CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Designing a Graphical User Interface for Energy Prediction and Data Analysis for Volvo Group

Bachelor's thesis in Computer Science and Engineering

Noa Cavassi
Kevin Collins
Max Dreifeldt
Adam Kvarnsund

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Bachelor's thesis 2025

Designing a Graphical User Interface for Energy Prediction and Data Analysis for Volvo Group

Noa Cavassi
Kevin Collins
Max Dreifeldt
Adam Kvarnsund

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Designing a GUI for Energy Prediction and Data Analysis for Volvo Group
Noa Cavassi Kevin Collins Max Dreifeldt Adam Kvarnsund

Supervisors: Robin Adams, Department of Computer Science and Engineering
Examiners: Arne Linde and Patrik Jansson, Department of Computer Science and Engineering
Graded by teacher (Rättande lärare): Thommy Eriksson, Department of Computer Science and Engineering

Bachelor's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Abstract

This Bachelor's Thesis report presents the development of an application that uses Volvo Group's Energy Prediction Algorithm (EPA) to simulate electric truck journeys and support users with data analysis. The target audience, primarily project managers at Volvo Group, can run simulations with custom truck and route configurations, enabling data-driven decision-making. The application can also be used by engineers and data scientists, forming the secondary target audience, to seamlessly carry out simulations and analyze the results to support their daily operations. The project focuses on maximizing usability by applying relevant user interface design theory, including design principles and patterns. The application is refined iteratively through continuous feedback from a user group representing the target audience. This approach ensures that theoretical insights guide the user experience design while the consistent dialogue with users keeps the application aligned with their needs.

Keywords: Graphical User Interface, Design, User Experience, Analytics, React, TypeScript

Sammandrag

Detta kandidatarbete presenterar utvecklingen av en applikation som använder Volvo Groups Energy Prediction Algorithm (EPA) för att simulera resor med eldrivna lastbilar och stödja användare med dataanalys. Målgruppen, främst projektledare inom Volvokoncernen, kan köra simuleringar med anpassade lastbils- och ruttkonfigurationer, vilket möjliggör datadrivet beslutsfattande. Applikationen kan också användas av ingenjörer och dataanalytiker, som utgör den sekundära målgruppen, för att smidigt utföra simuleringar och analysera resultaten för att stödja sitt dagliga arbete. Projektet fokuserar på att maximera applikationens användbarhet genom att tillämpa relevant teori inom användargränssnittsdesign, inklusive designprinciper och designmönster. Applikationen förfinas iterativt genom kontinuerlig återkoppling från en användargrupp som representerar målgruppen. Detta tillvägagångssätt säkerställer att teoretiska insikter vägleder designen av användarupplevelsen, samtidigt som den löpande dialogen med användarna gör att applikationen uppfyller deras behov.

Nyckelord: Användargränssnitt, Design, Användarupplevelse, Dataanalys, React, TypeScript

Acknowledgments

We would like to thank our Chalmers thesis supervisor, Robin Adams, for providing us with invaluable guidance and feedback throughout the course of the project. We would also like to thank our Volvo Group supervisor, Simon Sundström, for his continuous support. Special thanks to Sudip Sarkar for making the thesis happen. Finally, we would like to thank all members of our user group who helped shape the development of this project.

Noa Cavassi, Kevin Collins, Max Dreifeldt, Adam Kvarnsund,
Gothenburg, June 2025

Declaration of Authors' Interest

The authors disclose that this bachelor's thesis project was conducted in collaboration with Volvo Group. Financial compensation may be received by the authors, subject to approval of the thesis by the university and subsequent assessment by Volvo Group of the study's value to the company. Volvo Group employees have contributed by providing user feedback during the development process. The authors have also been granted access to Volvo resources, including on-site premises and computers to support the execution of the project. All design decisions, analyses, and conclusions presented in this report have been made independently by the authors, based on academic sources and feedback received from the user group.

List of Abbreviations

API	Application Programming Interface. A set of definitions and protocols for building and integrating application software, used to interact with external systems like the Volvo Energy Prediction Algorithm (EPA).
CSS	Cascading Style Sheets. A language used to describe the presentation of HTML documents, defining layout, color, fonts, etc.
EPA	Energy Prediction Algorithm. An internal Volvo Group algorithm used to simulate and predict energy consumption for electric trucks based on parameters like vehicle configuration and route topography.
GUI	Graphical User Interface. An interface that allows users to interact with electronic devices through visual indicators and graphical icons.
HMR	Hot Module Replacement. A development feature that updates modules in a running application without a full reload, speeding up development feedback loops.
HTTPS	Hypertext Transfer Protocol Secure. A secure protocol used for transmitting hypermedia documents, such as web pages, between clients and servers.
KOLA Variants	Konstruktionsdata Lastvagnar. Internal Volvo identifiers for specific truck parts or configurations, used as parameters in the EPA API.
KPI	Key Performance Indicator. A quantifiable measure used to evaluate success or performance over time for specific objectives.
NDA	Non-Disclosure Agreement. A legal contract that restricts the sharing of confidential information with third parties.
PO	Product Owner. The role responsible for defining the vision of the product and ensuring the team delivers value to the business.

UI	User Interface. The space where interactions between users and a digital system occur, encompassing elements like buttons, forms, and navigation.
UX	User Experience. The overall experience of a user interacting with a product, particularly focusing on ease of use, efficiency, and satisfaction.
UX Debt	Accumulated issues in a user interface or experience resulting from suboptimal design decisions, often made for short-term gains at the cost of long-term usability.

Contents

List of Figures

List of Listings

1
Introduction

Climate change is one of the greatest societal challenges of our time, already impacting ecosystems, human health, food security, and economic systems across the globe. The global average temperature has already increased by more than 1°C compared to pre-industrial levels, and continued greenhouse gas emissions risk causing increasingly severe and irreversible consequences [1, p. 42]. To limit warming to well below 2°C, and ideally to 1.5°C, as established in the Paris Agreement, rapid and far-reaching reductions in emissions are required across all sectors of society [2, p. 3].

In 2019, the transport sector accounted for approximately 15% of global greenhouse gas emissions and around 23% of global energy-related CO_2 emissions, with emissions from the sector continuing to rise [3, p. 1056]. Within road transport, heavy-duty vehicles—such as trucks—are particularly emission-intensive. In Europe in 2020, heavy trucks were responsible for about 28% of road transport emissions, despite representing only 2% of the vehicle fleet [4, p. 1]. This imbalance has led to growing attention on reducing the climate impact of freight transport, with electrification increasingly highlighted as a key solution.

Electric trucks offer the potential to significantly reduce operational emissions, especially when powered by electricity from renewable sources [5]. At the same time, electrification introduces new technical challenges, such as limited range, long charging times, and high sensitivity to driving style and load weight. To enable the efficient use of electric trucks in real-world operations, new digital tools are needed, such as tools that can predict energy expenditure along a specific route for a particular truck configuration and support the planning of routes and charging strategies.

Volvo Group has developed an algorithm that predicts the energy consumption of an electric truck based on factors such as route topography, vehicle configuration, cargo weight, and current State of Charge. To make this algorithm accessible and useful to a broader range of users within an organization, it is important to provide an interactive and well-structured interface that allows users to explore different configurations and interpret the results effectively.

This bachelor's thesis aims to develop a graphical user interface (GUI) for Volvo's energy prediction algorithm. The interface will provide a comprehensive visualization of the algorithm's calculations, enabling users to effectively analyze predicted energy consumption and make informed decisions.

1.1 Purpose

The purpose of this project is to develop a user-friendly application that visualizes energy consumption for electric trucks in an accessible and comprehensible manner. The application aims to simplify the interpretation of large amounts of simulation data generated by Volvo Group's energy prediction algorithm (EPA)—data that is often extensive and presented in a format that can be difficult to survey. The goal is to make this data readily available to all project managers, regardless of technical background, in order to support effective decision-making. Additionally, the secondary target audience, consisting of employees with more technical roles, such as engineers and data scientists, should also be able to use the application to streamline their workflow.

Through an intuitive interface, users should be able to explore and quickly compare energy consumption across different routes and vehicle configurations. For example, it should be possible to vary vehicle types, battery counts, or the number of electric motors to gain a clear understanding of how different choices affect energy use. The design will follow established principles of good user experience, with a focus on simplicity, clarity, and efficient interaction with simulation results.

A central priority throughout the project is to place the needs and workflows of end users at the forefront of the design process. From early planning to final evaluation, decisions regarding functionality, layout, and interaction have been guided by direct feedback from users, ensuring the application remains intuitive and practical in real-world contexts.

1.2 Goals

The main goal of this project is to develop an application that is easy for Volvo employees to use. The application should make it straightforward for users to compare different trucks on the same route, without requiring technical assistance. The interface must be fast, intuitive, and accessible to project managers and other staff members.

Throughout the project, these goals have evolved dynamically in response to feedback and changing priorities from stakeholders and the user group, ensuring that the application meets Volvo's requirements and remains closely aligned with real-world needs. To make the objectives more concrete and actionable, the following four subgoals have guided the development:

- Develop a user-facing frontend application that interacts seamlessly with Volvo's internal energy prediction API. The solution should handle communication with the backend securely and efficiently, providing a robust foundation for future enhancements.

- Create an intuitive and accessible user interface, enabling project managers to configure and run simulations without technical assistance. Provide meaningful labels, tooltips, and feedback to minimize user errors. Minimize the number of steps required for common workflows, such as truck and route comparison. Continuously incorporate feedback from user groups and stakeholders into the design.
- Support robust scenario analysis by enabling users to compare different trucks on the same route. The application should allow quick reconfiguration of parameters and provide immediate visualization of key performance indicators. Results should be aggregated and presented to support both a high-level overview and a detailed inspection.
- Ensure the application aligns with Volvo's internal practices and recommendations, adherence to IT policies, and compliance with established integration standards. This involves implementing maintainable and well-documented solutions that accommodate current constraints, such as limited API access.

1.3 Scope and Limitations

The project focuses solely on the development of the user interface and does not include any validation or further development of EPA, the underlying energy prediction algorithm.

The application enables simulations with a wide range of truck configurations. However, it is not possible to adjust every individual component—only those parts of the truck that are deemed to have a noticeable impact on energy consumption. Features such as color, side mirrors, or interior design are not considered.

Not all truck components are necessarily compatible with each other, meaning that certain configurations available in the application may not reflect technically valid combinations. However, EPA does not account for this type of compatibility and will still perform calculations even with invalid input. Since no accessible or complete rulebase defining component compatibility is available, we have, in consultation with supervisors from both Volvo Group and Chalmers, chosen to leave to the user the responsibility of knowing which configurations are technically feasible.

The application is designed primarily for desktop environments and has not been optimized for mobile or tablet use. It is intended to run on Volvo Group computers, where it performs well across a range of common screen sizes and window dimensions. While the interface adapts to smaller desktop screens and resized windows, responsive design for mobile devices was not a focus in this version. This could be explored in future iterations to support broader accessibility.

1.4 Constraints

This section outlines the main constraints that influenced the development and evaluation of the application, including confidentiality restrictions and limitations related to the availability of the user group.

1.4.1 Confidentiality Constraints

Due to confidentiality agreements, the source code cannot be shared externally, and the application is intended for internal use within Volvo Group. As such, it can only be run in environments with access to the company's network and is not available for public distribution or testing outside the organization.

1.4.2 User Group Availability

The chosen user group is primarily made up of project managers. This target audience was chosen as the application's features align well with their work processes. However, their demanding responsibilities often mean busy schedules, making it challenging to secure consistent, in-depth feedback. To address this concern, individuals, as long as they were part of the target audience, were added to the user group throughout the process to make sure the user group is large enough. In turn, this increased the likelihood of valuable feedback and also represented a larger variety of perspectives [6, p. 215].

2
Background

To understand the system developed in this project, it is important to consider both the practical context in which it is used and the technical and design principles that have guided the development process. This chapter begins by describing the problem the application intends to address, with a particular focus on how Volvo's current solution works in practice and the challenges users face. This is followed by a review of relevant principles for user-centered design as well as an overview of the technical choices made during the project. Together, these elements provide the background necessary to understand both the purpose of the project and the approach taken during development.

2.1 User Needs and Problem Description

Volvo Group's EPA has already been developed and is available for use. However, there is currently no convenient or accessible interface for interacting with the algorithm. At present, users need to manually send HTTPS requests to appropriate endpoints, which requires technical knowledge and is both time-consuming and cumbersome. Furthermore, in order to make a request, the user must first retrieve an authentication token that is only valid for one hour, resulting in frequent interruptions to the workflow.

Modifying a single vehicle component or a segment of a route often requires several input parameters in the API request to be manually adjusted, such as coordinates for stops along the route or specific component codes. Components in EPA are specified using so-called KOLA variants, which are internal labels for different truck parts. These variants are not self-explanatory, and there is a lack of documentation explaining what each KOLA variant represents, further complicating the use of the system.

The response returned from the API is also difficult to interpret. While the results are deserialized from long Protocol Buffer (Protobuf) objects, Google's platform-neutral mechanism for serializing structured data [7], into a readable format, they still lack the structure needed for quick and intuitive understanding. The information is presented as large blocks of text without any clear visual hierarchy, making it challenging for users to gain an overall understanding of the results.

Another challenge concerns the variation in users' technical backgrounds and working conditions. Some project managers have sufficient technical experience to eventually learn how the API works and use it independently, while others lack either the technical knowledge or the time required to do so. Regardless of background, the workflow involved in using EPA is so time-consuming and cumbersome that the task is often delegated to other colleagues, such as developers or data analysts. This creates unnecessary dependencies and additional steps in the process, limiting accessibility and preventing EPA from being used as efficiently as it could be.

An additional limitation is the lack of adequate documentation for the API. It is often unclear which parameters are required, which are optional, and how they should be structured. The absence of clear and up-to-date documentation creates a high barrier to entry for new users, making it difficult to expand internal usage of the algorithm. It also increases the risk of incorrect API usage, which, in turn, may lead to misleading results.

There is a clear need for a more accessible and user-friendly way to interact with EPA. A graphical user interface (GUI) could eliminate many of the manual and technical barriers associated with the current API-based workflow. By providing a visual and interactive presentation of both input and results, a GUI would enable project managers to perform and interpret simulations independently, without the need for delegation or intermediaries. This would streamline the workflow and increase both the usage and the value of EPA within Volvo Group.

2.2 Design Patterns and Structure

Patterns for organizing information and structuring interactions to help navigate the user's focus play a central role in creating a user-friendly interface [8]. By reusing proven solutions, it is possible to build a logical structure where users can quickly find what they need and complete their tasks with minimal friction. Below is a selection of design patterns, including layout strategies, interaction structures, and stylistic elements such as typography and color, that were explored and implemented. The following patterns are all described in [9].

2.2.1 Dashboard

A common and well-established design pattern for presenting important information is the dashboard. It is frequently used as a landing page and is characterized by its ability to gather key data points, graphs, and other relevant information in a compact view [9, p. 78]. This pattern is popular because it provides users with a quick overview of the current status, key metrics, or tasks to be addressed. Since dashboards are a familiar format with well-established conventions, users can easily interpret the content without needing to learn a new interface.

2.2.2 Center Stage

Another relevant design pattern is Center Stage, which emphasizes placing the primary content or task "front and center" in the user interface [9, p. 231]. This layout ensures that the main purpose of the screen is immediately visible and unambiguous, guiding the user's eyes directly to the most important information or interaction. It is well suited for interfaces where a single unit of coherent content, such as a table, map, or data visualization, needs to be the focal point [9, p. 231]. By anchoring attention to a central entity, secondary content can be interpreted in relation to what is in the center, reducing confusion and helping the user understand the structure and flow of the interface more intuitively.

2.2.3 Accordion

The Accordion pattern organizes content into stacked, expandable panels, allowing users to show or hide sections of interface elements as needed [9, p. 245]. Accordions are useful when there is too much heterogeneous content to fit on a single page, and when users may want to view multiple modules simultaneously. They are commonly used in tool palettes or navigation systems to declutter the interface while maintaining access to related elements [9, p. 245].

2.2.4 Visual Framework

A Visual Framework is a design pattern that ensures consistency across all pages or views in an application by reusing key stylistic and structural elements, such as layout, font choices, colours, and navigation placement [9, p. 228]. This repetition helps users feel oriented and confident when moving between different screens, since they do not need to relearn the interface each time the context shifts. A strong visual framework also makes the interface feel cohesive and intentionally designed, supporting both usability and branding [9, p. 228]. Even if the content changes between views, the consistent design helps everything feel like it belongs to the same application.

2.2.5 Cards

The Cards pattern is a flexible way to display collections of content in self-contained units, each typically containing a mix of text, values, and sometimes images or actions [9, p. 353]. Cards are especially useful when presenting a list of items with a consistent structure but variable content, like metrics or graphs, because they adapt well to different screen sizes and layouts. Cards are widely recognized from mobile and web interfaces, which makes them familiar to users and easy to scan when displaying key information in a dashboard context [9, p. 353].

2.2.6 Typography and Color

Typography and color are essential tools for creating interfaces that are both readable and easy to navigate. Sans serif fonts are generally preferred in screen-based design because they maintain clarity at small sizes [9, p. 267]. Serif and display fonts, while useful for headings or branding, tend to lose legibility when scaled down and are therefore used more sparingly in user interfaces [9, p. 268].

The importance of color in supporting both usability and structure is not to be understated. High contrast between text and background—such as dark text on a light background or vice versa—is crucial for readability and accessibility [9, p. 259]. While saturated colors can help draw attention to key elements, they should be used selectively [9, p. 263]. A more neutral color palette is often better suited for reducing visual fatigue and keeping the interface clean. Color should, however, not be the sole method for conveying information; combining color with icons or text ensures accessibility for users with visual impairments [9, p. 264].

2.2.7 Components

Modern interface design relies heavily on reusable components to create consistent, extensible, and efficient user experiences [9, p. 534]. A component-based approach allows designers and developers to construct user interfaces from modular building blocks, ranging from small elements like buttons and input fields to more complex structures like forms or full templates. These components can be combined and reused across screens, helping teams avoid code-centric "UX debt" where inconsistencies arise from duplicated styling logic, such as copied CSS code that is gradually modified independently and causes visual mismatches across similar components [10, p. 3].

Using components also promotes speed and maintainability. Rather than designing every screen from scratch, designers can rely on shared component libraries that allow for consistent styling and faster development. This is part of a broader systems-thinking approach, sometimes referred to as Atomic Design, where interfaces are built from the bottom up: starting with atoms (simple elements), then grouping them into molecules (groups of elements), organisms (simple UI components), templates (sections of the page), and finally complete pages [11].

By standardizing both form and behavior, component-based UI systems reduce design variation, simplify updates, and ensure a more predictable experience across different parts of an application. As a result, end-users benefit from improved usability and greater clarity, while development teams can reuse code more effectively.

2.3 Design principles and Coding Practices

In addition to established design patterns, the development of user-friendly interfaces benefits from widely recognized design principles and coding practices. This section outlines selected usability heuristics and software development principles that served as the theoretical foundation for later design choices. The heuristics presented below are adapted from Jakob Nielsen's widely used framework of ten usability heuristics for user interface design [12].

- **Visibility of System Status:** The interface should always keep user informed about what is going on, through appropriate feedback within reasonable time. For example, loading indicators or confirmation messages help understand the state of the system [13].
- **Consistency and standards:** Users should not have to wonder whether different words, actions, or situations mean the same thing. Interfaces should follow platform conventions and standards, and apply them uniformly throughout the application [14].
- **Recognition rather than recall:** Interfaces should minimize the user's memory load by making elements, actions and options visible. The user should not have to remember information from one part of the interface to another [15].
- **Aesthetic and minimalist design:** Interfaces should avoid unnecessary elements and focus only on relevant information. Every extra unit of information competes with the relevant units and diminishes their visibility [16].

In addition to these usability heuristics, good software design also relies on proven coding practices that promote maintainability and extensibility.

A central idea is **Separation of Concerns**, which refers to structuring software in a way that separates different responsibilities into distinct parts of the system. This separation helps manage complexity, simplifies testing and debugging, and makes it easier to adapt or extend specific parts of the system without unintended side effects [17, pp. 312–313].

Several well-established object-oriented design principles also contribute to sustainable and extensible software development. Among these, a few of the SOLID principles, as described by Martin in *Clean Architecture*, are particularly relevant in guiding architectural decisions [18].

The **Single Responsibility Principle (SRP)** says that a module or class should only have one reason to change. This helps make the code easier to understand and change later, because each part is focused on doing just one thing. When responsibilities are mixed together, it often leads to more bugs and harder maintenance, especially when one change affects things it should not [18, pp. 61–68].

The **Open–Closed Principle (OCP)** means that code should be written in a way that allows new features to be added without changing existing code. Instead of editing old logic, developers can add new classes or modules to handle new behavior. This helps protect working code from breaking when the system grows [18, pp. 69–76].

The **Dependency Inversion Principle (DIP)** means that both high-level and low-level parts of the code should depend on abstractions, like interfaces, instead of depending directly on each other. This helps keep important parts of the system independent from details like frameworks or databases, which might change more often. It also makes the system easier to change or test later, since the core logic does not rely on specific implementations [18, pp. 87–92].

2.4 Underlying Technologies and Libraries

The implementation of the graphical user interface to utilize EPA was based on a modern web development stack designed to support maintainability, extensibility, and modularity. This section presents the tools and frameworks used throughout the development process.

2.4.1 Programming Language

TypeScript

TypeScript [19], a statically typed superset of JavaScript [19], was used to enhance code clarity and reduce runtime errors. Static typing enabled early detection of bugs and facilitated easier refactoring than using a dynamically typed language like JavaScript.

2.4.2 User Interface Frameworks

React

The frontend was built using React [20], a JavaScript library for building user interfaces. React's component-based architecture made it suitable for encapsulating visual and functional modules such as configuration panels, result visualizations, and interactive maps.

Volvo Design Component Library

To align with Volvo's internal design guidelines and ensure a consistent user experience, the project utilized Volvo's proprietary design system. This React-based library included pre-styled components such as buttons, forms, and modals, and was accessed via an internal registry requiring authentication.

Tailwind CSS

Styling was implemented using Tailwind CSS [21], a utility-first framework that directly allowed for rapid and consistent layout definition within component files. Tailwind facilitated a streamlined development process by eliminating the need for large custom CSS files.

2.4.3 Development and Tooling

Vite

Vite [22] was selected as the frontend build tool and development server. It provided fast hot module replacement (HMR), minimal configuration, and efficient production builds, making it well-suited for modern projects that utilized TypeScript and React.

ESLint and Prettier

Code quality and consistency were enforced using ESLint [23] and Prettier [24]. ESLint was configured to detect potential issues and enforce stylistic rules, while Prettier ensured consistent formatting throughout the codebase.

Figma

Interface prototypes and design flows were created in Figma, a design tool created for collaboratively working on mock-ups and prototypes. [25]. This tool was used to visualize application structure and gather feedback during the early design stages, enabling alignment with project stakeholders and efficient iteration.

2.4.4 Containerization and Runtime Environment

Node.js

Node.js [26] is an open-source JavaScript runtime built on the V8 [27] engine. It allows JavaScript to run outside the browser and was used to power a local proxy server, facilitating secure communication between the frontend and backend services.

Docker

The application was containerized using Docker [28]. A Dockerfile and accompanying docker-compose.yml enabled environment consistency. This ensured that external dependencies and configurations were abstracted away from the host system.

2.4.5 Source Control

GitHub

GitHub [29] was used for version control. The repository was used to support agile workflows, including feature branching, pull requests, and issue tracking. GitHub Projects and issue boards helped coordinate development and sprint planning.

3
Method

The project began with a planning phase, during which the target audience's needs were explored and a problem description was formulated. In parallel, research into design principles and patterns (Section 2.2) and a suitable technology stack (Section 2.4) was conducted. A workflow with recurring feedback sessions was also designed, following Agile principles. After this phase, the solution — a web application — was iteratively designed and developed based on continuous user feedback. This ensured that the user group regularly evaluated the application and fine-tuned the problem description to better match their needs. Throughout the process, attention was also given to maintaining a clean code architecture and effectively utilizing various libraries to build the application.

3.1 Agile Workflow

During the planning phase, it was decided to adopt an agile workflow for the development of the application. This allowed the product to be developed iteratively, continuously delivering features to satisfy customer requirements that might change throughout the process [30]. Since both the technical requirements and user needs were initially uncertain, an adaptive, user-centric approach was deemed most suitable. Elements from the Scrum methodology, such as structuring the development into sprints and introducing a product owner role, were incorporated and are further described in Section 3.1.1.

3.1.1 Sprint Structure

The development phase was organized into two-week sprints, where a sprint is defined as a "set of development activities conducted over a pre-defined period, usually one to four weeks" [31, p. 14]. Each sprint began on a Monday with a sprint planning session, during which the developers selected user stories from the backlog for that sprint. Each selected user story was estimated in terms of story points, a standard practice in agile development [32, p. 10], assigned to a developer, and prioritized based on discussions with the user group and Product Owner (PO). The PO's role was to maximize "the value of the product resulting from the work of the Scrum Team" [33], helping the team identify the most valuable features to develop. If too few user stories were available in the backlog, the sprint planning session was also used to write additional stories, particularly early in development. The user stories reflected requests from the user group, requirements from the PO, or features iden-

tified by the development team to address user-reported problems. An example of a user story is shown in Figure 3.1.

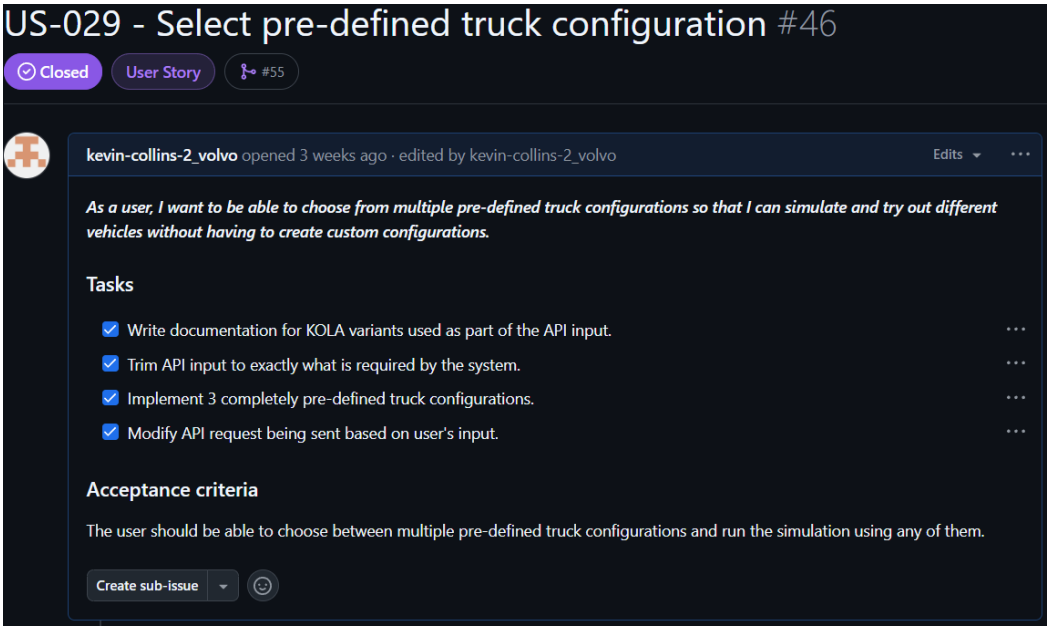


Figure 3.1: User story specifying the pre-defined truck configuration feature.

After one week from the sprint planning meeting, a mid-sprint meeting was held to serve as a status update and checkpoint. These meetings typically addressed administrative matters and major technical roadblocks, though specific implementation details or bugs were occasionally discussed.

Weekly meetings were also held on Tuesdays with Simon Sundström, who served both as the Volvo supervisor and Product Owner (PO). These meetings focused on reporting development progress, planning upcoming work, and ensuring that the product aligned with the user group's needs. In addition, Simon provided administrative support and acted as the primary contact point within Volvo.

At the end of each week, usually on Fridays, weekly meetings were held with the Chalmers supervisor, Robin Adams. Early meetings focused on discussing the project's scope, goals, and academic requirements. As the project progressed, the meetings shifted towards presenting prototypes for feedback, discussing features and implementation improvements, and occasionally addressing administrative matters related to the Volvo collaboration. In some cases, other parties participated when major administrative decisions, such as NDA agreements, were discussed.

Interviews with the user group were conducted approximately every other week, depending on availability. These sessions gathered feedback on the implemented features and allowed the user group to suggest additional functionality. This iterative feedback process was crucial to the application's development and is further described in Section 3.2.

Each sprint concluded with a sprint review, where the team discussed the sprint's progress, assessed how much development was complete, and evaluated whether the project remained on track to meet its goals. These meetings also included constructive discussions on areas for improvement to be addressed in the following sprints.

Immediately following each sprint review, a sprint retrospective was held to focus on improving the development process. Retrospectives typically followed the standard sprint retrospective format, using a whiteboard with three columns: 'start', 'stop', and 'continue' [34, p. 11], where each team member contributed feedback, followed by group discussion. At times, these sessions were held more informally as casual but still productive conversations. An example of a sprint retrospective is shown in Appendix A.

3.1.2 Tools Supporting the Agile Workflow

The codebase was hosted on Volvo Group's internal GitHub organization, and the repository was structured to support agile development. GitHub Projects was used to create a Kanban board, as shown in Figure 3.2, which served as the cornerstone of the development process.

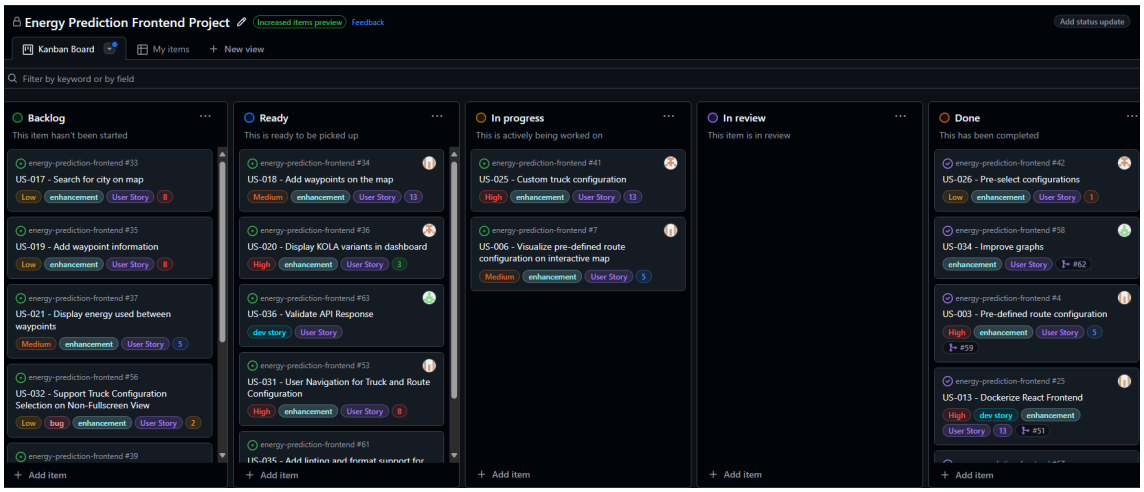


Figure 3.2: Snapshot of Kanban board during development.

When a user story was created, it appeared as a card in the 'Backlog' column. During sprint planning, selected user stories were assigned story points, prioritized, and moved to 'Ready'. As development progressed, cards moved through 'In Progress' and 'In Review', and upon approval and merging into the *develop* branch (which acted as the main branch), they were placed in 'Done'.

Once development of a user story was complete, a pull request (PR) was created from the feature branch into *develop*. A branch rule required an approved review before merging. After a reviewer was assigned, comments and annotations were provided to highlight any necessary changes, ranging from minor adjustments to

documentation or code style to larger refactoring. After the developer addressed the feedback, the reviewer would reassess the PR. This process was repeated as needed until the reviewer approved the changes, after which the PR was merged into *develop*.

In addition to GitHub, Google Drive [35] was used to store all project documentation, including sprint plans, meeting notes, project logs, interviews, and report drafts. This allowed real-time collaboration and maintained transparency within the group and with external parties who had access to the shared Drive. The only exception was when physical whiteboards were used for brainstorming during in-person meetings and sprint activities.

Trello [36] was used to track non-code-related tasks and deadlines, following a structure similar to the GitHub Kanban board to ensure that the team remained aware of ongoing administrative work.

3.1.3 Evolution of the Target Audience

Initially, the target audience for the application to be developed was project managers. However, during early discussions with the user group (who represented the target audience), their feedback signaled that engineers and data scientists could also benefit from the tool. After consultation with the PO, it was decided that project managers would remain the primary target audience, but more technical roles, such as engineers and data scientists, would be seen as a secondary target audience. This resulted in a higher chance that Volvo would use the application in day-to-day operations, but it also made the designing and development more challenging as more perspectives needed to be considered, given that a broader set of users influenced the application's interface and features.

3.2 Prototyping and Continuous Feedback

This section outlines the iterative prototyping process, starting from initial brainstorming to fully developed Figma prototypes, as well as the feedback process used to refine the application design.

3.2.1 Introductory Phase

Based on the initial discussions with the PO, a very high-level set of informal requirements/features was identified. This was used as a starting point to brainstorm ideas for how the GUI should look. Initially, the group restricted itself to paper sketches and brainstorming on whiteboards.

Once the group felt that everyone was somewhat on the same page, designing was shifted to Figma. The initial design work was done on both personal and shared workspaces, enabling parallel development.

Soon after initial Figma mock-ups were created, a session with the user group was scheduled to get some early feedback. At this stage, there was no unified Figma mock-up. Instead, each member had worked on different parts of the application. The designs also shared many elements such as a common color theme, an identical sidebar, an identical top bar, etc., which were designed and decided upon by the group together. However, due to the variations in the designs, the focus of the first feedback session was not on the graphical design, but rather on features the application should implement.

Following this, a synthesized Figma version was designed together by the group, heavily based on the literature review and research into design practices done by the group. At this point, some elementary "functionality" was added to the Figma project through the use of wireframes and transitions, slowly transforming the designed mock-up into a prototype.

3.2.2 Design and Prototyping Phase

The next session with the user group was conducted, and in order to make sure feedback about the graphical design was received, it was organized as a formal, semi-structured interview. This ensured that the focus remained on discussing the interface and revealed some unexpected needs from the user group. The prototype was designed with many design patterns in mind, and the goal was to have quite a modern look, similar to consumer software that most people are used to. However, the user group did not want a visually overloaded application; rather, they preferred a modest design that was simple and straightforward, not only in terms of usability, but also in appearance. They did not want images or anything similar taking up space, but rather a simple, clean user interface that minimized the number of clicks needed to do tasks.

This outlook heavily influenced the design of the last couple of screens of the Figma prototype. It was previously decided that the user group had the last say when it came to design decisions, as they would be the individuals using the developed application. To confirm that the user group's opinions were interpreted correctly, the new designs were presented to a couple of user group members outside of the feedback sessions to make sure that that was the sort of design they would prefer. After confirmation, the Figma design was made into a full-fledged prototype, which was presented to the PO and later, the user group, before development began.

To illustrate how this design philosophy was applied, Figure 3.3 shows the initial Figma mock-up for the pre-built trucks screen. Each truck is represented using a card ('Cards' design pattern described in 2.2.5) and spaced out neatly ('Grid of equals' design pattern [9, p. 235]). If the user would like more information about any of the pre-built truck configurations, the idea was that they could simply click on the card, and a modal panel with the necessary information would pop up.

During the user group session, they made clear that they were not fans of modal

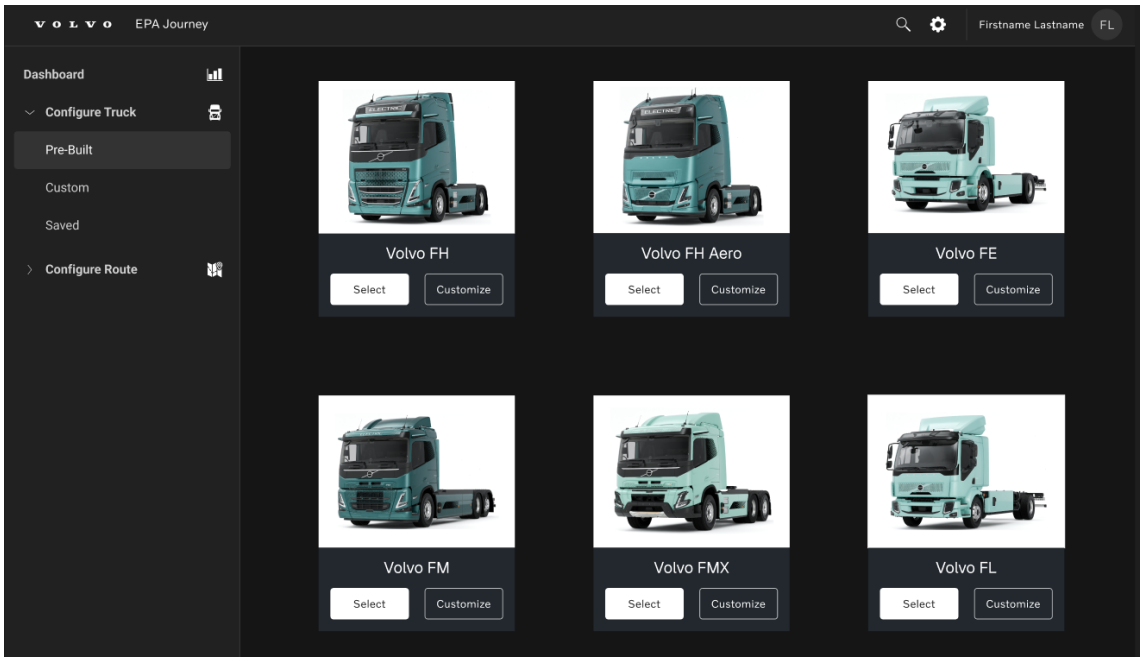


Figure 3.3: Initial pre-built trucks Figma mockup.

panels and did not like having the images of the truck models. Instead, they wanted some sort of user interface that just displayed all the information they needed about the truck configurations on the same screen, but still in a neat manner. To accommodate this feedback, another screen displaying saved truck configurations was designed as seen in Figure 3.4. The user group was asked whether they would like the pre-built truck configurations screen to have the exact same structure, with no images and just the information they need, and they responded with a resounding yes.

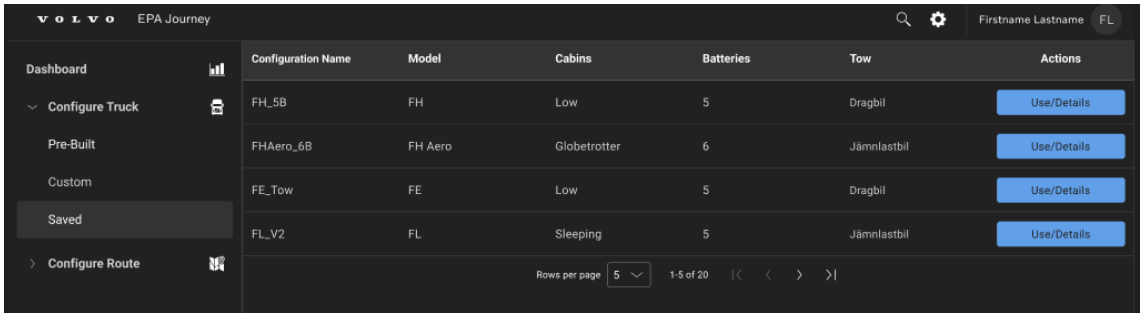


Figure 3.4: Saved truck configuration Figma mockup.

3.2.3 Development Phase

Once development of the web application began, interviews with the user group were conducted iteratively to receive feedback throughout the development phase. This sometimes resulted in the introduction of new features, but the main goal was to refine the interface and the application as a whole to best meet user needs. Feedback

was mostly gathered through formal user interviews, held approximately bi-weekly, and an example can be seen in Appendix B. Additionally, informal feedback sessions with the PO or just one or two members of the user group were conducted throughout the project lifecycle. The formal interviews typically consisted of 3-5 user group members and lasted for around an hour, as research suggests that this timeframe results in effective feedback sessions [37, p. 84].

Throughout the process, many requirements from the user group slightly changed after continued discussion through several weeks. At times, a couple of members even had clashing opinions, but they were usually able to come to some consensus. When no consensus could be reached, the decision was made based on foundational design principles and in consultation with the PO. Although many aspects of the GUI were modified and refined over time, one thing that never changed was the overarching design philosophy: to keep things simple, organized, and minimize clicks required to complete tasks. Another example of this can be seen when comparing the initial version of the custom truck configuration feature to the final version.

As seen in Figure 3.5, the process of configuring a custom truck is split into six steps. Each step requires the user to select one part of their custom truck, and the wizard-like structure enables them to seamlessly step through the process while knowing how much more is left due to the progress indicator. Additionally, the user would also be able to click on any of the steps in the stepper (the UI component at the bottom of the screen, which highlights the different steps) to jump between different parts of the configuration. This design was also inspired by Volvo's Truck Builder webpage [38], as seen in Figure 3.6. In their design, the large truck is a static image on all steps, and the user has to click on next or previous to navigate to different parts of the configuration process. The expectation was that the user group would be familiar with this page and be able to have a similar user experience.

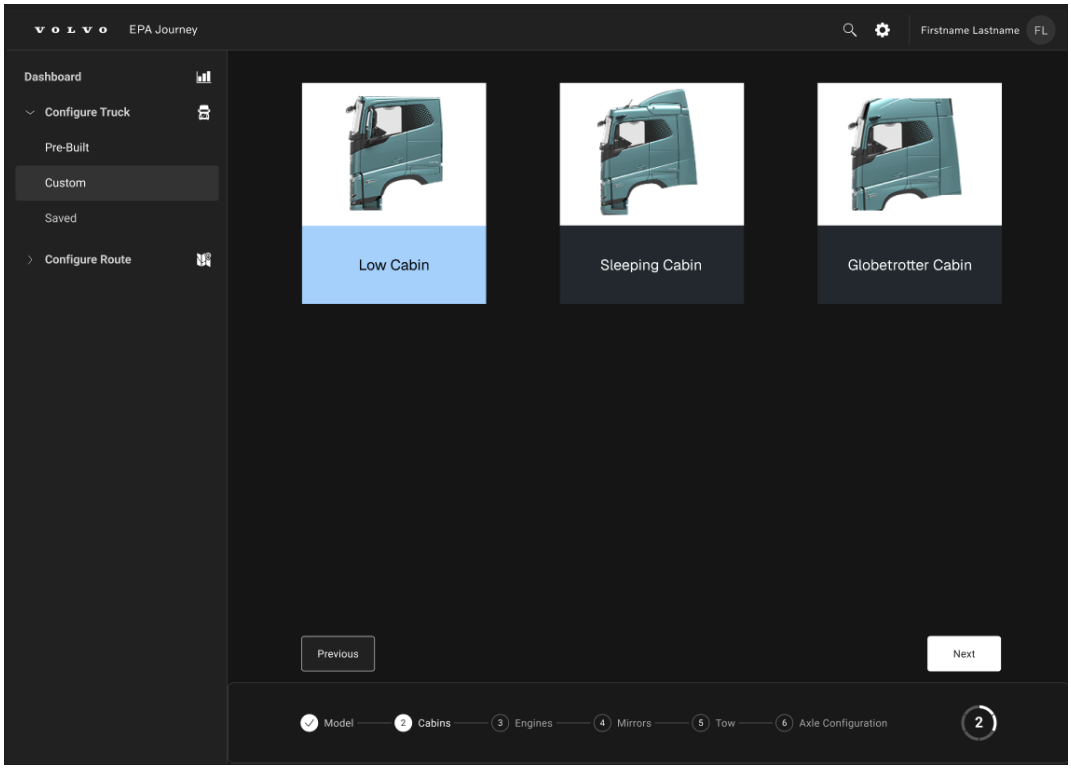


Figure 3.5: Custom truck configuration Figma mockup.

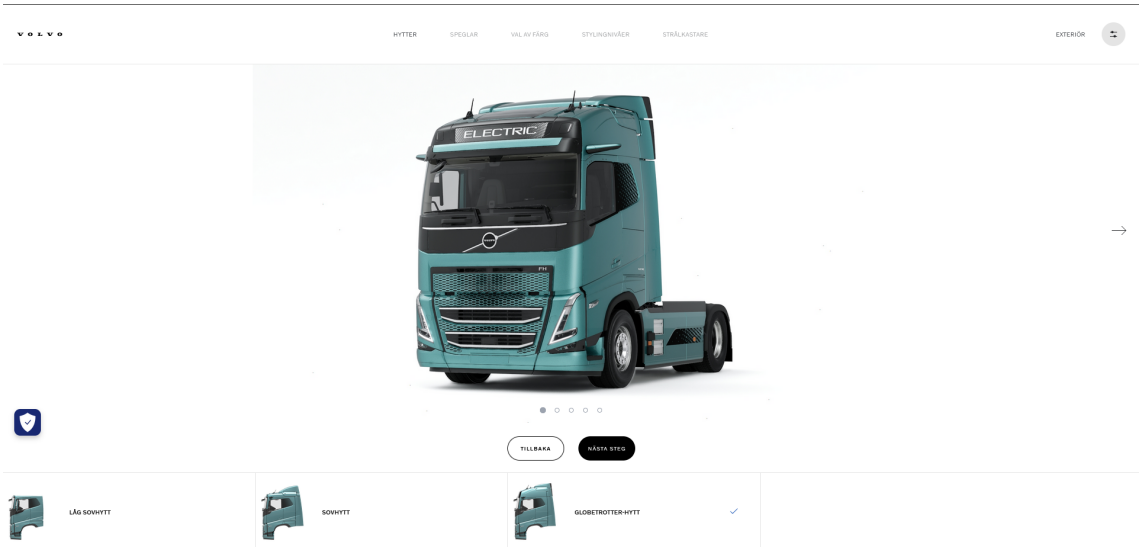


Figure 3.6: Snapshot of Volvo's Truck Builder webpage [38].

However, the user group was not quite happy with the user interface. As previously mentioned, they disliked the images and wanted to minimize clicks. They found the wizard-like process too cumbersome and were set on the fact that everything should be on one page. After multiple iterations and discussions, the final implementation of the custom truck configuration feature can be seen in Figure 3.7. Although not visually extravagant, this design addressed the two primary priorities of the user group: simplicity and enhancing efficiency. The user group was quite satisfied with the design and delighted at the fact that it took one click to configure each component, and no other intermediary clicks were required during the configuration process.

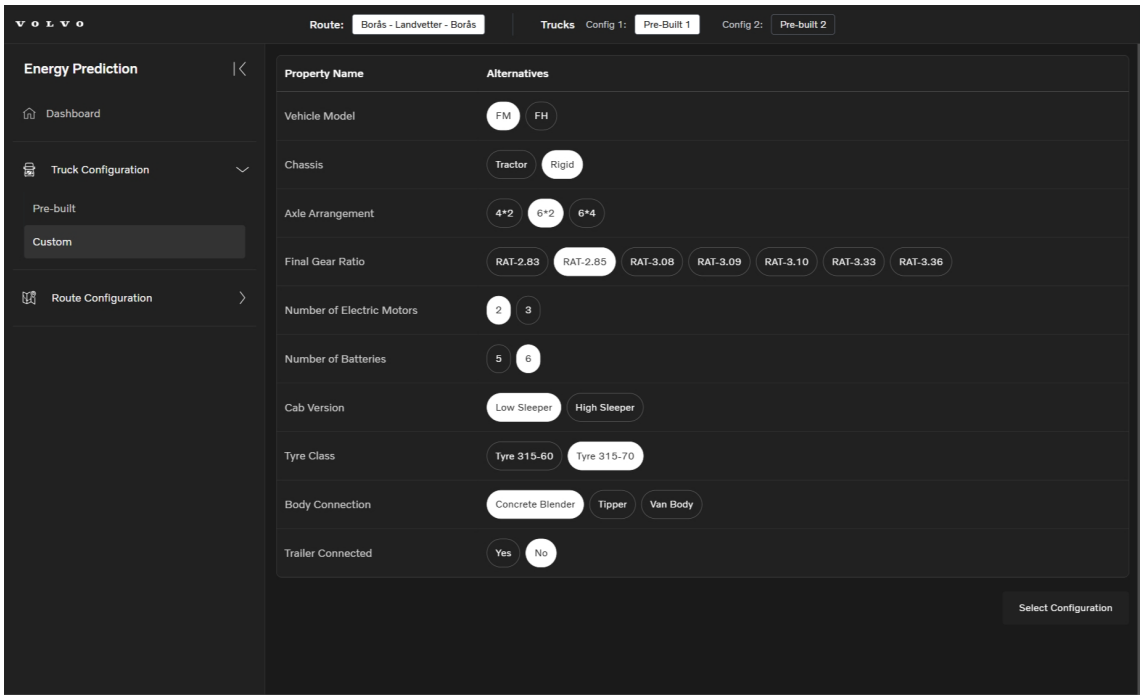


Figure 3.7: Final implementation of custom truck configuration.

Another part of the application that the user group was particularly interested in was its dashboard, especially regarding which information it displays. Since the data returned from EPA was quite complex, the idea of presenting critical metrics to summarize the simulation output was proposed. This can be seen as a form of data characterization and helps users quickly grasp key information [39, p. 4]. All user group members appreciated this approach, and in consultation with them, appropriate metrics were selected to be displayed on the dashboard.

For the visualization of detailed data, the initial request from the user group was to display multiple separate graphs, each showing a different metric, as seen in Figure 3.8. This was motivated by their desire to be able to view multiple simulation results simultaneously for easier comparison. However, after this solution was implemented, the users found that having multiple separate graphs required frequent scrolling, which made comparisons between metrics less convenient. As a result, the user

group suggested combining the graphs into a single, custom graph where different metrics could be visualized together. This approach was especially well received by project managers and was therefore implemented (as seen in Section 4.4). To preserve the ability to compare multiple metrics at once, the combined graph allows the user to select two metrics simultaneously, assigning one to the left y-axis and one to the right y-axis.

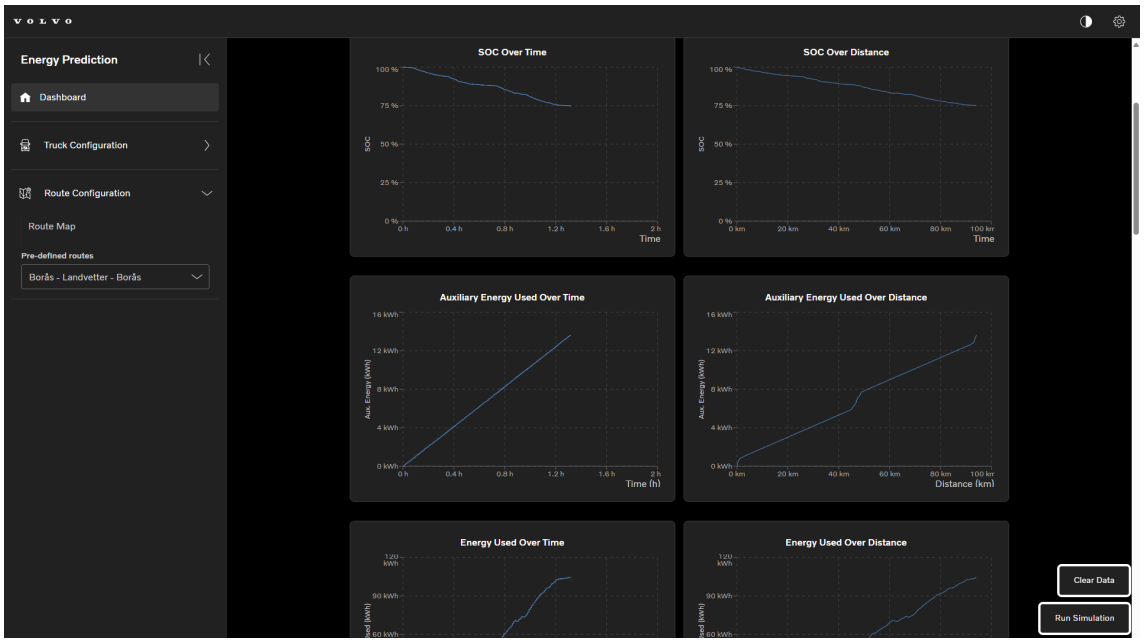


Figure 3.8: Initial implementation of dashboard displaying a subset of the graphs available.

3.3 Usage of React

To create a dynamic, modular, and maintainable frontend, the project utilized the built-in functionality of React. Its component-based architecture and robust ecosystem made it a suitable choice for building an interactive and user-friendly web application. Several key features of React were used in the project:

3.3.1 Custom Components

A typical approach when using React is to divide the application into smaller components. These components can then easily be reused, modified, and extended as needed.

For example, the KPI cards (seen in Figure 3.9) were built upon the same base implementation. The only difference between them was the corresponding data values. This approach is comparable to a function: the component receives input data and returns a standardized visual output.

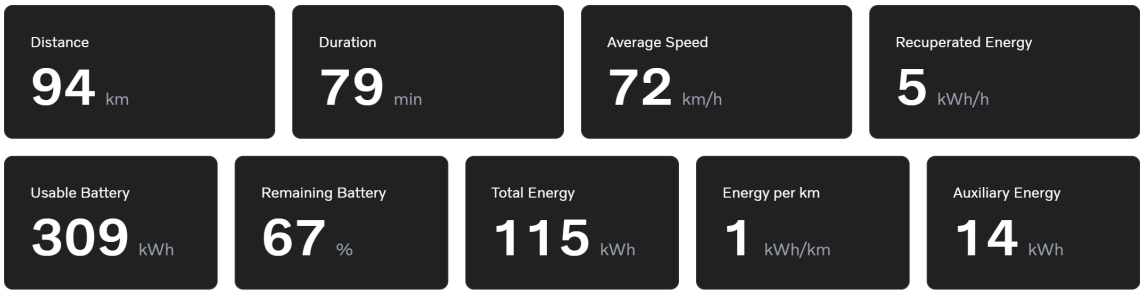


Figure 3.9: Dashboard card components.

An example of one of these functions can be seen in Listing 1, where a Dashboard Card component is defined. The `DashboardCard` component is a React functional component that accepts three props defined by the `DashboardCardProp` type: `title`, `value`, and `unit`. The component returns a card with consistent dimensions and styling, displaying the provided data in a structured format.

Changing a property, such as the card's width (e.g., from 260 pixels to another value), would automatically apply to all instances of the `DashboardCard` component.

```
const DashboardCard = ({ title, value, unit }: DashboardCardProp) =>
↪  {
  return (
    <Card className="w-[260px] flex flex-col">
      <CardContent>
        <Typography variant="subtitle1">{title}</Typography>
        <Typography
        ↪  variant="display2">{value.toString()}</Typography>
        <Typography variant="caption2">{unit}</Typography>
      </CardContent>
    </Card>
  );
};
export default DashboardCard;
```

Listing 1: Display of React `DashboardCard` component.

An example of usage of the `DashboardCard` component can be seen in Listing 2, where an array with elements of type `DashboardCardProp` is being mapped over, each creating new `DashboardCard` components.

```
const dashboardCards: DashboardCardProp[] = useDashboardCard();

{dashboardCards.map((card, index) => (
  <DashboardCard
    key={`dashboard-row1-${index}`}
    title={card.title}
    value={card.value}
    unit={card.unit}
  />
))}
```

Listing 2: Usage of DashboardCard component.

The array `dashboardCards` is retrieved by calling the custom hook `useDashboardCards`, seen in Listing 3. Each object is populated with a title, a value, and a unit. The value is retrieved by a utility function, presented later in Listing 5.

```
export function useDashboardCard(): DashboardCardProp[] {
  if (!isSimulationDataAvailable()) return [];
  const cards: DashboardCardProp[] = [
    { title: "Distance", value: getTotalDistanceKm(), unit: "km" },
    { title: "Duration", value: getDurationMinutes(), unit: "min"
    ↪  },
    { title: "Average Speed", value: getAverageSpeed(), unit:
    ↪  "km/h" }
    ...
  ];
  return cards;
}
```

Listing 3: Implementation of `useDashboardCard` hook.

3.3.2 Custom Hooks

React hooks [40] are special functions that let developers use state and other React features without writing a class. To promote code reusability and maintain a clean codebase, several custom React hooks were implemented. These hooks abstract repeated logic, simplify API communication, and manage form behaviors across the application.

For example, creating a list of `DashboardCardProps` is handled by calling the `useDashboardCard` function, as seen in Listing 3. This returns an array of objects of type `DashboardCardProp`, later used as input props for rendering `DashboardCard` components, as seen in Listing 2.

Another important hook is `useSimulationData`, seen in Listing 4, which manages the complexity of asynchronous simulation requests and associated states.

```
export function useSimulationData() {
  const [data, setData] = useState<PredictEnergyResponse |
  ↪  null>(null);
  const [loading, setLoading] = useState<boolean>(false);
  const [error, setError] = useState<string | null>(null);

  const fetchSimulation = async (route: RouteConfiguration, truck:
  ↪  TruckData) => {
    setLoading(true);
    setError(null);
    try {
      const result: PredictEnergyResponse = await
      ↪  runSimulation(route, truck);
      setData(result);
      setSimulationData(result);
    } catch (e) {
      setError(`Error: ${e}`);
    } finally {
      setLoading(false);
    }
  };
  return { data, loading, error, fetchSimulation };
}
```

Listing 4: Implementation of `useSimulationData` custom hook.

This hook encapsulates the workflow for running a simulation through the backend proxy server. It internally manages three states using React's `useState`:

- `data: PredictEnergyResponse | null` - the output of the energy prediction algorithm, or null if no simulation has been run.
- `loading: boolean` - a state indicating whether a simulation request is currently in progress.
- `error: string | null` - a string containing an error message if the request fails, or null if there is no error.

The hook exposes a `fetchSimulation` function, which triggers a `POST` request to the backend, sending the current truck and route configurations. It automatically handles loading and error states, and stores the successful response both locally (for immediate component use) and globally (through `setSimulationData`, enabling shared access across the application).

By abstracting this logic into a dedicated hook, the codebase benefits from:

- Cleaner UI components, as the loading and error-handling logic is abstracted into the custom hook rather than being repeated in every component that uses simulation data.
- Improved reusability, as any component can easily trigger a simulation by calling `fetchSimulation`.
- Better separation of concerns, by keeping side effects and data management outside the visual components.

Overall, `useSimulationData` played a central role in maintaining an organized architecture and ensuring consistent behavior across the application.

3.3.3 States

Efficient state management was crucial to ensure a smooth and responsive user experience. React's built-in `useState` and `useEffect` hooks were primarily used to manage local component state, handle asynchronous updates, and synchronize user input with system behavior.

Since the simulation API requires a few seconds to return results, it would have been a poor user experience if the data disappeared after navigating between pages. By saving the retrieved API response in local state, it became possible to switch between pages while still maintaining access to the most recent simulation run.

3.4 Code Architecture & Runtime Flow

The application adopts a modular client-server architecture with integration to an external backend API provided by Volvo Group. The architecture emphasizes a clean separation of concerns across the client (frontend), server (proxy backend), and third-party services (Volvo APIs), allowing for extensible development and maintainability.

The architectural flow can be interpreted through a typical user interaction: starting the application, initiating a simulation, and interpreting the results. This section presents a detailed walkthrough of how data flows through the system, highlighting the logical structure, runtime behavior, and key implementations made in the codebase.

The architectural flow of the application is illustrated in Figure 3.10, which shows the logical component layers and data flow between the client, proxy server, and external Volvo APIs.

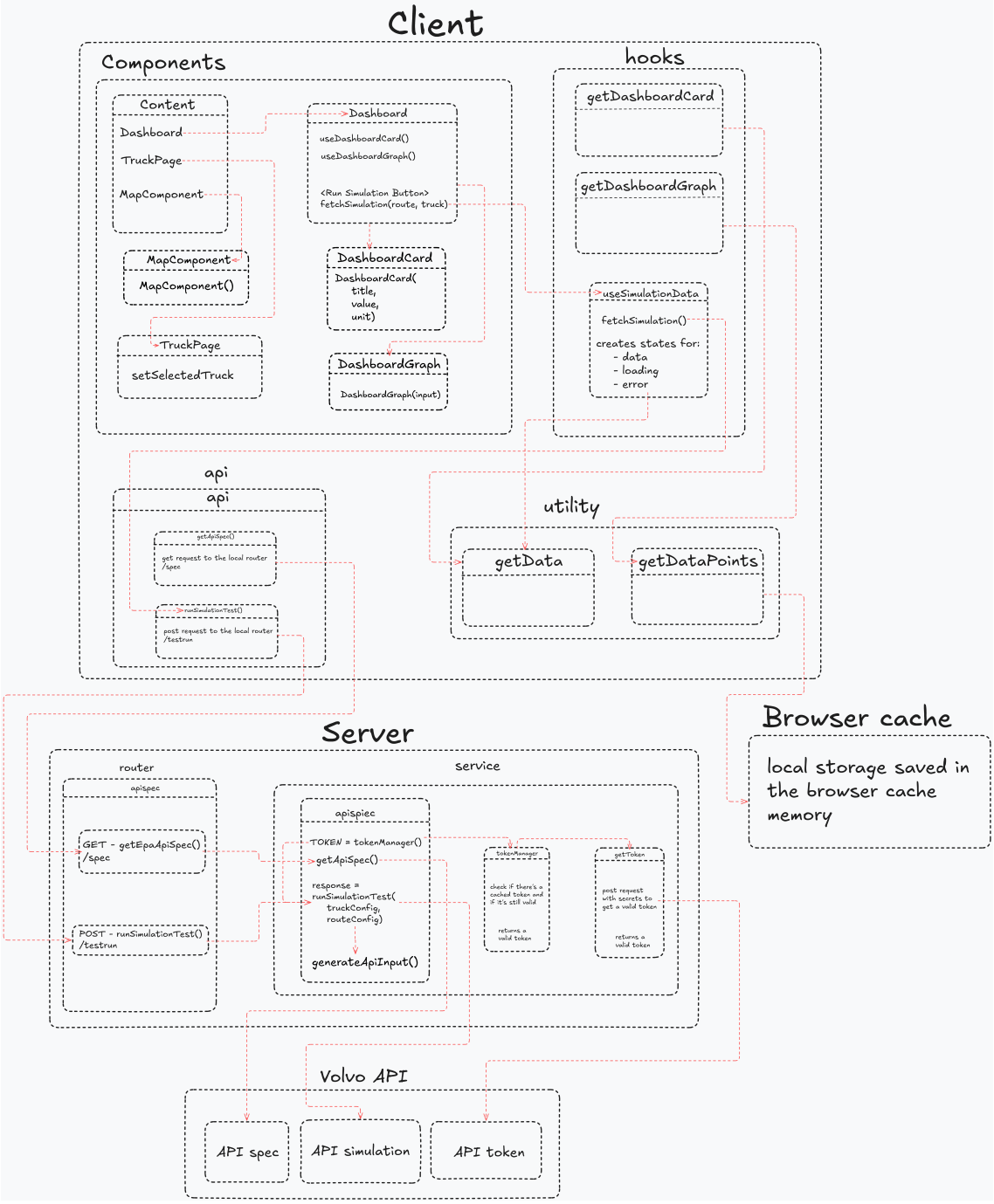


Figure 3.10: Diagram over the codebase architecture.

3.4.1 Initialization and Default State

To run the application locally, two Node.js servers need to be initialized: one serving the React frontend and another acting as an Express [41] proxy server. The frontend is initialized with a default truck and route configuration, enabling the user to trigger a simulation immediately without manual setup.

At startup, the dashboard page is rendered but remains visually blank until simulation data is retrieved. This state management ensures that the UI remains consistent and that unnecessary network calls are avoided until explicitly triggered by the user.

3.4.2 Executing a Simulation: Data Request Flow

When the user clicks the "Run Simulation" button on the dashboard, the function `fetchSimulation()` is invoked. This function is part of the custom hook `useSimulationData` 4, which encapsulates the entire simulation request lifecycle, including request status (loading, error) and the response data itself.

Internally, `fetchSimulation()` calls the backend's `/runsimulation` endpoint with the current truck and route configuration. This configuration is serialized and transmitted via a `POST` request to the proxy server.

3.4.3 Server-Side Proxy and Volvo API Integration

The Express server receives the simulation request at the `/runsimulation` endpoint and proceeds with the following steps:

1. **Token Management:** A function `getValidToken()` is called to ensure that a valid Volvo API token is available. If the existing token is expired or unavailable, `getToken()` issues a new `POST` request using secrets stored in environment variables in the proxy server. This mechanism ensures security and seamless operation without exposing sensitive data to the client.
2. **API Input Generation:** The incoming configurations are passed to `generateApiInput()`, which transforms them into a format accepted by Volvo's energy prediction API (EPA). This includes serializing KOLA codes, truck metadata, and route topography.
3. **Simulation Execution:** A request is then sent to the Volvo EPA simulation endpoint using the validated token and formatted input. The response is a hierarchical Protobuf object.
4. **Response Validation:** The response is then passed through a validation function that makes sure that the response contains the correct information, is of correct structure, and correct type. When validated, the response is passed to the client. Validating the data allows the client to handle it as if it already knows what the response looks like, without any error handling needed.

3.4.4 Rendering Simulation Results

Once the client receives the simulation response, it is stored both in a React state variable called `data`, as well as the browser's local storage. The `data` variable is globally accessible across components via shared state hooks, allowing for seamless re-rendering and conditional UI updates.

For example, the `Dashboard` component invokes the custom hook `useDashboardCard()`, which in turn queries functions such as `getDistance()` or `getEnergyConsumption()`, seen in Listing 5, to extract specific metrics from the data state.

```
export function getTotalDistanceKm(): number {
  const simulationData: PredictEnergyResponse = requireData();
  const distance: number = simulationData.Summary.distance;
  const distanceKm: number = parseFloat((distance /
  ↪  1000).toFixed(2));
  return distanceKm;
}
```

Listing 5: Function to retrieve the simulated route's total distance.

The array returned by `useDashboardCard` is later used in the `Dashboard` component by mapping each element to a `DashboardCard` component, as shown in Listing 2.

3.4.5 Component Hierarchy

The frontend architecture can be divided into four logical layers:

- **Components:** Pure visual elements like `DashboardCard`, `MapComponent`, and `DashboardGraph`.
- **Hooks:** Business logic handlers such as `useSimulationData` and `useDashboardCard`.
- **Utility:** General-purpose functions for data extraction and formatting (`getData`, `getDataPoints`).
- **API:** Communication handlers that interface with the Express proxy server (`runSimulation`, `getEpaApiSpec`).

This structure adheres to modern React best practices by promoting the single-responsibility principle and improving code reusability. Custom hooks isolate side effects, while stateless components focus solely on rendering.

3.4.6 Summary

The system's architecture was deliberately structured to support modularity, responsiveness, and security. Each user interaction, from loading the application to running a simulation, triggers a well-defined sequence of function calls, data transformations, and rendering logic. This flow-based architecture not only simplifies

debugging and maintenance but also improves performance and user experience.

By abstracting complex backend operations (e.g., token refresh, input serialization) from the user, the application empowers Volvo Group project managers to perform and analyze energy simulations efficiently, without deep technical knowledge of the underlying infrastructure.

3.4.7 Usage of Data Structures

Appropriate data structures were used throughout the implementation of the application, with a large focus on extensibility. The goal was to make sure that the application was extremely easy to extend and build upon so that Volvo Group could continue with its implementation after this thesis project.

To illustrate this, the implementation of pre-configured trucks could be considered. A custom type called `TruckData` was created, as seen in Appendix C, to make sure only relevant data that is required to configure a truck is stored. Then, a file called `PreBuiltTrucks.ts` was created with the sole responsibility of initializing and exporting an array of pre-built truck configurations; i.e. `const PreBuiltTrucks: TruckData[]`. This array was passed to any files where the list of pre-built truck configurations was required (such as the top bar or the pre-configured trucks page). If a pre-built truck needed to be modified, added, or deleted, one could modify the constant exported from `PreBuiltTrucks.ts` and the rest of the application would change accordingly.

Another example is the implementation of the interactive graph. A type called `DoubleGraphProps` was created, as seen in Appendix D, to encapsulate all configuration and data needed to render a double-lined graph. The graph component received this type as input and rendered the chart accordingly. To support flexible metric selection, a dictionary of available metrics was defined, allowing keys to be passed in and visualized dynamically. If a new metric needed to be added or existing data updated, this could be done in one place, and the changes would be reflected across the application.

3.5 Proxy Server for API Communication

The internal Volvo energy prediction API is configured with secure access control policies. Since the frontend application is run on a browser and served locally during development, direct HTTPS requests from the client to the Volvo APIs would trigger cross-origin errors, making direct communication impossible.

To solve this issue, a local proxy server was implemented. Since these restrictions apply within browsers, and not servers, a Node.js proxy server could act as an intermediary: receiving requests from the React frontend, forwarding them to the Volvo API, and returning the responses in compliance with access control requirements.

The communication flow is illustrated as follows:

- When the user initiates a simulation by pressing the "Run Simulation" button, the React client sends a `POST` request to the local proxy server, containing the truck and route configurations.
- The proxy server then attempts to forward this request to the Volvo EPA API. To authenticate, a valid access token is required.
- If a cached valid token is available, it is used for the API call. If the token has expired (tokens expire after one hour), the server automatically requests a new token by authenticating against an internal endpoint using secrets.
- These secrets are securely stored in `.env` files and are only accessed at runtime to ensure no sensitive information is exposed.
- Once a valid token is obtained, the server performs the `POST` request to the EPA API. After the external API call is complete, the response data is sent back to the React client through the proxy server.
- The frontend then processes the response and displays relevant simulation results and visualizations to the user.

3.6 Limitations

It has been truly exciting to collaborate with Volvo Group and design an application to solve a problem that real users are facing. However, this collaboration has resulted in a couple of limitations that have significantly influenced the methodology of this project as well as its scope.

Firstly, the initial plan was that two members of the group would be under contract with Volvo and onboarded onto Volvo systems, whereas the other two would not. This would mean that half the group would not have access to the Volvo API, nor write code directly into the Volvo GitHub repository. To compensate for this, Volvo would design a version of the algorithm with fake data or provide mock outputs, which all team members would have access to.

However, this plan was later changed, and instead, it was decided that all members of the group could be under contract and have access to Volvo systems. This would enable the group to work much more effectively, and all members would have equal access. Unfortunately, this process took longer than expected, and it was not until the 21st of March, the week after the half-time presentation, that all members had a Volvo laptop, which was required to access the repository, the Energy Prediction API, or any other Volvo system. This resulted in the development of the application being started much later than expected, but it also allowed the group to focus on prototyping and getting many of the details right from the get-go. However, due to the substantial reduction in development time, the scope of the application had to be adjusted accordingly.

Once everyone had their own Volvo laptops, quite a bit of administrative work was required to get access to various software and permissions. For example, all group

members needed admin access (to, for example, install Node JS), and that took a couple of days.

Finally, there were also some limitations regarding the internal hosting of the application within Volvo. At the beginning, there were some ideas for developing a full-stack web application, essentially turning the proxy server into a full-fledged server with more responsibilities. However, hosting such a server within Volvo proved to be quite challenging. Due to this, the focus of the project shifted to functionality that did not require a backend server.

4

Results and Discussion

This chapter evaluates how the developed energy prediction front-end application fulfills the core objectives of modularity, user-centric design, and robust scenario analysis. Key results are presented together with a discussion of design decisions, their implications, and the overall development process.

4.1 Application Overview and Navigation

This section provides an overview of the application's main user interface, describing its layout, navigation flow, and key functional areas.

4.1.1 General Layout

The application uses a dark color theme and both contrasting colors and sans-serif fonts to enhance legibility. Initially, support for both dark and light mode was considered to increase user flexibility. However, due to time constraints and prioritization of core functionality, implementation of a light mode was not completed within the project timeframe. The current dark theme, combined with careful use of contrasting colors and appropriate typography, has been found to meet usability requirements in practice during testing with the user group.

The application also features a static layout with a persistent side menu and title bar, complemented by a dynamic content area that updates as users navigate between pages. The title bar consistently displays the Volvo logo along with configuration-specific details, such as the currently selected route and active truck configurations, which update automatically when changes are made. Navigation is managed through the side menu, where users can expand accordion menus to access different sections of the application. When a user switches to a specific accordion menu, that menu is highlighted to indicate the current selection. The available pages include the dashboard, truck configuration, and route configuration pages, each offering distinct functionality:

- **Dashboard:** Shows the results of simulations, including performance numbers and simple graphs.
- **Truck Configuration:** Enables users to select or customize truck configurations for simulations.

- **Route Configuration:** Allows selection of predefined routes or the creation of custom routes through an interactive map interface.

4.1.2 Standard Configurations

When the application starts, specific configurations are already selected by default. In particular, the route "Borås-Landvetter-Borå" and truck configurations 1 and 2 are pre-selected as two ready-to-run setups that users can execute immediately from the dashboard. In addition to these defaults, both the Truck and Route Configuration pages offer a broader set of standard, pre-defined configurations that users can choose from according to their needs. This structure was implemented based on requirements from the user group, who emphasized the importance of having convenient, ready-made configurations for both immediate use and for common scenarios. If the user wants to change the active configuration, this is accomplished via the Truck and Route Configuration pages. Any changes update the active truck or route, which is shown in the title bar for clarity and consistency.

4.2 Title Bar

Results

The title bar is a persistent element that provides context and navigation support throughout the application. As illustrated in Figure 4.1, it displays the application name for branding and the currently active truck and route configuration for both configuration pairs. This ensures that users are always aware of their simulation context, regardless of which page they are on. When users modify either the truck or route configurations, the title bar updates to reflect these changes in real time. Additionally, the Volvo logo in the title bar acts as a home button, allowing users to quickly return to the dashboard.

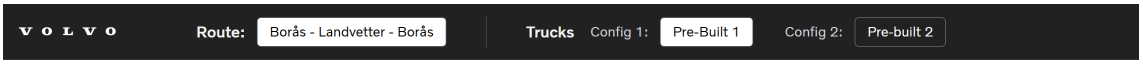


Figure 4.1: Title bar with standard configurations selected.

Discussion

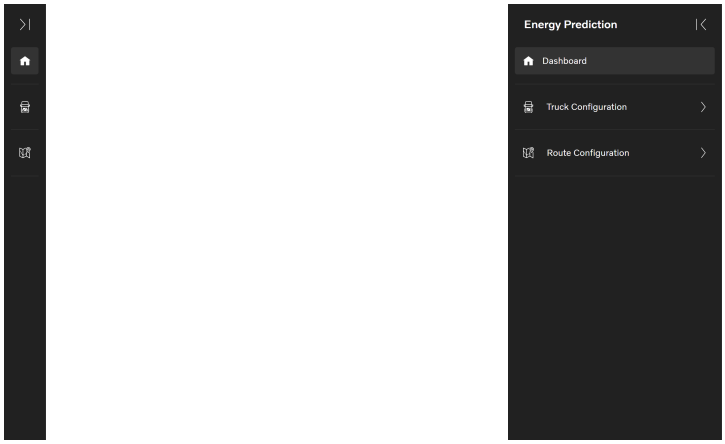
By centralizing vital contextual information, the title bar enhances both usability and situational awareness. Its persistent presence reduces confusion and helps navigation. The title bar displays the selected route for both configuration pairs, as well as the selected truck configuration for each pair. This makes it easy for users to compare different simulation setups. This design directly supports the usability heuristic Recognition rather than recall, which is explained in 2.3. By making key information, actions, and navigation elements always visible, the interface minimizes the user's memory load. Users do not have to remember which configurations are active or what routes are selected as they move between pages since the title bar

keeps this context readily accessible. Furthermore, the extensibility of the title bar allows future integration of more features, such as user settings, theme change management, or quick links. Its consistent placement, styling, and role in conveying contextual information reflect the Visual Framework pattern (Section 2.2.4), which helps establish a unified structure across the application. This supports the heuristic of consistency and standards by assisting users to stay oriented as they navigate between different views, as described in Section 2.3.

4.3 Side Menu

Results

The side menu serves as the application's main navigational component, providing direct access to the Dashboard, Truck Configuration, and Route Configuration pages. As illustrated in Figure 4.2, it visually highlights the active page, giving users immediate feedback about their current location within the application. The menu organizes related functions into groups, utilizing accordion elements to keep navigation clear and compact.



Side menu collapsed Side menu expanded

Figure 4.2: Comparison between collapsed and expanded side menu states, with active page highlighting and grouped navigation links.

Discussion

The side menu is the central mechanism for navigating between all main sections of the application, including the configuration pages and the dashboard. Its collapsible and expandable design not only enhances usability but also supports different user needs by providing either a compact icon-only view or a fully expanded menu with descriptive labels and grouped options. Visual feedback is provided during navigation, such as the filled white icon and automatic accordion expansion, which supports the heuristic of visibility of system status as described in Section 2.3 by immediately showing the user which view is active. This exclusive navigation behavior

ensures that only one configuration section is open at any time, reducing confusion and helping the user view or edit one property at a time.

From a design perspective, this modular and dynamic approach greatly facilitates future development. New features, such as a potential mission management module, can be seamlessly integrated as additional tabs or accordions within the side menu, without disrupting the overall structure. This aligns with the Open-Closed Principle 2.3 by allowing the menu to be extended without modifying its core behavior. It also reflects the heuristic of consistency and standards, as described in Section 2.3, as the menu layout and behavior remain uniform across different sections of the application. By making navigation intuitive, providing immediate feedback, and supporting modular expansion, the side menu contributes significantly to both the user experience and the long-term adaptability of the system.

4.4 Dashboard

Results

The Dashboard component functions as the primary interface for aggregating and visualizing all simulation result data. As depicted in Figure 4.3, it displays key performance metrics using summary cards and features a customizable graph that allows users to analyze simulation outcomes in detail.

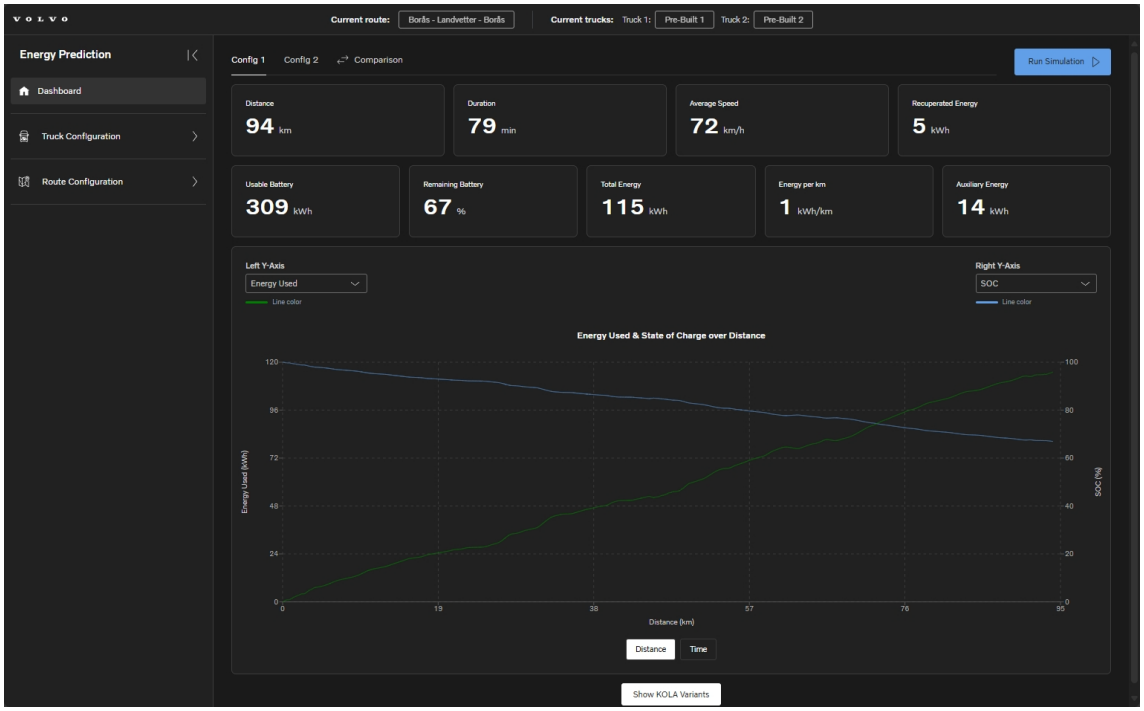


Figure 4.3: Dashboard layout containing default route and truck configuration data.

The Dashboard supports two independent configuration setups. Users can define and run two different simulation configurations, each accessible in a separate tab and simulated using the run configuration button to the right of the tabs. Results for each configuration are viewed by navigating to that configuration's tab and visually inspecting the outputs.

In the third tab, a comparison view is available, as shown in Figure 4.4. This view enables users to directly compare the simulation results of the two configurations. The results for each configuration are displayed in separate columns, with an additional column presenting the difference between them. Positive differences, such as lower energy consumption or higher remaining battery percentage, are highlighted in green, while negative differences are indicated in red.

The comparison view (Figure 4.4) also features a spider chart, providing a visual summary of multiple key metrics across the two configurations.

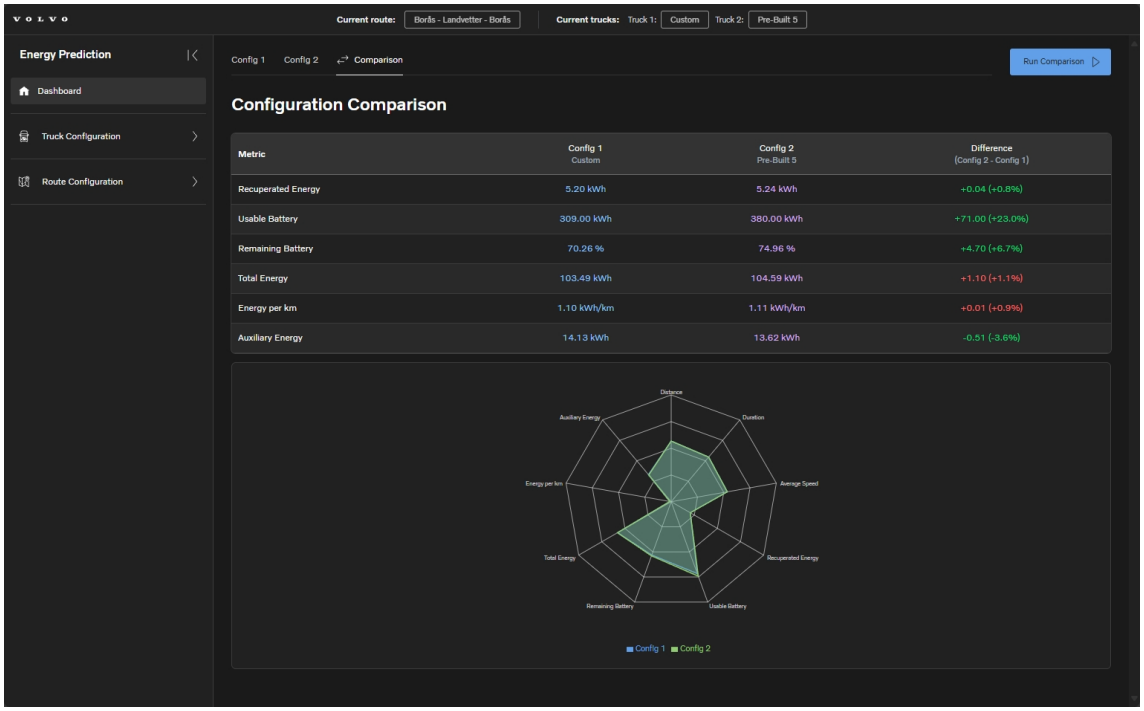


Figure 4.4: Comparison view of the Dashboard showing side-by-side metrics, calculated differences, and a spider chart visualizing two simulation configurations.

Discussion

The dashboard's design effectively supports both usability and analytical needs by aggregating simulation data and presenting it through clear visual components. The layout applies the dashboard pattern described in Section 2.2.1, as this approach allows users to view all relevant metrics in one place without having to navigate across multiple pages or views, thereby streamlining the analysis process. In addition, the Center Stage pattern (Section 2.2.2) is applied by placing all important

content front and center and using a clean, minimalist design, reflecting the heuristic of aesthetic and minimalist design. Real-time update capability further enhances workflow, supporting rapid experimentation and facilitating data-driven decision-making. Additionally, dashboard elements are designed for extensibility, allowing new features or metrics to be added without altering the core rendering logic, which aligns with the Open–Closed Principle (described in Section 2.3) and contributes to the system's long-term flexibility.

The dual-configuration setup, combined with the integrated comparison view, enables users to efficiently evaluate differences between simulation scenarios. Presenting the configurations side by side, along with the calculated differences and color-coded indicators, simplifies comparative analysis and reduces cognitive effort. This design allows users to quickly identify significant variations in key metrics without the need for manual inspection across multiple screens.

The inclusion of the spider chart further supports comparative analysis by providing a high-level, multidimensional summary of several metrics simultaneously. This visual representation allows users to easily detect trade-offs between different configurations and facilitates more informed decision-making. The spider chart was implemented in response to a request from the user group, who explicitly asked for this visualization in the comparison view.

Overall, the dashboard component provides a robust platform for analyzing and comparing simulation results. Its current functionality already offers strong support for the users' analytical workflows, while future extensions may focus on adding additional metrics or visualizations to further enhance its decision-support capabilities.

4.5 Truck Configuration

Results

The truck configuration page serves as the interface for updating the selected truck configuration for either of the two available setups. Users determine which configuration they are editing by checking the title bar at the top of the page. The active configuration is indicated by a white background.

Two primary configuration modes are provided: pre-built trucks and custom truck configuration. To speed up the standard simulation workflow for certain users, a set of pre-built truck configurations was introduced, chosen based on the most frequently used combinations identified through user feedback from Volvo. Users can select a pre-built truck by clicking "Select" on the desired entry within the pre-built trucks tab, as illustrated in Figure 4.5.

For more specialized or evolving needs, the custom truck configuration feature lets users define configurations aligned with Volvo's internal construction system. After

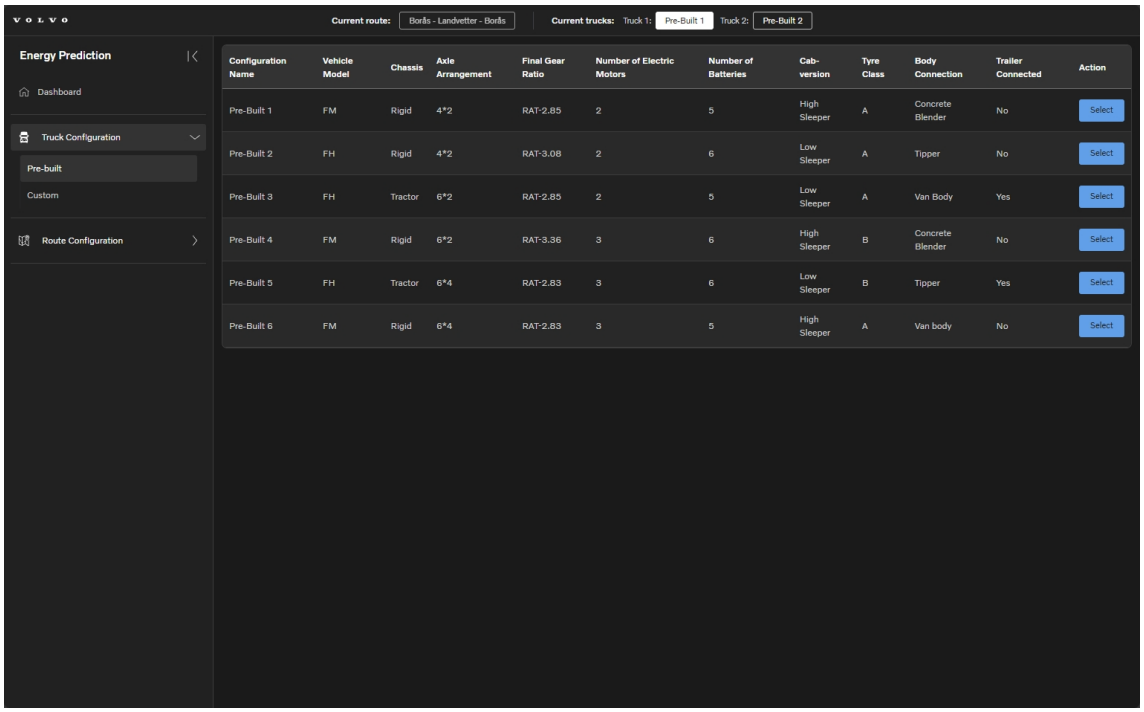


Figure 4.5: Pre-built trucks configuration page.

selecting the desired parameters, users confirm the configuration selection by clicking the "Select Configuration" button at the bottom right of the custom configuration page (Figure 4.6). This functionality was added in response to direct requests from user groups and to support future extensibility.

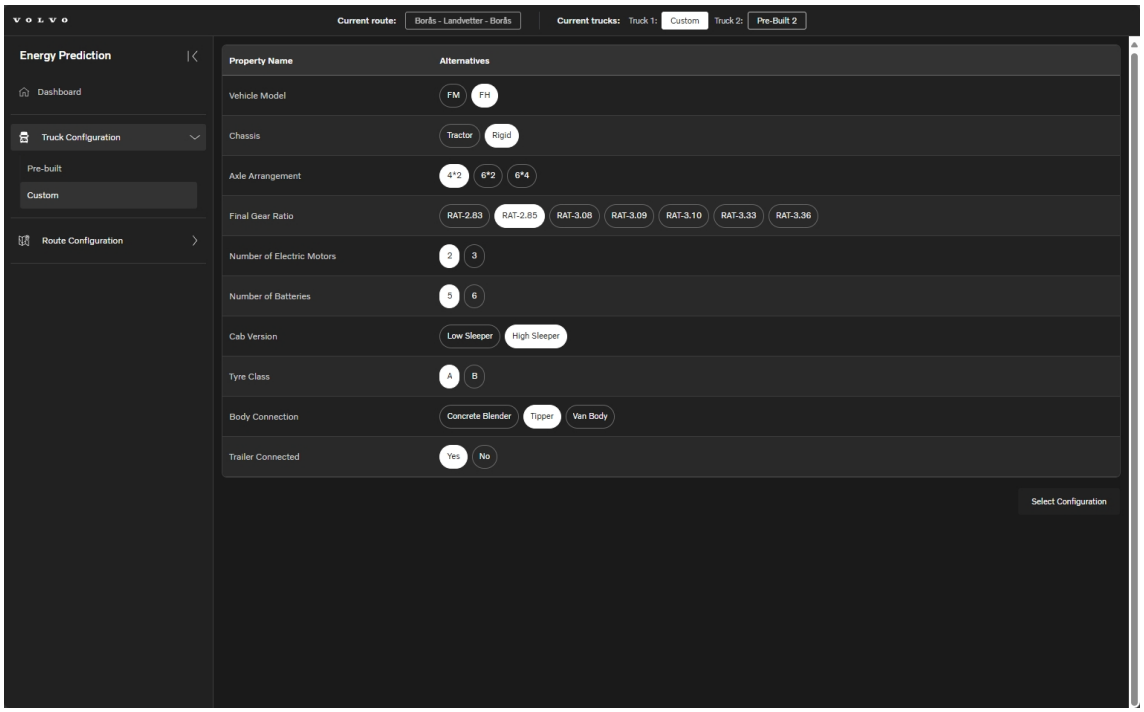


Figure 4.6: Custom truck configuration page.

Discussion

The two truck configuration options balance usability and adaptability. Pre-built configurations streamline common workflows, while the custom mode provides flexibility for specialized requirements. This design supports future extensibility, as new configuration options or validation rules can be integrated without major changes, aligning with the Open–Closed Principle presented in 2.3.

Usability is further enhanced by clearly labeled fields, visual validation, and sensible default options, which reduce cognitive load and support recognition over recall. These features help users confidently manage both configuration modes without needing to remember internal logic.

Currently, users may inadvertently select conflicting parameters when building custom configurations, potentially resulting in invalid setups per Volvo's internal rules. A future improvement is to implement a validation system that checks whether specific property combinations are allowed, thereby improving data quality and user trust.

Additional planned enhancements include expanding the set of pre-built trucks based on evolving user needs and feedback, and enabling users to save configurations to the cloud. This would facilitate sharing, collaboration, and workflow efficiency across teams, further aligning the tool with user needs and Volvo's operational requirements.

4.6 Route Configuration

Results

The route configuration module is designed to efficiently update the global route used for both configuration pairs. Two primary approaches are available: selecting from predefined routes or creating custom routes. Predefined routes accelerate common simulation tasks and are based on frequently used scenarios within the Volvo organization, as identified through user feedback. Users can access these routes from the selection menu in the sidebar, as shown in Figure 4.7. This menu contains four standard routes that are most relevant to Volvo's operations.

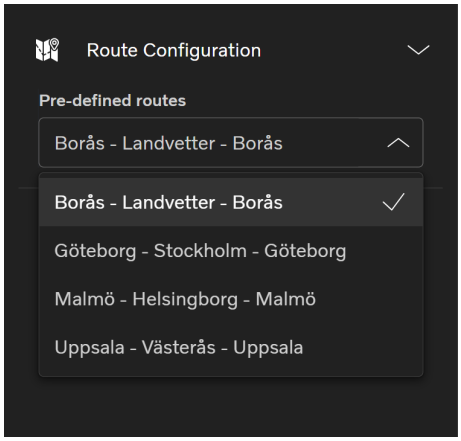


Figure 4.7: Predefined route configuration menu in the sidebar.

For greater flexibility, users can define custom routes by placing start, end, and waypoint markers on an interactive map in the content area. The users navigate to the custom route page, shown in Figure 4.8, by simply selecting the "Route Configuration" option in the side menu. The custom route page enables users to either edit an existing predefined route or create a new one using the controls at the top left of the map. The "Select Route" button becomes active once a valid path has been created. Selecting a route updates the global route used for all simulations.

Additionally, an indicator at the top right of the map provides visual feedback on the map's interactivity, making it easy for users to identify whether they are in edit mode.

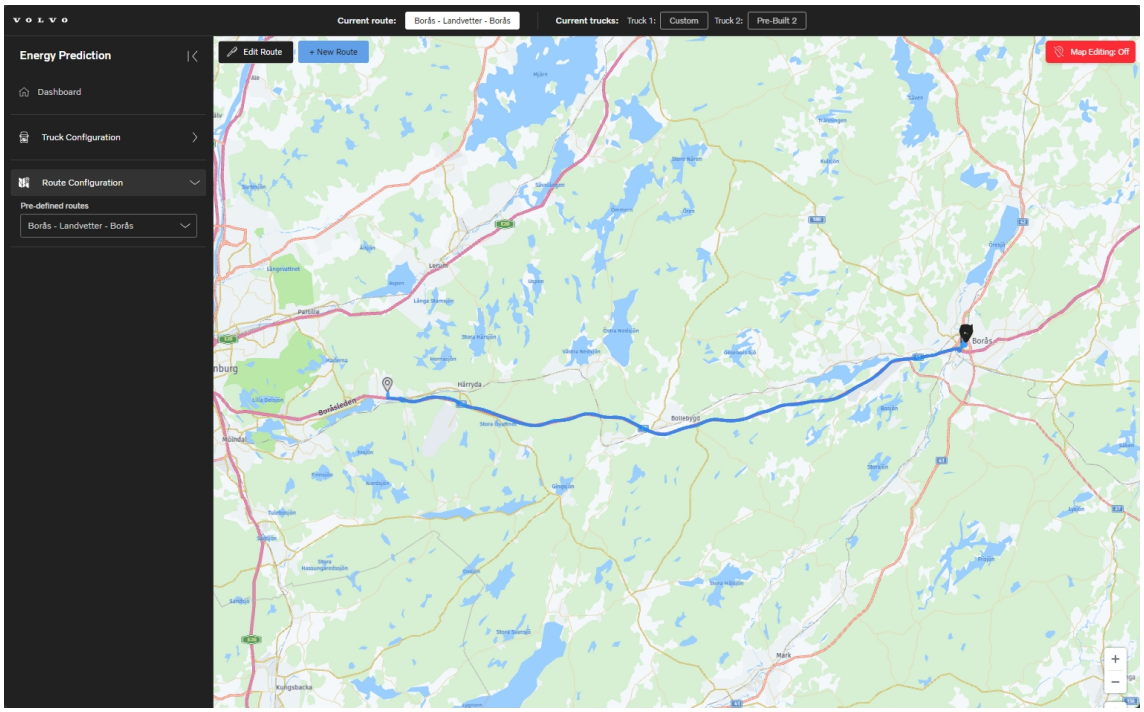


Figure 4.8: Custom route configuration page using the interactive map.

Discussion

The route configuration module's dual-mode approach balances efficiency and adaptability by allowing users to select from predefined routes for standard scenarios or define custom routes for more specialized needs. This modular design also supports future extensibility, enabling independent updates to predefined and custom routing features. It aligns well with the Open-Closed Principle described in 2.3, as new route options or validation logic can be introduced without major structural changes.

Several opportunities for improvement remain. Currently, users cannot assign specific types or properties to waypoints such as rest stops, delivery points, or charging stations, which limits the ability to fully model real-world transport scenarios. Introducing the ability to mark waypoints with different roles, edit routes directly, and display additional contextual information (such as truck details when hovering over markers) would enhance both usability and scenario realism.

Furthermore, integrating cloud-based route management, including saving, reusing, and sharing routes, would promote collaboration and standardization across teams. These enhancements would help the route configuration module better meet operational needs and user expectations within the Volvo organization.

4.7 System Architecture

Results

The system follows a three-tier architecture comprising a ReactJS front-end, an ExpressJS proxy server, and the Volvo Group's internal energy prediction API. ReactJS was selected for the front-end due to its extensibility, widespread adoption [42], and explicit approval by Volvo, ensuring both organizational alignment and technical robustness.

An ExpressJS proxy server was introduced solely to facilitate front-end development by enabling communication between the ReactJS application and Volvo's internal API. Currently, direct front-end access to the internal API has not yet been configured within Volvo's infrastructure, requiring this temporary intermediary solution. The proxy does not enhance security; rather, it is a development necessity. Importantly, this setup does not increase the risk of external attacks, as the API remains callable only from within Volvo's internal network and hardware infrastructure. Its compatibility with Node.js also aligns with the existing technology stack.

This layered approach enhances modularity by decoupling the user interface, middleware, and backend services, supporting independent development and future extensibility. An overview of the system architecture is depicted in Figure 4.9.

Figure 4.9: System architecture overview.

Discussion

The chosen architecture directly reflects the project's goals of modularity, security, and adaptability within organizational boundaries. ReactJS provides a robust and maintainable foundation for the user interface, consistent with Volvo's development standards. The ExpressJS proxy, while a pragmatic solution to current back-end access restrictions, also exemplifies the system's modular design: it allows the front-end and back-end to evolve independently and supports future changes with minimal disruption.

The architecture also aligns with the Open–Closed Principle defined in 2.3. Each part is designed to be extended with new functionality e.g., supporting additional APIs without altering existing components. The responsibilities within each layer are clearly defined, adhering to the Single-Responsibility Principle mentioned in Section 2.3.

Nevertheless, the proxy layer introduces additional complexity and potential maintenance overhead and may slightly impact response times. Ideally, direct front-end access to the internal API would further streamline the architecture, reducing latency and simplifying data flow. However, as direct access was not possible due to internal policies outside the project's control, the proxy was necessary to ensure both compliance and functionality.

4.8 Goal Fulfillment

The following subsections connect the delivered system to the original project objectives.

4.8.1 Modularity and Extensibility

The system architecture applies the Single Responsibility Principle by organizing logic into focused components and custom hooks. For example, simulation logic is handled separately from rendering, and navigation is encapsulated within distinct layout components. The system is open for extension: new dashboard cards, configuration options, or navigation sections can be added without modifying existing structures, reflecting the Open–Closed Principle. While dependency inversion is less explicit in frontend development, the idea is applied by injecting logic and configuration via props, rather than hard-coding dependencies into UI components. The three-tier architecture—React frontend, Express proxy, and Volvo API—further supports modular development and maintainability.

4.8.2 User-Centric Configuration

The interface design reflects several of Nielsen's usability heuristics. Visibility of system status is addressed by clear feedback elements, including loading indicators and error messages. The application also displays the currently selected trucks and route in the title bar, providing instant feedback to the user. The heuristic Recognition rather than recall guided the design of configuration panels, where users choose from visible options rather than entering parameters manually. The title bar again ensures users do not have to remember information between different parts of the application. We also prioritized Aesthetic and minimalist design by only displaying relevant fields and values at each step, helping users focus on the task at hand without distractions.

4.8.3 Rapid Scenario Analysis

The ability to configure two independent truck setups and select or create routes supported by a dashboard with real-time updates enables users to quickly run and compare multiple simulation scenarios. Predefined configurations and routes allow for immediate use, while custom options make the tool flexible for varied business needs. Planned features, such as a dedicated comparison dashboard, will further streamline scenario analysis.

4.8.4 Security and Compliance

The application adheres to Volvo's IT guidelines, with all sensitive operations and credentials managed securely on the back-end and no client-side exposure. An ExpressJS proxy server is currently used to facilitate development due to pending internal configurations within Volvo's infrastructure. This proxy is a temporary solution and will be replaced with a direct integration once the necessary internal access is established for production deployment. While not the optimal architecture, this interim setup was necessary to proceed with development and does not affect compliance, as API access remains restricted to Volvo's internal network and hardware.

4.9 Reflections and Lessons Learned

The project emphasized the value of continuous stakeholder communication, especially in navigating incomplete or ambiguous documentation. Iterative feedback cycles led to rapid improvements, such as the addition of predefined routes and enhanced validation.

Technical challenges, including proprietary code mapping and external API limitations, were overcome through collaboration and adaptability. In hindsight, early engagement with Volvo's IT and documentation teams could have streamlined development, especially regarding compatibility logic and API access.

Overall, the experience reinforced the efficacy of modular design, separation of concerns, and a user-focused development philosophy.

4.10 Evaluation by User Group

Toward the end of the development phase, a series of qualitative evaluations of the developed application were conducted with selected members of the user group. Due to scheduling constraints, it was not possible to conduct a single unified evaluation session involving all user testers simultaneously. Consequently, the depth and format of evaluations varied depending on individual availability. In some cases, formal demonstrations were held, while in others, evaluations consisted of brief meetings or informal conversations following exposure to the latest version of the application. Despite these differences, the primary goal remained consistent, which was to qualitatively assess how effectively the application met user needs, as the limited size of the user group did not support a quantitative approach.

Overall, the evaluations yielded positive feedback. Many participants indicated that they could envision integrating the application into their daily workflows, noting particularly the relevance and clarity of the data presented. Users expressed appreciation for specific features such as the comparison view, highlighting that the ability to directly compare two configurations significantly enhanced their analytical

capabilities. Additionally, the custom graph functionality received favorable comments for its flexibility, allowing users to precisely visualize data according to their individual needs and preferences. Some users highlighted the application's interface as a significant efficiency improvement, stating it would notably expedite their routine tasks.

Most participants agreed that the application would be particularly beneficial to project managers, the project's primary target audience, but also saw potential value for more technically oriented users, such as data scientists and engineers, aligning closely with the project's intended scope. To enhance usability specifically for technical users, several participants recommended adding advanced analytical features and the capability to export data and visualizations for integration with external analytical systems.

For a more comprehensive understanding of user satisfaction, formal interviews with all members of the user group would have been ideal, but this was unfortunately not achievable within the project's timeframe due to ongoing scheduling constraints. Nonetheless, the feedback gathered provided valuable insights, including suggestions for further application improvements, which are elaborated in Section 4.11.

4.11 Opportunities for Further Development

Several ideas for extensions emerged throughout the user feedback sessions as well as towards the end during the evaluation process. These suggestions have been documented and provided to Volvo Group for future development.

4.11.1 Saved Configurations

The user group would appreciate if they could save their custom configurations, both with regard to custom trucks and routes, so that they can re-use them when they need to. This would also improve the repetitive nature of configuring custom trucks and routes and would, in turn, improve workflow efficiency. This was an idea during the design phase, but it was decided to be out of scope as this would require a more extensive backend and database setup. Given that it has been challenging to host just the front-end on the Volvo network, hosting a backend server and a database would likely be difficult, but would create a lot of value for the user group.

4.11.2 Export Functionality

Similarly to saving configurations, the user group also expressed the wish to export configurations as well as simulation results. Exporting configurations would enable employees to share the configurations they use so that other employees could import these configurations into the application and run the simulation to independently verify results and test modifications to the configurations. Exporting simulation results would allow the user group, mostly project managers, to use the key values and

graphs in other contexts, such as presentations to different teams or external stakeholders. This would also enable data scientists to conduct more detailed analysis if raw data could be exported in the form of, for example, CSV files.

4.11.3 Separate Interfaces

Finding the balance between developing the application for a considerably different primary and secondary audience has been challenging in a number of ways. One member of the user group suggested that the application could have two separate views for project managers and data scientists. This would have made the designing process simpler, but could be seen as a crude solution. One could also develop the base application for project managers and have "advanced tools" that users, likely those with more technical roles, could opt into.

5
Conclusion

In conclusion, an application has been developed to serve as a graphical user interface to Volvo's energy prediction algorithm, lowering the barrier to its use. The application has all the necessary features required by Volvo employees, including configuring trucks and routes to use for the simulation. The application's dashboard visualizes key metrics often required by the target audience, as well as enables them to conduct data analysis supported by the various graphs produced.

Through recurring user feedback sessions and the final evaluation towards the end of the project phase, the application has been proven to meet user needs. Its features are mainly suited for project managers, the primary target audience of the application, but can also be used by employees in more technical roles, such as engineers and data scientists.

From a user interface design perspective, the study has shown that what is most important is meeting the target audience's needs. All the aforementioned design principles and patterns should not be the primary focus, but rather viewed as tools to achieve the main goal. Although a quantitative study was not feasible due to the size of the user group, continuous feedback loops have ensured that the application meets the target audience's needs, and the user group has expressed satisfaction with the final version of the application.

From an architectural perspective, the focus has been on ensuring that the system is modular and extensible so that Volvo Group can continue building upon it. This has been achieved by effectively using React's component-based architecture, designing custom data structures when appropriate, and separating modules to reduce coupling as much as possible. Some technical decisions, such as implementing a proxy server, impacted the architecture of the system in a manner that was not planned initially, but were required to address the external API limitations outside the development team's control.

Overall, the developed application fulfills its primary goals, providing Volvo Group with an intuitive and extensible tool for energy prediction and scenario analysis. The project's user-focused design ensures that the application will provide value to the target audience by streamlining their workflows, and its modular architecture establishes a strong foundation for future enhancements.

References

[1] "Climate Change 2023: Synthesis Report," Jul. 25, 2023, Edition: First. DOI: 10.59327/IPCC/AR6-9789291691647.

[2] UNFCCC, "Paris Agreement," United Nations, Dec. 12, 2015. [Online]. Available: https://unfccc.int/sites/default/files/english_paris_agreement.pdf.

[3] P. Jaramillo, S. Ribeiro, and P. Newman, "Transport," in *Climate Change 2022 - Mitigation of Climate Change*, Intergovernmental Panel On Climate Change (IPCC), Ed., 1st ed., Cambridge University Press, Aug. 17, 2023, pp. 1049–1160, ISBN: 978-1-009-15792-6. DOI: 10.1017/9781009157926.012.

[4] Transport & Environment, "Addressing the heavy-duty climate problem," Transport & Environment, Brussels, Sep. 16, 2022. [Online]. Available: https://www.transportenvironment.org/uploads/files/2022_09_Addressing_heavy-duty_climate_problem_final.pdf (visited on 05/18/2025).

[5] Z. Lyu, D. Pons, and Y. Zhang, "Emissions and Total Cost of Ownership for Diesel and Battery Electric Freight Pickup and Delivery Trucks in New Zealand: Implications for Transition," *Sustainability*, vol. 15, no. 10, p. 7902, May 11, 2023, ISSN: 2071-1050. DOI: 10.3390/su15107902.

[6] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Oct. 1994, 362 pp., ISBN: 978-0-08-052029-2.

[7] "Protocol Buffers," Protocol Buffers Documentation, [Online]. Available: https://protobuf.dev/ (visited on 05/15/2025).

[8] A. Cooper, R. Reimann, D. Cronin, and C. Noessel, *About Face: The Essentials of Interaction Design*, 4th ed. Wiley, 2014, 720 pp., ISBN: 978-1-118-76657-6.

[9] J. Tidwell, C. Brewer, and A. Valencia, *Designing Interfaces*, 3rd ed. O'Reilly Media, Inc., Jan. 2020, 599 pp., ISBN: 978-1-4920-5196-1.

[10] S. Baltes and V. Dashuber, *UX Debt: Developers Borrow While Users Pay*, Jan. 28, 2024. DOI: 10.48550/arXiv.2104.06908. arXiv: 2104.06908[cs].

[11] B. Frost, *Atomic Design*. Pittsburgh, Pennsylvania: Brad Frost, 2016, 193 pp., ISBN: 978-0-9982966-0-9.

[12] J. Nielsen, *10 Usability Heuristics for User Interface Design*, en, Apr. 2024. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/ (visited on 05/19/2025).

[13] A. Harley, *Visibility of System Status*, en, Jun. 2018. [Online]. Available: https://www.nngroup.com/articles/visibility-system-status/ (visited on 05/19/2025).

[14] R. Krause, *Maintain Consistency and Adhere to Standards (Usability Heuristic #4)*, en, Jan. 2021. [Online]. Available: https://www.nngroup.com/articles/consistency-and-standards/ (visited on 05/19/2025).

[15] R. Budiu, *Memory Recognition and Recall in User Interfaces*, en, Jan. 2024. [Online]. Available: https://www.nngroup.com/articles/recognition-and-recall/ (visited on 05/19/2025).

[16] T. Fessenden, *Aesthetic and Minimalist Design (Usability Heuristic #8)*, en, Jan. 2021. [Online]. Available: https://www.nngroup.com/articles/aesthetic-minimalist-design/ (visited on 05/19/2025).

[17] T. Mens and M. Wermelinger, "Separation of concerns for software evolution," en, *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 5, pp. 311–315, 2002, ISSN: 1532-0618. DOI: 10.1002/smr.257. (visited on 05/19/2025).

[18] R. C. Martin, *Clean Architecture: A Craftman's Guide to Software Strucutre and Design* (Robert C. Martin Series), en. Prentice Hall, 2017, ISBN: 978-0-13-449416-6.

[19] "TypeScript Documentation," TypeScript Language, [Online]. Available: https://www.typescriptlang.org/docs/ (visited on 05/02/2025).

[20] "React Reference Overview," Meta (formerly Facebook), [Online]. Available: https://react.dev/reference/react (visited on 05/18/2025).

[21] "Get started with Tailwind CSS," Tailwind Labs, [Online]. Available: https://tailwindcss.com/docs (visited on 05/18/2025).

[22] "Vite – The Build Tool for the Web," [Online]. Available: https://vitejs.dev/ (visited on 05/18/2025).

[23] "ESLint – Documentation," [Online]. Available: https://eslint.org/docs/latest/ (visited on 05/18/2025).

[24] "What is Prettier?" [Online]. Available: https://prettier.io/docs/en/ (visited on 05/18/2025).

[25] "Figma – the collaborative interface design tool," [Online]. Available: https://www.figma.com/ (visited on 05/02/2025).

[26] "Node.js Documentation," [Online]. Available: https://nodejs.org/en/docs (visited on 05/02/2025).

[27] "What is V8?" [Online]. Available: https://v8.dev/docs (visited on 05/18/2025).

[28] "Docker Docs," [Online]. Available: https://docs.docker.com/ (visited on 05/18/2025).

[29] "GitHub Docs," [Online]. Available: https://docs.github.com/ (visited on 05/18/2025).

[30] K. Beck, M. Beedle, A. Bennekum, and A. Cockburn, *Manifesto for Agile Software Development*, 2001.

[31] K. Schwaber, "SCRUM Development Process," in *Business Object Design and Implementation*, J. Sutherland, C. Casanave, J. Miller, P. Patel, and G. Hollowell, Eds., London: Springer London, 1997, pp. 117–134, ISBN: 978-3-540-76096-2 978-1-4471-0947-1. DOI: 10.1007/978-1-4471-0947-1_11.

[32] R. K. Mallidi and M. Sharma, "Study on Agile Story Point Estimation Techniques and Challenges," *International Journal of Computer Applications*, vol. 174, no. 13, Jan. 2021, ISSN: 0975-8887. DOI: 10.5120/ijca2021921014.

[33] "What is a Product Owner?" Scrum.org, [Online]. Available: https://www.scrum.org/resources/what-product-owner (visited on 05/02/2025).

[34] C. Matthies and F. Dobrigkeit, "Experience vs Data: A Case for More Data-Informed Retrospective Activities," in *Lean and Agile Software Development*, Jan. 6, 2021, pp. 130–144, ISBN: 978-3-030-67083-2. DOI: 10.1007/978-3-030-67084-9_8.

[35] "Google drive: Store and share files online," Google Drive, [Online]. Available: https://workspace.google.com/products/drive/ (visited on 05/19/2025).

[36] "Getting started with trello," Trello, [Online]. Available: https://trello.com/guide (visited on 05/19/2025).

[37] M. Sedlmair, P. Isenberg, D. Baur, and A. Butz, "Evaluating information visualization in large companies," in *3rd BELIV'10 Workshop*, ser. BELIV '10, New York, USA: Association for Computing Machinery, Apr. 10, 2010, pp. 79–86, ISBN: 978-1-4503-0007-0. DOI: 10.1145/2110192.2110204.

[38] "Build and experience your brand new Volvo truck," [Online]. Available: https://www.volvotrucks.se/sv-se/tools/truck-builder.html#/sv-se/ (visited on 05/18/2025).

[39] J. Walny, C. Frisson, M. West, *et al.*, *Data Changes Everything: Challenges and Opportunities in Data Visualization Design Handoff*, Aug. 1, 2019. DOI: 10.48550/arXiv.1908.00192. arXiv: 1908.00192[cs].

[40] "Built-in React Hooks," Meta (formerly Facebook), [Online]. Available: https://react.dev/reference/react/hooks (visited on 05/19/2025).

[41] "Express," Express.js Documentation, [Online]. Available: https://expressjs.com/en/guide/routing.html (visited on 05/19/2025).

[42] Statista Research Department. "Most used web frameworks among developers worldwide, as of 2024." (2024), [Online]. Available: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/ (visited on 05/16/2025).

A

Appendix 1

Sprint Retrospective Example

Below is a table that outlines the initial thoughts of the group during a sprint retrospective, specifically on the 14th of April after the review of sprint 4. These points were, of course, discussed within the group in more detail, but only a concise summary is presented.

Start	**Stop**	**Continue**
Using group chats with Volvo employees rather than direct messaging so that everyone is in the loop	—	Working from Volvo on Tuesdays
Write an update message towards the end of Thursday		Weekly meetings with Robin & Simon
		Recurring agenda point for Robin Meeting: What we're working on currently

B

Appendix 2

Interview Notes from a User Feedback Session

This is an excerpt from one of the user feedback sessions we had. This is based on notes taken in a Google Doc during the meeting and some parts have been removed or modified for clarity and focus. The excerpt is divided into 4 parts: Workflow/Demo, Interview Questions, Notes, and a Summary (which was used to create user stories).

Workflow/Demo:

- We'll provide a quick overview of the application.
- *Show Dashboard*: This is where you'll see the data after running your simulations.
- *Show Pre-Configured Trucks*:
  - To start off, you'll have to choose one of the preconfigured trucks.
  - The idea is that you will also be able to customise trucks as well as save custom configurations in the future.
- *Show Map*:
  - Similarly to trucks, we will have some predefined routes for you to choose from.
  - We'll also allow you to create custom routes by interacting with the map and setting out waypoints in the future.

Interview Questions:

- Truck Config
  - Which elements should be customisable?
  - Anything else you would like to see on pre-configured trucks?
  - Saved config: Would you like the saved config to have a similar layout to pre-defined trucks?

- Route Config
  - How would you want custom routes to work?
    - Waypoints on the map?
    - Anything else?
- Dashboard
  - Which figures would you like here?
    - Energy

        - Avg Power
    - Which graphs?
        - Energy consumption vs distance travelled?
    - Cards you want/don't want to see depending on input.
- Navigation
    - How should the workflow look like? Wizard? Force the user to input something first?

Notes:

- The attributes in the trucklist is good, but we can work on the order of them.
    - Confirm with Simon
- Battery type should be included in the list.
- Motor type should be prefixed based on the choice of battery type?
- Graphs should always have distance on x-axis
- In the dashboard, show what variables/variants are taken into account in a request. So the variables we are not allowing the user to change, still show them to the user in the dashboard so the user knows what's "under the hood". Show it using Kola variants
- Makes sense that the saved page looks the same. Could add a family name?
- Add a custom-built truck section. Would be nice to see the options for for example tyre class, since not everyone knows the variants.
- For default truck. FH and FM first. Maybe FL later.
- How would you like to create custom routes? Create our own route. Would like to be able to enter start point and end point with both long and lat, and with a search field, Should allow for both datasets, clicking on the map and by a search field.
- Userflow: Start in the dashboard, and have things preselected for me. Preselect BLB and preselect FH. Then, run simulation will directly provide the output.

Summary of interview results (in the form of requested features).

- Userflow:
    - Start in the dashboard, and have things preselected for me. Preselect BLB and preselect FH. Then, run simulation will directly provide the output.

- Truck pre-defined
    - **Ask Simon** about order of attributes
        - Vehicle Model, Chassis, Axle Arrangement, Final Gear Ratio, Number of Electric Motors, Number of Batteries, Battery type, Cab Version, Tyre Class, Body Connection, Trailer Connected.
    - Battery type should be included in the list.
    - **Ask Simon:** Could add a family name
        - When adding battery type, we should also add variant families (the options for battery type that the user can select from).

  - For default truck. FH and FM first. Maybe medium duty vehicles later so FL

- Trucks Custom
  - Simple setup, goal is that the user has 1 click per config
  - **Ask Simon:** Kola variant and/or name?
    * Doesn't matter too much, choose what's easiest, but important to stay consistent.
- Trucks saved:
  - Reuse pre-defined table
  - **Unclear:** Should we allow saving between sessions (database hosting)? Should we allow users to access each other's saved configurations?

- Route configuration
  - First get a prebuilt route, such as BLB. And then we could add attributes to the different waypoints. These attributes should be modifiable.
  - Would like to be able to enter start point and end point by both clicking on the map and with a search field.

Dashboard
- Graphs should always have distance on x-axis
- Kola Variant (at the bottom):
  - Motor type should be prefixed based on the choice of battery type.
  - In the dashboard, show what variables/variants are taken into account in a request. So the variables we are not allowing the user to change, still show them to the user in the dashboard so the user knows what's "under the hood". Show it using Kola variants.

C
Appendix 3

`TruckData` Type Implementation

```
/**
 * Represents the data structure for a truck configuration.
 */
export type TruckData = {
  /** The KOLA variants (i.e. codes used at Volvo) required to
   ↪  represent the different parts of a truck.  */
  kolaVariants: { [key: string]: string };
  /** The model name of the vehicle (e.g., "FH") */
  vehicleModel: string;
  /** The chassis type of the truck (e.g., "Tractor", "Rigid") */
  chassis: string;
  /** The axle arrangement of the truck (e.g., "4x2", "6x4") */
  axleArrangement: string;
  /** The final drive ratio for the truck (e.g., "RAT-3.75") */
  finalGearRatio: string;
  /** The amount of electric motors in the truck */
  electricMotors: number;
  /** The amount of batteries in the truck */
  numBatteries: number;
    /** The tyre class of the vehicle. Refers to the average
     ↪  rolling resistance of all tyres (e.g., "A", "B") */
  tyreClass: string;
  /** The type of cab in the truck (e.g., "High Sleeper") */
  cabVersion: string;
  /** The kind of superstructure connected to the vehicle (e.g.,
   ↪  "Concrete Blender", "Crane", "Tipper") */
  bodyConnection: string;
  /** A string representing whether a connected superstructure is
   ↪  attached or not ("Yes" or "No") */
  trailerConnected: string;
};
```

Listing 6: Implementation of the custom `TruckData` type.

D

Appendix 4

Graph Type Implementation

```ts
/**
 * Represents a single point in the graph with X and Y values.
 */
export type GraphDataPoint = {
  x: number;
  y: number;
};
```

Listing 7: Representation of a single graph data point.

```ts
/**
 * Represents the configuration and data for a double lined graph.
 */
export interface DoubleGraphProps {
  title: string;
  xLabel: string;
  xUnit: string;
  xDomain: [number, number];
  width: number;
  height: number;

  leftLabel: string;
  leftUnit: string;
  leftDomain: [number, number];
  dataLeft: GraphDataPoint[];
  labelLeft: string;
  colorLeft: string;

  rightLabel: string;
  rightUnit: string;
  rightDomain: [number, number];
  dataRight: GraphDataPoint[];
  labelRight: string;
  colorRight: string;
}
```

Listing 8: Props used for configuring a double lined graph.

```ts
/**
 * Props for layout component managing selected axis metrics and
 ↪  callbacks.
 */
export type GraphLayoutProps = {
  xKey: string;
  leftKey: string;
  rightKey: string;
  onLeftSelect: (key: string) => void;
  onRightSelect: (key: string) => void;
};
```

Listing 9: Layout props for selecting and updating graph metrics.