Politecnico di Torino

Politecnico di Torino

Computer Engineering

A.a. 2023/2024

Graduation Session March , 2026

Development of tool for visualizing open data about Italian public trains performance

Supervisors:
Antonio Vetrto'
Andrea Trentini
Enea Ahmedhodzic

Candidate:
Sayedali Noohi

Abstract

This thesis presents the design and implementation of a web application for railway performance analysis, developed as an extension of an existing open-data project.

The base *Railway OpenData* repository already provided data scraping, extraction, and analysis capabilities through command-line tools. The main contribution of this work is the development of a complete web platform that makes those data and analyses accessible to non-technical users through interactive visualizations and guided filtering.

The implemented system includes a FastAPI backend, a React single-page frontend, dataset upload and archival workflows, monthly precomputed analytics, and custom-range analysis. The user interface integrates a shared filtering model (date range, railway operator, regions, and stations), map-based station discovery, and statistical charts for delay distributions and traffic volumes. On the backend side, input validation, date-range safeguards, and robust serialization improve reliability for interactive usage.

The result is a practical research prototype that bridges technical data pipelines and citizen-facing exploration of railway service performance, while explicitly accounting for upstream data quality and completeness constraints.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Antonio Vetrò, for his guidance, support, and constructive feedback throughout this thesis work. His expertise and availability were essential in shaping both the direction of the project and the quality of the final results.

I would also like to thank the Department of Control and Computer Engineering (DAUIN) at Politecnico di Torino for providing an excellent academic environment and the resources that supported this research activity.

Finally, I would like to thank my family and friends for their patience, encouragement, and continuous support during the months dedicated to this work.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Despite the central role of public rail transport, quantitative information about the *operational performance* of Italian trains—such as delays, cancellations, and service reliability—is not provided as official open data in a way that is readily accessible for citizens, researchers, and local institutions. In practice, performance indicators are difficult to explore and communicate when data collection and analysis are available only through scripts and command-line tools.

From a civic-technology and service-quality perspective, this creates a practical gap: users may perceive problems in daily commuting, yet lack a transparent and reproducible way to quantify them across time and geography. A web-based exploration tool can reduce friction for basic questions ("How has punctuality changed this month?" "Which stations are most affected?"), while preserving the ability to validate results through the underlying dataset.

The *Railway OpenData* project addresses the data availability problem by collecting live train information from public endpoints (notably *ViaggiaTreno* and *Trenord*), preserving raw snapshots, and extracting daily datasets suitable for downstream analysis. However, a second gap remains: even when data exist, meaningful exploration is limited without an interactive interface that supports filtering, map-based navigation, and repeatable comparisons over time.

1.1.1 Accessibility gap

Non-technical users typically need a web interface to:

- inspect delays and traffic volumes without running scripts,
- focus analyses on specific regions or stations,

- compare time windows and observe trends over months.

1.1.2 Workflow fragmentation

Prior to this work, the pipeline was fragmented between data processing outputs and final user consumption. Users could generate statistics, but there was no unified interface for selecting filters, visual exploration, and managing dataset versions in a controlled and reproducible way (e.g., during demonstrations or iterative dataset updates).

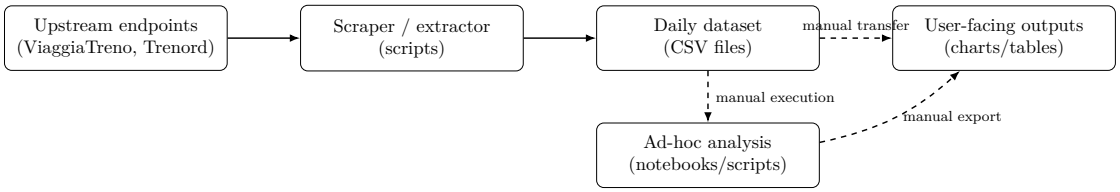


Figure 1.1: Workflow fragmentation before this work. The data pipeline could produce local datasets and ad-hoc analyses, but the path from datasets to user-facing outputs relied on manual execution and transfer, with no shared filtering model or integrated interactive exploration.

These limitations motivate a citizen-facing platform that bridges the existing data pipeline and interactive exploration through a browser, while remaining compatible with the original dataset structure and explicitly acknowledging upstream data quality constraints (missing values, inconsistencies, and incomplete station metadata).

1.1.3 Stakeholders and typical questions

The platform targets multiple stakeholder groups with partially overlapping needs:

- **Citizens and commuters**: quickly inspect punctuality and delay patterns for their lines and stations.
- **Researchers and students**: prototype analyses and validate hypotheses with an inspectable data pipeline.
- **Local institutions and civic groups**: communicate service issues with evidence grounded in observed operational data.

Common questions motivating the design include: how performance varies across regions and operators, which stations experience recurring delays, and whether changes over time are observable in aggregated indicators.

1.2 Research Objectives

This thesis aims to design and implement a web application that extends *Railway OpenData* with an end-to-end user experience for railway performance exploration. The main objectives are:

1. **Interactive access**: Provide an interface that makes railway performance indicators accessible to non-technical users through guided filters and visualizations.
2. **Unified exploration workflow**: Support a consistent filtering model across maps, dashboard summaries, and statistical analysis views.
3. **Performance-oriented analytics**: Combine precomputed monthly aggregates with on-demand analysis for user-defined time windows.
4. **Dataset lifecycle management**: Enable safe dataset updates through upload, archiving, and apply/revert operations for reproducible results.
5. **Robustness**: Improve reliability for interactive usage through input validation, date-range safeguards, and resilient serialization.

1.2.1 Research questions

The implementation is guided by the following research questions:

- **RQ1**: How can operational railway data be made explorable by non-technical users while keeping the analysis traceable and reproducible?
- **RQ2**: Which interaction model enables consistent filtering across spatial exploration (map) and statistical views without confusing users?
- **RQ3**: How can responsiveness be maintained on large datasets by combining precomputed aggregates with bounded on-demand computation?
- **RQ4**: Which dataset lifecycle operations are necessary to safely support updates (upload, archive, apply/revert) in a demonstrable and repeatable way?

1.2.2 Scope and assumptions

This work focuses on *exploration and visualization* of the datasets produced by the existing pipeline. It does not attempt to establish official ground truth about disruptions or causality, nor to replace institutional reporting. Results depend on

the coverage and quality of upstream sources; the system therefore emphasizes transparent metadata, validation, and graceful handling of missing or inconsistent records.

1.3 Method Overview

The approach follows an end-to-end pipeline that connects the existing dataset structure to an interactive web experience:

- an **offline data layer** that stores daily CSV datasets and optionally produces monthly precomputed artifacts;
- a **backend API** that exposes metadata, filtering primitives, statistics endpoints, and dataset lifecycle operations;
- a **frontend web interface** that provides a shared filters panel, map interaction, and statistics pages that reuse the same query parameters.

This separation allows heavy computations to be handled server-side, keeps the UI responsive, and makes deployment reproducible (e.g., through containerization).

1.4 Contributions

The main contributions of this work include:

- **Citizen-facing web platform**: A complete web application that exposes the *Railway OpenData* datasets through interactive maps and charts.
- **Full-stack implementation**: A FastAPI backend and a React single-page frontend integrated around a shared filtering and navigation model.
- **Dual analytics mode**: Monthly precomputed outputs for responsiveness on large datasets, plus custom-range analysis for flexibility.
- **Dataset versioning workflow**: Upload and archival features that support controlled dataset evolution and reproducible analyses.
- **Practical constraints analysis**: A discussion of limitations inherited from upstream data sources and third-party enrichment availability.

1.5 Thesis Structure

This thesis is organized as follows:

Chapter 2 presents background and related work on public-transport performance data, web-based data exploration, and open-data pipelines.

Chapter 3 describes requirements and design, including the data model, user workflows, and the overall architecture of the platform.

Chapter 4 details the implementation of the backend APIs, frontend user interface, and the dataset upload/archival mechanisms.

Chapter 5 reports validation and evaluation of the main user workflows, performance considerations, and observed limitations.

Chapter 6 concludes the thesis with a summary of achievements and directions for future work.

Chapter 2

Background and Related Work

2.1 Railway performance data and public transparency

Public rail transport is a critical service, and its perceived quality strongly depends on the ability to understand *operational performance*: delays, cancellations, service frequency, and how these vary across geography and time. While transport operators may publish timetables and service announcements, performance indicators are often fragmented, difficult to aggregate, and not always distributed as machine-readable open data.

From an analytical perspective, railway performance is naturally described at multiple granularities:

- **Network level**: global trends across regions and operators (e.g., delay distribution over months).
- **Line/relation level**: performance along a corridor connecting two areas.
- **Station/stop level**: punctuality and traffic volume observed at a given station.

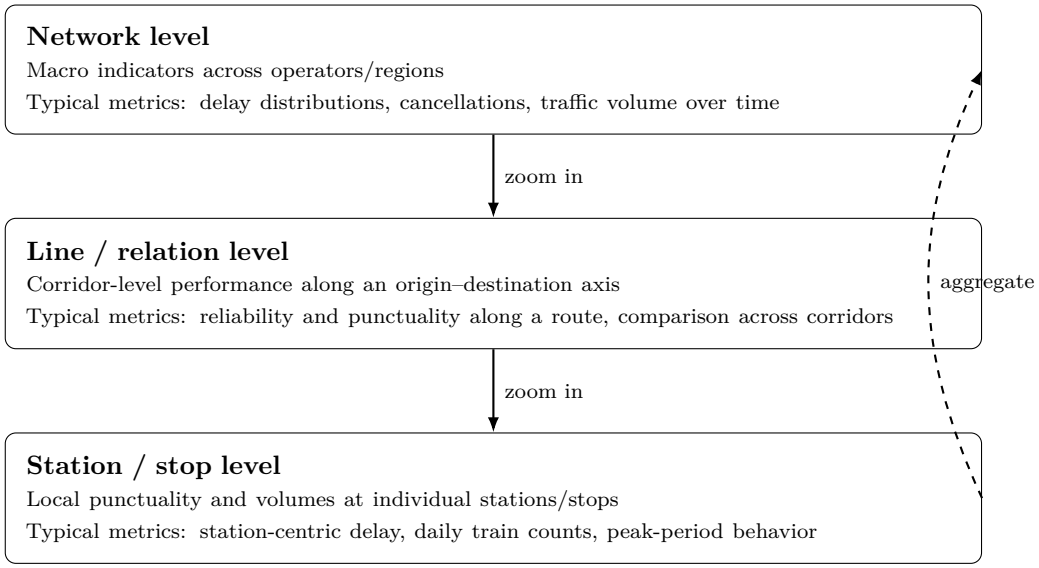


Figure 2.1: Multi-granularity perspective on railway performance. Analyses move between global indicators, corridor-level views, and station-level observations, with aggregation providing the link back to network-wide summaries.

Figure 2.1 summarizes this perspective and motivates interfaces that support consistent navigation and filtering across levels. To support practical exploration, a platform must allow users to move between these levels, apply consistent filters, and compare different time windows.

2.2 Open data and the challenges of operational datasets

In the open-data ecosystem, a dataset is most useful when it is discoverable, well documented, stable over time, and accessible via standard formats. Open-data guidance typically emphasizes availability, reuse, and machine-readability as prerequisites for meaningful downstream applications[1]. Operational datasets introduce additional challenges:

- **High update frequency**: live status can change minute-by-minute, requiring incremental collection.
- **Schema variability**: fields may be missing or inconsistent depending on the service, operator, or station.
- **Identifiers and normalization**: seemingly obvious keys (station codes, train numbers) may not be unique.

- **Legal and redistribution constraints**: terms of service may restrict redistribution even when data are publicly visible.

As a result, many projects adopt a *best-effort* approach: they collect what is available, explicitly track incompleteness, and focus on reproducibility of the extraction and analysis procedures.

2.3 Standard transit data formats

Many public-transport applications are built on standardized formats that separate *planned* service from *realized* operations.

2.3.1 GTFS (static schedules)

The General Transit Feed Specification (GTFS) is widely used to publish scheduled services, including stops, routes, trips, and stop times[2]. It is well suited for journey planning and timetable-based views, but it does not directly capture real-time delays, cancellations, or operational irregularities.

2.3.2 GTFS-Realtime (operational updates)

GTFS-Realtime complements static schedules with near-real-time updates such as trip status, vehicle positions, and service alerts[3]. When available, this format enables richer monitoring and passenger information. However, not all operators publish GTFS-Realtime feeds, and the semantics of "delay" or "cancellation" may differ across providers. Projects that rely on bespoke public endpoints often need to reconstruct operational records and define consistent aggregation rules to produce comparable performance indicators.

2.4 Operational performance indicators

Operational performance is multi-dimensional and depends on the unit of observation (stop, train run, station, corridor). In practice, platforms often expose a small set of indicators that are understandable by non-technical users while remaining computable from imperfect operational data.

Indicator	Interpretation and typical aggregation
Punctuality rate	Share of stops (or trips) with delay below a threshold (e.g., 5 minutes), aggregated by station, operator, or month.
Delay distribution	Distribution of delays summarized by quantiles (e.g., boxplots), robust to outliers and suitable for comparisons.
Cancellation / partial run proxy	Rate of missing or incomplete observations (e.g., expected runs not fully observed), used as a proxy when explicit cancellation flags are unavailable.
Traffic volume	Counts of trains or stop observations per day/month, used to contextualize performance changes.

Table 2.1: Examples of operational KPIs for railway performance exploration. In practice, indicator definitions depend on data availability and require explicit modeling choices when upstream sources are incomplete.

2.5 Italian railway public endpoints

In the Italian context, the *ViaggiaTreno* portal and the *Trenord* services expose public endpoints that can be queried to retrieve live information about train runs and station stops. Although these endpoints are not designed as an official open-data product, they can be used as sources for performance monitoring.

Working with these sources requires awareness of common data-quality and modeling issues, including:

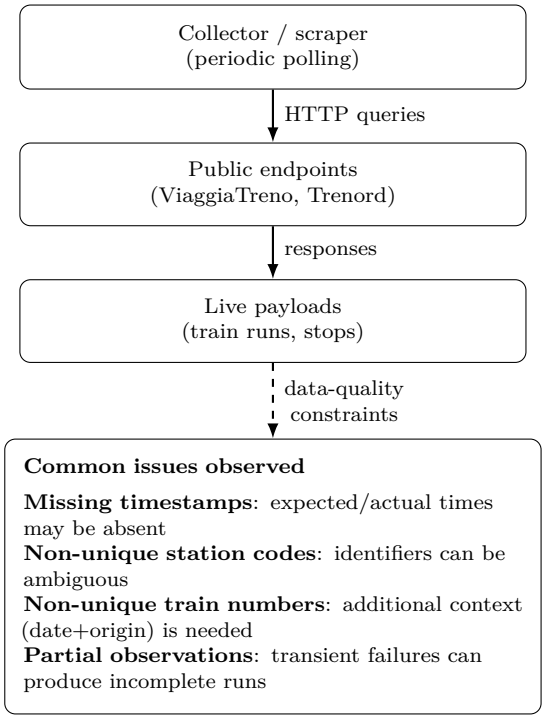


Figure 2.2: Public railway endpoints as data sources and typical pitfalls. Although information is publicly accessible, payloads are not tailored for open-data reuse and often require careful modeling and validation before downstream analysis.

- **Missing timestamps**: expected and actual arrival/departure times may be absent for some stops.
- **Non-unique codes**: station codes may not uniquely identify physical stations.
- **Non-unique train numbers**: two different trains can share the same number; the departing date and origin must be considered.
- **Partial observations**: a train may be only partially fetched due to temporary unavailability; such cases must be flagged.

These constraints shape both the extraction pipeline and the design of the web user interface, since the system must provide meaningful results while handling missing or incomplete fields.

2.6 From scraping to analysis-ready datasets

Data pipelines for operational monitoring typically separate three concerns: collection, normalization, and analysis.

- **Collection**: an unattended scraper periodically queries the upstream endpoints and stores raw snapshots. Persisting raw data is important for reproducibility and for re-running improved extractors over historical captures.

- **Normalization**: extractors convert raw snapshots into structured, analysis-ready formats such as CSV. In the *Railway OpenData* dataset, each extracted row represents a train stop, enabling both station-level and train-level aggregations.

- **Analysis**: statistics and visualizations are produced from normalized datasets, typically requiring grouping and aggregation over trains, stations, dates, and operators.

This architecture supports incremental collection and makes it possible to evolve the analysis layer without changing the upstream data acquisition strategy.

2.7 Web-based exploration of spatio-temporal data

Web applications are a natural medium for making operational datasets accessible to a broad audience. Two interaction patterns are particularly relevant:

- **Map-first exploration**: users discover and select stations spatially, then drill down into station-centric statistics.

- **Dashboard-first exploration**: users start from high-level indicators and progressively constrain the analysis scope through filters.

Both patterns benefit from a consistent filtering model (date ranges, operators, regions, stations) and from visual encodings that fit the problem: box plots for delay distributions, time series or bar charts for traffic volumes, and map markers/layers for spatial context.

Performance is a practical concern when the dataset spans many days and thousands of trains. A common strategy is to combine:

- **Precomputed aggregates** for standard periods (e.g., monthly summaries), enabling fast loading and stable comparisons.

- **On-demand computation** for custom ranges and exploratory queries, enabling flexibility at the cost of longer processing time.

2.8 Positioning of this work

This thesis builds on the existing *Railway OpenData* pipeline and focuses on the *user-facing* layer: a web platform that connects the dataset to interactive maps and statistical views. Compared to script-driven workflows, the proposed approach reduces the barrier to entry for non-technical users, provides a unified interaction model across multiple views, and introduces dataset lifecycle management (upload, archive, apply/revert) to support reproducible exploration and controlled updates.

In contrast to systems that start from GTFS-only inputs (primarily describing planned service), the proposed platform is designed around operational observations collected from public endpoints. As a result, the emphasis is on robust handling of missing fields, defensible aggregation rules, and an interaction model that makes the dataset boundaries and assumptions visible to users.

Chapter 3

System Requirements and Design

3.1 Goals and scope

This thesis extends the existing *Railway OpenData* toolkit with a citizen-facing web application for interactive exploration of railway performance. The scope of the web platform is focused on:

- exposing data and analyses through a browser without requiring programming skills;
- supporting a consistent filtering model across map, dashboard, and statistics views;
- providing both fast precomputed analytics and flexible on-demand analysis;
- enabling reproducible exploration through dataset upload and archival workflows.

3.2 System requirements

3.2.1 Functional requirements

The system is designed to satisfy the following functional requirements:

1. **Dataset discovery**: infer and expose the available data range from the local dataset.
2. **Filtering**: support filtering by date range, railway operator, region, and station (by code and by free-text search).

3. **Map exploration**: provide an interactive map for station discovery and station-centric navigation.

4. **Statistics**: provide performance indicators such as descriptive statistics, delay distributions, and traffic-volume views.

5. **Dual analytics mode**: support monthly precomputed outputs and custom-range on-demand analysis.

6. **Dataset lifecycle**: support upload and safe replacement of datasets with archive/apply/revert capabilities.

7. **External enrichment (optional)**: integrate third-party queries for additional station/relation/train details with graceful fallback.

Dataset discovery

The platform infers the available date range from the locally stored daily folders and exposes it through the backend metadata endpoint (`/data/info`). The frontend surfaces this information in the filters panel as the *current data range* and *current dataset*, so users immediately understand what time window can be selected.

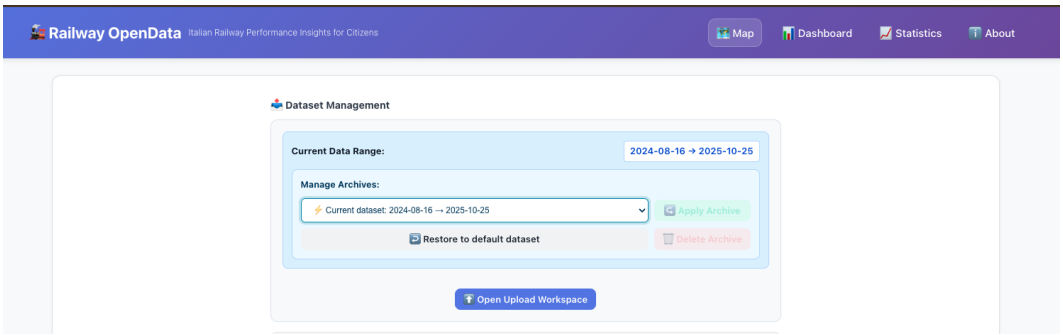


Figure 3.1: Dataset discovery in the user interface. The filters panel displays the inferred available time range and the active dataset, grounding subsequent filtering and analysis actions.

Filtering

The web application provides a consistent filtering model across the map, statistics, and dashboard views. Users can constrain analyses by selecting a date range, choosing one or more operators, narrowing the scope to a region, and selecting stations either by explicit code or through free-text search. This shared filter state enables reproducible exploration: once parameters are set, all subsequent API requests and visualizations reflect the same subset of the dataset.

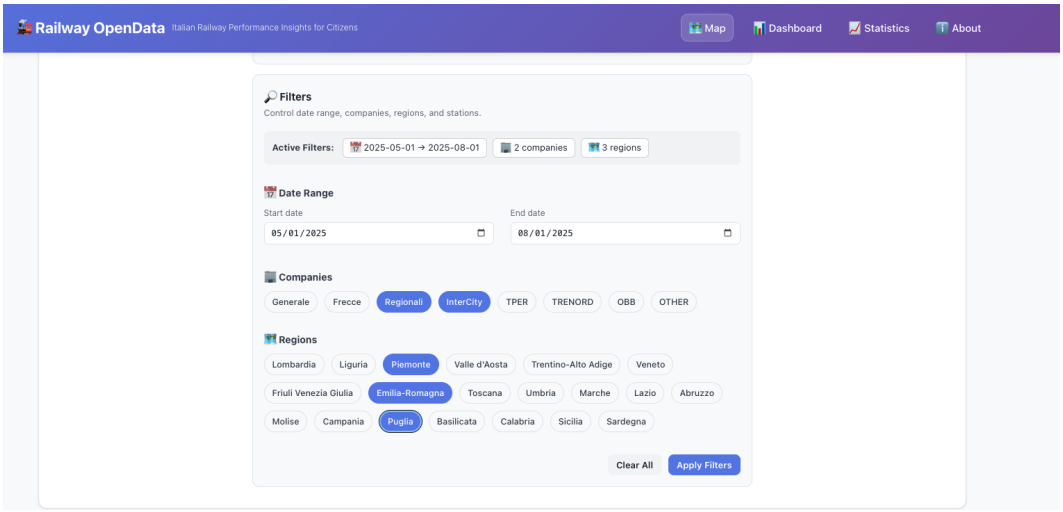


Figure 3.2: Filtering in the user interface. A single filters panel allows users to restrict the dataset by time window, operator, region, and stations (by code or free-text search), and the same selection is reused across analysis views.

Map exploration

The application includes an interactive map to support station discovery and station-centric navigation. Users can visually explore the rail network, locate stations, and select them to focus subsequent analyses on a specific area or set of stops. The map view reuses the same filtering parameters (time range, operators, regions, and stations), ensuring consistency between spatial exploration and statistical views.

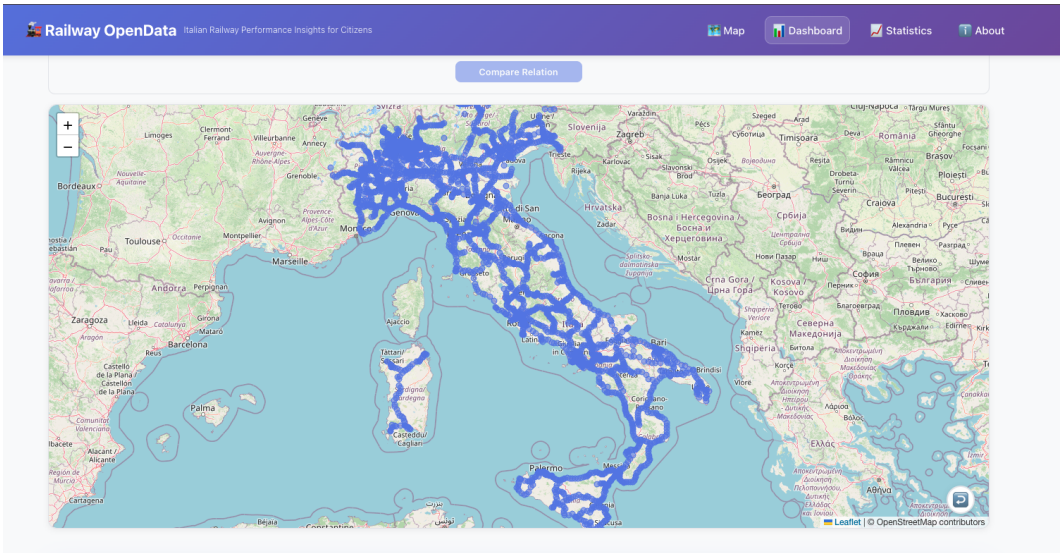


Figure 3.3: Map exploration in the user interface. The interactive map supports station discovery and selection, and remains consistent with the global filter state used by the other analytical views.

Statistics

The platform provides a set of statistical views to summarize performance and support exploratory analysis. These include descriptive statistics, delay distributions (e.g., boxplots), and traffic-volume indicators such as train counts. Users can compute statistics for a custom date range (on-demand) or retrieve monthly precomputed charts, which makes common queries fast while still enabling flexible investigation.

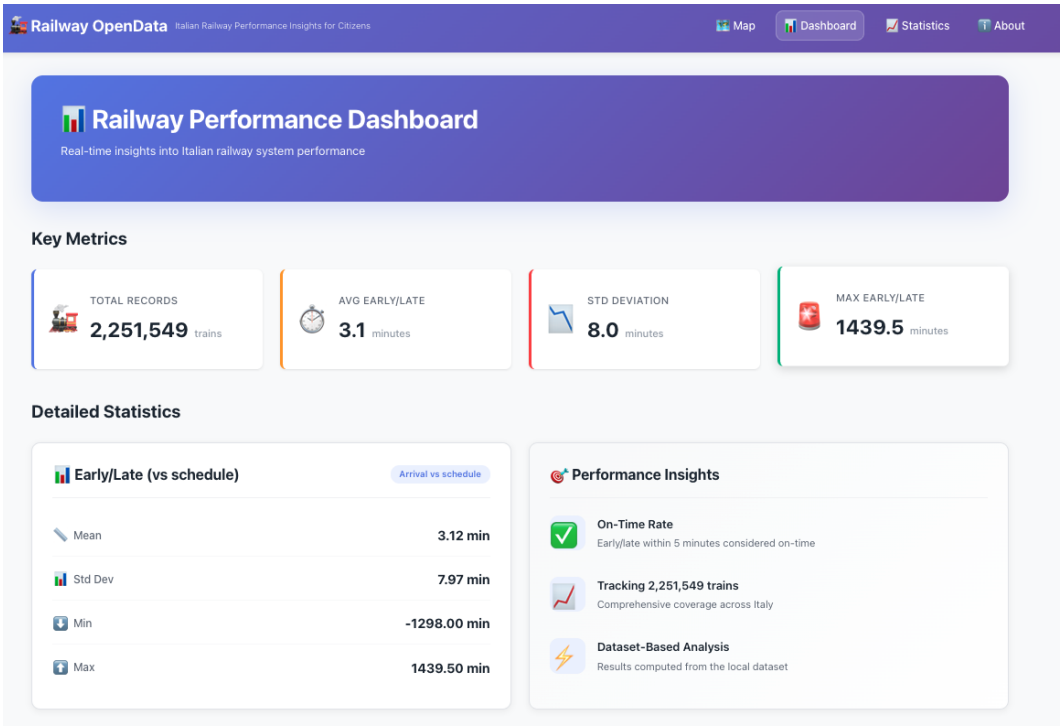


Figure 3.4: Statistics in the user interface. The application exposes key performance indicators (descriptive summaries, delay distributions, and traffic-volume views) that can be generated on-demand or loaded from monthly precomputed outputs.

Dual analytics mode

To keep the interface responsive on large datasets while preserving flexibility, the platform offers two complementary analytics paths. For common time windows (monthly), the backend serves precomputed charts stored under the outputs directory and exposed via the static files mount. For arbitrary user-selected ranges, the backend runs bounded on-demand computations and returns fresh results. The UI allows users to switch between these modes depending on the analysis goal.

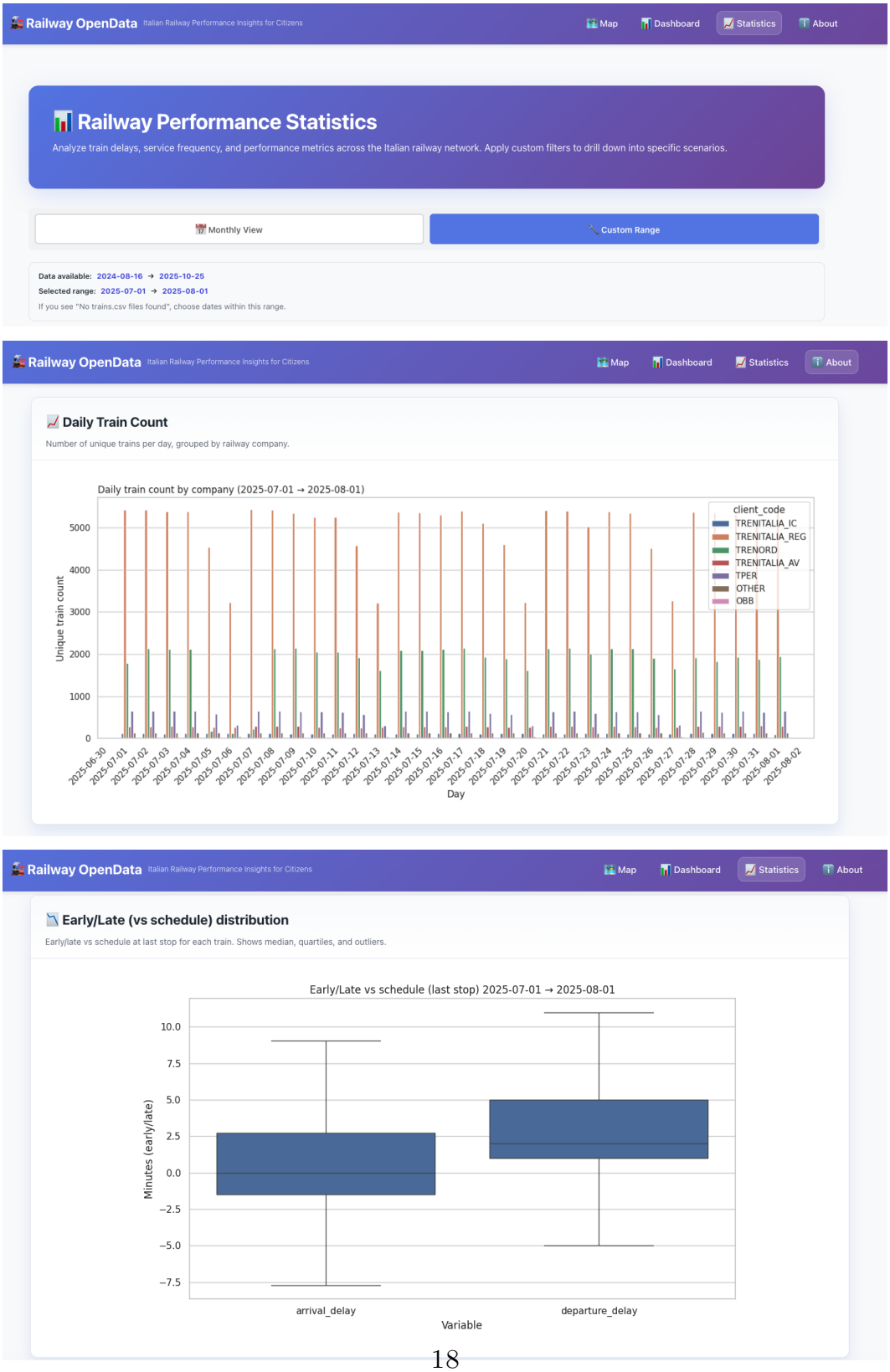


Figure 3.5: Dual analytics mode in the user interface. The platform supports both monthly precomputed outputs (fast retrieval) and bounded on-demand computations for arbitrary date ranges, allowing users to trade off latency and flexibility.

Dataset lifecycle

The platform supports dataset replacement and rollback to make explorations reproducible in the presence of evolving data collections. Users can upload a new dataset (e.g., a ZIP export) and safely switch between archived versions. This workflow is designed for iterative development and demonstrations: the system can archive the current state before changes, apply a chosen archive without creating further copies, and revert to a previous state while preserving the current one as a new archive.

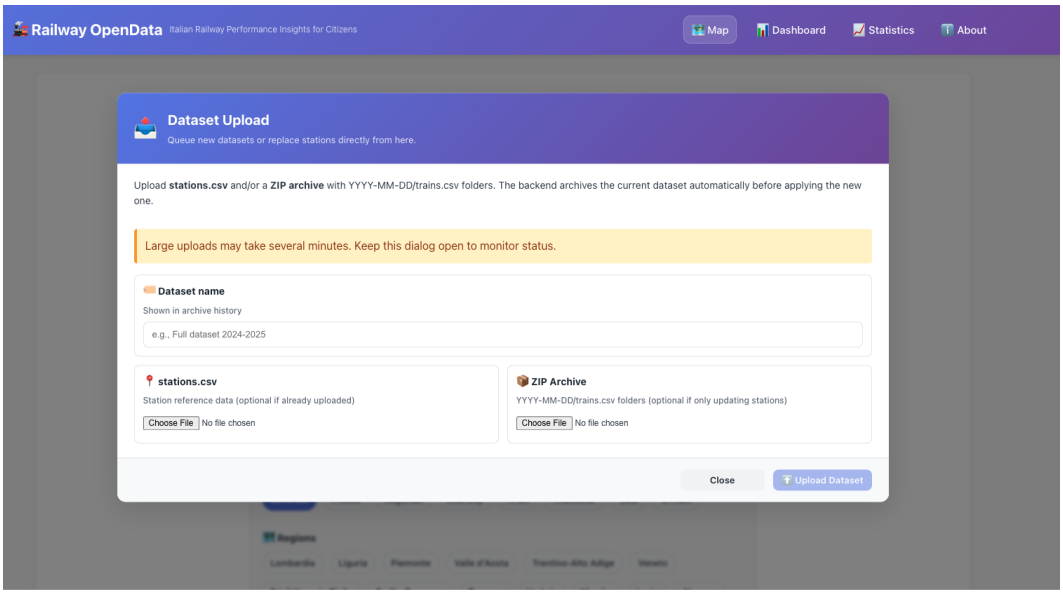


Figure 3.6: Dataset lifecycle controls in the user interface. Upload, archive selection, and apply/revert operations allow controlled dataset updates while keeping exploration reproducible.

External enrichment (optional)

To provide additional context beyond the locally stored dataset, the backend optionally integrates third-party queries for station, relation, or train details. Since external services may be unavailable or rate-limited, enrichment endpoints are treated as best-effort: failures return meaningful error messages and the UI falls back to the core analytics computed from the local dataset.

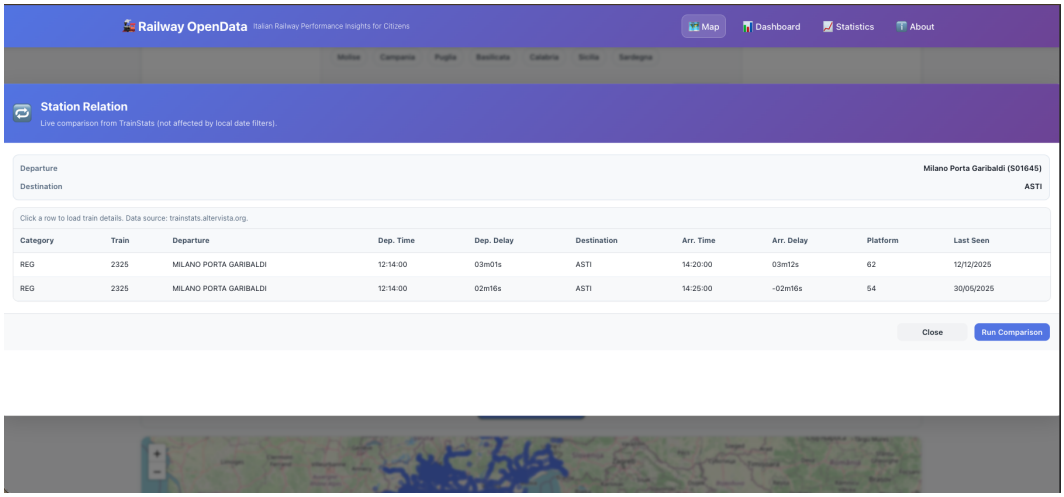


Figure 3.7: Optional external enrichment. When available, third-party lookups add details for stations or relations; otherwise, the UI continues to operate using local analytics.

3.2.2 Non-functional requirements

- **Usability**: the interface should remain understandable for non-technical users, reducing the need to know dataset structure and command-line tools.
- **Reliability**: API responses must remain well-formed in presence of missing values (e.g., NaN/Inf) and incomplete upstream records.
- **Performance**: the platform should remain responsive on large datasets by combining caching and precomputation with bounded on-demand computations.
- **Reproducibility**: dataset updates must be reversible and traceable through explicit archive management.
- **Maintainability**: the design should keep a clear separation between frontend, backend, and offline preprocessing scripts.
- **Deployability**: the system should run in a reproducible environment (local development and containerized deployment) with explicit dependencies.
- **Observability**: the backend should expose logs and clear error responses to support debugging and operational monitoring.

3.3 Architecture overview

The web platform follows a modular architecture:

- **Offline pipeline**: existing scraper/extractor tools produce daily datasets; helper scripts generate derived outputs.
- **Backend API**: a FastAPI service exposes data, statistics, and dataset-management endpoints.
- **Frontend UI**: a React single-page application (Vite) implements filters, map interaction, charts, and dataset controls.

The system uses the filesystem as the storage layer for the webapp:

- inputs under `webapp/data/` (e.g., `stations.csv` and `YYYY-MM-DD/trains.csv`);
- derived outputs under `webapp/data/outputs/` (monthly charts and cached artifacts);
- archived datasets under `webapp/data/_archive/` and a baseline snapshot under `webapp/data/_default/`.

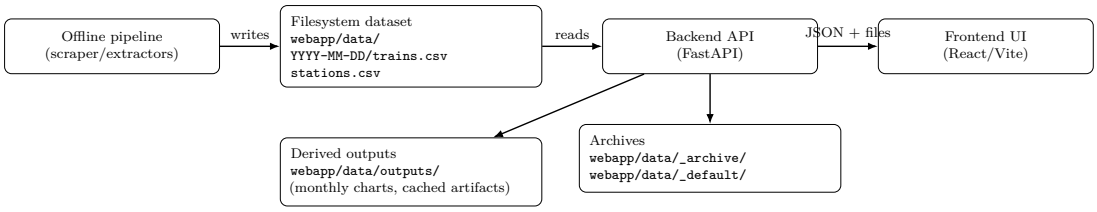


Figure 3.8: High-level architecture of the web platform. A filesystem-based dataset is shared between the offline pipeline and the backend; the frontend retrieves metadata, statistics, and precomputed artifacts through the API and static file endpoints.

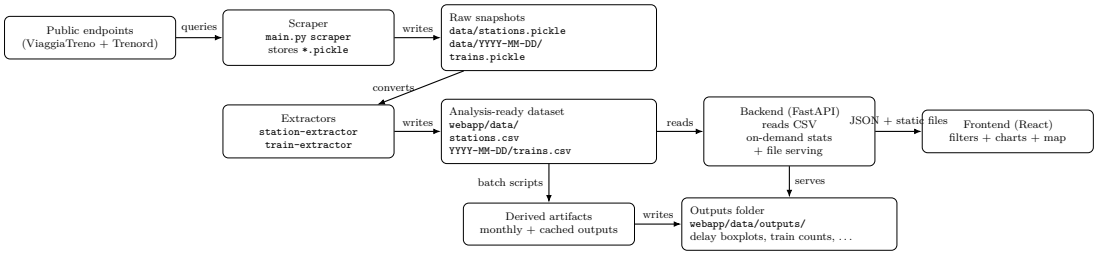


Figure 3.9: End-to-end data pipeline. Raw operational snapshots are scraped and stored as pickles, converted into CSV datasets (stations and per-day train-stop observations), and then used both for on-demand computations and for generating precomputed artifacts served to the web UI.

3.4 Data model

3.4.1 Logical data model

Although the storage layer is a set of CSV files, the dataset can be described using three conceptual entities: *stations*, *train runs* (identified by a stable hash), and *train-stop observations*. In practice, `trains.csv` is denormalized (one row per stop) and repeats train-level metadata alongside stop-level fields.

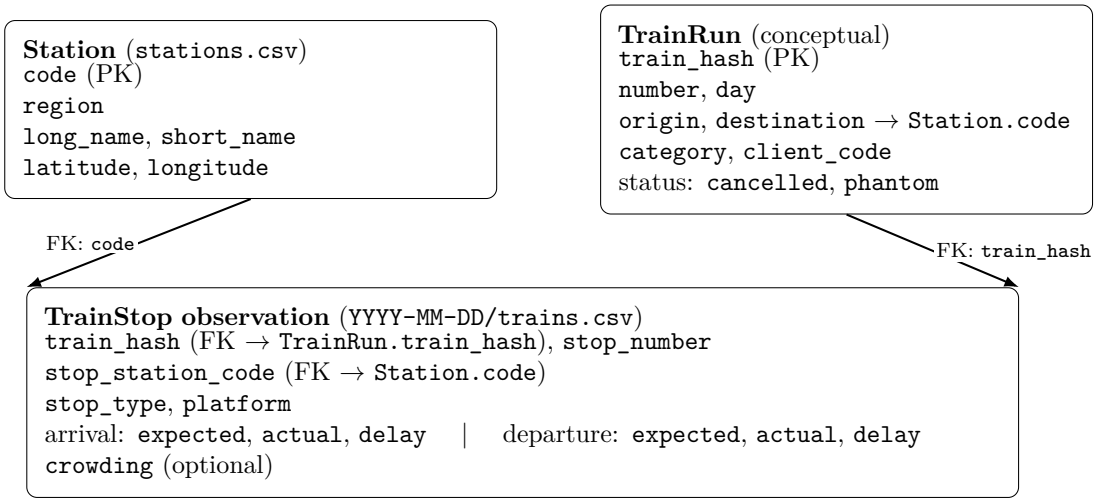


Figure 3.10: Logical data model of the dataset served by the web application. Stations are indexed by code; each train run is identified by a stable hash and produces multiple stop observations (one row per stop in `trains.csv`).

Storage note: `trains.csv` is denormalized (one row per stop) and repeats train-level fields (e.g., `day`, `origin`, `destination`) to keep the dataset self-contained.

3.4.2 Input dataset structure

The platform consumes two primary inputs:

- **Stations**: `stations.csv` with station identifiers and metadata (region code/-name and optional coordinates).
- **Train stops**: daily folders `YYYY-MM-DD/` containing `trains.csv`. Each row represents a *train stop* observation (not a station nor a train) and includes identifiers (e.g., a train hash), operator code, stop type, expected/actual timestamps, and delay fields.

This structure supports both station-level and train-level aggregations while keeping the storage format portable.

3.4.3 Derived outputs

To improve responsiveness, the backend serves precomputed artifacts when available (monthly delay boxplots and daily train-count charts). These files are stored under `webapp/data/outputs/` and are exposed through a static mount (`/files`), so the frontend can retrieve them by URL.

To avoid layout issues with long paths, file and URL references are typeset using `...`.

3.5 Filtering and query model

3.5.1 Filter dimensions

The filtering model is shared across UI sections and backend endpoints:

- **Date range**: `start_date` and `end_date` in ISO format. On-demand computations enforce an upper bound on the selectable range (one year by default) to limit runtime.
- **Operator/company**: a categorical filter based on the dataset operator codes.
- **Regions**: selection by region names, mapped to station-code sets through a stations index.
- **Stations**: selection by explicit station code(s) and by free-text query resolved using the stations endpoint.

3.5.2 Station indexing

To apply region and station filters efficiently, the backend builds an in-memory station index from `stations.csv` with an mtime-based cache. The index supports:

- code-based lookup (station code $\rightarrow$ station metadata);
- region-name lookup (region name $\rightarrow$ set of station codes).

3.5.3 Filter propagation

The shared filter state is translated into query parameters for backend endpoints. This makes results consistent across UI sections and keeps requests debuggable: the user-visible selection corresponds to a concrete API call. For example, an on-demand delay-boxplot request is expressed as:

```
/stats/delay-boxplot?start_date=2025-10-01&end_date=2025-10-31&
railway_companies=TI&regions=Piemonte&station_query=SN1234
```

In practice, the UI omits parameters that are not selected and relies on backend defaults where appropriate (e.g., empty station set means "all stations").

3.6 API design

The backend exposes a small REST API designed around two usage patterns: (i) low-latency retrieval of precomputed artifacts (static files) and (ii) bounded, on-demand computations for user-selected filters.

3.6.1 Design principles and conventions

The service is implemented in FastAPI and provides interactive documentation at `/docs`. Responses are JSON unless stated otherwise. Two conventions are used throughout the API:

- **Static artifact serving**: precomputed PNG/JSON/CSV artifacts are served under a static mount `/files` backed by the filesystem directory `webapp/data/outputs/`. Statistics endpoints that produce images return a JSON payload containing a `file_path` pointing to a `/files/...` URL.
- **Bounded computation**: on-demand endpoints validate input dates (ISO `YYYY-MM-DD`) and enforce a maximum window (one year by default). When the requested range exceeds the locally available data, the backend clamps it and reports whether clamping occurred.

To keep JSON robust across numerical edge cases, the backend sanitizes NaN/Inf values to `null` before serialization.

3.6.2 Common query parameters

Several endpoints share the same filtering dimensions, encoded as simple query parameters:

- **Date range**: `start_date` and `end_date` in ISO format. If omitted, dashboard-oriented endpoints fall back to a recent default window (e.g., the latest 30 days) to avoid scanning the entire dataset.
- **Companies and regions**: `railway_companies` and `regions` are comma-separated lists. This encoding keeps requests shareable and easy to debug.

- **Station filter**: `station_query` is a free-text value resolved by the backend into a set of station codes.
- **Cache control**: `recompute=true` forces regeneration of cached artifacts under `webapp/data/outputs/runtime/`.

Error handling. The backend relies on HTTP status codes and FastAPI's standard JSON error schema (`{"detail": "..."}`). Typical cases include `400` for invalid parameters (e.g., malformed dates), `404` when an expected artifact is missing, `501` for explicitly unsupported features (trajectories), and `502` for failures in optional third-party enrichment.

3.6.3 Endpoint catalogue

Service and metadata

These endpoints support UI initialization and operational checks:

- **GET** `/` returns a short service banner and the docs URL.
- **GET** `/health` returns `{"status": "ok"}` for health checks.
- **GET** `/meta/regions` returns the region names used by the region filter.
- **GET** `/meta/companies` returns operator/company codes used by the dataset (including a UI sentinel for "no filter").
- **GET** `/data/info` infers the locally available date range from `webapp/data/YYYY-MM-DD/` and returns `available_min_date`, `available_max_date`, and the current `dataset_name`.

Dataset lifecycle management

Dataset management endpoints use `POST` with `multipart/form-data` to support file uploads and avoid embedding large payloads in JSON. The API is designed to keep dataset replacement reversible via explicit archiving:

- **GET** `/data/archives`
  lists the current dataset, the baseline snapshot (`_default`), and any archived datasets under `webapp/data/_archive/`.
- **POST** `/data/upload` replaces the current dataset. It supports multiple upload modes (ZIP-based dataset replacement, folder-based uploads, and stations-only updates) and can optionally trigger background precomputation of default outputs.

- **POST** `/data/apply-archive` switches the current dataset to a selected archive stamp without creating a new backup (fast switching).
- **POST** `/data/revert` restores the most recent (or a specified) archive while archiving the current state first (reversible rollback).
- **POST** `/data/delete-archive` deletes a specific archive (protecting special stamps such as `_default` and the current dataset).
- **POST** `/data/clear-archives`
  clears `_archive/` while keeping a valid dataset (by restoring the default snapshot if needed).

Statistics (on-demand)

On-demand endpoints compute results from `webapp/data/YYYY-MM-DD/trains.csv` after applying the shared filters. To keep latency bounded, they enforce date-range limits and cache results under `webapp/data/outputs/runtime/`:

- **GET** `/stats/describe` computes descriptive statistics over numeric fields (e.g., delays and crowding). It returns a dashboard-friendly summary (mean, quartiles, etc.) and a full `describe` table.
- **GET** `/stats/delay-boxplot` generates a delay distribution boxplot (typically using the last stop per train to approximate end-to-end punctuality) and returns a `file_path` to the generated PNG.
- **GET** `/stats/day-train-count` aggregates the number of unique trains per day and operator and returns a `file_path` to a generated bar chart.

Statistics (precomputed monthly)

To make common explorations instantaneous, a separate set of endpoints serves monthly artifacts generated offline:

- **GET** `/stats/available-months` lists the `year/month` pairs for which precomputed outputs are present.
- **GET** `/stats/day-train-count/monthly` returns the precomputed train-count chart for a given `year` and `month`.
- **GET** `/stats/delay-boxplot/monthly` returns the precomputed delay boxplot for a given `year` and `month`.

Stations and map

These endpoints support station discovery, typeahead selection, and map rendering:

- **GET** `/stations`
  returns a GeoJSON `FeatureCollection` built from `stations.csv` (with optional substring search via `q`, result limiting via `limit`, and a `with_coords_only` flag for map markers).

- **GET** `/map/trajectories` is reserved for trajectories; in the live-data configuration it returns `501 Not Implemented`, documenting that full trajectories are not reliably obtainable from the upstream public endpoints.

External enrichment (optional)

External enrichment endpoints are treated as best-effort helpers for contextual information. They proxy or scrape third-party services and therefore include defensive timeouts, caching, and error handling:

- **GET** `/stats/external-station/{station_code}`
  returns station-level punctuality and traffic summaries for a given date (defaulting to the previous day).

- **GET** `/stats/external-relation` retrieves relation rows between two stations (departure/destination), with an optional debug mode to assist troubleshooting name matching.

- **GET** `/stats/external-relation-stations` returns the station list used by the external relation service.

- **GET** `/stats/external-relation-origins` returns the set of origins available in the external dataset, supporting origin selection and typeahead.

- **GET** `/stats/external-relation-destinations` returns possible destinations for a selected origin station (with caching to keep typeahead responsive).

- **GET** `/stats/external-train` fetches details for a selected train row from the external relation results.

3.6.4 Example interactions

The following calls illustrate the overall style of the API (query parameters, date validation, and file-path responses):

```
GET /data/info

GET /stats/delay-boxplot?start_date=2025-10-01&end_date=2025-10-31&
    railway_companies=TI&regions=Piemonte&station_query=SN1234

GET /stats/day-train-count/monthly?year=2025&month=10
```

3.7 Dataset lifecycle management

3.7.1 Upload and safety constraints

The dataset upload endpoint supports multiple modalities (ZIP-based replacement, stations-only updates, and folder-based uploads). The design includes safety constraints:

- path-safety checks rejecting directory traversal patterns (e.g., `..`);
- safe ZIP extraction that ignores system artifacts and prevents invalid paths;
- dataset root detection based on the presence of dated folders and expected filenames.

3.7.2 Archiving, apply, and revert

Before replacing the dataset, the backend can archive the current dataset state under `_archive/<stamp>`. Two complementary operations are supported:

- **Apply**: load a selected archive as the current dataset without creating a new backup (fast dataset switching).
- **Revert**: restore the most recent (or a specified) archived dataset while archiving the current state first, keeping the operation reversible.

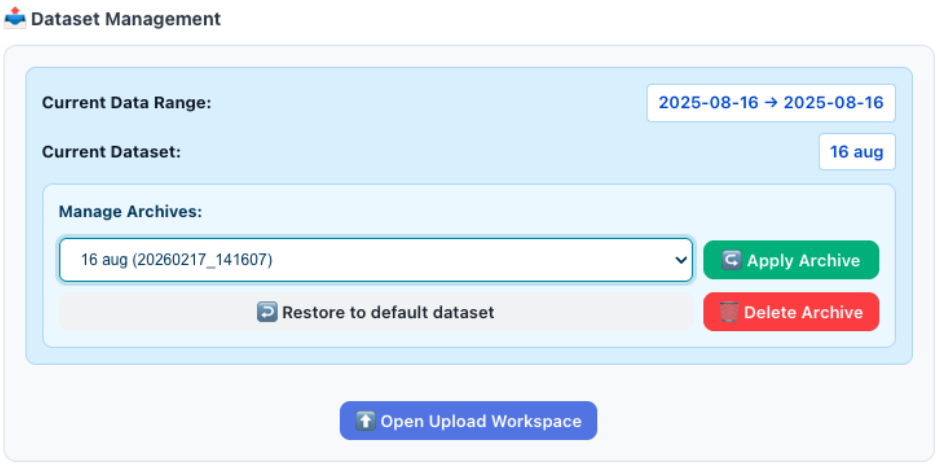


Figure 3.11: Dataset archive management in the UI. The user can select an archived dataset snapshot and apply it as the current dataset, or revert to the most recent archived state while keeping the operation reversible.

This workflow is essential for reproducible exploration and controlled dataset evolution during demonstrations.

3.8 Frontend design

3.8.1 One-page structure and shared state

The frontend is implemented as a React single-page application (Vite) organized as a single scrolling page with section anchors. A minimal router maps legacy paths (e.g., `/statistics`) to anchor redirects (e.g., `/#statistics`) to keep backwards compatibility while preserving a single interaction model.

The core interaction state is centralized in the page container and propagated to sections via props:

- **Applied filters**: a single object containing date range, selected companies and regions, and station constraints. This object is the only source of truth for downstream components.
- **Dataset version**: an integer counter incremented after dataset lifecycle operations (upload/apply/revert/delete/clear). Sections include it in their effect dependencies to trigger a consistent refetch of metadata and charts.

This design avoids duplicated state across sections and ensures that a change in one place (e.g., selecting a region) deterministically updates all dependent views.

3.8.2 Map-to-statistics workflow

The map is implemented with Leaflet (`react-leaflet`) and consumes station locations from
`/stations` (GeoJSON) to render station markers. When the user clicks a station marker, the UI performs a coordinated action:

- it updates the browser URL with a query parameter (`?stationCode=...`) and navigates to the statistics anchor;
- it dispatches a custom event (`stationSelected`) listened to by the page container, which updates the shared filter state (single-station selection);
- it scrolls to the relevant section using an offset to account for the fixed header.

Clearing a selection is symmetric (clicking on the map background clears `stationCode` and emits `stationCleared`), keeping navigation, state, and UI view aligned.

3.8.3 URL-driven reproducibility

The UI supports initialization of a minimal, shareable state from URL query parameters, primarily: (i) `stationCode` to reproduce station-centric views and (ii) optional `startDate`/`endDate` to reproduce a specific analysis window. On first load, the page container parses these parameters and seeds the shared filter state, enabling reproducible demonstrations without requiring manual UI interactions.

Section navigation uses the URL hash (e.g., `#statistics`); a dedicated scroll helper delays navigation by one animation frame to ensure correct layout measurements.

3.8.4 API client layer

All frontend-to-backend calls are centralized in a small service wrapper built on top of axios. The base URL is configured via the environment variable `VITE_API_URL` (defaulting to `http://localhost:8000`), which allows the same bundle to operate in local development and containerized deployments.

The service layer exposes explicit methods for each backend endpoint (metadata, stations, statistics, dataset lifecycle, external enrichment). This improves maintainability by avoiding scattered hard-coded URLs and keeps query parameter construction consistent with the backend contract (e.g., comma-separated `regions` and `railway_companies`).

3.8.5 Dataset controls in the UI

Dataset lifecycle controls (upload, archive selection, apply/revert, delete/clear) are exposed through the filters component. The implementation follows three UI principles:

- **Immediate feedback**: operations display progress states and user-facing toasts/messages to confirm success or describe failures.
- **Safe defaults**: the archive selector prefers the currently active dataset and prevents accidental invalid choices (e.g., attempting to delete protected stamps).
- **Consistent refresh**: after any dataset mutation, the filters component triggers `onDatasetApplied()`, incrementing the shared dataset version so that map data, available months, and statistics refresh coherently.

3.8.6 Robustness and user feedback

Sections implement explicit loading, error, and empty-data states. For example, the dashboard and statistics sections surface backend validation errors (returned as `{"detail": "..."}`) as readable messages, while treating optional enrichment failures as non-fatal. Additionally, the statistics section is wrapped in a React error boundary, ensuring that unexpected rendering errors do not break the entire single-page experience.

3.9 Design constraints

The platform inherits constraints from both the upstream data sources and the chosen storage/interaction model. These constraints influenced both the API contract and the frontend behavior.

3.9.1 Upstream data limitations

The upstream public endpoints used by the scraper provide partial, sometimes inconsistent observations. Consequently:

- **Missing values are expected**: some delay/crowding fields may be absent or non-numeric. The backend sanitizes these values for JSON safety, and the UI renders explicit “not available” states instead of failing.
- **Station metadata is incomplete**: coordinates may be missing or incorrect for some stations; the map uses a `with_coords_only` constraint when rendering markers.

- **Trajectories are not guaranteed**: full train trajectories cannot be reliably reconstructed for all services from the public endpoints; therefore `/map/trajectories` is intentionally not implemented in the current configuration.

3.9.2 Performance constraints (client and server)

The dataset can span hundreds of days and millions of stop rows, so the system avoids transferring raw records to the browser. Instead, the backend exposes aggregated endpoints and image artifacts:

- **Bounded on-demand computations**: requests are validated and date windows are limited to control latency.
- **Precomputation for common views**: monthly charts are served instantly via `/files/...` paths returned by monthly endpoints.
- **Caching**: runtime artifacts are cached on the backend, reducing recomputation when users explore the same filter combinations.

On the frontend, the station search is debounced and suggestion lists are limited to keep the UI responsive even on low-end machines.

3.9.3 Reproducibility and safety

Dataset lifecycle operations must remain reversible to support demonstrations and iterative data cleaning. This requirement leads to the archive/apply/revert workflow and to a conservative upload pipeline with path-safety checks and controlled extraction. On the UI side, dataset mutations trigger a full refresh (via the dataset version mechanism) so that all sections remain consistent with the active dataset snapshot.

3.9.4 External dependency constraints

External enrichment is intentionally treated as optional: third-party services may be slow, change format, or be unavailable. The frontend therefore renders enrichment results only when present and continues to provide local analytics without degradation when enrichment fails.

Chapter 4

Implementation

4.1 Overview

This chapter describes the concrete implementation choices behind the web platform introduced in Chapter 3. The implementation is organized as a full-stack system:

- a **FastAPI backend** that reads the dataset from the filesystem, computes statistics on demand, serves precomputed artifacts, and exposes dataset-management endpoints;
- a **React single-page frontend** (Vite) that provides a one-page user experience with shared filters, an interactive map, and analytics views;
- **offline scripts** used to precompute monthly charts and speed up common views.

4.2 Backend implementation

4.2.1 Technology stack and structure

The backend is implemented in Python using `FastAPI`. The service lives in `webapp/backend/main.py` and follows a pragmatic design: the filesystem is treated as the storage layer, and computation is performed by reading the daily `trains.csv` files under `webapp/data/`.

Two directory roles are relevant:

- **Dataset root**: `webapp/data/`, containing `stations.csv` and the daily folders `YYYY-MM-DD/`.
- **Derived outputs**: `webapp/data/outputs/`, containing precomputed images for the monthly mode and other cached artifacts.

4.2.2 Static file serving

The backend mounts `webapp/data/outputs/` under a static route (`/files`). This allows the API to return lightweight JSON payloads containing the file URL, while the browser fetches the actual image directly.

Two important consequences are:

- the frontend can render charts as standard `<img>` tags, which is robust and avoids heavy client-side plotting for large datasets;
- the backend can cache generated artifacts on disk and reuse them across requests.

4.2.3 Core filtering and range validation

Most analytics endpoints accept query parameters for:
`start_date`, `end_date`, `railway_companies`, `regions`, and `station_query`.

The implementation enforces practical safeguards:

- ISO date parsing and validation of the `start_date` $\leq$ `end_date` invariant;
- clamping of requested windows to the available dataset range (so the UI can be guided by real bounds);
- a maximum selectable range (one year by default) for on-demand computations, preventing accidental expensive requests.

4.2.4 Station indexing and the `/stations` endpoint

Station data are exposed via `GET /stations` as a GeoJSON `FeatureCollection`, enabling both map rendering and typeahead search. The endpoint supports:

- substring search over station properties (`q`);
- optional limiting (`limit`) to support dropdowns;
- optional filtering to return only stations with coordinates (`with_coords_only`).

To keep the UI responsive, the backend maintains an in-memory stations index (with modification-time based invalidation) used to resolve region and station constraints into station-code sets.

4.2.5 Statistics endpoints

The main analytics endpoints are implemented as synchronous handlers that:

1. determine a date window (explicit user range or default "last N days" window for the dashboard),
2. load only the necessary columns from the daily `trains.csv` files within that window,
3. apply filters (company, regions, and station constraints),
4. compute the requested aggregate, and
5. cache the result (JSON or PNG) so subsequent calls can be served quickly.

Descriptive statistics (`/stats/describe`). The endpoint returns a backward-compatible summary at the top level (e.g., `count`, `mean`, `std`) and also returns a full `describe` table for numeric columns. Importantly, the implementation counts *unique trains* (based on `train_hash`) rather than the number of stop rows, to avoid misleading totals.

Delay distribution (`/stats/delay-boxplot`). To avoid overcounting multiple stops for the same train, the boxplot is computed on a per-train basis by selecting the *last observed stop* per `train_hash`. The endpoint returns the file path of a generated PNG saved in a runtime cache folder.

Service frequency (`/stats/day-train-count`). Daily train counts are computed as the number of unique `train_hash` values per day and company. The output is a bar chart saved as a PNG. Label density on the x-axis is adaptively reduced to keep the chart readable for long ranges.

Monthly precomputed mode. For the monthly view, the backend exposes:

- `/stats/available-months`
- `/stats/day-train-count/monthly`
- `/stats/delay-boxplot/monthly`

These endpoints return paths to precomputed images stored under:
`webapp/data/outputs/day_train_count/YYYY-MM/`.

4.2.6 Robust JSON serialization

Operational data often contains missing or invalid numeric values. The backend applies a recursive sanitizer that converts NaN/Inf to `null` before encoding to JSON. This prevents frontend crashes and avoids non-standard JSON payloads.

4.2.7 Dataset lifecycle management (upload and archives)

The backend implements an explicit dataset lifecycle in `webapp/data/`:

- **Upload**: `POST /data/upload` accepts multiple upload modes (stations-only, ZIP-based dataset replacement, or a "full" mode with both stations and ZIP).
- **Archives**: before replacing the dataset, the current content is moved under `webapp/data/_archive/<timestamp>/`.
- **Apply**: `POST /data/apply-archive` loads a selected archive as the current dataset without creating a new backup, enabling fast dataset switching.
- **Revert**: `POST /data/revert` restores an archive while keeping operations reversible.

To mitigate common security pitfalls, ZIP extraction is performed with explicit path-safety checks that reject directory traversal and ignore system artifacts.

4.2.8 Data quality issues and preprocessing scripts

In practice, a major portion of the implementation effort was spent on making the dataset *ingestable* and *stable* across different sources and operating systems. The most recurring issues were:

- **Station CSV encoding and quoting.**
  Some station lists were distributed with non-UTF-8 encodings or inconsistent quoting. In particular, the `long_name` field occasionally contained unescaped commas, which breaks naive CSV parsing. To make the pipeline deterministic, the project includes scripts such as `scripts/fix_station_encoding.py` (tries common encodings and rewrites as UTF-8) and `scripts/clean_stations.py` / `scripts/batch_clean_stations.py` (repairs unescaped commas by reconstructing fields and writing a properly quoted UTF-8 file).
- **Missing or invalid coordinates.** Map rendering requires consistent latitude/longitude. Some stations had (0,0) coordinates or values outside plausible Italian bounds, which placed markers in irrelevant locations or prevented rendering. The script `scripts/fix_zero_coordinates.py` detects these anomalies and optionally queries the upstream ViaggiaTreno station endpoint to fill missing coordinates.

- **Dataset packaging artifacts.** When datasets were exchanged as ZIP archives, macOS-specific entries (`__MACOSX/`, `.DS_Store`) appeared frequently and had to be ignored. This was addressed directly in the backend extraction routine.

These preprocessing steps are not "nice-to-have": without them, the web application would fail to load stations correctly (typeahead and region filters), and the map would either be empty or misleading.

4.2.9 Performance and caching trade-offs

On-demand analytics over multiple daily `trains.csv` files can become expensive if the requested window is too large. Several mitigations were introduced to keep the UI responsive:

- **Bounded computation windows.** Requests are validated with a maximum range (one year by default). The dashboard also defaults to a recent window (e.g., last 30 days) rather than computing over the entire dataset.
- **Column projection and per-train aggregation.** Both the backend endpoints and the offline generators read only the necessary columns (via `usecols`) and aggregate per train (`train_hash`) where appropriate (e.g., boxplots computed on the last stop), reducing memory pressure and avoiding stop-level overcounting.
- **Stable cache keys.** Cache filenames incorporate a short hash (SHA-256) of the effective request parameters (date range and filters). This prevents accidental collisions while keeping the artifacts reusable across repeated requests.

4.2.10 Notes on map trajectories

The API includes `GET /map/trajectories` as a placeholder, but in the current implementation this endpoint returns `501 Not Implemented`. The map section focuses on station exploration and station-to-statistics navigation; train trajectory visualization would require an additional reliable upstream source or an offline precomputed trajectories dataset.

4.3 Frontend implementation

4.3.1 One-page architecture and shared state

The frontend is implemented as a React single-page application. Instead of separate pages for each feature, the UI is composed of sections (Map, Dashboard, Statistics,

About) on a single route. A shared filters state is managed at the top level and passed down to the sections.

The application supports lightweight shareable URLs:
`?stationCode=<code>#statistics` preselects a station and opens the statistics section;
`?startDate=YYYY-MM-DD&endDate=YYYY-MM-DD` initializes the global range.

4.3.2 Filters component

The Filters panel is the control center of the application. It exposes:

- date range selection with basic validation (both dates required, ordering constraints);
- multi-select company and region selection (rendered as chips);
- station search with server-backed suggestions (`/stations?q=...`);
- dataset management tools: upload new datasets, list archives, apply/revert, and delete/clear archives.

Filters are applied explicitly via an `Apply Filters` action to avoid unnecessary network calls while the user is still editing inputs.

This explicit apply step was introduced after observing two practical problems during implementation:

- rapidly changing inputs (typing in station search, toggling chips) can cause request bursts and stale intermediate states;
- expensive analytics calls (especially chart generation) should be triggered intentionally, not on each keystroke.

4.3.3 Map section

The map is built using `react-leaflet`. The map loads station GeoJSON from `/stations` (restricted to stations with coordinates) and renders markers and popups. Selecting a station triggers a station-selection event and updates the URL with `stationCode=...`, allowing a direct handoff to the Statistics view.

4.3.4 Dashboard and Statistics sections

The Dashboard section calls `/stats/describe` with the active filters and presents compact summary metrics for quick inspection.

The Statistics section provides two modes:

- **Monthly**: selects a month from `/stats/available-months` and displays precomputed charts.
- **Custom**: uses either the global range or an optional section-specific range and calls `/stats/delay-boxplot` and `/stats/day-train-count`.

When exactly one station is selected, the section shows a station details card and attempts to enrich the view with optional external station statistics.

4.4 Precomputation and deployment

4.4.1 Monthly chart generation

To support fast and reproducible "monthly" views, the project includes an offline generator (`scripts/webapp_generate_monthly_charts.py`). The script iterates over daily folders, groups them by month, and produces for each month:

- a daily train-count chart by company (`day_train_count_YYYY-MM.png`);
- a delay distribution boxplot based on the last stop per train (`delay_boxplot_YYYY-MM.png`).

The generated artifacts are stored under:
`webapp/data/outputs/day_train_count/YYYY-MM/`.
They are served directly by the backend through the `/files` static mount.

Compared to the online endpoints, this generator emphasizes throughput and robustness: it reads only the columns needed for each chart, filters out phantom rows when the dataset provides those flags, and continues month processing even if individual day files are malformed.

4.4.2 Containerized execution

The web application is deployable via Docker Compose. The backend container runs the FastAPI application and mounts `webapp/data/` as a volume so the dataset can be replaced without rebuilding images. The frontend container runs the Vite development server and is configured via `VITE_API_URL` to point to the backend.

This approach supports demo-friendly deployments and makes it easy to reproduce the interactive exploration environment across machines.

Chapter 5

Testing and Evaluation

5.1 Testing methodology

Testing and validation span three layers of the project:

- the **scraper layer**, responsible for fetching and normalizing operational data from upstream services;
- the **analytics backend**, responsible for range validation, filtering, aggregation, and caching of derived artifacts;
- the **web frontend**, responsible for correctly wiring user interactions (filters, station selection, modes) to API calls.

Given the heterogeneous nature of the system, the adopted strategy combines:

- **automated tests** (primarily for the scraper) using `pytest`, with both deterministic fixture-based tests and live integration checks;
- **offline sanity scripts** to reproduce backend calculations on a known slice of the dataset and compare outputs;
- **manual exploratory validation** for end-to-end user flows in the deployed web application (Docker Compose), focusing on correctness of interactions and robustness to invalid inputs.

This mixed approach is intentional: fully deterministic end-to-end tests are difficult when the upstream APIs are external and subject to availability, while the analytics backend depends on potentially large CSV inputs where correctness is best validated by reproducible offline computations.

5.1.1 Automated tests in the scraper module

The scraper includes a test suite under `src/scraper/tests/`. The tests exercise both parsing logic and API integration:

- **Unit/fixture tests** validate object construction and string representations from stored JSON fixtures (e.g., station and stop parsing). These tests target correctness of assumptions (non-null fields, consistent time parsing) and are independent from network conditions.
- **Integration tests** perform real requests against the upstream API (e.g., station arrivals/departures, station lists by region) to detect contract breaks or unexpected schema changes.

The use of `pytest.mark.parametrize` enables the same logic to be validated on multiple station codes, regions, and train categories, improving coverage without duplicating code.

5.1.2 Backend validation approach

The backend computations are designed around explicit invariants and defensive checks (date parsing, range clamping, maximum selectable range, NaN/Inf sanitization). These behaviors are validated through a combination of:

- direct API calls in a running deployment (checking that invalid ranges are rejected and that out-of-bounds ranges are clamped to the dataset limits);
- an offline computation script (`scripts/test_backend_calculation.py`) that reproduces the logic of selected statistics on a controlled time window.

The offline script is particularly useful to validate semantics such as counting *unique trains* (via `train_hash`) rather than raw stop rows.

5.2 Functional validation

Functional validation focuses on the core user stories described in Chapter 3 and implemented in Chapter 4. The main flows were verified on a Docker Compose deployment:

- **Dataset exploration:** the map loads stations from `GET /stations` and allows station selection, which updates the URL and drives the statistics view.
- **Filtering correctness:** applying date range, company, region, and station constraints updates the backend requests, and the dashboard summaries reflect the active filters.

- **Statistics modes:** the monthly mode resolves precomputed images via `/stats/available-months` and `/files`, while the custom mode triggers on-demand generation via `/stats/delay-boxplot` and `/stats/day-train-count`.
- **Input robustness:** the UI prevents common invalid states (missing dates, inverted ranges) and the backend provides an additional safety net (range validation and maximum allowed span).
- **Dataset lifecycle:** the upload and archive workflow (create archive, apply/revert, delete) was validated by switching between datasets and confirming that subsequent API calls use the active snapshot.

Two recurring correctness checks were used during validation:

- **Train cardinality:** totals exposed to the user are interpreted as train counts based on `train_hash` (as opposed to stop rows), preventing inflated numbers.
- **Delay semantics:** delay distributions are computed using the last observed stop per train, so a train contributes once to the boxplot even if it has many intermediate stops.

5.3 Performance considerations

5.3.1 Design choices for interactive performance

Interactive performance is achieved through complementary mechanisms:

- **range bounding:** on-demand analytics endpoints enforce a maximum range (one year by default), reducing worst-case computation time;
- **disk caching of artifacts:** generated charts are written to the runtime outputs folder and reused on subsequent identical requests;
- **monthly precomputation:** commonly accessed views are generated offline and served as static files, reducing backend load to lightweight metadata responses.

5.3.2 Examples of generated charts

This section reports examples of the main visual outputs produced by the platform, grouped by the same categories presented in the UI.

Daily train count. Figure 5.1 shows the daily number of unique trains by company over the selected time window.

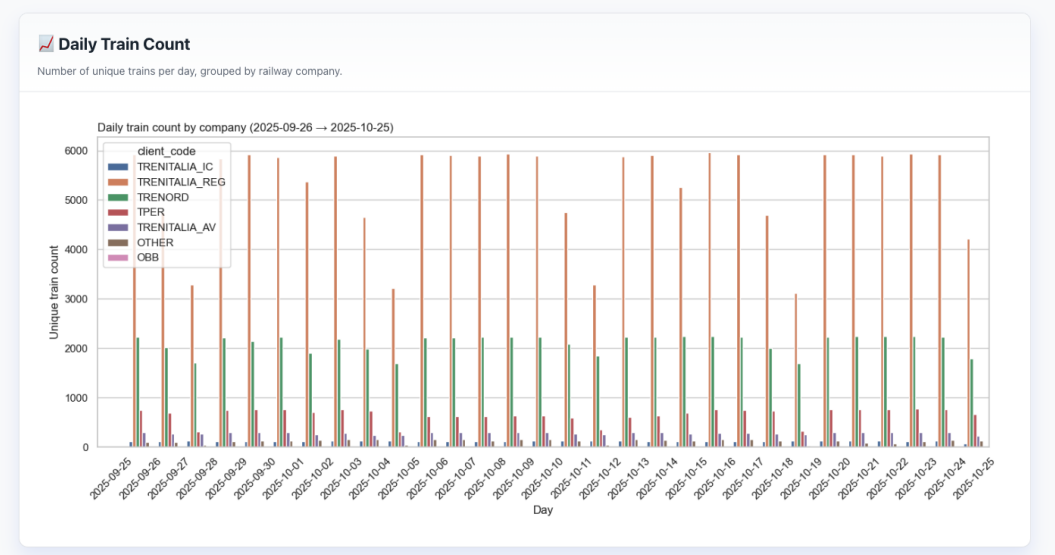


Figure 5.1: Daily train count chart (unique trains by company).

Delay box plot. Figure 5.2 shows the delay distribution computed per train (one contribution per train, based on the last observed stop).

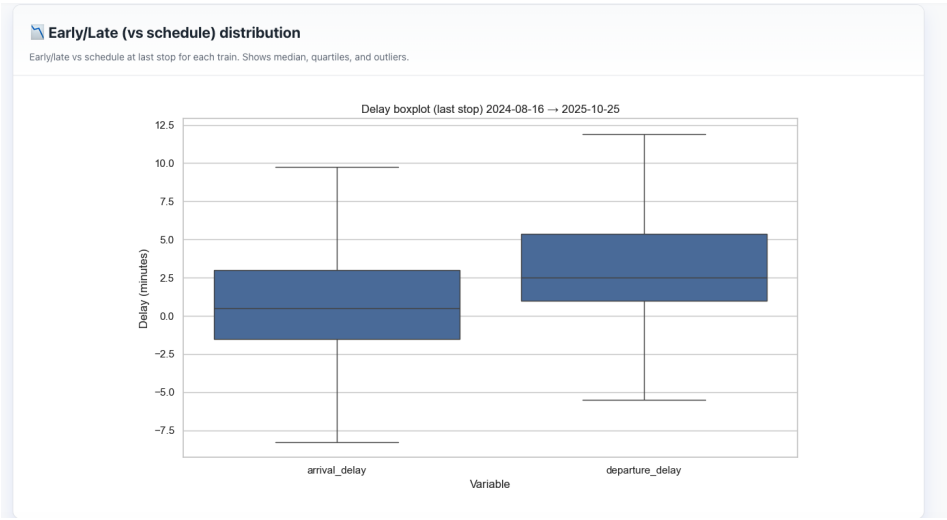


Figure 5.2: Delay distribution box plot (per-train, last-stop semantics).

General key metrics. Figure 5.3 provides an example of the compact key metrics view used for quick validation after filters are applied.

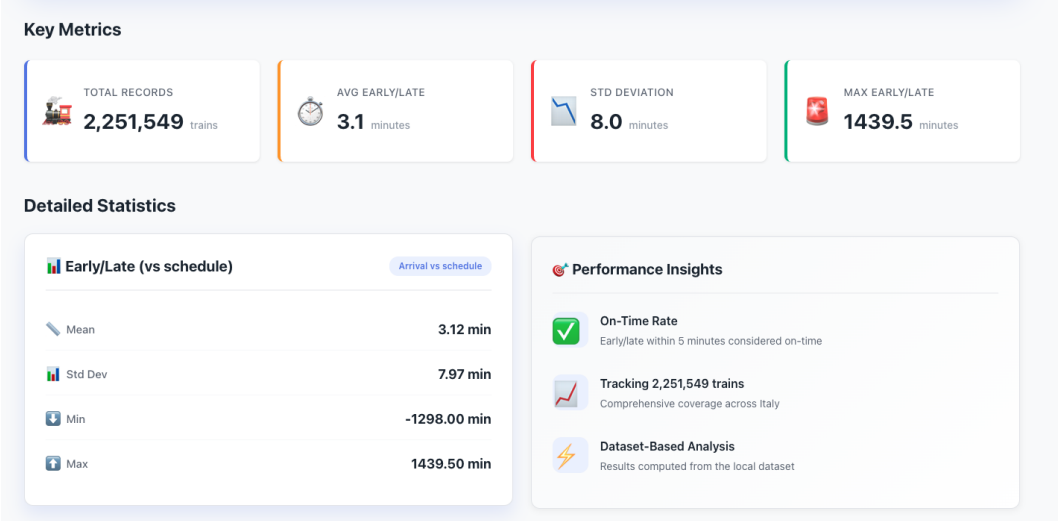


Figure 5.3: General key metrics (dashboard summary with active filters).

Station relational charts. Figure 5.4 shows an example of a station relational visualization (e.g., origin–destination relationships or related-destinations summary) used to contextualize a selected station.

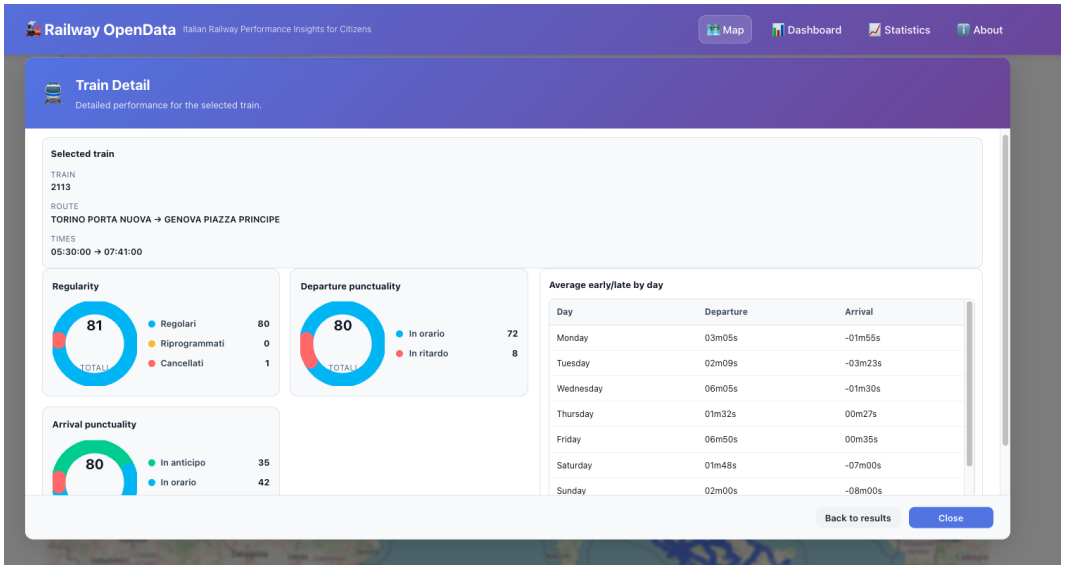
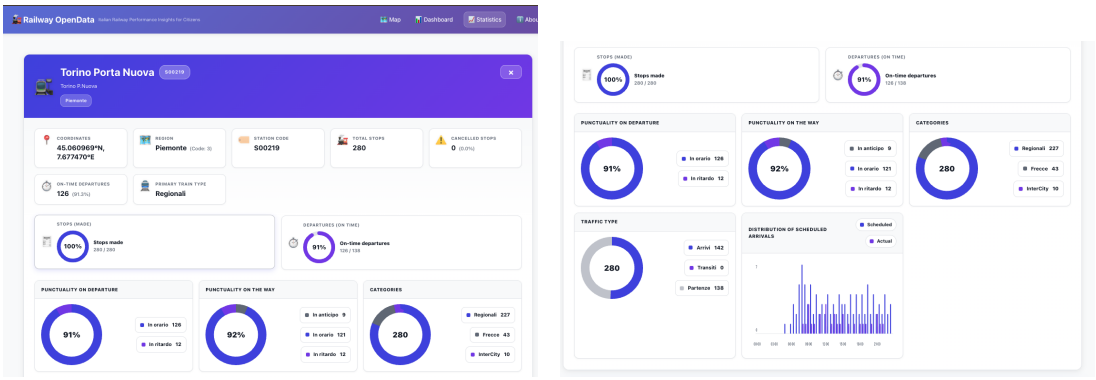


Figure 5.4: Station relational chart example.

Station charts (1,2). Figure 5.5 includes two station-specific charts shown when a station is selected.



(a) Station chart (1) **(b)** Station chart (2)

Figure 5.5: Station-specific charts displayed for a selected station.

5.3.3 Dataset scale and empirical scan time

To contextualize performance, a scale measurement was performed on a 30-day window at the end of the available dataset range (`2025-09-26` to `2025-10-25`). The scan processed the daily `trains.csv` files and computed basic cardinalities using the standard Python CSV reader.

On the development machine used for this thesis draft, the 30-day slice contained:

- 3,165,342 stop rows;
- 269,601 unique trains (distinct `train_hash`);
- an average of 11.74 stops per train.

The scan completed in approximately 5.84 seconds.

This measurement is not meant to be a definitive benchmark of the backend (which uses higher-level data processing libraries and includes additional filtering and aggregation), but it provides an order-of-magnitude estimate of the I/O and data volume involved in typical interactive ranges.

5.4 Limitations and threats to validity

5.4.1 External dependencies and non-determinism

Part of the scraper test suite relies on live calls to upstream services. While this provides valuable integration coverage, it introduces non-determinism due to

network availability and upstream behavior changes. For reproducible regression testing, fixture-based tests (stored JSON samples) are preferable.

5.4.2 Data quality and measurement bias

Operational datasets include missing values and outliers. In the measured window, the `arrival_delay` field includes negative values (early arrivals) and can contain extreme values. This affects aggregate statistics (mean and standard deviation) and motivates the use of robust visual summaries (boxplots) and explicit sanitization of invalid numerics.

5.4.3 Representativeness of performance numbers

Performance depends on hardware, filesystem caching, dataset density, and selected filters. Results obtained on a specific 30-day window should be interpreted as indicative rather than universal. In particular, on-demand computations over wider ranges will be slower, which is why the system includes both a maximum allowed range and a monthly precomputation pipeline.

5.4.4 Feature completeness

Some parts of the platform are intentionally scoped. For example, trajectory visualization is not implemented in the current backend (`/map/trajectories` returns `501 Not Implemented`), and optional external station enrichment depends on third-party sources.

Chapter 6

Conclusion

This thesis delivered a practical research prototype for interactive railway performance analysis, developed as an extension of the existing *Railway OpenData* project. The original repository already supported data scraping, extraction, and analysis through scripts; the main added value of this work is a complete web platform that makes those analyses accessible to non-technical users through guided filtering, map-based exploration, and reproducible views.

6.1 Summary of contributions

Starting from the requirements and UX goals described in Chapter 3, the work produced an end-to-end system that couples data engineering choices with an interactive user experience.

The main contributions are:

- **A reproducible dataset layout and lifecycle**: a filesystem-based dataset organization (daily folders with `trains.csv` and shared station metadata) with explicit upload/archive/apply/revert mechanics, enabling dataset switching without rebuilding the application.

- **A full-stack analytical web application**: a FastAPI backend exposing stations, statistics, and dataset-management APIs, paired with a React single-page frontend that unifies exploration and analytics through a shared filtering model.

- **An exploration-first workflow**: map-based station discovery with a direct map-to-statistics handoff, allowing users to move from geographic context to quantitative analysis with minimal friction.

- **Two complementary analytics modes**: an on-demand mode for custom date ranges and a monthly mode based on offline precomputation, enabling both flexibility and responsiveness.
- **Reliability safeguards for interactive usage**: range validation and clamping to the available dataset window, NaN/Inf sanitization for JSON payloads, and defensive dataset upload handling.
- **A validation strategy**: scraper-level automated tests (fixtures and integration checks) plus functional validation of key user flows and a performance characterization of dataset scale (Chapter 5).

Overall, the system bridges technical data pipelines and citizen-facing exploration of railway service performance, addressing the "accessibility gap" that arises when analytics are only available via scripts and command-line interfaces.

6.2 Discussion

The implementation (Chapter 4) demonstrates that a "files + compute" architecture can be effective for exploratory analytics when paired with pragmatic constraints and clear semantics. In particular, four design decisions proved important:

- **Explicit bounds and clamping**: limiting the maximum on-demand time window and clamping user requests to available data ranges reduces failure modes and keeps response times predictable.
- **Semantics-first aggregation**: counting unique trains (via `train_hash`) and computing delay distributions per train (one contribution per train) avoids misleading metrics that would arise from stop-level row counts.
- **Artifact-based visualization**: returning static chart images (served through a dedicated static mount) simplifies the frontend, improves robustness across browsers, and benefits from caching and precomputation.
- **Dataset versioning as a first-class feature**: archive/apply/revert mechanics make experiments reversible and improve reproducibility for demos and evaluation sessions.

The evaluation in Chapter 5 supports these choices: functional validation confirms that filters are applied consistently across the map, dashboard, and statistics sections, and that monthly and custom-range modes produce coherent results. The measured dataset scale (millions of stop rows in a 30-day slice) further motivates bounded ranges and reuse of cached artifacts.

At the same time, limitations remain. The analytics are constrained by the quality and completeness of the upstream operational data (missing values, outliers, and potential inconsistencies across sources), and some features are intentionally scoped (e.g., trajectories are not implemented in the current API). Optional external enrichment is also dependent on third-party availability. Finally, while the system is deployable via Docker Compose, production-grade deployments would require additional operational hardening (authentication/authorization, monitoring, rate limiting, and structured observability).

The complete source code is maintained in the project repository: `https://github.com/alinhi75/railway-opendata`.

6.3 Future work

Several directions can extend the platform while preserving its current strengths:

- **Trajectory visualization**: implement a reliable trajectories endpoint by either persisting per-train stop sequences during ingestion, or by introducing an offline job that generates trajectory GeoJSON artifacts.
- **Stronger automated testing**: isolate network-dependent scraper tests (mark as "slow"/"integration"), expand fixture coverage for edge cases, and add backend API-level tests for filtering semantics and range guards.
- **Scalable computation**: introduce asynchronous jobs for expensive analytics (e.g., via a task queue) and incremental pre-aggregation to reduce repeated scanning of daily CSV files.
- **Data quality monitoring**: add systematic checks for missing coordinates, encoding anomalies, and extreme delay values, and surface these diagnostics in the UI to improve interpretability.
- **Richer interaction**: allow users to compare multiple stations or corridors, and provide consistent export mechanisms for computed aggregates (CSV/JSON downloads) for downstream analysis.
- **Operational hardening**: add basic authentication, structured logging, and rate limiting to support safer public-facing deployments.

Overall, the current prototype provides a solid foundation for iterative improvements: its separation between dataset management, API-level computation, and UI-level exploration makes it feasible to evolve individual components without reworking the entire system.

Bibliography

[1] Open Knowledge Foundation. *Open Data Handbook*. n.d. URL: https://opendatahandbook.org/ (visited on 03/08/2026) (cit. on p. 7).

[2] Google. *GTFS Schedule Reference*. n.d. URL: https://developers.google.com/transit/gtfs/reference (visited on 03/08/2026) (cit. on p. 8).

[3] Google. *GTFS Realtime Reference*. n.d. URL: https://developers.google.com/transit/gtfs-realtime/reference (visited on 03/08/2026) (cit. on p. 8).

[4] Python Software Foundation. *Python 3 Documentation*. n.d. URL: https://docs.python.org/3/ (visited on 03/10/2026).

[5] FastAPI. *FastAPI Documentation*. n.d. URL: https://fastapi.tiangolo.com/ (visited on 03/10/2026).

[6] Uvicorn. *Uvicorn Documentation*. n.d. URL: https://www.uvicorn.org/ (visited on 03/10/2026).

[7] Pydantic. *Pydantic Documentation*. n.d. URL: https://docs.pydantic.dev/ (visited on 03/10/2026).

[8] python-dotenv. *python-dotenv Documentation*. n.d. URL: https://saurabh-kumar.com/python-dotenv/ (visited on 03/10/2026).

[9] python-multipart. *python-multipart Documentation*. n.d. URL: https://andrew-d.github.io/python-multipart/ (visited on 03/10/2026).

[10] pandas development team. *pandas Documentation*. n.d. URL: https://pandas.pydata.org/docs/ (visited on 03/10/2026).

[11] NumPy Developers. *NumPy Documentation*. n.d. URL: https://numpy.org/doc/ (visited on 03/10/2026).

[12] Matplotlib Development Team. *Matplotlib Documentation*. n.d. URL: https://matplotlib.org/stable/ (visited on 03/10/2026).

[13] seaborn contributors. *seaborn Documentation*. n.d. URL: https://seaborn.pydata.org/ (visited on 03/10/2026).

[14] Kenneth Reitz and Requests contributors. *Requests: HTTP for Humans*. n.d. URL: https://requests.readthedocs.io/ (visited on 03/10/2026).

[15] Leonard Richardson. *Beautiful Soup Documentation*. n.d. URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (visited on 03/10/2026).

[16] folium contributors. *folium Documentation*. n.d. URL: https://python-visualization.github.io/folium/ (visited on 03/10/2026).

[17] React. *React Documentation*. n.d. URL: https://react.dev/ (visited on 03/10/2026).

[18] React Router contributors. *React Router Documentation*. n.d. URL: https://reactrouter.com/ (visited on 03/10/2026).

[19] Vite. *Vite Documentation*. n.d. URL: https://vite.dev/ (visited on 03/10/2026).

[20] Axios contributors. *Axios Documentation*. n.d. URL: https://axios-http.com/ (visited on 03/10/2026).

[21] Leaflet. *Leaflet Documentation*. n.d. URL: https://leafletjs.com/ (visited on 03/10/2026).

[22] React Leaflet contributors. *React Leaflet Documentation*. n.d. URL: https://react-leaflet.js.org/ (visited on 03/10/2026).

[23] OpenStreetMap contributors. *OpenStreetMap Copyright and License*. n.d. URL: https://www.openstreetmap.org/copyright (visited on 03/10/2026).

[24] OpenStreetMap Foundation. *OpenStreetMap Tile Usage Policy*. n.d. URL: https://operations.osmfoundation.org/policies/tiles/ (visited on 03/10/2026).

[25] Docker, Inc. *Docker Documentation*. n.d. URL: https://docs.docker.com/ (visited on 03/10/2026).

[26] Docker, Inc. *Docker Compose Documentation*. n.d. URL: https://docs.docker.com/compose/ (visited on 03/10/2026).

[27] Trenitalia. *ViaggiaTreno REST API Endpoints*. n.d. URL: http://www.viaggiatreno.it/infomobilita/resteasy/viaggiatreno/ (visited on 03/10/2026).

[28] TrainStats. *TrainStats (altervista.org) – Train search and station statistics*. n.d. URL: https://trainstats.altervista.org/ (visited on 03/10/2026).