# OraOra Rail: Interactive Railway Timetable Visualizer

Intelligent Agent - Automated Planning module

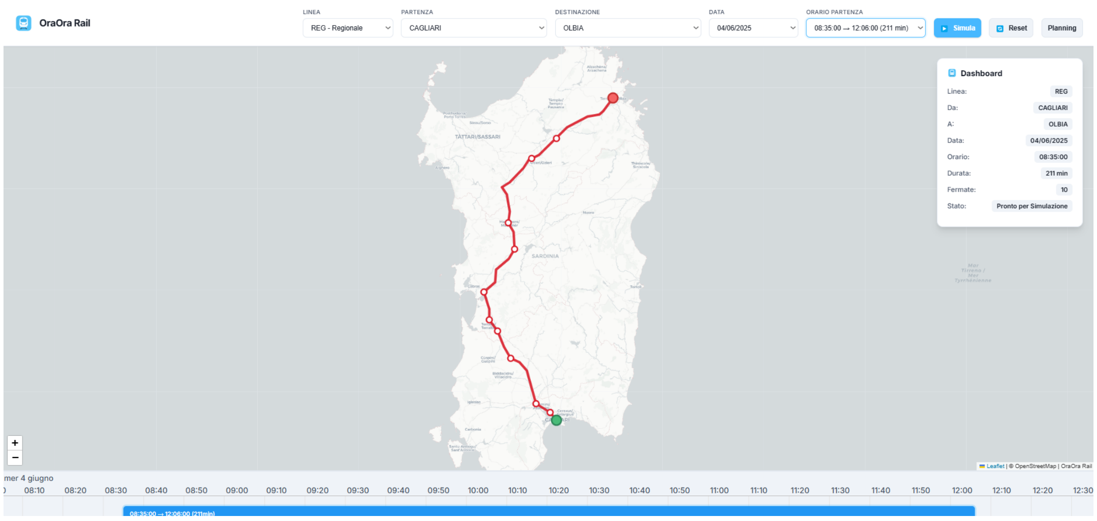
Ilaria Frandina
Student ID: 264232

July 2025

## 1 Introduction

This report presents **OraOra Rail**, a complete system for the automatic visualization and planning of public transport itineraries. The project evolves from a simple railway timetable viewer into a system for planning trips using Trenitalia data for the Sardinian train and bus network.

The system architecture includes two main components: a Flask backend that manages data and performs planning calculations and a JavaScript frontend for interactive visualization. The system supports both the display of direct routes (trains and buses) and the automatic planning of complex itineraries. Said planning introduces changes between various modes of transport and permits optimization for either travel time or number of changes.



| LINEA | PARTENZA | DESTINAZIONE | DATA | ORARIO PARTENZA |
|---|---|---|---|---|
| REG - Regionale | CAGLIARI | OLBIA | 04/06/2025 | 08:35:00 → 12:06:00 (211 min) |

| | |
|---|---|
| Linea: | REG |
| Da: | CAGLIARI |
| A: | OLBIA |
| Data: | 04/06/2025 |
| Orario: | 08:35:00 |
| Durata: | 211 min |
| Fermate: | 10 |
| Stato: | Pronto per Simulazione |

Figure 1: OraOra Rail main interface with selection controls, interactive map and information dashboard

## 2 System Architecture and Technology Stack

OraOra Rail is built on a three-tier architecture that clearly separates data preprocessing, business logic and presentation. The architecture includes a planning layer that implements search algorithms for solving routing problems in transport.

## 2.1 Data prepocessing

The **preprocess.py** module represents the fundamental step of transforming raw data into structures optimized for automatic scheduling and visualization.

The final result of preprocessing is a set of optimized JSON files: routes.json contains line information with modes and colors, stops.json stops with geographic coordinates, shapes.json with geographic routes, timeable.json containing all trips with structured schedules, and calendar.json with the dates that serve to determine when each service is active. This transformation significantly reduces loading times and parsing during planning operations.

## 2.2 Backend Flask and Planning System

The backend, implemented in Python with Flask, manages the processing and manipulation of raw data, and provides REST APIs for automatic scheduling and retrieval of preprocessed information.

```python
@app.route("/plan", methods=["POST"])
def plan_endpoint():
    data = request.get_json(force=True)
    origin = data.get("origin")
    destination = data.get("destination")
    date = data.get("date")
    optimize = data.get("optimize", "time")  # 'time' o 'transfers'

    res = planner.plan(origin, destination, date, depart_after, optimize)
    return jsonify(res)
```

Listing 1: POST type Rest service made available by the backend for planning with parametric optimization type

### 2.2.1 Planning algorithms

The automatic scheduling system contained in **planner.py** implements two algorithms, each optimized for a different criteria: minimum travel time and minimum number of changes.

The basic idea is quite simple: the system considers every possible move from one stop to the next as a "step" that can be taken, taking into account the real timetables of the vehicles. For each stop it keeps track of when you can get there and by what means, so you know if it is possible to take a subsequent connection. The change of vehicle occurs only when you move from one trip (trip-id) to a different one.

The **Earliest Arrival** algorithm implements a search that maintains the best arrival time achieved up to that point for each stop. The Label data structure represents the state of exploration and contains the arrival time, the number of changes made, the last trip used and the pointers to reconstruct the route. When chronologically scanning connections, the algorithm checks whether a new connection can improve the time to arrive at a stop; considering that there must be sufficient time to possibly make a vehicle change. The exchange rate count occurs by comparing the identifier of the current trip with the previous one: if they are different, the transfer counter is increased.

```python
def _plan_earliest_arrival(self, conns: List[Connection], origin: str,
    destination: str, dep_after: int) -> Optional[Label]:
        best_arrival: Dict[str, Label] = {}
        best_arrival[origin] = Label(arr_time=dep_after, transfers=0, last_trip=
    None, prev=None, reached_by=None, stop=origin)

        for c in conns:
            if c.dep_time < dep_after:
                continue
            if c.dep_stop not in best_arrival:
                continue
            cur_label = best_arrival[c.dep_stop]
            if cur_label.arr_time <= c.dep_time:
                if (c.arr_time < best_arrival.get(c.arr_stop, Label(math.inf, 0, None, None, None, c.arr_stop)).arr_time):
                    new_label = Label(
                        arr_time=c.arr_time,
                        transfers=cur_label.transfers + (0 if cur_label.last_trip == c.trip_id else 1 if cur_label.last_trip is not None else 0),
                        last_trip=c.trip_id,
                        prev=cur_label,
                        reached_by=c,
                        stop=c.arr_stop
                    )
                    best_arrival[c.arr_stop] = new_label

        return best_arrival.get(destination)
```

Listing 2: Earliest Arrival Algorithm

The **Minimum Transfers** algorithm works differently because for each stop it retains multiple possible solutions instead of just one. The idea is simple: one solution is better than another if it has fewer changes and arrives earlier (or at least not worse in both aspects). This allows the system to find itineraries that could be more comfortable for those who prefer to make fewer changes even if it takes a little longer. To prevent the number of solutions from growing too much and slowing down the system, the algorithm constantly eliminates those that are clearly worse, maintaining only those that offer a good compromise between speed and convenience.

```python
def _plan_min_transfers(self, conns: List[Connection], origin: str, destination: str, dep_after: int) -> Optional[Label]:
        labels: Dict[str, List[Label]] = defaultdict(list)
        labels[origin].append(Label(dep_after, 0, None, None, None, origin))

        def dominated(new_l: Label, existing: List[Label]) -> bool:
            for l in existing:
                if l.transfers <= new_l.transfers and l.arr_time <= new_l.arr_time:
                    return True
            return False

        for c in conns:
            if c.dep_time < dep_after:
                continue

            for lab in list(labels[c.dep_stop]):
                if lab.arr_time <= c.dep_time:
                    transfers = lab.transfers + (0 if lab.last_trip == c.trip_id else 1 if lab.last_trip is not None else 0)
                    new_label = Label(
                        arr_time=c.arr_time,
                        transfers=transfers,
                        last_trip=c.trip_id,
                        prev=lab,
                        reached_by=c,
                        stop=c.arr_stop
                    )
                    if not dominated(new_label, labels[c.arr_stop]):
                        labels[c.arr_stop] = [l for l in labels[c.arr_stop] if not (new_label.transfers <= l.transfers and new_label.arr_time <= l.arr_time)]
                        labels[c.arr_stop].append(new_label)

        if not labels[destination]:
            return None

        return min(labels[destination], key=lambda l: (l.transfers, l.arr_time))
```

Listing 3: Minimum Transfers Algorithm

Both algorithms share the path reconstruction phase through backtracking on the pointers contained in the labels. The **reconstruct** method retraces the label chain from destination to origin, building the sequence of connections that form the optimal itinerary. The system naturally manages multimodality through various modes of transport, allowing itineraries that combine regional trains and buses.
The final output of the planning is a complete JSON structure that includes: total time, number of changes, type of algorithm used, the detailed sequence of segments with times and stops. This structured information feeds both the cartographic visualization and the animation of the route, providing all the details necessary for a complete representation of the planned itinerary.

### 2.3 Frontend JavaScript Modular

The frontend is organized into JavaScript modules which it handles separately:

- **DataManager**: Loading and managing data
- **MapManager**: Interactive map view with Leaflet
- **TimelineManager**: Travel timeline with Vis.js
- **AnimationManager**: Train and bus entertainment along the routes
- **UIManager**: Interface management and application states
- **PlanningManager**: Planning interface

## 3 User Interface and Operating Modes

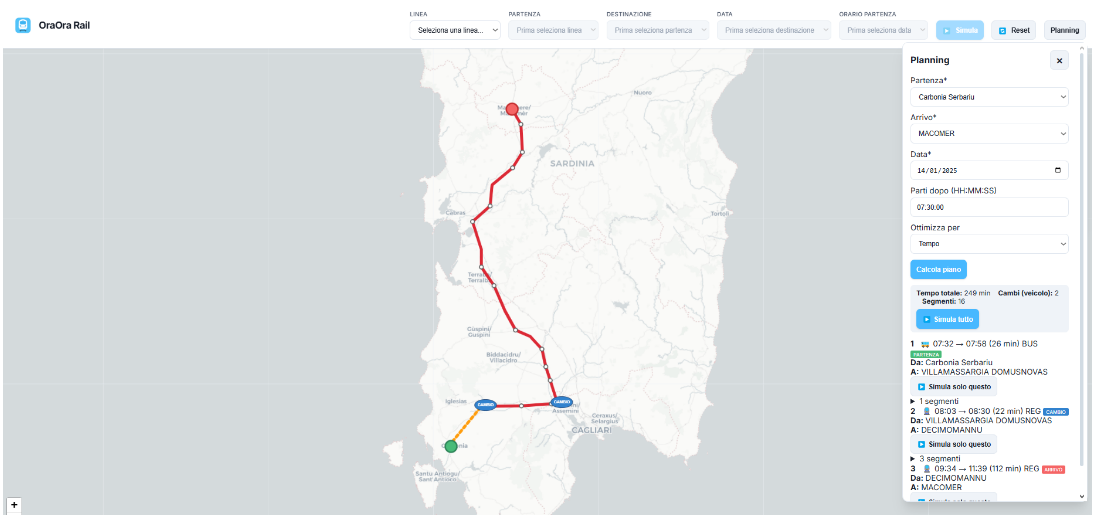
The system features two distinct operating modes that adapt to different use scenarios.

### 3.1 Classic View Mode

In classic mode you need to enter, in sequence, the following data: line (BUS or REG), start, destination, date, time (only the times actually present for the selected date are shown) for displaying direct paths, as shown in Figure 1.

### 3.2 Planning Mode

The planning mode introduces a dedicated panel for planning trips between any station present:



Planning

Partenza*: Carbonia Serbariu

Arrivo*: MACOMER

Data*: 14/01/2025

Parti dopo (HH:MM:SS): 07:30:00

Ottimizza per: Tempo

Calcola piano

**Tempo totale:** 249 min **Cambi (veicolo):** 2
**Segmenti:** 16

Simula tutto

1 07:32 → 07:58 (26 min) BUS PARTENZA
**Da:** Carbonia Serbariu
**A:** VILLAMASSARGIA DOMUSNOVAS
Simula solo questo
▶ 1 segmenti

2 08:03 → 08:30 (22 min) REG CAMBIO
**Da:** VILLAMASSARGIA DOMUSNOVAS
**A:** DECIMOMANNU
Simula solo questo
▶ 3 segmenti

3 09:34 → 11:39 (112 min) REG ARRIVO
**Da:** DECIMOMANNU
**A:** MACOMER

Figure 2: Interface for itineraries with Planning mode (the BUS line in orange - the TRAIN line in red)

The planning panel offers:

- Free selection of start and destination from all available stations
- Configuration of minimum departure date and time
- Choice of optimization criteria (time vs. number of transfers)
- Structured display of results with details for each route
- Possibility to simulate the entire itinerary or individual routes (as for viewing direct routes)

## 4 Results and Conclusions

The system successfully manages:

- Direct itineraries on individual lines (trains and buses)
- Itineraries with changes between trains and buses
- Optimization for minimum time or minimum number of transfers
- Visualization and animation of routes

The strong point of the project is that by loading more data into the resources folder; the system adapts and works on any data set with the same format. This makes the entire system completely scalable.

Furthermore, the system is containerized with Docker to simplify deployment and dependency management. Contained in the folder `docker` are the Dockerfile and the docker-compose file for the full project start.