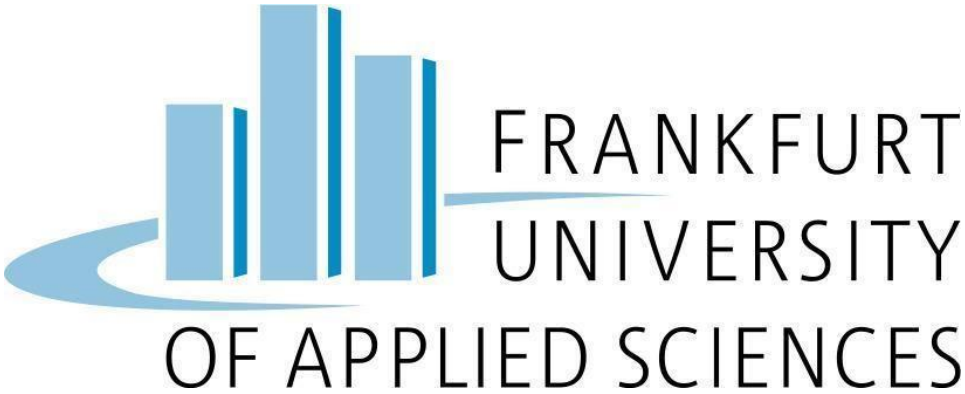


DOCUMENTATION

Object-Oriented Programming in Java

Project

Winter Semester 24-25

Real-Time Traffic Simulation with Java

Lecturer: Prof. Dr. Ghadi Mahmoudi

Faculty: Computer Science, Frankfurt University of Applied Sciences

Acknowledgements

We express our sincere gratitude to the FRANKFURT UNIVERSITY OF APPLIED SCIENCES for creating an environment that promotes research and creativity. The resources and assistance offered by the institution were pivotal in bringing this project to fruition.

We would like to especially thank Prof. Dr. Ghadi Mahmoudi for his exceptional mentorship and unwavering support throughout the development of this project. His profound knowledge and thoughtful advice have been crucial in guiding our efforts and shaping the outcomes.

Our thanks also go to the open-source community, whose extensive tools and libraries were valuable during the development process.

This project has been a rewarding learning experience, and We are grateful to everyone who played a part, directly or indirectly, in the completion of this project.

Table of Contents

List of Figures

(GUI ↔ SimulationController ↔ TraaS/SUMO ↔ Metrics/Export)

Group Members

Name	Matriculation Number
Akshat Kumar Sharma	1596535
Ranjit Jaiswal	1582576

Project Overview

Traffic Grid Simulation is a Java-based application that helps users simulate and monitor traffic flow in real time using the SUMO (Simulation of Urban Mobility) traffic simulator through the TraCI/TraaS interface. It visualizes a road network on an interactive map and displays key runtime information such as active vehicles, filtered visible vehicles, average waiting time, congestion index, throughput (vehicles/hour), and mean speed through a user-friendly dashboard. The system also provides a live trend chart to help users quickly understand how traffic performance changes over time.

This documentation explains how the simulation works, how data is collected and processed, and how results are presented in the GUI. It covers the interaction between the backend simulation controller and the frontend interface, the logic for spawning vehicles on predefined routes, the traffic light control features (manual control and optional rule-based stop/go control), and the filtering system used to visualize only selected vehicle types or speeds. It also describes how errors are handled and logged to keep the application stable during runtime.

The goal of this documentation is to help developers and future contributors understand the system architecture and implementation, so they can maintain, extend, or improve features such as routing strategies, metrics, map interaction, reporting (CSV/PDF export), and advanced traffic-light control rules.

2. User Handbook

2.1 Requirements and Setup

1. **Install Java + JDK 22**
   Make sure Java is installed and your project is configured to use **JDK 22**.
2. **Install and set up SUMO (with GUI)**
   Ensure **SUMO** is installed on your machine and `sumo-gui` is accessible from the terminal/command prompt (added to PATH).
3. **Keep required SUMO project files in the correct folder**
   Place these files in the same working directory as the program (or ensure paths are correct): `final.sumocfg` , `final.net.xml` route file(s) referenced inside `final.sumocfg` (e.g., `.rou.xml`)
4. **Run the Main class**
   Start the application by running the **`Main`** class (`org.example.Main`).
   The GUI dashboard will open and the system will validate that `final.sumocfg` exists.
5. **Start simulation from the GUI**
   Press **Start** to begin stepping the SUMO simulation.
   After SUMO starts, the route dropdown and traffic light dropdown will populate automatically.
6. **If the map/roads or labels don't appear properly**
   Ensure the net file path inside `final.sumocfg` is correct (the program reads **net-file** and loads road geometry + TLS positions from it).
7. **Exporting results (CSV/PDF)**
   Use **Export CSV** for metrics logging and **Export PDF** for a one-page summary report (includes the live trend chart snapshot).
8. **Running as JAR (optional)**
   If you have a runnable JAR, open terminal in the JAR folder and run:
   `java -jar traffic-sim.jar`
   (Note: SUMO must still be installed and accessible via PATH for `sumo-gui` to launch.)

2.2 Control Panel

2.2.1 Simulation Controls

Figure 1.1: Simulation Controls

This figure shows the **Simulation Controls** section of the GUI. The **Start** button launches the SUMO simulation and begins the live updates on the map and metrics panel. The **Stop** button terminates the simulation loop safely. The **Export CSV** button saves the collected runtime metrics (e.g., active vehicles, congestion, waiting time, throughput) into a CSV file, while the **Export PDF** button generates a one-page summary report that includes key statistics and a snapshot of the live trend chart.

2.2.2 Simulation Speed Control (1×–10×)

Figure 1.2: **Simulation Speed Control (1×–10×)**

This figure shows the **Speed slider** used to control the runtime speed of the traffic simulation. By moving the slider from **1× to 10×**, the user increases or decreases how fast the simulation steps are executed (i.e., the delay between SUMO simulation steps). A higher speed factor runs the simulation faster and updates the map/metrics more frequently, while a lower factor slows it down for easier observation and debugging.

2.2.3. Route / Scenario Selection Dropdown (Same Destination, Long Variants)

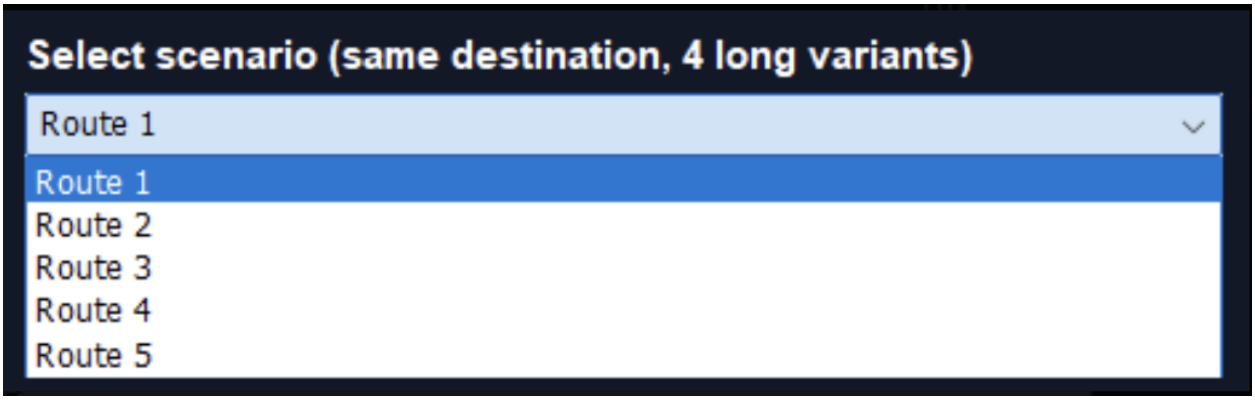


Figure 1.3: **Route / Scenario Selection Dropdown (Same Destination, Long Variants)**

This figure shows the **route selection dropdown** used to choose a traffic scenario before spawning vehicles. Each option (Route 1, Route 2, Route 3, Route 4, etc.) represents a **different long route variant** that still leads to the **same destination**, allowing comparison of traffic behavior across alternative paths. After selecting a route, vehicles spawned via the **Car/Truck/Bus** buttons are injected onto the chosen route, making it easy to test congestion, waiting time, throughput, and traffic-light effects under different routing choices.

2.2.4. Number of Vehicles

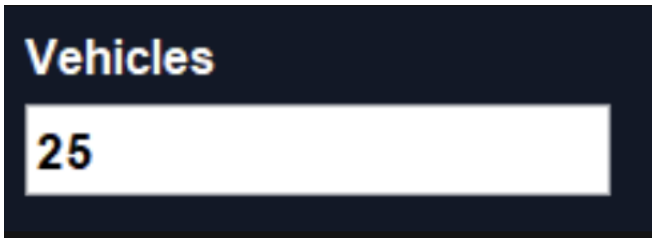


Figure 1.4: **Number of Vehicles Input Field**

This figure shows the **“Vehicles”** input field, where the user specifies how many vehicles should be spawned into the simulation at once. The entered value (e.g., **25**) is used when the user clicks any **Spawn Type** button (**Car**, **Truck**, or **Bus**). After pressing one of these buttons, the system injects the selected number of vehicles onto the currently chosen route/scenario, enabling controlled load testing and comparison of traffic metrics such as congestion, average waiting time, throughput, and mean speed.

2.2.5. **Spawn Type Buttons (Car / Truck / Bus)**

Figure 1.5: **Spawn Type Buttons (Car / Truck / Bus)**

This figure shows the **Spawn Type** controls used to inject different categories of vehicles into the simulation. By clicking **Car**, **Truck**, or **Bus**, the system spawns the selected vehicle type onto the currently chosen route/scenario. The number of vehicles created is taken from the **Vehicles** input field, allowing the user to generate controlled traffic loads and observe how different vehicle mixes impact congestion, waiting time, throughput, and overall traffic behavior.

2.2.6. Map Filtering

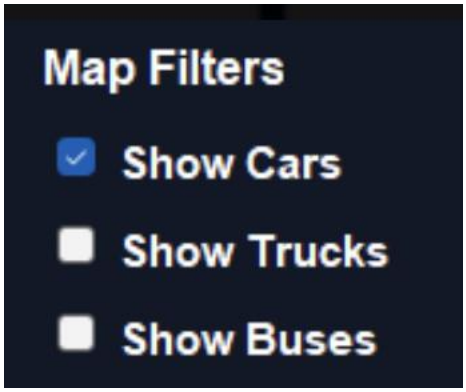


Figure 1.6 : **Map Filters (Vehicle Type Visibility)**

This figure shows the **Map Filters** section used to control which vehicle types are displayed on the map. By enabling or disabling **Show Cars**, **Show Trucks**, and **Show Buses**, the user can filter the visualization so only the selected categories remain visible. This filtering affects the **visible vehicles count** in the metrics

panel (filtered view) while the simulation itself continues running with all vehicles, making it easier to analyze traffic behavior for specific vehicle types without clutter.

2.2.7. Minimum Speed Filter Slider

Figure 1.7: **Minimum Speed Filter Slider**

This figure shows the **Min Speed Filter** control used to filter vehicles on the map based on their current speed. The slider sets a threshold value (shown in **m/s** and converted to **km/h**), and only vehicles moving at or above this minimum speed remain visible in the map view. This helps highlight **slow/stopped vehicles** (by setting a low threshold) or focus only on **actively moving traffic** (by increasing the threshold), reducing clutter and making congestion analysis easier.

2.3 Traffic Light Control and Selection (Manual TLS Management)

Figure 1.8: **Traffic Light Control Panel with Expanded TLS Selection List**

This figure shows the **complete Traffic Light control module** of the dashboard. The upper part contains the **TLS dropdown** (currently selected `t1`) and the manual control buttons **Red**, **Green**, and **Reset**. The lower part shows the **expanded dropdown list**, displaying all available traffic lights (`t1, t2, …`) with their corresponding SUMO IDs. By selecting a TLS from the list, the user can apply manual overrides (force Red/Green) for testing traffic behavior, and use **Reset** to restore the selected signal back to its default automatic program.

2.3.1 **Rule-Based Traffic Light Control**

Figure 1.9: **Rule-Based TLS Toggle (Stop/Go Control)**

When the **"Rule-based TLS (stop/go)"** option is enabled, the system controls each traffic light automatically based on **traffic demand** near that junction. In every simulation step, it checks the lanes controlled by the selected TLS and reads the **number of halting vehicles** (or, as a fallback, the number of vehicles present).

- **If demand = 0 (no vehicles waiting):** the traffic light is returned to its **default SUMO program (AUTO)**.
- **If demand > 0 (vehicles detected):** the light switches into a fixed stop/go cycle:
  - **RED for 6 seconds** (vehicles are stopped / held)
  - then **GREEN for 12 seconds** (vehicles are released)
  - then repeats **6s RED → 12s GREEN** as long as demand stays > 0.

Manual override has priority: if the user forces a light to **Red** or **Green** using the manual buttons, the rule-based logic does not overwrite that traffic light until the manual mode is reset.

2.4 **Live Traffic Map Visualization (Road Network + Vehicles + TLS Markers)**

Figure 2.1 : **Real-Time Map View Showing Road Network, Vehicles, and Traffic Light Nodes**

This figure shows the **main simulation map panel** where the road network is rendered from the SUMO `net.xml` lane shapes and displayed in a clean, zoomable view. Vehicles are drawn in real time using their **current SUMO positions** and are color/shape-coded by type (**car, truck, bus**). The small labeled markers (e.g., `t1`, `t2`, …) represent **traffic light locations**, loaded from junction/TLS data in the network file.

The map updates continuously as the simulation steps forward, and it also respects the dashboard filters (vehicle type visibility and minimum speed filter). Users can interact with the map using **zoom (mouse wheel), pan (left-drag), and rotation (right-drag)** to inspect congestion areas and traffic flow behavior around junctions.

2.5 **Live Metrics Trend Chart (Real-Time Performance Monitoring)**

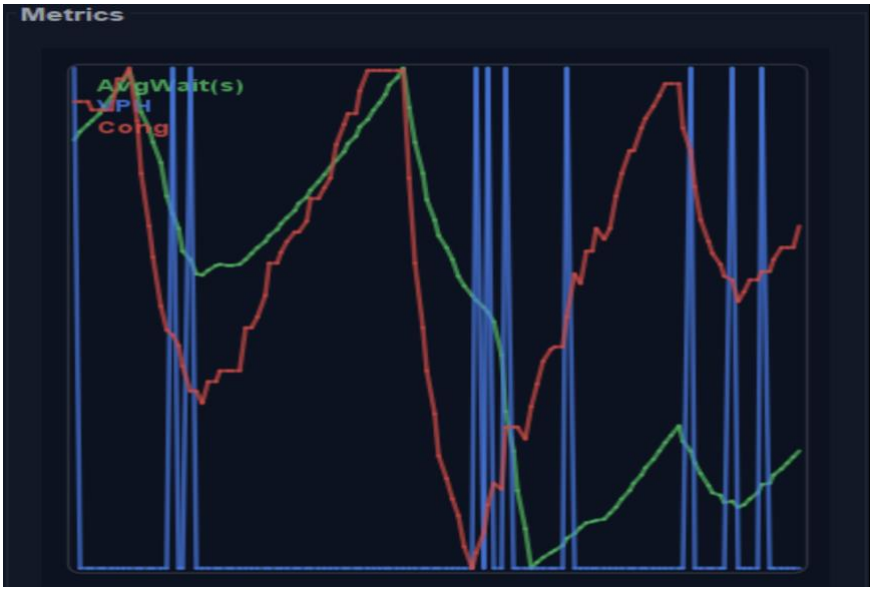


Figure 2.2 **: Live Trend Chart for Avg Wait Time, Throughput, and Congestion**

This figure shows the **live trend chart panel** on the Metrics side of the GUI. The chart plots three time-series that are continuously updated while the SUMO simulation runs:

- **AvgWait(s)**: the average vehicle waiting time (in seconds), calculated from the vehicles currently active in the simulation.
- **VPH (Vehicles per Hour)**: the estimated throughput, computed using a rolling time window (your code uses a **300-second window**) and converted to vehicles/hour.
- **Cong (Congestion Index)**: a congestion ratio derived from the fraction of vehicles that are nearly stopped (speed < 0.1 m/s) compared to all active vehicles.

The graph helps quickly visualize how traffic conditions evolve over time—e.g., congestion spikes usually push **AvgWait(s)** up and can reduce **throughput**, making it easy to compare the impact of route choices, vehicle injections, and traffic-light control strategies.

2.6 **Real-Time Metrics Summary (Simulation KPIs Panel)**

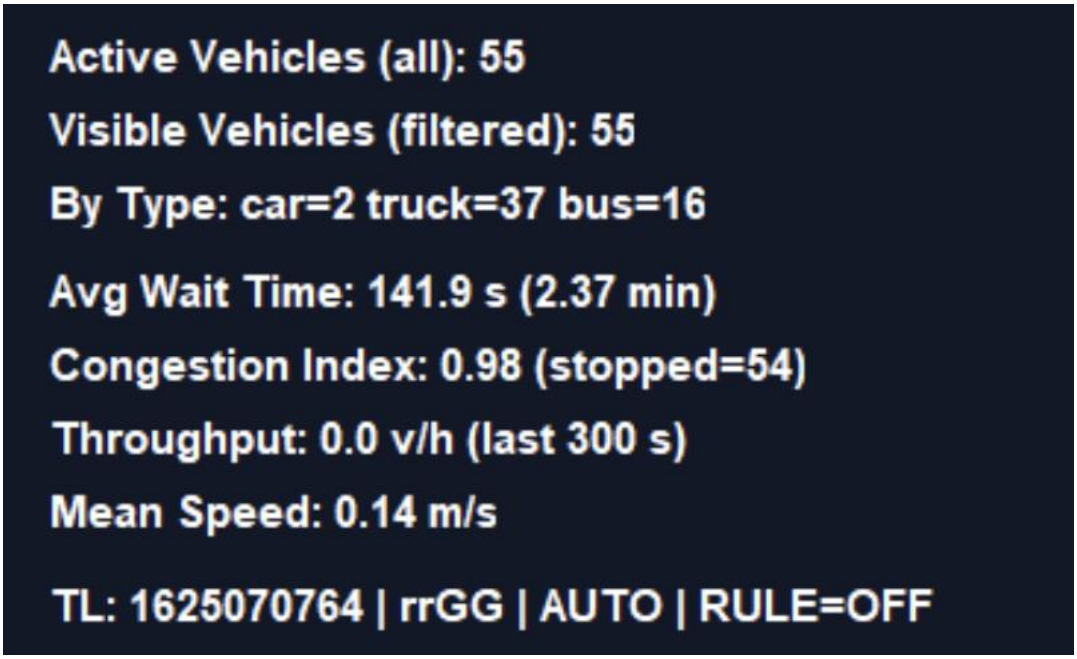


Figure 2.3 : **Metrics Panel Showing Live Vehicle Counts, Waiting Time, Congestion, Throughput, Speed, and TLS Status**

This figure shows the **live KPI summary** that updates every simulation step and gives a quick "health report" of the traffic system:

- **Active Vehicles (all):** total vehicles currently present in the SUMO simulation.
- **Visible Vehicles (filtered):** vehicles currently shown on the map after applying the GUI filters (type + minimum speed).
- **By Type:** breakdown of visible vehicles into **car, truck, and bus** counts.
- **Avg Wait Time:** average waiting time across active vehicles (displayed in seconds and minutes).
- **Congestion Index:** ratio of **stopped vehicles (speed < 0.1 m/s)** to total active vehicles; the text also shows how many are stopped.
- **Throughput (v/h):** vehicles per hour computed using a rolling **300-second window** ("last 300 s").
- **Mean Speed:** average speed of vehicles currently active in the simulation.
- **TL Status Line:** displays the currently selected traffic light ID, its **Red/Yellow/Green state string**, and the current control mode (e.g., **AUTO**, plus whether **RULE-based TLS is ON/OFF**).

Overall, this panel makes it easy to show the professor, in one screenshot, how route selection, vehicle spawning, and TLS control affect congestion, waiting time, and traffic flow in real time.

3.1 **Main Dashboard Overview (Traffic Grid Simulation GUI)**

Figure 2.4 : **Complete Traffic Grid Simulation Dashboard (Controls + Map View + Metrics Panel)**

This figure shows the **full GUI dashboard** of the Traffic Grid Simulation system. The interface is divided into three main areas:

- **Left Control Panel:** provides simulation controls (**Start/Stop**, **Export CSV/PDF**), a **speed slider (1×–10×)**, **route/scenario selection** (same destination with long route variants), **vehicle injection** settings (number of vehicles + spawn type **Car/Truck/Bus**), **map filters** (vehicle type visibility + minimum speed filter), and **manual traffic light control** (TLS dropdown with **Red/Green/Reset**) including the **Rule-based TLS toggle**.
- **Center Map Panel:** displays the **SUMO road network** and renders vehicles in real time using their live positions. Traffic light nodes are labeled (e.g., **t1–t14**), helping visualize movement, queues, and congestion areas on specific junctions and corridors.
- **Right Metrics Panel:** shows a **live trend chart** (Avg Wait time, Throughput/VPH, Congestion) and a **real-time KPI summary** including active/visible vehicles, type breakdown, average waiting time, congestion index, throughput over the last **300 seconds**, mean speed, and the current traffic light control mode (**AUTO / RULE ON-OFF**).

3.2. **SUMO Reference Simulation View (Baseline Network Execution)**

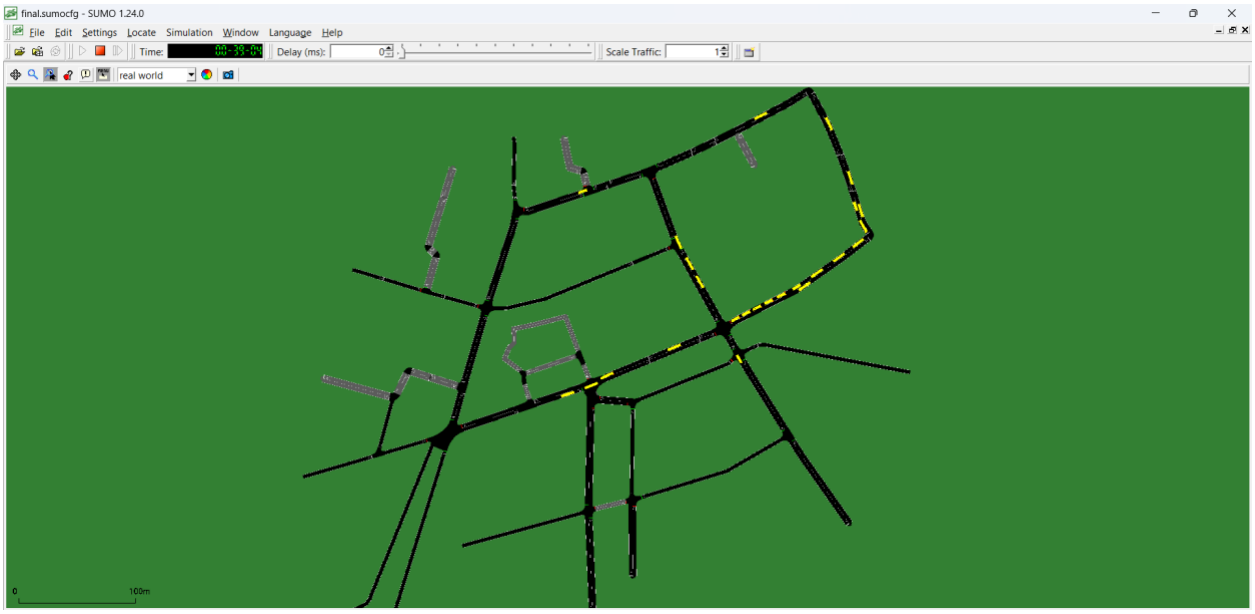


Figure 2.5 : **SUMO-GUI View of `final.sumocfg` Network During Simulation**

This figure shows the traffic network running directly inside **SUMO-GUI** using the configuration file **`final.sumocfg`**. It represents the **baseline simulation environment** from which the Java dashboard receives live data through **TraCI/TraaS**. The black road geometry is the loaded network (`net.xml`), and the highlighted segments indicate active lanes/vehicle movement during runtime. This view is useful for validating that the SUMO scenario (network, routes, and traffic lights) is correct before or alongside using the custom Java GUI for interaction, metrics monitoring, and exporting results.

3.3. Export Metrics to CSV (File Save Dialog)

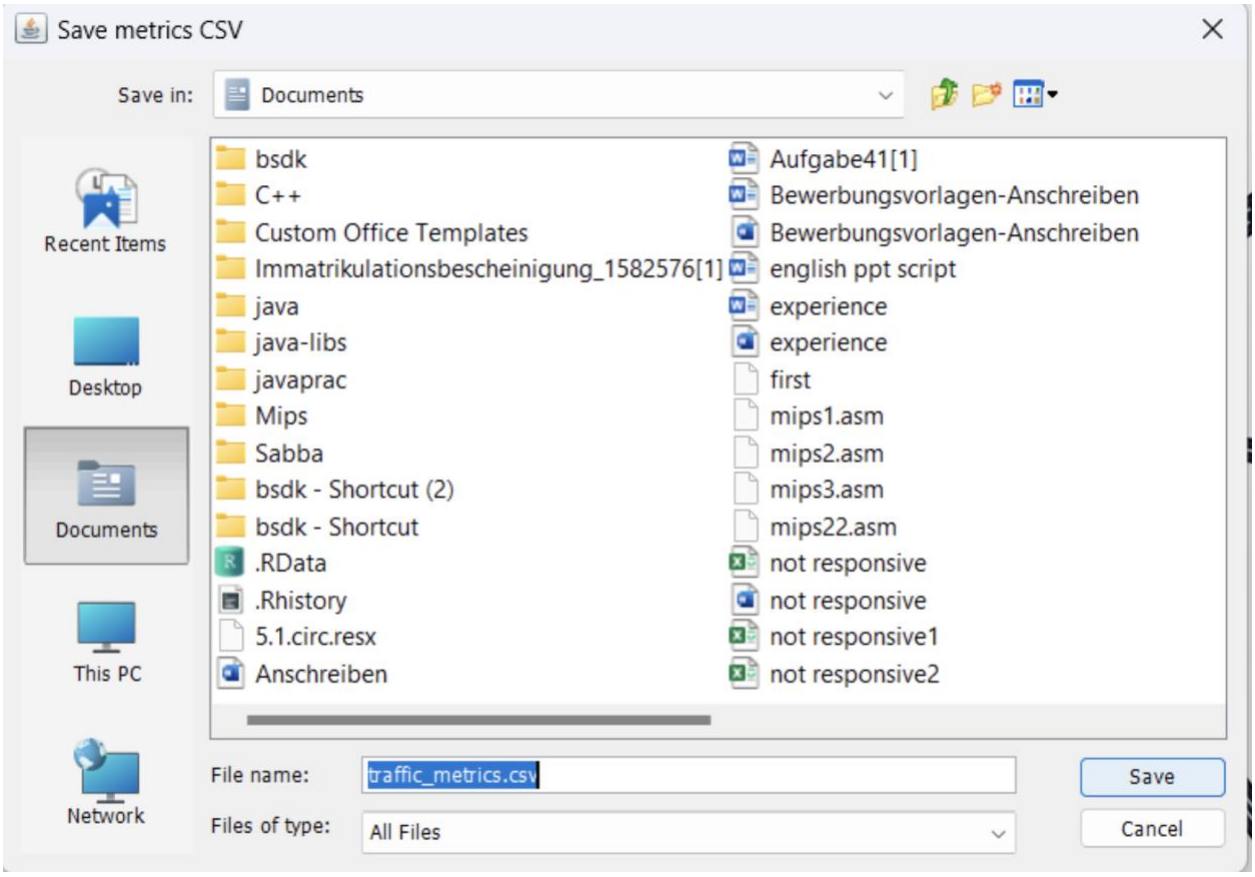
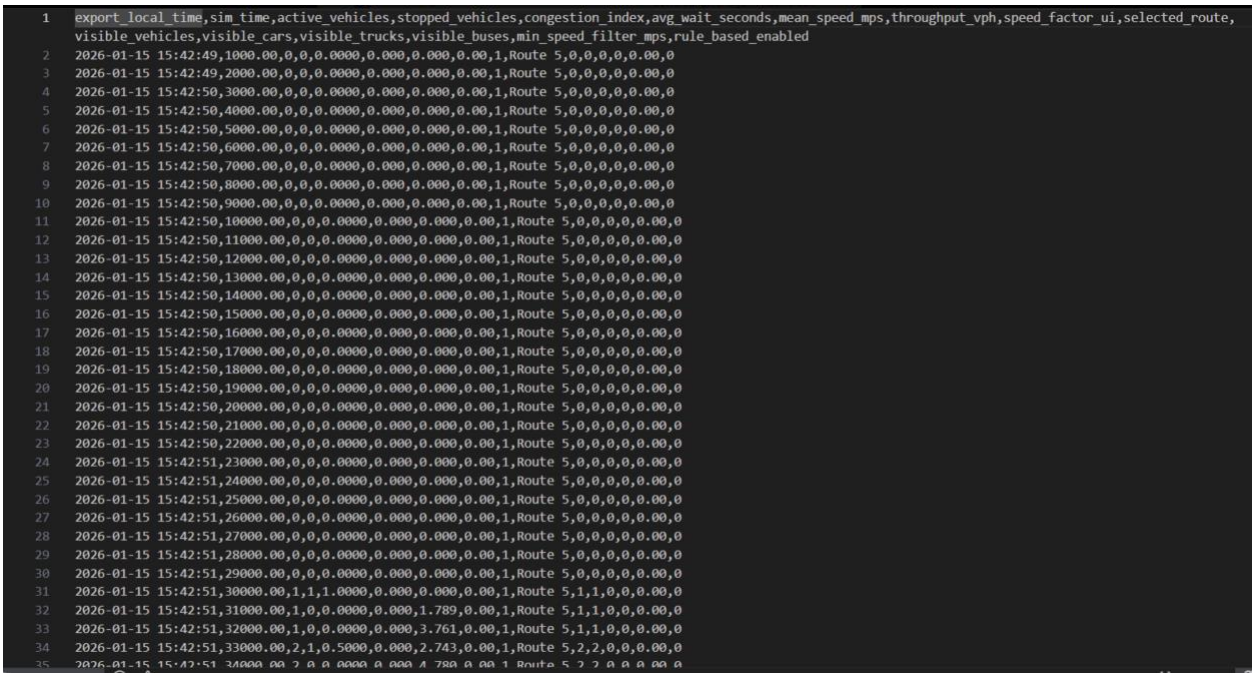


Figure 2.6 **: "Save Metrics CSV" Dialog for Exporting Simulation Results**

This figure shows the **file save dialog** that appears when the user clicks **Export CSV** in the dashboard. The user can choose a folder/location and enter a filename (e.g., `traffic_metrics.csv`). After pressing **Save**, the application exports the collected simulation metrics to a **CSV file**, typically including values such as active/visible vehicles, vehicle-type counts, average waiting time, congestion index, throughput (vehicles/hour), and mean speed recorded over time. This export allows the results to be opened later in Excel or any data analysis tool for reporting and comparison between different scenarios.

3.3.1 CSV Export Output (Recorded Simulation Metrics File)



```
export_local_time,sim_time,active_vehicles,stopped_vehicles,congestion_index,avg_wait_seconds,mean_speed_mps,throughput_vph,speed_factor_ui,selected_route,
visible_vehicles,visible_cars,visible_trucks,visible_buses,min_speed_filter_mps,rule_based_enabled
2026-01-15 15:42:49,1000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:49,2000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,3000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,4000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,5000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,6000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,7000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,8000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,9000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,10000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,11000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,12000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,13000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,14000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,15000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,16000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,17000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,18000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,19000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,20000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,21000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:50,22000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,23000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,24000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,25000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,26000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,27000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,28000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,29000.00,0,0,0.0000,0.000,0.000,0.00,1,Route 5,0,0,0,0,0.00,0
2026-01-15 15:42:51,30000.00,1,1,1.0000,0.000,0.000,0.00,1,Route 5,1,1,0,0,0.00,0
2026-01-15 15:42:51,31000.00,1,0,0.0000,0.000,1.789,0.00,1,Route 5,1,1,0,0,0.00,0
2026-01-15 15:42:51,32000.00,1,0,0.0000,0.000,3.761,0.00,1,Route 5,1,1,0,0,0.00,0
2026-01-15 15:42:51,33000.00,2,1,0.5000,0.000,2.743,0.00,1,Route 5,2,2,0,0,0.00,0
```

Figure 2.7 **Example of Exported `traffic_metrics.csv` Showing Logged Metrics per Simulation Step**

This figure shows the **content of the exported CSV file** generated by the **Export CSV** feature. The first row is the header, defining each metric column recorded during the run, and each following row represents one logged snapshot at a specific time/step. Typical columns include:

- **export_local_time**: real clock time when the row was written
- **sim_time**: SUMO simulation time / step time
- **active_vehicles** and **stopped_vehicles**: current traffic load and how many are nearly stopped
- **congestion_index**: stopped ÷ active (ratio-based congestion measure)
- **avg_wait_seconds**: average waiting time of vehicles
- **mean_speed_mps**: mean vehicle speed (m/s)
- **throughput_vph**: vehicles per hour, computed using a rolling window (last 300 s)
- **speed_factor_ui**: current speed slider value (1×–10×)
- **selected_route**: chosen scenario/route (e.g., Route 5)
- **visible_vehicles + by-type counts (cars/trucks/buses)**: map-filtered view data
- **min_speed_filter_mps**: current minimum speed filter value
- **rule_based_enabled**: whether Rule-based TLS was ON/OFF at that moment

This CSV is designed for **easy analysis in Excel/Python**, so you can compare different route scenarios, vehicle loads, and TLS strategies using real collected data.

4. Technical Description

4.1 Core Business Classes

1. MapVisualisation Class

As the core visualization component of the simulation system, MapVisualisation is responsible for rendering SUMO's abstract traffic elements into an intuitive graphical interface. It parses geometric and status data of roads, lanes, and vehicles, dynamically draws real-time simulation states, supports basic map interactions , and provides visual annotations for key elements like traffic lights. Focused solely on data visualization without direct simulation control logic, it relies on data from LiveConnectionSumo to ensure synchronization between the visual interface and the backend simulation.

2. LiveConnectionSumo Class

Serving as the communication hub between the system and the SUMO backend, LiveConnectionSumo manages real-time connections with SUMO, encapsulates the official libtraci API, and provides a unified interface for data reading and control command delivery. It handles simulation startup/shutdown, parses sumocfg configuration files, retrieves simulation data, and executes control instructions . Shielding version differences of SUMO APIs, it acts as the sole entry for other components to interact with the SUMO backend, ensuring stable and reliable data transmission and simulation control.

3. TrafficControl Class

Specialized in traffic light management and manual control, TrafficControl enables human intervention in simulation traffic lights. It constructs traffic light dropdown menus for the GUI, supports forced red/green light operations, resets manually controlled traffic lights to their default programs, and continuously applies manual control states to ensure persistent effects. With thread-safe data storage, it synchronizes traffic light status with both the SUMO backend and the GUI interface, providing real-time feedback on operation results and ensuring compatibility with SUMO's native traffic light control logic.

4. VehicleInjection Class

Responsible for vehicle injection and route management, VehicleInjection parses route files (rou.xml) to load predefined travel routes, validates route validity for specific vehicle types, and generates temporary configuration files for vehicle types . It supports batch injection of vehicles with specified routes and types, automatically generates unique IDs for vehicles and routes to avoid duplicates, filters invalid routes, and simplifies the GUI route selection dropdown

5. GUI Class

As the user interaction portal of the system, GUI integrates all core functions into a visual interface. It provides simulation control buttons , operation panels for vehicles and traffic lights, and displays real-time simulation metrics and the map visualization component. Supporting thread-safe interface updates to avoid cross-thread exceptions, it offers user-friendly operation feedback and error prompts, enabling users to control the simulation, adjust parameters, and monitor results without direct interaction with backend code.

6. Logging Class

As the global unified logging utility class of the system, Logging initializes the Java built-in logging framework automatically when the class is loaded, configuring dual log output targets . It unifies the log format , supports log file rolling strategy , and provides a global accessible LOG instance for other core classes to record runtime information . It also offers a time formatting utility method to generate standard time tags, and handles logging initialization exceptions gracefully to avoid affecting the overall system startup.

4.2 Core Technical Dependencies

1. SUMO libtraci Library

sumo.libtraci is the official Java interaction library provided by SUMO simulation software, serving as the core dependency for the system to communicate with the SUMO backend, read real-time simulation data (e.g., roads, vehicles, traffic lights), and issue control commands (e.g., forcing traffic light states, injecting vehicles). It encapsulates the underlying communication protocol, offers object-oriented APIs and a dedicated StringVector data structure, shielding the complexity of SUMO interaction and lowering the development threshold for simulation control.

2. java.util

This package provides core collection components such as ArrayList, LinkedHashMap, and ConcurrentHashMap. It is used for internal data caching of core classes, offers thread-safe data access in multi-threaded environments , and provides convenient data operations to support business logic like route filtering and traffic light state management.

3. java.util.logging

This package includes components like Logger and FileHandler, supporting the Logging class to implement global unified logging. It provides hierarchical logging capabilities, supports dual output to the console and local files, and meets the core needs of recording system runtime information and troubleshooting without introducing third-party logging libraries.

4. java.io

This package offers file and stream operation components such as File and PrintWriter. It supports the VehicleInjection class to create and write temporary vehicle configuration files, completes path splicing and existence verification of SUMO configuration files.

5. java.lang.reflect

This package provides components like Method and Class, enabling the VehicleInjection class to be compatible with different versions of the libtraci API. It achieves robust method calls, prevents system crashes caused by missing APIs in specific versions, and enhances system compatibility and robustness.

6. java.time

This package supplies thread-safe date and time components such as LocalDateTime and DateTimeFormatter. It supports the Logging class to generate standard-format time tags, ensures the accuracy and readability of time in logs, and avoids concurrency issues in date processing.

7. javax.swing

This package provides GUI components like JFrame, JComboBox, and SwingUtilities. It supports the GUI class to build a visual user interface, realizes thread-safe interface updates to avoid cross-thread exceptions, and feedbacks the operation results of traffic lights and vehicle injection to users in a user-friendly manner.

5. Utilities

5.1 Milestone3Exception (Custom Exception)

`Milestone3Exception` is a custom exception used during project setup validation—mainly when the SUMO configuration file (`final.sumocfg`) is missing, not readable, or not located in the correct working directory.
Instead of letting the application crash with a generic exception, this custom exception provides a clear, user-friendly error message (shown via a GUI popup and also written to the log). This also helps fulfill the project requirement of having at least one custom exception.

Where used:

- `Main.validateProjectSetup()` checks whether `final.sumocfg` exists and can be read.
- If invalid: throws `Milestone3Exception`, then `Main.main()` logs the error and shows a dialog.

5.2 Logging Utility (Console + File Logger)

The `Logging` class provides a centralized logging setup using `java.util.logging`. It supports:

- Console logging (INFO level)
- File logging to `traffic_sim.log` (ALL level) with log rotation
- A consistent timestamped log format

This is especially useful for debugging SUMO start/stop, route building, vehicle injection, export actions, and runtime errors.

Where used:

- Used across the project (`Main`, `MapVisualisation`, `VehicleInjection`, `TrafficControl`, `LiveConnectionSumo`, `GUI`) via `Logging.LOG`.

5.3 renderComponentToImage (UI Snapshot Helper)

`MapVisualisation.renderComponentToImage(...)` is a helper method that renders any Swing `JComponent` (especially the trend chart panel) into a `BufferedImage`.
This is primarily used for PDF export, where the live chart is embedded as an image in the summary report.

Where used:

- `GUI` → when the user clicks **Export PDF**, the trend chart panel is converted to an image and passed to `LiveConnectionSumo.exportSummaryPdf(...)`.

5.4 CSV Escape Helper

`LiveConnectionSumo.csvEscape(...)` ensures strings are safely written into CSV format. If a string contains commas, quotes, or line breaks, it applies standard CSV escaping rules (doubling quotes and wrapping values in quotes when required).

Where used:

- `LiveConnectionSumo.exportMetricsCsv(...)` while writing values like timestamps and selected route names.

5.5 SimplePdfWriter (Minimal One-Page PDF Generator)

`SimplePdfWriter` is a lightweight internal PDF generator that creates a simple one-page summary PDF without external libraries. It supports:

- A title and multiple text lines (metadata + latest metrics snapshot)
- Optional embedding of a chart image (stored as JPEG)

Where used:

- `LiveConnectionSumo.exportSummaryPdf(...)` calls `SimplePdfWriter.writeOnePageReport(...)`.

5.6 XML Parsing Utilities (SUMO Config, Network, and Trips)

The project includes several XML parsing utilities to load SUMO files and extract configuration details:

- `MapVisualisation.readNetFileFromSumocfg(...)`: reads `<net-file>` from `final.sumocfg`
- `MapVisualisation.loadRoadGeometriesFromNet(...)`: loads lane shapes from `.net.xml` for road rendering
- `MapVisualisation.loadTlsPositionsFromNet(...)`: loads junction positions and maps traffic lights to coordinates

- `VehicleInjection.readRouteFilesFromSumocfg(...)`: reads `<route-files>` from `final.sumocfg`
- `VehicleInjection.parseTrips(...)`: extracts `<trip from="..." to="..." via="...">` from the route file(s)

These utilities enable map bounds initialization, road drawing, TLS marker placement, and scenario dropdown population.

5.7 Reflection Fallback Helpers (LibTraCI API Compatibility)

Some LibTraCI methods can differ slightly across versions (for example different method signatures for `Simulation.findRoute`, or different waiting-time APIs). To make the project more robust, reflection-based helper methods are used:

- `VehicleInjection.tryInvokeRet / tryInvokeDouble / tryInvokeInt`
- `LiveConnectionSumo.safeVehicleWaitingTimeSec(...)`

This allows the program to try multiple available API signatures and continue working even when the exact method signature differs.

6. Simulation Lifecycle & Thread Interaction (Swing EDT + LiveConnectionSumo)

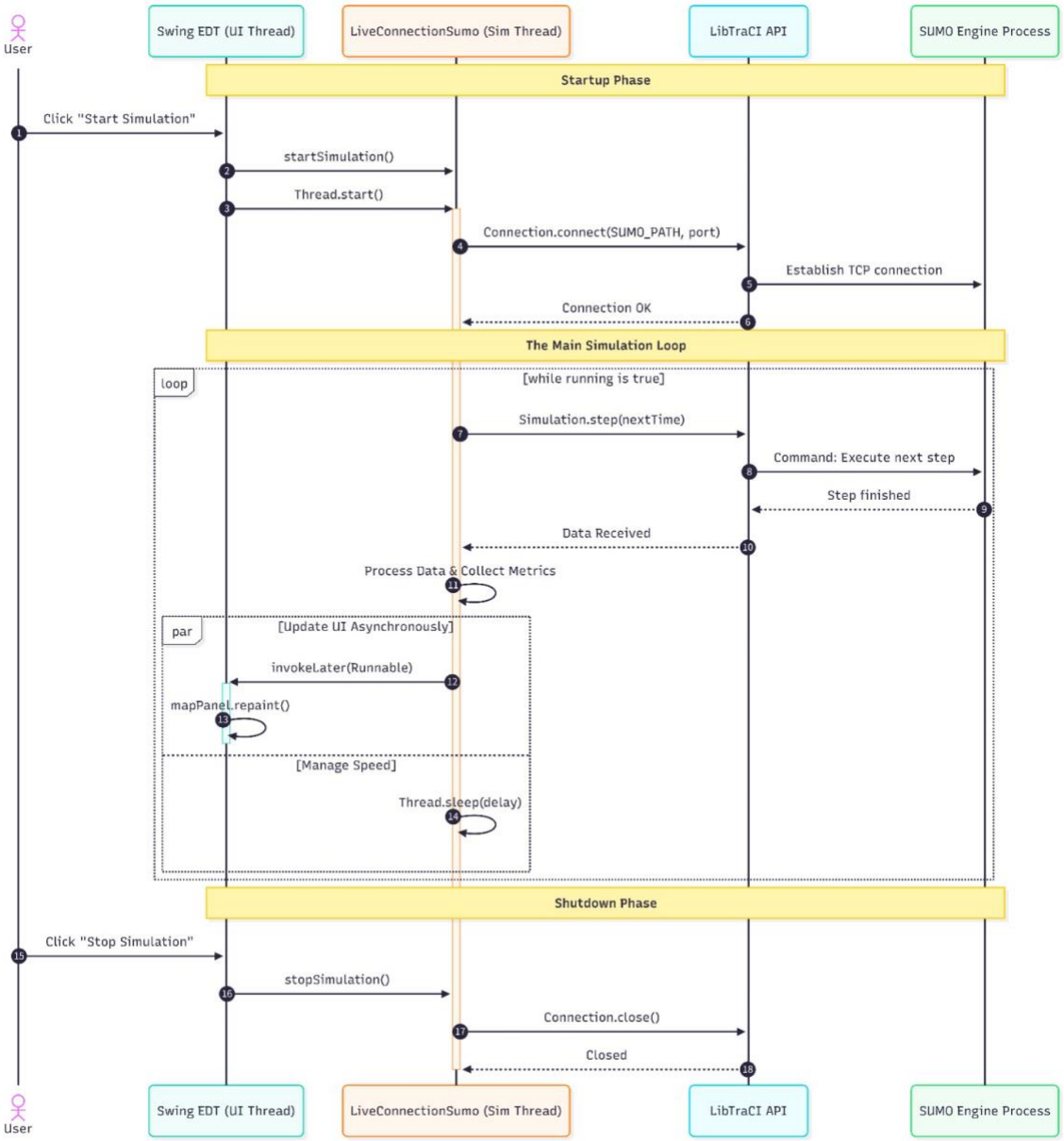


Figure 2.8: **Sequence Diagram – SUMO/TraCI Simulation Loop with Asynchronous UI Updates**

This sequence diagram illustrates how the application starts, runs, and stops the SUMO simulation while keeping the GUI responsive. After the user clicks **Start Simulation**, the Swing **EDT (UI thread)** triggers `startSimulation()`, while the actual simulation runs in a dedicated background thread (**LiveConnectionSumo**). The simulation thread establishes a TCP connection via the **LibTraCI API** to the **SUMO engine process**.

During the main loop (`while running == true`), the simulation thread repeatedly calls `Simulation.step(...)` to advance SUMO by one time step, receives updated traffic data, and computes live metrics (e.g., vehicle counts, waiting time, congestion index, throughput). To avoid blocking or freezing the UI, visual updates (e.g., map repaint, labels, charts) are dispatched asynchronously to the EDT using `SwingUtilities.invokeLater(...)`. The simulation speed is controlled by sleeping for a configurable delay (`Thread.sleep(delay)`).
When the user clicks **Stop Simulation**, the UI triggers `stopSimulation()`, the loop terminates, and the connection is cleanly closed (`Connection.close()`), ensuring a controlled shutdown of both simulation and UI updates.

6.1. Logging and Error Handling

To keep the system stable and easy to debug, the application includes a centralized logging utility (`Logging.java`). This class configures a consistent log format and writes messages both to the console and to a rotating log file (`traffic_sim.log`). Important lifecycle events (startup, SUMO connection, simulation start/stop), warnings (missing/invalid route files, fallback bounds, limited route variety), and runtime failures (TraCI calls, export errors) are recorded through `Logging.LOG`. These logs help developers quickly trace issues and verify that the simulation workflow is running as expected.

In addition, the project includes a custom exception (`Milestone3Exception`) inside `Main.java` to handle critical setup errors early. For example, if the SUMO configuration file (`final.sumocfg`) is missing or not readable, the program throws `Milestone3Exception`, logs the error, and shows a clear GUI error dialog instead of crashing with a generic exception. This improves reliability and ensures the user receives immediate feedback about configuration problems.

For runtime robustness, many TraCI API calls are protected with try/catch blocks and safe fallback logic (including reflection-based method attempts in some cases). As a result, the simulation can continue running even if certain optional API methods are unavailable or a single TraCI call fails, while still reporting the issue in the logs for debugging.

7. Milestones

7.1 Milestone 1 — System Design & Prototype (27.11.2025)

Story

In the first team meetings, we aligned on the overall idea: a **real-time traffic simulation** that connects to **SUMO via LibTraCI (TraaS-style API)** and provides a **Java Swing GUI** for interaction (map view, controls, and basic metrics). We split responsibilities into two main tracks:

- **Backend / SUMO integration:** establishing a stable connection, stepping the simulation, reading traffic light IDs, and exploring how vehicle injection and traffic light control should work.
- **Frontend / UI foundation:** creating an initial Swing frame with panels, buttons, and placeholders so we could test backend features continuously.

Since this was our first time building a larger Swing application, we started with a small prototype UI (simple frame + minimal panels) and iterated on it while the backend team tested TraCI calls. Once the SUMO connection was stable, we used the prototype to demonstrate basic functionality such as **starting SUMO**, **stepping the simulation**, and **listing traffic lights**. In parallel, we prepared the first technical documentation artifacts (architecture sketch, component responsibilities, and early class design ideas).

Things achieved in Milestone 1

1. **Project overview + initial architecture planning** (how UI thread and simulation thread interact).
2. **First Swing prototype UI** (basic frame, buttons, panels) to support backend testing.
3. **SUMO connection demo**: connection established, simulation stepping verified, traffic light IDs accessible.
4. **Early design draft for core components** (map view module, simulation loop module, traffic light control idea, vehicle injection idea).
5. **Project setup in Git** (basic repository structure, initial commits, and division of tasks).

7.2 Milestone 2 — Functional Prototype (14.12.2025)

Progress since Milestone 1

After Milestone 1, we shifted from “prototype + experiments” to a **working application** with core features implemented end-to-end. The GUI was expanded into a dashboard-like layout, while the backend logic was structured into separate classes that handle different responsibilities

(logging, map visualization, vehicle injection, traffic light control, and the live simulation thread).

On the backend side, the main focus was to make the system interactive: **inject vehicles**, **control traffic lights**, and **continuously collect metrics** during the simulation. On the frontend side, we improved usability by adding a control sidebar (simulation start/stop, speed control, scenario selection, injection buttons, filters) and a metrics panel including a **live trend chart**.

Things achieved in Milestone 2

1) Working application (core features)

- **Live SUMO connection + simulation loop** running in a dedicated thread (`LiveConnectionSumo`).
- **Vehicle injection** (Car/Truck/Bus) via selectable scenarios (`VehicleInjection`).
- **Traffic light control**
  - manual forcing (Red/Green/Reset)
  - optional rule-based behavior toggle (`TrafficControl`).
- **Map visualization** including **zoom / pan / rotation**, vehicle rendering, and TLS markers (`MapVisualisation.MapPanel`).
- **Vehicle filtering** (by type + minimum speed) applied consistently on the map and visible counts (`GUI.VehicleFilter`).

2) Logging & stability

- Central logging with **console + rotating log file** (`Logging.java`).
- Proper startup validation with a custom exception (`Milestone3Exception` in `Main.java`) to catch missing/unreadable config early.
- Try/catch safety around TraCI calls so the simulation stays resilient even if a call fails.

3) Reporting / export

- **CSV export** of time-series metrics (`exportMetricsCsv`).
- **PDF summary export** (one-page report with key values + embedded chart image) (`exportSummaryPdf` + `SimplePdfWriter`).

4) Documentation + engineering practice

- Added meaningful inline documentation and clear separation of responsibilities across classes.
- Improved commit structure and continued work distribution across UI/backend tasks.

8. Challenges

Developing a real-time traffic simulation with SUMO (LibTraCI) and a Java Swing GUI involved multiple technical challenges. The main difficulties were related to real-time UI updates, multithreading safety, performance of continuous TraCI calls, route generation requirements, and reliable export/reporting features.

8.1 Real-Time UI Refresh without Freezing Swing (EDT Issue)

A major challenge was ensuring that the GUI remains responsive while continuously updating the map and metrics during simulation runtime. Java Swing uses a single **Event Dispatch Thread (EDT)** for UI updates. If expensive operations (TraCI calls, metric computation, file access) are executed on the EDT, the application becomes slow or unresponsive.

Solution implemented:
We moved all simulation stepping and data collection into a dedicated background thread (`LiveConnectionSumo implements Runnable`). UI updates (labels and map refresh) are executed safely using:

- `SwingUtilities.invokeLater(...)`

This ensures that the EDT is only used for painting and lightweight UI updates, while all heavy simulation work runs in the background.

8.2 Thread Safety and Consistency of Live Data (Race Condition Risk)

Since simulation data changes every step (vehicle IDs, speeds, positions, waiting time), reading and updating shared structures incorrectly can cause inconsistent UI behavior (e.g., random values in counts, partial updates on the map).

Solution implemented:
We avoided unstable shared state by generating a fresh data snapshot each cycle:

- building `positions`, `types`, and `speeds` maps during the simulation loop
- passing these snapshots into `MapPanel.updateVehicles(...)`

Inside the map panel, thread-safe collections (`ConcurrentHashMap`) are used. UI updates are performed only on the EDT through `invokeLater`, which improves consistency and avoids race-condition-like behavior.

8.3 Performance Constraints from Frequent TraCI Calls

Collecting live metrics can become expensive when vehicle count increases. The system repeatedly queries:

- vehicle positions (`Vehicle.getPosition`)
- speeds (`Vehicle.getSpeed`)
- waiting time (`Vehicle.getWaitingTime` / `getAccumulatedWaitingTime`)
- traffic light states, controlled lanes, halting vehicles (Lane APIs via reflection)

If metrics were logged every step, this would increase overhead significantly.

Solution implemented:
We introduced controlled logging and sampling:

- metrics are appended only every **0.5 simulation seconds** (`LOG_EVERY_SIM_SECONDS = 0.5`)
- trend chart updates follow the same schedule
- throughput is computed using a **time-window queue** (`arrivalTimes` deque) to avoid costly recomputation

This reduces overhead while still preserving meaningful live data.

8.4 Long Route Generation and Branching Requirement

The project required vehicles to follow **longer routes** (not the shortest path) and to split at a junction following a fixed branching rule:

- Variant A: 30%
- Variant B: 40%
- Variant C: 20%
- Variant D: 10%

Additionally, routes should share a common prefix before splitting and should still end at the same destination.

Solution implemented:
In `VehicleInjection`, routes are generated dynamically using:

- a pool of candidate "via edges"
- repeated `Simulation.findRoute(...)` calls to build multi-segment paths
- filtering rules such as minimum edge length, disallowing repeated edges (`DISALLOW_EDGE_REPEATS`)
- scoring routes (based on length + number of edges) and selecting the best variants

- installing routes via `Route.add(...)`
- probabilistic selection using the branching distribution (`pickVariantIndex()`)

This approach produces long, diverse routes while preserving the defined branching behavior.

8.5 Stability, Debugging, and Defensive Error Handling

The system depends on external configuration files and a TraCI-based runtime environment. Failures can occur due to missing files, incompatible TraCI methods, or route generation edge cases.

Solution implemented:

- Centralized logging (`Logging.java`) outputs both:
  - console logs
  - rotating file logs (`traffic_sim.log`)
- startup validation in `Main.java` using a custom exception:
  - `Milestone3Exception` for missing or unreadable `final.sumocfg`
- defensive try/catch blocks around optional TraCI calls
- reflection-based fallbacks for API compatibility differences

This improves stability and makes debugging easier during development and grading demonstrations.

8.6 Export Features (CSV + PDF Summary)

Generating exports reliably was also challenging because the system needed to export:

- structured time-series metrics (CSV)
- a one-page PDF summary including key values and a chart screenshot

Solution implemented:

- CSV export uses safe escaping (`csvEscape`) and exports metrics collected in `metricsLog`
- PDF export generates:
  - a rendered image of the live trend chart (`renderComponentToImage`)
  - a compact one-page PDF summary (`SimplePdfWriter`) including the chart image and key simulation statistics

This produces grading-friendly outputs without requiring additional external dependencies.

9. Conclusion

The Traffic Grid Simulation project successfully integrates a Java Swing-based GUI with the SUMO simulation engine using the LibTraCI interface to create an interactive and measurable traffic simulation environment. The system combines real-time simulation control (start/stop, speed adjustment, vehicle injection), live visualization (map rendering with zoom/pan/rotation), and operational traffic management features such as manual and rule-based traffic light control.

A major strength of the project is its real-time data pipeline: the simulation runs on a dedicated background thread (`LiveConnectionSumo`), while the GUI stays responsive through safe EDT updates (`SwingUtilities.invokeLater`). During runtime, the application continuously computes and visualizes key performance indicators—active vehicles, congestion index, average waiting time, mean speed, throughput (vehicles/hour), and filtered visibility—providing meaningful insight into traffic behavior.

In addition, the project supports practical reporting and evaluation through export features. Collected metrics can be exported as CSV for further analysis, and a compact PDF summary report can be generated including the latest simulation statistics and a snapshot of the live trend chart. Centralized logging and startup validation improve reliability and make debugging and grading demonstrations easier.

Overall, this project demonstrates solid implementation of core software engineering concepts such as modular design, GUI programming, multithreading, real-time data handling, and structured reporting. It provides a strong foundation for future improvements such as richer OOP wrappers for TraCI calls, enhanced route planning strategies, more advanced traffic light policies, and deeper analytics dashboards.

10. Sources

- **Java Swing & AWT (GUI framework)** – frames, panels, layouts, painting, event handling.
  https://docs.oracle.com/javase/tutorial/uiswing/
  https://docs.oracle.com/en/java/javase/
- **Swing threading model (EDT) + concurrency utilities** – background thread + `SwingUtilities.invokeLater(...)`.
  https://docs.oracle.com/javase/tutorial/uiswing/concurrency/
- **Java Logging (java.util.logging)** – custom logger + file handler.
  https://docs.oracle.com/en/java/javase/ (search: `java.util.logging`)
- **Java XML parsing (DOM)** – reading `.sumocfg`, `.net.xml`, `.rou.xml` via `DocumentBuilderFactory`.
  https://docs.oracle.com/en/java/javase/ (search: `javax.xml.parsers` and `org.w3c.dom`)

Data Export / Reporting

- **CSV format basics** – escaping commas/quotes/newlines for safe exports.
  (General reference) https://www.rfc-editor.org/rfc/rfc4180
- **PDF basics (single-page export)** – the project uses a lightweight custom PDF writer approach (image embedding + text drawing).
  (General reference) https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/

Learning Resources / Inspiration

- **University course materials (Lecture PDFs/videos)** – used for general OOP structure, clean code practices, and design decisions in Java.
- **YouTube – Bro Code (Java Swing GUI basics)** – layouts, UI components, event handling.
  https://www.youtube.com/@BroCodez
- **YouTube – Engineering Digest (Threads, synchronization, Java concepts)** – background threading patterns, common pitfalls, debugging mindset.
  https://www.youtube.com/@EngineeringDigest
- **ChatGPT** – used as a debugging assistant and for idea exploration (e.g., UI structuring, export formatting, and error-handling patterns).
  https://chatgpt.com/

11. **Code File and Responsibility**

Code File	**Primary Authors**	**Role/Responsibility**
Main.java	Ranjit Jaiswal	Application entry point. Validates project setup (SUMO config presence), initializes route/map data (`VehicleInjection.loadTripRoutesFromRou()`, `MapVisualisation.initBoundsFromFiles()`), and launches the GUI (`GUI.launch()`). Contains custom project exception (`Milestone3Exception`).
Logging.java	Akshat Kumar Sharma	Central logging utility. Configures console + file logging (`traffic_sim.log`) using `java.util.logging`. Provides timestamp helper (`nowTag()`), ensures consistent formatted logs for debugging and grading evidence.
GUI.java	Ranjit Jaiswal	Main Swing GUI assembly. Builds the dashboard layout: map center, left control panel (simulation controls, vehicle injection, filters, TLS tools), right metrics + live trend chart. Handles all UI actions (buttons, sliders, checkbox toggles) and connects UI state to backend components.
LiveConnectionSumo.java	Akshat Kumar Sharma	Core simulation runtime controller (background thread). Starts SUMO via LibTraCI, steps simulation, computes live metrics (active vehicles, congestion index, average wait, mean speed, throughput), updates GUI safely via EDT, collects metrics log for export, and triggers CSV/PDF export.
VehicleInjection.java	Ranjit Jaiswal	Trip/scenario handling + vehicle spawn logic. Reads route/trip definitions from `.sumocfg` and `.rou.xml`. Builds **long route variants** (A/B/C/D split 30/40/20/10) using TraCI route finding, prevents loops/repeated edges, installs routes in SUMO, and injects vehicles robustly for car/truck/bus types.
MapVisualisation.java	Akshat Kumar Sharma	Map model + rendering. Parses network bounds and lane geometry from `final.net.xml`, loads TLS marker positions/labels, and renders roads + vehicles. Provides interactive `MapPanel` with **zoom/pan/rotation** + filtering support. Includes `TrendChartPanel` for live metric visualization and `renderComponentToImage()` helper for PDF snapshot images.

TrafficControl.java	Ranjit Jaiswal	Traffic light management layer. Builds TLS dropdown, supports **manual persistent forcing** (RED/GREEN) with reset to original program. Implements optional **rule-based TLS mode** (stop/go cycle) using demand from controlled lanes. Provides readable TLS status string for UI.
SimplePdfWriter *(inside LiveConnectionSumo.java)*	Ranjit Jaiswal	Minimal one-page PDF generator. Writes summary PDF with title, metrics lines, and embedded chart image as JPEG (manual PDF object writing, no external libs). Used by `exportSummaryPdf()`.
MapPanel *(inside MapVisualisation.java)*	Akshat Kumar Sharma	`MapPanel`: real-time rendering of map + vehicles with interaction and filtering. `TrendChartPanel`: live chart of AvgWait / Throughput / Congestion using in-panel painting, updated every sample.

12. Requirements Checklist (Traffic Simulation Project)

Runnable Program:

- A runnable Java program is provided and can be executed from the project folder (e.g., via IDE run configuration or runnable JAR, depending on submission setup). The application launches the Swing GUI and starts the SUMO simulation using `final.sumocfg`.

Milestones:

- Milestone 1 and Milestone 2 are included in the documentation (project planning, prototype progress, and functional implementation progress).

Object Orientation (OOP):

- The project is structured into clearly separated classes with single responsibilities: `Main`, `GUI`, `LiveConnectionSumo`, `MapVisualisation`, `VehicleInjection`, `TrafficControl`, and `Logging`.
- A custom exception (`Milestone3Exception`) is included as part of structured error handling and clean project setup validation.

Collections:

- The system uses Java collections for runtime data handling, for example:
  - `List` / `Deque` for metric logs and throughput window tracking
  - `Map` and concurrent maps for vehicle positions/types/speeds and safe UI updates
  - dropdown models for scenarios and traffic light selections

Error Handling:

- A custom exception `Milestone3Exception` is used to handle critical setup errors (e.g., missing or unreadable `final.sumocfg`) with clear user feedback (GUI dialog + logging).
- Runtime TraCI calls are protected using defensive `try/catch` blocks to prevent unexpected crashes during simulation.

Streams and Files:

- File-based logging is implemented via `Logging.java` (writes to `traffic_sim.log` with rotation).
- Export functionality is implemented:
  - **CSV export** for time-series metrics
  - **PDF export** for a one-page summary report with an embedded chart image

Threads:

- The simulation runs on a dedicated background thread (`LiveConnectionSumo`), while the GUI runs on the Swing **Event Dispatch Thread (EDT)**.
- UI updates are dispatched using `SwingUtilities.invokeLater(...)` to keep the interface responsive and thread-safe.

Clean Code:

- Code is organized into modules/classes based on clear responsibilities (logging, simulation loop, map rendering, traffic light control, route injection, UI).
- Logging and error messages are designed to support debugging and stable behavior during demonstrations.

Documentation:

- Complete documentation is included in the project folder, including architecture explanation, milestones, features, and diagrams.

Extra Features / Additional Information (Dashboard / Live Metrics)

The application provides multiple live metrics beyond simple visualization, including:

- **Active vehicles (total)**
- **Visible vehicles (filtered by type and minimum speed)**
- **Average waiting time**
- **Congestion index (ratio of stopped vehicles)**
- **Throughput (vehicles/hour over a rolling time window)**
- **Mean speed**
- **Traffic light status display** (manual mode + rule-based on/off)

A **live trend chart** visualizes selected metrics over time.

Data Refresh / Real-Time Updates

- The simulation data refreshes continuously during the run loop (`Simulation.step()` in `LiveConnectionSumo`).
- Metrics and chart samples are logged at a fixed interval (e.g., every `0.5` simulation seconds) to balance performance and readability.
- Map vehicle positions and labels update in real time via safe EDT updates.