COMP.SE.110: Software Design

DESIGN AND PROTOTYPE

Group05

Emilija Nikita

Teiva Trillard

Yaqin Chen

Yixin Hu

16.11.2025

1. INTRODUCTION

The purpose of this project is to design and implement a prototype that demonstrates how software systems can integrate multiple data sources to provide meaningful, data-driven insights. Our application focuses on exploring the relationship between **weather** and **road traffic** flow in Finland. This goal is reached by combining the information from two external APIs – **Digitraffic** and **Open-Meteo**. By collecting and visualizing this data, our system aims to provide a clearer understanding of how weather impacts average vehicle speed, traffic density and road conditions throughout the year.

The application will allow users to:

- View an overview of current and historical weather and traffic data for a selected area.
- Compare different locations or time periods to observe seasonal trends.
- Visualize correlations between traffic performance and environmental conditions through graphs and charts.
- Access live or recent road camera images to complement the data contextually.

From a software design perspective, the project demonstrates the use of modular architecture, API integration, and interactive visualization. The backend, built in Java using Spring Boot, retrieves and processes data from the two external APIs, while the frontend presents the results through clear, user-friendly visualizations.

This document outlines the key components, interactions, and design choices of the planned system, providing an overview of its structure and intended functionality before full implementation.

2. Use Case Description

The Use Case diagram illustrates the primary interactions between the user and the system, showing how different features and data sources are connected. The main actor is the **User**, who accesses the application through a web-based dashboard to explore, filter and visualize weather and traffic data. The system communicates with two external APIs – **Digitraffic** for traffic metrics and road camera data and **Open-Meteo** for historical weather conditions.

The diagram highlights the main functionalities of the prototype, such as viewing combined datasets, generating comparative visualizations, exploring correlations and accessing live camera images. These use cases describe the expected behavior of the system from the user's perspective, serving as a foundation for designing both the frontend interactions and backend data handling.

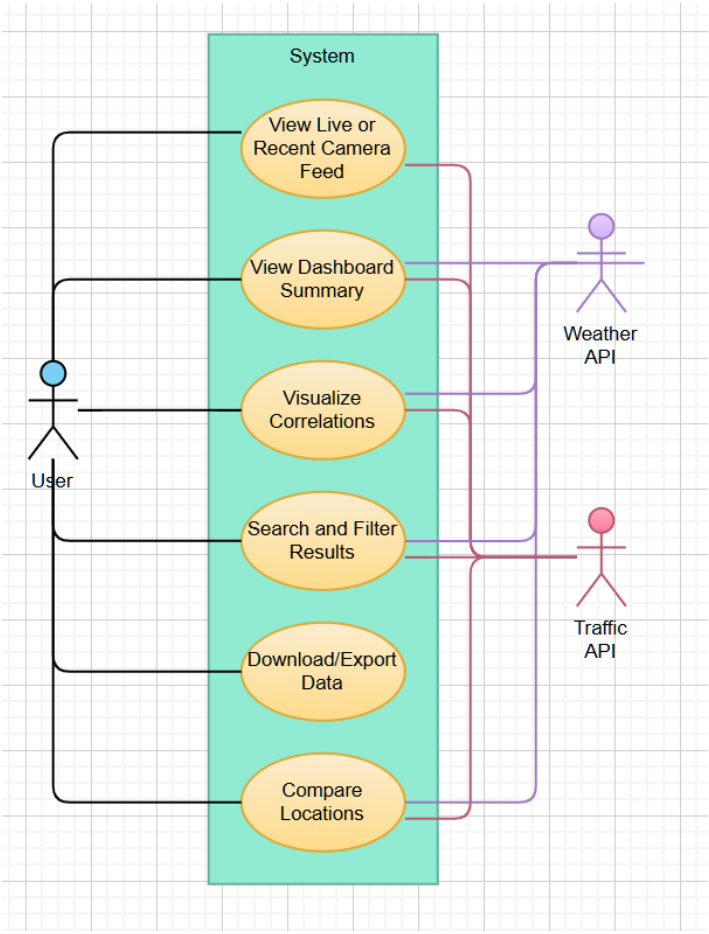


Figure 1. Use Case diagram

3. Use Cases

This sub-section describes the main use cases of the proposed system. Each use case focuses on a specific user interaction or system function, illustrating how the user can retrieve, explore, and interpret traffic and weather information through the application interface.

3.1. UC-01. View Live or Recent Camera Feed

This use case enhances the user experience by providing real-world context through road camera images. It allows users to observe actual road conditions (such as snow coverage or heavy traffic), supporting the data visualizations with live or recent imagery from Digitraffic's TMS cameras.

Table 1. Use Case 1: View Live or Recent Camera Feed

UC-01	**View Live or Recent Camera Feed**	
Goal	To allow user to view live or recently captured road camera images from selected traffic monitoring stations in Finland. This feature provides visual context for traffic and weather conditions.	
Preconditions	- User has accessed the web application. - Traffic API is available and responds. - A valid station or camera ID is selected (either automatically or by the user)	
Ordinary Sequence	**Step**	**Action**
	1.	User navigates to the "Camera Feed" section or selects a specific station from the map or list.
	2.	The system requests camera metadata (URLs, timestamps) from Digitraffic API.
	3.	The API responds with a list of recent images for that camera.
	4.	The system displays the most recent image and begins a slideshow (if multiple timestamps exist).
	5.	The user can pause, resume or manually browse through images.
Postconditions	- The latest available camera images are displayed on the dashboard - If supported, the slideshow cycles through images over a recent time range if specific time range is not selected.	
Exceptions	- **No images available**: If no images in the selected time interval exist for a selected station, a message is shown "No camera data available". - **API Unavailable**: The system displays an error message and a placeholder image. - **Slow Connection**: Low-resolution images may load first, followed by full versions.	
Trigger	The user selects a camera or station to view its live/recent feed.	
Comments	- Display timestamp and location on each image.	

	- Option to browse through the slideshow images.

3.2. UC-02. View Dashboard Summary

The dashboard summary is the main view of the application, presenting an overview of current or selected traffic and weather conditions. It acts as the entry point for most interactions, allowing the user to quickly assess overall patterns before exploring specific analyses.

Table 2. Use Case 2: View Dashboard Summary

UC-02	**View Dashboard Summary**	
Goal	To provide the user with an overview of current or selected period traffic and weather conditions in a selected Finnish city or station. The dashboard summarizes key metrics (average speed, vehicle count, temperature, precipitation etc.) in visual form.	
Preconditions	- User has accessed the web application. - APIs are reachable and respond successfully. - Location and date range (default or user-selected) are defined.	
Ordinary Sequence	**Step**	**Action**
	1.	User opens the application.
	2.	The system initializes the dashboard with a default location and date-time range.
	3.	The backend requests weather data from the Weather API and traffic from the Traffic API.
	4.	Data is merged and summarized (averages, totals, trends).
	5.	The system displays dashboard cards, charts and summary indicators (e.g., avg speed, precipitation, daylight duration).
	6.	The dashboard automatically refreshes or allows user-triggered refresh.
Postconditions	- Dashboard displays the most recent or selected period data summary - Data visualization components (charts, image slideshow, tables, etc.) are updated with new values.	
Exceptions	- API Unavailable: If one API fails, the dashboard displays partial data with “Data not available” notice.	
Trigger	The user opens the dashboard or selects a new date/location to view.	
Comments	- Dashboard contains visual elements such as charts and tables. - Units are shown in metric format (°C, km/h, mm).	

3.3. UC-03. Visualize Correlations

Through this use case, users can examine how weather parameters (e.g., precipitation, temperature, daylight duration) relate to traffic flow metrics (vehicle speed, count). The system merges datasets and generates interactive visualizations, helping to uncover patterns such as reduced speeds during rain or snow.

Table 3. Use Case 3: Visualize Correlations

UC-03	**Visualize Correlations**	
Goal	To allow the user to explore and visualize the relationship between traffic and weather parameters (e.g., how precipitation or temperature affects vehicle speed or traffic volume). The system presents the results as interactive charts or graphs.	
Preconditions	- The user has accessed the dashboard and selected at least one time interval and location. - Weather and traffic data have been successfully fetched from the APIs. - Visualization module is available and initialized	
Ordinary Sequence	**Step**	**Action**
	1.	User navigates to the “Correlations” section of the dashboard.
	2.	User selects two or more variables to compare.
	3.	The system requests the relevant data (if not already loaded) from the APIs.
	4.	The backend merges and aligns datasets based on time and location.
	5.	The system calculates correlations or statistical relationships and renders the visualizations (e.g., scatter plot, trend line, bar chart).
	6.	The user can interact with the visualization (hover for details, zoom or change axes).
Postconditions	- A correlation visualization (e.g., line chart, scatter plot) is generated and displayed. - The user can filter or select data or change date ranges to update the visualization.	
Exceptions	- **Incomplete data**: if one variable lacks data, a message is displayed (“Insufficient data for correlation”). - **API unavailabe**: the system notifies the user (“APIs are unavailable. Try again later.”).	
Trigger	The user selects the “Correlations” tab or chooses to compare specific weather and traffic parameters.	
Comments	- Provide dropdown menus for variable selection (e.g., temperature, precipitation, avg speed). - Include tooltips showing exact values and timestamps. - Visualization should be clear, responsive and color accessible.	

3.4. UC-04. Search and Filter Results

The search and filtering functionality allows users to refine the dataset by location, time range or weather and traffic variables. This ensures that the dashboard remains relevant to the user's are of interest and provides focused, interpretable results for analysis.

Table 4. Use Case 4: Search and Filter Results

UC-04	**Search and Filter Results**	
Goal	To allow the user to search for specific stations, cities or time periods and apply filters (e.g., by date, weather condition or traffic parameter) to customize the displayed data on the dashboard.	
Primary actor	User	
Preconditions	- The user has opened the dashboard view. - Weather and traffic data endpoints are working. - The interface components (search bar, filters) are loaded and active.	
Ordinary Sequence	**Step**	**Action**
	1.	User navigates to the main dashboard or results page.
	2.	User types a query into the search bar or selects filters.
	3.	The system validates the query or filter selections.
	4.	The backend retrieves the corresponding subset of traffic and weather data.
	5.	Dashboard visualizations and summary statistics update accordingly.
	6.	User can clear filters or modify them to explore different results.
Postconditions	- Dashboard updates to display only the data matching the user's search or filters.	
Exceptions	- **Invalid Search Query**: if no results match, the system displays "No data available" message. - **API Timeout**: if filtering triggers an API request that fails, the dashboard displays partial data where available. - **Empty filters**: if no filter is applied, the system displays all available data for the last 30 days.	
Trigger	The user clicks "search" button after entering search query or selecting filters.	
Comments	- Filters should be intuitive and easily resettable. - Display active filters clearly at the top of the dashboard. - Use loading indicators while updating filtered results.	

3.5. UC-05. Download/Export Data

This use case enables users to export selected data or visualizations for offline use. It supports the project's goal of openness and data accessibility without requiring local database storage.

Table 5. Use Case 5: Download/Export Data

UC-05	**Download/Export Data**	
Goal	To allow the user to download or export selected weather and traffic datasets or generated visualizations for offline use.	
Preconditions	- The user has performed a search, filter or comparison that generated results. - The displayed data is already fetched and available in the system memory. - The user selected the data to export. - Export functionality is available and connected to relevant data objects.	
Ordinary Sequence	**Step**	**Action**
	1.	User selects "Download" or "Export" option from the dashboard or chart view.
	2.	The system prompts the user to choose the export format (CSV, JSON, PNG, PDF).
	3.	The user confirms the download request.
	4.	The system generated the file with the current dataset or chart contents.
	5.	The file is made available for download and saved locally on the user's device.
	6.	A success message is displayed ("File exported successfully").
Postconditions	- The selected data or visualization is exported and downloaded as a file (e.g., CSV, JSON or PNG). - A confirmation is show upon successful download.	
Exceptions	- **File generation error**: if export fails, a message prompts user to retry.	
Trigger	The user clicks the "Download" or "Export" button on a chart, table or comparison view.	
Comments	- Provide export format options (e.g., CSV for data, PNG for charts). - Include timestamp and metadata (e.g., location, date range) in exported files. - Keep the process lightweight (no server-side file storage).	

3.6. UC-06. Compare Locations

This use case builds on the filtering functionality by enabling direct comparisons between multiple locations or time periods. Users can, for instance, compare how weather impacts traffic differently in Helsinki versus Tampere, providing valuable information into regional variations.

Table 6. Use Case 6: Compare Locations

UC-06	**Compare Locations**	
Goal	To allow the user to compare weather and traffic conditions between two or more locations or time periods. The feature helps to identify how different cities or stations respond to varying weather patterns in terms of traffic flow and speed.	
Primary actor	User	
Preconditions	- The user has opened the dashboard view and selected at least two location stations. - The APIs are available and return data for all selected locations. - Comparison view components are initialized.	
Ordinary Sequence	**Step**	**Action**
	1.	User navigates to the "Compare Locations" section of the dashboard.
	2.	User selects multiple cities, road stations or months to compare.
	3.	The system fetches weather and traffic data for each selected location/time range vis APIs.
	4.	The system generates visual comparisons such as side-by-side graphs, dual-line charts or summary tables.
	5.	The user reviews differences and may adjust filters or comparison parameters
Postconditions	- The dashboard displays comparative charts, summaries or tables showing weather and traffic metrics across locations or time ranges. - The user can toggle between different variables.	
Exceptions	- **Missing data for a location**: if one city lacks sufficient data, the system shows partial results and a warning message. - **API Timeout**: display results for locations that successfully returned data with an error message for failed ones.	
Trigger	The user selects multiple locations, stations or periods to compare from the dashboard interface.	
Comments	- Limit the number of locations simultaneously compared. - Provide clear legends and color-coding for each location. - Visualizations should update dynamically when filters change.	

4. Functional Design

4.1. Key Features

- Displaying the key traffic and weather information on the main dashboard
- Displaying the feed from a traffic camera around the area specified
- Searching for a specific city / area
- Obtain old traffic data

4.2. User Interface Mockups

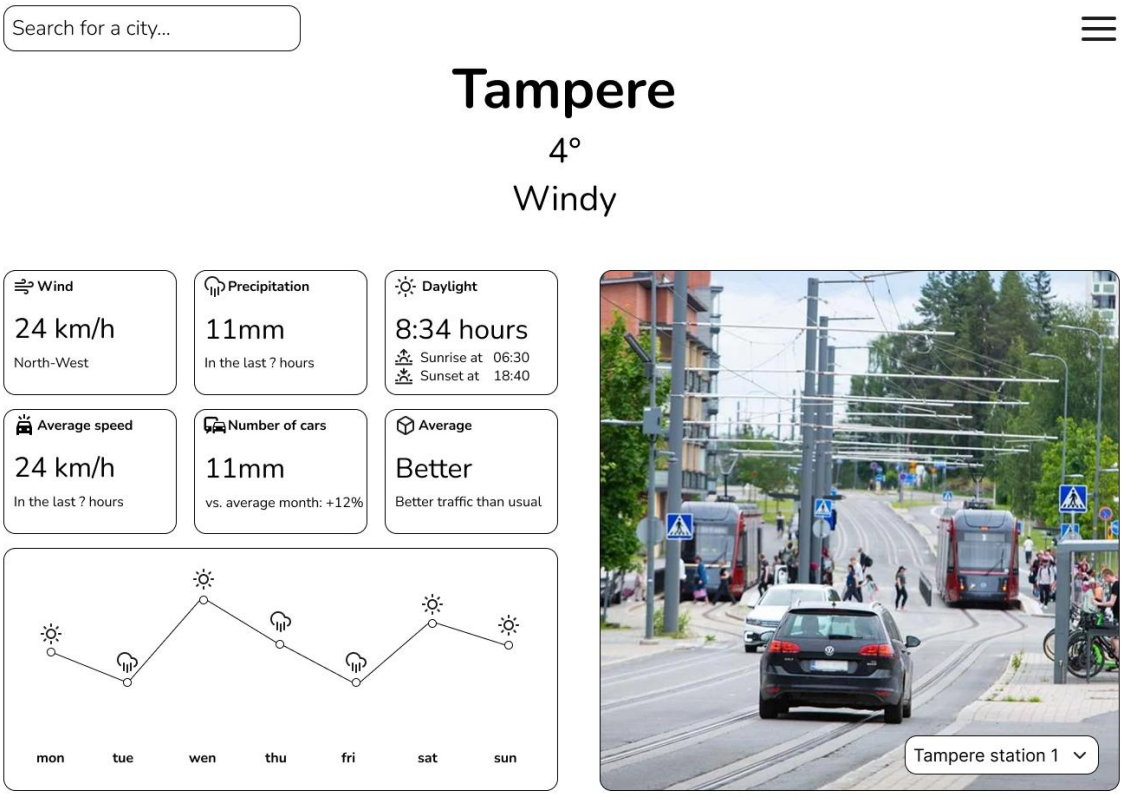


Figure 2. Dashboard page mockup

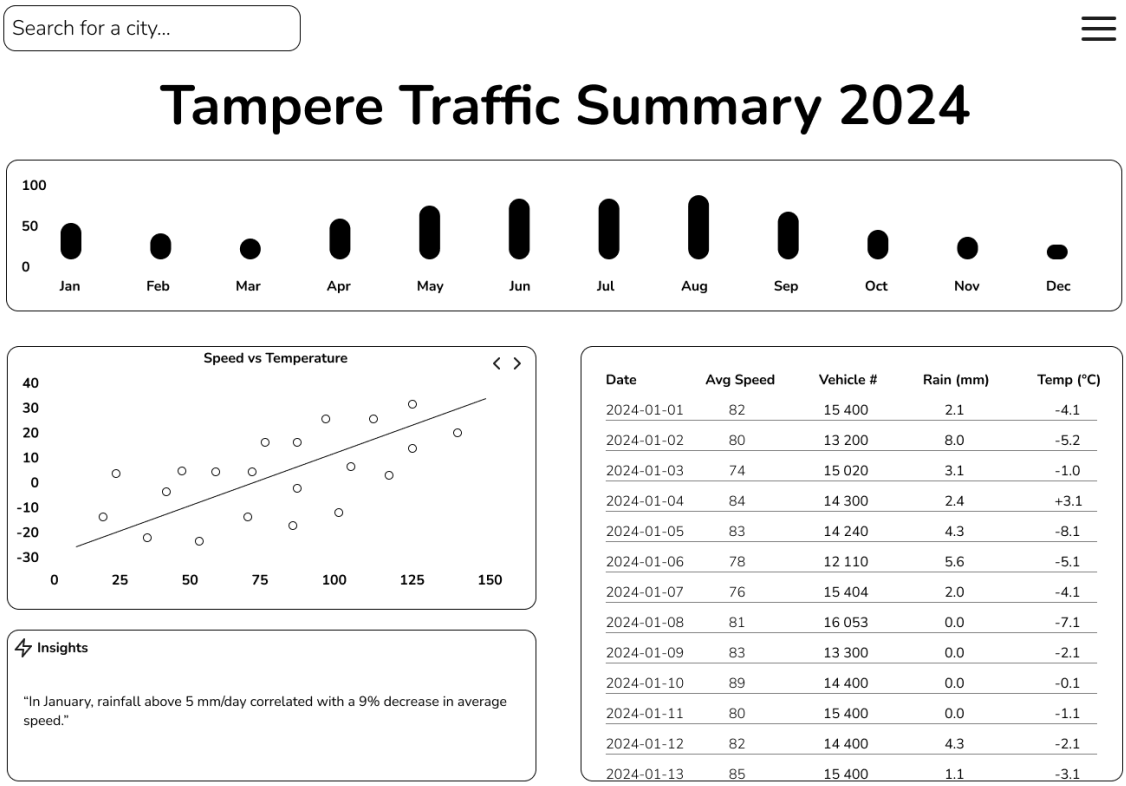


Figure 3. Correlation View

4.3. Interaction Flow Example

https://www.figma.com/proto/h3gaLZLAsXo4hSev38veD6/Software-design-prototype?page-id=0%3A1&node-id=1-23&viewport=-1%2C-96%2C0.28&t=t6JLyVu9nxRdf2M1-1&scaling=min-zoom&content-scaling=fixed&starting-point-node-id=1%3A23

5. Technical Design

The system is designed as a web-based data visualization platform that integrates two external APIs:

- **Digitrafic API (TMS Data).** Provides live and historical traffic information such as average speed, vehicle counts and camera images from Finnish road stations.
- **Open-Meteo API (Historical Weather)**. Provides weather parameters such as temperature, precipitation, cloud cover, wind speed and daylight duration for any location in Finland (it allows using coordinates).

The frontend communicates with the backend through RESTful API endpoints. The backend is responsible for fetching and merging data from external APIs, while the frontend visualizes this combined information through interactive graphs and maps.

The simplified communication between Components are illustrated in component diagram below (see **Klaida! Nerastas nuorodos šaltinis.**):

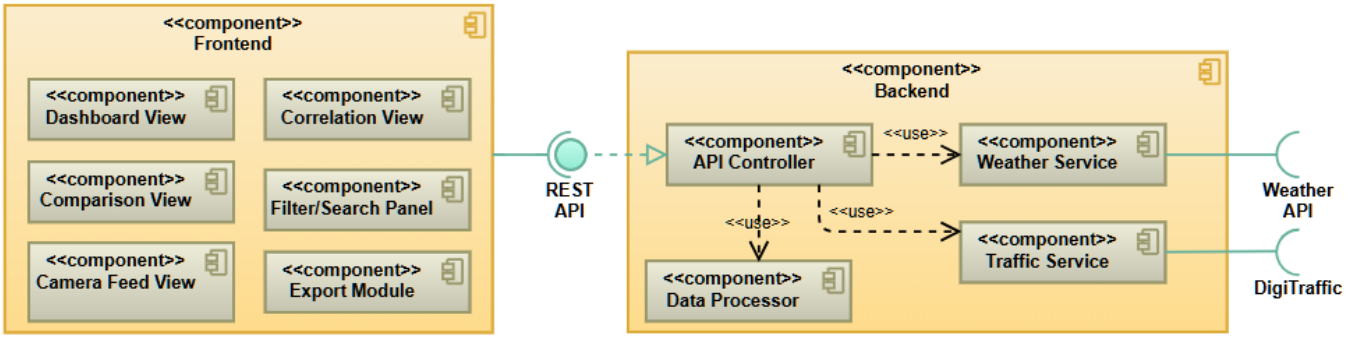


Figure 4. Component Diagram

5.1. Backend Structure

The backend acts as an intermediary layer between the frontend and external APIs. It is implemented in **Java** using **Spring Boot** framework. It serves as the core logic layer that retrieves, processes and integrates data. The backend exposes several REST enpoints that the frontend can query. Data is fetched in real time, processed using Java Streams and returned in a unified JSON format. Key technologies include **Java** + **Spring Boot**, **`Spring WebClient`** for HTTP communication and **`Junit`** for unit testing.

Spring Boot was chosen for the backend because it provides a mature, reliable, and well-supported framework for building RESTful services. It offers built-in support for dependency injections, modular architecture, and structured layering (controllers,

services, repositories), which aligns well with the project's goal of integrating multiple external data sources. Spring Boot simplifies common backend tasks such as API routing, asynchronous HTTP calls, caching, validation, and scheduled tasks - features that are valuable when fetching real-time traffic data and historical weather information. Its ecosystem (Spring Web, WebClient, actuator, testing libraries) allows the group to focus on implementing core logic rather than configuring low-level infrastructure. Additionally, Java's strong typing and Spring's robust tooling (IntelliJ, Maven, etc.) help prevent errors early, improve maintainability, and make the backend easier to extend later.

5.1.1. Controllers

- Controllers
    - Each endpoint + responsibility

CombinedController

Endpoint		`GET /api/combined`
Fetches combined traffic and weather information		
Parameters	`lat`	Latitude value, double
	`lon`	Longitude value, double

HelloController

Endpoint	`GET /hello`
Test endpoint	

TrafficController

Endpoint	`GET /api/traffic`
Fetches traffic information	

WeatherController

Endpoint		`GET /api/weather`
Fetches weather information		
Parameters	`lat`	Latitude value, double
	`lon`	Longitude value, double

5.2. Frontend Structure

The frontend is implemented as a single page web application (SPA) using React.js for responsive design.

React was selected as the frontend framework because it is widely adopted, component-based, and ideal for building highly interactive dashboards and visualizations. Its state management model and reusable components make it easier to implement dynamic UI elements such as charts, filters, map views, and camera feeds. React also integrates well with modern visualization libraries and REST APIs, which is essential for our data-driven application.

Vite was chosen as the build tool because it provides extremely fast startup times, hot-module replacement, and a much smoother development experience. This reduces wait times during frontend development and allows the team to iterate quickly on UI design. Vite also generates optimized production builds with minimal configuration, making deployment straightforward. Together, React + Vite offer a lightweight, modern, and highly productive development stack for creating a smooth and responsive user interface.

In the prototype part it was considered to use JS libraries such as Recharts or Plotly.js for visualization purposes. But in the implementation stage it was discovered that **react-chartjs-2** fit our project better.

We chose **react-chartjs-2**, a React wrapper for the Chart.js library, because it provides a powerful and flexible solution for creating interactive data visualizations. Chart.js offers a wide set of chart types: line, bar, scatter, radar, doughnut, and more. That suits our need to display time-series weather data and traffic metrics in a clear and interpretable way. Since the library renders charts using the Canvas API, it is well-optimized for dynamic data updates and handles moderate data volumes smoothly. The react-chartjs-2 wrapper integrates naturally with React's component model, making it easy to re-render charts when users apply filters, change locations, or select different time ranges. The library strikes a good balance between ease of use and customizability, allowing the team to design clean and accessible visualizations without needing low-level graphic programming. Overall, react-chartjs-2 was a practical choice for building an interactive dashboard within the project timeline.

Main Frontend Components:

- Dashboard View – overview of traffic and weather statistics.
- Correlation View – graphs showing relationships between weather and traffic data.
- Comparison View – side-by-side location or time period comparisons.

- Filter/Search Panel – enables data filtering by date, station, city.
- Camera Feed View – creates a slideshow of saved station camera images for a chosen time interval.
- Export Module – used to generate exportable files (graphs, datasets, etc.).

All frontend components communicate with the backend through REST calls (e.g., `/api/weather`, `/api/traffic`, `/api/correlations`, etc.)

6. Boundaries and Interfaces

6.1. Backend Services

The backend services form the **core logic layer** of the application, responsible for processing, transforming, and aggregating data fetched from external APIs. Each service encapsulates specific functionality, exposing methods used by controllers or other services.

Service	Responsibility	Interfaces / Methods	Consumers
WeatherService	Fetches and processes weather data from Open-Meteo API. Handles caching, filtering, and basic aggregation.	`getWeather(lat, lon, startDate, endDate) -> returns List<WeatherDTO>`	CombinedController, CorrelationView, ComparisonView
TrafficService	Retrieves traffic data including vehicle count, average speed, and density from Digitraffic API.	`getTrafficData(stationId, startDate, endDate) → returns List<TrafficDTO>`	CombinedController, CorrelationView, ComparisonView
AggregationService	Combines traffic and weather datasets into a single unified model aligned by timestamp and location. Performs basic statistical computations.	`getMerged(String stationId, Instant from, Instant to, double lat, double lon)`	CombinedController, ComparisonView

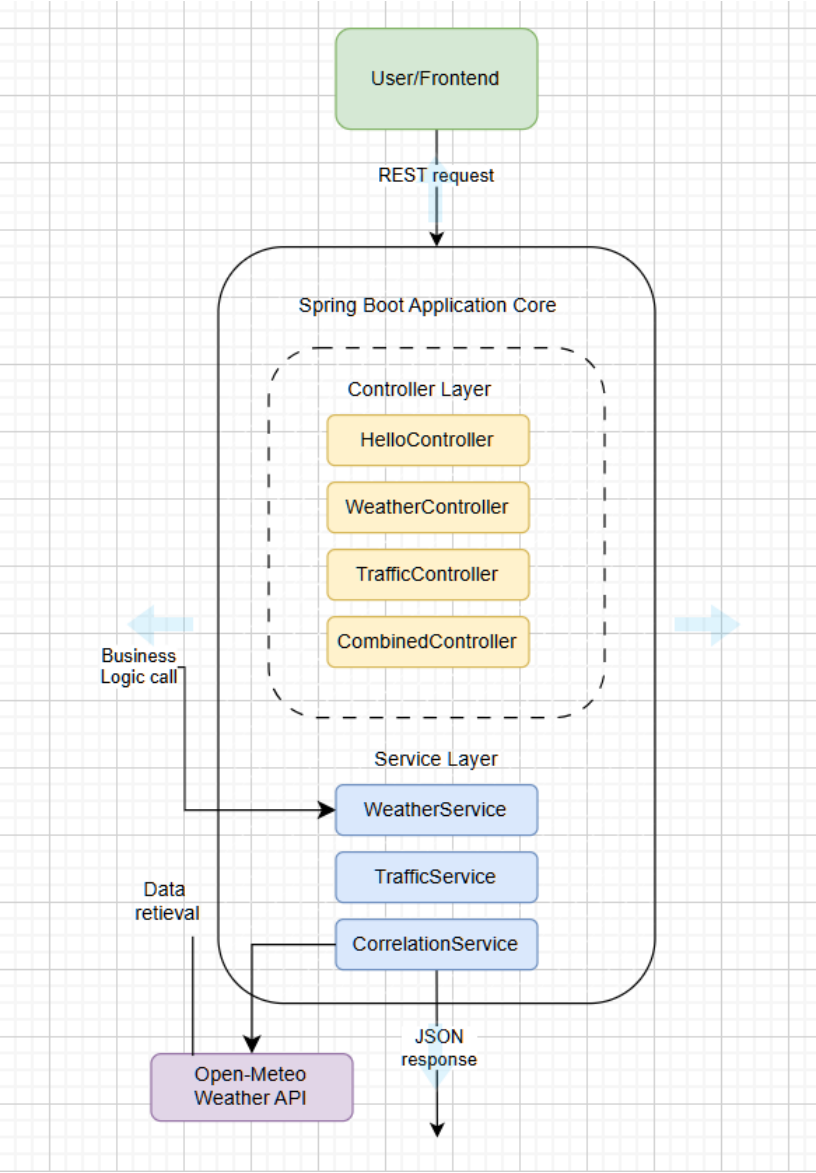
TO BE CLARIFIED FURTHER...

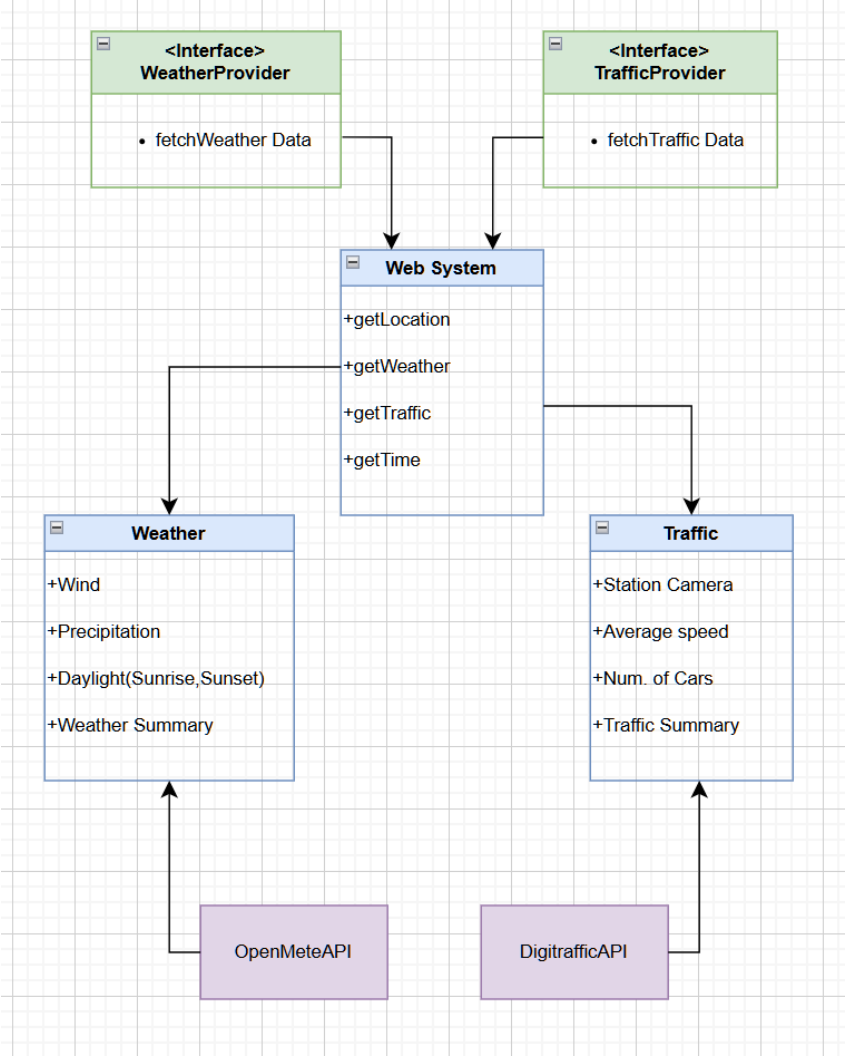
6.2. Frontend Interfaces

The frontend is a **React.js SPA** that consumes backend REST endpoints to display traffic, weather, and camera data. This subsection details the interfaces between the frontend components and backend services. So far there are no components present – everything is written in one file as a merged system. In the future, it is planned to have the following list of frontend components.

Component	Backend Endpoint	Request Parameters	Expected JSON Response	Notes
Dashboard View	/api/combined	`lat, lon, startDate, endDate`	`{ trafficMetrics[], weatherMetrics[], location, dateRange }`	Displays summary cards and charts.
CorrelationView	/api/combined	`lat, lon, startDate, endDate, variables[]`	`{ trafficMetrics[], weatherMetrics[] }`	Generates interactive scatter or line charts for correlation analysis.

ComparisonView	/api/combined	`Multiple lat/lon pairs, startDate, endDate`	`{ location1[], location2[], ... }`	Side-by-side comparisons for multiple locations/time periods.
CameraFeedView	/api/camera	`stationId`	`{ imageURL, timestamp }[]`	Slideshow of live or recent camera images.





6.3. External Interfaces

The system relies on two external APIs for data: **Digitraffic API** for traffic and camera data, and **Open-Meteo API** for historical weather data. These APIs act as **suppliers**, while our backend services are the **demanders**.

6.3.1. Digitraffic API

Purpose: Provides traffic measurements and live/recent camera images for Finnish roads.

Endpoint	Request Parameters	Response	Notes
/tms/api/v1/stations/{stationId}/measurements	`stationId, startDate, endDate`	`{ avgSpeed, vehicleCount, density, timestamp }`	Used by TrafficService to fetch historical traffic data.
/tms/api/v1/camera-images/{stationId}	`stationId`	`{ imageURL, timestamp, stationId }`	Used by CameraService for live/recent road images.

6.3.2. Open-Meteo API

Purpose: Provides historical weather data for any location in Finland using latitude and longitude coordinates.

Endpoint	Request Parameters	Response	Notes
/v1/forecast	`latitude, longitude, start_date, end_date, parameters`	`{ temperature[], precipitation[], windSpeed[], daylightDuration[], cloudCover[], timestamp[] }`	Used by WeatherService for historical weather retrieval.

7. Self-Evaluation

This section evaluates the implemented prototype in terms of functionality, adherence to design, encountered challenges, and lessons learned. It also explains the reasoning behind key design decisions.

7.1. Prototype Adherence

- **Implemented Features:**
    - Dashboard summary with key traffic and weather metrics.
    - Correlation visualizations showing relationships between traffic and weather parameters.
    - Comparison view for multiple locations or time periods.
- **Changes from Original Design:**
    - Visualization library: initially considered **Recharts**, but **react-chartjs-2** was chosen for better performance and integration with React.
    - UI adjustments: some mock-up layouts were modified for better usability.
    - Additional error handling should be added for API unavailability and missing data.
    - We decided to wrap up the whole application in Docker compose so that the application deployment would be easier for developers and TA.
- **Pending Improvements:**
    - Search and filter functionality for location, date range, and variables.
    - Live/recent camera feed from Digitraffic.
    - Implement caching for repeated API requests to improve performance.
    - Export functionality (CSV, JSON, PNG) for charts and datasets.
    - Add pagination or lazy loading for long time-series data.
    - Include more comprehensive unit and integration tests.
    - Optional: authentication or user preferences for personalized dashboards.

7.2. Problems Encountered

- **API Limitations:** Digitraffic rate limits required throttling of requests, which impacted the responsiveness of live dashboards.
- **Visualization Performance:** Rendering large datasets with Chart.js required optimization to prevent slow re-renders.

7.3. Design Support for Further Implementation

- The **layered architecture** (Controllers → Services → API Clients) allows adding new APIs or endpoints without affecting the frontend.
- The **modular frontend components** (DashboardView, CorrelationView, ComparisonView) support adding new visualizations or filters with minimal changes to existing code.
- Using **React Query** and Context API ensures state management scales as new pages or features are added.

7.4. Lessons Learned

- Planning a clear separation between **data fetching, processing, and visualization** simplifies both development and debugging.
- Modular and reusable frontend components make it easier to add new functionality without breaking existing features.
- Early evaluation of API limitations is crucial to avoid unexpected delays or data gaps.
- Documenting **DTOs and internal interfaces** early ensures consistent communication between backend and frontend components.