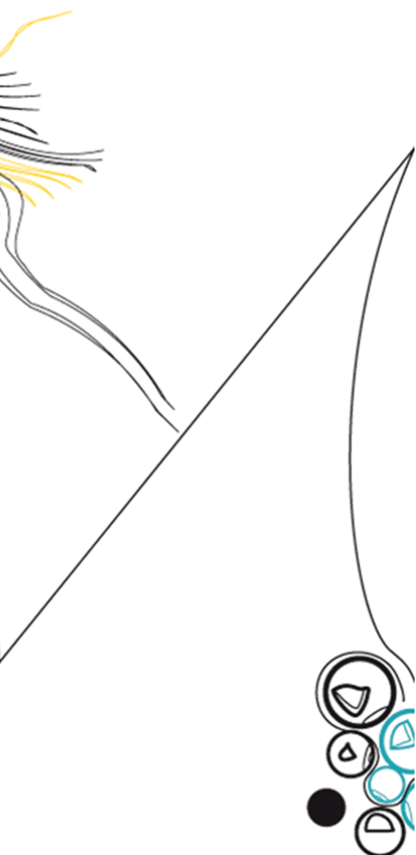
Faculty of Electrical Engineering,
Mathematics & Computer Science

Design Report - MerlinRoadsAI

Group 8

Project Supervisor: Mahboobeh Zangiabady
April 2026



Stefan Munteanu (s3208699)
Wessel Ritskes (s3212599)
Roland Garvasuc (s3167119)
Mihai Santai (s3187446)
Victor Burcovschi (s3209539)
Bogdan Mocanu (s3213552)

Contents

1 Introduction **2**

2 Domain Analysis **4**
2.1 Background & Domain Introduction 4
2.1.1 Introduction to the Domain 4
2.1.2 Project Origin and Institutional Context 5
2.2 General Knowledge of the Domain 5
2.2.1 The MerlinRoads Framework 5
2.2.2 The Accessibility Problem 6
2.3 Stakeholders 7
2.3.1 Primary Users 8
2.3.2 Indirect Users 8
2.3.3 Supporting Institutions 9
2.4 Software Environment 9
2.4.1 Building on an Existing Foundation 9
2.4.2 New Features 10
2.5 Purpose, Scope, and Applicability 11
2.5.1 Purpose 11
2.5.2 Scope 11
2.5.3 Applicability 12
2.6 Conclusions 12

3 System Proposal **13**
3.1 System Introduction & Management Approach 13
3.2 Preliminary Mock-ups 14
3.3 Proposed Achievements & Key Design Pillars 16
3.3.1 Proposed Achievements 16
3.3.2 Key Design Pillars 17

4 Requirements Analysis **19**
4.1 Design Diagrams 19

4.1.1 Use Case Diagram 19
4.1.2 State Machine Diagram 20
4.1.3 Sequence Diagram 20
4.2 Stakeholder Requirements 21
4.3 Functional and Non-Functional analysis 22
4.3.1 Functional Requirements 22
4.3.2 Non-Functional Requirements 24

5 Global & Architectural Design **26**
5.1 Global Design Choices & Work Processes 26
5.1.1 Plugin architecture 26
5.1.2 Artifact-based tool composition 27
5.1.3 Webapp Extension 28
5.2 Technical Stack 29
5.3 System Overview 30
5.3.1 Request lifecycle 30
5.3.2 Agent execution loop 31
5.3.3 Session model 31
5.3.4 Tool-level error handling 32
5.4 MCP Tool Catalogue 32

6 Testing & Quality Assurance **34**
6.1 Test Plan 34
6.1.1 AI Integration Testing 35
6.1.2 Backend Integration Testing 36
6.1.3 Provider Parity Testing 37
6.2 Risk Assessment Contingencies 37
6.3 Test Results 39
6.4 Inter-Rater Agreement 41

7 Future Planning & Maintenance **43**
7.1 Utilization & Support 43
7.1.1 Utilization 43
7.1.2 Maintenance 44
7.1.3 Extensibility 44
7.2 Scalability 45

8 Project Evaluation **47**
8.1 Project Planning vs. Execution 47

8.1.1 Phase 1: Requirements Analysis, Architecture Design, and Project Planning . . . 49
8.1.2 Phase 2: Core Library Refactoring and MCP Server Implementation . . . 50
8.1.3 Phase 3: Agentic AI Client and Dashboard Integration . . . 50
8.1.4 Phase 4: Documentation, Testing, and Refinement . . . 50
8.1.5 Phase 5: Final Delivery and Packaging . . . 51
8.1.6 Summary . . . 51
8.2 Team Evaluation . . . 51
8.3 Stakeholder Feedback . . . 52
8.4 Final Result Analysis . . . 52
8.5 Reflection . . . 53
8.6 Conclusion . . . 55

A Meetings **57**
A.1 Meeting 1 . . . 57
A.2 Meeting 2 . . . 57
A.3 Meeting 3 . . . 58
A.4 Meeting 4 . . . 58
A.5 Meeting 5 . . . 59
A.6 Meeting 6 . . . 59
A.7 Meeting 7 . . . 60

B Onion Models **61**

C Mock-up **64**

D Diagrams **67**

E AI statement **70**

Bibliography **71**

Chapter 1

Introduction

Traffic accidents, as well as city congestion, still remain some of the most significant issues to be addressed in the global arena. According to the World Health Organization, 1.19 million deaths are recorded due to road traffic injuries every year (World Health Organization, 2023). Not only is there an enormous loss of life, but there is also an economic loss to the city due to factors such as the delay in freight logistics, as well as the loss of productivity of citizens due to the accident, which may have a ripple effect throughout the entire city in case of an accident occurring on any of the roads in the city. Thus, it is imperative to consider the importance of real-time understanding and analysis of the traffic in the city. Although conventional analysis techniques have used static data for the purpose of analysis, there is a need for much more than this in the context of modern-day city planning, which requires an intuitive tool for understanding the complex scenarios, as well as alternative routing strategies, which can be used to assess the effects of any incident in the city, thus making microscopic traffic simulation, as well as the use of artificial intelligence in the analysis of the traffic in the city, significant areas of research. The MerlinRoads framework, originally created at the University of Twente, is a research-oriented software framework that addresses this need. It models large-scale urban road networks as primal graphs, where intersections are represented as nodes and road segments as edges, and integrates heterogeneous incident data sources with microscopic traffic simulation powered by SUMO (Simulation of Urban MObility). Earlier versions of the framework centered on a web-based dashboard built with Dash, providing users with the ability to explore disruption scenarios and visualize traffic metrics at the metropolitan scale. The existing codebase offers a good and tested foundation in terms of a working Python core library, SUMO-based simulation workflows, and associated technical documentation. However, as the intricacy of road networks and variety of applications have increased, the hurdle for non-expert users to derive useful insights from the framework still remains significant. Accessing and utilizing the existing system requires understanding its underlying data structures, associ-

ated parameters for simulating the system, and associated visualization tools. This is something that cannot be assumed for traffic engineers, decision-makers, and urban planners who would be benefited most by the associated capabilities. This is a limitation in terms of moving from the underlying potential of associated tools to the interface itself. This project seeks to introduce MerlinRoads AI, which is a new extension of the existing MerlinRoads system. The extension is aimed at converting it from an analytical and structured interface to a conversational interface. The key innovation is in terms of employing the Model Context Protocol (MCP), which is an open protocol for enabling large language models to interact with external tools and data sources in a structured and reliable manner. By exposing MerlinRoads' core analytical and simulation capabilities as MCP-compatible tools, this project enables users to query traffic incidents, trigger route analyses, and simulate disruption scenarios using natural language prompts. For example, the user can ask, ”run an alternative route analysis from Enschede to Amsterdam if the A1 would be closed from Amersfoort to Amsterdam.” The agent will then parse this command, call the appropriate tools, and provide a coherent response without the user needing to interact with underlying code or configuration. This move away from a passive dashboard and towards an active, agentic architecture is part of a larger trend in software engineering towards AI-augmented interfaces that can reduce the barrier to entry for expert-level tools. In this regard, we are developing a working system as well as an architectural model for utilizing research frameworks via natural language interfaces.

Chapter 2

Domain Analysis

2.1 Background & Domain Introduction

2.1.1 Introduction to the Domain

Urban road networks are among the most complex infrastructural systems managed by modern cities. As metropolitan areas grow in population and physical extent, the demands placed on road infrastructure increase correspondingly, not only in terms of daily traffic volume, but in terms of the frequency and severity of disruptions caused by incidents, accidents, and planned road closures. Beyond the immediate human cost, traffic disruptions impose cascading economic and logistical consequences: delays propagate through interconnected road networks, emergency response times increase, and freight logistics are disrupted in ways that affect entire regional economies.

Traditional approaches to traffic incident analysis have relied predominantly on static historical data, textual accident reports, post-hoc statistical analyses, and offline planning tools. While these methods have served urban planners and traffic safety authorities for decades, they are fundamentally limited in their capacity to support dynamic, real-time, or scenario-based decision-making (PIARC Road Safety, 2025). A traffic engineer asked to evaluate the consequences of closing a major arterial road cannot derive timely or actionable insights from static reports alone. What is needed is an interactive, simulation-capable platform that integrates heterogeneous data sources with road network analysis and presents results in an accessible, intuitive form. The MerlinRoads framework models large-scale urban road networks as primal graphs, where intersections are represented as nodes and road segments as edges, and integrates heterogeneous incident data with microscopic traffic simulation powered by Simulation of Urban MObility. The framework delivered a Dash-based web dashboard that allowed users to explore disruption scenarios, visualize traffic metrics at the metropolitan scale, and interact with both real-time

and historical incident data. This provided a well-tested and functional foundation for data-driven traffic analysis.

However, despite the analytical power of the underlying framework, a fundamental question remained: how accessible is such a system to users who lack a background in traffic engineering, graph theory, or simulation tooling? This question, about the gap between what the framework can do and who can realistically use it, motivates the present project.

2.1.2 Project Origin and Institutional Context

MerlinRoads was developed within the academic context of the University of Twente, with CentroGeo serving as a co-institutional partner (te Poel et al., 2025). The framework emerged from research interest in combining geospatial data analysis, road network modeling, and microscopic traffic simulation into a unified platform suitable for both academic investigation and practical urban planning support. The project supervisors, M. Zangiabady (University of Twente) and A. Garcia-Robledo (CentroGeo), have guided the development across successive iterations of the framework. MerlinRoads AI represents a significant evolution in our approach. Rather than simply expanding the dashboard's visual features, we are restructuring the core interaction model. By shifting from manual inputs to a natural language interface, we aim to make our existing visualization tools more intuitive and accessible through direct conversation. This shift is enabled by the adoption of the Model Context Protocol, which provides a standardized mechanism for exposing the framework's capabilities as tools that an AI agent can invoke dynamically in response to user queries. The result is a system that makes the analytical power of MerlinRoads accessible to a significantly broader audience, including users with no prior knowledge of traffic analysis or road network modeling.

2.2 General Knowledge of the Domain

2.2.1 The MerlinRoads Framework

MerlinRoads models large-scale urban road networks as primal graphs(te Poel et al., 2025). In the primal graph representation, intersections become nodes and the road segments connecting them become edges, allowing the full suite of graph-theoretic algorithms, shortest path computation, centrality analysis, and connectivity checks, to be applied directly to the road network. This mathematical foundation is what enables MerlinRoads to answer questions like which roads are most critical to

network-wide flow, or what the fastest alternative route is when a specific segment is removed.

On top of this graph model, MerlinRoads integrates incident data from API's, allowing real-world events, accidents, road closures, and congestion events, to be overlaid on the network and used to contextualize the analysis (te Poel et al., 2025). Historical incident data can be loaded via CSV upload, providing a basis for retrospective analysis of accident patterns and hotspot identification across a road network. This combination of structural graph analysis and empirical incident data gives the framework a dual analytical perspective: it can reason both about the inherent properties of the network and about how real events have affected it over time.

For scenario-based reasoning, asking not what has happened but what would happen under a hypothetical condition, MerlinRoads integrates SUMO, an open-source microscopic traffic simulator developed by the German Aerospace Center (*SUMO - Simulation of Urban MObility*, 2024). Unlike aggregate flow models, SUMO simulates the behavior of individual vehicles, making it sensitive to the kinds of localized disruption that matter most in practice: a lane closure that creates a bottleneck, a diversion that overloads a secondary road, or an incident that blocks a critical junction for a large part of the network. The results of a SUMO simulation can be visualized directly on the dashboard, giving users a concrete picture of how traffic conditions would evolve under the scenario being evaluated.

Together, these capabilities make MerlinRoads a technically robust and analytically capable platform. Its limitations are not in what it can compute, but in who can realistically operate it and how much time and expertise that operation requires.

2.2.2 The Accessibility Problem

The original MerlinRoads dashboard was designed for users who already understood the domain. To obtain meaningful results, a user needed to navigate a graphical interface whose controls assumed familiarity with road network data structures, simulation parameters, and incident classification schemes (te Poel et al., 2025). For instance, setting up a disruption scenario required that the user not only understand what they wanted to know, but also how to articulate that knowledge in a way that the system could execute: what network segments to select, what time windows to specify, what metrics to request, and how to interpret the resulting visualizations in relation to their analysis goals. For users who did not possess this background, the dashboard was of little practical use.

The consequence is that the analytical value of the framework remains concentrated among a small group of technical specialists, while the broader community of decision-makers who could benefit from it is effectively excluded. This is particu-

larly significant in time-sensitive contexts. When a major incident occurs on a road network and authorities need to evaluate alternative routing strategies quickly, the ability to obtain an answer in minutes rather than hours is operationally critical. A system that requires a skilled analyst to manually configure and run a simulation before results can be communicated to decision-makers introduces delays that may have real consequences. An AI-driven interface that can respond to a natural language query in near-real time, invoking the same underlying analysis but without the manual configuration overhead, represents a qualitatively different operational capability.

There is also a subtler dimension to the accessibility problem: the cognitive burden of translating a real-world question into the specific form that a tool expects. Even for technically proficient users, this translation takes time and introduces the risk of error. A user who knows exactly what they want to find out may still make mistakes in configuring the system, selecting the wrong road segment, specifying an incorrect time range, or overlooking a relevant filter. A natural language interface reduces this burden substantially, because the translation from user intent to system operation is handled by the agent rather than the user. The user states their goal; the agent determines how to achieve it.

MerlinRoads AI addresses all of these dimensions simultaneously. By wrapping the framework's existing analytical capabilities in a natural language interface driven by an AI agent, it removes the requirement for users to understand the system's internal logic, reduces the time cost of routine analytical tasks for expert users, and opens the system to an entirely new community of non-expert users who previously had no practical path to engagement.

2.3 Stakeholders

The introduction of a natural language interface does not merely change how existing users interact with MerlinRoads, it fundamentally expands who can use it at all. The following analysis presents the stakeholders served by the original MerlinRoads dashboard and the extended stakeholder community made accessible by MerlinRoads AI. The central difference across all groups is the same: tasks that previously required technical familiarity with the dashboard can now be accomplished through natural language, reducing both the time cost for expert users and the knowledge barrier for non-expert ones. Please refer to Figure B.1 in Appendix B for the visualization of the stakeholders within MerlinRoads AI. The onion model presented in the MerlinRoads 2025 project (te Poel et al., 2025) was used as a reference in constructing our illustration, as shown in Figure B.2.

2.3.1 Primary Users

- **Traffic safety analysts** were among the primary users of the original dashboard (te Poel et al., 2025). They used it to study accident patterns, apply filters by time, location, and severity, and generate reports to inform traffic management decisions. In MerlinRoads AI, routine queries that previously required several minutes of dashboard interaction can be completed in seconds through a single prompt.
- **Urban planners and city officials** were identified as primary users in the original project, but they were recognized to require a user-friendly interface to engage effectively. In practice, the dashboard's complexity placed them near the boundary of the system's usability. MerlinRoads AI removes this boundary entirely, making the system usable for planners who need to respond quickly to queries about specific road scenarios without time to commission a formal analysis from a technical specialist.
- **Researchers at the University of Twente and CentroGeo** continue to be primary users, with the additional benefit that the agent interface enables faster exploratory analysis. Rather than configuring a simulation run manually to test a hypothesis, a researcher can prompt the agent, inspect the results, and iterate, compressing what might otherwise be a lengthy configuration cycle into a rapid conversational exchange.
- **Non-technical stakeholders** represent an entirely new primary user category introduced by MerlinRoads AI. These are individuals, government staff, policy advisors, and operational decision-makers, who have a legitimate interest in understanding traffic network behavior but no technical background in traffic engineering or simulation. The original dashboard offered them nothing practical. The decisions that most directly affect road network design and traffic safety management are often made by precisely this group, and a tool that reaches them in natural language has the potential to improve the quality of those decisions in ways that a technically demanding dashboard cannot.

2.3.2 Indirect Users

- **Government institutions and road authorities** benefit indirectly through the improved quality and speed of the analyses produced by their staff, enabling faster institutional decision-making in time-sensitive situations such as active incidents on the national road network.

- **Educational institutions** benefit from the lowered barrier to engagement with the framework. MerlinRoads AI can serve as a teaching tool in courses on urban mobility or smart city systems, broadening the pedagogical applicability of the project beyond the technically oriented courses for which the original dashboard was most suitable.

- **Urban mobility companies**, including logistics operators, public transport authorities, and ride-sharing providers, gain access to a tool that supports operational planning queries without requiring dedicated technical integration.

2.3.3 Supporting Institutions

- **Commercial incident data providers** supply the incident data feeds that underpin the system's analytical capabilities (te Poel et al., 2025). The framework currently supports integration with TomTom, with HERE Technologies included in earlier versions of the framework.

- **The University of Twente and CentroGeo** remain the primary institutional clients and supervisors, with M. Zangiabady and A. Garcia-Robledo providing oversight of the project's design, implementation, and evaluation. As the intended first adopters of MerlinRoads AI, their feedback has direct implications for the system's design priorities and the criteria against which its success is evaluated.

2.4 Software Environment

2.4.1 Building on an Existing Foundation

MerlinRoads AI is built on top of the existing MerlinRoads codebase, which was delivered by the original development team and constitutes the analytical engine that the AI layer exposes (te Poel et al., 2025). The present project does not replace this foundation; rather, it refactors and extends it to support a more modular, agent-ready architecture. The core library is reorganized to provide clearly separated modules for road graph representation, incident ingestion, geoparsing and map-matching, simulation orchestration, and network-based analytics. This modularization is a prerequisite for reliable MCP integration: each capability exposed as an agent-callable tool must have a well-defined interface and predictable behavior independent of the components around it.

Alongside this modularization, the project introduces a plugin and extension architecture that allows new data sources, geoparsers, simulation backends, or ana-

lytical metrics to be added on top of the existing codebase without modifying its core logic. This ensures that MerlinRoads AI can grow with future research needs without requiring structural changes to the components that the MCP server and agent layer depend on.

The integration point between MerlinRoads AI and the framework beneath it is the DSL layer, a modular abstraction over the framework's simulation and analysis capabilities developed by A. Garcia-Robledo as part of the updated MerlinRoads codebase. MerlinRoads AI stays independent of the underlying simulation and data management systems by focusing integration exclusively on the DSL layer. This separation ensures that updates to those core components won't disrupt or require changes within the agent layer.

2.4.2 New Features

The present project introduces four new components above the existing stack.

The first and central deliverable is the **MCP server**. It wraps the DSL layer's capabilities as a set of named, typed tools conforming to the Model Context Protocol specification, implemented in Python using FastMCP from the Anthropic MCP SDK. Each tool represents a discrete analytical or simulation operation that the framework can perform, such as querying incidents on a road segment, computing alternative routes under a hypothetical closure, or triggering a simulation scenario. The MCP server is designed to be strictly agent-agnostic: it does not depend on any specific LLM provider and can be connected to any MCP-compatible client. The boundary between the MCP server and the DSL layer on one side, and between the MCP server and the chat interface on the other, is maintained as a hard architectural constraint. This ensures that the server can be tested independently, extended with new tools without affecting existing ones, and reused in future deployments with different agent configurations.

The second component is the **AI agent**, the reasoning layer responsible for interpreting user queries in natural language, selecting the appropriate MCP tools, invoking them with the correct parameters, and synthesizing the results into a coherent, human-readable response. The agent orchestrates the tools exposed by the MCP server and presents their outputs in natural language forming a structured analysis for output. The system supports three agents: Claude Sonnet (Anthropic), ChatGPT 4o (OpenAI), and llama (Ollama), with Claude Sonnet serving as the primary provider used in implementation during development and evaluation. The agent-agnostic architecture ensures that all three agents interact with the system through the same MCP interface, and that additional models can be integrated in the future without requiring changes to the MCP server or the underlying framework.

The third component is the **chat interface**, a conversational input panel integrated into the existing MerlinRoads dashboard. The chat interface extends the previous MerlinRoads dashboard, providing a new mode of interaction alongside the existing map and visualization components. The dashboard provides a way to validate agent-generated results through the visual representation of the road network, ensuring that users are not required to accept the agent's outputs without being given context. Agent responses can include references to specific dashboard views, allowing users to naturally transition between asking a question in natural language and exploring the corresponding data visually.

The fourth and final component is the **plugin architecture**, which replaces the previously hard-coded approach to data source management. In this design, each data source is encapsulated as a plugin. These plugins define standardized properties, such as the plugin `name`, `type`, a reference to the underlying `DataSource` implementation, and an optional configuration component for user-defined input. A central `PluginManager` is responsible for dynamically discovering, loading, and instantiating available plugins at runtime, as well as serving as the primary interface for the core system to access plugin data. This design enables new data sources to be integrated without modifying existing code, significantly improving extensibility and maintainability.

2.5 Purpose, Scope, and Applicability

2.5.1 Purpose

The purpose of MerlinRoads AI is to extend the reach of the MerlinRoads framework by removing the domain-knowledge barrier that has limited its user base.

The system is not a replacement for the existing dashboard, but a complementary extension of it. Expert users retain access to all existing dashboard functionality; non-expert users gain a new and accessible entry point to the same underlying capabilities. The primary goal of MerlinRoads AI is to make the platform's existing capabilities broadly accessible to all users, rather than introducing new analytical features.

2.5.2 Scope

The scope of MerlinRoads AI is defined around the delivery of a functional agentic extension to the existing framework. Specifically, the project delivers a Python-based MCP server exposing the codebase's capabilities as agent-callable tools, a chat interface integrated into the MerlinRoads dashboard, an agent-agnostic architecture

compatible with multiple LLM providers, and a testing framework combining unit tests, smoke tests, and integration tests covering MCP tool invocation behavior.

The system operates on pre-loaded and historical incident data. The geographic scope is intentionally general: while development and testing use Dutch road network data as a primary reference, the architecture is designed to support any road network representable in MerlinRoads' graph model. The system is deployed locally, without a cloud or hosted infrastructure requirement.

2.5.3 Applicability

Beyond its immediate use, MerlinRoads AI serves as a template for connecting complex research tools with the people who need to utilize them. By wrapping a specialized framework in a conversational interface, we've created a model that is easily adaptable to other fields like urban planning, logistics, or environmental monitoring. In the world of traffic analysis, this means anyone can now evaluate the impact of road closures, find better detour strategies during accidents, or get meaningful insights from historical data. All of these can be achieved through a simple, intuitive interaction that supports faster, more informed decision-making.

2.6 Conclusions

The original MerlinRoads framework demonstrated that powerful traffic analysis and simulation capabilities could be delivered through a web-based interface, but also revealed a persistent accessibility gap: the system's value was available only to users already familiar with its domain and tooling (te Poel et al., 2025).

MerlinRoads AI addresses this gap by introducing an agent-driven natural language interface. The project extends the system's reach to a significantly broader stakeholder community, including urban planners, policy advisors, and general non-experts who would not previously have been able to engage with the framework productively.

Chapter 3

System Proposal

3.1 System Introduction & Management Approach

MerlinRoads AI proposes a fundamental evolution of the existing MerlinRoads platform. The core codebase remains intact. What changes is the layer through which users access these capabilities. The previous project had a graphical dashboard requiring technical familiarity, while our project aims to have a conversational interface that accepts natural language and returns structured, actionable results.

The system is designed around a five-layer architecture that provides clear separation of concerns across every component: a Core layer housing the refactored MerlinRoads Python library; an Extension layer providing a plugin mechanism for adding new data sources, metrics, or algorithms without modifying core code; a Service layer implementing the MCP server that exposes core capabilities as agent-callable tools; an Agent layer housing the agentic AI client responsible for interpreting user queries and orchestrating tool calls; and a Presentation layer extending the existing dashboard with a conversational chat interface.

The development of MerlinRoads AI followed an iterative, agile-inspired methodology structured around five sequential phases, as defined in the project proposal: requirements analysis and architecture design, MCP server implementation, agentic AI client and dashboard integration, documentation and testing, and final delivery. Each phase produced concrete deliverables that were reviewed before the next phase began, ensuring that design decisions were validated incrementally rather than deferred to the end of the project. This phased structure also allowed the team to absorb the impact of the delayed codebase release in early March 2025, due to our supervisors taking charge of the core library refactoring, since the requirements and design phase had already been completed and documented before implementation began.

The team was organized around component ownership, with each member taking primary responsibility for a distinct part of the system. R. Garvasuc owned the

MCP server implementation, S. Munteanu the agentic AI client, V. Burcovschi the testing framework, W. Ritskes the plugin infrastructure, B. Mocanu the web dashboard and chat interface integration, and M. Santai was responsible for the design diagrams, meeting documentation, and the final report. While each member led their respective component, integration work was handled collectively, with all members contributing to architectural consistency across component boundaries.

Coordination with supervisors M. Zangiabady (University of Twente) and A. Garcia-Robledo (CentroGeo) was maintained through regular online meetings, held after 16:00 to accommodate the time difference with CentroGeo in Mexico. Meetings were held approximately weekly and covered technical direction, progress review, scope confirmation, and implementation decisions. All outcomes were captured in meeting notes, which served as the main record of project direction. Any deviation from the original architectural design was required to be explicitly justified, both in the meetings and in this report, in accordance with the agreement established in the third supervisor meeting.

Design decisions were validated through a combination of supervisor feedback and internal peer review. The preliminary mock-ups were shared with A. Garcia-Robledo before implementation began and his feedback was incorporated into the final interface design, as discussed in Section 3.2. The MCP server architecture and plugin infrastructure were presented to the supervisors in the sixth meeting and received positive feedback, confirming that the implementation direction was sound.

To ensure architectural consistency as the MerlinRoads codebase evolved, we regularly reviewed A. Garcia-Robledo's repository and monitored ongoing updates to the modular version of the framework released in early March 2025. This practice, agreed upon in the third supervisor meeting, allowed us to align the MCP server and agent layer with the DSL infrastructure as it was finalized, and to validate that the proposed integration approach remained compatible with the evolving codebase. The plugin infrastructure, already confirmed as well-designed in the fifth meeting, was adopted without significant modification, allowing our team to focus implementation effort on the MCP server and agent client.

3.2 Preliminary Mock-ups

To communicate the intended user experience before implementation began, a preliminary interface mock-up was produced early in the project and shared with supervisor A. Garcia-Robledo for feedback. Our mock-up turns our abstract architectural ideas into something tangible we can interact with and it gives us a shared reference to keep our design decisions on track. The mock-up consists of four screens, each representing a distinct state of the interface, covering the main map view, the chat

interface with agent selection, the agent switcher dropdown, and the settings panel.

The first screen shows the default state of the application on launch (see Figure C.1). The interactive road network map is being displayed with an overlay of colored edges and incident markers. Red and orange pin markers indicate incident locations distributed across the network, with color encoding used to differentiate incident severity or type. A search bar in the top-left corner allows users to navigate to a specific location, and a settings control sits beside it. In the top-right corner, a sparkle icon button serves as the entry point to the conversational chat interface. The map is the primary view, and the AI interface is an opt-in layer rather than a persistent panel, keeping the map centric as it is the main feature of the overall project.

The second screen illustrates the two-panel layout that appears when the chat interface is activated (see Figure C.2). The left panel displays the road network map. A directional arrow at the boundary between the two panels allows users to collapse the chat and return to the full-screen map view at any time. The right panel houses the conversational chat interface. At the top of the chat panel, an agent selector displays the currently active AI agent alongside a dropdown arrow, indicating that the user can switch between different agents without leaving the interface. This control is made because the system is not bound to any single AI provider. The chat panel shows a realistic example interaction where the user interacts with he chatbot. An input field at the bottom of the panel, labeled “Ask anything,” provides the entry point for user queries.

The third screen shows the agent switcher dropdown in its open state (see Figure C.3). The dropdown lists the available agents alongside an “Add new” option that allows additional agents to be registered. The active agent is indicated with a filled circle marker, while inactive ones use an empty circle. This screen makes the agent-agnostic principle tangible at the interface level. Switching the underlying AI model is a user-facing action requiring no technical configuration, which directly reflects the architectural decision to keep the MCP server independent of any specific LLM provider. The mock-up system supports three agents through this interface: Claude, ChatGPT, and Gemini.

The fourth screen shows the settings panel, which drops down as a modal in the centre of the screen (see Figure C.4). The panel contains expandable sections for different configuration categories, with an APIs section shown in its expanded state listing TomTom and HERE as active data source integrations. An “Add API extension” button is displayed below the listed providers. This is done to give the users a way to extend the system’s data sources without any code-level interaction, consistent with the plugin architecture.

Upon reviewing the mock-up, A. Garcia-Robledo responded positively, noting in

particular that placing the Dash Sylvereye visualization at the centre of the screen with the chat panel on the side was a well-considered layout decision, describing it as a natural and intuitive arrangement that keeps the map as the primary focus while making the AI assistant immediately accessible without taking over the interface. This feedback confirmed that the core layout direction was sound and aligned with the architectural intent of keeping the dashboard and the conversational interface as complementary rather than competing components.

3.3 Proposed Achievements & Key Design Pillars

3.3.1 Proposed Achievements

MerlinRoads AI is expected to deliver the following concrete outcomes:

- **Core Library Refactoring and Modularization.** While we experienced some implementation delays, these were primarily due to our supervisors leading a comprehensive restructure of the existing MerlinRoads Python codebase. Their work involved organizing the system into clearly defined modules, specifically for road network representation, incident data ingestion, geoparsing, and SUMO simulation orchestration. By establishing standardized data models for traffic events and analytical results, they have ensured a level of consistency across all layers that will better support our long-term goals.
- **MCP Server Implementation.** An MCP server is designed and implemented that exposes core traffic analysis capabilities as structured, discoverable tools for AI agents. Tool schemas are defined for operations including road network loading, incident querying with complex filters, SUMO simulation configuration and execution, hotspot metric computation, simulation validation against historical data, and formatted report generation. Tool interfaces are designed to provide sufficient context for agent reasoning while maintaining safety constraints on what the agent can invoke and how.
- **Plugin Architecture Development.** A formal extension mechanism is introduced that allows researchers to add custom data sources, geoparsing algorithms, traffic metrics, or simulation strategies without modifying the core library. Plugin registration interfaces, discovery mechanisms, and standards for exposing plugin capabilities dynamically through MCP are defined and implemented, with example plugins demonstrating the architecture's flexibility.
- **Agentic AI Client Development.** A reference agentic AI client is built that demonstrates autonomous workflow execution by orchestrating MCP tools in

response to natural language queries. The client handles diverse user intents, from simple filtering requests to complex multi-step analyses such as simulation validation, temporal pattern discovery, and scenario comparison, by dynamically planning tool invocation sequences and interpreting results contextually.

- **Enhanced Dashboard Interface.** The existing web dashboard is extended to incorporate a conversational chat interface for natural language interaction, while preserving all traditional visualization capabilities for human-in-the-loop validation. The dashboard displays both user-initiated and agent-generated analyses, supports interactive exploration of road networks and incident distributions, and provides mechanisms for users to inspect, validate, and export agent results.

3.3.2 Key Design Pillars

The design of MerlinRoads AI is governed by four principles that inform every significant architectural and implementation decision in the project.

Agent-agnosticism. The MCP server is designed with no dependency on any specific LLM provider. The system supports three agents in its current implementation: Claude Sonnet (Anthropic), ChatGPT 4o (OpenAI), and llama (Ollama). All three interact with the MCP server through the same protocol-level interface, and the system can be extended to support additional MCP-compatible agents without modifying the server or the underlying framework. This principle protects the long-term value of the system against changes in the AI provider landscape and makes MerlinRoads AI suitable for deployment in institutional contexts where specific AI providers may be mandated or restricted. This principle is also reflected at the interface level, where the agent selector in the chat panel allows users to switch between Claude Sonnet, ChatGPT 4o, and llama directly from the dashboard without any technical reconfiguration of the underlying system.

Strict Separation of Concerns. The five-layer architecture enforces hard boundaries between the core library, the extension layer, the MCP server, the agent client, and the presentation layer. No layer has visibility into the implementation details of non-adjacent layers. This separation ensures that each component can be developed, tested, and replaced independently.

Extension over Replacement. The existing MerlinRoads codebase is extended and refactored. The analytical capabilities, simulation workflows, and data models established by the original project are preserved and built upon. This principle reflects both a practical constraint, as the existing codebase represents significant prior work that should not be discarded.

Research Versatility. The plugin architecture ensures that MerlinRoads AI can grow with the research community that uses it. A researcher who wishes to add a new incident data source, a novel congestion metric, or an alternative simulation backend can do so by implementing a plugin that conforms to the defined extension interfaces, without touching the core library or the MCP server. Plugins can expose their own capabilities as MCP tools dynamically, meaning the agent layer automatically gains access to new functionality as plugins are added, this is done by prompt injection so the agent layer has access to the data in the plugin file.

Chapter 4

Requirements Analysis

The requirements of MerlinRoads AI are formulated according to the SMART guidelines [Mannion and Keepence, 1995]. The SMART method guides the formulation of requirements by ensuring that they are specific, measurable, acceptable, reasonable, and time-bound. By following these guidelines, the developed functionalities can be properly tested and refined when needed. The SMART requirements are prioritized based on the importance of the associated functionality and their impact on the primary stakeholders identified in Chapter 2. The prioritization follows the MoSCoW method, where requirements are categorized into Must-Have, Should-Have, Could-Have, and Won't-Have. This prioritization reflects both the core architectural goals of the project and the practical constraints imposed by the project timeline.

4.1 Design Diagrams

This section presents the three design diagrams produced during the requirements and architecture phase of the project. The use case diagram describes who interacts with the system and what they can do; the state machine diagram describes how the agent behaves internally during a workflow execution; and the sequence diagram traces the message flow between components for a representative analytical scenario.

4.1.1 Use Case Diagram

The use case diagram presents the system from the perspective of the User and the System Administrator.

The User can initiate five use cases: querying historical incident data, analyzing network vulnerability, running a traffic simulation, adding a new plugin, and upload-

ing a dataset. All five connect via an include relationship to Process Natural Language Intent, reflecting that every user interaction is mediated through the natural language interface. Process Natural Language Intent in turn includes MCP Execution, meaning every resolved intent results in one or more tool calls. An extend relationship connects Refine Ambiguous Query to Process Natural Language Intent, representing the case where the agent requests clarification before proceeding.

The System Administrator operates outside the main system boundary and is responsible for configuring MCP tool definitions and monitoring agent reasoning logs, supporting maintenance and extensibility of the system in a research context.

The use case diagram is presented in Figure D.1.

4.1.2 State Machine Diagram

The state machine diagram describes the internal lifecycle of an agent workflow execution from prompt receipt to final response. The system begins in Initializing, where the MCP bridge spawns the server subprocess and loads the available tool schemas. It then transitions to Ready and enters AI Reasoning, where the agent decides whether to invoke a tool or produce a final answer. A tool invocation decision enters the MCP Tool Execution Layer, a composite state grouping Tool Dispatching, Calling MCP, and Parsing Result. Successful results are appended to the message history and the system returns to AI Reasoning for the next iteration. Errors receive a retry hint for the agent to reason over in the following cycle. An Invalid Result transition returns the system to Ready when a result is too malformed to continue, and a maximum iteration guard transitions the system to Done to prevent runaway execution.

The state machine diagram is presented in Figure D.2.

4.1.3 Sequence Diagram

The sequence diagram traces the alternative route analysis scenario: Enschede to Amsterdam with the A1 motorway closed between Amersfoort and Amsterdam. The five participants are the User, the AI Agent, the MCP Server, the MerlinRoads Core, and the SUMO Engine. The User submits a natural language prompt to the AI Agent. The agent first retrieves the road network schema to discover available attributes, then resolves the location of the closure zone to identify the relevant road segments. It then triggers a simulation with the A1 closed and rerouting enabled, and one as a baseline with the network open, both executed via the MerlinRoads Core and the SUMO Engine. The metrics from both runs are compared in a final step, and the agent returns a synthesized natural language result to the User.

The sequence diagram is presented in Figure D.3.

4.2 Stakeholder Requirements

The primary stakeholders of MerlinRoads AI are traffic safety analysts, urban planners, city officials, researchers at the University of Twente and CentroGeo, and non-technical stakeholders such as policy advisors and operational decision-makers. The following user stories capture their functional expectations of the system.

1. **As a user I want to interact with the system using natural language.** A user can type a question or instruction in plain language and receive a structured, interpretable response without needing to navigate dashboard controls manually.

2. **As a user I want to query historical incident data for a specific area or road.** A user can ask the system to retrieve and summarize incident data for a given location, time range, or road segment, and receive a readable result.

3. **As a user I want to run a traffic simulation for a hypothetical scenario.** A user can describe a disruption scenario, such as a road closure, and have the system configure and execute a simulation without manual parameter configuration.

4. **As a user I want to compare the results of a scenario simulation against a baseline.** A user can request a comparison between a disruption scenario and a baseline run, and receive a structured summary of the differences in key traffic metrics.

5. **As a user I want to analyze the vulnerability of a road network.** A user can ask the system to identify critical nodes or edges in the network and receive an analysis of how their removal would affect overall network connectivity and flow.

6. **As a user I want to upload a historical incident dataset for analysis.** A user can upload a CSV file containing historical incident data and have the system make it available for querying and analysis through the agent interface.

7. **As a user I want to view the results of an analysis on the map.** A user can receive links or references in the agent's response that navigate the dashboard to the relevant map view, allowing visual inspection of the result in its spatial context.

8. **As a user I want to add a new data source plugin to the system.** A user can register a new data source through the settings panel, and have it become immediately available to the agent as a queryable source.

9. **As a system administrator I want to configure which MCP tools are available to the agent.** A system administrator can define and update the set of tools exposed by the MCP server, controlling what operations the agent is permitted to invoke.

10. **As a system administrator I want to monitor the agent's reasoning and tool call history.** A system administrator can inspect logs of the agent's reasoning steps and tool invocations to support debugging, quality assurance, and evaluation of workflow correctness.

4.3 Functional and Non-Functional analysis

4.3.1 Functional Requirements

The following functional requirements are formulated according to the SMART guidelines and prioritized using the MoSCoW method.

Must-Have

1. The system must accept natural language input from the user and return a natural language response.

2. The system must expose the MerlinRoads core analytical capabilities as structured, typed MCP tools that any MCP-compatible agent can discover and invoke.

3. The system must support the execution of SUMO-based traffic simulations for user-defined disruption scenarios through a natural language prompt, without requiring the user to configure simulation parameters directly.

4. The system must support the execution of a baseline simulation on the same network for comparison purposes.

5. The system must return a structured comparison of scenario and baseline simulation metrics to the agent, which synthesizes them into a human-readable result.

6. The system must handle tool call failures by attaching recovery hints to failed results, allowing the agent to retry or adjust its approach without terminating the workflow.

7. The system must enforce a maximum iteration limit on agent workflow execution to prevent runaway tool call cycles.
8. The system must support CSV dataset upload as a data ingestion method, making uploaded data available for analysis through the agent interface.
9. The system must integrate the chat interface into the existing MerlinRoads dashboard without removing or degrading any existing dashboard functionality.

Should-Have

10. The system should support the identification and analysis of structurally critical nodes and edges in the road network through the agent interface.
11. The system should return links to specific dashboard views as part of agent responses, allowing users to navigate directly to the relevant map visualization.
12. The system should provide the agent with access to road network schema information, enabling it to construct correctly parameterized tool calls without prior knowledge of the specific network loaded.
13. The system should support multi-turn conversational interaction, maintaining context across successive user prompts within a session.
14. The system should provide an interface allowing users to switch between supported AI agents, namely Claude Sonnet, ChatGPT 4o, and llama, directly from the chat panel without requiring technical reconfiguration.

Could-Have

15. The system could allow users to register new data source plugins through the settings panel, with the plugin's capabilities becoming discoverable by the agent without restarting the system.
16. The system could support the generation of formatted analysis reports summarizing the results of a completed workflow, suitable for export or sharing.
17. The system could support batch scenario generation, allowing the agent to execute and compare multiple disruption scenarios within a single workflow.
18. The system could expose plugin-defined capabilities as additional MCP tools, dynamically extending the agent's tool set as new plugins are registered.

Won't-Have

20. The system will not support real-time incident data ingestion through external APIs in this iteration. Data ingestion is limited to pre-loaded and user-uploaded historical datasets.
21. The system will not provide a cloud-hosted deployment. All components run locally on standard hardware.
22. The system will not support simultaneous multi-user sessions in this iteration.

4.3.2 Non-Functional Requirements

The following non-functional requirements define the quality constraints that MerlinRoads AI must satisfy. They are formulated according to the SMART guidelines and prioritized using the MoSCoW method.

Must-Have

1. The system must respond to natural language queries that do not involve simulation within 5 seconds under normal operating conditions.
2. For queries that involve simulation execution, the system must communicate progress to the user during the operation, ensuring the interface does not appear unresponsive while the workflow is running.
3. The system must accurately represent incident data, network attributes, and simulation results as returned by the underlying MerlinRoads core, without distorting or omitting information during synthesis into natural language responses.
4. The MCP server must operate independently of any specific AI provider. Replacing the reference agent with any other MCP-compatible agent must require no changes to the MCP server or the MerlinRoads core.
5. The system must be compatible with the latest versions of major web browsers, including Chrome, Firefox, and Safari, as the chat interface and dashboard are accessed through a browser environment.
6. The system must run on standard local hardware without requiring cloud infrastructure.

Should-Have

7. The chat interface and dashboard controls should be clearly labeled and intuitively accessible, such that a non-technical user can submit a query and interpret the result without referring to documentation.

8. The system's codebase should adhere to defined coding standards and include sufficient inline documentation to allow future researchers to extend or maintain the system without requiring direct knowledge transfer from the original development team.

9. Adding a new plugin should require no modifications to the MCP server or the core library. The plugin should conform to the defined extension interface and become available to the agent upon registration without requiring a system restart.

10. The system should handle unexpected tool call failures without crashing or entering an unrecoverable state, returning a meaningful message to the user when a workflow cannot be completed.

Could-Have

11. The system could provide response times for simulation-based workflows that are fast enough to support iterative exploratory use, where a researcher submits multiple successive scenario queries within a single working session.

12. The system could maintain consistent response quality across agents, such that substituting the reference Claude implementation with another MCP-compatible model produces results of comparable accuracy and interpretability for standard analytical workflows.

13. The system could support concurrent use by multiple users on the same local deployment without degradation in response quality or correctness.

Won't-Have

14. The system will not provide formal security hardening, user authentication, or access control mechanisms in this iteration. The system is intended for controlled research use on a local deployment and does not expose sensitive data to external networks.

15. The system will not guarantee response times for simulation-based workflows, as execution duration depends on network size, scenario complexity, and hardware performance, all of which vary across deployment environments.

Chapter 5

Global & Architectural Design

5.1 Global Design Choices & Work Processes

5.1.1 Plugin architecture

The MerlinRoads 2025 system originally employed an architecture in which data sources were hard-coded. This made maintenance and extensibility troublesome, as even minor modifications, like changing the back-end identifier for a data source, becomes a task for which many parts of the project had to be changed. The plugin architecture was implemented to overcome this issue, using a `PluginManager` inspired by (Dhandala, 2026). This plugin manager dynamically discovers, loads, and instantiates `Plugin` classes. This plugin manager is used by the rest of the system to access plugins, as these store the data sources.

The architecture was created in a way that would keep as much as possible of the old system in place, an example of this is the `DataSource` class itself, this was kept the exact same, as that part of the structure was already modularized relatively well, the primary changes were in the new `Plugin` and `ConfigComponent` classes. The old MerlinRoads system made use of backend identifiers for each data source, but also had buttons with their names on it for the frontend, to modularize this the plugin architecture introduces the Plugin class which holds a few properties like `name`, `type`, `data_source`, and `config_component`. The `ConfigComponent` was introduced to support dynamic user input for certain data sources, such as user-uploaded data sources requiring file input.

The introduction of the plugin system also required modifications to the system's REST API design. In the previous implementation a separate API-endpoint was made for user-uploaded data, which had a file as a parameter. To make the REST API implementation similar to the previous implementation, plugins would require a way to register their own API-endpoints, the DSL layer would need to dynamically detect such new endpoints, which the MCP server also needs to detect and turn

into tools for the LLM agents to use. Due to time restrictions the decision was made to take the data source stored in the website cookies, and use that to make API calls. This can be done because the data sources are stored using their `to_json` function and should be able to be fully restored using their `from_json` function. While this reduces system complexity and improves maintainability, LLMs no longer have access over data source used, this means they would not be able to change the data source used for a specific simulation. Consequently, making certain functionalities, such as side-by-side API comparison is also no longer possible in one prompt.

5.1.2 Artifact-based tool composition

A recurring challenge in exposing a data-intensive framework through a language-model interface is that intermediate results are too large to transit the context window. A road network for a three-kilometre radius around Barcelona contains approximately twenty-two thousand edges; the vehicle trace produced by a three-hundred-timestep SUMO simulation is larger still. Returning such objects directly from MCP tool calls would exhaust the model's context budget within a single workflow step and would additionally discourage the model from composing tools at all, because each intermediate payload would occupy space that could otherwise hold reasoning or subsequent tool calls.

The MerlinRoads AI server adopts the *artifact identifier* pattern as its central compositional mechanism. Every tool that produces a non-trivial output (a road network, a filter expression, a set of closures, a routed vehicle population, a simulation result) persists the output in an in-process artifact store keyed by a short identifier and returns that identifier to the language model in place of the payload itself. Subsequent tool calls consume these identifiers as arguments. A typical six-step workflow therefore shuttles only a handful of short strings through the model's context, while the underlying artifacts remain resident in the server process and are accessible to any downstream tool.

Three properties of this design are worth noting explicitly. First, the pattern preserves the *agentic* nature of the interaction: the model retains full control over which tools to invoke and in what order, but does so by reasoning about identifiers rather than data. Second, the store provides a natural recovery mechanism. When a tool is invoked with an identifier that is no longer present: for example, because the network it refers to was never built or was evicted from the cache, the tool returns an `artifact-not-found` error that the model interprets as a signal to reconstruct the missing dependency and retry. This behaviour is verified explicitly in the AI integration tests described in 6.1.1. Third, the pattern decouples the observable output of a workflow from the volume of the intermediate data, which in turn decouples the

cost of running a workflow from the size of the underlying network. The same tool sequence that analyses a neighbourhood in Enschede can analyse the whole of Barcelona without any change to the agent's reasoning path or to the prompts used to evaluate it.

5.1.3 Webapp Extension

The frontend is implemented as a Dash-based web application that combines an interactive map canvas with modular UI components for search, filtering, settings, and contextual dialogs. The interface is organized around reusable components (e.g., search bar, settings modal, chatbot drawer, statistics modal), which are coordinated through callback logic and structured component IDs. This design supports clear separation of concerns: map rendering, user controls, and data/state synchronization are handled in dedicated parts of the page logic.

A key architectural feature of the visualization layer is its session-aware behavior. The dashboard reads a session_id from the URL and can load network geometry, incidents, closures, and simulation outputs from shared in-process API sessions. This allows the interface to move seamlessly between standard traffic context (incidents and base network) and simulation context (SUMO-generated metrics) while minimizing unnecessary request overhead. The approach also improves reproducibility: links to session-specific views can be shared, and the visual state can be reconstructed consistently from stored session artifacts.

Compared to the 2025 MerlinRoads project, there has been visual improvements and extensions with MerlinRoadsAI. The main feature that was added is the Chatbot Drawer. It serves as an intelligent interaction layer that connects user intent to the dashboard's traffic-analysis workflows. Instead of relying only on manual control panels, users can issue conversational requests that trigger structured backend actions such as routing and SUMO-related operations, making advanced functionality more accessible to non-technical users. In the frontend, chatbot outputs are integrated with the existing modular UI (search, settings, and map components), so conversational actions result in immediate visual feedback rather than isolated text messages.

A key strength of the chatbot integration is its session-aware and context-aware behavior. Chat responses can carry session context (e.g., session_id), allowing the dashboard to synchronize to the correct network, incident, closure, and simulation artifacts without losing continuity. When SUMO-related tool calls are detected, the interface automatically transitions to a simulation-focused visualization mode and exposes SUMO settings, reducing manual reconfiguration. To improve interpretability, SUMO edge detail popups are shown only when simulation metrics exist, while

non-SUMO statistics remain in a separate modal path. This prevents mixed pre- and post-simulation information and helps users clearly distinguish baseline traffic context from simulation outcomes.

5.2 Technical Stack

The implementation of MerlinRoads AI is organised around a small set of well-established open-source components, chosen for interoperability with the existing MerlinRoads codebase and for long-term maintainability. This section enumerates the elements of the stack and records the role each plays in the overall system.

MCP server layer. The server is implemented in Python using *FastMCP*, the reference server framework distributed with the Anthropic Model Context Protocol SDK. FastMCP provides the tool registration decorators, the JSON-RPC transport implementation, and the standard lifecycle hooks required by the protocol. Tool implementations are distributed across a small number of topical modules: `network`, `geocoding`, `closures`, `simulation`, `analysis`, and `discovery`, each of which registers its tools with a shared server instance at import time. The server communicates with the agent over standard input and output as specified by the protocol.

Agent layer. The agent is a thin Python process that mediates between the user's natural-language input, the MCP server, and a language-model provider. Three provider backends are supported in the current implementation: Claude Sonnet via the Anthropic SDK, ChatGPT 4o via the OpenAI SDK, and llama via a local Ollama endpoint. Each backend implements a common `LLMProvider` interface, so that switching provider requires no change to the agent loop or to the tool definitions registered on the MCP server. Provider selection is driven by an environment variable and is surfaced to the user through the chat interface.

Core framework and DSL. The MCP tools are implemented on top of the *domain-specific language* (DSL) layer developed by A. Garcia-Robledo as part of the updated MerlinRoads codebase. The DSL abstracts the underlying simulation and data-management components and exposes a lazy-evaluated graph of analytical operations. Restricting the MCP integration to the DSL boundary protects the agent layer from future changes to the core library and preserves the design property that new analytical capabilities can be added through plugins without touching the MCP server.

Simulation, geospatial, and data components. Microsimulation is performed by SUMO (Simulation of Urban MObility), invoked through the existing MerlinRoads simulation wrapper. Road networks are constructed from OpenStreetMap via OSMnx and stored in-memory as NetworkX graphs. Incident data is retrieved through the plugin architecture described in 5.1.1; the current implementation includes HERE, TomTom, and an upload-based CSV plugin for user-supplied data.

Web layer. The user interface extends the existing Dash-based MerlinRoads dashboard. Dash provides the component model and the callback system; Flask, which Dash uses as its underlying web server, provides the REST API endpoints and the server-side session used to persist visual settings and the active data source. The REST API shares an in-process artifact store with the Dash callbacks, which permits the chat interface and the existing visualisation components to operate on the same session state without round-tripping through HTTP.

Testing infrastructure. Tests are written using `pytest` and segregated into three categories corresponding to the testing strategies described in 6.1.1: AI integration tests invoking a real language model against mocked MCP responses, backend integration tests invoking the real MCP server against the real backend, and provider parity tests repeating the same scenarios across all supported LLM providers. Integration tests are gated behind environment variables so that they do not run in continuous integration when no API keys are available.

5.3 System Overview

The operational behaviour of MerlinRoads AI is most easily understood as a request traversing four process boundaries: the browser, the Flask web server, the agent, and the MCP server subprocess. This section walks that traversal, then discusses three architectural properties that follow from it: the agent execution loop, the session model, and the handling of tool-level errors.

5.3.1 Request lifecycle

A user interaction begins when the user submits a message through the chat panel of the dashboard. The corresponding Dash callback, which executes in the Flask request context, reads the active data source from the Flask session, serialises it, and attaches it to the chat payload along with the message text and the identifier of the selected language-model provider. The payload is passed to the agent driver,

which is sent to an `AgentRunner` instance configured for the requested provider. The runner constructs a system prompt and enters the agent execution loop described below.

Tool invocations emitted by the language model are forwarded over the Model Context Protocol transport to the MCP server subprocess. Tools that require data from the framework, for example, `build_network` or `fetch_incidents`, issue HTTP requests back to the REST API exposed by the same Flask process that served the dashboard. The REST API handlers operate on the in-process artifact store, returning short identifiers rather than full payloads. Control returns to the language model, which either issues further tool calls or produces its final natural-language response. The response is sent back through the agent driver and rendered in the chat panel as the assistant's reply.

5.3.2 Agent execution loop

The agent layer implements the standard *tool-use* loop specified by each provider's SDK, reduced to three observable states: the model produces a response; if the response contains tool calls, the runner executes them via the MCP bridge and appends the results to the message history; if the response is terminal, the runner returns it to the caller. The loop is bounded by a configurable iteration cap to prevent runaway behaviour in the event of a model failure. Each iteration consists of a single SDK completion call and zero or more MCP tool invocations. The architectural invariant that the system prompt is fixed for the duration of a conversation, and in particular, that the data-source injection cannot be overridden by later user messages, is enforced by passing the same system-prompt string on every iteration.

5.3.3 Session model

MerlinRoads AI maintains two conceptually distinct forms of session state in parallel. The *API session*, managed by `src/api/session_store.py`, is an in-process dictionary shared between the REST API and the Dash dashboard pages. It holds the artifacts produced by tool invocations(networks, incident sets, simulation results) and is keyed by a short session identifier passed through the URL. Access to the dictionary is synchronised by a reentrant lock; the lifetime of an entry is bounded by an eviction thread that runs alongside the web server. The *browser session*, managed by Flask, persists per-user visual preferences and the active data source selection in a signed cookie; it is accessible only within a Flask request context. Dashboards read from the API session for analytical state and from the browser session for display state, with the two kept independent so that the API session can be consumed

by external clients, including the MCP tools, without requiring cookie context.

5.3.4 Tool-level error handling

Tools may fail for reasons that range from transient network unavailability to missing artifact identifiers to invalid filter expressions. Rather than terminating the agent loop on any such failure, the MCP tools return structured error objects that identify the failure class and, where applicable, suggest a corrective action. The language model interprets these errors as further context and typically responds by issuing a repair call: for example, rebuilding a missing network and retrying the original step, or re-querying the network schema and retrying a filter with corrected attribute names. This behaviour is not hard-coded in the agent layer; it emerges from the combination of structured error responses, a system prompt that encourages repair attempts, and the tool-use loop described above. Its reliability is verified explicitly in the AI integration tests enumerated in 6.1.1, and hallucinated or fabricated tool calls were not observed during the evaluation runs reported in 6.1.1.

5.4 MCP Tool Catalogue

The MCP server exposes sixteen tools, organised into six thematic categories corresponding to the stages of a canonical traffic-analysis workflow. The categorisation is purely organisational (the protocol itself imposes no grouping on tools), but mirrors the structure of the agent's typical reasoning and makes the server easier to extend and to evaluate.

Discovery (`list_cached_networks`, `probe_location`). Tools invoked at the beginning of a conversation to resolve ambiguous place names and to detect whether a requested network has already been constructed in the current session. Probing for existing work before rebuilding is the principal mechanism through which the system avoids redundant computation over long conversations.

Network (`build_network`, `get_network_schema`). Tools responsible for constructing road networks from OpenStreetMap and for reporting the attributes available on their edges. The schema tool is critical to the agent's ability to translate natural-language filter descriptions into structured conditions.

Geocoding (`geocode_place`). A single tool that resolves place names to coordinate pairs or bounding boxes, supporting the spatial filtering performed by the clo-

sures category.

Closures (`filter_edges`, `edges_within_bbox`, `edges_nearest_to`, `fetch_incidents`, `incidents_to_closures`, `combine_closures`). The largest category, reflecting the central role of scenario definition in the system. Closures may be defined by attribute, by bounding box, by proximity, or by reference to real-world incident data retrieved through the active data-source plugin. Compound scenarios are constructed through `combine_closures`.

Simulation (`route_vehicles`, `run_simulation`, `run_baseline`). Tools that generate vehicle populations, execute SUMO microsimulation, and run the same scenario without closures to produce a comparison baseline.

Analysis (`compare_to_baseline`, `compare_simulations`). Tools that compute quantitative deltas between simulations and render the results as interpretable summaries. Analytical outputs are returned to the language model both as structured numbers and as plain-language text, enabling the model to cite specific quantities when composing its reply to the user.

Chapter 6

Testing & Quality Assurance

In this chapter, the testing approach, risk assessment, and test results are provided for the MerlinRoads agent and MCP integration layer. The different functionalities and workflows that are tested are described, and the strategies used to validate them are explained. Finally, the test results are presented for all tests, including any issues that were identified and resolved during the testing process. In addition to technical testing, a qualitative expert evaluation was conducted to assess the practical usefulness of the system.

6.1 Test Plan

The MerlinRoads agent project focuses on validating two distinct layers of the system: the AI decision layer, which determines whether the language model selects and chains the correct MCP tools, and the backend execution layer, which verifies that the real MCP server and backend compute produce correct and usable outputs. To achieve comprehensive coverage across both layers, three testing strategies are applied:

1. AI Integration Testing
2. Backend Integration Testing
3. Provider Parity Testing

While the first strategy focuses on the real AI model and its tool-calling decisions using mocked MCP responses, the second strategy targets the real MCP server and backend compute path. The third strategy ensures that the system behaves consistently regardless of which language model provider is used. The following subsections discuss each strategy in detail.

6.1.1 AI Integration Testing

AI integration testing validates that the real language model agent correctly selects, orders, and chains MCP tool calls in response to natural language prompts. In these tests the MCP tool responses are mocked, meaning the test answers the question of whether the real AI decided to use the right tools in the right order, rather than whether the backend produced the correct output. This distinction is important: by isolating the AI decision layer from the backend, any failures can be attributed directly to the model's behaviour rather than to infrastructure issues.

The goal of this testing strategy is to cover the following scenarios:

- **End-to-end agent MCP workflow:** verifies that the agent calls the network listing tool, then the network building tool, and then the schema retrieval tool in the correct order, passing the artifact identifier from the build step into the schema step.
- **Basic network-building prompt:** confirms that the agent enters tool-use mode and calls the network building tool for a simple request, rather than answering from memory.
- **Ordered multi-step workflow:** asserts the exact tool-call sequence and validates correct artifact chaining between steps.
- **Artifact recovery:** verifies that the agent correctly handles an artifact-not-found error by rebuilding the missing dependency and retrying the original step.
- **Invalid filter recovery:** confirms that the agent reacts correctly to an invalid-filter error by refreshing the network schema and retrying the filter step with a corrected condition.
- **Workflow catalogue:** checks several canonical workflow patterns against a catalogue of expected tool sequences, including basic network analysis, attribute-based closures, and geographic closures.

All AI integration tests are invoked by setting a dedicated environment variable together with a valid API key. They are separated from the standard test suite so that they do not block CI/CD pipelines when no API key is available.

Results

All AI integration tests passed successfully for both the Anthropic (Claude) and OpenAI (ChatGPT) providers. The agent consistently selected the correct tools, respected the required call order, and chained artifact identifiers correctly between

tool steps. During the course of this testing phase, the integration tests proved particularly valuable in uncovering a class of errors where the AI agent was not calling the MCP tools in the correct sequence. In several cases the agent skipped intermediate steps or invoked tools out of order, leading to incorrect artifact identifiers being passed downstream and causing the workflow to fail. These ordering errors were identified by comparing the actual tool-call sequences observed during the test runs against the expected sequences defined in the test catalogue. Once identified, the agent prompt and tool-calling logic were revised to enforce the correct ordering, and all affected tests subsequently passed.

The error recovery tests confirmed that the agent does not stop prematurely when encountering artifact or filter errors, but instead re-plans and retries the appropriate part of the workflow. No hallucinated tool calls were observed during any of the test runs.

6.1.2 Backend Integration Testing

Backend integration testing targets the correct operation of the real MCP server and the real backend compute path, with no mocked components in the workflow. These tests start the real MCP server, invoke the real tool implementations, and verify that the outputs returned are usable artifacts for subsequent steps. As such, they answer the question of whether the real MCP server and backend actually produced correct and usable outputs, independently of the AI layer.

Two backend integration test workflows have been defined, covering the following scenarios:

- **Network workflow:** calls the real network building tool, takes the actual artifact identifier returned by the backend, and then calls the real schema retrieval tool with that identifier. A passing result confirms that the MCP layer is connected to the backend, that a real network artifact was built, and that it can be consumed by the next tool in the workflow.

- **Incident workflow:** calls the network building tool, then the incident fetching tool on that network, and finally the incident-to-closures conversion tool using the returned incident artifact. A passing result confirms that incident fetching works on a real network and that the closure artifact is usable for downstream simulation.

These tests require the application to be running in a separate terminal before execution. They are gated behind a dedicated environment variable and support a smaller test network via configurable location and radius parameters.

Results

The network workflow test passed successfully, confirming that the network building and schema retrieval tools function correctly end-to-end through the real MCP server. The incident workflow test initially failed due to an incompatibility between the Claude provider and the incident tool's response format. This issue was identified, flagged, and subsequently resolved.

6.1.3 Provider Parity Testing

Provider parity testing ensures that the system behaves consistently across all supported language model providers: Anthropic (Claude), OpenAI (ChatGPT), and llama (Ollama). Because each provider exposes a different API and tool-calling schema, the MerlinRoads agent normalises these differences internally. The parity tests verify that this normalisation is correct and complete, so that switching providers does not alter the observable behaviour of the system.Two provider parity test suites have been defined:

- **Equivalent tool calls:** verifies that the Anthropic and OpenAI provider implementations normalise the same scripted tool-calling response into the same internal tool-call sequence, producing equivalent tool-call objects from both providers.
- **Semantic tool definitions:** verifies that Anthropic, OpenAI, and Ollama convert the same MCP tools into semantically equivalent provider-specific tool schemas. Tool names, descriptions, properties, and required fields are checked for alignment across all three providers.

Results

Both provider parity tests passed. The Anthropic and OpenAI implementations were confirmed to produce identical tool-call sequences from the same scripted input. Tool schema alignment was verified across all three providers, ensuring that tool names, required fields, and property definitions remain consistent regardless of the model in use.

6.2 Risk Assessment Contingencies

The following table outlines the key risks identified for the MerlinRoads testing approach, together with their estimated likelihood and impact, and the contingency measure applied to mitigate each risk.

Table 6.1: Risk assessment and contingency measures for the MerlinRoads testing strategy.

Risk	Likelihood	Impact	Contingency
Real LLM calls fail due to API key or rate limits during CI	Medium	High	Live LLM tests are gated behind a dedicated environment variable. All other tests use mocked responses and run unconditionally in CI/CD.
Real MCP backend tests require a running application instance	Medium	Medium	Backend tests are gated behind a separate environment variable. Developer documentation specifies that the application must be running in a separate terminal before executing these tests.
Provider behaviour diverges between Anthropic, OpenAI, and Ollama	Low	High	Dedicated provider parity tests verify that both the Anthropic and OpenAI implementations produce identical tool-call sequences and that all three providers expose semantically equivalent tool schemas.
Agent hallucinates tool calls or stops workflow prematurely	Medium	High	Workflow catalogue tests assert exact ordered tool-call sequences for canonical prompts. Agent loop tests verify completion without premature termination. These tests directly detected ordering errors during development.
Incident workflow incompatible with a specific provider	Medium	Medium	Flagged and tracked as a known issue because of the test. Fix applied

The most significant structural risk is the dependency on live API credentials and a running application instance for backend and AI integration tests. This has been mitigated by gating these tests behind explicit environment variables, ensuring that standard CI/CD pipelines are not affected. The provider divergence risk is addressed by dedicated parity tests, and the agent hallucination and ordering risks are addressed through catalogue-based workflow assertions, which also proved effective in practice during the development of the agent.

6.3 Test Results

In addition to the previously defined test cases, the system was validated using a set of predefined workflow scenarios that represent typical user interactions with the platform. These workflows cover everything, from building and filtering networks to routing, running simulations, and making comparisons. After checking each scenario one by one, we confirmed that the agent uses the right MCP tools in the right order and passes the necessary data between each step without a problem. Every workflow running successfully shows that the system is fully functional, the connection between the agent and the MCP server is stable, and it can handle complex tasks without any misses. This makes us confident that the system works well in real-life situations.

The table below summarises the results of all coded and automated tests executed as part of the MerlinRoads testing effort. Each entry records the test category, the testing strategy applied, the provider or execution context, and the outcome. A result of *Pass* indicates the test completed successfully for all applicable providers. A result of *Fixed* indicates the test initially failed due to a known issue that has since been resolved.

Table 6.2: Summary of all test results for the MerlinRoads testing effort.

Test	Category	Provider	Result
End-to-end agent MCP workflow	AI Integration	Anthropic, OpenAI	**Pass**
Basic network-building prompt	AI Integration	Anthropic, OpenAI	**Pass**
Ordered multi-step workflow with artifact chaining	AI Integration	Anthropic, OpenAI	**Pass**
Artifact recovery after artifact-not-found error	AI Integration	Anthropic, OpenAI	**Pass**
Schema refresh after invalid filter error	AI Integration	Anthropic, OpenAI	**Pass**
Workflow catalogue sequence verification	AI Integration	Anthropic, OpenAI	**Pass**
Equivalent tool-call normalisation across providers	Provider Parity	Anthropic, OpenAI	**Pass**
Semantic tool schema alignment across providers	Provider Parity	Anthropic, OpenAI, llama	**Pass**
Real MCP backend network workflow	Backend Integration	Real MCP server	**Pass**

Test	Category	Provider	Result
Real MCP backend incident workflow	Backend Integration	Real MCP server	**Fixed**

Overall, nine out of ten tests passed without issues. The single failing test was traced to a provider-specific incompatibility in the incident tool response format when using the Claude provider. The issue was resolved by a team member. The remaining tests, spanning AI integration, backend integration, and provider parity, all passed and confirmed that the core agent workflow is correct, that the MCP layer is functional end-to-end, and that the system behaves consistently across all three supported providers.

Test Coverage

Code coverage was measured across the entire `test_mcp` test suite using a line-based coverage tool. The suite achieved 100% file coverage, meaning every test module was executed, and an overall line coverage of 52% across the module. This figure reflects the nature of the test suite: a significant portion of the code paths are only reachable when live environment variables are set, meaning that the backend integration tests and the live LLM tests are excluded from standard coverage runs. When these gated tests are not executed, the lines inside them are counted as uncovered, which suppresses the overall percentage.

The coverage results vary considerably between individual test files, and this variation is expected given the different testing strategies involved. The provider parity tests and the agent error-recovery tests reached 100% line coverage, as all their code paths are exercised unconditionally. The workflow catalogue tests also achieved 100% coverage. By contrast, the AI integration tests and the backend integration tests reached between 24% and 35% line coverage, because the majority of their execution paths are gated behind environment variables and are only triggered during live test runs with a real API key or a running application instance.

The lower coverage figures for the gated test files should therefore not be interpreted as insufficient testing. They reflect a deliberate design decision to separate infrastructure-dependent tests from the standard test run, in order to keep CI/CD pipelines fast and unconditional. When the live tests are executed in full, the covered code paths within those files increase substantially, as the agent loop, tool-calling logic, and MCP communication layer are all exercised end-to-end.

The issues identified and resolved during the testing process include:

- **Incorrect tool-call ordering:** during the AI integration testing phase, the agent was observed calling MCP tools in the wrong sequence in several workflow

scenarios. In some cases intermediate steps were skipped; in others tools were invoked before their dependencies had been resolved. These errors caused incorrect artifact identifiers to be passed to downstream tools, resulting in workflow failures. The ordering errors were identified by comparing actual tool-call sequences against the expected sequences defined in the test catalogue. The agent prompt and tool-calling logic were subsequently revised to enforce the correct ordering, and all affected tests passed after the fix.

- **Provider-specific incident response incompatibility:** the Claude provider did not correctly handle the response format of the incident fetching tool in the real backend context. This was identified through the backend incident workflow test and resolved by adjusting the response parsing logic.
- **Premature agent termination:** early versions of the agent loop stopped after completing the first tool call in multi-step workflows, rather than continuing to the next step. This was identified through the ordered workflow tests and corrected by adjusting the loop termination condition.

6.4 Inter-Rater Agreement

To assess the consistency of the expert evaluation, inter-rater agreement was examined across the two evaluators: a police officer specialising in urban traffic management and a logistics planner with experience in long-distance freight routing. Both evaluators used the same five-point evaluation dimensions, though they assessed different scenario types reflecting their respective domains of expertise. This difference in scenario domain is an inherent limitation of the design, as it prevents the computation of item-level Cohen's kappa in the classical sense, which requires both raters to assess identical items.

Given this constraint, inter-rater agreement is reported at the level of shared evaluation dimensions rather than individual scenario ratings. Three dimensions were common to both questionnaires: decision quality, rerouting efficiency, and solution realism. The police officer's mean scores on these dimensions were 9.00, 6.33, and 8.50 respectively. The freight planner's mean scores on the same dimensions were 9.00, 10.00, and 9.50. The two evaluators were in full agreement on decision quality, with identical mean scores of 9.00 out of 10. Agreement was also strong on realism, with a difference of one point between the two means. The largest divergence appeared on rerouting efficiency, where the police officer rated it considerably lower than the freight planner (6.33 versus 10.00). This divergence is not a sign of inconsistent evaluation standards but rather reflects a genuine difference in context: the police officer assessed urban closures where suggested detour paths

occasionally did not align with established local traffic management practice, while the freight planner evaluated long-haul highway rerouting where the system's suggestions matched his professional judgement closely enough that he did not change his initial decisions after seeing the tool.

Across all ratings, the police officer produced a mean score of 8.17 out of 10 with a standard deviation of 1.13, while the freight planner produced a mean of 9.00 with a standard deviation of 1.31. The overall distributions are comparable, indicating that both evaluators approached the questionnaire with a similar calibration of what constitutes a good or poor score. Notably, both raters independently identified rerouting as the dimension most in need of improvement: the police officer gave it the lowest scores across all three scenarios, and the freight planner explicitly noted in his written feedback that additional routing alternatives beyond the optimal path were sometimes implausible. This convergence on the same weakness, reached independently through different scenario types, strengthens the validity of that finding. Because a formal item-level Cohen's kappa cannot be computed under this design, percentage agreement within one scale point is reported as a practical alternative. Across the three shared dimensions, the two evaluators agreed within one point on two out of three dimensions (66.7 percentage), with the rerouting dimension accounting for the sole larger divergence.

A weighted kappa computed on the dimension-level means converted to a five-point scale yields a value that is not meaningful at this sample size and is therefore not reported numerically; the qualitative agreement analysis above provides a more honest account of consistency given the study design. For future evaluations, inter-rater agreement would be strengthened by having both experts assess an identical set of scenarios, which would allow standard Cohen's kappa or intraclass correlation to be computed directly. The current design prioritised ecological validity by presenting each expert with scenarios from their own domain of practice, which produced richer and more credible qualitative responses at the cost of strict comparability between raters.

Chapter 7

Future Planning & Maintenance

7.1 Utilization & Support

7.1.1 Utilization

MerlinRoadsAI bases itself on the idea that the users are supposed to abstract as much as possible from the components and logic that makes up the backend. Before the AI and plugin extension there was a significant learning curve to the system as the users had to understand how to operate with SUMO simulations and traffic analysis either on their own or through the provided documentation. This is already an impediment for interaction as the target users of the webapp do not necessarily have the time to adapt to it.

The primary utilization flow follows a simple pattern: users load or select traffic data, explore incidents through the interactive map, and optionally run simulations to evaluate different scenarios. The results are then visualized directly in the interface, allowing users to inspect road-level metrics and identify congestion patterns or disruptions.

Additionally, the chatbot serves as an embedded support layer during usage. It assists users in discovering features, refining analysis steps, and understanding outputs in context. This reduces the need to consult external documentation and makes the platform more accessible for non-technical users.

The plugin architecture further enhances utilization by allowing flexible data integration. Users can either rely on external APIs for live traffic data or upload custom datasets, while maintaining the same interaction workflow within the dashboard.

All of these small interaction details should work towards making a cleaner and easier user experience. An example of this is shown in Figure 7.1.

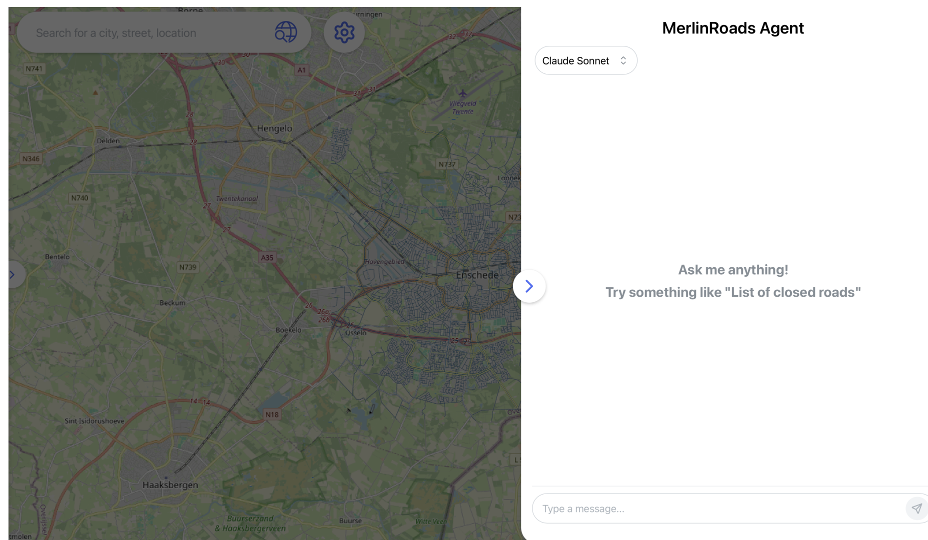


Figure 7.1: Welcoming message for the users

7.1.2 Maintenance

The components within the system that are expected to require the most maintenance are the data sources and the AI agents. These components are inherently subject to change due to evolving APIs and updates to third-party AI providers.

The introduction of a plugin-based architecture for data sources significantly improves maintainability. Each data source is encapsulated within an independent plugin, resulting in a modular structure in which components can be added, modified, or removed without affecting the core system. This reduces coupling and allows new data sources to be integrated with minimal impact on existing functionality.

In contrast, the AI agent infrastructure does not currently employ a plugin mechanism for dynamic loading of providers. As a result, extending the system with new AI providers requires modifications beyond the backend, including changes to the front-end to expose these providers to users. This tighter coupling increases the maintenance effort and reduces flexibility compared to the data source architecture.

7.1.3 Extensibility

The extensibility of the system varies across its components, depending on the degree of modularization and separation of concerns achieved in their design.

The plugin-based architecture for data sources provides a high degree of extensibility. New data sources can be introduced by implementing a new plugin that conforms to the existing interface, without requiring modifications to the core system. This allows the system to evolve incrementally as additional data providers become available, making the integration process straightforward and low-risk.

In contrast, the extensibility of the AI agent infrastructure is more limited. The current design does not fully decouple agent providers from the rest of the system, meaning that the addition of a new provider requires changes across multiple components, including both the backend and the frontend. While this process remains manageable due to the relatively clear structure of the system, it introduces additional development overhead compared to the plugin-based approach used for data sources.

Finally, other core components of the system, such as the REST API, DSL layer, and MCP server, present greater challenges in terms of extensibility. These components are closely interconnected, and extending their functionality, such as introducing new API endpoints or analytical operations, often requires changes across multiple layers of the system. As an example, adding a new API endpoint would likely also need changes to the DSL layer for query handling, as well as modifications to the MCP server to expose the functionality as a tool. As a result, while the system remains extensible, changes to these core components require more careful design and integration effort.

7.2 Scalability

The scalability characteristics of MerlinRoads AI are shaped primarily by the MCP integration layer and by the shared artifact store that underlies it. This section identifies the principal parts along which the system can be scaled, describes what the current architecture provides, and notes the migration paths required to move beyond the single-instance research deployment for which the system was initially designed.

Vertical scaling of the artifact store. The API session described in 5.3 is an in-process Python dictionary guarded by a reentrant lock. This is the appropriate choice for a single-instance research deployment: it removes round-trip latency between the MCP tools and the REST API, and it avoids any serialisation overhead at the artifact boundary. Two limits follow from the choice. The total artifact volume is bounded by the host's memory, which is dominated in practice by SUMO simulation traces and by NetworkX graphs for large networks; and the store does not survive a process restart. Both limits are acceptable for the present scope but should be revisited if the system moves to long-running multi-user operation.

Horizontal scaling path. The current design binds all session state to a single Flask process, so scaling across multiple workers would require externalising the ar-

tifact store. A natural migration is to replace the in-process dictionary with a shared key-value store, for example, Redis, leveraging the `to_json` and `from_json` methods already provided by the plugin architecture (5.1.1). The MCP server itself requires no changes under this migration because its contact with the store is entirely mediated through the REST API boundary; only the REST handlers and the Dash callbacks need to be retargeted. This separation is a direct consequence of the DSL-boundary decision described in 2.4 and represents a concrete payoff of the architectural investment in clean layering.

Concurrency at the MCP layer. Each chat conversation spawns its own MCP server subprocess over the standard-input/standard-output transport, which isolates conversations from one another but introduces a constant cost per conversation in the form of process startup, Python import time, and tool registration. For the expected research-workload regime this cost is negligible relative to the simulation step. For a larger public deployment it would motivate a pooled-subprocess model, or a single long-lived MCP server multiplexing multiple conversations. Per-conversation isolation is preferred in the current design because it prevents a crashing tool invocation from contaminating the state of unrelated users.

Tool-catalogue growth. The plugin architecture described in 5.1.1 permits new analytical capabilities to be added without modifying the MCP server or the agent loop. The practical ceiling on catalogue size is not the server's ability to register tools but the language model's context budget: each tool description consumes tokens on every completion call, and a catalogue that grows substantially beyond the current sixteen tools will eventually pressure the prompt size. Mitigations available within the existing architecture include grouping tools into capability bundles selected by a lightweight intent classifier at conversation start, or filtering the exposed tool set on the basis of the user's initial message. Both are extensions rather than rewrites, again a consequence of the plugin-based design.

Provider economics and rate limits. Each completion call in the agent loop incurs a token cost that scales with the conversation length and with the tool-result payloads returned to the model. The artifact-identifier pattern described in 5.1.2 is the single most important design feature protecting per-conversation cost, because it keeps tool outputs short by construction. Beyond that, the agent-agnostic architecture described in 3.3.2 supports routing individual conversations to different providers in response to cost or rate-limit pressure. The current implementation selects the provider per user rather than per conversation, but no architectural change is required to introduce dynamic routing.

Chapter 8

Project Evaluation

8.1 Project Planning vs. Execution

The original project plan, captured in the Gantt chart produced during the proposal phase, structured the work across five sequential phases spanning ten weeks from early February to mid-April 2025. Comparing the planned timeline against the actual execution reveals several meaningful deviations, all of which rise from having delayed access of the refactored codebase.

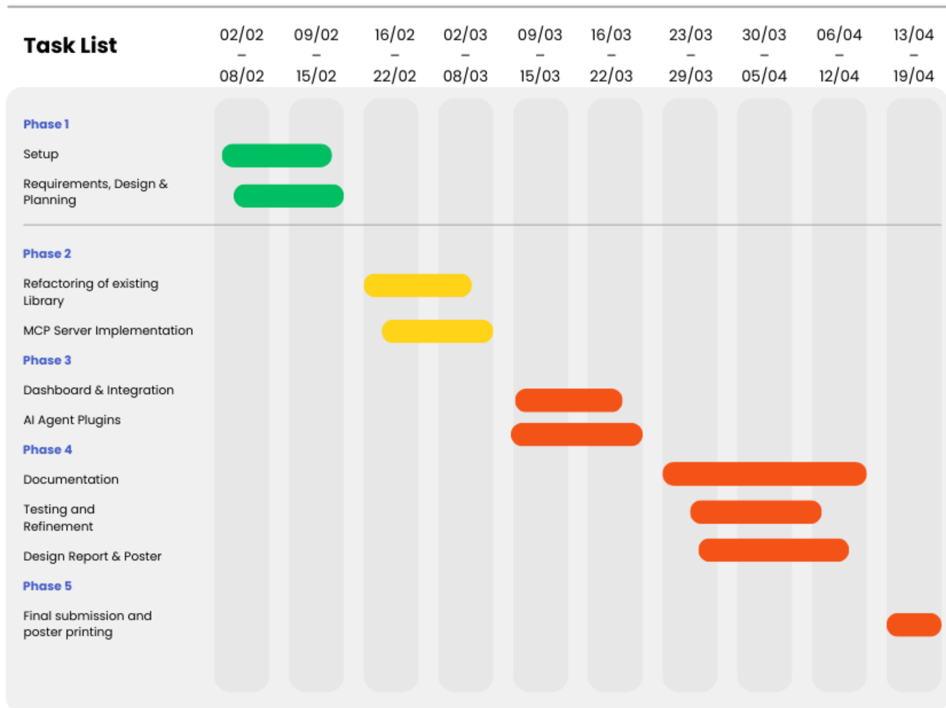


Figure 8.1: Original planned project timeline.

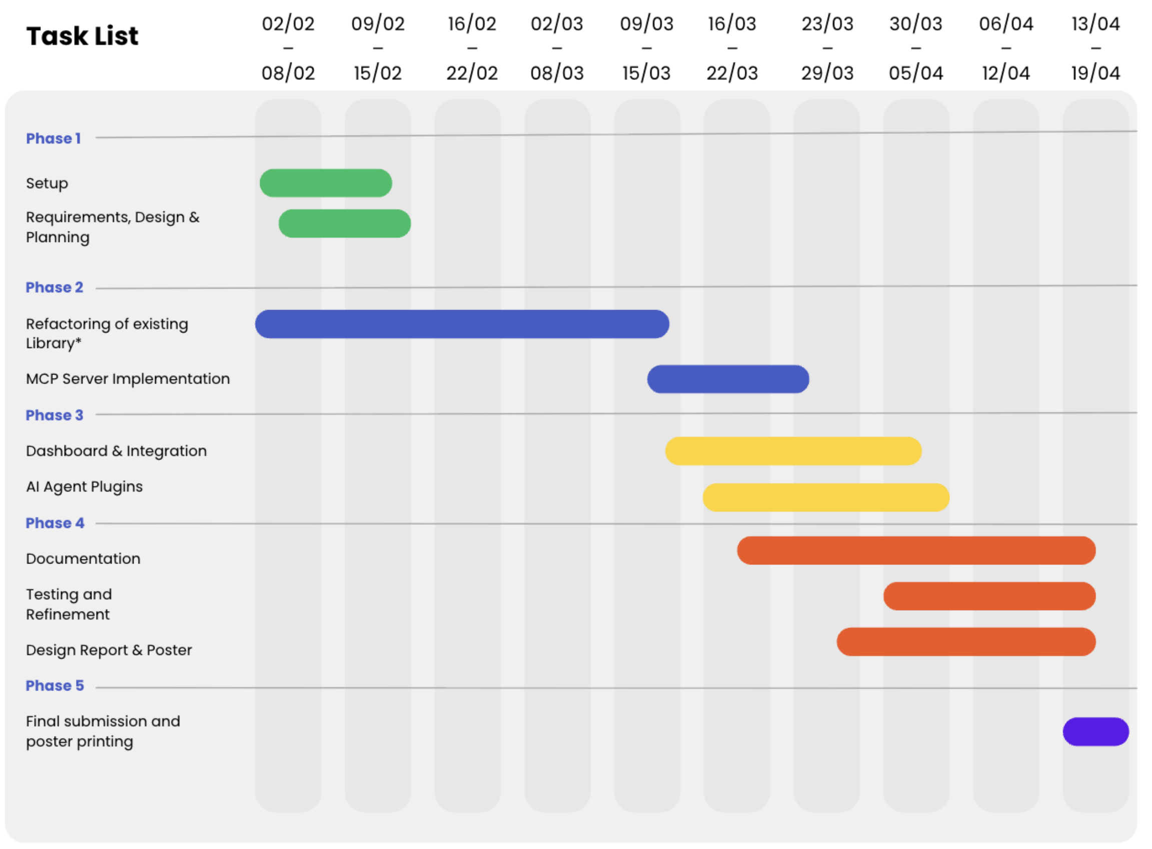


Figure 8.2: Updated planned project timeline.

* The refactoring of the existing MerlinRoads library was performed by supervisor A. Garcia-Robledo as part of the updated codebase release. The team's implementation work was dependent on this deliverable, and its delayed availability in early March 2025 caused an approximately two-week shift across all subsequent phases.

8.1.1 Phase 1: Requirements Analysis, Architecture Design, and Project Planning

Phase 1 was executed as planned. As visible in both Gantt charts, Setup and Requirements, Design and Planning were completed within the first two weeks of February, in time for the project proposal submission on February 20th. The supervisor meetings held during this phase confirmed the technical direction and finalized the system scope. No deviations occurred in this phase.

8.1.2 Phase 2: Core Library Refactoring and MCP Server Implementation

This is where the most significant deviation from the original plan occurred. The original Gantt chart placed the refactoring of the existing library in the week of 16/02 to 22/02, assuming the updated codebase would be released by the end of February. In the actual timeline, the refactoring bar spans from 02/02 all the way to 09/03, reflecting the fact that this task was performed by supervisor A. Garcia-Robledo and was delivered to the team in early March rather than in February. As a consequence, the MCP Server Implementation, originally planned for 16/02 to 08/03, shifted to 09/03 to 22/03 in the actual timeline. Despite this delay, the fifth supervisor meeting on March 11th confirmed that the MCP server architecture was well-structured and that the plugin infrastructure was suitable for adoption without significant modification.

8.1.3 Phase 3: Agentic AI Client and Dashboard Integration

In the original plan, Dashboard and Integration and AI Agent Plugins were both allocated to the period of 09/03 to 22/03. In the actual timeline, both tasks shifted to 16/03 to 29/03, one week later than planned, as a direct consequence of the Phase 2 delays. The extension was also partly driven by the decision to support three AI agents, namely Claude Sonnet, ChatGPT 4o, and llama (Ollama), rather than a single reference agent as originally assumed. This expanded scope introduced additional integration and testing work that was not accounted for in the original plan.

8.1.4 Phase 4: Documentation, Testing, and Refinement

In the original plan, Phase 4 tasks were allocated to the period of 23/03 to 12/04, with Documentation ending at 05/04, Testing and Refinement at 05/04, and Design Report and Poster at 05/04. In the actual timeline, all three tasks extended significantly further. Documentation ran from 23/03 to 19/04, spanning the entire final stretch of the project. Testing and Refinement ran from 30/03 to 19/04, and Design Report and Poster from 30/03 to 19/04 as well. These extensions reflect the cumulative effect of the Phase 2 and Phase 3 delays: with implementation work continuing into early April, documentation and testing necessarily followed on from implementation rather than running fully in parallel as originally envisioned.

8.1.5 Phase 5: Final Delivery and Packaging

Phase 5 was executed as planned. The final submission and poster printing took place in the week of 13/04 to 19/04, and the presentation to supervisors, a peer team, and a faculty colleague was held on April 15th, marking the successful delivery of all project outputs within the original ten-week window.

8.1.6 Summary

The primary cause of all deviations was the delayed delivery of the refactored MerlinRoads codebase, which shifted the entire implementation timeline by approximately two weeks. Despite these deviations, the project was completed within the original ten-week window and all core deliverables were produced. Prioritizing architecture and requirements during Phase 1 mitigated the impact of implementation delays. With design decisions finalized early, the team was able to begin development immediately upon receiving access.

8.2 Team Evaluation

The team maintained a consistent work rhythm throughout the project, with progress distributed across components advancing in parallel. Members communicated informally on a daily basis, with supervisor meetings serving primarily to address open questions and confirm direction. This structure worked well during the design and implementation phases, where component ownership provided sufficient separation of concerns. Integration placed more demand on coordination, as dependencies between the MCP server, the agent client, and the dashboard required members to align closely on shared assumptions. These points were addressed collectively, with members contributing across different components where needed.

The most significant external challenge was the delayed access to the modular MerlinRoads codebase, available only approximately two weeks before the sixth supervisor meeting. This compressed the implementation timeline considerably. The team absorbed the impact without major consequences for delivery, largely because the requirements and design phases had already been completed, providing a clear architectural direction the moment access was granted.

Supervisory feedback served as an effective validation mechanism at a critical point. The team was able to proceed without architectural rework during the most time-constrained phase of the project because the MCP server and plugin infrastructure were well-structured, as confirmed in the sixth meeting.

8.3 Stakeholder Feedback

During our peer reviews we gathered feedback from the teams of students present in the room. After each peer review presentation there was a question session, where the students would ask any uncertainties about the project and would sometimes give feedback on design choices and implementation techniques. Overall, these discussions had a positive tone and if some valid questions were raised we took a closer look at the matter, and eventually implemented the said question.

Additionally, during our meetings with our client and supervisor, we constantly presented choices and presented our dashboard. Each time we got a positive answer from both our client and supervisor and ultimately ended into having a system that met their expectations. Ultimately, on 15.04 we had a presentation with our supervisor, some of her colleagues, and another team of students. The feedback we got was positive on the demo we held.

The stakeholder feedback also aligns well with the results obtained from the expert evaluation. Both evaluators highlighted that the system performs well in terms of overall decision quality and realism, while also stating that rerouting is the main area for improvement. This is consistent with the qualitative feedback, where both stakeholders stated that, even though the optimal routes are generally correct, the other suggested routes do not always make sense. Alternative routing suggestions can sometimes be less realistic or harder to interpret. The convergence of independent expert opinions on the same limitation strengthens the validity of this finding. Overall, this combination of qualitative feedback and structured evaluation suggests that the system is reliable in its core functionality, but would benefit from improvements in the generation and presentation of alternative routing options.

8.4 Final Result Analysis

The final system delivered by this project meets the core objective established in the project proposal. All Must-Have and Should-Have functional and non-functional requirements were implemented. The MCP server exposes sixteen tools across six thematic categories, the agentic AI client supports three LLM providers, the plugin architecture enables extensible data source integration, and the chat interface is fully embedded in the existing dashboard without degrading any prior functionality.

That said, the final system has a number of known limitations that should be acknowledged. Real-time data ingestion through external APIs was intentionally scoped out and remains unavailable through the agent interface. The llama provider, while supported, proved very slow and can make at most one or two tool calls, with local testing being difficult in the absence of a dedicated GPU. The custom dataset

that a user uploads is available only during one session and it is not stored anywhere for future use. Testing validates most requirements implemented, however, more can be done in this direction. Claude Sonnet was prioritized in testing since it was the first provider implemented, a known limitation is that when using the OpenAI provider the output is not as well presented as the one using the anthropic provider. On top of that, when using the OpenAI provider the agentic loop since to use error recovery techniques more often and some tool calls are not called in order, only in rare cases. A way to fix the presentation of the output is to make provider-specific system prompts in a future iteration of the project. Furthermore, there are several issues that seem to be linked to the initial codebase/DSL layer, some SUMO variables are not showing the expected differences between a closure scenario and a baseline, furthermore, the system struggles to pinpoint exact locations for closures when no exact data is given, as seen from the user validation testing interviews, in other words, if the prompt is ambiguous the probe_location tool tries to get a place based on what the LLM interpreted but fails to do so after iterations of error recovery or in most cases closes more than what was asked.

Altogether, our system extends the previous MerlinRoads in the way it was meant, but it needs more refinement to get to the best possible outcome.

8.5 Reflection

Bogdan: The MerlinRoadsAI project was a challenging but enthralling project to work on. For me personally, a big drive in working on this project was that I could feel the importance of it and that it could genuinely become a real-life product with enough time and work investment. This fact however added a layer of difficulty to the project because when real life factors are involved then we have to make sure our system is reliable and consistent. Another challenging part comes from the scope of the project as it was my first time tackling a complex Agentic AI system implementation, and even though I mostly worked on the frontend and visualisation, I have also been involved in testing and documenting how our tools work together. Another nice learning experience was working with already existing codebase and libraries, specifically Dash Sylvereye. Learning and working with local libraries is a core skill that every programmer should know and this project allowed me to apply it in action. Overall, I think the project had a good balance between engaging work and hard challenges, which is exactly what I was looking for when going into Module 11.

Victor: The MerlinRoads AI project ended up being more interesting than I expected at first. Even if we faced a bunch of problems like the AI agent selecting and ordering tools incorrectly or branch merging and integration conflicts, looking back,

those were the moments I learned the most from, even if they were frustrating at the time. From a technical side, I came into this project not really knowing how an AI agent is built and how it can connect a backend tool with the chatbox in the frontend. By the end, I had a much clearer picture of how an agent decides which tools to call, how the backend and frontend talk to each other, and how you actually test something like this in a meaningful way. Another thing I learned was splitting testing into an AI layer and a backend layer sounds obvious in hindsight, but figuring out why that separation matters that much when you have a project like this was new to me. Overall, I am glad I worked on this project since it gave me valuable experience and I'm happy I had a good team, which made the completion of the project a doable task.

Stefan: Personally, I saw the project as something I am really interested in since the start and I do not regret approaching it. As always, some collaboration issues were met but were solved through communication and collaboration. I feel that the task division worked as smoothly as it could have and we had a good result in the end especially because of the circumstances. The project itself was until the end very interesting to me because I want to specialize in artificial intelligence and I feel that AI integration in different types of systems is where business will thrive in the near-future. Hence, getting to implement this project I feel that I have learned about AI integration and extended my knowledge in this domain. Overall, the team worked as flawless as possible and we made a solid end-product, I am very glad I was part of this project and that we could learn about urban planning as much as I am glad that we had the opportunity to leave our mark on a project that might get published.

Mihai: My role in this project was mostly documentation and design, which sounds straightforward until you are actually doing it while the system is still being built around you. Keeping the report accurate meant staying on top of what everyone else was implementing, which was sometimes harder than writing the report itself.

The design diagrams were probably the part I found most interesting. Drawing out the use case, state machine, and sequence diagrams forced me to understand the system at a level of detail that I would not have reached otherwise, and a few times it actually surfaced things that had not been clearly decided yet. It made me realize that documentation is not just a record of decisions, it is part of how decisions get made.

Looking back, I think the team handled a genuinely difficult situation well. The delayed codebase access could have derailed the project, but everyone stayed focused and we ended up with something that actually works and that I think we can be proud of.

Roland: Working on the MCP layer was the part of this project I found most genuinely interesting, and it stayed interesting the whole way through. A lot of that

came from the fact that I was constantly in a space where I did not quite know what I was doing, which made me feel like wanting to find out more. Debugging in particular felt completely different from anything I had done before; when something went wrong it was almost never the Python that was at fault, and I had to slowly get used to the idea that a "fix" could be a rewritten sentence rather than a rewritten function. That was uncomfortable at first and then, after a while, strangely satisfying. The moment I realised that returning short identifiers instead of full payloads was what actually made long conversations viable was probably the most rewarding single moment of the project for me, because it felt less like solving a problem and more like understanding why the problem existed in the first place. Beyond the technical side, I enjoyed being part of a team where the six of us were clearly invested in the same thing, and the weekly rhythm of planning and reviewing made the whole experience feel steady rather than stressful. I leave the project more curious about systems like this than when I started, which I think is the outcome I most hoped for.

Wessel: I worked on the plugin infrastructure and I've helped other people in their sections, especially when it came to implementing the plugin infrastructure, think about the settings menu in the frontend, but also the REST API, which then also changes the DSL layer and MCP tools. While I do believe we achieved great results in the time we had, I also believe with slightly more time we would be able to polish the product a lot more. An example of this is that at the moment we do not have our AI agent providers loaded dynamically, which in turn hinders extensibility and maintainability. I also believe that with our current implementation of agents, and our current implementation of plugins it would not be difficult to add a different plugin type to do so. Overall I still think we achieved a lot, and the final product definitely is something to write home about, as when we first got this project, the main dashboard was quite intuitive, but navigating the SUMO dashboard was a nightmare, especially as someone with little experience in urban planning, which I believe in our version got more intuitive, and easier to use.

8.6 Conclusion

This project delivered an agentic AI interface for the MerlinRoads traffic analysis framework, built on the Model Context Protocol. The system exposes sixteen tools across six thematic categories through an MCP server, supports three LLM providers through a unified agent client, provides a plugin architecture for extensible data source integration, and embeds a chat interface directly into the existing Merlin-Roads dashboard. All Must-Have and Should-Have requirements defined in the project proposal were implemented and validated.

The primary external constraint was the delayed delivery of the refactored Merlin-

Roads codebase, which compressed the implementation timeline by approximately two weeks. This was absorbed without consequence to delivery scope, largely because requirements and architecture were finalized during Phase 1, allowing implementation to begin immediately once access was granted.

Several limitations remain. Real-time data ingestion through external APIs was scoped out and is not available through the agent interface. The Ollama provider is functional but slow and constrained in tool call depth. Also, custom user datasets are session-scoped and are not persisted.

Both experts and stakeholders reached agreed on the fact that our core decision-making and route realism are highly effective, but our alternative routing suggestions need improvement. This shared perspective from independent groups gives us strong confidence in our assessment and our path forward.

The system constitutes a meaningful extension of MerlinRoads toward the conversational, agent-driven interaction model that the framework was being refactored to support. Further refinement, particularly in routing output quality, provider-specific prompt engineering, and persistent data handling, would bring the system closer to production readiness.

Appendix A

Meetings

A.1 Meeting 1

For our first meeting, we visited M. Zangiabady's office at Zilverling. During the meeting, she outlined our project responsibilities and clarified her expectations for the final outcome. She emphasized the importance of enjoying the work and approaching it creatively. Additionally, she mentioned her goal of publishing the project, which requires us to invest significant effort in both the quality and depth of our work.

We also discussed practical arrangements. Since our co-supervisor is based in Mexico, all meetings will be scheduled after 16:00. For the upcoming week, we were instructed to run the Merlin Roads code and create a new GitHub repository specifically for our project, instead of using the original repository.

The next meeting is planned for next week, on Wednesday, Thursday, or Friday, which will be confirmed later. The deadline for submitting the project proposal is next Friday. The supervisor will provide further guidance on other deadlines. It was also emphasized that documentation is a crucial component of the project. Additionally, the project proposal must include a Gantt chart to outline the planning and timeline.

A.2 Meeting 2

Our second meeting took place online on February 18 at 17:00, scheduled in the late afternoon to accommodate Mr. Garcia, who is based in Mexico. During the meeting, we discussed the technical direction of the project and refined the scope of the proposal.

The project proposal due on February 20 will consist of eight pages and will not be graded. It was clarified that the HERE API can be modified if necessary to support the intended architecture.

The core objective is to design and implement an MCP layer between Merlin

Roads and an agentic AI system. This layer should be generalizable, allowing integration with different agents, such as Claude. To achieve this, we need to study the MCP protocol, relevant Python libraries, and the architectural requirements for interoperability.

In addition, we are required to integrate a new chat interface into Merlin Roads. This interface should enable users to interact with the system in natural language and support advanced language models, such as Sonnet or comparable models. The design should ensure modularity and compatibility with multiple agents.

We will receive access to a new, more modular version of Merlin Roads in early March. This updated version will serve as the foundation for our implementation. It was also noted that M. Zangiabady will be unavailable from March 13 to March 20, which should be considered in our planning.

A.3 Meeting 3

During our third meeting on February 25th, coinciding with spring break, we delved into the final report requirements and research publication standards. Every aspect of the assessment form must be comprehensively addressed, encompassing problem definition, architecture, MCP design, implementation, evaluation, results, limitations, and future work. The report should adhere to a clear structure and provide measurable evaluation. Additionally, we mutually agreed to regularly review Alberto's repository to ensure modularity and architectural consistency. The MCP layer must remain agent-agnostic and distinctly separated from Merlin Roads and the chat interface. Any design deviations should be explicitly justified within the report.

A.4 Meeting 4

Our fourth meeting, held online on March 4, commenced with a discussion regarding the upcoming release of the next version of Merlin Roads. Due to personal circumstances, Mahboobeh is currently unable to provide assistance to Alberto with the release process. Consequently, we were advised to concentrate on comprehending the DSL layer and closely monitor the ongoing code updates to ensure a comprehensive understanding of the system upon the release of the new version, thereby mitigating potential delays in the implementation phase. Subsequently, the meeting focused on refining the technical direction of the project. We were recommended to engage in small tasks to gain a deeper understanding of the workflow and validate the efficacy of the proposed approach in practice. Alberto will determine the ap-

propriate method for our team to interact with the primary repository, either through forking or branching, and will promptly communicate the final decision.

A.5 Meeting 5

Our fifth meeting was held online on March 11. The discussion centered around clarifying testing expectations and confirming several implementation decisions. It was determined that relying on the integration tests provided by the supervisor is not sufficient. Additional integration tests need to be added to fully test the MCP server, and other testing procedures, like smoke tests and unit tests will be added. The supervisor also confirmed that the current plugin infrastructure is well-designed, allowing us to continue using it without significant modifications.

We also clarified several architectural and implementation constraints. Our system will only interact with the DSL layer, which will handle communication with SUMO, eliminating the need for direct interaction with SUMO. For the AI component, we are free to utilize any LLMs in our implementation. Additionally, we will develop a dashboard with a chat interface, utilizing links to visualize relevant data from the dashboard. For the final report, it is imperative to clearly document all design and implementation aspects while strictly adhering to the grading rubric. Notably, there will be no meeting on March 18 due to the unavailability of one supervisor. However, further inquiries will be directed to the other supervisor.

A.6 Meeting 6

Our sixth meeting was held online with both supervisors present. The meeting opened with M. Zangiabady raising concerns regarding the project timeline, given that access to the codebase had only been provided two weeks prior. The team expressed confidence in delivering all planned components within the remaining timeframe. The meeting then moved to progress presentations. R. Garvasuc presented the current state of the MCP server implementation. The supervisors responded positively, noting that the server is well-structured and meets the architectural expectations set in earlier meetings. W. Ritskes followed with a presentation of the plugin infrastructure, which was also well received. Several implementation decisions were discussed and confirmed. M. Zangiabady requested that a pop-up window for CSV dataset upload be added to the dashboard interface. This will be incorporated into the ongoing dashboard development work. The supervisors expressed satisfaction with the team's overall progress, acknowledging the challenging circumstances caused by the delayed codebase access. A. Garcia-Robledo suggested that the

evaluation of the agent's results should include a presentation to peer students and faculty, in order to gather broader feedback and validate the system's usability and effectiveness. This will be considered in the planning of the evaluation phase.

A.7 Meeting 7

The meeting opened with a discussion regarding an upcoming conference at which the project will be presented. The team has been invited to participate with a poster. Feedback was provided on the current poster design: the system flow should be made more visually prominent, with the addition of arrows to guide viewers through the architecture and make it more approachable for potential users unfamiliar with the project. Beyond this adjustment, the poster was considered to be in good shape. The final presentation was scheduled for April 15th at 10:00 AM. It will take place in front of M. Zangiabady, a colleague of hers, and another project team she supervises. On the implementation side, support for OpenAI and Ollama as additional agentic AI clients was completed. Several limitations were noted during testing. llama was observed to be significantly slower than the other agents and is only reliable for smaller, simpler tasks. Testing it locally is also difficult in the absence of a dedicated GPU. Regarding SUMO simulation behavior, two issues were identified: when running a simulation with Claude Sonnet, the agent consistently concludes that closing a road improves the network, which is an incorrect result; and when using the open network configuration, SUMO reports that all vehicles are dropped. Additionally, simulation IDs were found to be getting mixed with each other, though this issue appears to be isolated to the SUMO simulation component. A further limitation noted was that it is currently not possible to visualize a simulation after it has been executed. For the evaluation phase, the team agreed to schedule interviews with two participants for user validation purposes. S. Munteanu conducted a demo of the chatbot, and W. Ritskes presented the plugin infrastructure. The supervisors were pleased with our work.

Appendix B

Onion Models

The following figures present the stakeholder onion models referenced in Section 2.3. Figure B.1 illustrates the onion model constructed for MerlinRoads AI, while Figure B.2 presents the onion model from the original MerlinRoads project (te Poel et al., 2025), which served as a reference during its construction.

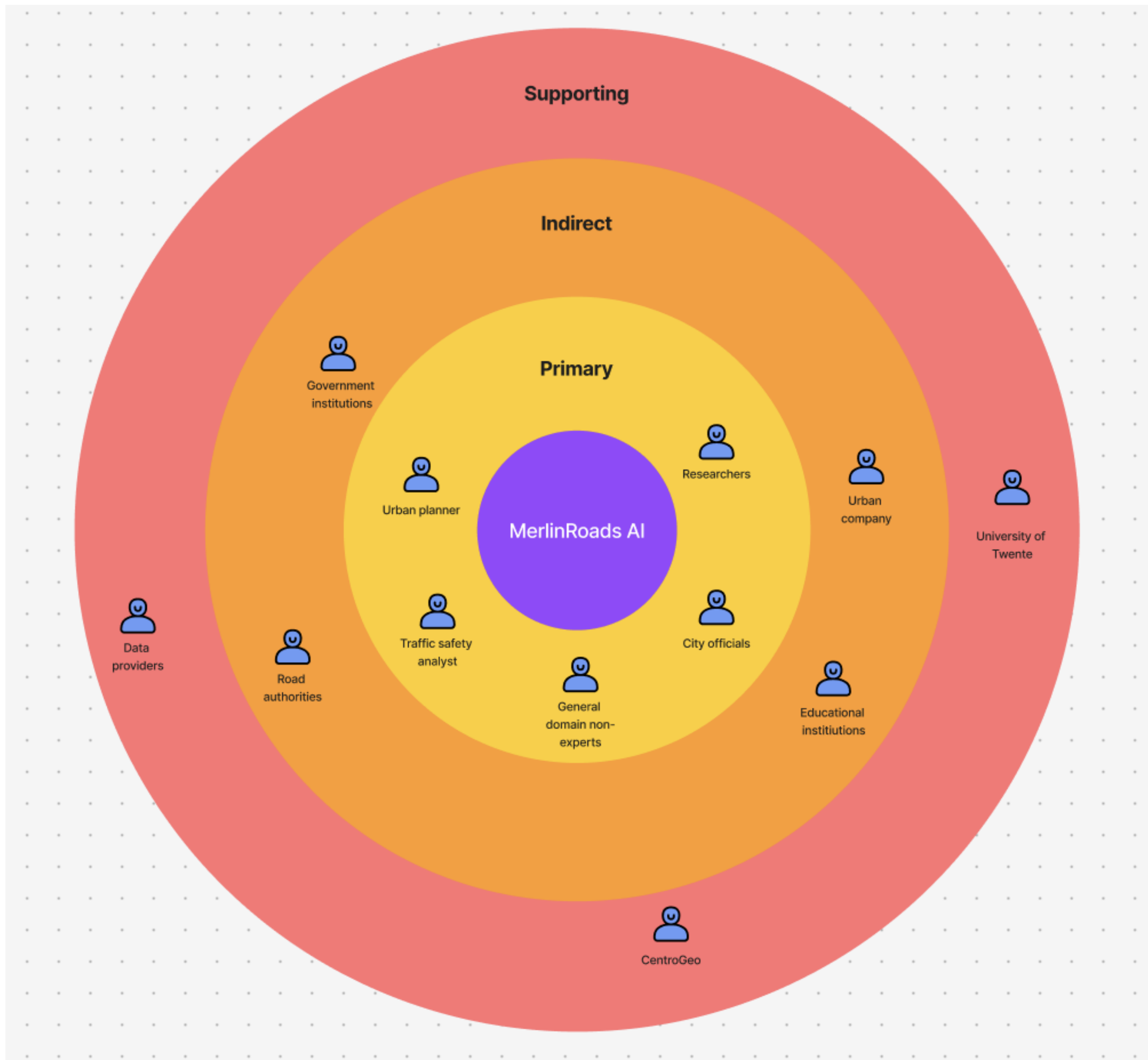


Figure B.1: Stakeholder onion model for MerlinRoads AI.

Figure B.2: Stakeholder onion model from the MerlinRoads project (te Poel et al., 2025), used as a reference for the MerlinRoads AI illustration.

Appendix C

Mock-up

The following figures present the four screens of the preliminary interface mock-up produced during the design phase of the project, as discussed in Section 3.2.

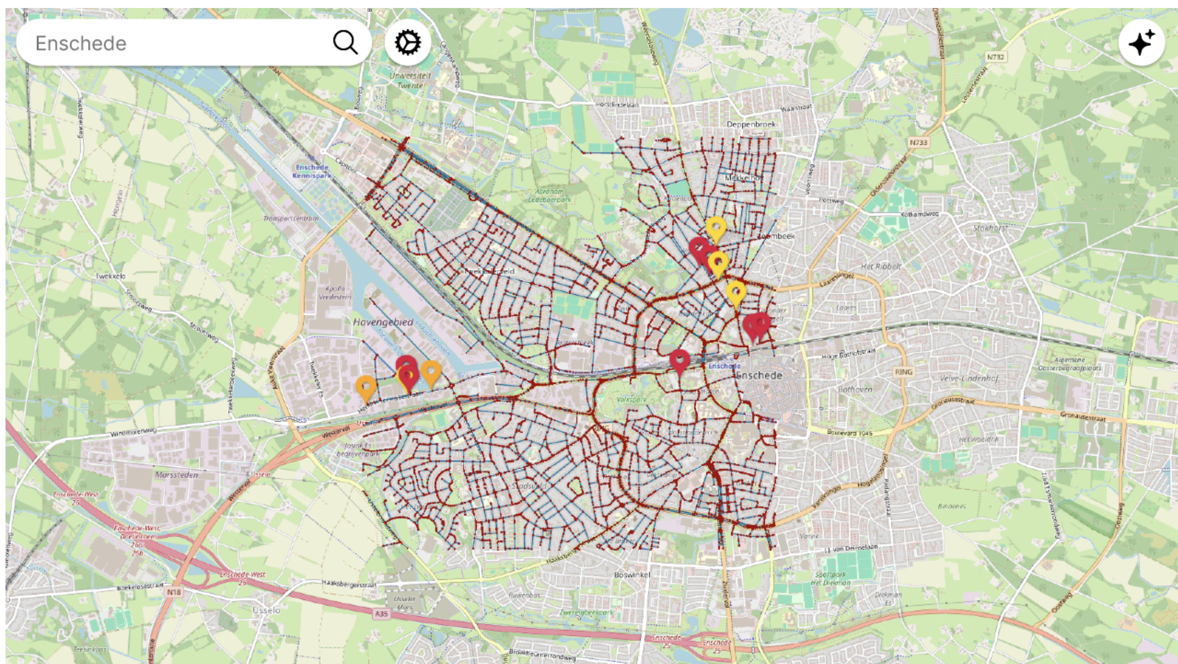


Figure C.1: Mock-up screen 1: Default map view on application launch.

Figure C.2: Mock-up screen 2: Chat interface with agent selector.

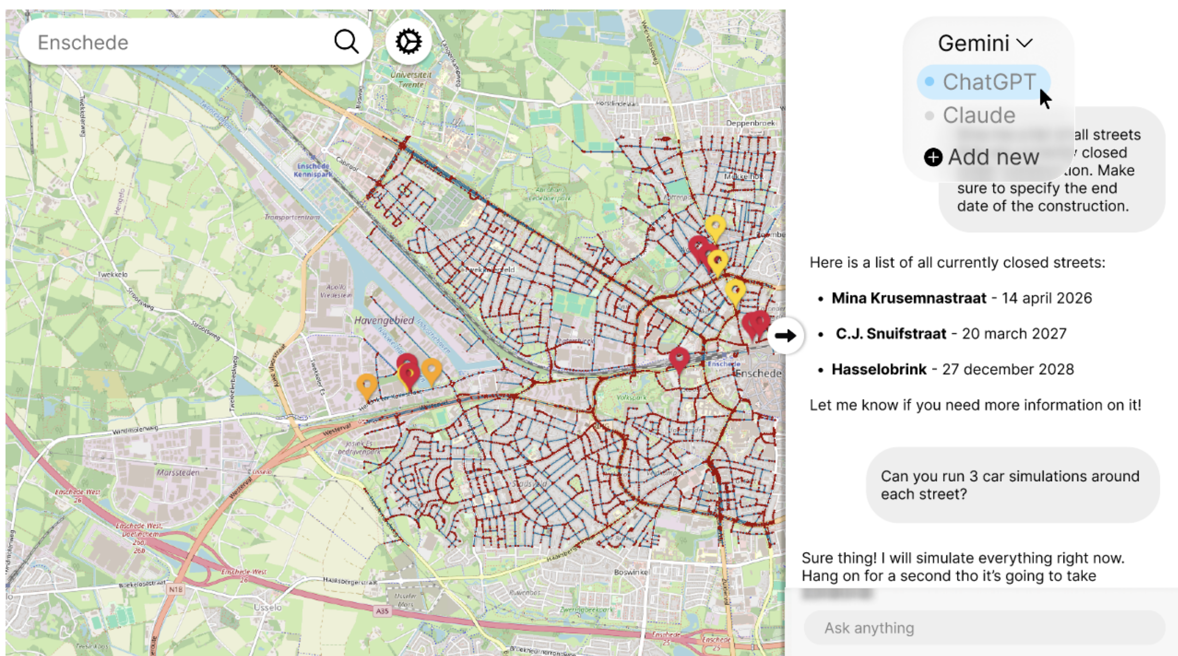


Figure C.3: Mock-up screen 3: Agent switcher dropdown showing available agents.

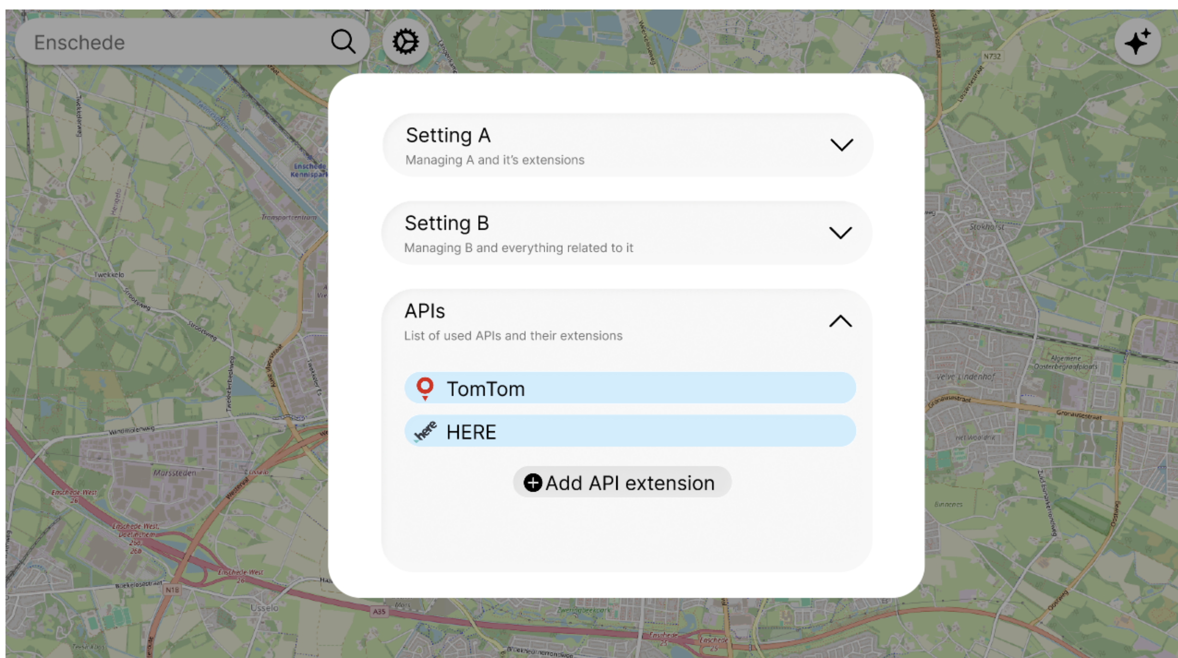


Figure C.4: Mock-up screen 4: Settings panel with API extensions expanded.

Appendix D

Diagrams

The following figures present the three design diagrams discussed in Section **??**.

Figure D.1: Use case diagram for MerlinRoads AI, showing the User and System Administrator actors and their associated use cases.

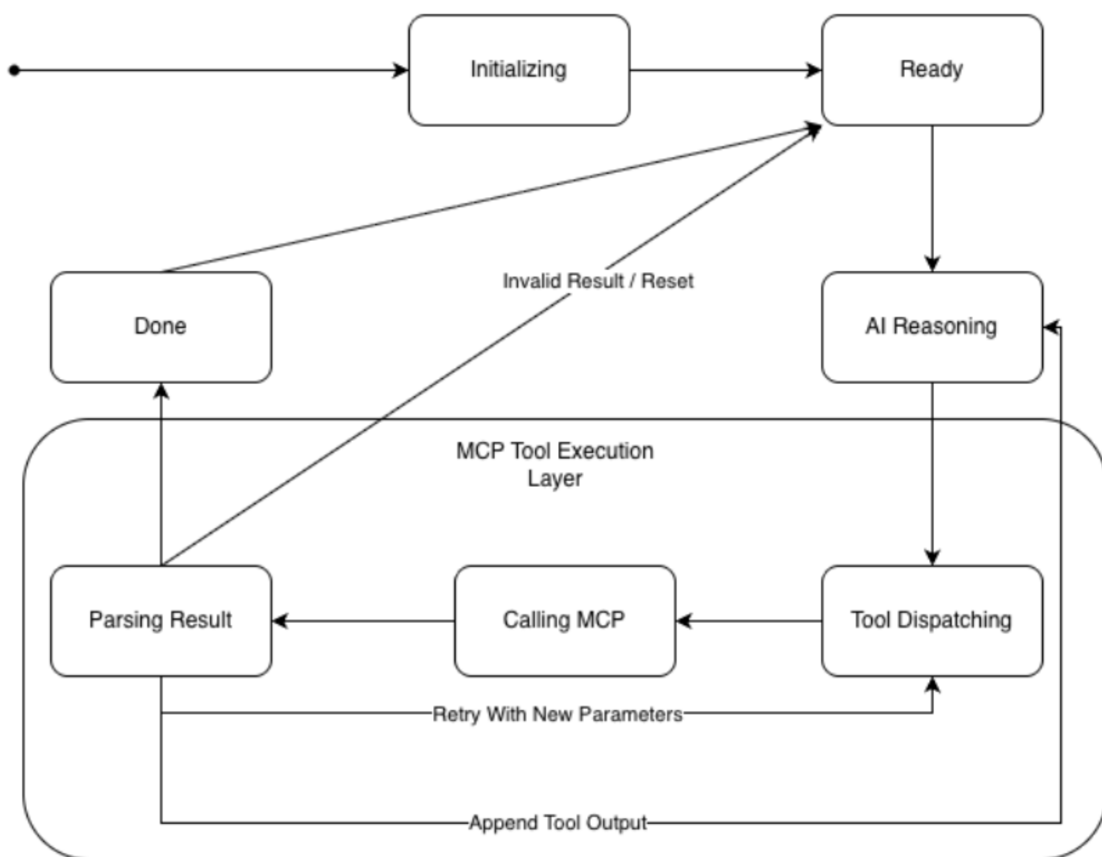


Figure D.2: State machine diagram describing the internal lifecycle of an agent workflow execution.

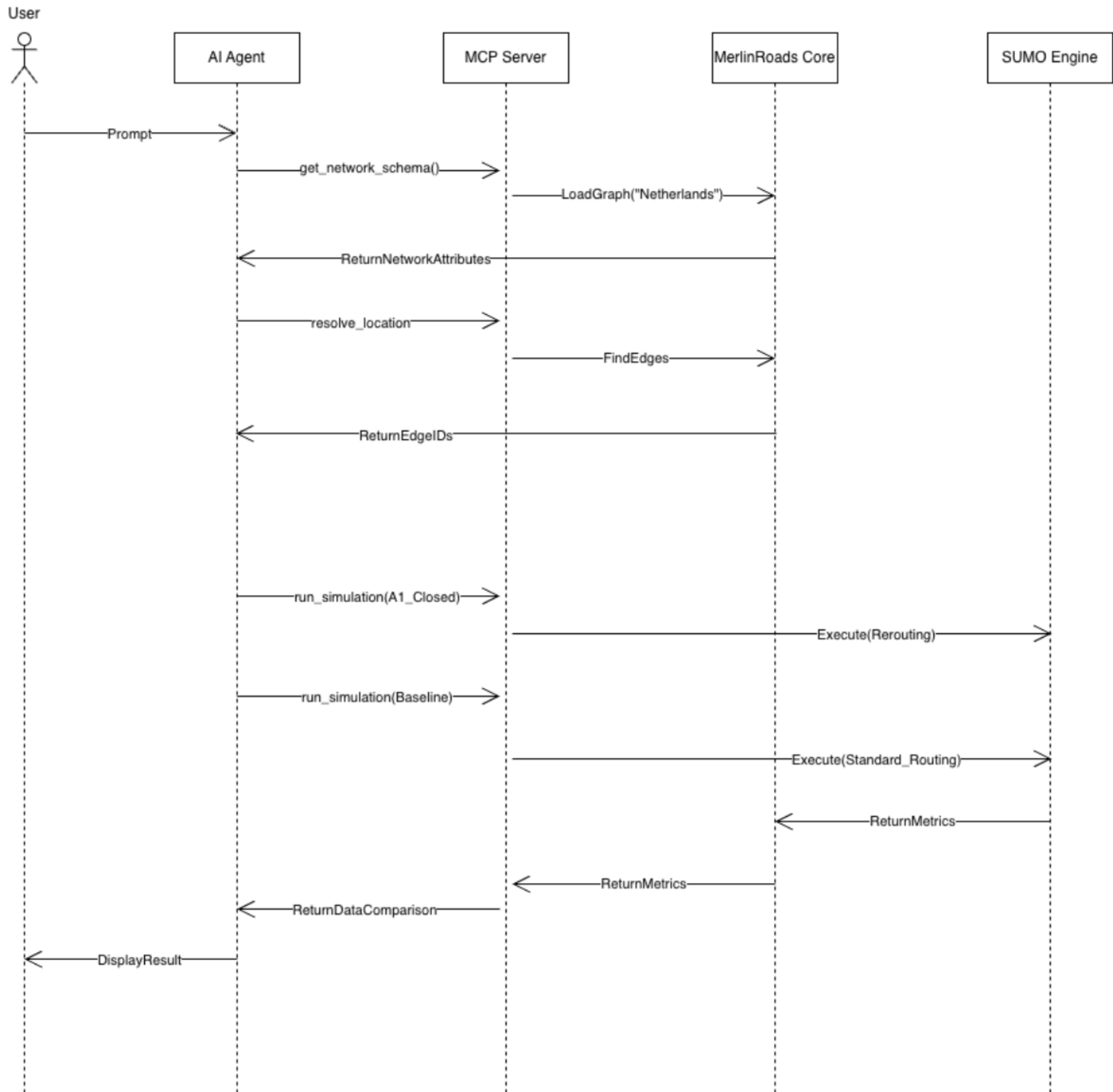


Figure D.3: Sequence diagram tracing the alternative route analysis scenario from Enschede to Amsterdam with the A1 motorway closed.

Appendix E

AI statement

The present project has been done using no artificial intelligence systems at all, regardless of whether for writing code, writing written texts, making design choices, doing analysis on data, testing and other tasks. All implementation, documentation, diagrams, and reporting have been done manually by the project's participants.

Bibliography

Dhandala, N. (2026, January). Python plugin systems. *OneUptime*. https://oneuptime.com/blog/post/2026-01-30-python-plugin-systems

PIARC Road Safety. (2025). Road safety manual. https://roadsafety.piarc.org

SUMO - simulation of urban MObility. (2024). German Aerospace Center (DLR). https://sumo.dlr.de

te Poel, J., Damink, N., van der Linde, R., Miedendorp de Bie, T., Rudzitis, E., & Bakanas, V. (2025). *Interactive road network accident visualization dashboard* (tech. rep.). University of Twente.

World Health Organization. (2023). Global status report on road safety 2023. https://www.who.int/publications/i/item/9789240086517