

UNIVERSITÀ
DI PAVIA

UNIVERSITÀ DEGLI STUDI DI PAVIA

DIPARTIMENTO DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INDUSTRIAL AUTOMATION ENGINEERING

OPTIMIZATION ALGORITHM FOR EV STOP MANAGEMENT IN V2G SERVICES

RELATRICE

Prof.ssa Antonella Ferrara

CORRELATORE

Nikolas Sacchi, PhD

CANDIDATO

Gianluca Broglia

Matricola n. 557998

To all the people I love, who walk beside me every day of my life,
and to those who are no longer with me, but never truly gone.

Abstract

This thesis presents work carried out within the framework of the E-COSMOS project (Electric Charging Optimization for Sustainable MObility Systems). Building on the project's objectives, it develops an integrated software platform for the simulation, scheduling, and real-time control of an electric vehicle (EV) operating in an urban environment. The platform simultaneously addresses two interdependent problems: the optimization of the daily task visit order under flexible time-window constraints, and the management of the vehicle battery to exploit Vehicle-to-Grid (V2G) energy discharge while preserving the charge needed for driving.

The simulation backbone is provided by SUMO (Simulation of Urban MObility), a microscopic open-source traffic simulator, controlled step-by-step through the TraCI Python API. The road network is derived from OpenStreetMap data for a portion of the city of Pavia, Italy, and comprises 2,051 nodes and 4,136 edges. Background traffic is generated stochastically, and the EGO vehicle must navigate through it.

The scheduling module performs an exhaustive permutation search for instances of up to eight tasks; for larger instances, it relies on a greedy earliest-deadline-first (EDF) heuristic refined by 2-opt local search. A battery partitioning rule reserves, on top of the absolute minimum state of charge, the estimated energy required for driving, increased by a 30% safety margin, and allocates the surplus to V2G discharge. The routing module combines two mechanisms: periodic traffic-aware rerouting, executed every 5 seconds, and event-driven congestion response, which monitors the road ahead every 2.5 seconds and triggers rerouting whenever congestion is detected. Both mechanisms rely on an energy-aware edge cost function whose weight increases with the battery reserve required for driving.

A key technical contribution is the identification and resolution of the same-station consecutive stop problem, a scenario in which two successive tasks share the same charging station. The standard TraCI stop API fails in this configuration due to a geometric constraint: the vehicle's lane position at the end of the first stop coincides with the station's downstream boundary, making the second stop command invalid. The proposed solution combines all same-station consecutive tasks into a single SUMO stop with aggregated duration and processes subsequent tasks analytically upon stop completion, while the route is extended during the active stop to ensure the correct onward journey.

The platform was evaluated on five simulation runs covering same-station consecutive stops, rigid-priority tasks, a battery-stress scenario and injected congestion under sparse and dense background traffic. In all runs the vehicle completed its tasks, the combined stop worked as intended and the battery partition limited V2G discharge as designed. The measured mean cost of a rerouting call (about 80–115 ms across the runs) remains well below SUMO's 0.5-second step, as required for real-time operation, and is dominated by the update of the edge travel times of the whole network, while an event-driven check costs about 1 ms. The tests also revealed the main limitations of the approach: Euclidean distance estimates underestimate the energy needed for driving by a factor of about two, and congestion can only be detected where other vehicles provide speed observations.

Sommario

La presente tesi illustra il lavoro svolto nell'ambito del progetto E-COSMOS (Electric Charging Optimization for Sustainable MObility Systems). Partendo dagli obiettivi del progetto, sviluppa una piattaforma software integrata per la simulazione, la pianificazione e il controllo in tempo reale di un veicolo elettrico (EV) che opera in un contesto urbano. La piattaforma affronta contemporaneamente due problemi interdipendenti: l'ottimizzazione dell'ordine giornaliero di visita delle attività sotto vincoli di finestre temporali flessibili, e la gestione della batteria del veicolo per sfruttare la scarica di energia Vehicle-to-Grid (V2G), garantendo al contempo una carica sufficiente per la guida.

La struttura portante della simulazione è fornita da SUMO (Simulation of Urban MObility), un simulatore di traffico microscopico open-source, controllato passo dopo passo tramite l'API Python di TraCI. La rete stradale è ricavata dai dati di OpenStreetMap relativi a una porzione della città di Pavia e comprende 2.051 nodi e 4.136 archi. Il traffico di fondo viene generato in modo stocastico, e il veicolo EGO deve muoversi al suo interno.

Il modulo di pianificazione esegue una ricerca per permutazione esaustiva per istanze con un massimo di otto attività; per istanze più grandi, si affida a un'euristica greedy "earliest-deadline-first" (EDF) perfezionata dalla ricerca locale 2-opt. Una regola di partizionamento della batteria riserva, oltre allo stato di carica minimo assoluto, l'energia stimata necessaria per la guida, aumentata di un margine di sicurezza del 30%, e assegna l'eccedenza alla scarica V2G. Il modulo di routing combina due meccanismi: il ricalcolo periodico del percorso in base al traffico, eseguito ogni 5 secondi, e la risposta alla congestione guidata dagli eventi, che controlla la strada davanti al veicolo ogni 2,5 secondi e attiva il ricalcolo del percorso ogni volta che viene rilevata una congestione. Entrambi i meccanismi si basano su una funzione di costo degli archi sensibile al consumo energetico, il cui peso cresce con la riserva di batteria necessaria per la guida.

Un contributo tecnico fondamentale è l'identificazione e la risoluzione del problema delle fermate consecutive nella stessa stazione, uno scenario in cui due attività successive condividono la stessa stazione di ricarica. L'API standard TraCI per le fermate non funziona in questa configurazione a causa di un vincolo geometrico: la posizione del veicolo nella corsia al termine della prima fermata coincide con il confine a valle della stazione, rendendo invalido il comando per la seconda fermata. La soluzione proposta raggruppa tutte le attività consecutive presso la stessa stazione in un'unica fermata SUMO con durata aggregata ed elabora analiticamente le attività successive al completamento della fermata, mentre il percorso viene esteso durante la fermata attiva per garantire il corretto proseguimento del viaggio.

La piattaforma è stata valutata su cinque simulazioni, che coprono le fermate consecutive nella stessa stazione, le attività a priorità rigida, uno scenario di stress della batteria e una congestione indotta con traffico di fondo rado e denso. In tutte le simulazioni il veicolo ha completato le proprie attività, la fermata combinata ha funzionato come previsto e la partizione della batteria ha limitato la scarica V2G secondo il progetto. Il costo medio misurato di un ricalcolo del percorso (circa 80–115 ms nelle diverse simulazioni) resta ben al di sotto del passo di simulazione di 0,5 secondi di SUMO, come richiesto per un funzionamento in tempo reale ed è dominato dall'aggiornamento dei tempi di percorrenza di tutti gli archi della rete, mentre un controllo guidato dagli eventi costa circa 1 ms. I test hanno inoltre evidenziato i principali limiti dell'approccio: le stime basate sulla distanza euclidea stimano circa la metà l'energia necessaria

per la guida, e la congestione può essere rilevata solo dove altri veicoli forniscono osservazioni di velocità.

Declaration on the Use of Artificial Intelligence Tools

In accordance with the University of Pavia Guidelines on the use of Artificial Intelligence (Resolution no. 153/2026), the author declares that the following artificial intelligence tools were used during the preparation of this thesis, within the scope and in the manner described below.

Tools

- Claude (Anthropic), a generative AI assistant.
- DeepL, a machine translation tool.

Scope of Use

The tools were used as support in the following activities:

- linguistic revision and editing of the text, including the rewriting of some passages on the basis of the author's notes, source code and simulation results;
- conversion of the document from Microsoft Word to LaTeX;
- support in reviewing and debugging the source code;
- processing of the simulation output files, including the preparation of some figures.

Manner of Use

All the content produced or suggested with the help of these tools was critically reviewed, verified and, where necessary, modified by the author. The design of the platform, the definition of the test scenarios, the execution of the simulations, the interpretation of the results and the conclusions are the author's own. The author takes full responsibility for the content of this thesis.

Contents

Declaration on the Use of AI Tools **v**

List of Acronyms **xii**

1 Introduction **1**
1.1 Context and Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Objectives of the Thesis . . . 2
1.4 Document Structure . . . 3

2 State of the Art **4**
2.1 Vehicle-to-Grid Technology . . . 4
2.1.1 Origins and Foundational Work . . . 4
2.1.2 Hardware Standards and Communication Protocols . . . 4
2.1.3 Italian Regulatory Framework . . . 4
2.2 Electric Vehicle Scheduling . . . 5
2.2.1 Problem Classification . . . 5
2.2.2 Heuristic Approaches . . . 5
2.2.3 Energy-Constrained Scheduling . . . 5
2.3 Microscopic Traffic Simulation with SUMO . . . 6
2.3.1 SUMO Architecture . . . 6
2.3.2 The TraCI Interface . . . 6
2.3.3 Charging Station Modeling in SUMO . . . 7
2.4 Related Work . . . 7
2.4.1 EV and Fleet Simulation . . . 7
2.4.2 Task Scheduling with Traffic-Aware Routing . . . 7
2.4.3 The Same-Station Stop Problem . . . 7

3 Simulation Tool **8**
3.1 Design Principles . . . 8
3.2 Module Structure . . . 8
3.3 Pipeline Execution Flow . . . 9
3.4 Working Directory Structure . . . 10
3.5 Data Model . . . 10
3.6 The Task Dataclass . . . 10
3.6.1 Core Fields . . . 10
3.6.2 Flexibility Windows and Priority . . . 12
3.6.3 The stop_duration_s Property . . . 13
3.7 The ChargingStation Dataclass . . . 13
3.8 Shared State (state.py) . . . 14
3.9 Graphical User Interfaces . . . 14
3.10 Task Placement GUI . . . 15

3.10.1 Nearest-Edge Mapping . . . 15
3.10.2 Task Dialog . . . 15
3.11 Charging Station GUI . . . 16
3.11.1 Junction-Based Placement . . . 16
3.11.2 Validation Rules . . . 16
3.12 Calendar CSV and Task Persistence . . . 17
3.12.1 CSV Format and Validation . . . 17
3.12.2 Click-to-Edit and Selective Persistence . . . 18
3.13 Simulation Engine . . . 18
3.13.1 Overview . . . 18
3.13.2 SUMO Configuration Generation . . . 18
3.13.3 Background Traffic and Congestion Injection . . . 19
3.13.4 EGO Vehicle Injection . . . 19
3.14 Main Control Loop . . . 20
3.15 Stop Management: _add_task_stop() . . . 22
3.16 PID Speed Controller . . . 23
3.16.1 Road Type Classification . . . 23
3.16.2 PID Equations . . . 24
3.16.3 Following-Distance Override . . . 24
3.17 Floating Car Data Collection . . . 25

4 Battery and V2G Model **26**
4.1 Overview and Modeling Philosophy . . . 26
4.2 Battery Parameters . . . 26
4.3 Driving Energy Consumption . . . 27
4.4 Battery Partition Model . . . 28
4.4.1 Motivation . . . 28
4.4.2 Driving Energy Estimation . . . 28
4.4.3 Dynamic Minimum SoC . . . 28
4.4.4 Worked Numerical Example . . . 29
4.5 V2G Energy Model . . . 30
4.5.1 Energy Calculation . . . 30
4.5.2 Worked Numerical Example . . . 31
4.5.3 Relationship Between Scheduler Estimate and Simulation Result . . . 31

5 Task Optimization **33**
5.1 Station Assignment . . . 33
5.1.1 Greedy Nearest Assignment . . . 33
5.1.2 Walking Time Estimation . . . 33
5.2 Schedule Evaluation: _evaluate_order() . . . 34
5.2.1 Travel Time Estimation . . . 34
5.2.2 Task Start Time Computation . . . 34
5.2.3 Same-Station Consecutive Detection in the Evaluator . . . 35
5.2.4 V2G Energy Calculation per Stop . . . 35
5.3 Optimization Strategies . . . 35
5.3.1 Brute-Force Search for $N \leq 8$ . . . 35
5.3.2 Greedy Initialization for $N > 8$ . . . 36
5.3.3 2-opt Local Search . . . 36
5.4 Output: ScheduleResult and ScheduleEntry . . . 37
5.5 The Combined Stop: Problem and Solution . . . 38
5.5.1 Problem Statement . . . 38
5.5.2 Root Cause Analysis . . . 38

5.5.3 The Combined Stop Solution . . . 39
5.5.4 Verification . . . 42

6 Dynamic Routing and Replanning **43**
6.1 Routing Architecture Overview . . . 43
6.2 Energy-Aware Edge Cost Function . . . 43
6.2.1 Base Travel Time . . . 43
6.2.2 Congestion Penalty . . . 43
6.2.3 Energy Component . . . 44
6.2.4 Adaptive Energy Weight . . . 44
6.3 Periodic Rerouting . . . 45
6.3.1 Trigger and Implementation . . . 45
6.3.2 Why setStop() Must Be Recalled After setRoute() . . . 45
6.4 Event-Driven Rerouting . . . 45
6.4.1 Congestion Detection . . . 45
6.4.2 Independence from Periodic Rerouting . . . 46
6.4.3 Performance Characteristics . . . 46
6.5 Dynamic Replanning . . . 47
6.5.1 Motivation and Problem Setting . . . 47
6.5.2 Trigger Conditions . . . 47
6.5.3 The Replanning Algorithm: _try_replan() . . . 48
6.5.4 Replanning and Rigid Tasks . . . 49
6.5.5 Combined-Stop Guard . . . 49
6.5.6 Design Rationale for Trigger Parameters . . . 49
6.6 Complexity and Scalability Analysis . . . 50
6.6.1 Motivation . . . 50
6.6.2 Theoretical Complexity Analysis . . . 50
6.6.3 Empirical Timing Results . . . 51

7 Simulation Result **53**
7.1 Simulation Setup . . . 53
7.2 Test Scenarios . . . 54
7.3 Scenario Results . . . 54
7.3.1 Scenario 1: Same-Station Consecutive . . . 54
7.3.2 Scenario 2: Rigid Priority . . . 56
7.3.3 Scenario 3: Battery Partition Stress . . . 59
7.3.4 Scenario 4: Mixed Priority + Congestion . . . 61
7.3.5 Computational Performance . . . 64
7.3.6 Issues Identified During Testing . . . 65
7.4 Discussion . . . 65

8 Future Developments **67**
8.1 Improvements Suggested by the Simulation Results . . . 67
8.2 Extensions of the Platform . . . 68

9 Conclusions **69**
9.1 Summary of Contributions . . . 69
9.2 Limitations . . . 70
9.3 Threats to Validity . . . 71
9.4 Final Remarks . . . 71

A System Parameter Reference **76**

B Function Quick Reference **79**

C Implementation Details **81**

Acknowledgements **83**

List of Figures

3.1 Execution pipeline of the platform: six sequential phases, from charging station configuration to network analysis. 9
3.2 Task placement GUI: road network with placed task markers and list of added tasks. 16
3.3 Charging station GUI: road network with the three placed stations. 17

4.1 Battery partition for the two worked examples (60 kWh total capacity): absolute minimum, driving reserve and energy available for V2G. 30

6.1 Periodic and event-driven rerouting events over a 60 s window of Scenario 1 (Chapter 7), taken from reroute_timing.csv. 46

7.1 Charging station configuration used in all scenarios: CS01 (west), CS02 (center) and CS03 (east). 53
7.2 Scenario 1: task placement on the map. 55
7.3 Scenario 1: state of charge of EGO. The first step corresponds to the combined stop of T1 and T2 at CS01, the second to T3 at CS03. 56
7.4 Scenario 2: task placement on the map. 57
7.5 Scenario 2: optimized schedule. The flexible task T3 is moved before the rigid task T2, which is kept at 10:00; EGO is planned to arrive at CS02 at 09:30 and wait for T2. 57
7.6 Scenario 2: state of charge of EGO. The first step corresponds to T1 at CS01, the second to the combined stop of T3 and T2 at CS02. 58
7.7 Scenario 3: task placement on the map, with the tasks at opposite ends of the network. 60
7.8 Scenario 3: optimized schedule with an initial SoC of 45%. The battery partition panel shows a floor of 23% and 13.3 kWh available for V2G. 60
7.9 Scenario 3: state of charge of EGO. The step at the end of the first stop corresponds to the V2G energy delivered, accounted for at stop completion; the dashed line shows the partition floor. 61
7.10 Scenarios 4 and 4b: optimized schedule (identical in both runs). EGO is planned to wait at CS01 before the rigid task T2. 62
7.11 Scenario 4 (sparse traffic): EGO speed profile. The plateau at about 1.6 m/s between $t = 445$ s and $t = 913$ s corresponds to the passage through the congested edges. 63
7.12 Scenario 4b (dense traffic): EGO speed profile. The passage through the congested edges lasts from $t = 452$ s to $t = 1085$ s, longer than in S4 because of the queue formed by the background vehicles. 64

List of Tables

3.1 Module summary. 8
3.2 Task dataclass fields. 11
3.3 Priority levels and corresponding flexibility windows. 12
3.4 ChargingStation dataclass fields. 14
3.5 SUMO stop state values relevant to this work. 21

4.1 Battery and V2G model parameters. 26

5.1 Permutation counts by task number. Rows 1–8 use an exact search, while rows 9–10 use a heuristic. 36

7.1 Test scenario definitions. 54
7.2 Scenario 1 results. 55
7.3 Scenario 2 results. 56
7.4 Scenario 3: evolution of the state of charge. 59
7.5 Scenario 4 results with sparse (S4) and dense (S4b) background traffic. 62
7.6 Computational cost of periodic rerouting and event-driven checks. 64

A.1 Main configurable parameters (config.py). 76

B.1 Main functions of the platform. 79

C.1 Working directory file structure. 81
C.2 Shared state variables in state.py. 82
C.3 FCD output schema (fcd.csv) 82

List of Acronyms

Acronym	Meaning
AC	Alternating Current
API	Application Programming Interface
ARERA	Autorità di Regolazione per Energia Reti e Ambiente (Italian Regulatory Authority for Energy, Networks and Environment)
CCS	Combined Charging System
CSV	Comma-Separated Values
DC	Direct Current
DLR	Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)
E-COSMOS	Electric Charging Optimization for Sustainable MObility Systems
EDF	Earliest Deadline First
EV	Electric Vehicle
E-VRPTW	Electric Vehicle Routing Problem with Time Windows
EVCC	Electric Vehicle Communication Controller
FCD	Floating Car Data
GSE	Gestore dei Servizi Energetici (Italian Energy Services Operator)
GUI	Graphical User Interface
IEA	International Energy Agency
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
MPC	Model Predictive Control
OCPI	Open Charge Point Interface
OCPP	Open Charge Point Protocol
ORM	Object-Relational Mapping
OSM	OpenStreetMap
PID	Proportional-Integral-Derivative (controller)
PNIEC	Piano Nazionale Integrato per l'Energia e il Clima (Italian National Integrated Energy and Climate Plan)
SECC	Supply Equipment Communication Controller
SoC	State of Charge
SUMO	Simulation of Urban MObility
TCP	Transmission Control Protocol
TDTSPTW	Time-Dependent Traveling Salesman Problem with Time Windows
TraCI	Traffic Control Interface
TSP	Traveling Salesman Problem
TSPTW	Traveling Salesman Problem with Time Windows
UVAM	Unità Virtuali Abilitate Miste (Virtual Aggregated Mixed Units)
V2G	Vehicle-to-Grid
VRPTW	Vehicle Routing Problem with Time Windows
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context and Motivation

As part of the broader transition toward smart cities, urban mobility and energy systems are increasingly converging, with transportation, energy and communication infrastructures being managed as interoperable, coordinated layers rather than isolated subsystems [1, 2].

The global automotive fleet is undergoing a profound transformation, driven by tightening emissions regulations, falling battery costs and growing consumer awareness of environmental impacts. In its 2024 Global EV Outlook, the International Energy Agency (IEA) [3] reported that almost 14 million new electric cars were registered worldwide in 2023, bringing the global stock to 40 million. Specifically in Italy, the national energy transition plan (PNIEC) [4] aims to have 6.6 million electric vehicles (EVs) on the road by 2030, which is a tenfold increase on the 2023 baseline.

This rapid deployment creates two interrelated challenges. The first is grid stress: if millions of EVs were to charge simultaneously during peak hours (typically in the early evening, when people return from work), the additional load could exceed the capacity of local distribution transformers and medium-voltage feeders. As noted by Daina et al. [5], unmanaged EV charging can strain grid capacity, particularly at the distribution level, where capacity bottlenecks are most easily reached. The second challenge is driver range anxiety. Unlike internal combustion vehicles, an EV with a depleted battery cannot be refuelled in minutes at any petrol station, so drivers must plan charging stops carefully to avoid becoming stranded.

Vehicle-to-Grid (V2G) technology addresses both of these issues simultaneously. By enabling EVs to discharge stored energy back to the grid, V2G supports frequency regulation, peak shaving, and the integration of renewable sources. The vehicle owner receives economic compensation for the energy delivered, which turns the battery from a cost center into a revenue-generating asset. Crucially, the only additional infrastructure V2G requires is a bidirectional charger compliant with the ISO 15118-20 standard [6]: essentially the same hardware used for conventional AC charging, extended with power electronics that enable reverse current flow.

However, V2G's potential is only realized when the discharge schedule is coordinated with the vehicle's mobility needs. For example, a driver with four appointments across the city cannot simply discharge the battery to its minimum level without risking having insufficient energy for the return journey. Conversely, a driver who keeps the battery fully charged to avoid range anxiety forgoes all V2G revenue. The optimal policy lies between these two extremes and depends on the day's specific task sequence, the road network, prevailing traffic conditions and the battery's current state of charge.

This thesis was carried out within the E-COSMOS project (Electric Charging Optimization for Sustainable MObility Systems). At the fleet level, the project encompasses a district-level dispatch and control architecture for connected autonomous EVs, which combines resource-constrained path planning with decentralized model predictive control (MPC)-based speed regulation to

support the grid [7, 8]. Within this framework, this thesis addresses the single-vehicle level: it presents a simulation platform that models and optimizes the trade-off between mobility needs and V2G discharge in a realistic urban environment. Unlike purely analytical approaches, which assume known travel times and ignore traffic, the platform couples a schedule optimizer with a microscopic traffic simulator (SUMO). This allows the system to respond to real-time congestion events, adapt the task order mid-simulation, and evaluate the V2G energy actually delivered along the simulated trip, rather than relying only on planned estimates.

1.2 Problem Statement

More formally, the problem addressed in this thesis can be stated as follows.

Given:

- a set of N tasks, $\mathcal{T} = \{T_i,\ i = 1, \ldots, N\}$, each characterized by a geographic location, a target time, a duration, and a priority level that determines the width of its flexibility window;
- a set of M charging stations, $\mathcal{S} = \{S_j,\ j = 1, \ldots, M\}$, each with a known geographic location and nominal V2G power;
- an electric vehicle (EGO) characterized by its battery capacity C_{bat}, initial state of charge SoC_0, and energy consumption rate e_{cons};
- a road network $G = (V, E)$, where V is the set of nodes (junctions) and E the set of directed edges (road segments), whose edge travel times vary over time according to traffic conditions;

find a visiting order π of the tasks and a set of route segments $R = \{r_k,\ k = 1, \ldots, N + 1\}$ that:

(i) maximize the number of tasks served within their flexibility windows;

(ii) minimize the total delay with respect to the tasks' target times;

subject to the following constraints:

(a) EGO visits all tasks in the order specified by π;

(b) the state of charge never falls below the minimum level SoC_{min} at any point of the journey;

(c) every route segment is feasible on G under the current traffic conditions.

The two objectives are ordered lexicographically: task coverage dominates punctuality. V2G discharge is not an objective of the ordering problem: for the selected schedule, it is estimated at each station stop, bounded by the battery state of charge. This reflects operational reality: a driver will not miss an appointment to earn a few cents of V2G revenue.

1.3 Objectives of the Thesis

The specific technical objectives of this work are as follows:

O1. Design and implement a task scheduler that assigns each task to the nearest charging station, finds the optimal visit order for up to eight tasks (and a near-optimal order for larger instances), and computes the battery partition between driving reserve and V2G headroom.

O2. Develop a microscopic simulation engine that controls an EV in SUMO via TraCI, executing the scheduled task sequence while adapting to real-time traffic.

O3. Implement a dual-mode routing architecture that combines periodic and event-driven rerouting with an energy-aware edge cost function.

O4. Implement a task replanning mechanism that invokes the scheduler during the simulation, after each stop and whenever estimated delays exceed configurable thresholds.

O5. Identify and resolve the same-station consecutive stop problem, a TraCI-specific failure mode which, to the best of the author's knowledge, has not been analyzed or addressed in the scientific literature on EV scheduling.

O6. Provide a quantitative scalability analysis of the routing algorithms.

1.4 Document Structure

Chapter 2 provides an overview of the current state of V2G technology, EV scheduling and microscopic traffic simulation. Chapter 3 describes the simulation platform developed in this thesis, covering the overall system architecture, data model, graphical user interfaces and SUMO-based simulation engine. Chapter 4 presents the battery and V2G energy model, including the battery partition between driving reserve and V2G. Chapter 5 covers task optimization, including the scheduler and the combined-stop mechanism. Chapter 6 addresses dynamic routing and replanning, together with the complexity and scalability analysis of the routing algorithms. Chapter 7 reports the simulation result. Chapter 8 outlines directions for future work and Chapter 9 discusses the thesis's contributions, limitations, and threats to validity. Appendix A lists the configuration parameters of the platform, Appendix B its main functions, and Appendix C further implementation details.

Chapter 2

State of the Art

2.1 Vehicle-to-Grid Technology

2.1.1 Origins and Foundational Work

Kempton and Tomić [9, 10] formalized the concept of using parked electric vehicles as distributed grid resources in a pair of landmark papers published in 2005. Their key insight was statistical: personal vehicles are parked and idle for over 90% of the day. A fleet of 10,000 EVs, each with 20 kWh of usable V2G capacity, represents 200 MWh of distributed storage, comparable to a small peaking power plant, available at a fraction of the capital cost. They demonstrated that V2G could economically provide frequency regulation, spinning reserve and peak power services, with regulation being the most lucrative due to its high payment rate per kWh.

Subsequent work by Sortomme and El-Sharkawi [11] addressed V2G from the perspective of an aggregator participating in electricity markets, formulating the optimal combined bidding of regulation and spinning reserve services provided by a fleet of EVs. In their model, each EV is treated as a controllable resource whose charging and discharging must remain compatible with the energy requirements of its owner by the time of departure. This family of 'smart charging' formulations, in which the needs of the driver act as constraints on the grid service, remains a dominant paradigm in the academic literature. A broader, more multidisciplinary overview of the role of EVs in grid stability and flexibility can be found in [12].

2.1.2 Hardware Standards and Communication Protocols

Practical V2G deployment requires bidirectional power electronics and a standardized communication protocol between the vehicle and the charging station. The ISO 15118 standard [6] defines the Vehicle-to-Grid Communication Interface (EVCC/SECC), which covers both AC and DC charging. The 2022 revision (ISO 15118-20) explicitly incorporates bidirectional power transfer, thus enabling V2G at both AC Type-2 chargers and DC CCS chargers.

The Open Charge Point Protocol (OCPP 2.0.1) [13] defines communication between charging stations and network operator backend systems, and the Open Charge Point Interface (OCPI) handles roaming between different operator networks. These standards together form the communication stack assumed in this thesis, where the 22 kW Type-2 AC station is the default hardware model.

2.1.3 Italian Regulatory Framework

In Italy, the participation of distributed resources in grid services has been enabled by the pilot projects launched by ARERA with Resolution 300/2017/R/eel [14]. Among these, the Virtual Aggregated Mixed Units (UVAM) project, approved with Resolution 422/2018/R/eel [15] and managed by Terna, allows aggregated consumption, production and storage units to provide

dispatching services through an aggregator, and explicitly includes storage systems used for electric mobility. The specific criteria for V2G integration were then defined by the Ministerial Decree of 30 January 2020 [16], in implementation of which GSE published an information procedure to protect the owners of vehicles taking part in V2G services [17].

In this framework, V2G energy is remunerated through the dispatching services market rather than through a fixed feed-in tariff. For modeling purposes, this thesis therefore assumes a constant default tariff of 0.10 €/kWh, computed as 20% of a reference electricity price of 0.50 €/kWh; the impact of alternative tariff structures is identified as future work (Chapter 8).

2.2 Electric Vehicle Scheduling

2.2.1 Problem Classification

Scheduling an electric vehicle (EV) to visit a set of locations while satisfying time window constraints is a special case of the Traveling Salesman Problem with Time Windows (TSPTW). Unlike the classical TSP, the objective in the TSPTW is not merely to minimize the total distance, but rather to identify a feasible route that visits each node within its designated time window. The problem is NP-hard, i.e. no algorithm is known that solves all its instances in a time that grows polynomially with the number of nodes, so that the computational effort of exact methods grows exponentially in the worst case [18]. Even determining whether a feasible solution exists is NP-complete [19], i.e. it belongs to the hardest class of decision problems whose solutions can nevertheless be verified in polynomial time. For the symmetric metric case, the best-known exact algorithm has $O(2^N \times N^2)$ complexity via dynamic programming (Held–Karp), making exact solutions impractical for N $>$ 20.

The time-dependent variant (TDTSPTW), in which travel times depend on departure time due to traffic patterns, is even more challenging and has received less theoretical attention. In practice, however, traffic simulation systems such as SUMO provide travel time estimates for any given departure time, meaning that the time-dependent case is handled implicitly by the simulation rather than being modeled analytically in the scheduler.

2.2.2 Heuristic Approaches

Due to the NP-hardness of the general problem, most practical EV scheduling systems rely on heuristics and metaheuristics, such as genetic algorithms, simulated annealing and tabu search. Among local search methods, the 2-opt operator, originally proposed by Croes [20], is one of the most widely used for routing problems: it reverses a segment of the tour and accepts the change if the objective improves. Savelsbergh [19] showed that local search operators of this kind can be applied efficiently to routing problems with time windows, which is the setting of this thesis. One 2-opt sweep requires $O(N^2)$ candidate evaluations and converges quickly from a good initial solution.

For small instances (N $\leq$ 8–10), exact enumeration remains competitive. At 40,320 evaluations for N = 8, and assuming an evaluation time of the order of 10 μs on modern hardware, the complete search finishes in under half a second, well within the interactive response time budget. This project uses N = 8 as the threshold, applying an exact search below this and a greedy + 2-opt search above it.

2.2.3 Energy-Constrained Scheduling

Integrating battery constraints into the scheduling problem creates a link between the order in which tasks are carried out and the amount of energy available for each segment. Schneider et al.'s [21] work on the Electric Vehicle Routing Problem with Time Windows (E-VRPTW) showed

that energy constraints can render otherwise feasible schedules infeasible, and that optimal routes may require intermediate charging detours.

This project's system addresses this coupling through its battery partition model. By reserving a dynamic minimum state of charge (SoC) floor that adds the estimated driving energy, plus a safety margin, to the absolute minimum state of charge, the system prevents V2G discharge from consuming the energy needed for driving. Energy feasibility is therefore not an explicit constraint of the task ordering optimization, but is handled separately by the partition, a simplification that is adequate for the short urban itineraries considered in this work.

2.3 Microscopic Traffic Simulation with SUMO

2.3.1 SUMO Architecture

SUMO (Simulation of Urban MObility) [22, 23] is an open-source traffic simulation package that has been developed at the German Aerospace Center (DLR) since 2001. It implements a space-continuous, time-discrete simulation model, whereby each vehicle is represented as an individual agent with its own route, vehicle type and behavioral parameters. Time is advanced in fixed steps (typically 0.1–1.0 s) in the simulator, and the position, speed and acceleration of each vehicle are updated at every step using a car-following model.

The default car-following model in SUMO is the Krauss model [24], which is a stochastic variant of the classic Gipps model [25]. This model computes the safe speed for each vehicle based on the gap to the leading vehicle, the leader's speed and a random perturbation term that produces realistic speed variance. Lane-changing behavior is governed by the LC2013 model [26], which balances the desire to overtake slow vehicles with the safety requirements of lane-changing maneuvers.

SUMO reads the road network from a .net.xml file generated by the NetConvert tool, which can import data from OpenStreetMap [27], VISUM, Vissim and several other formats. The OSM import pipeline used in this thesis extracts road geometry, speed limits, lane counts, junction types and traffic light programs. The resulting network, covering a portion of the city of Pavia, contains 2,051 nodes and 4,136 edges, with road types ranging from motorway-level arterials to residential, single-lane streets. The SUMO network setup and base simulation pipeline, as described in this section, were developed in collaboration with Galassi [28] and Faiola [29]. Their theses apply the same simulation platform to related problems in the context of driving rehabilitation.

2.3.2 The TraCI Interface

The TraCI (Traffic Control Interface) is a client-server API which makes SUMO's internal state available to an external controller process. The API is implemented as a TCP socket protocol with a well-defined binary message format. At each simulation step, the external controller can read vehicle positions, speeds and states; modify vehicle routes and speeds; insert and remove vehicles; and query the road network structure. The SUMO distribution provides Python, Java and C++ bindings; the platform developed in this thesis uses the Python binding exclusively.

A key performance feature of TraCI is the subscription mechanism. Rather than issuing individual queries for each vehicle at each step (which incurs one round trip per variable per vehicle), the controller subscribes to a set of variables for all vehicles in a single call. Subsequent steps then return all subscribed values in a single message, reducing the number of socket round trips from $O(N \times V)$ to a single exchange per step, where N is the number of vehicles and V is the number of variables (the amount of data transferred still grows with $N \times V$). The platform subscribes to seven variables for all vehicles at initialization and uses getAllSubscriptionResults() at each step.

2.3.3 Charging Station Modeling in SUMO

SUMO supports charging stations as special lane segments, which are defined in an additional XML file. Each charging station is specified by its lane ID, start position, end position and charging power. Vehicles can be programmed to stop at a charging station using the setStop() TraCI command and the STOP_CHARGING_STATION flag. When a vehicle occupies a charging station, SUMO automatically applies the charging power to the battery model of the vehicle (if a battery device is configured).

This work uses the SUMO charging station XML file to define the physical stop locations for V2G events. However, battery accounting (state of charge tracking and V2G energy calculation) is handled by the Python layer rather than SUMO's built-in battery device. This gives full control over the V2G model and avoids the complexity of configuring SUMO's internal electric vehicle extension ('battery' device).

2.4 Related Work

2.4.1 EV and Fleet Simulation

Several works have coupled models of urban traffic with models of EV energy demand. For example, Arias et al. [30] predicted the time-spatial charging-power demand at public fast-charging stations in an urban area, combining a Markov-chain traffic model of a real road network with traffic data obtained from closed-circuit television cameras. Approaches of this kind model traffic at an aggregate level in order to estimate the charging demand, rather than controlling individual vehicles during the simulation.

At a larger scale, Bischoff and Maciejewski [31] simulated the city-wide replacement of private cars with autonomous taxi fleets in Berlin using the agent-based simulator MATSim, with real-time dispatching of the private car trips of the whole city. Their work shows how external dispatching logic can be coupled with a transport simulation at city scale, although with a mesoscopic, queue-based traffic model rather than the microscopic, step-by-step vehicle control via TraCI adopted in this thesis.

2.4.2 Task Scheduling with Traffic-Aware Routing

In terms of problem formulation, the work of Gendreau et al. [32] is among the closest to this one. They study a dynamic vehicle routing problem with soft time windows, in which new requests arrive during the execution of the current plan and the planned routes must be updated in real time. Their parallel tabu search re-optimizes the routes whenever new information becomes available, which is similar in spirit to the replanning mechanism of this project, although in their case re-optimization is driven by the arrival of new requests rather than by estimated delays. The main difference is that their work considers a fleet of vehicles serving customer requests, whereas this one focuses on a single vehicle following the agenda of its driver and integrates V2G discharge at the charging stops.

2.4.3 The Same-Station Stop Problem

To the best of the author's knowledge, the consecutive stop problem at the same station (a scenario in which two successive tasks in a TraCI-controlled SUMO simulation are assigned to the same charging station, resulting in the second stop command failing) has not been analyzed or addressed in the scientific literature on EV scheduling. While the SUMO issue tracker contains several reports of 'stop not downstream' errors in various contexts, none specifically address charging-station stops for EV task scheduling. The combined stop solution presented in Chapter 5 of this thesis is an original contribution.

Chapter 3

Simulation Tool

3.1 Design Principles

This project was designed based on four guiding principles. The first of these is separation of concerns: each module has a clearly defined responsibility and only communicates with other modules through specified data structures. This design facilitates unit testing and enables individual modules to be replaced or updated without impacting the rest of the system.

Secondly, configurability: all parameters that may require adjustment, such as battery capacity, routing weights, timing thresholds and V2G tariffs, are centralized in config.py, rather than being scattered as magic numbers throughout the code. This enables systematic sensitivity analysis by modifying a single file.

Thirdly, the system is designed to run correctly even when optional components (charging stations, GUI) are absent, ensuring graceful degradation. If no stations are configured, tasks are visited directly at their map locations. If event-driven rerouting fails to find an alternative route, the periodic rerouter corrects the situation at the next cycle.

Fourthly, observability: routing and replanning events (periodic rerouting, event-driven rerouting and replanning) and the state of charge of EGO are logged in structured data structures (state.py), while V2G discharges and stop completions are reported in the console log; traffic data and rerouting timings are written to CSV files. This enables post-simulation analysis without the need to rerun the simulation.

3.2 Module Structure

The codebase is organized into 14 Python modules. Table 3.1 lists each module with its primary responsibility and the other modules it depends on.

Table 3.1: Module summary.

Module	**Dependencies**	**Responsibility**
config.py	None	Global constants: battery, routing, SUMO, V2G parameters
tasks.py	config	Task dataclass with computed properties
charging_stations.py	config, sumolib	ChargingStation dataclass, JSON I/O, SUMO XML generation
scheduler.py	config, tasks, charging_ stations, sumolib	Schedule optimizer, battery partition, V2G estimation

Table 3.1 (continued)

Module	Dependencies	Responsibility
task_gui.py	config, tasks, sumolib, tkinter	Interactive map for task placement; loads and persists calendar_tasks.csv; click-to-edit on existing task markers
charging_station_gui.py	config, charging_stations, sumolib, tkinter	Interactive map for station placement
state.py	None	Global in-memory state: route history, rerouting logs, SoC history
utils.py	config	Utility functions: clamp(), edge ID parsing, Euclidean distance
routing.py	config, sumolib	Candidate edge filtering, A* for background traffic
simulation.py	config, state, routing, traci	Shared simulation utilities: speed control, FCD, background traffic
simulation_ecosmos.py	config, tasks, charging_ stations, state, simulation, traci	EGO control loop: stop management, V2G, replanning, combined stop
main_ecosmos.py	all modules	Pipeline orchestrator: GUI → scheduler → simulation → analysis
plots.py	config, state	Post-simulation visualizations: EGO speed and SoC, traffic diagrams, complexity
graph_analysis.py	config, state, networkx	NetworkX analysis: route snapshots, scalability

3.3 Pipeline Execution Flow

The execution flow is orchestrated by main_ecosmos.py via a series of sequential phases. Each phase is implemented as a separate function, either local to main_ecosmos.py or provided by the corresponding module, and is called by the main() entry point. Figure 3.1 illustrates this sequence of phases.

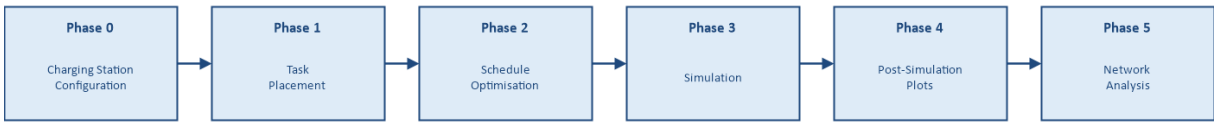


Figure 3.1: Execution pipeline of the platform: six sequential phases, from charging station configuration to network analysis.

Phase 0 (Charging station configuration): The user is presented with a menu offering the option to add new stations via the graphical user interface (GUI), reload an existing configuration from JSON or skip station-based routing (task-edge stops only). The selected configuration is saved to charging_stations.json for reuse.

Phase 1 (Task placement): The Tkinter GUI is launched and displays the SUMO road network as a polyline canvas. The user clicks on locations on the map to define tasks and enters a name, target time, priority and duration for each task. The GUI then returns a list of task objects with the target time in raw wall-clock seconds.

Phase 2 (Schedule optimization): The scheduler assigns stations, optimizes the task order and computes the battery partition using the wall-clock target times, so that the schedule can be displayed in HH:MM format; the planned departure of EGO is set 30 minutes before the earliest task. A confirmation dialog box displays the schedule summary and V2G estimates, and the user can either accept them or re-enter the GUI to modify the tasks. Once the schedule is confirmed, the _normalize_task_times() function converts the target times to simulation seconds, mapping the earliest task to 1800 s (30 minutes) after EGO_DEPART, consistently with the departure assumed by the scheduler.

Phase 3 (Simulation): run_simulation_with_tasks() opens a SUMO process, injects EGO and background traffic, and runs the step-by-step control loop until all tasks are completed. FCD and rerouting logs are written to CSV files in the working directory.

Phase 4 (Post-simulation plots): The plotting functions read the FCD output and SUMO's edge data and generate the EGO speed and acceleration profiles, a space-time diagram of the most trafficked lane and the fundamental diagrams (flow, density and speed) of the corresponding edge. The SoC history of EGO is plotted together with the partition floor, and make_complexity_plots() generates the computational cost comparison and scalability projection figures from the rerouting logs, which are also saved to reroute_timing.csv. The figures selected for this thesis are also exported as vector PDF files.

Phase 5 (Network analysis): run_networkx_analysis() loads the route history and the road network into NetworkX graphs, checks their consistency with the SUMO network and generates snapshots of the successive routes followed by EGO.

3.4 Working Directory Structure

Each simulation run uses a dedicated working directory (nave_run by default) that contains all the input and output files. This design ensures that all the artifacts needed to reproduce a result are located together. Since each run overwrites the output files of the previous one, the outputs of each simulation run must be copied to a separate folder before starting the next run. The key files of the working directory are listed in Table C.1 in Appendix C.

3.5 Data Model

The data model of this thesis is intentionally simple, consisting of two primary entities (Task and ChargingStation), plus a shared state singleton (state.py). This minimalism is deliberate. The problem does not require a relational scheme or a persistent object graph: all data is created at the start of a session, used by the simulation and then discarded at the end. The Python dataclass decorator is used throughout to provide automatic ___init___, ___repr___ and ___eq___ methods, eliminating the need for a full ORM framework.

3.6 The Task Dataclass

3.6.1 Core Fields

A task represents a single stop that the EGO driver must make during the day. The concept is intentionally generic and can be used to model a medical appointment, parcel delivery, business meeting or V2G charging stop, for example. The system does not distinguish between these use cases: a task is simply a location with a desired arrival time and minimum stay duration.

Table 3.2: Task dataclass fields.

Field	Type	Description
task_id	str	Unique identifier, e.g. 'T1'. Used in logging and GUI labels.
name	str	Human-readable label entered by the user.
edge_id	str	SUMO edge ID nearest to the clicked map location.
x, y	float	Map coordinates in the SUMO coordinate system (meters, origin = bottom-left corner of the bounding box).
target_time	float	Desired start time in simulation seconds. Normalized from wall-clock by main_ecosmos.py.
priority	int	Integer 1–5 controlling the flexibility window. 1 = rigid (±0 s), 5 = very flexible (±3600 s).
duration	float	Duration of the activity at the task location [s]. Does not include walking time.
original_target_time	Optional[float]	Wall-clock seconds as entered by the user. Preserved for display only; not used in calculations.
scheduled_time	Optional[float]	Estimated start time computed by the scheduler (set after optimization).
scheduled_order	Optional[int]	Position in the optimized visit sequence (1-indexed).
assigned_station_id	Optional[str]	ID of the charging station assigned by the scheduler. None if no stations are configured.
stop_edge_id	Optional[str]	SUMO edge ID of the assigned station. EGO parks here, not at edge_id.
walk_time_s	float	One-way walk time to reach this task: from the station for a standalone stop or the first task of a combined stop, from the previous task's location for the rest of a combined stop (direct transfer, no trip back to the car in between).

Table 3.2 (continued)

Field	Type	Description
walk_out_time_s	float	One-way walk time back to the car after this task: equals walk_time_s for a standalone stop, 0 for a non-last task in a combined stop, and task-to-station for the last task of a combined stop.
source	str	'calendar' if loaded from / saved to calendar_tasks.csv, 'adhoc' if added during the session and not persisted (Section 3.12).

3.6.2 Flexibility Windows and Priority

The priority field represents a trade-off between schedule rigidity and operational flexibility. A task with a priority of 1 is rigid: its flexibility window has zero width, so it is on time only if it starts exactly at its target time. This is not imposed as a hard constraint, but follows from the lexicographic objective of the scheduler, which favors any order that keeps rigid tasks on time (Section 6.5.4). In contrast, a priority-5 task can be visited up to one hour early or late without penalty. The mapping from priority to window size reflects common real-world time sensitivities.

Table 3.3: Priority levels and corresponding flexibility windows.

Priority Level	Window [seconds]	Typical Use Case
1 - Rigid	±0 s (= 0 min)	Medical appointments, official meetings with fixed start times
2 - Low flexibility	±900 s (±15 min)	Business meetings with a brief grace period
3 - Medium	±1800 s (±30 min)	Deliveries with a half-hour window
4 - High	±2700 s (±45 min)	Informal appointments, flexible pickups
5 - Very flexible	±3600 s (±60 min)	Background errands with no strict timing requirement

The scheduler uses the earliest_time and latest_time computed properties to determine whether a candidate ordering is feasible:

$$\text{earliest_time}(T) = \text{target_time}(T) - \text{flex_window}(\text{priority}(T)) \tag{3.1}$$

$$\text{latest_time}(T) = \text{target_time}(T) + \text{flex_window}(\text{priority}(T)) \tag{3.2}$$

A task is considered 'on time' if its scheduled start time falls between the earliest and latest times given by Equations (3.1) and (3.2). For priority-1 tasks, this interval reduces to a single point, meaning that arriving on time is both necessary and sufficient for schedule feasibility.

3.6.3 The stop_duration_s Property

The 'stop_duration_s' property calculates the time that this task contributes to the total time that EGO remains parked at the charging station. This is the sum of three components: the time spent walking to reach the task, the duration of the task itself, and the time spent walking onwards.

$$\text{stop_duration_s}(T) = \text{walk_time_s}(T) + \text{duration}(T) + \text{walk_out_time_s}(T) \tag{3.3}$$

For a standalone stop (i.e. not part of a combined stop), walk_out_time_s in Equation (3.3) equals walk_time_s, as the driver walks from the station to the task and back along the same route.

Therefore,

$$\text{stop_duration_s}(T) = 2 \times \text{walk_time_s}(T) + \text{duration}(T) \tag{3.4}$$

However, for a task that is part of a same-station combined stop (see Section 5.5.3), the semantics change. In this case, walk_time_s represents the one-way walk from the previous stop point (i.e. the station for the first task in the chain or the previous task's location for subsequent tasks). Furthermore, walk_out_time_s is only non-zero for the final task in the chain, representing the walk from that task back to the car. Every other task in the chain has walk_out_time_s equal to zero because the driver walks directly to the next task instead of returning to the car in between.

This distinction is important because summing stop_duration_s for every task in a combined-stop chain gives the total time that the EGO is parked without counting the return trip to the car between consecutive tasks twice (Section 5.5.3). This property is also critical for the simulation. SUMO's setStop() command requires the exact duration in seconds; an oversized duration would keep EGO idle at the station for longer than necessary, delaying the V2G energy it could deliver at the next stop.

3.7 The ChargingStation Dataclass

A Charging Station represents a physical, V2G-capable parking space on the road network. Each station is linked to a specific SUMO edge and has a nominal power rating. This edge is used as both the destination for the EGO route and the location of the SUMO charging station element in the additional XML file.

The station_id is the identifier used by the code, whereas the name is only displayed; the two should be kept consistent, since the confirmation dialog and the console log show the name, while the combined-stop logic compares the IDs.

The write_charging_stations_additional() function generates the SUMO XML for each station. The stop position is computed as max(5.0, lane_length − 1.0) to ensure that the stop endpoint remains within the lane bounds, regardless of the lane's length. The start position is set 20 meters upstream of the endpoint, providing EGO with a 20-meter stopping window. This generous stopping window reduces the risk of SUMO rejecting the stop due to a position mismatch.

Table 3.4: ChargingStation dataclass fields.

Field	Type	Description
station_id	str	Unique identifier, e.g. 'CS01'. Must be unique across all stations.
name	str	Human-readable label shown in the GUI and log output.
edge_id	str	SUMO edge where EGO will stop. Must be a valid edge in the network.
x, y	float	Centroid coordinates of the nearest junction. Used for distance calculations.
power_kw	float	Nominal V2G power [kW]. Default: 22.0 kW (Type-2 AC).
junction_id	Optional[str]	SUMO junction ID of the nearest network node.

3.8 Shared State (state.py)

The state module provides a set of variables at module level that are written by the simulation engine and read by the analysis modules. Using these globals avoids the complexity of passing large state objects through multiple layers of function calls while maintaining a clear boundary between the simulation layer, which writes, and the analysis layer, which reads.

The complete list of shared variables is given in Table C.2 in Appendix C.

3.9 Graphical User Interfaces

Both GUIs are implemented using the standard Python Tkinter toolkit to avoid external dependencies. The road network is rendered as a set of polylines on a Tkinter Canvas widget. Each edge is drawn as a sequence of line segments connecting SUMO shape points. The rendering code uses sumolib's net.getEdges() to iterate over all edges, and shape.toList() to obtain the sequence of (x, y) coordinates for each one.

A coordinate transformation is then applied to map SUMO's metric coordinate system (with the origin at the bottom left of the network bounding box) to Tkinter's pixel coordinate system (with the origin at the top left of the canvas). The transformation is:

$$\text{px} = (x_{\text{sumo}} - x_{\min})/(x_{\max} - x_{\min}) \times \text{canvas_width} \tag{3.5}$$

$$\text{py} = \text{canvas_height} - (y_{\text{sumo}} - y_{\min})/(y_{\max} - y_{\min}) \times \text{canvas_height} \tag{3.6}$$

where the y coordinate is flipped because the SUMO coordinate system is bottom-up, whereas the canvas is top-down.

The scale factors are computed once at initialization and reused for all subsequent coordinate conversions, including the reverse transformation from click position to SUMO coordinates.

3.10 Task Placement GUI

3.10.1 Nearest-Edge Mapping

When a user clicks on the canvas, their click position is converted into SUMO coordinates, and the nearest road edge is identified. The distance metric used is the Euclidean distance from the click point to the midpoint of each edge. Three classes of edge are excluded from the candidate set:

- Very short edges: too short to support a reliable stop position.
- The EGO start edge (EGO_START_EDGE_ID): placing a task here would cause the first route segment to become invalid.
- Dead-end edges with no outgoing passenger arcs: EGO would be teleported out of the network after stopping.

The method net.getEdge(edge_id).getOutgoing() is used to exclude dead-end edges: among the outgoing connections of each edge, those whose vehicle class is not 'passenger' are filtered out. This prevents a class of runtime errors that would otherwise be very difficult to diagnose, as SUMO's error message ('Vehicle ... may not use lane ...') does not mention the stop command.

3.10.2 Task Dialog

After selecting the nearest edge, a Tkinter Toplevel dialog box with four input fields is displayed.

- Name: free-text label for the task (e.g. 'Hospital appointment')
- Target time: entered in HH:MM format and validated with a regular expression. The wall-clock time is stored in original_target_time and is later converted to simulation seconds by _normalize_task_times() in main_ecosmos.py.
- Priority: selected via a radio button group showing all five levels with their English descriptions. The default is Priority 3 (Medium).
- Duration: entered in minutes and converted to seconds on save.

The dialog box includes a 'Delete last task' button that removes the most recently placed task from both the internal list and the canvas marker. This enables users to correct mistakes without having to restart the GUI.

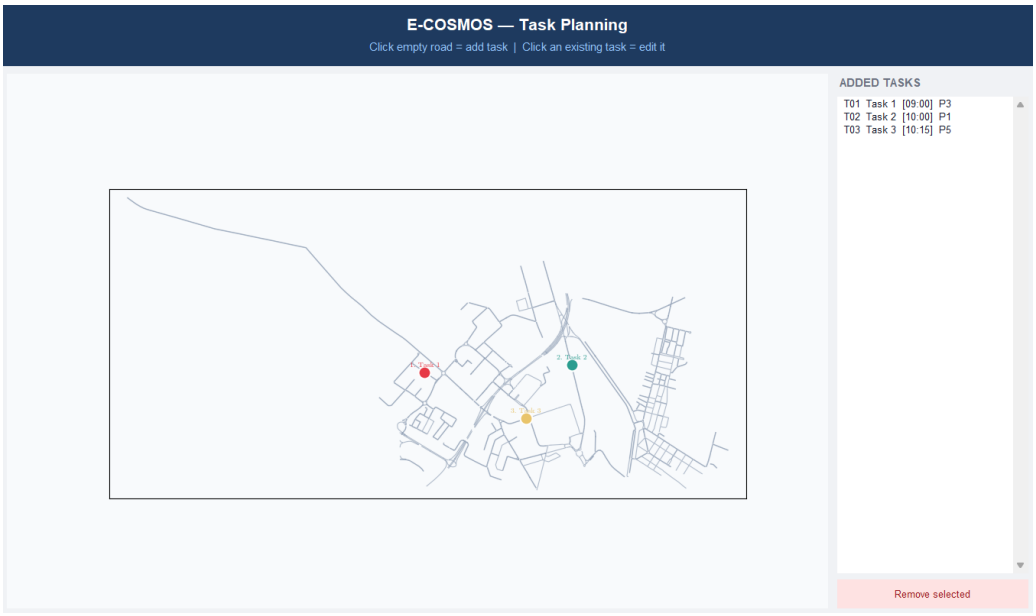


Figure 3.2: Task placement GUI: road network with placed task markers and list of added tasks.

3.11 Charging Station GUI

3.11.1 Junction-Based Placement

Unlike the task GUI, which maps clicks to edges, the station GUI maps them to network junctions. The nearest junction to the click point is identified and the station is anchored to the first incoming edge of that junction. This junction-based placement method makes the station easier to reach, since junctions are the endpoints of multiple edges and can usually be approached from several directions.

The incoming edge is selected according to a preference order: multiple-lane edges are preferred over single-lane edges, and longer edges are preferred over shorter ones. This maximizes the probability that the generated stop position is valid.

3.11.2 Validation Rules

Before accepting a station placement, the GUI applies three validation rules.

- Rule 1: EGO start edge exclusion: if the selected incoming edge is EGO_START_EDGE_ID, the placement is rejected and an error dialog appears. Placing a station on the start edge would interfere with EGO's initial departure.
- Rule 2: Dead-end check: if the selected edge has no outgoing passenger-accessible arcs, the placement is rejected. This addresses the second issue identified in the combined stop analysis (Section 5.5.3), whereby EGO is removed from the network after stopping on a terminal edge.
- Rule 3: Duplicate check: If a station already exists close to the click point, a warning is displayed, asking the user to confirm the placement. This prevents the accidental creation of overlapping stations.

Confirmed stations are immediately written to charging_stations.json, so the configuration is preserved even if the GUI closes unexpectedly. If it exists, the JSON file is loaded at startup, allowing the user to resume a previous configuration without re-entering all stations.

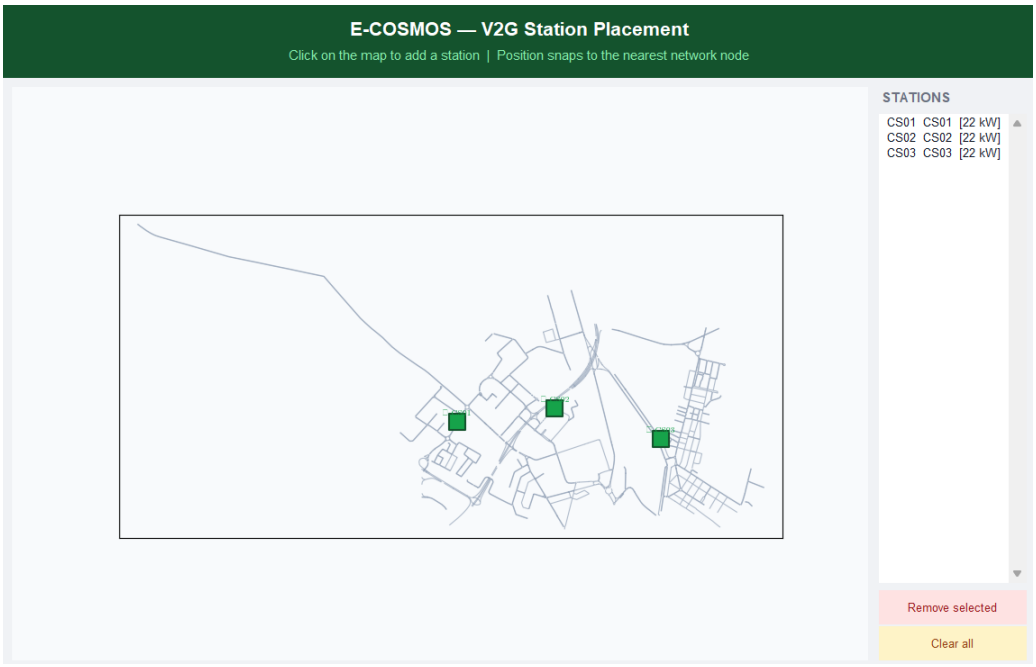


Figure 3.3: Charging station GUI: road network with the three placed stations.

3.12 Calendar CSV and Task Persistence

Both the task and charging station GUIs support the persistence of user input across simulation runs. Charging stations are always persisted (see Section 3.11.2), whereas tasks are persisted selectively. This distinguishes between two types of input that naturally arise in the target use case: tasks that are planned in advance, such as entering an appointment into a phone calendar, and tasks that occur spontaneously to the driver when they start the car for the day.

3.12.1 CSV Format and Validation

If present, pre-planned tasks are read from the calendar_tasks.csv file in the working directory before the task-placement map is shown. Each row specifies:

- name: free-text label
- Edge ID: the SUMO edge ID of the task location. This must be a valid, selectable edge and is subject to the same exclusion rules as manual placement (see Section 3.10.1)
- priority: an integer from 1 to 5
- time: target time in HH:MM format
- duration: the activity's duration in minutes.

Each row is validated independently. Rows with an out-of-range priority, a duration of less than zero, an unparsable time or an edge ID that is not a valid, selectable edge are rejected and reported to the user in a single warning dialog listing every rejected row by line number. This approach avoids failing the whole import. Valid rows are converted to Task objects as if they had been placed by clicking on the map with a source of 'calendar'.

After loading, a single confirmation dialog summarizes every loaded task (name, time, priority and duration) and asks whether the user wants to modify any of them. This avoids opening one dialog per row (with a calendar of six tasks, six sequential dialogs would be tedious), while still giving the user a chance to correct entries before the map opens.

3.12.2 Click-to-Edit and Selective Persistence

The task-placement map (Section 3.10) remains unchanged for both calendar and ad hoc tasks. After loading, the user can click anywhere on the map to add a new task, just as they would in a session without a calendar. The only difference is how an existing task can be edited. Clicking within a small radius of an existing marker opens the same task dialog pre-filled with the task's current values instead of creating a new task. The dialog's title and confirmation button change accordingly ('Edit task'/'Update' rather than 'New task'/'Add'). However, this click-to-edit interaction only allows the name, time, priority and duration of a task to be changed. Moving a task to a different edge still requires removing it and placing it again, since the dialog does not re-run the nearest-edge lookup.

The task dialog also features a checkbox labeled 'Save to permanent calendar'. This is unchecked by default when a task is added by clicking the map. When checked, it marks the resulting task's source field as 'calendar'. The checkbox mirrors the task's current source when editing an existing task. This means that a task can be promoted to, or removed from, the permanent calendar at any time. When the user confirms, the run_task_selection_gui() function only writes the tasks with a source of 'calendar' back to the calendar_tasks.csv file (Table B.1), overwriting the file. This mirrors exactly how the charging_stations.json file is persisted in Section 3.11.2. Tasks with a 'source' value of 'adhoc' are used for the current simulation, but are never written to disk. This means that a task added on a whim when starting the car will not clutter the calendar for the next run.

This design provides the scheduler with a single, uniform task list, regardless of origin: the distinction only exists at the GUI/persistence boundary and never in the scheduling or simulation logic downstream.

3.13 Simulation Engine

3.13.1 Overview

The simulation engine (simulation_ecosmos.py) is the most complex module in this project. It opens a SUMO process, injects EGO and background traffic, and executes a step-by-step control loop that manages stop sequences, battery state, V2G discharge, route adaptation and replanning simultaneously. The module communicates exclusively with SUMO through TraCI calls: no direct access to SUMO's internal state is required.

The entry point is the run_simulation_with_tasks() function, which receives an ordered list of tasks from the scheduler and returns the final state of charge (SoC) of EGO at the end of the simulation. The V2G energy delivered at each stop and the task progress are reported in the console log.

3.13.2 SUMO Configuration Generation

Before the simulation starts, the _write_sumocfg_for_tasks() function generates a .sumocfg XML configuration file. The key parameters are:

- <net-file>: nave_2.net.xml, the Pavia OSM road network.
- <additional-files>: a comma-separated list of additional files, including the charging station definitions (charging_stations.add.xml) when stations are configured.
- <gui-settings-file>: the visualization settings used by SUMO-GUI.
- <begin> and <end>: simulation time interval, from SIM_BEGIN to sim_end.
- <step-length>: 0.5 s (STEP_LENGTH in config.py).

- <tripinfo-output>: tripinfo.xml, written at the end of the simulation.

The end time of the simulation is computed by _compute_sim_end() as follows:

$$\text{sim_end} = t_{\text{depart}} + \sum_i \text{stop_duration_s}(T_i) + (N+1) \times t_{\text{leg}} + t_{\text{buffer}} \tag{3.7}$$

where $t_{\text{leg}} = 300$ s is the estimated duration of each inter-task leg and $t_{\text{buffer}} = 600$ s is a safety buffer.

This estimate is intentionally generous. If all the tasks are completed before sim_end, the simulation will end early as soon as EGO has completed all its tasks and left the network. The extra buffer prevents premature termination in the event of heavy congestion.

3.13.3 Background Traffic and Congestion Injection

Background traffic is generated at the start of the simulation by generate_background_trip_specs() and inserted into the network by inject_background_vehicles(). The departures of the background vehicles are spread uniformly over the whole simulation, with a period of max(TRAFFIC_PERIOD, sim_end / BG_VEHICLE_COUNT), so that traffic is present throughout the trip of EGO. With the default value BG_VEHICLE_COUNT = 100, the network carries a light traffic load; higher values produce denser traffic.

For testing purposes, the engine can also inject a congestion event: when CONGESTION_ENABLED is set, the maximum speed of the edges listed in CONGESTION_EDGE_IDS is reduced to CONGESTION_SPEED at the simulation time CONGESTION_START_S, using traci.edge.setMaxSpeed(). This feature is disabled by default and was used for the congestion scenarios of Chapter 7.

3.13.4 EGO Vehicle Injection

Initial Route Computation

The _inject_ego_first_segment() function computes the initial route from the EGO_START_EDGE_ID to the effective_stop_edge of the first task.

```
dest = first_task.effective_stop_edge
stage = traci.simulation.findRoute(EGO_START_EDGE_ID, dest, vType="egoType")
edges = list(stage.edges) if stage and stage.edges else []
if not edges:
    raise RuntimeError(f"No route to task {first_task.name!r} (edge {dest!r})")
```

The initial route is computed with SUMO's current edge travel times. Since background traffic has not yet built up at this point, this route mainly reflects free-flow conditions; it is then refined by the periodic and event-driven rerouting mechanisms (Chapter 6) once EGO is moving.

If findRoute() fails or returns an empty edge list (no path found), the function raises a RuntimeError with a descriptive message including the task name and edge ID, which allows the user to diagnose network connectivity issues or relocate the task.

Vehicle Insertion

EGO is inserted into the simulation using the traci.vehicle.add() function.

- vehID = 'ego';
- routeID = 'egoRoute': the named route computed in the previous step, registered with traci.route.add()

- typeID = 'egoType': the vehicle type defined for EGO, which specifies the Krauss car-following model with EGO-specific parameters.
- depart = EGO_DEPART = 150 s: delayed departure, so that the first background vehicles are already in the network when EGO enters it.
- departSpeed = EGO_INITIAL_SPEED = 80/3.6 m/s (80 km/h): initial speed at insertion.

Once in the network, EGO (like every other vehicle) is subscribed to seven variables (position, angle, speed, lane position, lane ID, slope and vehicle type), eliminating the need for individual queries. The stop command for the first task is registered separately, so that a combined duration can be passed when the first tasks share the same station.

3.14 Main Control Loop

This section gives an overview of the control loop executed at each simulation step. Several of the mechanisms it invokes are only outlined here and are described in detail in the following chapters: the battery and V2G model in Chapter 4, the scheduler and the combined stop in Chapter 5, and dynamic routing and replanning in Chapter 6.

The main loop uses traci.simulationStep() to advance the simulation one step at a time. Each step executes eight operations in a fixed order. The order is significant because operations that occur later in the sequence can use data produced by earlier operations within the same step.

Step 1: FCD Collection and Per-Edge Speed Measurement

getAllSubscriptionResults() returns a dictionary mapping vehicle IDs to their subscribed variable values. This single call replaces what would otherwise be $O(N_{\text{vehicles}} \times 7)$ individual TraCI queries. The per-vehicle speed values of all non-EGO vehicles on regular (non-internal) lanes are then aggregated by edge to compute speed_by_edge, a dictionary of mean speeds used by the routing cost function; the underlying per-vehicle samples are also passed to the event-driven congestion check.

```
all_subs = traci.vehicle.getAllSubscriptionResults()
speed_samples_by_edge = {}
for vid, data in all_subs.items():
    ...                                          # FCD row written to fcd.csv
    if vid != EGO_ID and lane and not lane.startswith(":"):
        eid = lane_to_edge_id(lane)
        speed_samples_by_edge.setdefault(eid, []).append((pos, speed))

speed_by_edge = {
    eid: sum(s for _, s in samples) / len(samples)
    for eid, samples in speed_samples_by_edge.items()
}
```

In the same loop, the FCD record of each vehicle is written to fcd.csv as a semicolon-delimited row (see Table C.3 in Appendix C for the schema).

Step 2: SoC Drain

While EGO is moving, its odometer reading is obtained with traci.vehicle.getDistance(). The distance traveled since the previous step is converted to an SoC decrement using the constant consumption rate, as described in Chapter 4 (Equations (4.2) and (4.3)). The driving consumption is always subtracted, even if the SoC falls below the partition floor $\text{SoC}_{\text{min,dyn}}$, which only limits V2G discharge; a warning is printed if the SoC falls below the absolute minimum. Every 10 steps (5 s), the current SoC is recorded in state.SOC_HISTORY.

Step 3: Stop-Completion Detection

The stop state of EGO is read at every step with traci.vehicle.getStopState(). The returned value is a bitmask, distinct from the flags passed to setStop(); the bits relevant to this work are:

Table 3.5: SUMO stop state values relevant to this work.

Stop State Value	Meaning
0	Vehicle is moving (no stop active)
1 (bit 0)	Vehicle is stopped
2 (bit 1)	Vehicle is parked off the driving lane
64 (bit 6)	Vehicle is stopped at a charging station
67 (= 1 + 2 + 64)	Vehicle is parked at a charging station (typical EGO state during a V2G stop)

The simulation tracks the previous stop state and detects a stop-completion event when the stop state changes from (stop_state & 1 == 1) to (stop_state & 1 == 0). Upon detecting this transition:

- The V2G energy delivered during the stop is computed, bounded by the energy available above the partition floor, and subtracted from the SoC.
- The task index, next_task_idx, is incremented by 1 + combined_stop_advance.
- combined_stop_advance is reset to 0.
- If there are still tasks, the route is extended to the next task's station and _add_task_stop() is called.
- _try_replan() is invoked if eligible (see Chapter 6).

Step 4: Route Extension during Combined Stop

If the EGO is currently stopped and combined_stop_advance > 0, the system checks whether the current route ends at the combined-stop station edge. If so, the route is extended toward the next destination, starting from the edge on which EGO currently is. This step is described in detail in Chapter 5.

Step 5: Periodic Rerouting

If EGO is moving, _reroute_to_waypoint() is called every EGO_REROUTE_PERIOD = 5 s (every 10 steps), toward the current task or, once all tasks are completed, toward the final destination. This function updates the travel times for each edge with the current traffic data, calls findRoute() and applies the result with setRoute(). Since setRoute() clears all queued stops, _add_task_stop() is immediately re-called to restore the current stop command.

The wall-clock duration of each periodic rerouting call (measured with time.perf_counter) is recorded in state.PERIODIC_REROUTE_LOG for use in the complexity analysis of Section 6.6.

Step 6: Event-Driven Rerouting

Every EGO_EVENT_CHECK_INTERVAL = 2.5 s (every 5 steps), the edges within EGO_EVENT_LOOKAHEAD_DISTANCE (100 meters) of the EGO's current position are scanned for congestion. An edge is considered congested if mean_speed(e) $<$ speed_limit(e) $\times$ EGO_EVENT_CONGESTION_THRESHOLD [EGO_EVENT_CONGESTION_THRESHOLD = 0.50].

If any congested edges are found within the look-ahead window, an immediate reroute is triggered using the same mechanism as for periodic rerouting. All checks (congested or not) are logged to state.EVENT_REROUTE_LOG with the triggered flag and the duration of the whole check, enabling the analysis of the ratio of 'scans that trigger a reroute' to 'total scans'.

Step 7: In-Flight Replanning

Every 120 steps (60 seconds), if EGO is moving and no combined stop is pending, the estimated arrival time at the current task is computed using traci.simulation.findRoute() with the current position. If estimated_arrival > latest_time + REPLAN_DELAY_THRESHOLD_S, _try_replan() is invoked. This mechanism is described in Chapter 6.

Step 8: PID Speed Control

At every step, the speed of the EGO is regulated by applying the EGO speed profile, which determines the target speed based on the current road type. It then calls the control_ego_speed() function to run the PID controller. This mechanism is described in detail in Section 3.16.

3.15 Stop Management: _add_task_stop()

The _add_task_stop() function is responsible for registering the SUMO stop command for the current task. It implements a three-level fallback strategy to handle the variety of edge lengths and lane configurations encountered in the Pavia network.

Level 1: Charging Station Stop (Primary)

```
traci.vehicle.setStop(
    EGO_ID, task.assigned_station_id,
    duration=stop_dur,          # stop_duration_s or combined duration
    flags=_tc.STOP_CHARGING_STATION | _tc.STOP_PARKING,
)
```

When a stopping-place flag such as STOP_CHARGING_STATION is set, SUMO interprets the second argument as the ID of the stopping place (here, the charging station defined in the additional XML file) rather than as an edge ID, and uses the station's own position on the lane.

The STOP_CHARGING_STATION flag (value 32) marks the stop as taking place at a charging station, so that SUMO associates EGO with the corresponding station element. The STOP_PARKING flag (value 1) makes EGO park off the driving lane, so that it does not block the traffic behind it for the whole duration of the stop. The bitwise OR of the two flags evaluates to 33 before the TraCI call. Note that V2G discharging is not handled by SUMO: the energy exchange is computed by the Python layer at the end of the stop.

Level 2: Edge Stop (Fallback if the Charging Station Stop Fails)

```
def _attempt_edge_stop(edge_id, duration):
    edge = edge_by_id[edge_id]
    edge_len, n_lanes = edge.getLength(), edge.getLaneNumber()
    for lane in range(n_lanes):
        for frac in (0.5, 0.25, 0.75):
            pos = min(max(edge_len * frac, 1.0), edge_len - 1.0)
            try:
                traci.vehicle.setStop(EGO_ID, edge_id, pos=pos,
                                      laneIndex=lane, duration=duration)
                return True
            except traci.TraCIException:
                continue
    return False
```

Level 3: Task Edge Fallback (If All Station Edge Attempts Fail)

```
# Fall back to the task's own edge, with the activity duration only
# (no walking component, no V2G)
if task.stop_edge_id and task.edge_id != stop_edge:
    if _attempt_edge_stop(task.edge_id, task.duration):
        return True
print(f"[WARN] setStop failed for {task.name!r}: task skipped")
return False
```

Level 3 is logged as a warning because it means that the EGO will stop directly at the task location without V2G capability. While this preserves schedule correctness (EGO still completes the task), it results in a loss of V2G revenue. If all three levels fail, a warning is printed and the task is skipped.

3.16 PID Speed Controller

The speed of EGO is governed by a proportional-integral-derivative (PID) controller that is implemented in simulation.py. The PID operates in velocity space in incremental form: at each step it computes a speed correction, which is added to the current speed of EGO and applied with traci.vehicle.setSpeed().

3.16.1 Road Type Classification

The target speed depends on the type of road that EGO is currently traveling on. apply_ego_speed_profile() obtains the current lane's edge type from SumoLib and classifies it into one of two categories:

- Normal roads (residential, primary, secondary and unclassified): target = 13.9 m/s (≈ 50 km/h); max = 16.66 m/s (≈ 60 km/h).
- Fast roads (motorway, trunk, motorway_link, trunk_link): target = 27.7 m/s (≈ 100 km/h); max = 30.55 m/s (≈ 110 km/h).

The fast road classification is applied conservatively; only edges that are explicitly tagged as 'motorway' or 'trunk' in OSM qualify. This prevents EGO from driving at 100 km/h on wide urban arterials that happen to be tagged as 'primary'.

Whenever the active profile changes, apply_ego_speed_profile() updates the maximum speed of EGO in SUMO and resets the PID state, so that the integral term accumulated on one road type is not carried over to the next.

3.16.2 PID Equations

$$e_t = v_{\text{target}} - v_{\text{current}} \tag{3.8}$$

$$I_t = \text{clamp}\left(I_{t-1} + e_t\,\Delta t,\ -I_{\max},\ I_{\max}\right) \tag{3.9}$$

$$u_t = K_p\,e_t + K_i\,I_t + K_d\,\frac{e_t - e_{t-1}}{\Delta t} \tag{3.10}$$

$$v_{\text{raw},t} = \text{clamp}\left(v_{\text{current}} + u_t,\ v_{\min},\ v_{\max}\right) \tag{3.11}$$

$$v_{\text{cmd},t} = \text{clamp}\left(v_{\text{raw},t},\ v_{\text{cmd},t-1} - \Delta v_{\text{step}},\ v_{\text{cmd},t-1} + \Delta v_{\text{step}}\right) \tag{3.12}$$

Parameters: $K_p = 0.07$, $K_i = 0.03$, $K_d = 0.01$, $\Delta t = \text{STEP_LENGTH} = 0.5$ s, $I_{\max} = 30.55$ m/s, $v_{\min} = 0$ m/s, $\Delta v_{\text{step}} = 0.5$ m/s per step; $v_{\max}$ is the maximum speed of the active road profile.

The integral term is clamped to $\pm I_{\max}$ in Equation (3.9) to prevent integrator wind-up, which is a common issue when the PID output saturates for extended periods (e.g. when EGO is stopped at a red light and the error is large). Equation (3.12) acts as a rate limiter: the commanded speed can change by at most 0.5 m/s per step (1 m/s^2), which avoids abrupt speed jumps and produces smooth acceleration profiles.

3.16.3 Following-Distance Override

If a leading vehicle is detected within EGO_FOLLOWING_LOOKAHEAD = 100 m and the gap to it is smaller than EGO_FOLLOWING_DISTANCE = 20 m, the following-distance controller overrides the PID output. The commanded speed is scaled in proportion to the gap: when the gap equals the target distance, EGO matches the speed of the leader; as the gap shrinks toward zero, the commanded speed decreases linearly to zero. This allows EGO to naturally reopen the target gap and prevents the unrealistic, jerky behavior that could otherwise arise when SUMO's car-following model interacts with the speed commands issued by the PID.

```
def get_following_distance_speed(vehicle_id):
    leader_info = traci.vehicle.getLeader(vehicle_id, EGO_FOLLOWING_LOOKAHEAD)
    if leader_info is None:
        return None                                # no leader in range
    leader_id, gap = leader_info
    gap = max(gap, 0.0)
    if gap >= EGO_FOLLOWING_DISTANCE:
        return None                                # enough room, no override
    leader_speed = traci.vehicle.getSpeed(leader_id)
    safety_ratio = gap / EGO_FOLLOWING_DISTANCE  # 0 ... 1
    return max(leader_speed * safety_ratio, 0.0)
```

3.17 Floating Car Data Collection

At every simulation step, the FCD record of each active vehicle is written to fcd.csv, using the values already returned by the subscription mechanism. The file uses semicolons as delimiters to avoid conflicts with comma-containing field values. The schema of the file is reported in Table C.3 in Appendix C.

Chapter 4

Battery and V2G Model

4.1 Overview and Modeling Philosophy

The battery model in this thesis is a first-order energy-counting model. It does not simulate electrochemical dynamics, such as voltage curves, internal resistance and thermal effects, nor does it simulate battery degradation. Instead, it tracks the state of charge as a dimensionless scalar between 0 and 1. This design choice prioritizes simplicity and computational speed over physical accuracy, which is appropriate for a planning and routing application, where the primary decision variables (task order and route selection) are far more uncertain than the battery parameters themselves.

The model is adequate for the timescales involved (single-day operation and a total driving distance of typically 10–20 km in the scenarios of Chapter 7) because, at this scale, the dominant source of uncertainty is traffic, which the simulation models explicitly, rather than battery physics.

4.2 Battery Parameters

Table 4.1 lists all battery-related parameters defined in config.py.

Table 4.1: Battery and V2G model parameters.

Parameter	**Value**	**Unit**	**Description**
BATTERY_CAPACITY_KWH	60.0	kWh	Total usable battery capacity. Reference value for a typical mid-size EV.
BATTERY_INITIAL_SOC	1.00	fraction	State of charge at simulation start (100%).
BATTERY_MIN_SOC	0.20	fraction	Absolute lower bound, treated as a hard limit for V2G discharge (conservative design choice).
BATTERY_CONSUMPTION_KWH_PER_KM	0.20	kWh/km	Energy consumption at urban speeds (50 km/h).
BATTERY_DRIVE_SAFETY_MARGIN	0.30	fraction	Reserve margin added to driving energy estimate.

Table 4.1 (continued)

Parameter	Value	Unit	Description
V2G_FEED_IN_TARIFF_EUR_KWH	0.10	€/kWh	V2G remuneration per kWh delivered (the parameter name is historical: no feed-in tariff is assumed, see Section 2.1.3). Derived as ELECTRICITY_PRICE_EUR_KWH × V2G_FEED_IN_TARIFF_PCT_OF_PRICE, not a flat constant.
EGO_ENERGY_ROUTING_LAMBDA	50.0	s/kWh	Converts energy cost to equivalent travel time for routing.
ELECTRICITY_PRICE_EUR_KWH	0.50	€/kWh	Reference average retail price for recharging at a station.
V2G_FEED_IN_TARIFF_PCT_OF_PRICE	0.20	fraction	V2G remuneration as a fraction of ELECTRICITY_PRICE_EUR_KWH, chosen so that the V2G revenue remains a realistic fraction of the energy price.

4.3 Driving Energy Consumption

The traci.vehicle.getDistance() function provides, at each simulation step, the odometer reading of the EGO vehicle, from which the incremental distance driven since the previous step is computed.

$$\Delta d_{\text{km}} = \max(0, \text{dist}_t - \text{dist}_{t-1})/1000.0 \tag{4.1}$$

where the division by 1000 converts the distance returned by TraCI from meters to kilometers.

The $\max(0, \dots)$ guard prevents negative increments caused by SUMO's occasional trajectory corrections. The energy drained from the battery is:

$$\Delta\text{SoC}_{\text{drain}} = \Delta d_{\text{km}} \times e_{\text{cons}}/C_{\text{bat}} \tag{4.2}$$

where $e_{\text{cons}} = 0.20$ kWh/km and $C_{\text{bat}} = 60.0$ kWh.

$$\text{SoC}_t = \max(0, \text{SoC}_{t-1} - \Delta\text{SoC}_{\text{drain}}) \tag{4.3}$$

The driving consumption is always subtracted from the SoC, even if this brings it below the partition floor $\text{SoC}_{\text{min,dyn}}$ computed by the scheduler (Equation (4.6)): the floor limits V2G discharge, not driving. Under normal operation the floor is not reached while driving, because the driving reserve is placed on top of the absolute minimum and includes a 30% safety margin. If the actual driving energy exceeds the reserve (e.g. because road distances are much longer than the Euclidean estimate, or because of a long detour), the SoC falls below the floor and, in the worst case, below the absolute minimum; in that case a warning is printed in the console log. An earlier version of the model clamped the SoC at the partition floor while driving, which hid any energy deficit; this was corrected during testing (Chapter 7).

The SoC is recorded in state.SOC_HISTORY every 10 simulation steps (every 5 seconds) and plotted at the end of the simulation, together with the partition floor. Figure 7.9 in Chapter 7 shows a representative SoC history: driving appears as a slowly decreasing slope, stops as flat segments, and V2G discharges as downward steps at the end of each stop, since the V2G energy is accounted for at stop completion.

4.4 Battery Partition Model

4.4.1 Motivation

A naïve V2G strategy would discharge as much energy as possible at each stop, disregarding the energy required for subsequent driving. This approach risks leaving the electric vehicle (EV) with insufficient battery charge to reach the next destination. Conversely, if the battery is never discharged (i.e. if the SoC is kept at 100% throughout), the entire V2G opportunity is wasted. The battery partition model defines a dynamic minimum SoC threshold that reserves the estimated energy required for driving on top of the absolute minimum state of charge, leaving the surplus available for V2G. This design is conceptually similar to the virtual battery partitioning approaches that have been proposed for V2G-capable vehicles [33]. These approaches also separate a physical battery into software-managed mobility and grid-service shares, but do not require any physical modification of the battery pack.

4.4.2 Driving Energy Estimation

The total driving energy for the planned itinerary is estimated as:

$$E_{\text{drive}} = \sum_i d_i[\text{km}] \times e_{\text{cons}} \quad [\text{kWh/km}] \tag{4.4}$$

where d_i is the Euclidean distance of segment i and $e_{\text{cons}} = 0.20$ kWh/km.

A 30% safety margin is added to account for routing detours, elevation changes, and estimation errors from using Euclidean rather than road distances:

$$E_{\text{reserve}} = E_{\text{drive}} \times (1 + \text{margin}) = E_{\text{drive}} \times 1.30 \tag{4.5}$$

As shown in Chapter 7 (Scenario 3), road distances in the Pavia network can be about twice the Euclidean ones, so that this margin does not fully compensate for the use of Euclidean distances; in that case, the absolute minimum SoC acts as the ultimate safeguard.

4.4.3 Dynamic Minimum SoC

The dynamic minimum SoC is computed by adding the driving reserve, expressed as a fraction of the battery capacity, to the absolute minimum state of charge, and capping the result below the initial SoC:

$$\text{SoC}_{\text{min,dyn}} = \min\left(\text{SoC}_{\text{abs,min}} + \frac{E_{\text{reserve}}}{C_{\text{bat}}},\ \text{SoC}_0 - \delta\right) \tag{4.6}$$

The parameters of this expression require explicit explanation:

- $C_{\text{bat}} = 60.0$ kWh: the total usable battery capacity. This is the denominator that converts an energy amount (kWh) to a state of charge (SoC) fraction (0–1). This value appears in the code as 'BATTERY_CAPACITY_KWH' and represents a typical battery for a mid-size EV.
- $\text{SoC}_{\text{abs,min}} = 0.20$: the absolute minimum state of charge. This is a conservative design choice: keeping a minimum charge preserves a range reserve for unforeseen trips and limits the battery wear associated with deep discharges. In this work, it is treated as a hard limit that V2G discharge must never cross. The driving reserve is added on top of it, so that the energy needed for driving is always available above this limit.

- $\mathrm{SoC}_0 - \delta = 1.00 - 0.05 = 0.95$: the upper bound. Since EGO starts at SoC = 1.00 and the minimum SoC cannot exceed the initial SoC, this term prevents the result of $\mathrm{SoC}_{\mathrm{min,dyn}} > \mathrm{SoC}_0$. The 5% headroom ($\delta = 0.05$) ensures at least 3 kWh of V2G capacity, even when the driving reserve is close to the battery capacity.

An earlier version of the model computed the floor as the maximum, rather than the sum, of the absolute minimum and the driving reserve. For short urban routes, the driving reserve was then entirely absorbed by the absolute minimum, allowing V2G to discharge the battery down to 20% and leaving no energy for the remaining driving. This issue was identified during testing (Chapter 7) and corrected with the additive formulation above.

The partition floor limits only V2G discharge: during the simulation the driving consumption is always subtracted from the SoC, even if this brings it below the floor, so that the SoC always reflects the energy actually consumed.

4.4.4 Worked Numerical Example

Consider the following three-task itinerary, with driving distances as shown:

- Depot–CS01: 2.0 km
- CS01–CS02: 3.5 km
- CS02–CS03: 2.8 km
- CS03–Home: 1.5 km

Total distance = 9.8 km.

$$\begin{aligned} E_{\mathrm{drive}} &= 9.8 \times 0.20 = 1.96\,\mathrm{kWh} \\ E_{\mathrm{reserve}} &= 1.96 \times 1.30 = 2.548\,\mathrm{kWh} \\ E_{\mathrm{reserve}}/C_{\mathrm{bat}} &= 2.548/60.0 = 0.0425 \\ \mathrm{SoC}_{\mathrm{min,dyn}} &= \min(0.20 + 0.0425,\ 0.95) = 0.2425 \end{aligned}$$

$$E_{\mathrm{V2G,available}} = (1.00 - 0.2425) \times 60.0 = 45.45\,\mathrm{kWh}$$

In this example, the driving requirement is small: the floor rises only slightly above the absolute minimum, leaving 45.45 kWh available for V2G, while 2.55 kWh remain reserved for driving above the 20% absolute minimum. This is close to the best-case scenario for V2G revenue.

For a longer route of 100 km in total:

$$\begin{aligned} E_{\mathrm{drive}} &= 100 \times 0.20 = 20.0\,\mathrm{kWh} \\ E_{\mathrm{reserve}} &= 20.0 \times 1.30 = 26.0\,\mathrm{kWh} \\ E_{\mathrm{reserve}}/C_{\mathrm{bat}} &= 26.0/60.0 = 0.433 \\ \mathrm{SoC}_{\mathrm{min,dyn}} &= \min(0.20 + 0.433,\ 0.95) = 0.633 \\ E_{\mathrm{V2G,available}} &= (1.00 - 0.633) \times 60.0 = 22.0\,\mathrm{kWh} \end{aligned}$$

In this case, 63.3% of the battery is reserved (20% as the absolute minimum and 43.3% for driving), leaving 22.0 kWh for V2G. With a 20% safety margin, the same route would leave 24.0 kWh available; the 30% margin adopted here therefore trades 2.0 kWh of V2G capacity for a larger buffer against routing and estimation errors. The system correctly limits V2G discharge to prevent range anxiety. Figure 4.1 compares the partition of the battery in the two examples.

For very long routes, whose driving reserve exceeds the energy available above the absolute minimum, the upper bound $\mathrm{SoC}_0 - \delta$ becomes active and the reserve can no longer be fully guaranteed: such itineraries would require an intermediate recharge, which is outside the scope of the current model.

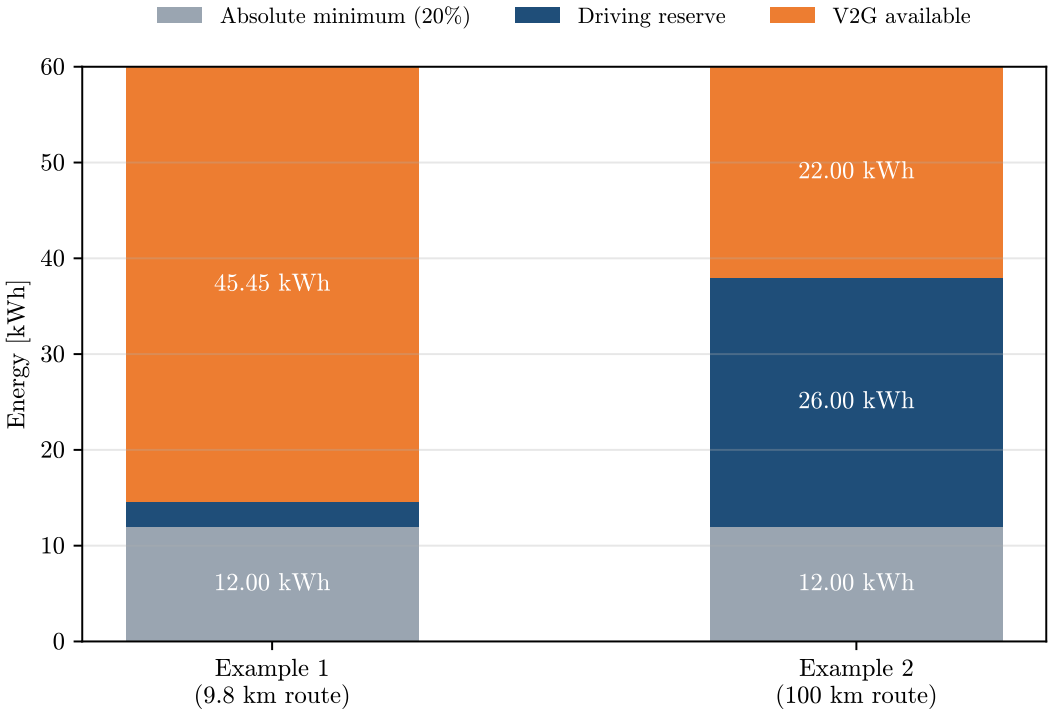


Figure 4.1: Battery partition for the two worked examples (60 kWh total capacity): absolute minimum, driving reserve and energy available for V2G.

4.5 V2G Energy Model

4.5.1 Energy Calculation

A V2G event occurs each time EGO completes a stop at a charging station. The energy available for discharge at that stop is the minimum of two quantities:

First, the raw energy that the station could theoretically deliver at its nominal power over the stop duration:

$$E_{\mathrm{raw}} = \mathrm{power_kw}(S^*) \times \mathrm{stop_duration_s}(T_i)/3600.0 \quad [\mathrm{kWh}] \tag{4.7}$$

where the division by 3600 converts kW·s into kWh.

Second, the energy that can be discharged without violating the minimum SoC floor:

$$E_{\mathrm{avail}} = \max(0, (\mathrm{SoC}_{\mathrm{before}} - \mathrm{SoC}_{\mathrm{min,dyn}}) \times C_{\mathrm{bat}}) \quad [\mathrm{kWh}] \tag{4.8}$$

The actual V2G energy discharged is the minimum of the two:

$$E_{\mathrm{V2G}} = \min(E_{\mathrm{raw}}, E_{\mathrm{avail}}) \quad [\mathrm{kWh}] \tag{4.9}$$

The SoC is then updated:

$$\mathrm{SoC}_{\mathrm{after}} = \mathrm{SoC}_{\mathrm{before}} - E_{\mathrm{V2G}}/C_{\mathrm{bat}} \tag{4.10}$$

Since $E_{\text{V2G}} \leq E_{\text{avail}}$ by Equation (4.9), V2G discharge can never bring the SoC below the partition floor, and no further clamping is needed. In particular, if the SoC is already below the floor when a stop begins (for instance because driving consumed more energy than estimated), $E_{\text{avail}} = 0$ and the SoC is left unchanged. An earlier version of the model applied an additional clamp at the floor after each stop, which raised the SoC back to the floor in this situation; this was corrected during testing.

4.5.2 Worked Numerical Example

Consider a stop at a 22 kW station with stop_duration_s = 2400 s (40 minutes), $\text{SoC}_{\text{before}} = 0.75$, and $\text{SoC}_{\text{min,dyn}} = 0.30$.

$$\begin{aligned} E_{\text{raw}} &= 22.0 \times 2400/3600 = 14.67\,\text{kWh} \\ E_{\text{avail}} &= (0.75 - 0.30) \times 60.0 = 27.0\,\text{kWh} \\ E_{\text{V2G}} &= \min(14.67, 27.0) = 14.67\,\text{kWh} \\ \text{revenue} &= 14.67 \times 0.10 = 1.47\,\text{EUR} \\ \text{SoC}_{\text{after}} &= 0.75 - 14.67/60.0 = 0.75 - 0.244 = 0.506 \end{aligned}$$

In this case, it is the station's power output that limits the V2G, rather than the available capacity of the battery. The driver earns €1.47 from this stop. Over a full day, however, the battery soon becomes the binding constraint: a second identical stop would already be limited to $E_{\text{avail}} = (0.506 - 0.30) \times 60.0 = 12.4$ kWh (€1.24), bringing the SoC down to the floor, and a third stop would deliver no energy at all. The daily revenue in this example is therefore about €2.71, consistent with a V2G tariff set at 20% of the €0.50/kWh reference recharge price. The same behavior was observed in Scenario 3 of Chapter 7, where the first stop discharged the battery down to the floor and the following stops delivered no energy.

4.5.3 Relationship Between Scheduler Estimate and Simulation Result

The scheduler and the simulation compute the V2G energy with the same formulas (Equations (5.8)–(5.10) and (4.7)–(4.9)), but they can give different results for three reasons.

First, the charging time. The simulation uses stop_duration_s (which reduces to 2× walk_time_s + duration for a standalone stop and follows the chain-aware form of Section 5.5.3 for combined stops), since the same value is passed to SUMO as the stop duration. The scheduler, instead, also counts as charging time any waiting at the station before the task window opens, for example when EGO is planned to arrive early for a rigid task. The simulation does not model this waiting: EGO starts the activity as soon as it arrives. In Scenario 2 of Chapter 7, this accounts for a difference of 7.88 kWh (33.03 kWh estimated against 25.15 kWh simulated).

Second, the state of charge at each stop. The scheduler estimates it from Euclidean distances, whereas the simulation uses the actual distance driven on the road network, which in the Pavia network can be about twice the Euclidean one.

Third, the floor. The scheduler evaluates each order using the absolute floor $\text{SoC}_{\text{abs,min}}$, because the partition floor is computed only after the best order has been selected, whereas the simulation uses the partition floor $\text{SoC}_{\text{min,dyn}}$. When the battery becomes the binding constraint, as in Scenario 3, the simulated V2G energy is therefore lower than the estimate (11.62 kWh against 14.29 kWh).

When no waiting is required and the battery never approaches the floor, as in Scenario 1, the two values coincide. The scheduler's estimate is used for the battery partition calculation and for display, both in the confirmation dialog and in the end-of-day summary window, which reports the planned V2G energy and revenue together with the final SoC measured in the simulation.

The energy actually delivered at each stop is computed by the simulation at stop completion and reported in the console log; the corresponding revenue follows by multiplying it by the tariff.

Chapter 5

Task Optimization

The scheduler module (scheduler.py) forms the core of the platform's planning functionality. Taking the user-defined task list and available charging stations as input, it produces an optimized visit order together with battery partition parameters to guide V2G behavior during simulation. The scheduler operates entirely offline, using Euclidean distances and constant average speeds as proxies for actual road travel times, rather than starting a SUMO simulation.

5.1 Station Assignment

5.1.1 Greedy Nearest Assignment

The first step in the scheduling pipeline involves assigning a charging station to each task. The assignment criterion is the minimum Euclidean distance:

$$\text{station}^*(T_i) = \arg\min_{S_j \in S} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{5.1}$$

This greedy, per-task assignment has a complexity of O(N × M) and runs in microseconds for typical values of N and M. While it does not optimize the station-to-task mapping globally (which would require solving an assignment problem in $O(N^3)$ using the Hungarian algorithm [34]), the simpler approach is adequate as the task ordering optimization already accounts for travel times between stations.

If no charging stations are configured (M = 0), the assignment step is skipped and all tasks receive station_id = None, stop_edge_id = None and walk_time_s = 0.0. In this case, EGO stops directly at the task's nearest edge and V2G revenue is zero.

5.1.2 Walking Time Estimation

After assigning a station to each task, the walking time between the station and the task location is estimated as follows:

$$\text{walk_time_s}(T_i, S_j) = d(T_i, S_j)/v_{\text{walk}} \qquad [v_{\text{walk}} = 1.39\,\text{m/s} = 5.0\,\text{km/h}] \tag{5.2}$$

The configuration defines a reference maximum walking distance, MAX_WALK_DISTANCE_M = 800 m, corresponding to a one-way walk of approximately 576 seconds (9.6 minutes). In the current implementation, however, this threshold is not enforced: the scheduler always assigns the nearest station, regardless of distance. For tasks located far from every station, the walking component can therefore dominate the stop duration. Enforcing the threshold, for example by reverting to direct edge stopping, is a straightforward extension.

5.2 Schedule Evaluation: _evaluate_order()

At the heart of the scheduler lies the _evaluate_order() function, which simulates a complete itinerary for a given permutation of tasks and returns a ScheduleResult containing timing, energy and quality metrics. Understanding this function is essential to understanding why the scheduler produces the results it does.

5.2.1 Travel Time Estimation

Inter-task travel times are estimated using Euclidean distance and a constant speed.

$$t_{\text{drive}}(P_1, P_2) = d_{\text{Euclid}}(P_1, P_2)/v_{\text{avg}} \tag{5.3}$$

where $v_{\text{avg}} = 8.33$ m/s (30 km/h) is an average speed used only for scheduling.

The 30 km/h value is a planning constant: it is not the speed at which EGO drives during the SUMO simulation. During the simulation, the EGO vehicle is controlled by a PID governor at 50 km/h on urban roads and 100 km/h on motorways (see Chapter 3). The 30 km/h planning value is deliberately lower than the speeds used in the simulation, in order to partially compensate for congestion and for routes being longer than the straight-line distance.

Nevertheless, the resulting arrival time estimates remain optimistic: real routes can be considerably longer than the Euclidean distance (about twice as long in the scenarios of Chapter 7), and traffic adds further delay. This is acceptable for the planning phase, as the replanning mechanism (Chapter 6) corrects any resulting delays during the simulation.

5.2.2 Task Start Time Computation

For each task in the sequence, the evaluator computes the time at which EGO arrives at the charging station (arrive_at_stop) and the time at which the activity begins (task_start). The computation differs by priority:

For priority-1 (rigid) tasks:

$$\text{arrive_at_stop} = \text{current_time} + t_{\text{drive}}(\text{prev_stop}, \text{station}^*(T_i)) \tag{5.4}$$

$$\text{task_start} = \max(\text{target_time}(T_i), \text{arrive_at_stop} + \text{walk_time_s}(T_i)) \tag{5.5}$$

The max() function ensures that, if EGO arrives early, it is planned to wait at the station until the exact target time. Since the flex_window is 0 for P1 tasks, arriving later than the target time is considered a failure (on_time is False).

For priority 2–5 (flexible) tasks:

$$\text{task_start} = \max(\text{arrive_at_stop} + \text{walk_time_s}(T_i), \text{earliest_time}(T_i)) \tag{5.6}$$

The max() function with the earliest_time parameter prevents EGO from starting the task before the window opens, even if it arrives early. If task_start $\leq$ latest_time(T_i), the task is considered on time.

The waiting time before the task window is spent parked at the station and is therefore counted by the evaluator as additional V2G discharge time. This waiting is modeled only in the scheduler: in the SUMO simulation each stop lasts exactly stop_duration_s, so the V2G energy associated with waiting is not reproduced (see Chapter 7).

5.2.3 Same-Station Consecutive Detection in the Evaluator

The evaluator contains a special case for consecutive tasks at the same station. When tasks T_i and T_{i+1} are assigned to the same station S*, the evaluator sets arrive_at_stop for T_{i+1} to current_time (i.e. zero driving time) since EGO is already parked at S*. Importantly, the walking distance for T_{i+1} is no longer measured from the station; it is measured directly from the location of T_i to the location of T_{i+1}, because the driver walks straight from one task to the next instead of returning to the car in between.

The evaluator determines this by looking one task ahead in the visit order. If T_{i+1} shares T_i's station, then T_i's departure carries no walk-back cost yet, as the driver is already effectively en route to T_{i+1}. Only the last task of a same-station chain incurs the walk back to the car. The departure time after completing T_i is therefore:

$$\begin{aligned} \text{depart}(T_i) &= \text{task_start}(T_i) + \text{duration}(T_i) + \text{walk_out}(T_i) \\ &= \text{task_start}(T_i) + \text{stop_duration_s}(T_i) \end{aligned} \tag{5.7}$$

where $\text{walk_out}(T_i) = 0$ if T_{i+1} continues the chain, otherwise it is the walk from T_i back to the station; the second form holds because stop_duration_s already includes the correct walk-out.

The next task's station arrival uses the departure time of Equation (5.7) as the new current time. This chain-aware bookkeeping allows the combined-stop duration (Section 5.5.3) to be computed by simply summing the stop durations over the chain.

5.2.4 V2G Energy Calculation per Stop

For each stop, the evaluator computes the V2G energy and revenue.

$$E_{\text{raw}}(T_i) = \text{power_kw}(S^*) \times t_{\text{charge}}(T_i)/3600 \quad [\text{kWh}] \tag{5.8}$$

$$E_{\text{avail}}(T_i) = \max(0, (\text{SoC}_{\text{before}} - \text{SoC}_{\text{abs,min}}) \times C_{\text{bat}}) \quad [\text{kWh}] \tag{5.9}$$

$$E_{\text{V2G}}(T_i) = \min(E_{\text{raw}}, E_{\text{avail}}) \quad [\text{kWh}] \tag{5.10}$$

$$\text{revenue}(T_i) = E_{\text{V2G}}(T_i) \times \tau_{\text{V2G}} \quad [\text{EUR}] \tag{5.11}$$

where $t_{\text{charge}}(T_i)$ is the time spent parked at the station, i.e. stop_duration_s plus any waiting time before the task window. The SoC before is estimated by subtracting the driving energy for all preceding segments from the initial battery state of charge (SoC), using the constant consumption rate (e_{cons}). At this stage the absolute floor $\text{SoC}_{\text{abs,min}}$ is used, because the dynamic minimum SoC (Equation (4.6)) depends on the total driving energy of the itinerary and is therefore computed only once the best order has been selected.

5.3 Optimization Strategies

5.3.1 Brute-Force Search for $N \leq 8$

When N is at most 8, the scheduler uses itertools.permutations to enumerate all N! permutations and evaluates each one with _evaluate_order(). The best permutation is selected according to a lexicographic objective.

min (−tasks_on_time, +total_positive_delay)

where total_positive_delay is the sum of max(0, task_start(T_i) − target_time(T_i)) for all tasks. This two-level objective first maximizes the number of tasks completed within their flexibility window and then minimizes the total tardiness of tasks outside the window.

Table 5.1: Permutation counts by task number. Rows 1–8 use an exact search, while rows 9–10 use a heuristic.

N (tasks)	N! permutations evaluated
1	1
2	2
3	6
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880 → greedy+2-opt used
10	3,628,800 → greedy+2-opt used

5.3.2 Greedy Initialization for $N > 8$

For larger instances, the _greedy_order() function builds an initial solution using an earliest-deadline-first (EDF) strategy.

```
def _greedy_order(tasks, sx, sy):
    """Earliest-deadline-first greedy order (good starting point for 2-opt)."""
    remaining = list(tasks)
    order = []
    cx, cy = sx, sy
    while remaining:
        remaining.sort(key=lambda t: (t.latest_time, math.hypot(t.x - cx, t.y - cy)))
        chosen = remaining.pop(0)
        order.append(chosen)
        cx, cy = chosen.x, chosen.y
    return order
```

EDF is known to be optimal for preemptive single-machine scheduling [35] and provides a good starting point for the 2-opt refinement process. Distance tie-breaking ensures that, among tasks with equal deadlines, the one closest to the previously selected task is preferred, thereby reducing unnecessary backtracking.

5.3.3 2-opt Local Search

The _two_opt_improve() function applies the 2-opt operator to the greedy solution. The operator selects two indices, i and j, and reverses the order of the sub-sequence [i:j], keeping the result if it improves the lexicographic objective (more tasks on time or, at equal count, a smaller total delay). This process is repeated in an outer loop until no further improvements are found.

```
def _two_opt_improve(order, assignments, sx, sy, depart,
                     best_entries, best_delay, best_on_time):
    """Standard 2-opt local search."""
    improved = True
    while improved:
        improved = False
        n = len(order)
        for i in range(n - 1):
            for j in range(i + 2, n):
                candidate = order[:i] + list(reversed(order[i:j])) + order[j:]
                entries, delay, on_time = _evaluate_order(
                    candidate, assignments, sx, sy, depart)
                if on_time > best_on_time or (
                        on_time == best_on_time and delay < best_delay):
                    order = candidate
                    best_entries, best_delay, best_on_time = entries, delay, on_time
                    improved = True
    return order, best_entries, best_delay, best_on_time
```

The complexity of one 2-opt sweep is $O(N^2)$ candidate evaluations, each of which costs O(N) to evaluate, giving a total of $O(N^3)$ per sweep. Starting from the greedy solution, the local search is expected to converge in a few sweeps; assuming, for instance, four sweeps, the total cost for N = 15 would be of the order of $4N^3 = 13{,}500$ evaluations, comparable to the brute-force cost at N = 8 and well within the interactive time budget. The number of sweeps has not been measured, since the scenarios of Chapter 7 involve only three tasks.

5.4 Output: ScheduleResult and ScheduleEntry

The scheduler returns a ScheduleResult data class containing:

- order: the Task objects in optimized visit order.
- entries: a list of ScheduleEntry objects, one per task, with a detailed timing and energy breakdown.
- total_delay_s: the total positive delay with respect to the target times [s].
- tasks_on_time: the number of tasks completed within their flexibility window.
- suggestions: a list of human-readable strings describing the schedule decisions.
- total_energy_kwh: the total V2G energy for the itinerary [kWh].
- total_discount_eur: the total estimated V2G revenue [EUR] (named “discount” in the code for historical reasons).
- total_duration_s: the total itinerary time [s].
- soc_final: the predicted state of charge at the end of the itinerary.
- total_drive_kwh: the total driving energy [kWh].
- dynamic_min_soc: the computed drive-reserve SoC floor.
- v2g_available_kwh: the energy above dynamic_min_soc available for V2G [kWh].

The suggestions are generated by _build_suggestions() after optimization is complete. They describe the optimizer’s decisions (e.g. ‘Task T2 moved to position 1 to avoid missing the 09:00 deadline’), but do not modify any task attribute themselves. They are purely informational text displayed in the confirmation dialog.

5.5 The Combined Stop: Problem and Solution

5.5.1 Problem Statement

The scheduler independently assigns each task to the nearest charging station. If two or more consecutive tasks are geographically clustered around the same station, they receive the same station assignment. In the Pavia network, this scenario naturally arises for tasks within 400 meters of each other when fewer than three stations are deployed, which is a typical configuration for pilot testing.

In this scenario, the simulation must keep the EGO vehicle parked at the station for the combined duration of all consecutive tasks at the same station, without driving away and returning. The most straightforward implementation (issuing a separate setStop() command for each task) fails silently in SUMO, causing subsequent tasks to be skipped without any visible error to the simulation operator.

5.5.2 Root Cause Analysis

SUMO Stop Validation Rules

SUMO validates each setStop() command against three geometric rules.

1. The stop must be on a lane that is part of the vehicle's current route.
2. The stop position must be strictly downstream of the vehicle's current lane position.
3. The stop must not overlap with an existing queued stop on the same lane.

Rule 2 is the source of the failure. When EGO is stopped at a charging station (CS), its lane position is equal to the station's endPos, i.e. the downstream boundary of the 20 m stopping zone. A second setStop() command for the same station uses the same endPos value, which is not strictly greater than the vehicle's current lane position.

$$\text{Required:} \quad \text{new_stop_pos} > \text{pos_EGO} = \text{CS.endPos} \tag{5.12}$$

$$\text{Actual:} \quad \text{new_stop_pos} = \text{CS.endPos} = \text{pos_EGO} \tag{5.13}$$

Result: new_stop_pos > pos_EGO is False → TraCIException

Note that this is not a bug in SUMO: it correctly enforces the physical constraint that a stop must be reachable from the vehicle's current position by driving forward. This constraint cannot be bypassed by adjusting the stop position because the station's endPos is fixed in the XML definition and cannot be moved to a downstream position on the same edge.

The Silent Failure Mode

The failure manifests as a TraCIException with the message 'stop not downstream'. In the original implementation, this exception was caught by a broad except clause and logged as a warning, but execution continued normally. This meant that the second task's stop was never registered and, when the first stop was completed, EGO immediately began routing to the third task, silently skipping the second.

This silent failure is particularly insidious because the symptom (EGO visiting fewer tasks than expected) could be attributed to several causes, such as missed deadlines, unreachable stations and route failures. The bug was identified by adding a systematic check after every setStop() call to verify that the stop had been queued and by comparing the actual and expected stop counts at the end of each simulation run.

Alternative Solutions Considered and Rejected

Three alternative solutions were considered before adopting the combined stop approach:

- Alternative 1 (rejected): Offset the second stop position to endPos + 1.0 m. This is geometrically infeasible because endPos is already at the edge boundary. Adding 1 m would place the stop outside the edge.
- Alternative 2 (rejected): Insert a short driving segment between the two stops by routing EGO around the block. While this would produce correct timing, it is unrealistic (the driver's car would drive around the block while the driver is inside a building) and would complicate the V2G accounting.
- Alternative 3 (rejected): Reschedule the conflicting task to a different station. This changes the scheduling result for topological reasons and may significantly worsen the schedule quality.

The combined stop approach was adopted because it is geometrically sound (one stop and one setStop() call), preserves V2G revenue exactly (analytical processing produces the same result as a real stop) and requires no changes to the task or station data model.

5.5.3 The Combined Stop Solution

Chain Detection

The _same_station_chain() function computes the number of tasks that immediately follow the task with the given from_idx and that share the same assigned station.

```
def _same_station_chain(tasks, from_idx):
    """Count consecutive tasks after from_idx that share the same assigned station."""
    if from_idx >= len(tasks):
        return 0
    base = tasks[from_idx].assigned_station_id
    if not base:
        return 0
    count = 0
    i = from_idx + 1
    while i < len(tasks) and tasks[i].assigned_station_id == base:
        count += 1
        i += 1
    return count
```

Examples:

- [CS01, CS01, CS02]: chain at index 0 = 1 (T0 and T1 share CS01).
- [CS01, CS01, CS01, CS02]: chain at index 0 = 2 (T0, T1 and T2 share CS01).
- [CS01, CS02, CS01]: chain at index 0 = 0 (T0 and T2 share CS01 but are not consecutive).
- [CS01]: chain at index 0 = 0 (no successor).

The function returns 0 in all non-combined-stop cases, rendering the entire combined-stop logic ineffective when the condition does not apply. This is the key property that ensures backward compatibility with scenarios without consecutive tasks at the same station.

Combined Duration Computation

At the time of injection, if the value of combined_stop_advance is greater than zero, the combined duration is computed by adding together the stop durations of all bundled tasks.

$$d_{\text{combined}} = \sum_{k=0}^{m} \text{stop_duration_s}(T_{n+k}) \tag{5.14}$$

where m = combined_stop_advance is the number of tasks bundled with T_n.

The accuracy of this sum depends on how the evaluator assigned walk_time_s and walk_out_time_s to each task (see Section 5.2.3). For every task except the last one in the chain, walk_out_time_s = 0 because the driver walks directly from one task to the next without returning to the car in between. Only the last task (T_{n+m}) has a non-zero walk_out_time_s value, namely the time taken to walk from its location back to the station. Summing stop_duration_s therefore reproduces the real-world sequence exactly: walk from the station to T_n, complete T_n, walk directly to T_{n+1}, complete T_{n+1}, and so on until T_{n+m}, then walk back to the station.

Consider a chain of two tasks, T1 (duration 1200 seconds, i.e. 20 minutes) and T2 (duration 900 seconds, i.e. 15 minutes), assigned to the same station. T1 is located 100 meters from the station and T2 is located 250 meters from the station. There are 200 meters separating T1 and T2. At a walking speed of 1.39 m/s (5 km/h):

Since the chain continues from T1 to T2, T1 has no walk back to the car, whereas T2, the last task of the chain, includes the walk back to the station:

$$\text{walk_in}(T_1) = 100/1.39 = 71.9\,\text{s}, \qquad \text{walk_out}(T_1) = 0\,\text{s} \tag{5.15}$$

$$\text{walk_in}(T_2) = 200/1.39 = 143.9\,\text{s}, \qquad \text{walk_out}(T_2) = 250/1.39 = 179.9\,\text{s} \tag{5.16}$$

$$\begin{aligned} \text{stop_duration_s}(T_1) &= 71.9 + 1200 + 0 = 1271.9\,\text{s} \\ \text{stop_duration_s}(T_2) &= 143.9 + 900 + 179.9 = 1223.8\,\text{s} \\ d_{\text{combined}} &= 1271.9 + 1223.8 = 2495.7\,\text{s} \approx 2496\,\text{s} \quad (41.6 \text{ minutes}) \end{aligned}$$

An earlier version of this computation measured the walk_time_s for each task as the one-way distance from the station and doubled it, effectively assuming that the driver returns to the car between T1 and T2 before walking back out again. This approach would give stop duration (T1) as $2 \times 71.9 + 1200 = 1343.8$ s and stop duration (T2) as $2 \times 179.9 + 900 = 1259.8$ s, totalling 2603.6 s, i.e. 107.9 s longer than the correct value. This overestimation increases with the distance between consecutive tasks. According to the triangle inequality, the direct walk from T1 to T2 is never longer than walking back to the station and out again.

This aggregated duration, computed with Equation (5.14), is passed to _add_task_stop() via the duration_override parameter. The function then issues a single setStop() command for T_n with a duration of d_{combined}. SUMO keeps the EGO parked at the station for the correct total time. When the combined stop ends, the bundled tasks T_{n+1}, ..., T_{n+m} are processed analytically, as described below.

Route Extension during the Active Stop

A second issue arises when the EGO route ends at the edge of the combined stop station. Upon completion of the stop, SUMO attempts to advance EGO beyond the end of the route. On a non-terminal edge, SUMO randomly selects an outgoing arc, which could potentially send EGO in the wrong direction. On a terminal edge, EGO is removed from the network entirely.

The solution is to extend the route to the next task's destination while EGO is still stopped using the setRoute() function. The critical property that makes this possible is that setRoute() does not cancel the active stop. It only removes future queued stops (stops that have not yet started). Since the combined stop is already active, it is not affected by the call to setRoute(). EGO will complete its combined stop as normal and then follow the extended route.

A further detail is essential for the extension to be accepted by SUMO. The getRoute() function returns the full route of the vehicle, including the edges that have already been traveled, whereas setRoute() requires the new route to start from the edge on which the vehicle currently is. The extension is therefore appended to the remaining part of the current route, starting from the index returned by getRouteIndex(). In an earlier version, the full route was passed to setRoute() and SUMO rejected every extension attempt; since the exception was caught, the error went unnoticed, and EGO continued correctly only because the stop-completion handler recomputes the route as soon as the stop ends. The issue was identified and corrected during testing (Chapter 7).

```
# Executed while EGO is stopped and combined_stop_advance > 0
cs_edge = tasks[next_task_idx].effective_stop_edge
cur_route = list(traci.vehicle.getRoute(EGO_ID))
if cur_route and cur_route[-1] == cs_edge:          # not yet extended
    after_idx = next_task_idx + combined_stop_advance + 1
    ext_dest = (tasks[after_idx].effective_stop_edge
                if after_idx < len(tasks) else EGO_END_EDGE_ID)
    ext_stage = traci.simulation.findRoute(cs_edge, ext_dest, vType="egoType")
    ext_edges = list(ext_stage.edges) if ext_stage and ext_stage.edges else []
    if len(ext_edges) >= 2:
        # The new route must start from the edge EGO is currently on
        route_idx = traci.vehicle.getRouteIndex(EGO_ID)
        # Append the extension, skipping the first edge (= cs_edge)
        traci.vehicle.setRoute(EGO_ID, cur_route[route_idx:] + ext_edges[1:])
        # The active stop is preserved: setRoute only clears FUTURE stops
```

Analytical Processing at Stop Completion

When the combined stop ends (stop_state transitions from stopped to moving), the simulation loop processes the bundled tasks analytically:

```
# combined_stop_advance = m > 0: process tasks T_{n+1} ... T_{n+m}
for _ in range(combined_stop_advance):
    if next_task_idx < len(tasks):
        extra = tasks[next_task_idx]
        cs_e = station_of(extra)                    # assigned station, if any
        if cs_e:
            raw_e = cs_e.power_kw * extra.stop_duration_s / 3600.0
            avail_e = max(0.0, (soc - min_soc) * BATTERY_CAPACITY_KWH)
            v2g_e = min(raw_e, avail_e)
            soc = soc - v2g_e / BATTERY_CAPACITY_KWH
        next_task_idx += 1
combined_stop_advance = 0
```

Analytical processing uses the same energy formula as the normal stop-completion handler in the simulation, ensuring identical V2G accounting, whether the task was part of a combined or standalone stop. The only difference is that analytically processed tasks do not have an associated SUMO stop event: they are processed entirely in Python.

5.5.4 Verification

The combined stop solution can be verified by comparing the console output of two scenarios: one involving consecutive tasks at the same station, and one involving consecutive tasks at different stations. In both cases, the expected output is as follows:

- The V2G energy delivered for each task, including the analytically processed ones, is logged.
- The SoC decreases consistently with the energy delivered at each stop and never falls below the partition floor as a result of V2G discharge.
- No 'stop not downstream' TraCIException appears in the log.
- EGO's route should end correctly at the final destination.

Additionally, the combined stop scenario should print lines such as the following, taken from Scenario 1 of Chapter 7:

```
[EGO] Combined stop T1..T2 (2955s) at 'CS01'
[EGO] Route extended beyond 'CS01'
```

The first line is generated by the combined-stop detection code before calling _add_task_ stop(), providing immediate confirmation that the mechanism is active for the current scenario; the second confirms that the route has been extended while EGO is stopped.

Chapter 6

Dynamic Routing and Replanning

6.1 Routing Architecture Overview

The routing system of this project comprises two independent components that serve complementary functions. The periodic component keeps the route optimal with respect to the current edge costs, re-computing it from scratch every five seconds. Meanwhile, the event-driven component provides a faster response to localized congestion by checking the immediate path ahead every 2.5 seconds and triggering an immediate reroute if congestion is detected, thus avoiding the need to wait for the next periodic cycle.

Both components share the same underlying routing function (_reroute_to_waypoint()) and the same traffic-aware edge cost update function (apply_traffic_aware_travel_times()). The only difference is the triggering condition.

6.2 Energy-Aware Edge Cost Function

6.2.1 Base Travel Time

The base travel time for each edge is the free-flow travel time. This is defined as the length of the edge divided by the posted speed limit.

$$t_{\text{ff}}(e) = \text{length}(e)/v_{\text{limit}}(e) \quad [\text{s}] \tag{6.1}$$

This value forms the basis of all routing cost computations. The speed limit is taken from the static network description loaded with sumolib; changes applied to the edges at runtime (for example, the speed reductions used to inject congestion in Chapter 7) are therefore not reflected in t_{ff}.

6.2.2 Congestion Penalty

Congestion is estimated from the mean observed speed of the non-EGO vehicles currently traveling on each edge. An edge is treated as congested when its mean observed speed falls below the low-speed threshold EGO_LOW_SPEED_THRESHOLD = 4.0 m/s. For these edges only, the free-flow travel time is replaced by the observed travel time, multiplied by a penalty factor:

$$t_{\text{obs}}(e) = \text{length}(e)/\max(\text{mean_speed}(e), \varepsilon) \tag{6.2}$$

where $\varepsilon = 0.1$ m/s prevents division by zero.

$$t_{\text{penalized}}(e) = t_{\text{obs}}(e) \times \text{EGO_LOW_SPEED_PENALTY} \tag{6.3}$$

where EGO_LOW_SPEED_PENALTY = 4.0.

The traffic component of the travel time of an edge is therefore:

$$t_{\text{traffic}}(e) = \begin{cases} t_{\text{penalized}}(e) & \text{if mean_speed}(e) < 4.0 \text{ m/s} \\ t_{\text{ff}}(e) & \text{otherwise} \end{cases} \tag{6.4}$$

Edges with no vehicles, or whose vehicles travel above the threshold, keep their free-flow travel time, even if the observed speed is lower than the speed limit. The fourfold penalty on the observed travel time of near-stationary edges strongly discourages EGO from choosing a congested route, even when the Dijkstra distance would otherwise prefer it. This is equivalent to assuming that the congestion will persist for at least four times the edge's observed travel time.

Since the speed of EGO itself is excluded from the observations, congestion can only be detected on edges where other vehicles are present. As shown in Chapter 7, this makes the mechanism dependent on the density of background traffic.

6.2.3 Energy Component

The energy cost of an edge represents the energy needed to traverse it, and is approximated as proportional to the length of the edge.

$$E(e) = \text{length}(e)/1000.0 \times e_{\text{cons}} \quad [\text{kWh}] \tag{6.5}$$

where $e_{\text{cons}} = 0.20$ kWh/km.

The energy cost is converted into an equivalent travel time using the energy routing weight λ.

$$t_{\text{energy}}(e) = \lambda \times E(e) \quad [\text{s}] \tag{6.6}$$

The total routing cost of an edge is:

$$\text{cost}(e) = t_{\text{traffic}}(e) + t_{\text{energy}}(e) \tag{6.7}$$

This cost is written to SUMO's edge travel time table using traci.edge.adaptTraveltime(), with a validity window of EGO_TRAFFIC_WEIGHT_HORIZON = 500 s starting from the current simulation time. The values are overwritten at the next rerouting call, so that the router always uses the most recent observations. Since the energy term is non-zero on every edge, the cost is rewritten for all the edges of the network at each rerouting call; as discussed in Section 6.6, this update dominates the computational cost of rerouting.

6.2.4 Adaptive Energy Weight

The adaptive energy weight is computed as follows:

$$\lambda = \text{SoC}_{\text{min,dyn}} \times \text{EGO_ENERGY_ROUTING_LAMBDA} \tag{6.8}$$

where EGO_ENERGY_ROUTING_LAMBDA = 50.0 s/kWh.

Here $\text{SoC}_{\text{min,dyn}}$ is the partition floor of Equation (4.6). This design makes the energy weight proportional to the share of the battery that must be reserved for driving. When $\text{SoC}_{\text{min,dyn}} = 0.80$ (long route, large driving reserve), $\lambda = 0.80 \times 50 = 40$ s/kWh, meaning that each kWh of energy consumed adds 40 seconds to the effective route length. When $\text{SoC}_{\text{min,dyn}}$ is close to its lower bound of 0.20 (short route, small driving reserve), $\lambda \approx 0.20 \times 50 = 10$ s/kWh, which has a much weaker influence on routing. In the scenarios of Chapter 7, the partition floor ranged from 20% to 23%, giving $\lambda = 10.0$–11.4 s/kWh.

This adaptive behavior is desirable: when the driving reserve required by the itinerary is small, EGO primarily optimizes for time; when it is large, EGO increasingly favors shorter routes, even at the cost of a slightly longer travel time.

6.3 Periodic Rerouting

6.3.1 Trigger and Implementation

The periodic rerouting fires every 5.0 seconds (or every 10 steps). The trigger condition is:

```
if not is_stopped and step % periodic_interval == 0:
    cur_edge = get_vehicle_current_edge_id(EGO_ID)
    if cur_edge and cur_edge != current_dest:
        new_route = _reroute_to_waypoint(cur_edge, current_dest, ...)
```

Each periodic rerouting step performs the following five operations in sequence (the first three inside _reroute_to_waypoint(), the last two in the main loop):

1. Calls apply_traffic_aware_travel_times() to update edge costs in SUMO's internal routing table.
2. Calls traci.simulation.findRoute() with the updated costs.
3. If a valid route is returned, it applies it with traci.vehicle.setRoute().
4. Calls _add_task_stop() to restore the stop command (setRoute() clears queued stops).
5. Records the call duration and route length in state.PERIODIC_REROUTE_LOG.

6.3.2 Why setStop() Must Be Recalled After setRoute()

This is a critical implementation detail. The SUMO setRoute() command replaces the vehicle's entire planned path and removes all queued stops. Therefore, every call to setRoute() must be followed by a call to setStop() to restore the stop command for the current task. Failure to do so will cause EGO to drive through the task's station without stopping.

If applicable, the combined-stop duration must also be passed to _add_task_stop() via the duration_override parameter to preserve the combined-stop behavior. The combined_stop_advance counter is therefore checked before every re-call of _add_task_stop() in the periodic rerouting code path.

6.4 Event-Driven Rerouting

6.4.1 Congestion Detection

The event-driven mechanism checks for congestion in EGO's immediate path every EGO_EVENT_CHECK_INTERVAL = 2.5 s (every 5 steps). The lookahead scan works as follows:

```
def _check_congestion_ahead(remaining_route, edge_by_id, event_speed_by_edge):
    """Return congested edge IDs on the remaining route."""
    congested = []
    for edge_id in remaining_route:
        edge = edge_by_id.get(edge_id)
        if edge is None:
            continue
        mean_speed = event_speed_by_edge.get(edge_id)
        if mean_speed is None:
            continue
        if mean_speed < edge.getSpeed() * EGO_EVENT_CONGESTION_THRESHOLD:
            congested.append(edge_id)
    return congested
```

The observed speeds are provided by get_lookahead_edge_speeds(), which only includes edges within EGO_EVENT_LOOKAHEAD_DISTANCE of EGO. Edges with no other vehicles have no speed sample and are skipped, which is equivalent to assuming free-flow conditions and prevents false positives on lightly traveled roads. The counterpart of this choice is that a congested edge with no other vehicles on it cannot be detected. Note that the event-driven check uses a different notion of congestion from the routing cost function of Section 6.2.2. In the cost function, an edge is penalized only when its mean speed falls below the absolute threshold of 4.0 m/s, i.e. when traffic is nearly stationary; in the event-driven check, an edge is flagged as soon as its mean speed falls below half of its speed limit. The relative threshold is more sensitive and is meant to trigger a reroute early, whereas the absolute threshold restricts the strong fourfold penalty to edges where traffic is almost at a standstill.

6.4.2 Independence from Periodic Rerouting

The event-driven mechanism fires at 2.5-second intervals, interspersed with the periodic mechanism at 5-second intervals. In the worst case, this means that EGO detects a congestion event 2.5 seconds after it enters the look-ahead window rather than waiting up to five seconds for the next periodic cycle. The two mechanisms are fully independent: an event-driven reroute does not reset the periodic timer and a periodic reroute does not suppress event-driven checks. Figure 6.1 shows the resulting interleaving of the two mechanisms over a representative time window.

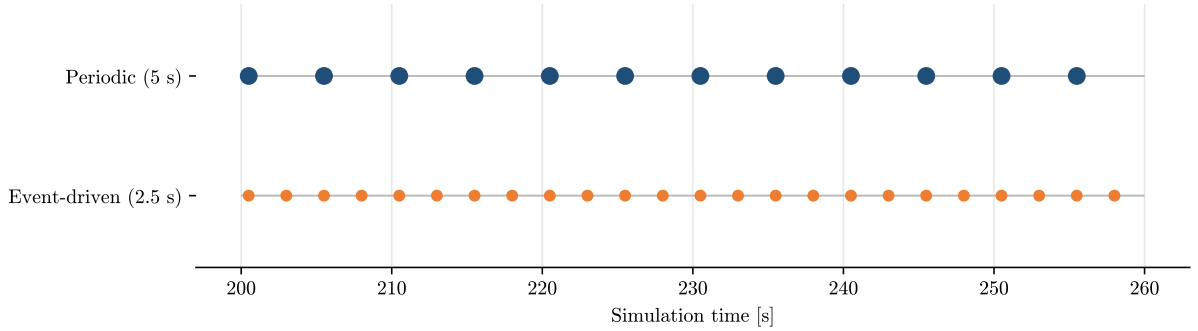


Figure 6.1: Periodic and event-driven rerouting events over a 60 s window of Scenario 1 (Chapter 7), taken from reroute_timing.csv.

6.4.3 Performance Characteristics

The event-driven check exhibits two distinct performance profiles. When no congestion is detected, which is the common case in normal traffic, the check reduces to a few TraCI queries and a Python loop over the few edges of the lookahead window. In the representative run analyzed in Section 6.6, these checks took 0.72 ms on average, with a maximum of 1.75 ms. When congestion is detected, the full rerouting sequence is executed, with a cost comparable to that of a periodic rerouting call (about 74–110 ms in the same run of Scenario 1).

It is this conditional execution pattern that gives the event-driven mechanism its scalability advantage over periodic rerouting, as discussed in Section 6.6. The advantage holds as long as congestion is detected only occasionally. In the dense-traffic scenario of Chapter 7 (S4b), EGO remained on a congested corridor with no alternative route, and the check detected the same congestion every 2.5 s, triggering 232 reroutes that all returned the same route; the mean cost of the check rose to 34 ms. Skipping the reroute when the new route coincides with the current one, or introducing a short cooldown after a triggered reroute, would avoid this redundant work.

6.5 Dynamic Replanning

6.5.1 Motivation and Problem Setting

Dynamic routing adjusts the route according to the current task, but does not reconsider the order of tasks. However, in heavy traffic, EGO may fall so far behind schedule that reordering the remaining tasks would result in fewer late arrivals than simply following the original order. Dynamic replanning addresses this higher-level adaptation by asking: given the vehicle's current state and the remaining task list, is the current ordering still optimal?

Replanning requires the full scheduler to be run again on the remaining tasks. For short remaining lists its cost is negligible, but it grows rapidly with the number of tasks: close to the exact-search limit (N = 8), a single replanning call evaluates up to 40,320 permutations and can take several hundred milliseconds, i.e. several times a routing call. Replanning therefore occurs much less frequently than rerouting, at a rate of at most once every 60 seconds rather than every 2.5–5 seconds.

The 60-second cooldown period also prevents oscillation, where the scheduler repeatedly swaps two tasks back and forth as traffic conditions fluctuate.

6.5.2 Trigger Conditions

A replanning event can be triggered by two conditions:

Post-Stop Trigger

Every time EGO completes a stop and there are at least two remaining tasks, a replanning check is performed. The rationale behind this is that completing a stop provides new information, such as the actual stop duration (which may differ from the planned duration) and the current position and time of EGO, which are passed to the scheduler as the new starting point for re-optimization. The post-stop trigger is always subject to the cooldown period. If a replanning event occurred within REPLAN_CHECK_INTERVAL_S (60 seconds), the post-stop trigger is suppressed. This prevents replanning at every stop in rapid succession.

In the scenarios of Chapter 7, the post-stop trigger was executed after the first stop and always confirmed the current order, which is reported in the console log as 'order unchanged'.

In-Flight Trigger

While driving, the estimated arrival time at the current task is computed using a fresh findRoute() call every 120 simulation steps (60 seconds). If the estimated arrival time exceeds the latest time for the task by more than REPLAN_DELAY_THRESHOLD_S (300 seconds), replanning is triggered immediately.

```
if (not is_stopped and not all_tasks_done
        and combined_stop_advance == 0
        and next_task_idx < len(tasks)
        and step % replan_interval == 0
        and sim_time - last_replan_time >= REPLAN_CHECK_INTERVAL_S):
    cur_task = tasks[next_task_idx]
    stage = traci.simulation.findRoute(
        cur_edge, cur_task.effective_stop_edge, vType="egoType")
    est_arrival = sim_time + stage.travelTime
    overshoot = est_arrival - cur_task.latest_time
    if overshoot > REPLAN_DELAY_THRESHOLD_S:
        if _try_replan(tasks, next_task_idx, sim_time,
                       ego_x, ego_y, stations, reason):
            last_replan_time = sim_time
```

The 300 s threshold prevents replanning on small delays that the PID speed controller could recover from naturally. Only when EGO is more than 5 minutes behind schedule does the system resort to reordering. In the scenarios of Chapter 7, the in-flight trigger never fired: since EGO departs 30 minutes before the first task and does not wait for the task windows, it remained ahead of schedule even when crossing the injected congestion.

6.5.3 The Replanning Algorithm: _try_replan()

The _try_replan() function operates on the sub-list tasks[next_task_idx:], which represents tasks that have not yet been completed. It calls optimize_task_schedule() with two modifications to the standard scheduling setup.

- The starting position is EGO's current (x, y) coordinates rather than the depot's, reflecting the fact that EGO is mid-journey.
- The starting time is the current simulation time rather than t_{depart}.

The current state of charge is not passed to the scheduler, which always starts from BATTERY_INITIAL_SOC. This does not affect the new visit order, which depends only on the number of tasks on time and on the total delay; only the V2G estimates associated with the new schedule are indicative.

If the returned order differs from the current order, _try_replan() updates tasks[next_task_idx:] in place and logs the replanning event; otherwise, it reports that the order is unchanged.

```
def _try_replan(tasks, next_task_idx, sim_time, ego_x, ego_y, stations, reason):
    """Re-optimise tasks[next_task_idx:] given the EGO's current position and time."""
    remaining = tasks[next_task_idx:]
    if len(remaining) < 2:
        return False

    new_result = optimize_task_schedule(
        remaining,
        start_x=ego_x,
        start_y=ego_y,
        depart_time=sim_time,
        stations=stations,
    )

    old_ids = [t.task_id for t in remaining]
    new_ids = [t.task_id for t in new_result.order]
    if old_ids == new_ids:
        print(f"[REPLAN] {reason}  ->  order unchanged")
        return False

    tasks[next_task_idx:] = new_result.order
    for i, t in enumerate(tasks):
        t.scheduled_order = i + 1

    state.REPLAN_TRAFFIC_STATES.append({
        "sim_time": sim_time,
        "reason": reason,
        "new_order": new_ids,
    })
    return True
```

6.5.4 Replanning and Rigid Tasks

A key invariant that replanning must preserve is the priority-1 (rigid) task constraint. Since flex_window = 0 for rigid tasks, latest_time equals target_time, so any permutation in which a rigid task starts after its target time counts that task as not on time in _evaluate_order(). Because the first criterion of the lexicographic objective is the number of tasks on time, such a permutation is dominated by any permutation that keeps the rigid task on time, whenever one exists. Replanning can therefore move a rigid task within the sequence only if doing so keeps it on time (e.g. by routing directly to the rigid task when another task's delay would otherwise cause it to arrive late).

This property is a consequence of the objective function design rather than an explicit constraint: no special-case code is required for rigid tasks.

6.5.5 Combined-Stop Guard

Replanning is suppressed when combined_stop_advance > 0, which prevents an inconsistency. If two tasks, T_n and T_{n+1}, have been bundled into a single SUMO stop, the stop duration has already been communicated to SUMO as stop_duration_s(T_n) + stop_duration_s(T_{n+1}). Reordering T_{n+1} away from T_n at this point would result in SUMO running a stop that is longer than necessary for a task that has been reassigned, producing incorrect timing and V2G calculations.

The guard condition is:

```
replan_eligible = (
    combined_stop_advance == 0                # no active combined stop
    and next_task_idx < len(tasks)            # tasks remain
    and (len(tasks) - next_task_idx) >= 2     # at least 2 remaining
    and sim_time - last_replan_time >= REPLAN_CHECK_INTERVAL_S
)
```

6.5.6 Design Rationale for Trigger Parameters

The threshold values introduced in Section 6.5.2 were not chosen arbitrarily. Each value reflects a trade-off between responsiveness and computational or behavioral cost.

- Why 60 s: Replanning reruns the full scheduler on the remaining tasks. For task lists close to the exact-search limit, this operation can be several times more expensive than a single routing call (see Section 6.6), because it may explore N! permutations for small N, or it may run the greedy-plus-2-opt heuristic for larger N.

  A shorter cooldown period would enable the scheduler to react more quickly to changes in traffic. However, this would incur a higher computational cost. More importantly, there would be a higher risk of oscillation. This could result in two tasks with similar cost estimates being repeatedly swapped back and forth as traffic conditions fluctuate slightly from one check to the next. This would produce a schedule that never settles. A longer cooldown would reduce this risk further, but EGO would follow a stale, possibly suboptimal order for longer after a significant change of state. 60 seconds was chosen as a value large enough for traffic conditions to meaningfully change between checks, yet short enough so that a missed reordering opportunity would not cause a task to become infeasible by itself.

- Why 300 s for the delay threshold: The in-flight trigger (Section 6.5.2) must distinguish between a transient delay, which the PID speed controller (Chapter 3) can often absorb independently, and a persistent, schedule-breaking delay that genuinely warrants reordering. A short threshold would cause the in-flight trigger to fire during ordinary traffic light waits

and short queues, generating unnecessary replanning events. The 300-second (five-minute) threshold is long enough to filter these out, while still identifying a delay before it results in a missed task deadline.

- Sharing one cooldown for both triggers: Instead of giving the post-stop and in-flight triggers their own cooldown timers, they are both gated by the same REPLAN_CHECK_INTERVAL_S. This prevents the two triggers from firing in quick succession (for example, an in-flight replan immediately followed by a post-stop replan a few seconds later), which would double the computational cost of two consecutive scheduler invocations without improving the schedule quality significantly, since little new information arrives within a few seconds.

6.6 Complexity and Scalability Analysis

6.6.1 Motivation

In the current setup, SUMO and the Python controller are coupled through TraCI in lockstep: SUMO waits for the controller before advancing to the next step, so a slow step only slows down the simulation, without affecting its results. However, if the platform is to run in real time, for example coupled with a driving simulator or deployed on board a vehicle, the per-step computation time must not exceed the simulation step length, which in this project is 0.5 seconds; otherwise, the simulation would fall progressively behind the real-time clock. The most expensive per-step operations are the routing operations described in this chapter (findRoute() + setRoute()), making their complexity the primary performance concern.

6.6.2 Theoretical Complexity Analysis

TraCI Routing Complexity

SUMO's internal routing engine uses a variant of Dijkstra's algorithm that employs a binary heap priority queue. The theoretical complexity is O((V + E) log V), where V is the number of nodes and E is the number of edges. For the Pavia network (V = 2,051 nodes, E = 4,136 edges):

O((2,051 + 4,136) log 2,051) ≈ O(6,200 × 11) ≈ O(68,000) operations.

At a modern CPU speed of 10^9 operations per second, this gives a theoretical lower bound of approximately 0.07 ms per findRoute() call. In practice, however, each periodic rerouting call also includes the update of the edge travel times performed by apply_traffic_aware_travel_times() before findRoute(), as well as the overhead of the Python TraCI binding (serialization, socket I/O and deserialization). As shown in Section 6.6.3, the measured time per periodic call is about 74–110 ms in the representative run of Scenario 1, i.e. about a thousand times the theoretical cost of the shortest-path computation alone.

Event-Driven Check Complexity

When no congestion is detected, the event-driven check involves a Python loop over the edges of the lookahead window. The worst-case loop length is bounded by the number of edges within EGO_EVENT_LOOKAHEAD_DISTANCE, which is set to 100 meters. For the urban edges of the Pavia network, which are typically a few tens of meters long, this corresponds to only a few edges per check. The loop itself runs in O(L) time, where L is the small number of edges in the lookahead window, which is negligible; the measured cost of a check (about 0.7 ms) is dominated by the TraCI queries needed to obtain the remaining route and the vehicles within the lookahead window.

When congestion is detected, the full rerouting path is taken, adding the same cost as a periodic call. The amortized cost is therefore a weighted average of these two cases, depending

on traffic. Section 6.6.3 reports the empirically observed per-call cost, and Chapter 7 reports the behavior observed in a congested scenario.

6.6.3 Empirical Timing Results

The theoretical bounds are compared with real measurements collected from a representative simulation run on the Pavia network (Scenario 1 of Chapter 7, log file: reroute_timing.csv). The run comprised 470 routing-cycle events spanning 5,070 seconds of simulated time, from time t = 150.5 to time t = 5220.5.

Key observations from the empirical data:

- Periodic rerouting (137 calls): mean 82.2 ms, median 77.6 ms, minimum 73.9 ms, maximum 109.7 ms.
- Event-driven checks (333 calls, none of which detected congestion during this run): mean 0.72 ms, minimum 0.50 ms, maximum 1.75 ms.
- Call frequency: rerouting and event-driven checks are only performed while EGO is driving (about 830 s out of the 5,070 s of the run, the rest being spent at the stops), at the configured intervals of 5 s and 2.5 s respectively. The number of calls therefore reflects the driving time rather than the total duration of the run.
- Combined routing overhead: 11.5 seconds of computation across the 5,070-second window, which is approximately 0.23% of the simulated time. Each individual call remains well below the 500 ms per-step budget described in Section 6.6.1.

In this particular run, no congestion event was detected during any of the 333 event-driven checks. Therefore, only the lightweight look-ahead scan cost (Section 6.6.2) is represented in the above figures. The cost of a triggered event-driven call invokes the same rerouting path as the periodic case and is therefore comparable to it. This was confirmed in the congested scenario S4b of Chapter 7, where repeated triggered checks raised the mean cost of the event-driven check to 34 ms.

A linear regression analysis of call duration against the number of remaining route edges, based on the same 470 measurements, yields the following results:

$$t_{\text{periodic}}\,(\text{ms}) = 87.6 - 0.24 \times E_{\text{remaining}} \qquad [R^2 = 0.16, n = 137] \tag{6.9}$$

$$t_{\text{event}}\,(\text{ms}) = 0.70 + 0.001 \times E_{\text{remaining}} \qquad [R^2 = 0.01, n = 333] \tag{6.10}$$

The near-zero slope and R^2 of Equation (6.10) for the event-driven check provide useful empirical confirmation of the theoretical analysis in Section 6.6.2. Because the check only scans a fixed look-ahead window of a few edges, its cost is essentially independent of the number of remaining edges on the route.

The periodic call, however, does not grow with the route length either (Equation (6.9)): its cost is dominated by a fixed component of about 80 ms per call. The bottleneck is therefore not the shortest-path computation, whose cost would depend on the route, but the work performed before it in apply_traffic_aware_travel_times(). Since the energy-aware cost term is always active (Section 6.2.3), this function updates the travel time of every edge of the network before each findRoute() call, issuing one traci.edge.adaptTraveltime() call per edge, i.e. several thousand TraCI round trips per rerouting. The routes observed covered between 2 and 62 remaining edges.

This result has two consequences. First, the per-call cost is expected to grow with the size of the network (the number of edges whose travel times are updated) rather than with the length of the route. On the Pavia network, the margin with respect to the 500 ms per-step budget is

about a factor of five; on a network several times larger, the periodic rerouting could approach the budget. Second, the cost could be reduced substantially by updating only the edges whose observed speed or energy cost has changed, or those within a limited distance of EGO, instead of the whole network; since the energy cost of an edge does not change during a run, it could also be set once at the start of the simulation.

These extrapolations should be treated as indicative rather than validated, as they assume that the fixed component scales linearly with the number of edges, as expected from one TraCI call per edge. This assumption has not yet been tested empirically (see Section 9.3). Direct validation on a larger network, such as Milan or Rome, is identified as future work (Chapter 8).

Chapter 7

Simulation Result

7.1 Simulation Setup

All the results reported in this chapter were obtained in simulation, using SUMO as a realistic microscopic traffic simulator; no field tests with a real vehicle on real roads were performed. All simulations were run on the Pavia road network (nave_2.net.xml), with three V2G charging stations: CS01 in the western part of the network, CS02 in the central part and CS03 in the eastern part. All stations have a power output of 22.0 kW and were kept in the same positions in all scenarios, so that the results are comparable. Background traffic consisted of about 100 vehicles distributed uniformly over the simulation time (BG_VEHICLE_COUNT = 100), generated with a fixed random seed (TRAFFIC_SEED = 123, an arbitrary value chosen at the beginning of the project), so that the background traffic is reproducible across runs with the same parameters; in scenario S4b, the number of background vehicles was increased to 600.

Simulation parameters: STEP_LENGTH = 0.5 s; EGO_DEPART = 150 s, with the first task placed 1800 s (30 minutes) after the departure of EGO; BATTERY_INITIAL_SOC = 1.00 (0.45 in scenario S3); BATTERY_CAPACITY_KWH = 60.0 kWh; BATTERY_DRIVE_SAFETY_MARGIN = 0.30; and V2G_FEED_IN_TARIFF_EUR_KWH = 0.10.

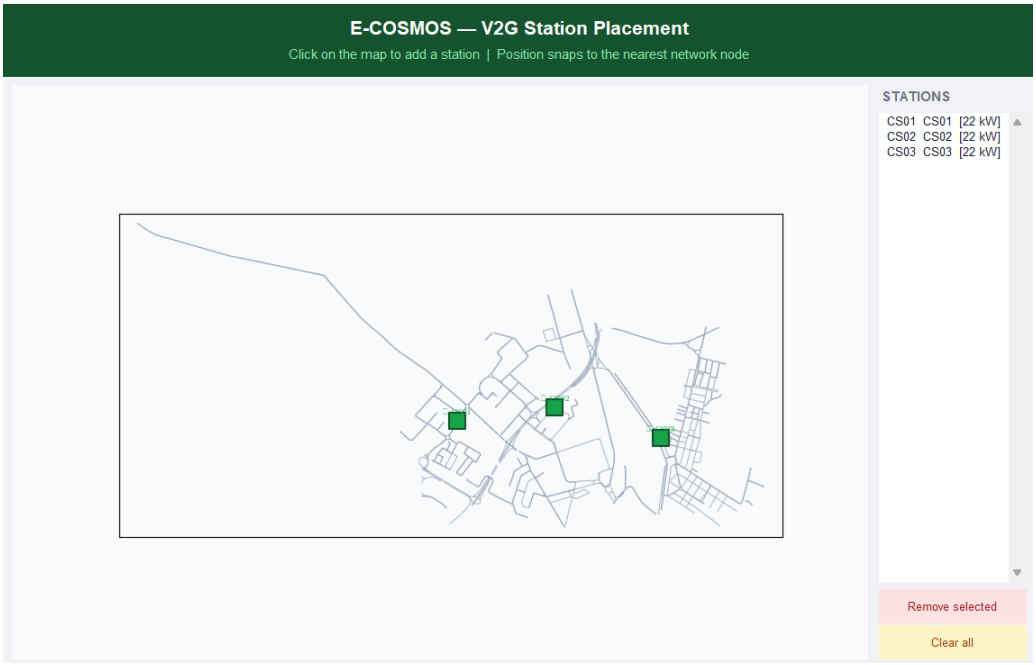


Figure 7.1: Charging station configuration used in all scenarios: CS01 (west), CS02 (center) and CS03 (east).

For each run, the console log, the FCD output, the SUMO trip information and the rerouting timing log (reroute_timing.csv) were saved. All the figures reported in this chapter are taken directly from these files.

7.2 Test Scenarios

Four test scenarios were designed to exercise specific system behaviors, plus a variant of the fourth. Each scenario is defined by a set of three tasks with specified station assignments, priorities, durations, and target times.

Table 7.1: Test scenario definitions.

ID	Name	Tasks	Test Focus
S1	Same-Station Consecutive	T1: CS01, P2, 20 min, 09:00 T2: CS01, P3, 15 min, 09:25 T3: CS03, P4, 10 min, 10:00	Exercises combined stop logic. Tests: no 'stop not downstream' error; T2 V2G computed analytically; route extended toward CS03 while EGO is stopped at CS01.
S2	Rigid Priority	T1: CS01, P3, 20 min, 09:00 T2: CS02, P1, 30 min, 10:00 T3: CS02, P5, 15 min, 10:15	Exercises the P1 constraint. Tests: flexible tasks are reordered around T2, which stays at its target time; post-stop replanning never moves T2.
S3	Battery Partition Stress	T1: CS03, P2, 40 min, 09:00 T2: CS01, P3, 40 min, 10:00 T3: CS02, P4, 30 min, 11:00	Long inter-task distances, long stops and initial SoC of 45%. Tests: V2G limited by the partition floor; SoC evolution with respect to the floor and to the absolute minimum.
S4	Mixed Priority + Congestion	T1: CS03, P2, 20 min, 09:00 T2: CS01, P1, 20 min, 09:45 T3: CS02, P3, 20 min, 10:30	Congestion injected at $t = 150$ s on the main west–east corridor, sparse background traffic. Tests: detection of the congestion, rerouting, replanning; T2 (P1) preserved.
S4b	Dense Traffic Variant	As S4	As S4, with 600 background vehicles.

7.3 Scenario Results

7.3.1 Scenario 1: Same-Station Consecutive

The scenario confirms that the combined stop mechanism works correctly. The scheduler assigned T1 and T2 to CS01 and T3 to CS03, and the console log shows:

```
[EGO] Combined stop T1..T2 (2955s) at 'CS01'
[EGO] Stop #1 done: 'Task 1'  t=3293.5s
[EGO] Stop #2 done (combined stop): 'Task 2'  t=3293.5s
[EGO] Stop #3 done: 'Task 3'  t=4953.0s
```

The 'stop not downstream' exception does not appear. A single SUMO stop of 2955 s, equal to the sum of the stop durations of T1 and T2, was issued at CS01, and the route was extended toward CS03 while EGO was still stopped (the log reports the corresponding message 'Route

extended beyond CS01'). T2 was processed analytically at the end of the combined stop, with a V2G energy of 9.20 kWh, and T3 was then visited at CS03. Table 7.2 summarizes the results.

Table 7.2: Scenario 1 results.

Quantity	**Value**
V2G energy T1 / T2 (analytical) / T3	8.86 / 9.20 / 7.86 kWh
Total V2G energy (simulation)	25.92 kWh (€2.59)
Total V2G energy (scheduler estimate)	25.92 kWh (€2.59)
Distance driven	12.5 km
Trip duration (of which stopped)	5071 s (4242 s)
Final SoC	52.6%

The simulated V2G energy coincides with the scheduler estimate, because no task required EGO to wait for its flexibility window and the partition floor was never approached. The final SoC is also consistent with the energy balance of the battery model (Equations (4.3) and (4.10)): $1.00 - (25.92 + 12.5 \times 0.20)/60.0 = 0.526$, i.e. 52.6%.

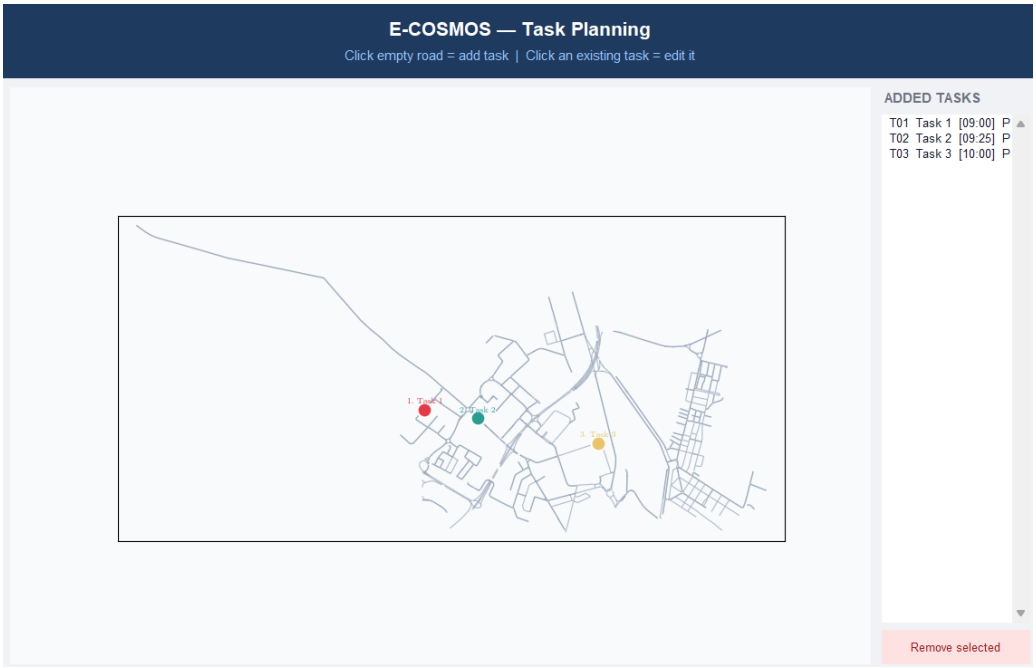


Figure 7.2: Scenario 1: task placement on the map.

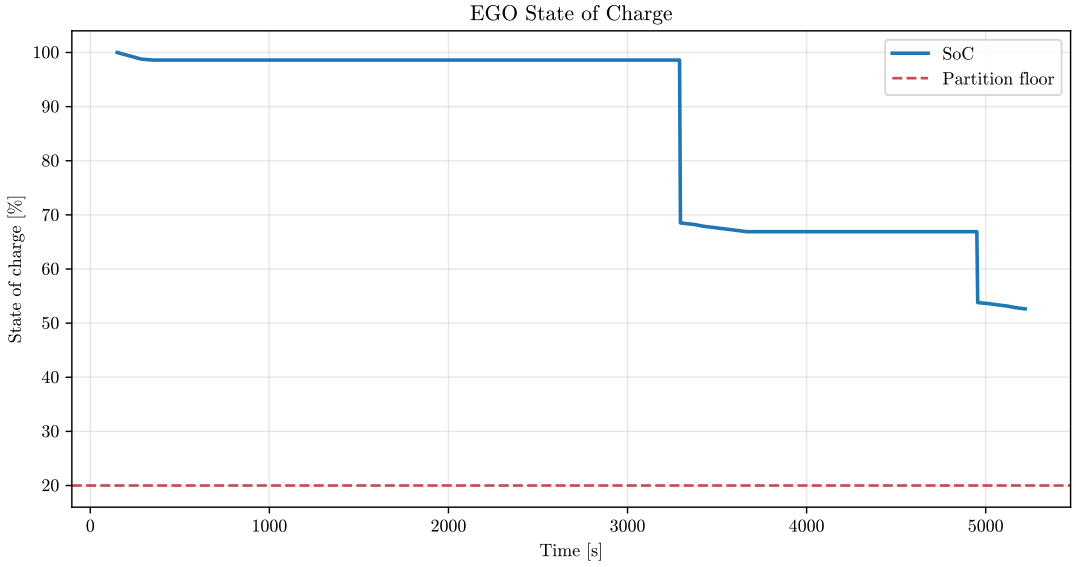


Figure 7.3: Scenario 1: state of charge of EGO. The first step corresponds to the combined stop of T1 and T2 at CS01, the second to T3 at CS03.

7.3.2 Scenario 2: Rigid Priority

The scheduler produced the order T1 → T3 → T2: the very flexible task T3 (P5, ±60 min) was moved before the rigid task T2 (P1), which remained fixed at its target time of 10:00. Since T3 and T2 were both assigned to CS02, they were executed as a combined stop, showing that the combined-stop mechanism also handles chains that include a rigid task:

```
[EGO] Stop #1 done: 'Task 1'  t=1791.5s
[REPLAN] post-stop 'Task 1' t=1792s  ->  order unchanged
[EGO] Combined stop T2..T3 (2663s) at 'CS02'
[EGO] Stop #2 done: 'Task 3'  t=4588.0s
[EGO] Stop #3 done (combined stop): 'Task 2'  t=4588.0s
```

In the log, the label T2..T3 refers to the positions of the tasks in the optimized order (second and third), i.e. to Task 3 and Task 2. The post-stop replanning check performed after T1 returned the same order [T3, T2], indicating that no reordering was necessary. The P1 constraint on T2 was never violated.

Table 7.3: Scenario 2 results.

Quantity	**Value**
V2G energy T1 / T3 / T2 (analytical)	8.88 / 7.46 / 8.81 kWh
Total V2G energy (simulation)	25.15 kWh (€2.52)
Total V2G energy (scheduler estimate)	33.03 kWh (€3.30)
Distance driven	7.7 km
Trip duration (of which stopped)	4620 s (4116 s)
Final SoC	55.5%

Unlike Scenario 1, the scheduler estimate exceeds the simulated V2G energy by 7.88 kWh. The scheduler planned EGO to arrive at CS02 at 09:30 and to wait until the rigid start time of T2 (10:00), counting the waiting time as additional discharge time. In the simulation, however, the duration of each SUMO stop is always stop_duration_s and EGO does not wait for the

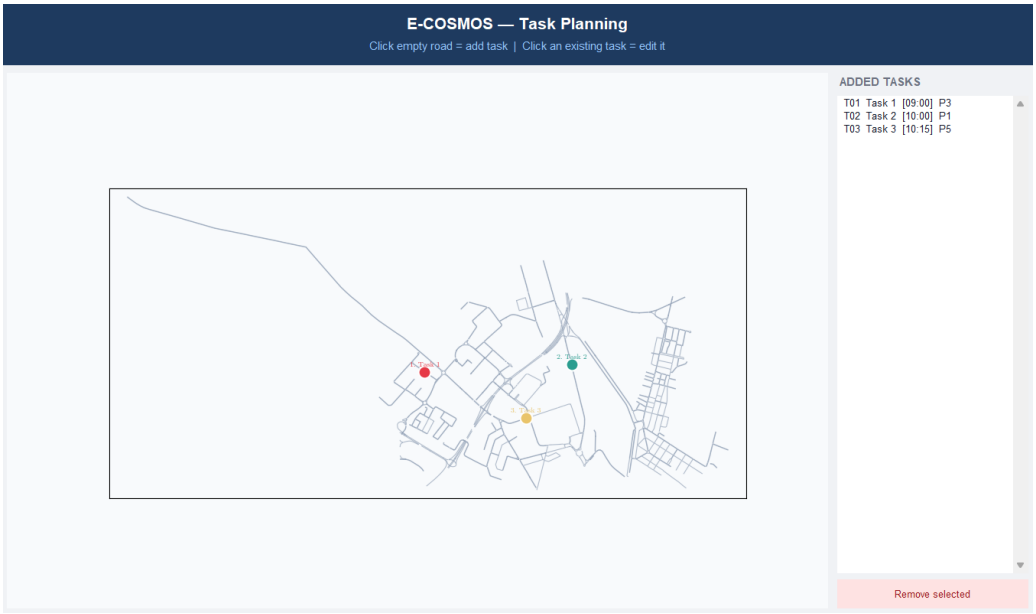


Figure 7.4: Scenario 2: task placement on the map.

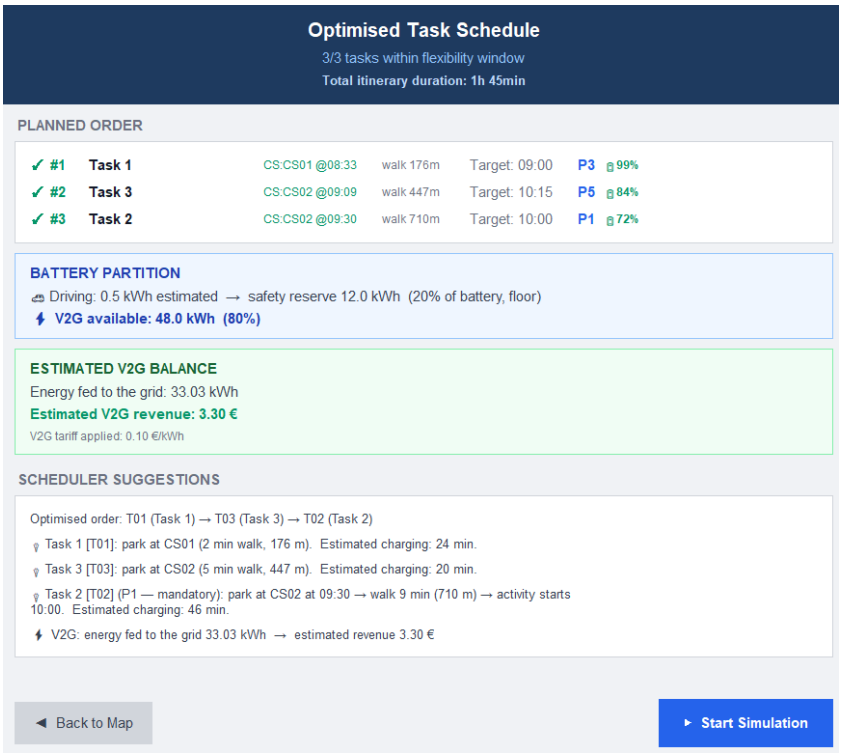


Figure 7.5: Scenario 2: optimized schedule. The flexible task T3 is moved before the rigid task T2, which is kept at 10:00; EGO is planned to arrive at CS02 at 09:30 and wait for T2.

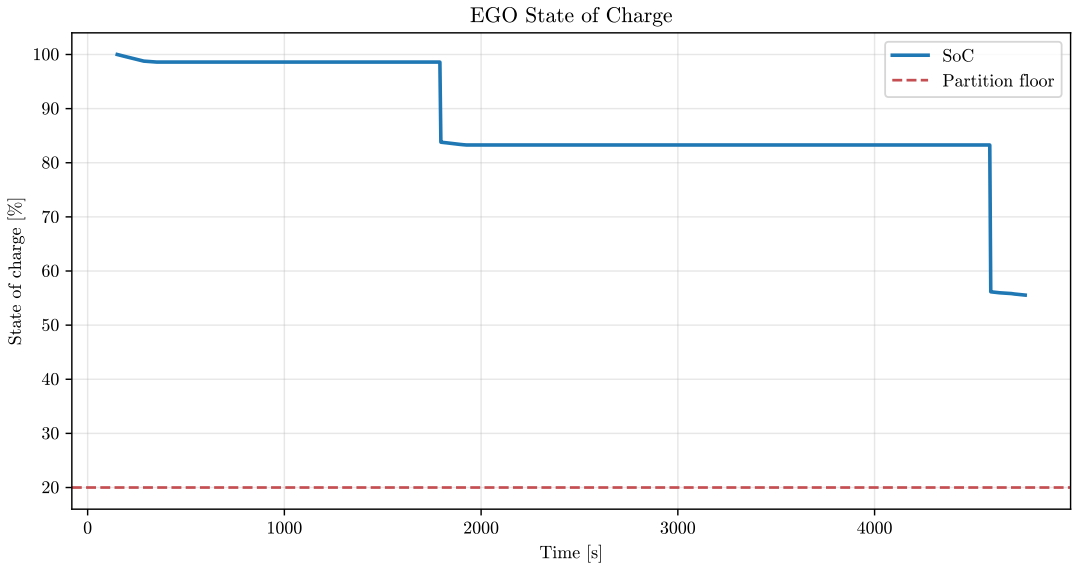


Figure 7.6: Scenario 2: state of charge of EGO. The first step corresponds to T1 at CS01, the second to the combined stop of T3 and T2 at CS02.

task window; the waiting time, and the corresponding V2G energy, are therefore not reproduced. This discrepancy quantifies a limitation of the current simulation model (Section 4.5.3).

7.3.3 Scenario 3: Battery Partition Stress

With tasks placed at opposite ends of the network, long stops and an initial SoC of 45% (representing a vehicle that was not recharged overnight), the battery partition model (Equation (4.6)) computes:

$$
\begin{aligned}
E_{\text{drive}} &\approx 1.3\,\text{kWh} \quad \text{(Euclidean estimate)} \\
E_{\text{reserve}} &= E_{\text{drive}} \times 1.30 \approx 1.69\,\text{kWh} \\
\text{SoC}_{\text{min,dyn}} &= \min\left(\text{SoC}_{\text{abs,min}} + E_{\text{reserve}}/C_{\text{bat}},\ \text{SoC}_0 - \delta\right) = \min(0.20 + 0.028,\ 0.40) = 0.228
\end{aligned}
$$

leaving 13.3 kWh available for V2G. The console log shows:

```
[EGO] Stop #1 done: 'Task 1'  t=3410.5s
[REPLAN] post-stop 'Task 1' t=3410s  ->  order unchanged
[EGO] Stop #2 done: 'Task 2'  t=6629.0s
[EGO] Stop #3 done: 'Task 3'  t=9594.0s
[EGO] Trip ended  time=9600.0s  tasks completed=3/3
```

Table 7.4 reports the evolution of the state of charge.

Table 7.4: Scenario 3: evolution of the state of charge.

Event	**SoC**	**Note**
Departure	45.0%	
After T1 (CS03)	22.8%	11.62 kWh delivered, limited by the partition floor
At T2 (CS01)	21.5%	0 kWh delivered (SoC below the floor)
At T3 (CS02)	20.9%	0 kWh delivered
End of day	20.3%	above the absolute minimum of 20%

The partition behaved as intended: V2G discharge stopped exactly at the floor after the first stop, and no energy was delivered at the following stops. However, EGO actually drove 15.9 km, corresponding to 3.19 kWh: almost twice the reserve estimated by the scheduler from Euclidean distances (1.69 kWh). As a result, EGO ended the day below the partition floor (20.3% against 22.8%), and it was the absolute minimum of 20% that prevented a genuine energy shortfall. This shows that the 30% safety margin does not compensate for the systematic underestimation of road distances by Euclidean distances, and motivates the use of road-network distances for the driving reserve (Chapter 8).

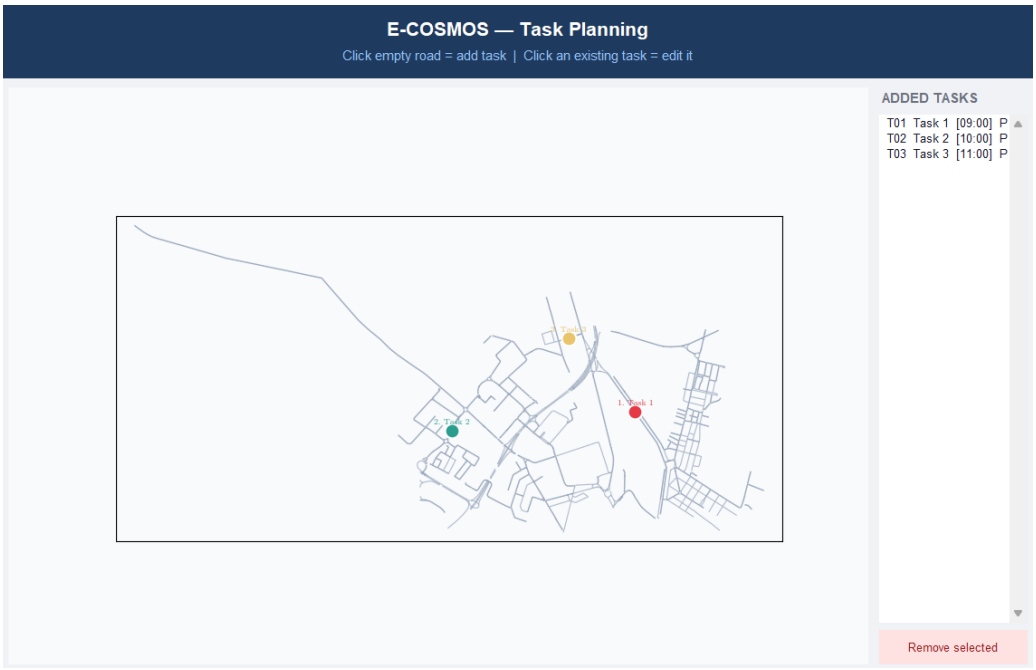


Figure 7.7: Scenario 3: task placement on the map, with the tasks at opposite ends of the network.

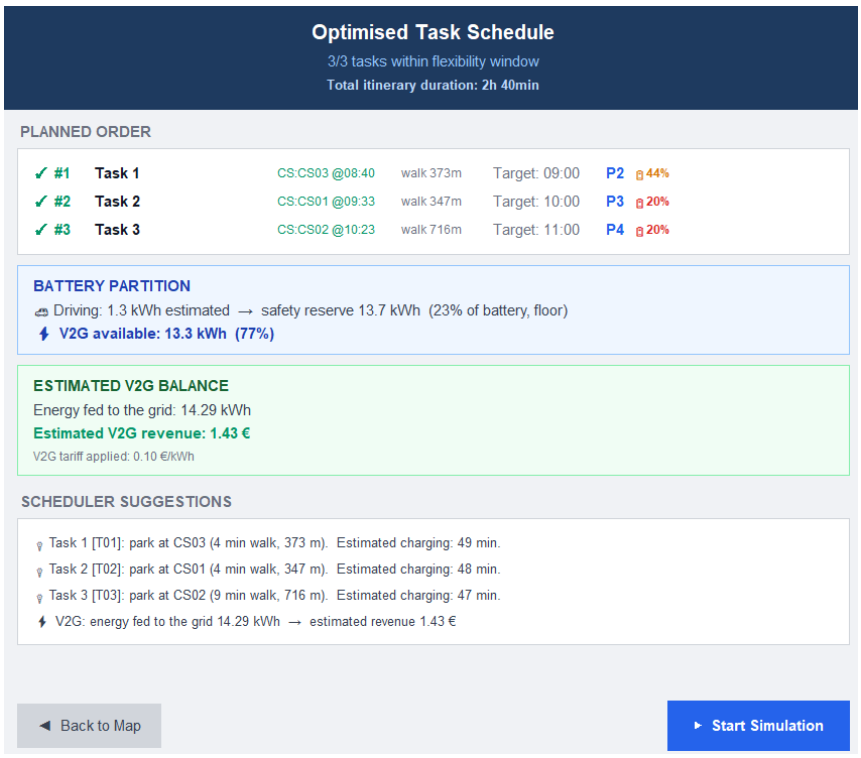


Figure 7.8: Scenario 3: optimized schedule with an initial SoC of 45%. The battery partition panel shows a floor of 23% and 13.3 kWh available for V2G.

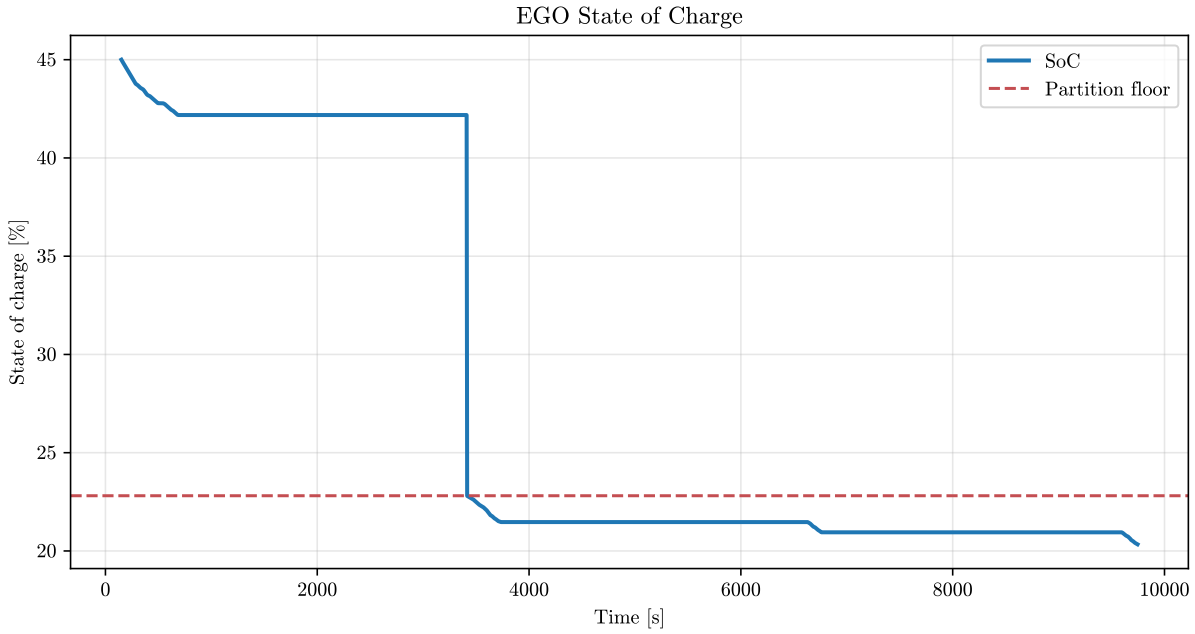


Figure 7.9: Scenario 3: state of charge of EGO. The step at the end of the first stop corresponds to the V2G energy delivered, accounted for at stop completion; the dashed line shows the partition floor.

7.3.4 Scenario 4: Mixed Priority + Congestion

Congestion is introduced at sim_time = 150 s by reducing the maximum speed of three consecutive edges of the main west–east corridor (about 735 m in total) to 1.5 m/s (5.4 km/h). The initial route of EGO toward T1 (CS03) crosses these edges:

```
[CONGESTION] Max speed set to 1.5 m/s on 3 edge(s) at t=150s
```

Sparse Traffic (S4)

With 100 background vehicles, only one vehicle traveled on the congested edges while EGO was approaching them. Since congestion is estimated exclusively from the speeds of the other vehicles, the congested edges appeared free: none of the 626 event-driven checks detected congestion, and the periodic rerouting kept the original route. EGO crossed the congested edges between $t = 445$ s and $t = 913$ s, spending about 7.8 minutes on a stretch that would take less than one minute in free-flow conditions. The speed of EGO itself is excluded from the speed samples, so the system did not detect the congestion even while EGO was experiencing it. Moreover, the free-flow travel time used by the router is based on the static speed limits of the network, so the reduced maximum speed of the congested edges was not reflected in the routing costs (Section 6.2.1).

Dense Traffic (S4b)

With 600 background vehicles, up to eight vehicles were traveling on the congested edges at about 1.3 m/s during the approach of EGO, and the event-driven mechanism detected the congestion as soon as the edges entered the look-ahead window:

```
[EGO-EVENT] Reroute at t=440.5s  (1 congested edges) -> 24 edges to '-346382169#2'
```

A total of 232 event-driven reroutes were triggered, and the periodic rerouting also had speed samples for the congested edges. Nevertheless, EGO still crossed them (from $t = 452$ s to

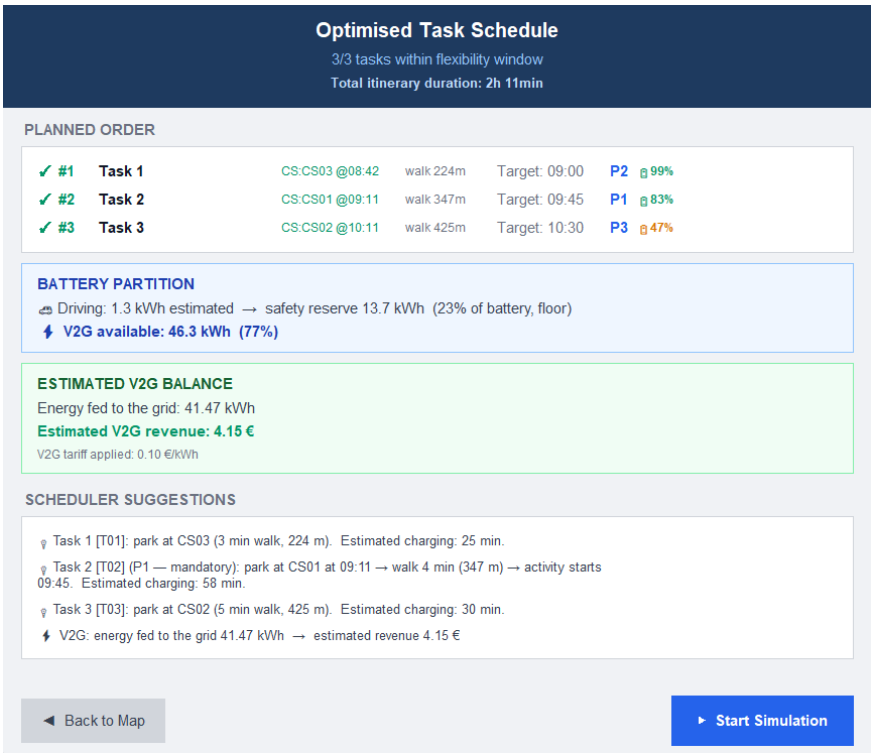


Figure 7.10: Scenarios 4 and 4b: optimized schedule (identical in both runs). EGO is planned to wait at CS01 before the rigid task T2.

$t = 1085$ s), because every rerouting computation returned the same route. The analysis of the background traffic explains this result: 90 of the 93 background vehicles that crossed the network between the western and the eastern areas used the congested corridor, which is effectively the only link between the two areas. The rerouting mechanism therefore worked as intended, but no reasonable alternative route existed.

Table 7.5: Scenario 4 results with sparse (S4) and dense (S4b) background traffic.

Quantity	**S4 (sparse)**	**S4b (dense)**
Background vehicles	100	600
Congestion detected	No	Yes (232 event-driven reroutes)
Congested edges avoided	No	No (no alternative route)
Time on congested edges	about 7.8 min	about 10.5 min
Trip duration	6600 s	6781 s
V2G energy (simulation / estimate)	30.77 / 41.47 kWh	30.77 / 41.47 kWh
Final SoC	43.4%	43.4%

In both runs, T2 (P1) was never moved from its position, and the post-stop replanning after T1 confirmed the current order. The in-flight replanning was not triggered: since EGO departs 30 minutes before the first task and does not wait for the task windows, it remained ahead of schedule even with the injected congestion, and the estimated delay never exceeded the 300 s threshold. As in Scenario 2, the scheduler estimate of the V2G energy (41.47 kWh) exceeds the simulated value (30.77 kWh), because the scheduler counts the waiting time at CS01 before the rigid task T2 as discharge time.

Finally, once EGO was inside the congested corridor in S4b, the event-driven check kept detecting the congestion ahead and recomputed the same route every 2.5 s. This raised the mean cost of an event-driven check from about 1 ms to 34 ms (Table 7.6); a simple improvement would be to skip the reroute when the new route coincides with the current one.

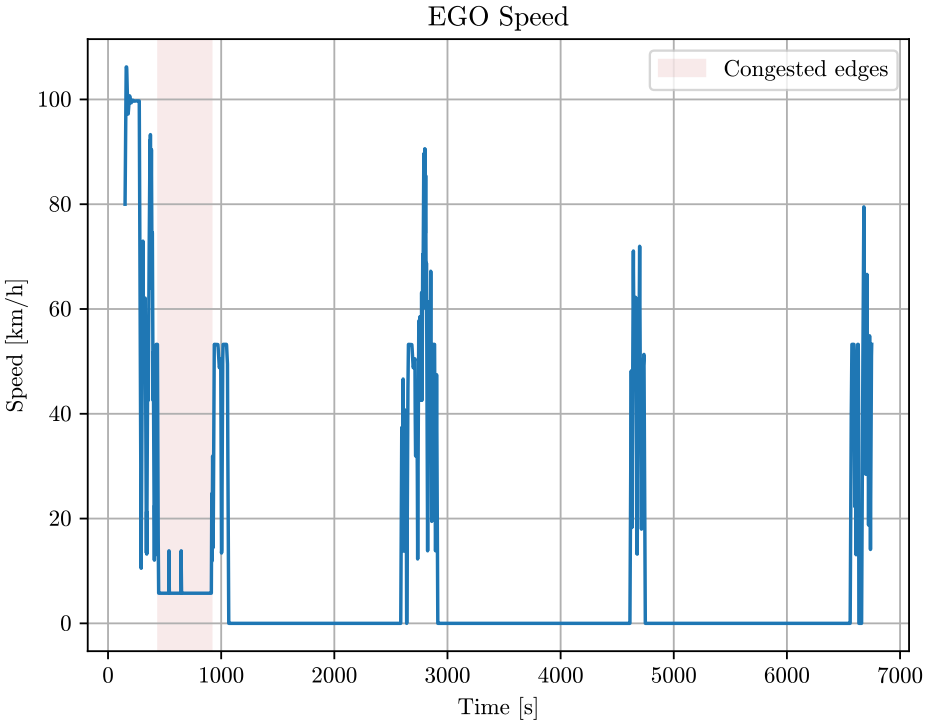


Figure 7.11: Scenario 4 (sparse traffic): EGO speed profile. The plateau at about 1.6 m/s between $t = 445$ s and $t = 913$ s corresponds to the passage through the congested edges.

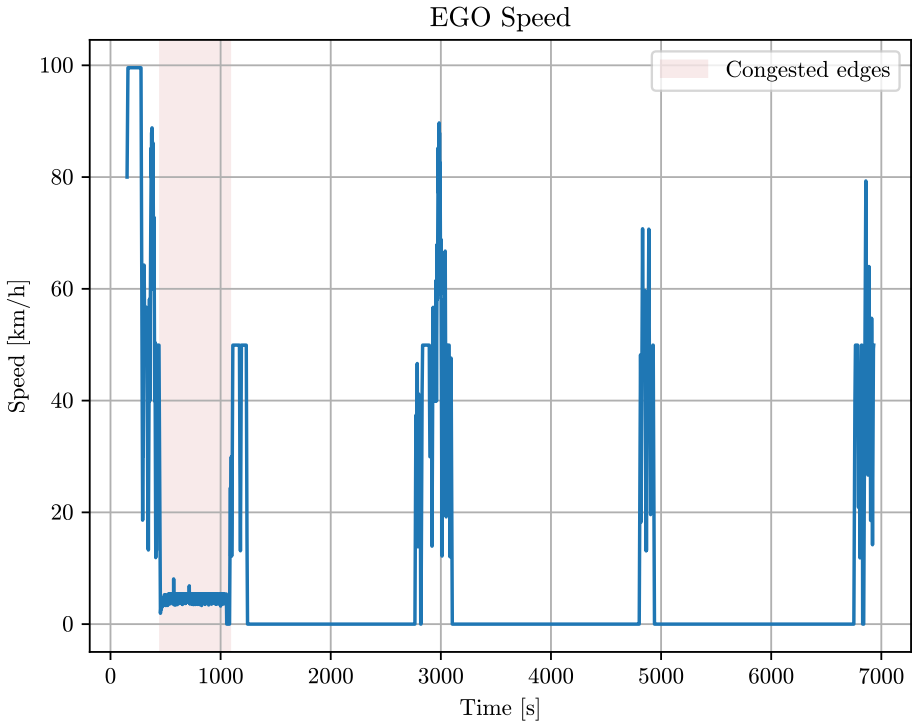


Figure 7.12: Scenario 4b (dense traffic): EGO speed profile. The passage through the congested edges lasts from $t = 452$ s to $t = 1085$ s, longer than in S4 because of the queue formed by the background vehicles.

7.3.5 Computational Performance

Table 7.6 reports the wall-clock time of the periodic rerouting calls and of the event-driven checks in each run, as recorded in reroute_timing.csv.

Table 7.6: Computational cost of periodic rerouting and event-driven checks.

Run	**Periodic rerouting: calls, mean (min–max)**	**Event-driven checks: calls, mean (min–max)**
S1	137, 82.2 ms (73.9–109.7)	333, 0.72 ms (0.50–1.75)
S2	77, 82.3 ms (73.9–117.3)	202, 0.65 ms (0.46–1.60)
S3	186, 115.0 ms (90.3–168.6)	458, 1.43 ms (0.72–3.15)
S4	262, 83.1 ms (73.7–125.8)	626, 0.84 ms (0.59–2.67)
S4b	299, 95.7 ms (75.6–214.1)	698, 34.0 ms (0.61–169.7)

A periodic rerouting call has a mean cost of about 80–115 ms across the runs, largely independent of the length of the remaining route. This suggests that the cost is dominated by the update of the travel times of the whole network rather than by the shortest-path computation itself (Section 6.6). An event-driven check without congestion costs about 1 ms, roughly two orders of magnitude less than a periodic call. In all runs, the per-call cost remains well below the 500 ms step length, although the margin (about a factor of five for the periodic calls) is smaller than a pure shortest-path analysis would suggest.

7.3.6 Issues Identified During Testing

Running the scenarios revealed several implementation issues, which were corrected during the simulation campaign:

- Route extension during the combined stop: the extended route was built from the full route of EGO, including the edges already traveled, and was therefore rejected by SUMO. The route now starts from the current route index (getRouteIndex()).
- Battery partition floor: the floor was computed as the maximum of the absolute minimum and the driving reserve, so that for short routes the driving reserve was absorbed by the absolute minimum. It is now computed as their sum (Section 4.4).
- State of charge bookkeeping: the driving consumption was clamped at the partition floor, and the SoC was raised back to the floor after each stop. Both clamps have been removed, so that the SoC always reflects the energy actually consumed, and a warning is issued if it falls below the absolute minimum.
- Timing of the event-driven check: the timer started after the congestion scan, so that only the rerouting part was measured; it now includes the whole check.

Scenarios S1 and S2 were run before the correction of the partition floor. Since the SoC never approached the floor in these runs, their V2G and SoC results are not affected; only the energy routing weight differs slightly (10.0 instead of about 11 s/kWh).

7.4 Discussion

The results show that the technical objectives defined in Chapter 1 have been met in the configurations considered, and at the same time identify the main limitations of the platform.

O1 (Scheduler): Correct station assignment and ordering for all scenarios, with the brute-force optimizer finding optimal solutions for N = 3. In Scenario 2, the flexible tasks were reordered around the rigid task, which remained at its target time.

O2 (Simulation engine): EGO successfully completed all tasks in all scenarios without generating any stop failure or network removal event.

O3 (Routing): Event-driven rerouting correctly detected the injected congestion when enough probe vehicles were present (S4b), but it was blind to it with sparse traffic (S4), and it could not avoid the congested corridor because no alternative route existed.

O4 (Replanning): The post-stop replanning was executed after the first stop in Scenarios 2, 3 and 4 and correctly confirmed the current order, preserving the P1 constraint. The in-flight replanning was never triggered, because EGO remained ahead of schedule in all scenarios.

O5 (Combined stop): There is no 'stop not downstream' error in Scenarios 1 and 2, the V2G energy of the bundled tasks is computed correctly, and the route is extended while EGO is stopped.

O6 (Performance): The per-step computational cost remained well within the real-time budget in all runs.

The tests also highlight four limitations: the underestimation of the driving energy based on Euclidean distances (S3); the absence of waiting for the task windows in the simulation, which makes the scheduler overestimate the V2G energy when a task requires waiting (S2, S4); the dependence of congestion detection on the density of probe vehicles (S4); and the dependence of congestion avoidance on the network topology (S4b). These limitations are taken up in Chapter 8.

It is worth noting that energy-aware routing does not significantly affect the results in these scenarios, because the partition floor remains close to the absolute minimum (20–23%) and the

resulting energy weight is low ($\lambda \approx 10$–11.4 s/kWh). The benefits of energy-aware routing would be more apparent in scenarios involving longer routes or a more depleted battery at the start of the day. These scenarios are left as future work.

Chapter 8

Future Developments

The following directions for future development have been identified. The first group addresses the limitations that emerged from the simulation tests of Chapter 7; the second extends the scope of the platform.

8.1 Improvements Suggested by the Simulation Results

- Accurate pre-simulation routing: Replace the Euclidean travel time and distance estimates in the scheduler with a SUMO-based static routing query. This requires sumolib's routing engine to be run before the simulation starts, adding a few seconds to the start-up time, but significantly improving schedule quality. It would also improve the driving energy reserve of the battery partition: in Scenario 3, the energy actually needed for driving was about twice the Euclidean estimate, and the 30% safety margin was not sufficient to compensate for this difference.
- Waiting at the stations in the simulation: Model the waiting time before the task window in the SUMO simulation, by extending the stop duration until the earliest start time of the task. This would make the simulated V2G energy consistent with the scheduler estimate, which already includes the waiting time (a difference of 7.88 kWh was observed in Scenario 2).
- Closed-loop battery model: Implement a closed-loop energy model that updates the battery partition parameters mid-simulation based on actual energy consumption, correcting for estimation errors without triggering a full replanning event. In the same spirit, the current state of charge could be passed to the scheduler during replanning, which currently always starts from the initial SoC.
- More robust congestion detection: The current mechanism estimates congestion only from the speeds of the other vehicles on each edge, and ignores both the speed of EGO itself and the runtime changes to the edge speed limits. As shown in Scenario 4, this makes it blind to congestion when traffic is sparse. The observations could be complemented with the speed of EGO and with the aggregated edge measurements provided by SUMO (for example, the mean speed of the last step), or, in a real deployment, with external traffic data.
- More efficient rerouting: Each rerouting call currently rewrites the travel time of every edge of the network, which dominates its computational cost (about 80 ms per call on the Pavia network). Since the energy cost of an edge does not change during a run, it could be set once at the start of the simulation, and only the edges whose observed speed has changed would then need to be updated. In addition, a reroute could be skipped when it

returns the current route, or followed by a short cooldown, to avoid the repeated identical reroutes observed in Scenario 4b.

- Enforcement of the maximum walking distance: Apply the MAX_WALK_DISTANCE_M threshold defined in the configuration, which is currently not enforced, reverting to direct edge stopping when no station lies within walking distance of a task.
- Additional test scenarios: Test the system on networks and scenarios that exercise the mechanisms not triggered in Chapter 7: congested corridors with alternative routes, tight schedules that make EGO fall behind and trigger the in-flight replanning, and longer routes or partially depleted batteries that increase the influence of energy-aware routing.
- Large network validation: Validate the system on larger networks (e.g. Milan and Rome) to confirm the scalability projections of Section 6.6 using real data. Since the cost of rerouting grows with the number of edges of the network rather than with the length of the route, this validation is particularly relevant for city-scale deployments.

8.2 Extensions of the Platform

- Dynamic V2G pricing: Integrate a time-of-use V2G tariff profile, reflecting the remuneration of dispatching services rather than a constant tariff, to enable the scheduler to prioritize discharging during periods of high revenue.
- Multi-vehicle fleet: Extend the platform to multi-vehicle scenarios (i.e. fleet scheduling) with shared charging stations and coordinated charging and discharging strategies to avoid multiple vehicles discharging simultaneously at the same station.
- Economic impact study: Evaluate the economic impact of V2G revenue under different tariff structures and ownership models (private vs. fleet management) to quantify the business case for V2G-enabled electric vehicles (EVs).
- Driver behavior: Incorporate a driver behavior model that can override the scheduled plan (e.g. accepting or refusing a V2G request based on remaining range anxiety) to make the simulation more realistic for behavioral studies.

Chapter 9

Conclusions

9.1 Summary of Contributions

This thesis presents a software platform that integrates the scheduling and simulation of V2G-capable electric vehicles within urban road networks. The work makes the following five original contributions:

1. A dual-objective task scheduler that optimizes visit order using an exact search for up to eight tasks, and a greedy + 2-opt approach for larger instances. This is combined with a dynamic battery partition model that reserves the estimated driving energy on top of the absolute minimum state of charge and allocates the surplus to V2G discharge. The partition floor is computed as the sum of the absolute minimum and the driving reserve, capped below the initial state of charge, so that the energy needed for driving is always kept above the absolute minimum.

2. A step-by-step simulation engine that controls an EGO vehicle via TraCI through an eight-step structured loop. This integrates stop management (three-level fallback), battery tracking, V2G accounting and speed regulation via a PID controller with road-type adaptive targets, and records the state of charge and the rerouting timings for post-simulation analysis.

3. A dual-mode routing architecture incorporating periodic, traffic-aware rerouting every 5 seconds and event-driven congestion response every 2.5 seconds. Both features use an energy-aware edge cost function with an adaptive weight that increases with the share of the battery reserved for driving.

4. A task replanning mechanism that re-optimizes the remaining tasks during the simulation, both after each stop and while driving, when estimated delays exceed the flexibility threshold. This mechanism is suppressed during active combined stops and limited by a cooldown period, while rigid-priority tasks are preserved by the design of the objective function rather than by explicit constraints.

5. Identification and resolution of the same-station consecutive stop problem, which is a TraCI-specific failure mode caused by SUMO's geometric stop validation rules. The combined stop solution aggregates consecutive same-station tasks into a single SUMO stop, processing bundled tasks analytically with route extension during the active stop to ensure correct network topology.

These contributions correspond to the objectives outlined in Section 1.3, and the simulation campaign of Chapter 7 provides evidence for each of them. The scheduler (O1) produced feasible orders in all scenarios and, in Scenario 2, reordered the flexible tasks around a rigid task that

remained at its target time. The simulation engine (O2) completed all the tasks in every run without stop failures. The routing architecture (O3) detected the injected congestion when enough probe vehicles were present, although it could not avoid a corridor for which no alternative route existed. The replanning mechanism (O4) was executed after each first stop and correctly preserved the current order, while the in-flight trigger was never activated because EGO always remained ahead of schedule. The combined stop (O5) was executed correctly in Scenarios 1 and 2, including the extension of the route during the stop. Finally, the analysis of computational cost (O6), developed in Section 6.6 and confirmed by the measurements of Chapter 7, showed that the per-step cost remains well within the real-time budget on the Pavia network.

The simulation campaign also proved valuable as a validation tool in its own right: running the scenarios revealed several implementation issues, concerning the battery partition floor, the bookkeeping of the state of charge, the route extension during the combined stop and the timing of the event-driven check, which were identified through the analysis of the logs and corrected before producing the final results.

9.2 Limitations

For a complete evaluation of the system, several limitations of the current implementation deserve acknowledgment. Possible ways to address them are discussed in Chapter 8.

- Scheduler accuracy: The scheduler estimates travel times and distances from Euclidean distances and a constant speed of 30 km/h. This underestimates both the actual travel times in congested conditions and the actual driving distances: in Scenario 3, the energy needed for driving was about twice the Euclidean estimate.
- Safety margin: The battery partition model uses a single safety margin of 30%, applied uniformly to all routes. As shown in Scenario 3, this margin does not compensate for the difference between Euclidean and road distances, and the absolute minimum state of charge acts as the ultimate safeguard.
- Energy model: The energy model does not account for regenerative braking, which can recover a significant share of the kinetic energy during urban driving, nor for road slope and speed. Including these effects would change both the effective range and V2G availability.
- Waiting at the stations: The simulation does not model the waiting time before a task window: EGO starts each activity as soon as it arrives. As a result, the simulated V2G energy can be lower than the scheduler estimate when a task requires waiting, as observed in Scenarios 2 and 4.
- Congestion awareness: Congestion is estimated only from the speeds of the other vehicles on each edge. With sparse traffic, congested edges may appear free, as observed in Scenario 4.
- Station assignment: Station assignment uses Euclidean distance, which ignores actual walking routes and pedestrian infrastructure. In cities with waterways, highways or other barriers to pedestrians, the actual walking distance may be considerably longer than the Euclidean estimate. Moreover, the maximum walking distance defined in the configuration is not enforced.
- V2G economics: The V2G tariff is constant. In practice, V2G is remunerated through the dispatching services market, with prices that vary over time and reward discharge during hours of peak grid stress.

9.3 Threats to Validity

In addition to the specific limitations listed above, the conclusions drawn from the simulation campaign in Chapter 7 are constrained by several methodological factors, which should be considered when interpreting the results.

- Simulation-only evaluation: All results were obtained in simulation, on a realistic microscopic traffic simulator, and no field tests on real roads were performed. Aspects that are not modeled in simulation, such as real driver behavior, the detailed behavior of the battery and of the charging hardware, or communication delays, are therefore not reflected in the results.
- Single-network validation: All simulations were conducted on a single road network (Pavia) with a single random seed for background traffic (TRAFFIC_SEED = 123), chosen arbitrarily at the beginning of the project. The scalability projections in Section 6.6 can be extrapolated to larger networks in theory, but no empirical validation on a second real-world network was performed. Topological differences, such as a denser grid layout or a different mix of arterial and residential roads, could alter both the cost of rerouting and the availability of alternative routes, which in Scenario 4b was the decisive factor.
- Scenario coverage, not statistical replication: The test scenarios in Section 7.2 were designed to demonstrate specific, identifiable system behaviors rather than to broadly sample the space of possible task configurations. Each scenario was run once; no repeated trials across different random traffic realizations (i.e. different values of TRAFFIC_SEED) were performed to establish confidence intervals on the reported metrics. The qualitative conclusions are therefore better supported than any specific quantitative figure derived from a single run. In addition, Scenarios 1 and 2 were run before the correction of the partition floor; since the state of charge never approached the floor in these runs, their results are not affected.
- Computational measurements: The timing measurements depend on the hardware and on the background load of the machine on which the simulation was run; for example, the mean cost of a periodic rerouting call ranged from 82 ms to 115 ms across the runs of Chapter 7. The absolute values should therefore be interpreted as indicative of the order of magnitude.
- Synthetic background traffic: Background traffic is generated randomly rather than being derived from real historical traffic counts for Pavia, and in most runs its density is low (about 100 vehicles over the whole simulation). The congestion patterns produced in this way are plausible, but they have not been validated against observed real-world congestion. This limits the extent to which the reported rerouting behavior can be claimed to represent real urban conditions.
- Driver-compliance assumption: The evaluation implicitly assumes that EGO adheres to the scheduled plan and the PID-controlled speed profile precisely. However, no model is included for driver override, distraction or non-compliance with a suggested V2G stop, which would be necessary for a deployed system.

9.4 Final Remarks

This thesis has shown that the everyday stops of an electric vehicle, planned around the activities of its driver, can be turned into V2G opportunities without compromising the driver's schedule or the energy needed for driving. The platform developed in this work combines task scheduling,

battery partitioning, traffic-aware routing and dynamic replanning in a single closed loop with a microscopic traffic simulator, and the simulation campaign confirmed that these components work together as designed in the scenarios considered.

At the same time, the tests showed that the quality of the results depends largely on the quality of the information available to the system: Euclidean estimates underestimate the energy needed for driving, and congestion cannot be detected where no probe vehicles are present. These findings indicate that the most promising next steps are not new algorithms, but better inputs, such as road-network distances, richer traffic data and a more detailed simulation of the time spent at the stations. The modular structure of the platform, with its centralized configuration and detailed logging, provides a solid basis for these developments and for the extension to fleets of vehicles and time-varying V2G tariffs.

Bibliography

[1] Q.-S. Jia, H. Panetto, M. Macchi, S. Siri, G. Weichhart, and Z. Xu, "Control for smart systems: Challenges and trends in smart cities," *Annual Reviews in Control*, vol. 53, pp. 358–369, 2022.

[2] I. Mavlutova, D. Atstaja, J. Grasis, J. Kuzmina, I. Uvarova, and D. Roga, "Urban transportation concept and sustainable urban mobility in smart cities: A review," *Energies*, vol. 16, no. 8, p. 3585, 2023.

[3] International Energy Agency, "Global EV outlook 2024," IEA, Paris, Tech. Rep., 2024. [Online]. Available: https://www.iea.org/reports/global-ev-outlook-2024

[4] Ministero dell'Ambiente e della Sicurezza Energetica, "Piano nazionale integrato per l'energia e il clima (PNIEC) 2023," Rome, Italy, 2023.

[5] N. Daina, A. Sivakumar, and J. W. Polak, "Modelling electric vehicles use: A survey on the methods," *Renewable and Sustainable Energy Reviews*, vol. 68, pp. 447–460, 2017.

[6] International Organization for Standardization, "ISO 15118-20:2022 – road vehicles – vehicle to grid communication interface – part 20: 2nd generation network layer and application layer requirements," ISO, Geneva, 2022.

[7] N. Sacchi, G. Basile, S. Siri, and A. Ferrara, "Dispatch and control of connected autonomous electric vehicles for grid support and traffic efficiency in smart cities," 2026, unpublished manuscript.

[8] N. Sacchi, G. Basile, S. Siri, M. Minetti, A. Bonfiglio, and A. Ferrara, "Optimal dispatch of connected and autonomous electric vehicles to enhance short-term grid flexibility in smart cities," in *Proceedings of the IEEE International Conference on Intelligent Transportation Systems (ITSC)*, Naples, Italy, Sep. 2026.

[9] W. Kempton and J. Tomić, "Vehicle-to-grid power fundamentals: Calculating capacity and net revenue," *Journal of Power Sources*, vol. 144, no. 1, pp. 268–279, 2005.

[10] ——, "Vehicle-to-grid power implementation: From stabilizing the grid to supporting large-scale renewable energy," *Journal of Power Sources*, vol. 144, no. 1, pp. 280–294, 2005.

[11] E. Sortomme and M. A. El-Sharkawi, "Optimal combined bidding of vehicle-to-grid ancillary services," *IEEE Transactions on Smart Grid*, vol. 3, no. 1, pp. 70–79, 2012.

[12] O. Štogl, M. Miltner, C. Zanocco, M. Traverso, and O. Starý, "Electric vehicles as facilitators of grid stability and flexibility: A multidisciplinary overview," *Wiley Interdisciplinary Reviews: Energy and Environment*, vol. 13, no. 5, p. e536, 2024.

[13] Open Charge Alliance, "Open charge point protocol 2.0.1 – technical specification," https://openchargealliance.org/download-ocpp/, 2020.

[14] Autorità di Regolazione per Energia Reti e Ambiente (ARERA), "Deliberazione 5 maggio 2017, 300/2017/r/eel – prima apertura del mercato per il servizio di dispacciamento (MSD) alla domanda elettrica e alle unità di produzione anche da fonti rinnovabili non già abilitate," 2017.

[15] ——, "Deliberazione 2 agosto 2018, 422/2018/r/eel – approvazione del regolamento, predisposto da terna s.p.a. ai sensi della deliberazione dell'autorità 300/2017/r/eel, relativo al progetto pilota per la partecipazione di unità virtuali miste al mercato per il servizio di dispacciamento (MSD)," 2018.

[16] Ministero dello Sviluppo Economico, "Decreto 30 gennaio 2020 – criteri e modalità per favorire la diffusione della tecnologia di integrazione tra i veicoli elettrici e la rete elettrica, denominata vehicle to grid," Gazzetta Ufficiale della Repubblica Italiana, n. 37, 14 Feb. 2020, 2020.

[17] Gestore dei Servizi Energetici (GSE), "Vehicle to grid – procedura informativa a tutela dei detentori dei veicoli elettrici," GSE, Rome, 2020.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[19] M. W. P. Savelsbergh, "Local search in routing problems with time windows," *Annals of Operations Research*, vol. 4, pp. 285–305, 1985.

[20] G. A. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958.

[21] M. Schneider, A. Stenger, and D. Goeke, "The electric vehicle routing problem with time windows and recharging stations," *Transportation Science*, vol. 48, no. 4, pp. 500–520, 2014.

[22] D. Krajzewicz, G. Hertkorn, C. Rössel, and P. Wagner, "SUMO (Simulation of Urban MObility): An open-source traffic simulation," in *Proceedings of the 4th Middle East Symposium on Simulation and Modelling (MESM)*, 2002, pp. 183–187.

[23] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic traffic simulation using SUMO," in *Proceedings of the 21st IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2018, pp. 2575–2582.

[24] S. Krauß, "Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics," Ph.D. dissertation, Universität zu Köln, 1998.

[25] P. G. Gipps, "A behavioural car-following model for computer simulation," *Transportation Research Part B: Methodological*, vol. 15, no. 2, pp. 105–111, 1981.

[26] J. Erdmann, "SUMO's lane-changing model," in *Modeling Mobility with Open Data*, ser. Lecture Notes in Mobility. Springer, 2015, pp. 105–123.

[27] OpenStreetMap contributors, "OpenStreetMap," https://www.openstreetmap.org, 2024.

[28] A. Galassi, "Traffic-simulated driving assistance in a rehabilitative framework," Master's thesis, Department of Industrial and Information Engineering, Università degli Studi di Pavia, 2026.

[29] D. Faiola, "Microscopic traffic simulation and patient-in-the-loop longitudinal control in a driving rehabilitation context," Master's thesis, Department of Industrial and Information Engineering, Università degli Studi di Pavia, 2026.

[30] M. B. Arias, M. Kim, and S. Bae, "Prediction of electric vehicle charging-power demand in realistic urban traffic networks," *Applied Energy*, vol. 195, pp. 738–753, 2017.

[31] J. Bischoff and M. Maciejewski, "Simulation of city-wide replacement of private cars with autonomous taxis in Berlin," *Procedia Computer Science*, vol. 83, pp. 237–244, 2016.

[32] M. Gendreau, F. Guertin, J.-Y. Potvin, and É. Taillard, "Parallel tabu search for real-time vehicle routing and dispatching," *Transportation Science*, vol. 33, no. 4, pp. 381–390, 1999.

[33] A. Bonfiglio, M. Minetti, and R. Procopio, "Vehicle-to-home service via electric vehicle energy storage virtual partitioning," *IEEE Transactions on Industry Applications*, 2025.

[34] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1–2, pp. 83–97, 1955.

[35] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

Appendix A

System Parameter Reference

Table A.1 lists the main configurable parameters of the platform, defined in config.py, with their default values, units and a brief description. Two planning constants are defined directly in scheduler.py: the average driving speed used by the scheduler (_AVG_SPEED_MS = 8.33 m/s, i.e. 30 km/h) and the walking speed (_WALK_SPEED_MS = 5.0/3.6 m/s); the size limit of the exact permutation search (N = 8) is also fixed in scheduler.py.

Table A.1: Main configurable parameters (config.py).

Parameter	**Default**	**Unit**	**Description**
Simulation and EGO vehicle			
SIM_BEGIN	0	s	Simulation start time
STEP_LENGTH	0.5	s	SUMO simulation step duration
EGO_DEPART	150	s	EGO departure time after simulation start
EGO_INITIAL_SPEED	80/3.6	m/s	EGO speed at insertion (80 km/h)
EGO_START_EDGE_ID	'57733293'	–	SUMO edge ID of the EGO starting position
EGO_END_EDGE_ID	'26585503'	–	SUMO edge ID of the final destination
Battery and V2G			
BATTERY_CAPACITY_KWH	60.0	kWh	Total usable battery capacity
BATTERY_INITIAL_SOC	1.00	–	State of charge at simulation start
BATTERY_MIN_SOC	0.20	–	Absolute minimum SoC (conservative design limit)
BATTERY_CONSUMPTION_KWH_PER_KM	0.20	kWh/km	Energy consumption at urban speeds
BATTERY_DRIVE_SAFETY_MARGIN	0.30	–	Relative margin added to the driving energy estimate
ELECTRICITY_PRICE_EUR_KWH	0.50	€/kWh	Reference retail recharge price
V2G_FEED_IN_TARIFF_PCT_OF_PRICE	0.20	–	V2G credit as a fraction of the electricity price
V2G_FEED_IN_TARIFF_EUR_KWH	0.10	€/kWh	V2G tariff, derived as price × fraction
Routing			
EGO_REROUTE_PERIOD	5.0	s	Periodic rerouting interval

Table A.1 (continued)

Parameter	Default	Unit	Description
EGO_EVENT_CHECK_INTERVAL	2.5	s	Event-driven congestion check interval
EGO_EVENT_LOOKAHEAD_DISTANCE	100.0	m	Lookahead distance of the congestion scan
EGO_EVENT_CONGESTION_THRESHOLD	0.50	–	Speed/limit ratio below which an edge is congested
EGO_LOW_SPEED_THRESHOLD	4.0	m/s	Observed speed below which the low-speed penalty applies
EGO_LOW_SPEED_PENALTY	4.0	–	Multiplicative penalty on the travel time of slow edges
EGO_TRAFFIC_WEIGHT_HORIZON	500.0	s	Validity window of the adapted edge travel times
EGO_ENERGY_ROUTING_LAMBDA	50.0	s/kWh	Energy-to-time conversion factor of the routing cost
Replanning			
REPLAN_CHECK_INTERVAL_S	60.0	s	Minimum time between replanning events (cooldown)
REPLAN_DELAY_THRESHOLD_S	300.0	s	Estimated delay that triggers in-flight replanning
Speed control			
EGO_SPEED_KP / KI / KD	0.07 / 0.03 / 0.01	–	PID gains
EGO_SPEED_INTEGRAL_LIMIT	30.55	m/s	Anti-windup bound of the integral term
EGO_SPEED_COMMAND_STEP	0.5	m/s	Maximum change of the commanded speed per step
EGO_MIN_SPEED	0.0	m/s	Minimum commanded speed
EGO_NORMAL_TARGET_SPEED / MAX_SPEED	13.9 / 16.66	m/s	Target and maximum speed on normal roads (≈ 50 / 60 km/h)
EGO_FAST_TARGET_SPEED / MAX_SPEED	27.7 / 30.55	m/s	Target and maximum speed on fast roads (≈ 100 / 110 km/h)
EGO_FOLLOWING_DISTANCE	20.0	m	Gap below which the following-distance override applies
EGO_FOLLOWING_LOOKAHEAD	100.0	m	Scan range for the leading vehicle
Background traffic and congestion injection			
TRAFFIC_PERIOD	3	s	Minimum period between background vehicle departures
TRAFFIC_SEED	123	–	Random seed for background traffic generation
BG_VEHICLE_COUNT	100	–	Number of background vehicles spread over the simulation
CONGESTION_ENABLED	False	–	Enables the injected congestion (testing only)
CONGESTION_EDGE_IDS	[ ]	–	Edges whose maximum speed is reduced
CONGESTION_START_S	1000.0	s	Simulation time at which the congestion starts

Table A.1 (continued)

Parameter	Default	Unit	Description
CONGESTION_SPEED	1.5	m/s	Imposed maximum speed on the congested edges
Walking			
WALK_SPEED_KMH	5.0	km/h	Pedestrian speed for walking time estimation
MAX_WALK_DISTANCE_M	800.0	m	Reference maximum walking distance (not enforced by the scheduler)

Appendix B

Function Quick Reference

Table B.1 lists the main functions of the platform, grouped by module, with their signature and a one-line description. Private helper functions used only internally are omitted.

Table B.1: Main functions of the platform.

Function	Description
main_ecosmos.py	
main()	Top-level pipeline: stations → task planning → simulation → plots → network analysis
_configure_charging_stations(net_path, workdir, net)	Reload the saved stations or open the station GUI
_run_task_planning(net_path, net, stations, workdir)	Task GUI, scheduler and confirmation dialog, repeated until the schedule is confirmed
_show_schedule_summary(result)	Confirmation window with the optimized schedule; returns True if the user starts the simulation
_show_v2g_end_summary(total_energy_kwh, total_discount_eur, final_soc)	End-of-day window with the planned V2G balance and the final SoC
_normalize_task_times(tasks)	Convert wall-clock target times to simulation seconds
_compute_sim_end(tasks, buffer_s=600.0)	Estimate the simulation end time
_get_start_position(net)	(x, y) centroid of the EGO start edge
scheduler.py	
optimize_task_schedule(tasks, start_x, start_y, depart_time, stations)	Entry point of the scheduler: returns a ScheduleResult
assign_tasks_to_stations(tasks, stations)	Greedy nearest-station assignment
_evaluate_order(order, assignments, sx, sy, depart)	Evaluate one visit order: timing, delay, tasks on time and V2G energy
_greedy_order(tasks, sx, sy)	Earliest-deadline-first initial order
_two_opt_improve(order, assignments, sx, sy, depart, ...)	2-opt local search refinement
_build_suggestions(...)	Human-readable description of the schedule decisions
_travel_time(x1, y1, x2, y2), _walk_time(dist_m)	Euclidean driving and walking time estimates
simulation_ecosmos.py	

Table B.1 (continued)

Function	Description
run_simulation_with_tasks(workdir, tasks, sim_end, net, stations, min_soc)	Main simulation entry point; returns the final SoC
_write_sumocfg_for_tasks(workdir, sim_end, cs_add_file)	Generate the .sumocfg configuration file
_inject_ego_first_segment(first_task, edge_by_id)	Compute the initial route and insert EGO
_same_station_chain(tasks, from_idx)	Count consecutive tasks at the same station
_add_task_stop(task, edge_by_id, duration_override)	Register the SUMO stop with the three-level fallback
_reroute_to_waypoint(current_edge_id, dest_edge_id, edge_by_id, speed_by_edge, sim_time, energy_weight)	Update the edge costs, compute and apply a new route
_check_congestion_ahead(remaining_route, edge_by_id, event_speed_by_edge)	Return the congested edges in the lookahead window
_try_replan(tasks, next_task_idx, sim_time, ego_x, ego_y, stations, reason)	Re-optimize the remaining tasks; returns True if the order changed
simulation.py	
apply_traffic_aware_travel_times(edge_by_id, speed_by_edge, sim_time, energy_weight)	Write the traffic and energy edge costs to SUMO
apply_ego_speed_profile(vehicle_id, edge_id, edge_by_id)	Select the speed profile for the current road type
control_ego_speed(vehicle_id, target_speed, kp, ki, kd, min_speed, max_speed)	Incremental PID speed controller
get_following_distance_speed(vehicle_id)	Speed cap based on the gap to the leading vehicle
plots.py and graph_analysis.py	
make_plots(workdir)	EGO speed and acceleration, space-time diagram and fundamental diagrams
make_soc_plot(workdir, floor)	SoC history with the partition floor
make_complexity_plots(workdir)	Computational cost comparison and scalability figures
run_networkx_analysis(workdir, workdir2)	NetworkX consistency check, route snapshots and speed animation
Stations, tasks and utilities	
run_charging_station_gui(net_path, workdir, net)	Interactive map for station placement
load_stations(workdir), save_stations(stations, workdir)	Load and save charging_stations.json
write_charging_stations_additional(stations, workdir, net)	Write the SUMO charging station XML file
run_task_selection_gui(net_path, net, calendar_csv)	Task-placement GUI; loads the calendar and saves calendar-origin tasks on confirm
save_calendar_tasks(tasks, csv_path)	Write the calendar-origin tasks to calendar_tasks.csv
clamp(value, lower_bound, upper_bound)	Clamp a value to an interval
lane_to_edge_id(lane_id)	Strip the lane suffix to obtain the edge ID
euclidean_distance(point_a, point_b)	Euclidean distance between two points

Appendix C

Implementation Details

This appendix collects some implementation details of the platform described in Chapter 3: the files of the working directory, the shared state variables and the schema of the floating car data output.

Table C.1: Working directory file structure.

File	Description
nave_2.net.xml	SUMO road network - Pavia OSM, converted by netconvert
charging_stations.json	Charging station configuration - persisted between runs
charging_stations.add.xml	SUMO additional file - charging station definitions
edge_data.add.xml	SUMO additional file - edge traffic data collector
sim.sumocfg	Generated SUMO configuration for the current run
fcd.csv	Floating Car Data - per-step position/speed for all vehicles
reroute_timing.csv	Rerouting event log - wall-clock timing and edge counts
ego_route_history.csv	All the routes adopted by EGO, with their simulation time
tripinfo.xml	SUMO per-vehicle trip statistics
calendar_tasks.csv	Pre-planned task calendar (name, edge_id, priority, time, duration), persisted between runs; calendar-origin tasks only (Section 3.12)

Table C.2: Shared state variables in state.py.

Variable	Type	Content
EGO_ROUTE_EDGE_IDS	list[str]	Current planned route as edge ID list. Updated at each reroute.
EGO_ROUTE_HISTORY	list	All route versions, with the simulation time at which each was adopted.
EGO_SPEED_CONTROL_STATE	dict	PID controller state: integral error, previous error and last command.
EGO_LAST_ACTIVE_TIME	float	Last simulation time at which EGO was in the network.
PERIODIC_REROUTE_LOG	list[dict]	Per call: {sim_time, duration_s, n_remaining_edges}.
EVENT_REROUTE_LOG	list[dict]	Per check: {sim_time, duration_s, n_remaining_edges, triggered, congested_edges, congested_count}.
REPLAN_TRAFFIC_STATES	list[dict]	Per replan: {sim_time, reason, new_order}.
SOC_HISTORY	list[tuple]	(sim_time, SoC) pairs, recorded every 10 simulation steps (5 s).

Table C.3: FCD output schema (fcd.csv)

Column	Type	Description
timestep_time	float	Simulation time [s]
vehicle_id	str	SUMO vehicle ID
vehicle_x	float	Position x-coordinate [m] in SUMO coordinate system
vehicle_y	float	Position y-coordinate [m]
vehicle_angle	float	Vehicle heading [degrees], 0 = North
vehicle_speed	float	Current speed [m/s]
vehicle_pos	float	Position along the lane [m]
vehicle_lane	str	Current lane ID
vehicle_slope	float	Road slope at current position [%]
vehicle_type	str	SUMO vehicle type ID

Acknowledgements

My sincere thanks go to Prof. Antonella Ferrara, who gave me the chance to work on this thesis and followed its development with competence and attention. I am grateful for the confidence she showed in my work and for the autonomy she granted me, which allowed me to grow both technically and personally.

I am equally grateful to Dr. Nikolas Sacchi, whose availability, clear explanations and constructive feedback were a constant point of reference during these months, and who always found the time to discuss problems and new ideas with me.

A heartfelt thank you goes to my family, who supported me with patience and encouragement throughout my studies and in every choice I made along the way. This milestone is also theirs.

I would also like to thank my friends and the colleagues I met during these years at the University of Pavia, with whom I shared lectures, exams and many good memories.

Finally, I want to thank Davide and Alessandro, my companions in the first phase of this work. Building the simulation platform together, between long debugging sessions and many laughs, was one of the most enjoyable parts of this experience. I wish them the best of luck for whatever comes next.