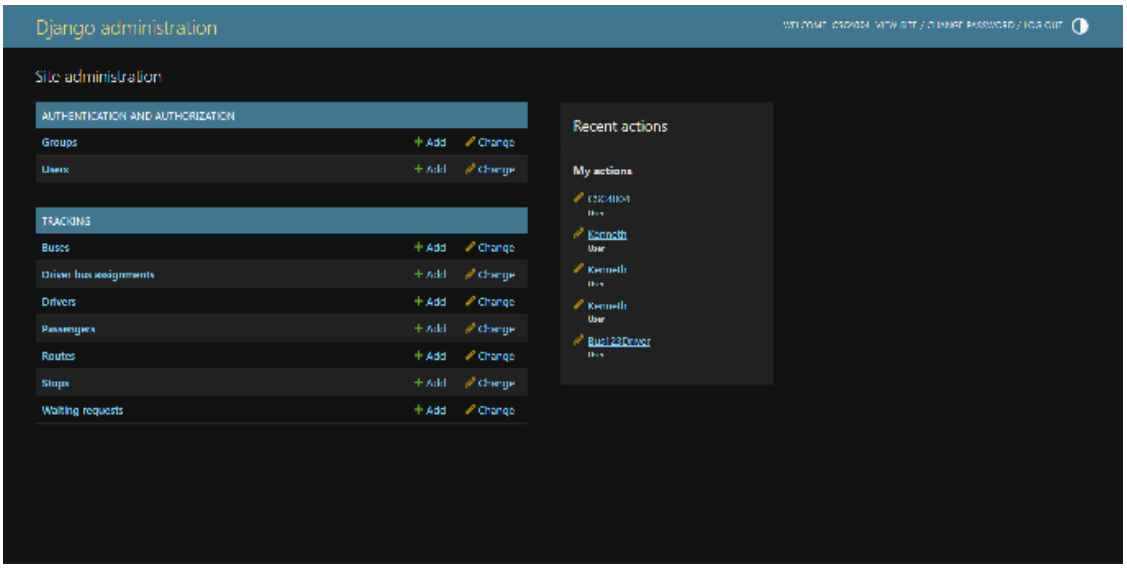


Figure 6.3.14: Django administration site of the deployed system

6.3.4 Manual Checks of the Passenger Flows

Table 6.3.3 lists the passenger flows that were checked, the outcome observed and the figure that shows it. The word *Observed* means that the outcome was seen on the deployed system and is recorded in the figure; it is not a formal pass mark. The last rows state the flows for which no screenshot evidence exists.

Table 6.3.3: Manual checks of the passenger flows on the deployed system

No.	Flow checked	Outcome observed	Evidence
M1	Choose the passenger role on the landing page	The landing page offers the passenger and driver sign-in	Figure 6.3.2
M2	Sign in with the test passenger account	The passenger sign-in screen was shown and used	Figure 6.3.9
M3	View the active bus on the map	A bus marker appeared at the position reported by the driver's device	Figure 6.3.8
M4	Search a route by typing *bauleni*	The route Bauleni – Kulima Tower was offered in the selection list	Figure 6.3.10
M5	Choose a destination	The destinations Bauleni Junction and Kamwala Market – Independence Ave were offered as buttons	Figure 6.3.10
M6	Choose a boarding stop	The ordered stops were listed; choosing Civic Center produced the message *Selected boarding stop: Civic Center*	Figure 6.3.11
M7	Create a waiting request	A confirmation banner appeared and the request was listed in the WAITING status	Figure 6.3.12
M8	Read the arrival estimate	The WAITING request showed an estimate of about 10 minutes; not compared with an observed arrival	Figure 6.3.12
M9	Use the request controls	WAITING requests offered Mark as Boarded and Cancel Trip; earlier requests were listed as CANCELLED; an ON_BOARD request offered the control to complete the trip	Figures 6.3.4, 4.8.4, 4.8.5
M10	See the request from the driver's side	The driver dashboard listed one passenger waiting at Civic Center, the stop chosen by the passenger	Figure 6.3.13
M11	Not evidenced by a screenshot	Registering a new account, the password-reset link, the 15-second refresh interval, a request whose estimate is unavailable, the completed state, and use on different browsers and devices were not recorded	None

6.3.5 Walkthrough of the Passenger Flow

The passenger map (Figure 6.3.8) shows the bus reported by the driver's device as a marker. Figure 6.3.9 shows the passenger sign-in, which precedes any request. Figure 6.3.10 shows the route search, in which the text *bauleni* returned the route Bauleni – Kulima Tower, and the destinations offered for it. Figure 6.3.11 shows the choice of a boarding stop, with the confirmation banner. Figure 6.3.12 shows the waiting request created for the journey, with its arrival estimate of about 10 minutes and the options to mark the passenger as boarded or to cancel. The estimate comes from the routing service; it was not compared with an observed arrival time. Figure 6.3.13 shows the same request with the driver dashboard alongside, which lists one passenger waiting at Civic Center.

6.3.6 Automated Server Tests that Support the Passenger Flows

The server developer ran python manage.py test on the deployed working copy. Django found 35 tests, ran all 35 and reported OK. Table 6.3.4 lists the tests that bear on what the passenger sees and does. The run does not show that untested modules are correct, and it does not test the passenger page.

Table 6.3.4: Automated server tests that support the passenger flows

Behaviour	Condition and asserted outcome	Test class	Result
Shortest valid duration is chosen	Two eligible buses with different mocked durations: the smaller duration, in minutes, is returned	WaitingRequestEtaTests	PASS
No ETA when GPS is unavailable	A bus without coordinates: eta_minutes is null	WaitingRequestEtaTests	PASS
Passed stop is excluded	A bus beyond the requested stop: ETA null	WaitingRequestEtaTests	PASS
Non-WAITING requests have no ETA	Requests in other statuses: ETA null	WaitingRequestEtaTests	PASS
Passenger scoping	A different passenger asks for the request: HTTP 404	WaitingRequestEtaTests	PASS
Unavailable routing response	The mocked routing function finds no route: ETA null	WaitingRequestEtaTests	PASS
Public reads kept	Routes, stops, buses, route search, destinations and trip stops return 200 without signing in	ApiAccessSecurityTests	PASS
Default permission	Anonymous requests to the waiting-request endpoints return 403	ApiAccessSecurityTests	PASS
Waiting-request integrity	Update and delete return 405; client-supplied status, passenger and timestamps are ignored; another passenger cannot board, cancel or complete a request	ApiAccessSecurityTests	PASS
Passenger account required	Anonymous and non-passenger users receive 403 on every waiting-request path; reads stay limited to the owner	ApiAccessSecurityTests	PASS

These cases confirm, within the fixtures that the tests build, that the estimate returned to the passenger page is null when its inputs are missing or invalid, that it is the shortest valid duration otherwise, and that a request is visible only to its owner. They support the design decision that the page needs to show nothing of its own when no estimate exists.

6.3.7 Manual Checks of the Driver Flows

Table 6.3.5 lists the driver flows that were checked on the deployed system, the outcome observed and the figure that shows it. The word *Observed* means that the outcome was seen on the deployed system and is recorded in the figure; it is not a formal pass mark. The last rows state the flows for which no screenshot evidence exists.

Table 6.3.5: Manual checks of the driver flows on the deployed system

No.	Flow checked	Outcome observed	Evidence
DM1	Choose the driver role on the landing page	The landing page offers the passenger and driver sign-in	Figure 6.3.2
DM2	Sign in with the test driver account	The driver sign-in screen was shown and used	Figure 6.3.3
DM3	Select the assigned bus	The bus BUS123 was listed and selected; its status was IDLE	Figure 6.3.4
DM4	Activate the bus and choose a route	A message confirmed the activation and the destinations of the route Bauleni – Kulima Tower were offered	Figure 6.3.5
DM5	Start GPS tracking	The browser asked for permission to read the location and the dashboard waited for a fix	Figure 6.3.6
DM6	Receive a position fix	The message that the GPS update was accepted by the backend appeared, with the coordinates, a speed of 0 and the status IN_TRANSIT, and the position was shown on the live map	Figure 6.3.7
DM7	Open the passenger map	A bus marker appeared at the position reported by the driver's device	Figure 6.3.8
DM8	View the waiting passengers while a passenger waits at Civic Center	The dashboard listed one passenger waiting at Civic Center	Figure 6.3.13
DM9	Refuse location permission, deactivate the bus, reset its operational state, stop tracking, reload the page while signed in, and log out	Not captured in a screenshot	No evidence in this report

These checks used a test device and a test bus, not an operating bus. They show that the flows can be followed on the deployed system. They do not measure the accuracy of the reported position or the behaviour in a moving vehicle.

6.4 Experimental Results / Performance Results

No performance experiment was carried out. There is no response-time benchmark, concurrency test, update-rate measurement or load test, so the proposal's planned targets (location updates within 3 seconds, ten or more buses, page loads under 4 seconds) are neither met nor shown to fail. The code review does show that one GPS update can call OSRM up to twice, once for the departure check and once for the status, and that the OSRM request has no timeout. These are structural observations that bear on performance, not measurements.

6.5 User Evaluation

No user evaluation was conducted. The proposal planned user acceptance testing with five or more users and a System Usability Scale score of at least 70. Neither was done, and no usability claim is made, including for the driver dashboard, which was not evaluated with real bus drivers.

6.6 Analysis of Results

The integrated automated tests give moderate confidence in the ETA rules, the waiting-overview counts and the access rules as specified by the fixtures, and the manual checks of the deployed flows add some confidence that the server, the passenger interface, the driver dashboard and the deployment work together. The branch result adds evidence about the author's development branch but, until it is reconciled with the integrated suite, it does not add to the evidence about the deployed system. None of this shows that ETA values are accurate for real buses, because the routing results are mocked and no observed arrival times were collected. The limitations of the project itself, such as the absence of user, field and performance evidence, are given in Section 1.7.

Table 6.6.1 lists the defects and technical limitations found by reviewing the code of the integrated system, rechecked against the final version. The severity is an assessment made from the code review. Items L1 and L2 were fixed in the final stage and are covered by the access-control tests of Table 6.3.1; the other items are open. Items L3, L4, L5 and L9 concern the path that the driver interface uses: GPS values, trip start, OSRM calls and the status thresholds.

Table 6.6.1: Defects and technical limitations identified in the system

ID	Finding	Impact	Severity and status
L1	Earlier in the project the Bus, Route, Stop and Passenger viewsets had no permission class and the settings defined no default, so Django REST Framework allowed unrestricted access [14]	Records could be read and modified without signing in	High; resolved. The default is IsAuthenticated, the transport viewsets are read-only and the passenger endpoint is removed
L2	The expire endpoint was open to anyone, and the waiting-overview endpoint did not check the driver	Anyone could end any WAITING request or read the waiting counts of a bus	Medium; resolved. Expire is limited to administrators and the overview needs an assigned driver
L3	GPS latitude, longitude and speed are stored without validation of type or range	Invalid values can be stored; the ETA ignores invalid coordinates but other consumers would see them	Medium; open

ID	Finding	Impact	Severity and status
L4	A destination with no matching trip raises an unhandled error in the start-trip view	A server error instead of a clear client error	Low; open
L5	The OSRM request has no timeout; status and departure services catch only request exceptions; the bus-ETA endpoint does not catch routing errors; one GPS update can call OSRM twice	Slow or malformed routing replies can delay or break a request	Medium; open
L6	Only 136 representative trips (08:00:00 variants) are used; the time of day is not modelled; shape data is unused	The system models route and direction, not a timetable	Low; open
L7	The 35 automated tests do not cover the GTFS import commands, the status and next-stop thresholds, validation of GPS values or a real OSRM instance, which is mocked	Faults in those parts would not be detected automatically	Medium; open
L8	Only the current state of a bus is stored	No history for audits or analysis	Low; open
L9	The thresholds of 30 for distance and 20 for speed are fixed values; the server does not state the unit of the speed it receives (the driver dashboard converts it to km/h before sending)	Status and next-stop decisions are not calibrated or documented against field data	Medium; open
L10	Sign-in throttling allows five attempts per minute for each client address and covers passenger and driver sign-in only	Guessing and bulk account creation are slowed but not reliably prevented	Medium; open

Findings L1 and L2 were the most serious, because they gave anonymous users a second route to the data that driver authorisation protects. They were resolved in the final stage, and the default permission now makes every endpoint require sign-in unless it explicitly allows anonymous access. The main open security point is L10. The table describes the final version of the system; any later change is not reflected. Item L7 refers to the integrated suite only; the tests on the Driver-Interface branch could not be examined.

6.6.1 Limitations of the Passenger Interface

Table 6.6.2: Limitations of the passenger interface

ID	Finding	Impact	Severity and status
P1	No usability evaluation or user acceptance test was carried out	It is not known whether passengers can complete the flow unaided	Medium; open
P2	Buses are shown on the map but routes and stops are lists; route shapes are imported but unused	A passenger cannot see where a stop lies relative to a bus on the map	Low; open
P3	Data reaches the page by polling every 15 seconds	A position can be older than 15 seconds; no push updates	Low; open
P4	The passenger page has no automated tests	Faults in the page would not be detected automatically	Medium; open
P5	The map library is loaded from a content-delivery network and the tiles from OpenStreetMap	Without access to these services the map does not display	Low; open
P6	Only representative trips are used, so time of day is not modelled	The passenger chooses a route and destination, not a departure time	Low; open
P7	The accuracy of the displayed estimate has not been established	An estimate could differ from the real arrival	Medium; open
P8	A passenger may hold only one active request	A journey with a change of bus cannot be recorded in advance	Low; open (design rule)
P9	Small-screen layout, accessibility and browser compatibility were not evaluated	Use on other devices and by users with different needs is unverified	Medium; open

6.6.2 Limitations of the Driver Interface

Table 6.6.3 lists the limitations of the driver interface found by reviewing its code and the checks of the deployed flow. The assessment is that of the author.

Table 6.6.3: Limitations of the driver interface

ID	Finding	Impact	Severity and status
D1	No usability evaluation or user acceptance test was carried out with drivers	It is not known whether drivers can use the dashboard unaided, or while driving	Medium; open
D2	The dashboard has no automated tests; its flows were checked by hand	Faults in the page would not be detected automatically	Medium; open
D3	The position is whatever the device reports; nothing shows that the device is on the bus	A position could be sent from the wrong place	Medium; open
D4	GPS values are not validated by the server (L3)	A wrong value would be stored	Medium; open

ID	Finding	Impact	Severity and status
D5	Updates are sent only while the page is open and the browser delivers fixes; behaviour with the screen locked, the page in the background or the signal lost was not examined	The position held by the server could become stale	Medium; open
D6	A position is sent for every fix the browser delivers, with only one request outstanding; battery and data use were not measured	The cost to the driver's device is unknown	Low; open
D7	The status and next-stop thresholds are fixed and not calibrated (L9)	The status shown to the driver may not match the real position	Medium; open
D8	Assignments of drivers to buses are made only by an administrator in the administration site	A driver cannot be assigned or reassigned from the dashboard	Low; open (design)
D9	The driver flow was checked with a test device and a test bus, not an operating bus	Behaviour on a moving bus is unverified	Medium; open

6.7 Objective-by-Objective Evaluation

Table 6.7.1 evaluates the group objectives. The last column states how the driver interface and the author's driver-side work relate to each objective. The evaluation is of the integrated prototype, not of the author's component alone.

Table 6.7.1: Evaluation of the group objectives

Objective	Evidence	Status	Driver-side relevance
Group objective 1: Real-Time Bus Tracking from real GPS	The driver dashboard sends the device position to the authorised update endpoint, which stores bus state and status (Sections 4.8, 5.4.2, 5.4.4); Figures 6.3.5 to 6.3.8 show activation, trip start, the accepted GPS update and the resulting marker on the passenger map; no field trial	Implemented; not field-tested	Direct: the driver dashboard is the source of the real GPS input
Group objective 2: passenger interface to view buses and routes	Passenger page shows bus markers on a Leaflet map with registration, status and speed, refreshed every 15 seconds, and lists routes, destinations and stops (Section 4.8.2); Figures 6.3.8 to 6.3.11	Met at prototype level	Indirect: the markers depend on the driver's GPS updates

Objective	Evidence	Status	Driver-side relevance
Group objective 3: Arrival Time Prediction	ETA algorithm and automated tests of its rules (Sections 4.9.5, 6.3.1); Figure 6.3.12 shows an estimate displayed to a passenger; accuracy not evaluated	Implemented; accuracy not established	Indirect: the estimate uses the position and current stop that the driver's updates maintain
Group objective 4: operators monitor and manage the fleet	Django administration site (Figure 6.3.14); no separate custom dashboard	Met at prototype level	Indirect: bus, driver and assignment records are created by operators and are used by the driver dashboard
Group objective 5: drivers see passengers at pickup points	Waiting-overview endpoint returns counts per stop and the driver dashboard displays them (Section 4.8.1); endpoint covered by automated tests; Figure 6.3.13 shows the dashboard listing a waiting passenger	Met at prototype level	Direct: the display is part of the driver dashboard

Table 6.7.2: Evaluation of the members' individual objectives

Objective	Evidence	Status
Backend objective 1: data model and GTFS import	Models, migrations and five import commands (Sections 5.4.1, 5.4.2)	Achieved
Backend objective 2: driver authentication and assignment checks	Section 5.4.3; default permission, read-only model endpoints and administrator-only expiry (L1, L2 resolved)	Achieved; sign-in hardening gaps remain (L10)
Backend objective 3: bus operations and trip start	Section 5.4.4; error handling gap L4	Achieved
Backend objective 4: GPS handling, status and next stop	Section 5.4.5; validation gap L3 and calibration gap L9	Achieved at prototype level
Backend objective 5: waiting requests and ETA	Sections 5.4.6 to 5.4.8 and Table 6.3.1	Achieved
Backend objective 6: tests and deployment configuration	Table 6.3.1, Section 5.4.9; the tests do not cover the imports, the thresholds or GPS validation (L7)	Partly achieved (test coverage and deployment verification)
Driver objective 1: sign-in, session and assigned buses	Section 5.4.10; DM2, DM3 (Table 6.3.5); session restore and log-out not recorded	Achieved; partly evidenced
Driver objective 2: bus operations and trip start	Section 5.4.10; DM4, DM8; deactivation and reset not recorded	Achieved; partly evidenced
Driver objective 3: device GPS reporting and display	Section 5.4.10; DM5 to DM7	Achieved at prototype level; position accuracy not established
Driver objective 4: waiting counts by stop	Section 5.4.10; DM8; waiting-overview tests (Table 6.3.1, rows 7 to 9)	Achieved at prototype level

Objective	Evidence	Status
Driver objective 5: trip-lifecycle and stop-progression work	Sections 4.9.3 and 5.4.5; the branch test run (Section 6.3.2); the status and next-stop thresholds have no test in the integrated suite (L7)	Implemented; not tested in the integrated suite
Driver objective 6: verify the driver flows on the deployed system	Table 6.3.5; no automated interface tests, no user study	Partly achieved
Passenger objective 1: live map with refresh	Section 5.4.11.2; M3	Achieved at prototype level
Passenger objective 2: register, sign in and sign out	Section 5.4.11.3; M2; registration and the password-reset link not recorded (M11)	Partly evidenced
Passenger objective 3: route search, destination and stops	Sections 5.4.11.4, 5.4.11.5; M4 to M6	Achieved
Passenger objective 4: boarding stop and waiting request	Section 5.4.11.6; M6, M7; one-active-request rule enforced by the server	Achieved
Passenger objective 5: view and manage own requests	Section 5.4.11.6; M9; completed state not recorded	Achieved; partly evidenced
Passenger objective 6: show the arrival estimate	Section 5.4.11.7; M8	Achieved; accuracy not established
Passenger objective 7: verify the flows on the deployed system	Table 6.3.3; no automated interface tests, no user study	Partly achieved

6.8 Individual Results

The results attributable to the author are the driver interface and the driver-side functionality described in Section 5.7. Their verified evidence consists of the following, kept at their separate levels.

- **Manual checks of the deployed driver flow** (Figures 6.3.3 to 6.3.7 and the driver part of Figure 6.3.13): sign-in, listing of the assigned bus, activation, trip start, the permission request, an accepted GPS update with the derived status IN_TRANSIT, and the waiting passenger shown at Civic Center.
- **The integrated automated test of driver authorisation** (row 13 of Table 6.3.1), which confirms that an assigned driver can update GPS and operate the bus and that other accounts are refused. This test belongs to the integrated suite and its authorship is credited to Kenneth Kaunda Musonda in the group report.
- **The Driver-Interface branch test run** of 94 passing tests (Section 6.3.2), whose composition and authorship are to be confirmed.
- **The source code** of the integration branch, which contains the driver dashboard, and the Driver-Interface branch (commit c816533, as recorded by the author), whose individual files are not itemised in this report.

No individual result is presented as a performance, usability or accuracy outcome, and the driver interface was not tested with real drivers or in an operating bus.

6.9 Chapter Summary

The available evidence supports the correctness of the server's ETA rules, waiting counts and access rules under the tested conditions, shows by manual checks that the passenger interface, driver dashboard and server work together on the deployed system, and records a passing test run of 94 tests on the author's development branch whose relation to the integrated suite of 35 tests is to be confirmed. It does not support claims about accuracy, performance, usability or field operation. The main limitations of the project are the absence of user, field and performance evidence and the open technical items in Table 6.6.1, and they are carried into the discussion.

CHAPTER 7: DISCUSSION

7.1 Introduction

This chapter interprets the findings in relation to the problem, the objectives, the literature and the limits of the evidence.

7.2 Interpretation of Findings

The results of Chapter 6 are interpreted below in relation to the problem stated in Section 1.2 and the objectives of Section 1.4, within the prototype setting. Table 6.7.1 gives the evidence for each objective.

Real-Time Bus Tracking from real GPS updates (group objective 1). A driver's device sends latitude, longitude and speed to an authenticated endpoint through the driver dashboard. The helper get_authorized_bus_for_driver accepts the update only from an active driver with an active assignment to the bus. The system stores the values, updates the next stop and derives the status, so each bus carries a current trip, current stop, position, speed and status. The method works for the implemented design, and the manual checks show the whole path from the driver's device to the marker on the passenger map. The thresholds used have not been calibrated against field data, and the values are not validated at entry. The finding is that the driver's own device can be a workable position source for a prototype, which supports the implementation gap identified in Section 2.5. It is not evidence that a driver's phone is an adequate position source for an operating bus, because no operating bus was used.

Journey planning with GTFS data (group objective 2). The passenger interface offers this flow on top of the imported data. Imported routes, trips, stops and ordered stop times let a passenger search a route, choose a destination and see the stops of a trip in order, then register at a boarding stop. Using representative trips makes selection unambiguous. It does so by treating a route and a direction as the unit of planning, not a timetable. The driver uses the same route and destination data to start a trip, which keeps the driver's choice and the passenger's choice on the same trip.

Arrival time from bus state and road durations (group objective 3). The ETA is the shortest road duration, from OSRM, among buses that are active, on the same trip, with valid coordinates and not yet past the stop. The system returns null when no such bus exists or when routing fails, instead of an invented value. Whether the numbers match real arrival times is not known. The ETA is only as good as the driver's position updates and the stop progression, which makes the driver-side logic a dependency of the prediction feature.

Waiting passengers and fleet management (group objectives 4 and 5). The driver dashboard shows the number of waiting passengers at each stop of the trip, taken from the waiting-overview endpoint, which only the assigned driver may call. Operators manage buses, drivers, assignments and transport records through the Django administration site. No separate operator dashboard was built.

Protecting operations and data. Driver operations are protected by the assignment check, and the passenger's waiting-request endpoints are scoped to the owner. In the final branch the default permission requires sign-in, the transport viewsets are read-only, the passenger directory is removed and expiry is limited to administrators. Sign-in is rate-limited, but the limit has weaknesses (L10), and no independent security review was carried out. The protection is therefore in place for the implemented controls, not as a security guarantee.

The development process. The sequence of an initial backend skeleton, parallel interface development with backend updates, and later integration allowed the two interfaces to be built at the same time. It also meant that the backend changed while the interfaces depended on it, and that the branches had to be reconciled at integration. The difference between the test counts of the Driver-Interface branch and the integrated suite is a visible result of this process and should be understood as such, not as a discrepancy in the system.

7.2.1 Interpretation of the Findings for the Passenger Interface

Real-Time Bus Tracking (group objective 1). The passenger map displays the buses that the server holds, each with its registration number, status and speed. The checks show that a bus reporting from a driver's device appears on the map at the reported position. The method works for the implemented design, but the display is only as current as the last 15-second refresh and the driver's last update, and the thresholds behind the status have not been calibrated against field data.

Journey planning with GTFS data (group objective 2). The passenger interface offers the flow on top of the imported data. Imported routes, trips and ordered stops let a passenger search a route, choose a destination and see the stops of a trip in order, then register at a boarding stop. Using representative trips makes the selection unambiguous. It does so by treating a route and a direction as the unit of planning, not a timetable, which suits a first prototype and limits what a passenger can plan.

Arrival time (group objective 3). The interface shows the estimate that the server calculates as the shortest road duration from OSRM among eligible buses, and nothing when none is valid. This keeps the interface honest, in line with the caution of Liu and Miller that real-time information does not automatically perform well [5]. Whether the numbers match real arrivals is not known.

Waiting passengers (group objective 5). A request created on the passenger page appears as a count on the driver dashboard. The passenger interface is therefore the source of the

demand information that the driver sees, which is the part of the project in which passengers and drivers interact.

The interface as the point of use. The literature treats the effect of real-time information as dependent on how it is delivered [5]. The design choices of the passenger interface are attempts to deliver it well: the map as the permanent base screen, a short ordered sequence from route to request, confirmation messages at the key steps and controls that follow the status of the request. This report cannot say whether those choices work for passengers, because no usability study was done.

7.3 Comparison with Existing Work

The system follows the conceptual flow found in the literature: positions are sent to a central server, combined with route and stop information, and turned into passenger information [2]. Like OneBusAway it presents next-bus style information, but it is far smaller; it has one passenger-facing flow, no multichannel delivery and no study of its effect on waiting time [3]. The system resembles the university system of Khalid et al. in taking positions from a device carried on the bus, although it applies to many public routes and adds waiting requests and driver authorisation [2]. The caution of Liu and Miller, that real-time information does not automatically reduce waiting time, applies equally: this project shows that the information can be produced, not that it improves passengers' waiting [5]. In relation to Rea Vaya, the project is a lightweight prototype with none of the infrastructure of a BRT system [9]. In relation to GTFS, the project adopts the static data model for routes, trips and stops and leaves GTFS Realtime for the future [7], [6].

7.3.1 Comparison with Existing Work: Passenger Interface

The system follows the conceptual flow found in the literature: positions are sent to a central server, combined with route and stop information, and turned into passenger information [2]. Like OneBusAway it presents next-bus style information to passengers through a web interface, but it is far smaller; it has one passenger-facing channel, no multichannel delivery and no study of its effect on waiting time [3]. Unlike a ride-hailing application such as Yango, which was used as an interaction reference, the passenger does not request a vehicle for a private trip but registers at a stop on a fixed route. In relation to GTFS, the project adopts the static data model for routes, trips and stops and leaves GTFS Realtime for the future [7], [6].

7.4 Contributions of the Project

The group produced an integrated prototype for Lusaka routes: a passenger interface with a live map, journey planning and waiting requests, a driver dashboard with GPS reporting and passenger counts, and a server with the data, ETA logic, access control and deployment configuration behind them. The members' contributions are distinct. Kenneth Kaunda

Musonda built the initial backend foundation and later integrated the branches, hardened access control and deployed the system. Ngosa Muleba developed the passenger interface. The author developed the driver interface and the driver-side functionality, which is the component that introduces real GPS data into the system, together with the related trip-lifecycle and stop-progression work on the Driver-Interface branch; the backend was updated alongside the interfaces and that work is shared. The group's documentation, including the conceptual and design diagrams drawn to match the code and an explicit list of limitations, provides a verified base for later work.

7.4.1 Contribution of the Passenger Interface

The group produced an integrated prototype for Lusaka routes: a passenger interface with a live map, journey planning and waiting requests, a driver dashboard with GPS reporting and passenger counts, and a server with the data, ETA logic, access control and deployment configuration behind them. Ngosa Muleba's own contribution, set out in Section 5.7 and Appendix A, is the passenger interface and the analysis it rests on: the live map with refresh, the route search and destination choice, the boarding stop selection, the creation and management of waiting requests, the display of the arrival estimate, and the manual verification of these flows on the deployed system. A further contribution is the documentation in this report, including the user guide in Appendix B and the explicit list of limitations.

7.5 Limitations and Implications

The limitations in Section 1.7 and Table 6.6.1 determine how the results may be used. The principal limitation was the absence of an operational bus fleet and an official operator data feed, so the GPS positions could not be verified as those of an operating bus and live service changes could not be received. In short, the system could be verified against its design but not validated against an operating bus service. The findings apply to a prototype verified functionally, with synthetic fixtures and mocked routing in the automated tests. They do not apply to a city-wide, continuously operating service. Without user, field and performance evidence, no claim can be made about ETA accuracy, usability or the effect on passengers, and the thresholds and the ETA method must be evaluated with observed data first. For the driver interface in particular, battery use, loss of signal, the browser being backgrounded on a phone and the driver's attention while driving were not examined. The GTFS feed supplied is a snapshot with a stated validity period, and keeping data current is an operational task that the project did not address. The browser interfaces were verified only by manual checks, not by automated tests. Technically, the open items in Table 6.6.1 and an independent security review should be dealt with before real users' data are handled.

7.5.1 Limitations and Implications for the Passenger Interface

The limitations in Section 1.7 and Table 6.6.2 determine how the results may be used. The principal limitation was the absence of an operational bus fleet and an official operator data feed, so the positions on the map could not be verified as those of an operating bus and live service changes could not be received. The passenger interface was verified against its design but not validated against an operating bus service. The findings apply to a prototype checked functionally on one test device. They do not apply to a city-wide, continuously operating service, and without user, field and performance evidence no claim can be made about usability, ETA accuracy or the effect on passengers. The passenger interface was verified only by manual checks, not by automated tests. The GTFS feed is a snapshot with a stated validity period, and keeping data current is an operational task that the project did not address.

7.6 Chapter Summary

The objectives were met at prototype level, with the protection of operations and data only partly satisfied. The results are consistent with the literature’s view that information can be provided but its benefit must be shown empirically.

CHAPTER 8: CONCLUSION AND RECOMMENDATIONS

8.1 Introduction

This chapter concludes the report, states whether the objectives were achieved and gives recommendations and future work.

8.2 Summary of the Project

The project set out to implement a web-based real-time public bus tracking system for Lusaka. It started from a proposal that simulated GPS data. On supervisor feedback, it was changed to accept real GPS coordinates and speed from a driver's device. The group followed an iterative and incremental process: an initial backend skeleton was built first, the driver and passenger interfaces were then developed with backend updates alongside them, and the branches were integrated and deployed. The group produced a passenger interface with a live map, journey planning and waiting requests; a driver dashboard that reports the device's GPS position and shows waiting passengers; and a Django and Django REST Framework server with the data model, GTFS imports, driver authorisation, real GPS handling, waiting requests, OSRM integration and ETA logic, configured for deployment. The system was verified by automated tests of the server's ETA, waiting-overview and access-control behaviour, by a passing test run on the author's development branch, by manual checks of the deployed interfaces, and by code review. The author's own part is stated in Section 5.7 and Appendix A.

8.3 Conclusions

- A Django application can maintain the operational state of buses from authorised real GPS updates sent by a driver's device and relate it to imported GTFS routes, trips and stops (group objectives 1 and 2).
- A conditional, road-based ETA can be produced from this state and an external routing service, and it can decline to estimate when the data is missing or invalid (group objective 3).
- Driver operations are protected by an active driver and an active assignment, and the final branch requires sign-in by default, so protection is implemented at prototype level but has not been independently tested.
- The three intended features, Real-Time Bus Tracking, Journey Planning and Arrival Time Prediction, are implemented at prototype level across the passenger interface, driver dashboard and server, with bus locations viewed as map markers and routes as lists.
- The project does not show city-wide operation, ETA accuracy, usability, user acceptance or performance.

8.3.1 Conclusions on the Passenger Interface

- A browser page can turn the state held by the server into usable passenger information: active buses on a map with their status and speed, a route-and-stop selection sequence built on GTFS data, and waiting requests that the driver can see (group objectives 1, 2 and 5).
- The arrival estimate calculated by the server can be presented to the passenger with a waiting request, and the interface shows it only when it exists (group objective 3).
- The three intended features, Real-Time Bus Tracking, Journey Planning and Arrival Time Prediction, are implemented at prototype level and reach the passenger through the passenger interface.
- The project does not show city-wide operation, ETA accuracy, usability, user acceptance or performance.

8.4 Achievement of Objectives

The system objectives were met at prototype level for real-GPS tracking, arrival-time prediction and waiting counts for drivers. The passenger interface objective was met, with bus locations viewed as map markers and routes as lists. The operator objective was met through the administration site. Table 8.4.1 summarises the status of each group objective against the evidence.

Table 8.4.1: Achievement of the group objectives

Objective	Status	Evidence
1. Real-Time Bus Tracking from real GPS	Implemented; not field-tested	Driver dashboard, update endpoint, manual checks (Section 6.3.3)
2. Passenger interface to view buses and routes	Met at prototype level	Section 4.8.2; Figures 6.3.8 to 6.3.11
3. Arrival Time Prediction	Implemented; accuracy not established	Section 4.9.5; Table 6.3.1 rows 1 to 6
4. Operators manage the fleet	Met at prototype level	Django administration site
5. Drivers see waiting passengers	Met at prototype level	Section 4.8.1; Table 6.3.1 rows 7 to 9; Figure 6.3.13

The author’s individual contribution to these results is the driver interface and driver-side functionality (Section 5.7). Its evaluation is limited to manual checks of the deployed driver flow and the branch test run, and it is not presented as evidence of usability or performance.

8.5 Recommendations

1. Resolve the open security item (L10): state the number of trusted proxies, keep throttle counters in a shared cache, and throttle registration and stop revealing which usernames exist; then arrange an independent security review.
2. Validate GPS input for type and range, handle the start-trip error cleanly, and set timeouts and error handling for every OSRM call (L3 to L5).
3. Extend automated tests to the GTFS imports, the status and next-stop thresholds and GPS value validation, and add an integration test against a real OSRM instance (L7).
4. Document the unit of the speed value and calibrate the status thresholds with field data (L9).
5. Before any operational use, establish privacy, consent and data-governance policies, verify the licence of the GTFS data, and add audit logging.
6. Carry out a usability evaluation with passengers, using task-based tests and the System Usability Scale planned in the proposal, before any operational use (P1), and a corresponding evaluation of the driver dashboard with drivers, including use in a moving vehicle (D1).
7. Add automated tests of the passenger page, covering the route-to-request sequence, the controls shown for each status and the handling of a missing estimate (P4), and of the driver dashboard (D2).
8. Evaluate both interfaces on a range of phones and browsers, and for accessibility (P9), and examine how the driver dashboard behaves with the screen locked, in the background or without signal (D5).
9. Collect observed arrival times and compare them with the displayed estimates before the estimate is relied on (P7).
10. Record the checks that were not captured in this report, namely registration, password reset, request completion and a missing estimate on the passenger side, and refused permission, deactivation, reset, stop of tracking, reload and log-out on the driver side (Tables 6.3.3 and 6.3.5).

8.6 Future Work

Future work may include: a historical location table for analysis; time-of-day schedules using all GTFS trips; use of the shape data for map-matching and for drawing the route and its stops on the map (P2); live-traffic-aware ETA; publication of a GTFS Realtime feed; delivering updates by push instead of polling (P3); self-hosting the map library and handling the loss of tile services (P5); notifications when a bus is near the boarding stop; use of the passenger's own position to suggest the nearest stop; support for journeys with more than one bus (P8); validation that a reporting device is on the bus it claims (D3); user research and usability evaluation with passengers and drivers; field trials with measured ETA accuracy;

performance and load testing; and an operator dashboard. These are possible extensions and are not features already delivered.

REFERENCES

[1] A. Fernando, K. Francisco, and M. Guillermo, “Intelligent transportation systems for sustainable urban mobility: A systematic literature review of research and applications in public transport,” *Urban Sci.*, vol. 10, no. 7, p. 380, 2026.

[2] S. K. A. Khalid, N. S. Mohd Salleh, and N. A. Samsudin, “A bus tracking information system using consumer grade GPS: A case study,” *J. Telecommun. Electron. Comput. Eng.*, vol. 8, no. 4, pp. 47–51, 2016.

[3] K. E. Watkins, B. Ferris, A. Borning, G. S. Rutherford, and D. Layton, “Where is my bus? Impact of mobile real-time information on the perceived and actual wait time of transit riders,” *Transp. Res. A, Policy Pract.*, vol. 45, no. 8, pp. 839–848, 2011.

[4] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2005.

[5] L. Liu and H. J. Miller, “Does real-time transit information reduce waiting time? An empirical analysis,” *Transp. Res. A, Policy Pract.*, vol. 141, pp. 167–179, 2020.

[6] GTFS, “GTFS Realtime,” General Transit Feed Specification Documentation, gtfs.org. Accessed: Jun. 16, 2026. [Online]. Available: https://gtfs.org/documentation/realtime/

[7] GTFS, “General Transit Feed Specification overview,” General Transit Feed Specification Documentation, gtfs.org. Accessed: Jun. 16, 2026. [Online]. Available: https://gtfs.org/documentation/overview/

[8] Project OSRM, “OSRM API documentation,” version 5.24.0. [Online]. Available: https://project-osrm.org/docs/v5.24.0/api/

[9] E. Adewumi and D. Allopi, “Rea Vaya: South Africa’s first bus rapid transit system,” *S. Afr. J. Sci.*, vol. 109, nos. 7/8, 2013.

[10] GSMA Intelligence, “The mobile economy Sub-Saharan Africa 2023,” GSMA, London, U.K., 2023.

[11] DigitalTransport4Africa, “Lusaka city public transport GTFS feed (feed version lusaka_city_pt, valid 2023-09-01 to 2024-12-30).” Accessed: Jun. 16, 2026. [Online]. Available: https://digitaltransport4africa.org/

[12] Django Software Foundation, “Django documentation, version 6.0.” [Online]. Available: https://docs.djangoproject.com/en/6.0/

[13] Django Software Foundation, “Writing and running tests,” Django documentation, version 6.0. [Online]. Available: https://docs.djangoproject.com/en/6.0/topics/testing/overview/

[14] Django REST framework, “Permissions,” API Guide. [Online]. Available: https://www.django-rest-framework.org/api-guide/permissions/

Reference check (delete before submission): the references are carried over from the group report. Reference [1] was confirmed by its DOI (10.3390/urbansci10070380) and author list. References [2], [3], [5], [9], [10] and [11] were not independently checked in this drafting session and must be confirmed against the publishers’ records before submission; do not add further sources unless they have been read.

APPENDICES

APPENDIX A: INDIVIDUAL CONTRIBUTION STATEMENT (GROUP PROJECTS)

Purpose of this Statement

This statement documents the individual contribution of Madalitso Nyirenda to the group project. Although the project was conceived, analysed and integrated collaboratively, this report presents individual work, and the statement is consistent with Chapters 5 and 6. The work of other group members is identified as theirs and is not claimed.

Project and Group Information

Table A.1: Project and group information

Item	Detail
Student Name	Madalitso Nyirenda
Computer Number	2021502597
Programme	Bachelor of Computer Science (Software Engineering), University of Zambia
Supervisor	Dr. Lighton Phiri
Group Size	3 students
Group Members	Madalitso Nyirenda (2021502597); Kenneth Kaunda Musonda (2020056283); Ngosa Muleba (2022044371)

Overall Group Project

The group built the Bus Tracking System, a web-based system that lets passengers in Lusaka follow buses, plan a journey on predefined routes and obtain an arrival-time estimate, and lets authorised drivers operate their buses and report GPS positions. The common problem is the lack of route-specific, real-time information for passengers. The group's solution is an integrated web application made of a Django backend, GTFS-derived route data, an OSRM routing service and browser interfaces. The proposal planned simulated GPS data; after supervisor feedback the system accepts real GPS updates from a driver's device.

Development History

The development proceeded in the following order. Kenneth Kaunda Musonda first built the initial backend skeleton. Ngosa Muleba and the author then developed their respective interfaces, the passenger interface and the driver interface, and the backend was updated to support the requirements that emerged from this work, so that not every backend change belongs to a single member. Kenneth Kaunda Musonda then integrated the branches and deployed the complete system. The remaining project activities were carried out collaboratively.

Individual Role and Responsibilities

The author's role was the driver interface and driver-side functionality. The author was also jointly responsible, with the other members, for the problem analysis and requirements. The author was not responsible for the initial backend skeleton, the integration of the branches, the deployment, or the passenger interface.

Detailed Individual Contribution

The following are the author's contributions, attributed on the basis described in Section 5.7.1.

- The driver dashboard: driver sign-in, listing and selection of the assigned bus, activation, deactivation and reset of the bus, route and destination selection, and trip start (Section 5.4.2).
- Device GPS tracking: use of the browser Geolocation API to send latitude, longitude and speed to the bus location endpoint, and display of the accepted position, speed and status, with the driver map (Section 5.4.2).
- Display of the number of passengers waiting at each stop of the driver's current trip (Section 5.4.2).
- Work on the trip lifecycle and on the location and stop-progression logic, with the related server-side changes, tests and supporting files, on the Driver-Interface branch, commit c816533 "Complete driver interface and trip lifecycle".
- Participation in the shared problem analysis, requirements and Yango interface analysis (Section 3.3).
- Preparation of this individual report from the group's source material, with the individual contribution statements in Chapters 5 and 6 and in this appendix.

Individual Deliverables

1. Driver interface: the driver dashboard template, scripts and styles in the Driver-Interface branch (individual files are not itemised in the available records).
2. Trip-lifecycle and stop-progression changes, and related server-side changes, in the Driver-Interface branch (individual files and functions are not itemised in the available records).
3. Tests added on the Driver-Interface branch (the number of tests written by the author was not established).
4. This individual report.

Contribution to the Main System or Research Output

The driver interface is the component through which real GPS position, speed and bus operations enter the integrated system. It supplies the data that the passenger map displays,

that the stop-progression and status logic process, and from which the arrival-time estimate is calculated, and it shows drivers the waiting passengers that the passenger interface records. The author's component works with the backend developed from Kenneth Kaunda Musonda's initial skeleton and with the passenger interface developed by Ngosa Muleba.

Individual Implementation and Technical Work

The technical work was a browser-based driver dashboard written as a Django template with HTML, CSS and JavaScript, using the Fetch API with CSRF protection, the browser Geolocation API and a Leaflet map; the use of the driver endpoints listed in Table 4.5.3.1; and trip-lifecycle and stop-progression changes on the server in the Driver-Interface branch.

Individual Testing and Evaluation Contribution

The author ran the test suite on the Driver-Interface branch, which reported 94 passing tests. This count is not presented as the number of tests written by the author, as a result for the integrated system or as a result for the deployed application. It is reported separately from the 35 tests of the integrated system (Section 6.3). The author took part in the manual checks of the deployed driver flow (Figures 6.3.3 to 6.3.7). No user testing, field trial or performance measurement was conducted, and none is claimed.

Collaboration with Other Group Members

The backend skeleton, the integration of the branches and the deployment of the system were carried out by Kenneth Kaunda Musonda. The passenger interface was developed mainly by Ngosa Muleba. Backend changes made while the interfaces were being developed were shared, and where the Git evidence does not attribute a change to one member it is treated as collaborative. The problem analysis, the Yango interface analysis, the requirements and the literature review were shared. The supervisor's review feedback led to the replacement of simulated GPS with real GPS and is acknowledged.

Contribution Summary

(1) Work primarily completed individually by the author: the driver dashboard and its device GPS tracking, and, according to the author's account, the driver-side trip-lifecycle and stop-progression work on the Driver-Interface branch (Section 5.7.1).

(2) Work completed jointly with other group members: the problem analysis, requirements analysis, Yango interface analysis, the literature review, the backend updates made while the interfaces were developed, the manual checks of the deployed system, and the group presentation.

(3) Work primarily completed by other group members but used in or alongside the individual work: the initial backend skeleton, the integration of the branches, the security hardening and the deployment (Kenneth Kaunda Musonda); and the passenger interface (Ngosa Muleba).

Estimated Contribution

The template asks for approximate percentages. No time or effort records are available from which percentages could be derived without inventing them, so Table A.2 states the nature of the contribution to each activity.

Table A.2: Nature of the individual contribution by activity

Activity	Individual	Collaborative
Problem Analysis	Contributed	Shared with the group
Literature Review	Reused and adapted for this report	Prepared jointly for the group report
Requirements Analysis	Contributed; driver-side requirements	Yango analysis shared
System/Research Design	Driver interface design	Backend and passenger interface design by other members; group diagrams reused
Implementation/ Development	Driver interface; driver-side trip lifecycle and stop progression	Backend updates alongside the interfaces; initial skeleton, integration and deployment by Kenneth Kaunda Musonda; passenger interface by Ngosa Muleba
Testing and Evaluation	Branch test run; manual checks of the driver flow	Integrated automated tests and manual checks shared
Documentation	This individual report	Group report and presentation prepared by the group

Evidence and Limits of the Attribution

The attribution in this statement rests on the group report and presentation, the individual report on the passenger interface, the author's account of the Driver-Interface development branch and the integrated code. Because the integrating member merged the interface branches, the history of the integration repository lists him against most commits, and it does not by itself itemise individual authorship; see Section 5.7.1.

Statement of Authorship and Contribution

I confirm that the contribution described in this statement accurately represents the work that I personally undertook as part of the group project. I have identified collaborative work and have not knowingly claimed the work of another group member as my own.

Student Name: Madalitso Nyirenda

Computer Number: 2021502597

Signature: _____

Date: _____

Supervisor Verification

I confirm that, to the best of my knowledge, the contribution described above is consistent with the student's participation in the group project.

Supervisor Name: Dr. Lighton Phiri

Signature: _____

Date: _____

APPENDIX B: USER MANUAL / INSTALLATION GUIDE

Running the system and loading the GTFS data

The commands below are the Django management commands defined in the project. They are run from the directory that contains manage.py, after the dependencies in requirements.txt are installed.

```
python manage.py migrate
python manage.py import_routes
python manage.py import_stops
python manage.py import_trips
python manage.py import_stop_times
python manage.py import_shapes
python manage.py runserver
python manage.py check     # configuration check only
python manage.py test      # runs the automated tests
```

Each import command prints the numbers of rows imported, updated and skipped. Routes and stops must be imported before trips, and trips before stop times. In production the settings expect the environment variables listed in Table F.2 in Appendix F.

Using the driver dashboard

1. Open the landing page and choose Driver Login. On the Driver Dashboard, enter the username and password and press Sign in. The account must be an active driver account assigned to a bus (accounts and assignments are created by an administrator in the Django administration site).
2. Under Assigned bus, choose one of your assigned buses. Its status is IDLE until a trip starts.
3. Press Activate. Under Start a trip, choose a Route and a Destination and press Start trip. Deactivate and Reset operational state are available for the selected bus.
4. Under GPS tracking, press Start GPS tracking and allow the browser to read the device's location when asked. The dashboard shows that the update was accepted, together with the location, speed and status, and shows the position on the Live bus location map.
5. Read the passenger count for each stop under Waiting requests by stop. Press Stop GPS tracking when the trip ends, and Log out when finished.

Using the passenger interface

This guide describes how a passenger uses the passenger page. It describes the controls of the deployed version shown in the figures of this report.

Opening the page

- Open the address of the deployed system in a web browser. On the landing page choose the passenger option (Figure 6.3.2).
- The map appears with the header bar *Lusaka Bus Tracker*. The header has four controls: Active Buses, Plan Trip, My Requests and a refresh control.

Watching the buses

- Active buses appear as bus markers. Touch or click a marker to see its registration number, status and speed.
- The map refreshes by itself every 15 seconds. Press the refresh control, in the header or at the lower right of the map, to refresh at once.
- If no marker appears, no bus is active: a bus appears only after its driver has activated it, started a trip and sent a position.

Registering and signing in

- Press Plan Trip. In the account area enter your details and press Register to create an account, or Login if you already have one.
- If you forget your password, press *Forgot password?* and follow the instructions sent to your e-mail address.
- Press Logout when you have finished.

Making a waiting request

- In Plan Trip, type part of a route name in *Search route* and choose the route from the list.
- Under *Trip and stop details* touch the destination you are travelling towards.
- From the list of stops, touch the stop where you will board. A message confirms *Selected boarding stop*.
- Press *Create Waiting Request*. A message confirms the request. You may have only one active request at a time.

Managing your requests

- Press My Requests to see your requests. Each shows the route, the boarding stop, the time and the status.
- While the status is WAITING you can see the estimated arrival, when available. Press *Mark as Boarded* when you board, or *Cancel Trip* if you no longer need the bus.
- When you are on board (ON_BOARD), use the control to complete the trip when you arrive.

- The estimate appears only when a bus on your trip is active, has a valid position and has not yet passed your stop. When no estimate is shown, none could be calculated.

APPENDIX C: ADDITIONAL SYSTEM DESIGNS / DIAGRAMS

Figure C.1 shows the entity-relationship diagram of the persisted data. The driver interface reads and writes the Driver, DriverBusAssignment, Bus and WaitingRequest entities, and reads Route, Trip, StopTime and Stop (Table 4.7.4.1); the passenger interface reads and writes the Passenger, WaitingRequest, Route, Trip, StopTime, Stop and Bus entities (Table 4.7.3.1).

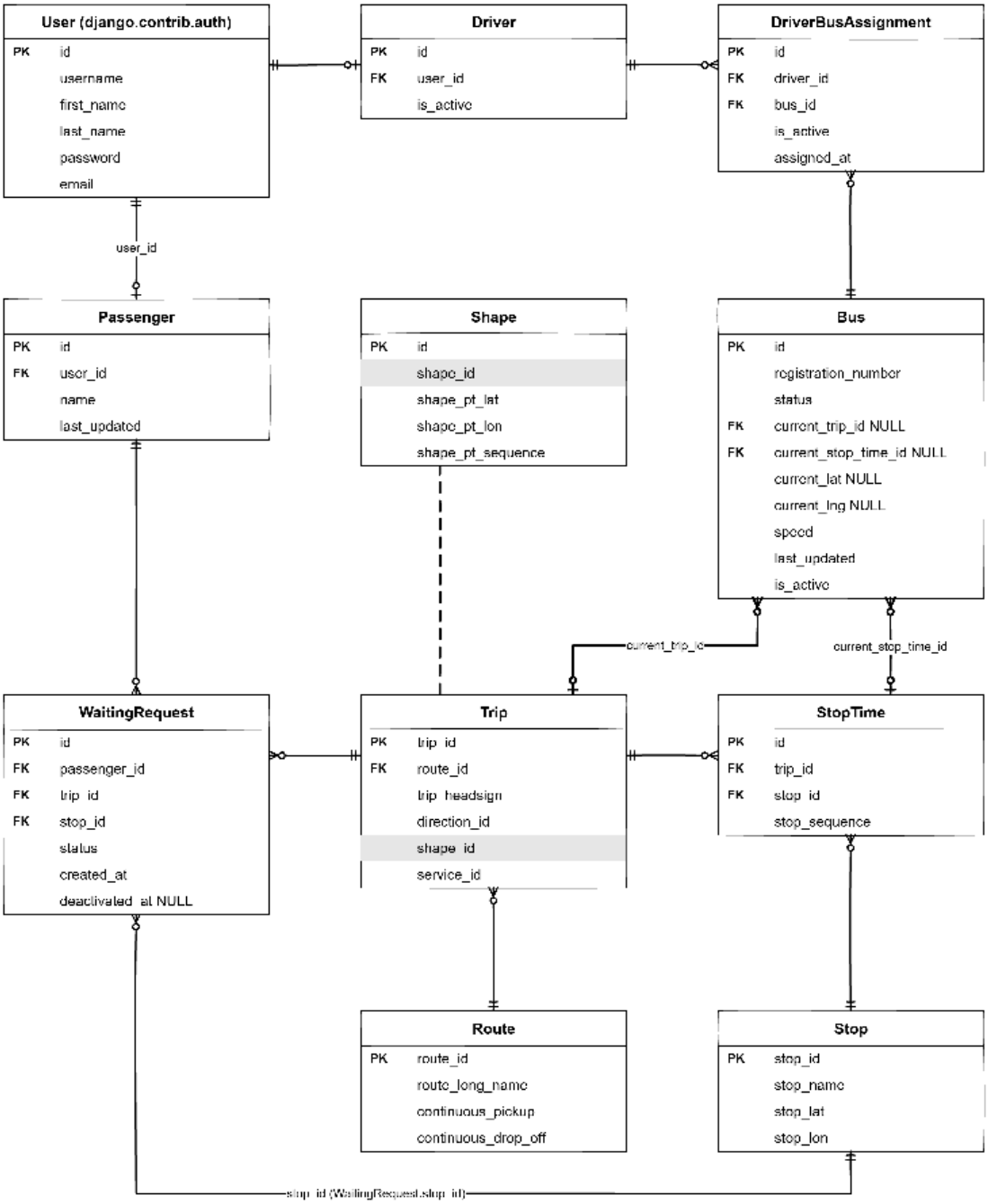


Figure C.1: Entity-relationship diagram of the persisted data

APPENDIX D: ADDITIONAL TEST CASES AND RESULTS

The tests are in tracking/tests.py. The names below are those of the test methods in the final version of the system, which defines 35 tests in five classes. A run of python manage.py test found 35 tests and all passed (Section 6.3, Figure 6.3.1). The classes are explained first and then listed in Table D.1.

WaitingRequestEtaTests (6 tests) check the arrival-time estimate for a waiting request. They verify that the shortest valid OSRM duration is used when several suitable buses are available, and that no estimate is produced when a bus has no GPS coordinates, when a bus has already passed the requested stop, when the request is not in the WAITING state, or when OSRM returns no usable route. They also verify that a passenger cannot read the ETA of another passenger's request.

WaitingOverviewTests (3 tests) check the waiting-passenger counts shown to a driver. They verify that a request is counted only at its own boarding stop, that requests which are not waiting or belong to another trip are excluded, and that a bus without an active trip returns an empty list of stops. The test data uses a stop named Bauleni station as its example.

ApiAccessSecurityTests (20 tests) check who may call which endpoint. They verify that routes, stops and buses are read-only and public, that the passenger directory no longer exists, that driver operations need an active driver with an active assignment, that only an administrator can expire a request, that registration applies password validators, and that a passenger can neither change a request's lifecycle fields directly nor act on another passenger's request.

LoginThrottleTests (4 tests) check that passenger and driver sign-in are limited to five attempts per minute for each client address, with separate counters, and that registration is not throttled. DebugDefaultTests (2 tests) check that debug mode is off unless it is explicitly enabled.

Table D.1: Test methods

Test class	Test method	What it verifies
WaitingRequestEtaTests	test_returns_smallest_osrm_duration_for_eligible_buses	With several eligible buses, the ETA is the shortest valid OSRM duration.
WaitingRequestEtaTests	test_returns_unavailable_when_no_bus_has_gps	No ETA is returned when no bus has GPS coordinates.
WaitingRequestEtaTests	test_excludes_bus_that_has_passed_the_requested_stop	A bus that has already passed the boarding stop is not used.
WaitingRequestEtaTests	test_non_waiting_requests_have_no_eta	Only a request in the WAITING state receives an ETA.

Test class	Test method	What it verifies
WaitingRequestEtaTests	test_another_passenger_cannot_read_the_request_eta	A different passenger asking for the ETA is refused.
WaitingRequestEtaTests	test_missing_osrm_route_returns_unavailable	No ETA is returned when OSRM gives no usable route.
WaitingOverviewTests	test_counts_bauleni_boarding_request_only_at_bauleni	A waiting request is counted only at its own boarding stop.
WaitingOverviewTests	test_excludes_non_waiting_requests_and_other_trips	Requests that are not waiting, or that belong to another trip, are not counted.
WaitingOverviewTests	test_no_active_trip_returns_no_stops	A bus with no active trip returns an empty list of stops.
ApiAccessSecurityTests	test_generic_transport_writes_are_disabled	Creating, replacing, changing or deleting routes, stops and buses returns 405 for anonymous, passenger and driver users, and the records remain.
ApiAccessSecurityTests	test_passenger_directory_is_removed	The passenger list and detail endpoints return 404.
ApiAccessSecurityTests	test_public_transport_reads_remain_available	Route, stop and bus reads, route search, destinations, trip stops, bus ETA and waiting count return 200 without signing in.
ApiAccessSecurityTests	test_registration_and_login_endpoints_are_public	Registration, passenger sign-in and driver sign-in work without an existing session.
ApiAccessSecurityTests	test_assigned_driver_can_update_gps	An assigned driver's GPS update is accepted and stored on the bus.
ApiAccessSecurityTests	test_assigned_driver_can_operate_bus	An assigned driver can activate, reset and deactivate the bus and start a trip.
ApiAccessSecurityTests	test_driver_operations_require_active_driver_and_assignment	Anonymous users, passengers, unassigned drivers, inactive drivers and inactive assignments receive 403 on bus operations and the waiting overview.
ApiAccessSecurityTests	test_expiry_requires_administrator	Only an administrator can expire a request; all other users receive 403 and the request stays WAITING.
ApiAccessSecurityTests	test_default_permission_requires_authentication	The default permission class is IsAuthenticated, and anonymous requests to protected endpoints return 403.
ApiAccessSecurityTests	test_registration_rejects_invalid_passwords_without_creating_accounts	Weak passwords are rejected with 400 and no user or passenger is created.
ApiAccessSecurityTests	test_registration_uses_configured_password_validators	Registration applies the validators in the settings, shown by a stricter minimum length being enforced.
ApiAccessSecurityTests	test_owner_cannot_update_or_delete_waiting_request_directly	Update and delete of a waiting request return 405, and its status and timestamps do not change.

Test class	Test method	What it verifies
ApiAccessSecurityTests	test_creation_ignores_client_supplied_lifecycle_fields	A new request is WAITING and belongs to the signed-in passenger, whatever status, passenger or timestamps the client sends.
ApiAccessSecurityTests	test_owner_can_board_and_complete_through_workflow_actions	The owner can list and read the request and move it from WAITING to ON_BOARD to COMPLETED through the workflow actions.
ApiAccessSecurityTests	test_other_passenger_cannot_use_workflow_actions	Another passenger receives 404 when trying to board, cancel or complete the request, and its status is unchanged.
ApiAccessSecurityTests	test_anonymous_waiting_request_paths_return_forbidden	Anonymous users receive 403 on every waiting-request path and nothing changes.
ApiAccessSecurityTests	test_non_passenger_waiting_request_paths_return_forbidden	Drivers and users without a passenger profile receive 403 on every waiting-request path.
ApiAccessSecurityTests	test_waiting_request_reads_remain_owner_scoped	Another passenger sees an empty list and receives 404 for the request.
ApiAccessSecurityTests	test_viewset_profile_guards_also_protect_direct_method_calls	The viewset's own queryset and create methods refuse anonymous and non-passenger users when called directly.
ApiAccessSecurityTests	test_non_passenger_creation_is_forbidden_before_payload_validation	A driver posting to the waiting-request endpoint receives 403 before the payload is validated.
LoginThrottleTests	test_passenger_login_rate_limit	The sixth passenger sign-in attempt within a minute from one address returns 429 with a Retry-After of 60 seconds; another address is unaffected and the limit lifts after a minute.
LoginThrottleTests	test_driver_login_rate_limit	The same limit applies to driver sign-in.
LoginThrottleTests	test_login_limits_are_separate_and_registration_is_unthrottled	Passenger and driver sign-in have separate counters; registration and route search are not throttled.
LoginThrottleTests	test_authenticated_requests_are_also_limited_by_ip	Signed-in requests to the sign-in endpoints are also counted by address.
DebugDefaultTests	test_debug_is_false_when_environment_variable_is_absent	Debug mode is off when DJANGO_DEBUG is not set.
DebugDefaultTests	test_debug_can_be_explicitly_enabled_for_local_development	Debug mode is on only when DJANGO_DEBUG is set to True.

The tests therefore cover the ETA, the waiting overview, access control, registration, login throttling and the debug default. They do not cover the GTFS import commands, the status and next-stop thresholds, validation of GPS values or a real OSRM instance; this is limitation L7 in Table 6.6.1.

Table D.2 gives a script for the manual check of the passenger interface, prepared by Ngosa Muleba. The column *Status* states whether the case is recorded in the evidence of Chapter 6. Cases marked *To be run* were not captured in a screenshot and should be run and recorded before the interface is relied on.

Table D.2: Manual passenger test script

ID	Steps	Expected result	Status
D1	Open the landing page and choose the passenger option	The passenger page opens with the map and the header bar	Recorded (Figure 6.3.2)
D2	With a driver reporting a position, open the map	A bus marker appears; its popup shows registration number, status and speed	Recorded (Figure 6.3.8)
D3	Wait 15 seconds after the driver sends a new position	The marker moves without the page being reloaded	To be run
D4	Press each refresh control	The bus data is refreshed at once	To be run
D5	Register a new account in Plan Trip	The account is created and the passenger is signed in	To be run
D6	Sign in with the test passenger account	The passenger is signed in	Recorded (Figure 6.3.9)
D7	Press *Forgot password?*	The password-reset page opens	To be run
D8	Search *bauleni*	The route Bauleni – Kulima Tower is offered	Recorded (Figure 6.3.10)
D9	Select the route	Its destinations are offered as buttons	Recorded (Figure 6.3.10)
D10	Select a destination	The ordered stops of the trip are listed	Recorded (Figure 6.3.11)
D11	Select the stop Civic Center	The message *Selected boarding stop: Civic Center* appears	Recorded (Figure 6.3.11)
D12	Press Create Waiting Request	A confirmation appears and the request is listed as WAITING	Recorded (Figure 6.3.12)
D13	Press Create Waiting Request while a request is WAITING	The server refuses a second active request	To be run
D14	Open My Requests with an active bus on the trip	An estimated arrival in minutes is shown	Recorded (Figure 6.3.12)
D15	Open My Requests with no active bus on the trip	No estimate is shown	To be run
D16	Press Mark as Boarded	The status changes to ON_BOARD and the complete control appears	Recorded (Figure 4.8.3)
D17	Press the control to complete the trip	The status changes to COMPLETED	To be run

ID	Steps	Expected result	Status
D18	Press Cancel Trip on a WAITING request	The status changes to CANCELLED	Recorded (Figure 6.3.12)
D19	On the driver dashboard, view the waiting counts	One passenger is counted at the stop chosen	Recorded (Figure 6.3.13)
D20	Repeat D8 to D12 on a phone and on a second browser	The panels remain usable	To be run

The list of the tests that were run on the Driver-Interface development branch (94 passing tests, Section 6.3.2) is not part of the final version of the system and is not reproduced here.

Table D.3 gives a script for the manual check of the driver interface, prepared by the author. The column *Status* states whether the case is recorded in the evidence of Chapter 6. Cases marked *To be run* were not captured in a screenshot and should be run and recorded before the interface is relied on.

Table D.3: Manual driver test script

ID	Steps	Expected result	Status
DD1	Open the landing page and choose the driver option	The driver sign-in opens	Recorded (Figure 6.3.2)
DD2	Sign in with the test driver account	The dashboard opens and lists the assigned bus	Recorded (Figure 6.3.3)
DD3	Sign in with a wrong password	A message states that the credentials are wrong	To be run
DD4	Sign in with a passenger account	A message states that the account is not an active driver	To be run
DD5	Reload the page while signed in	The dashboard opens without a new sign-in	To be run
DD6	Select the assigned bus	Its status is shown as IDLE	Recorded (Figure 6.3.4)
DD7	Press Activate	A message confirms that the bus was activated	Recorded (Figure 6.3.5)
DD8	Choose the route Bauleni – Kulima Tower	The destinations of the route are offered	Recorded (Figure 6.3.5)
DD9	Choose a destination and press Start trip	The trip starts and the stops of the trip are listed with their waiting counts	Recorded (Figure 6.3.13)
DD10	Press Start GPS tracking	The browser asks for permission to read the location	Recorded (Figure 6.3.6)
DD11	Allow the permission	The update is accepted; the coordinates, speed and status IN_TRANSIT are shown and the position appears on the map	Recorded (Figure 6.3.7)

ID	Steps	Expected result	Status
DD12	Refuse the permission	A GPS error message is shown and the dashboard stays usable	To be run
DD13	Open the passenger map	A bus marker appears at the reported position	Recorded (Figure 6.3.8)
DD14	Create a waiting request at Civic Center on the passenger page and look at the dashboard	One passenger is counted at Civic Center	Recorded (Figure 6.3.13)
DD15	Move the device within 30 metres of the current stop at a low speed	The status changes to AT_STOP	To be run
DD16	Move away from the stop at a higher speed	The current stop advances to the next stop of the trip	To be run
DD17	Press Stop GPS tracking	No further updates are sent	To be run
DD18	Press Deactivate, then Reset operational state	The status returns to IDLE and the trip is cleared	To be run
DD19	Try to operate a bus that is not assigned to the account	The server refuses with HTTP 403	Recorded (Table 6.3.1, row 13)
DD20	Press Log out	The sign-in panel is shown again	To be run

APPENDIX E: DATA COLLECTION INSTRUMENTS

Not applicable. No questionnaires, interviews or other data collection instruments were used, because no user study was conducted.

APPENDIX F: ADDITIONAL CODE / CONFIGURATION

This appendix gives selected configuration for reference: the API endpoints of the integration branch and the environment variables read by the application.

API endpoint reference

All routes are under /api/. “Auth” states whether the endpoint requires a signed-in user in the final version of the system. Table F.1 lists the endpoints of the integration branch with their purpose.

Table F.1: API endpoints of the integration branch

Endpoint	Method	Auth	Purpose
auth/register/	POST	No	Register a passenger account
auth/login/	POST	No	Passenger sign-in (five attempts per minute per client address)
auth/me/	GET	Yes	Current passenger
auth/logout/	POST	Yes	Passenger sign-out
auth/driver/login/	POST	No	Driver sign-in (five attempts per minute per client address)
auth/driver/me/	GET	Yes	Current driver
auth/driver/buses/	GET	Yes	Buses actively assigned to the driver
auth/driver/logout/	POST	Yes	Driver sign-out
buses/<registration_number>/ driver-session/	GET	Yes + assignment	Current trip and state of a bus
buses/<registration_number>/ activate/	POST	Yes + assignment	Activate a bus
buses/<registration_number>/ deactivate/	POST	Yes + assignment	Deactivate and reset a bus
buses/<registration_number>/start-fresh/	POST	Yes + assignment	Reset operational state
buses/<registration_number>/ location/	POST	Yes + assignment	Submit a GPS update
buses/<registration_number>/ waiting-overview/	GET	Yes + assignment	Waiting counts per stop of the bus’s trip
buses/<id>/eta/<stop_id>/	GET	No	Road duration from a bus to a stop
routes/search/?q=	GET	No	Search routes by name
routes/<route_id>/destinations/	GET	No	Destinations of a route

Endpoint	Method	Auth	Purpose
routes/<route_id>/stops/?destination=	GET	No	Ordered stops of the trip to a destination
waiting-requests/	GET, POST	Yes (passenger)	List own requests; create a request
waiting-requests/<id>/ (detail)	GET	Yes (passenger)	Own request (ViewSet; update and delete disabled)
waiting-requests/<id>/board/	POST	Yes	Mark as boarded
waiting-requests/<id>/cancel/	POST	Yes	Cancel a request
waiting-requests/<id>/complete/	POST	Yes	Complete an on-board request
waiting-requests/<id>/expire/	POST	Administrator	Expire a WAITING request
waiting-requests/<id>/eta/	GET	Yes	ETA for own request
trips/<trip_id>/stops/<stop_id>/waiting-count/	GET	No	Waiting count for a trip and stop
routes/, stops/, buses/	GET	No	Read-only ViewSets

Outside /api/, the project routes the landing page, the passenger and driver pages, Django's password-reset views under password-reset/ and the administration site under admin/.

Environment variables

Table F.2 lists the environment variables read by the application, with their purpose. Only names are given; values are never included in the repository or in this report.

Table F.2: Environment variables read by the application

Variable	Purpose
DJANGO_SECRET_KEY	Django secret key; required when debugging is off
DJANGO_DEBUG	Debug mode; off unless set to true
DJANGO_ALLOWED_HOSTS	Comma-separated allowed hosts; defaults to localhost, 127.0.0.1 and testserver
DJANGO_CSRF_TRUSTED_ORIGINS	Comma-separated trusted origins for CSRF checks
DATABASE_URL	Address of the PostgreSQL database
OSRM_URL	Base URL of the OSRM service; defaults to http://localhost:5000
BREVO_API_KEY	Credential for the Brevo e-mail backend
DEFAULT_FROM_EMAIL	Sender address for e-mail

When debugging is off, the settings also trust the forwarded protocol header of the hosting proxy and mark session and CSRF cookies as secure. The report does not state a deployed

address for OSRM, nor the OSRM host. The build and start commands used on Render are given in Section 5.4.9.

Driver dashboard: sending the device position

The following excerpt is from the driver dashboard template (tracking/templates/tracking/driver_interface.html). It shows how the dashboard sends the device position to the server.

```
async function publishPosition(position) {
    if (!state.selectedBus || state.publishing) return;
    state.publishing = true;
    try {
        const speed_kmh = (position.coords.speed ?? 0) * 3.6;
        const result = await api(`/buses/$
{encodeURIComponent(state.selectedBus)}/location/`, {
            method: 'POST',
            body: { lat: position.coords.latitude,
                    lng: position.coords.longitude,
                    speed: speed_kmh },
        });
        showBackendLocation(result);
        message('locationMessage', 'GPS update accepted by backend.', 'success');
        await loadWaitingOverview();
    } catch (error) {
        showApiError('locationMessage', error, 'publish GPS for this bus');
    } finally {
        state.publishing = false;
    }
}

state.watchId = navigator.geolocation.watchPosition(
    publishPosition,
    (error) => message('locationMessage', `GPS error: ${error.message}`, 'error'),
    { enableHighAccuracy: true, maximumAge: 1000, timeout: 15000 },
);
```

APPENDIX G: OTHER SUPPORTING MATERIAL

None.