
Integrating Heterogeneous Mobility Data Sources for Cross-Source Traffic Analysis

A thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Computer Sciences

Department of Computer Sciences
Faculty of Sciences
Université libre de Bruxelles

Author:
Gjini Johan

Supervisor:
Professor Zimanyi Esteban

Year:
2025–2026

Abstract

This thesis addresses a practical integration problem faced by Bruxelles Mobilité: operational traffic and mobility data are produced by multiple external providers, with heterogeneous formats, update cadences, and geometry conventions. The absence of a unified data platform limits cross-source analytics, slows operational response, and increases technical overhead for data engineering teams.

The work presented in this manuscript designs and implements a unified mobility data platform built around a modular ingestion layer, a micro-batch ETL pipeline, and a PostgreSQL/PostGIS storage model. The implemented system integrates six source families used by Bruxelles Mobilité workflows: ANPR aggregations (via Kafka), Brussels traffic count APIs, Waze partner feed objects, TomTom incident/flow services, Sensys magnetometer lane-level statistics (via XML API), and VISU traffic light controller data.

The core contributions are fourfold. First, this thesis proposes an architecture that separates source adapters from canonical transformation and loading rules, improving maintainability under schema drift. Second, it defines a shared geometry strategy based on a global geometry registry and segment-level linking through a common road identifier (`fid`), enabling cross-provider comparison on a single network axis. Third, it documents an executable implementation with idempotent loading, staged processing boundaries, configurable micro-batch cycles with per-source file caps, and reproducibility practices suitable for institutional deployment. Fourth, it presents the Brussels Fast Map server—an interactive vector tile visualization tool that renders all integrated sources on a single map with time-range filtering, segment-level statistics, and traffic light capacity analysis, demonstrating the operational value unlocked by cross-source spatial unification.

The manuscript positions this engineering contribution in the existing literature on mobility data integration, geospatial databases, and stream/batch data architectures. It reports implementation-grounded results through the Brussels Fast Map visualization and explicitly distinguishes current capabilities from future work (including potential MobilityDB-native temporal extensions and production-grade dashboard deployment).

Executive Summary

Context

Bruxelles Mobilite receives mobility signals from multiple systems that were historically processed in separate technical silos. This fragmentation makes it difficult to answer simple cross-source questions (for example: congestion vs incidents vs camera counts on the same road segment and time window).

Problem

The key challenge is not only ingestion. The difficult part is to create one coherent data platform where heterogeneous sources are:

- captured reliably;
- normalized with explicit temporal and geospatial semantics;
- loaded idempotently into a single queryable database;
- reusable for operations, reporting, and future analytics.

Delivered Solution

This thesis delivers:

- a modular multi-source ingestion toolkit covering six heterogeneous providers (ANPR, Brussels traffic, Waze, TomTom, Sensys magnetometers, and VISU traffic light controllers);
- a micro-batch ETL pipeline (raw → processing → staged → loaded/processed) with configurable per-source file caps, backlog monitoring, and single-instance locking;
- a relational-spatial schema in PostgreSQL/PostGIS with shared geometry linking and traffic light signal group mapping;
- the Brussels Fast Map server: an interactive vector tile visualization tool enabling real-time multi-layer traffic monitoring with time-range filtering and traffic light capacity analysis;
- a documented architecture and implementation that can be audited and extended.

Why It Matters

The resulting platform gives Bruxelles Mobilité one technical foundation for:

- near-real-time operational monitoring;
- reproducible historical analysis;
- segment-level cross-source correlation;
- progressive industrialization of mobility data products.

Main Takeaway

A robust public-sector mobility data platform can be built incrementally with pragmatic engineering choices, as long as architecture boundaries are explicit, data contracts are documented, and operational quality controls are treated as first-class design requirements.

Acronyms

ANPR	Automatic Number Plate Recognition (aggregated traffic counts/events in this project context).
API	Application Programming Interface.
CRS	Coordinate Reference System.
EPSG	European Petroleum Survey Group code used to identify coordinate systems.
ETL	Extract, Transform, Load.
FK	Foreign Key.
GIS	Geographic Information System.
HTTP	Hypertext Transfer Protocol.
KPI	Key Performance Indicator.
MVT	Mapbox Vector Tile (binary Protocol Buffer format for tiled geospatial data).
PK	Primary Key.
PostGIS	Spatial extension for PostgreSQL.
SNAPS	Sensys Networks Access Point Server (magnetometer management platform).
SQL	Structured Query Language.
TLC	Traffic Light Controller.
TMC	Traffic Message Channel metadata block used by incident feeds.
TSS	Traffic Signal Site (intersection-level traffic signal controller identifier).
VISU	Traffic signal controller system (DB264 platform) used by Bruxelles Mobilité.
WKT	Well-Known Text geometry representation.
XML	Extensible Markup Language.

Contents

Abstract	i
Executive Summary	ii
Acronyms	iv
1 Introduction	1
1.1 Institutional Context and Problem Motivation	1
1.2 The Brussels Mobility Data Ecosystem	2
1.2.1 ANPR Aggregations	2
1.2.2 Brussels Traffic Count API	2
1.2.3 Waze Partner Feed	2
1.2.4 TomTom Traffic Intelligence	3
1.2.5 Magnetometer Sensors (Sensys/SNAPS)	3
1.2.6 VISU Traffic Light Controllers	4
1.3 Problem Statement	4
1.4 Research Questions	4
1.5 Objectives	5
1.6 Contribution Scope	6
1.7 Scope Boundaries	6
1.8 Methodology	7
1.9 Manuscript Roadmap	8
2 Background and Related Work	9
2.1 Data Engineering Foundations	9
2.1.1 The Data Engineering Lifecycle	9
2.1.2 ETL Process Models and Formal Foundations	9
2.1.3 Architecture Patterns: Warehouse, Lake, Lakehouse	10
2.1.4 ELT, the Modern Data Stack, and Data Mesh	10
2.1.5 Data Contracts and Schema Evolution	11
2.1.6 Pipeline Reliability: Idempotency, Delivery Guarantees, and Error Containment	11
2.1.7 Data Quality and Observability	12
2.2 Streaming and Ingestion Architectures	12
2.2.1 Apache Kafka as an Integration Backbone	12
2.2.2 Batch, Micro-Batch, and Streaming Tradeoffs	12
2.3 Geospatial Database Foundations	13
2.3.1 Map Matching and Segment Linking	13
2.3.2 Vector Tiles for Spatial Data Serving	14

2.4	Spatio-Temporal Database Extensions and MobilityDB	14
2.5	Smart Mobility Data Integration	15
2.5.1	Multi-Source Urban Traffic Platforms	15
2.5.2	FIWARE and Smart City Middleware	15
2.5.3	Real-Time Traffic Visualization Platforms	16
2.6	Data Source Characterisation	16
2.6.1	ANPR Data in Traffic Analytics	16
2.6.2	Crowdsourced Traffic Data: Waze	16
2.6.3	Commercial Traffic Intelligence: TomTom	17
2.6.4	Magnetometer and Inductive Loop Detection	17
2.6.5	Traffic Signal Controllers and VISU Systems	17
2.7	Public-Sector Governance and Compliance Context	17
2.8	Positioning of this Thesis within Related Work	18
2.9	Chapter Summary	18
3	Data Ingestion	19
3.1	Ingestion Architecture	19
3.1.1	Configuration-Driven Multi-Source Runner	19
3.1.2	Shared Ingestor Contract	20
3.1.3	Graceful Shutdown	20
3.2	Kafka-Based Ingestion: ANPR	20
3.2.1	Transport: Kafka Consumer	21
3.2.2	File Rotation and Flush Behavior	21
3.2.3	Shutdown Protocol	21
3.3	HTTP JSON Polling	21
3.3.1	Source Parameters	22
3.3.2	TomTom: Dual-Mode and Vector Tile Decoding	22
3.3.3	Alternative Offline Decoding Pipeline	22
3.4	HTTP XML Ingestion: Magnetometer	23
3.4.1	Transport: HTTP Polling with XML Responses	23
3.4.2	Temporal Alignment and Deduplication	23
3.4.3	Location Group Normalization	24
3.5	VISU Traffic Light Ingestion	24
3.5.1	Transport: Multi-Installation HTTP Polling	24
3.5.2	Time Window Computation	25
3.6	Operational Specifications	25
3.6.1	Cadence and Refresh Frequencies	25
3.6.2	Reliability and Fault Tolerance	25
3.6.3	Idempotency Considerations	25
3.6.4	Traceability	26
3.7	Chapter Summary	26
4	ETL Pipeline	27
4.1	Directory Convention and File-State Model	27
4.2	Pipeline Orchestration	28
4.2.1	Phase 1: Transform (Micro-Batch)	28
4.2.2	Phase 2: Load and Cleanup	29
4.2.3	Continuous Execution and Single-Instance Lock	29

4.3	Shared Transformation Mechanisms	29
4.3.1	Content-Addressable Geometry Identification	29
4.3.2	Deterministic Record Identification	30
4.3.3	Snapshot Selection and Oldest-First Strategy	30
4.4	Source Adapters	30
4.4.1	ANPR	31
4.4.2	Waze	31
4.4.3	TomTom	31
4.4.4	Brussels Traffic	32
4.4.5	Magnetometer	32
4.5	VISU Traffic Light Pipeline	32
4.5.1	Bootstrap: API Polling	33
4.5.2	Transform: Signal Event Extraction	33
4.5.3	Load: Staging to Database	34
4.5.4	Mode Inference	34
4.5.5	Analytical Views and Validation Report	35
4.5.6	Integration Scope	35
4.6	Load Pipeline	35
4.6.1	Execution Order and Dependency Awareness	35
4.6.2	Commit-Before-Move Safety Pattern	36
4.6.3	Conflict-Safe Insert Pattern	37
4.6.4	JSONB Column Handling	37
4.7	Operational Behavior	37
4.7.1	Incremental Execution	37
4.7.2	Artifact Cleanup	37
4.7.3	Error Isolation	37
4.7.4	Observability and Performance Metrics	37
4.8	Chapter Summary	38
5	Database Structure and Geometry Construction	39
5.1	Design Goals and Technology Choice	39
5.2	Schema Architecture	39
5.3	Reference Tables	41
5.3.1	<code>source</code>	41
5.3.2	<code>resource</code>	41
5.4	Traffic Fact Tables	42
5.4.1	ANPR Tables: <code>count</code> and <code>event</code>	42
5.4.2	Waze Tables: <code>waze_alert</code> and <code>waze_jam</code>	42
5.4.3	TomTom Tables: <code>tomtom_incident</code> and <code>tomtom_flow</code>	42
5.4.4	Brussels Table: <code>brussels_traffic_event</code>	43
5.4.5	Magnetometer Table: <code>magneto_lane_stats</code>	43
5.4.6	Traffic Light Signal Group Mapping	43
5.5	VISU Schema	44
5.5.1	Dimension Tables: <code>tss</code> and <code>signalgroup_dim</code>	44
5.5.2	Event Table: <code>signalgroup_event</code>	44
5.5.3	Current State Table: <code>signalgroup_latest</code>	44
5.5.4	Geometry Tables: <code>signalgroup_geometry</code> and <code>signalgroup_geometry_map</code>	44
5.5.5	Mode Tables: <code>tss_mode</code> , <code>tss_mode_cycle</code> , and <code>tss_mode_interval</code>	45

5.5.6	Temporal Mode Column: <code>tss_mode_temporal</code>	45
5.5.7	Audit Tables: <code>poll_audit</code> and <code>tss_mode_inference_audit</code>	45
5.6	Spatial Bridge: Global Geometry and Segment Linking	46
5.6.1	The <code>global_geometry</code> Table	46
5.6.2	CRS Inference and Storage Convention	46
5.6.3	WKT Construction	47
5.6.4	The <code>geolink_lignes</code> Table	47
5.7	Geometry Linking Algorithm	47
5.7.1	Point Matching	47
5.7.2	Line Matching	48
5.7.3	Ambiguity and Tie Resolution	48
5.7.4	Bootstrap Behavior	48
5.7.5	Coverage Reporting	48
5.8	Indexing Strategy	49
5.8.1	Spatial Indexes (GiST)	49
5.8.2	Temporal Indexes (BRIN)	49
5.8.3	Conventional B-Tree Indexes	50
5.8.4	Expression Index	50
5.8.5	Partial Unique Index	51
5.8.6	VISU Schema Indexes	51
5.8.7	Design Rationale Summary	51
5.9	Chapter Summary	52
6	Brussels Fast Map: Visualization and Results	53
6.1	Server Architecture	53
6.2	Vector Tile Generation Pipeline	54
6.2.1	Tile Request Processing	54
6.2.2	Supported Layers	54
6.2.3	Spatial Indexing and Query Performance	55
6.3	Interactive Dashboard	55
6.3.1	Multi-Layer Visualization	55
6.3.2	Time Range Selection	55
6.3.3	Segment-Level Inspection	55
6.3.4	Traffic Light Analysis	56
6.4	API Endpoints for Detailed Analysis	56
6.4.1	Segment Statistics (<code>/api/fid/{fid}/stats</code>)	56
6.4.2	Segment Geometries (<code>/api/fid/{fid}/geometries</code>)	56
6.4.3	Traffic Light Capacity (<code>/api/traffic-light/{id}/{num}/capacity</code>)	56
6.4.4	Traffic Light Latest Cycle (<code>/api/traffic-light/{id}/{num}/latest-cycle</code>)	56
6.4.5	Waze Jam Segment Matching	56
6.5	Results and Operational Value	57
6.5.1	Cross-Source Spatial Correlation	57
6.5.2	Temporal Pattern Analysis	57
6.5.3	Traffic Signal Performance Assessment	57
6.5.4	Coverage and Data Quality Assessment	57
6.5.5	Performance Characteristics	57
6.5.6	Limitations and Future Directions	58
6.6	Chapter Summary	58

7	Evaluation and Results	59
7.1	The Mobility Pipeline Control Center	60
7.1.1	What the Operator Sees	60
7.1.2	Evidence of Live Operation	62
7.1.3	What the Tool Changes Operationally	62
7.2	The Brussels Fast Map at a Glance	63
7.2.1	Dashboard Layout	63
7.2.2	TomTom Flow Over the Region	64
7.3	Rue Scailquin: ANPR and TomTom Agree on the Same Event	65
7.4	Avenue Charles-Quint: Signal Capacity Meets Observed Demand	66
7.5	Value Delivered	67
7.6	Known Limits	68
7.7	Research Question Answers	69
8	Conclusion and Future Work	70
8.1	What the Thesis Delivered	70
8.2	Research Questions, Settled	70
8.2.1	RQ1: Coherent Spatio-Temporal Integration	70
8.2.2	RQ2: Robust Architecture Under Heterogeneous Updates	71
8.2.3	RQ3: Operational Value	71
8.3	Contribution Statement	71
8.4	Future Work	71
8.4.1	MobilityDB: From Temporal-Aware to Temporal-Native	71
8.4.2	Production Hardening	73
8.4.3	Correction-Aware Loading	73
8.4.4	Linkage Quality Instrumentation	73
8.4.5	Further Sources and Retrospective Analytics	74
8.5	Final Perspective	74

Chapter 1

Introduction

1.1 Institutional Context and Problem Motivation

Bruxelles Mobilite is the regional administration responsible for mobility policy, road infrastructure, and public-space management in the Brussels-Capital Region. On a daily basis, the agency oversees a dense urban road network serving roughly 370 000 commuters entering the Region, tens of thousands of cyclists and public-transport riders, and a logistics fleet whose movements are increasingly monitored through automated sensing. To carry out this mandate, Bruxelles Mobilite relies on a growing constellation of data-producing systems: infrastructure-mounted Automatic Number Plate Recognition (ANPR) cameras, municipal traffic-counting endpoints, the Waze for Cities partner feed, TomTom commercial traffic intelligence services, and various open-data portals.

Each of these systems was deployed independently, at different times, by different teams or external providers, with different contractual arrangements. The consequence is a fragmented information landscape where each source exposes a partial, technically isolated view of traffic conditions. ANPR cameras stream vehicle-count aggregations through an Apache Kafka broker; Brussels traffic counters publish JSON snapshots via a REST API; Waze delivers crowdsourced alerts and jam polylines through a partner HTTP feed; TomTom distributes incident metadata and flow-level observations through proprietary API endpoints combined with Mapbox Vector Tile encoding; Sensys magnetometer sensors provide lane-level vehicle detection statistics through a proprietary SNAPS XML API; and VISU traffic light controllers expose signal group state and detector data through a time-windowed HTTP interface. These feeds differ not only in transport protocol and update cadence, but also in payload schema, identifier conventions, coordinate reference system assumptions, and temporal semantics.

Before this project, each feed could be consumed and inspected in isolation. An analyst could examine ANPR vehicle counts on a per-camera basis, or browse Waze alerts for the Brussels bounding box, or decode a TomTom flow tile for a particular zoom level. However, answering an operationally meaningful cross-source question—for instance, *“did the Waze congestion event on Avenue Louise coincide with anomalous camera counts and a TomTom incident on the same road segment during the same time window?”*—required building a custom, one-off integration script. These scripts duplicated transformation logic, introduced inconsistent spatial matching assumptions, and provided no replayability or provenance guarantees. The result was a classic data-silo pattern: abundant raw signals, but limited capacity to exploit them jointly for either real-time operational decisions or retrospective analytical studies.

This thesis addresses that gap. It documents the design, implementation, and evaluation of a unified mobility data platform that integrates these heterogeneous source feeds into a single,

coherent relational-spatial database—without collapsing the source-specific semantics that give each feed its operational meaning. Beyond integration, the thesis also presents a real-time visualization layer—the Brussels Fast Map server—that renders the unified data as interactive vector tile maps, demonstrating the analytical value unlocked by cross-source spatial unification.

1.2 The Brussels Mobility Data Ecosystem

To appreciate the integration challenge concretely, it is useful to describe the six source families that the platform targets, and the specific technical and semantic characteristics that make their unification non-trivial.

1.2.1 ANPR Aggregations

The Brussels ANPR infrastructure consists of camera installations deployed at strategic points across the regional road network. These cameras perform automated license plate recognition and publish aggregated vehicle-count data—not individual plate readings—through an Apache Kafka message bus operated by an external data provider. Each Kafka message carries a structured JSON payload that includes a hierarchical identifier system (source, resource, and aggregation identifiers), a measurement time window (start and end timestamps), a vehicle count, and geometric metadata describing both the camera location (as a point) and the measurement corridor (as a line segment). The data also includes multilingual labels (French and Dutch) for each source and resource, reflecting the bilingual administrative reality of the Brussels-Capital Region.

From an integration perspective, ANPR data presents three distinctive challenges. First, the Kafka transport requires a dedicated consumer with offset management, which is architecturally different from the HTTP polling used by other sources. Second, the hierarchical identifier structure (source $\rightarrow$ resource $\rightarrow$ aggregation) must be preserved in the relational model to support operational queries at different granularities. Third, the dual geometry semantics—point locations for cameras and line segments for measurement corridors—mean that a single source contributes both point and line features to the shared geometry registry.

1.2.2 Brussels Traffic Count API

Bruxelles Mobilite operates a traffic-counting infrastructure whose observations are published through an HTTP REST endpoint. Each API response delivers a JSON payload containing nested structures: camera-level metadata, lane-level breakdowns, and period-specific count buckets. The temporal granularity is typically short (one- to two-minute polling windows), and the data includes both the observation period and the moment the data was fetched, enabling downstream consumers to distinguish event time from ingestion time.

The integration challenge for Brussels traffic data is primarily structural: the deeply nested JSON payloads must be flattened into segment-comparable tabular records without losing the lane-level and period-level detail that gives the data its operational meaning. Additionally, the API's identifier conventions must be reconciled with the ANPR identifier hierarchy, since some counting stations overlap geographically with ANPR camera locations.

1.2.3 Waze Partner Feed

Waze, a Google-owned navigation application, provides real-time crowdsourced traffic information through its Waze for Cities (formerly Connected Citizens) programme. The partner feed delivers

two principal entity types: *alerts* (user-reported events such as accidents, hazards, road closures, and police presence) represented as point geometries, and *jams* (detected traffic congestion segments) represented as polyline geometries. Both entity types include confidence scores, severity levels, timestamps, and Waze-internal identifiers that can change between successive feed snapshots.

Waze data is valuable for its timeliness—crowdsourced reports often appear minutes before official detection systems register an event—but it introduces specific reliability challenges. Identifiers are volatile: a jam’s UUID may change between consecutive API responses even when the underlying congestion persists. Report redundancy is common: multiple users may report the same accident, producing duplicate alert entries that must be handled at ingestion or loading time. Geometry precision varies with user GPS quality, and the polyline representations of jams do not align natively with any official road network segmentation. These characteristics make Waze a high-value but high-complexity integration target.

1.2.4 TomTom Traffic Intelligence

TomTom provides commercial traffic data through two complementary services. The *incident feed* delivers structured metadata about traffic disruptions (accidents, road works, lane restrictions, weather-related closures) via an HTTP JSON API. Each incident includes a severity classification, estimated delay, TMC (Traffic Message Channel) location codes, and a line geometry representing the affected road section. The *flow service* distributes real-time traffic-level observations as Mapbox Vector Tiles—a binary Protocol Buffer format that encodes geometric features within a tiled coordinate system.

Integrating TomTom data requires solving problems that are distinct from the other three sources. Vector tile decoding involves binary parsing, coordinate transformation from tile-local pixel space to geographic coordinates via inverse Web Mercator projection, and deduplication of features that appear in overlapping tiles. The incident feed uses its own identifier scheme and geometry conventions that must be normalized to the platform’s shared spatial model. The combination of HTTP JSON and binary tile formats within a single provider makes TomTom the most technically heterogeneous source in the platform.

1.2.5 Magnetometer Sensors (Sensys/SNAPS)

The Brussels road network is instrumented with in-pavement magnetometer sensors manufactured by Sensys Networks, managed through the SNAPS (Sensys Networks Access Point Server) platform. These sensors detect individual vehicle passages by measuring disturbances in the local magnetic field, and report aggregated lane-level statistics—volume, occupancy, average speed, and median speed—through an HTTP XML API. Each API response covers a configurable measurement period (typically 60 seconds) and is organized hierarchically: location groups (representing intersections or road sections) contain approaches, which in turn contain individual lanes.

From an integration perspective, magnetometer data presents three specific challenges. First, the API returns XML rather than JSON, requiring a different parsing strategy than all other sources in the platform. Second, the location group hierarchy uses a Brussels-specific naming convention (prefixed identifiers such as “BXL 101”) that must be normalized for cross-source spatial queries. Third, the polling model requires careful temporal alignment: the API reports statistics for fixed time windows, and the ingestor must compute appropriate start times with configurable lag offsets to avoid requesting periods for which data is not yet available. Additionally, the SNAPS API may return duplicate responses for the same time window when polled at high frequency, necessitating a deduplication mechanism at the ingestion layer.

1.2.6 VISU Traffic Light Controllers

Bruxelles Mobilité manages a network of traffic signal controllers whose operational state is accessible through the VISU DB264 raw data API. This interface exposes three complementary data endpoints: *detectors* (reporting vehicle presence and occupancy at signal-controlled intersections), *signalgroups* (reporting the current phase state—green, amber, red—for each signal group at an intersection), and *ptTelegrams* (reporting public transport priority request messages). Each endpoint accepts a time window specified in milliseconds since epoch, and returns JSON payloads containing event-level data for all monitored installations within that window.

Traffic signal data differs fundamentally from the other source families in its temporal granularity and operational semantics. While traffic counts and flow observations are aggregated over measurement periods, signal group events occur at sub-second resolution and represent discrete state transitions (phase changes, detector activations). The integration challenge lies in capturing this high-resolution event stream within the platform’s file-based micro-batch architecture without losing temporal precision, while also supporting downstream analysis of signal timing patterns, cycle lengths, and capacity utilization. The VISU source also introduces a multi-installation polling model: each request targets a specific traffic signal installation (identified by a unique key), requiring the ingestor to iterate over a configured list of installations per polling cycle.

1.3 Problem Statement

The thesis addresses the following core engineering problem:

How can a unified, operationally reliable mobility data platform be designed and implemented that integrates heterogeneous source feeds into one coherent relational-spatial model, while preserving source-specific semantics, ensuring replay safety, and remaining maintainable within the operational constraints of a public-sector agency?

This problem is multi-dimensional. **Protocol heterogeneity** manifests as the coexistence of Kafka streaming, HTTP JSON polling, HTTP XML polling, and binary vector tile decoding within the same pipeline. **Schema heterogeneity** appears in the diverse identifier conventions, nesting structures, and metadata vocabularies across providers—ranging from deeply nested JSON (Brussels traffic) to hierarchical XML (magnetometers) to binary Protocol Buffers (TomTom tiles). **Geometry heterogeneity** emerges from the mix of point and line features, different coordinate reference systems (WGS84 versus Belgian Lambert 72, EPSG:31370), and fundamentally different spatial encoding strategies (GeoJSON coordinates versus tile-based vector features). **Temporal heterogeneity** is present in the different update cadences, time-window semantics, and timestamp interpretations across sources—from sub-second signal group events (VISU) to multi-minute aggregation windows (ANPR, magnetometers). **Payload format heterogeneity** adds a fifth dimension with the coexistence of JSON, XML, and binary Protocol Buffer formats across sources. All five dimensions interact within a single operational pipeline that must run reliably, be auditable, and remain comprehensible to a small engineering team.

1.4 Research Questions

The work is structured around three research questions that decompose the core problem into tractable sub-problems.

RQ1. *How can heterogeneous mobility feeds—differing in transport protocol, schema structure, geometry encoding, and temporal semantics—be integrated into one coherent spatio-temporal data model while preserving the source-specific meaning that makes each feed operationally useful?*

This question targets the data modelling and transformation dimensions of the problem. It encompasses the design of canonical staging contracts, the construction of a shared geometry registry, and the definition of a segment-linking strategy that enables cross-source spatial queries without erasing provenance information.

RQ2. *Which architectural and ETL design choices provide robust, replay-safe pipeline operation under the heterogeneous update behaviours and failure modes characteristic of multi-source mobility data?*

This question targets the systems engineering dimension. It covers the choice of a file-first micro-batch architecture over a full streaming design, the implementation of idempotent loading semantics, the commit-before-move safety invariant, and the per-source error containment strategy that prevents a single malformed payload from halting the entire pipeline.

RQ3. *What concrete operational and analytical value is unlocked once previously siloed feeds become queryable in a single relational-spatial database, and what are the practical limitations of the implemented approach?*

This question targets the evaluation and reflection dimensions. It motivates the assessment of cross-source queryability, spatial linkage coverage, and the identification of tradeoffs and boundary conditions that define what the current platform can and cannot deliver.

1.5 Objectives

The implementation objectives are deliberately concrete and verifiable, reflecting the applied engineering nature of the thesis.

1. **Multi-source ingestion.** Implement source-specific ingestors for the six source families (ANPR, Brussels traffic, Waze, TomTom, Sensys magnetometers, and VISU traffic light controllers), each encapsulating its transport protocol and writing raw payloads with provenance metadata to a standardized filesystem envelope format.
2. **Canonical transformation.** Design and implement a transformation layer that converts heterogeneous raw payloads into canonical staged entities with deterministic primary keys, normalized geometries, and explicit temporal semantics.
3. **Unified persistence.** Build a PostgreSQL/PostGIS relational schema that stores all source data in a single database with referential integrity, spatial indexing, and conflict-safe insertion semantics.
4. **Geometry-centric cross-source linking.** Construct a shared geometry registry and a segment-linking mechanism that assigns heterogeneous source geometries to a common road-network reference axis, enabling spatial joins across providers.
5. **Replay safety and auditability.** Enforce file-based lifecycle state tracking and idempotent loading patterns that make the pipeline safe to re-run after partial failures, and that preserve enough provenance information for operational forensics.
6. **Real-time visualization.** Develop an interactive map-based visualization server (Brussels Fast Map) that renders the unified data as vector tiles, enabling real-time multi-layer traffic

monitoring with time-range filtering, segment-level statistics, and traffic light capacity analysis.

7. **Documentation and transferability.** Document the architecture, data model, and operational patterns in sufficient detail to support institutional handover, code audit, and future extension by teams who did not participate in the original implementation.

1.6 Contribution Scope

This thesis contributes primarily at the applied systems level. Its value lies not in proposing a new theoretical database model or a novel spatial algorithm, but in making an integration architecture explicit, executable, reproducible, and transferable. The contribution is a defensible engineering pattern for public-sector mobility data integration under realistic constraints—heterogeneous providers, moderate infrastructure budgets, small maintenance teams, and EU governance requirements.

The central contributions are:

1. A **modular ingestion and ETL architecture** that isolates source volatility at the ingestion edge and enforces structural stability only after canonical transformation, allowing source adapters to evolve independently without rewriting downstream processing logic.
2. A **relational-spatial data model with geometry-centric cross-source linking**, implemented through a global geometry registry and a segment-level foreign-key mechanism (**fid**) that enables SQL-based spatial joins across all integrated providers.
3. **Implementation-grounded operational patterns** for replay safety (commit-before-move semantics, conflict-safe inserts), error containment (per-source isolation in both ingestion and transformation), and lifecycle traceability (file-state directories that make pipeline progress observable by filesystem inspection).
4. A **documented analysis of architectural tradeoffs** relevant to institutional deployment, including the explicit boundary conditions and upgrade paths for each design compromise (conflict-ignore versus upsert semantics, heuristic CRS inference, file-based versus broker-backed state management).

The thesis is explicit about what it does *not* contribute. It does not propose a new database theory, a novel spatial indexing algorithm, or a formal ontology for mobility data. It does not deliver a production-hardened, fully monitored deployment with comprehensive observability dashboards. It does not perform formal GDPR compliance certification. These boundaries are intentional: the contribution is scoped to the integration architecture and its immediate operational properties, with clear pointers to the work that remains for full production deployment.

1.7 Scope Boundaries

The thesis includes the following activities within its scope:

- Design and implementation of source-specific data ingestors for ANPR (Kafka), Brussels traffic (HTTP), Waze (HTTP), TomTom (HTTP + vector tiles), Sensys magnetometers (HTTP + XML), and VISU traffic light controllers (HTTP).

- Design and implementation of a layered micro-batch ETL pipeline with raw, processing, staged, loaded, and processed file-state directories, configurable per-source file caps, backlog monitoring, and single-instance locking.
- Design and implementation of a PostgreSQL/PostGIS relational schema with reference tables, fact tables, a global geometry registry, and traffic light signal group mapping.
- Implementation of geometry-to-segment linking for both point and line features.
- Development of an interactive vector tile visualization server (Brussels Fast Map) with multi-layer rendering, time-range filtering, and traffic light capacity analysis.
- Operational design analysis including replay safety, error containment, and commit ordering.
- Documentation of the architecture, data model, and tradeoffs in thesis manuscript form.

The following activities are explicitly excluded from scope:

- Full cloud-native orchestration migration (e.g., Kubernetes, managed Kafka, cloud-hosted PostgreSQL).
- Legal GDPR certification or formal privacy impact assessment.
- Production-grade observability infrastructure (centralized logging, alerting, SLA monitoring).
- MobilityDB temporal type adoption (although the schema is designed to be forward-compatible with MobilityDB extensions).

1.8 Methodology

The research follows a **design science** methodology, where the primary artifact is a software system and the evaluation criteria are drawn from the operational requirements of the institutional context. The work proceeded through iterative cycles of design, implementation, and evaluation, guided by continuous interaction with the Bruxelles Mobilite data environment.

The methodology can be summarized in four phases:

1. **Problem analysis and requirements derivation.** Characterization of the source-specific constraints, stakeholder information needs, and institutional deployment conditions that shape the platform design (Chapter 1 and Chapter 2).
2. **Architecture design.** Selection and justification of the file-first micro-batch architecture, the layered directory state model, and the geometry-centric data model, informed by the data engineering literature and the specific constraints of the Brussels context (Chapter 4 and Chapter 5).
3. **Implementation.** Development of source ingestors, transformation modules, loaders, the geometry-linking mechanism, and the visualization server, with continuous attention to replay safety, error containment, and code modularity (Chapter 3–Chapter 6).
4. **Evaluation and reflection.** Assessment of cross-source queryability, spatial linkage coverage, and operational robustness, combined with a critical analysis of tradeoffs, limitations, and future work priorities (Chapter 7 and Chapter 8).

1.9 Manuscript Roadmap

The remainder of this manuscript is organized as follows.

Chapter 2 positions the work in the existing literature. It reviews the smart mobility data integration landscape, surveys architecture patterns for data platforms (warehouse, lakehouse, streaming-first), examines the geospatial database foundations provided by PostgreSQL and PostGIS, introduces spatio-temporal extensions including MobilityDB, characterizes the data sources used in this thesis through the lens of prior academic work, reviews reliability patterns in ETL systems, and discusses the public-sector governance and compliance context.

Chapter 3 presents the ingestion layer. It explains how the platform acquires raw data through Kafka, HTTP JSON polling, HTTP XML polling, TomTom vector tile decoding, and VISU time-windowed requests while preserving provenance in a common raw-file envelope.

Chapter 4 describes the ETL pipeline. It formalizes the file-state lifecycle, the micro-batch orchestration loop, the source-specific transformation adapters, the VISU analytics pipeline, and the commit-before-move loading pattern that makes retries safe.

Chapter 5 formalizes the implemented persistence layer. It specifies the PostgreSQL/PostGIS schema, the global geometry registry, the VISU schema, the geometry-to-segment linking algorithm, and the indexing strategy that supports spatial and temporal access patterns.

Chapter 6 presents the Brussels Fast Map visualization server. It describes the vector tile generation pipeline, dashboard features, analysis endpoints, and operational value of rendering all integrated sources on a single interactive map.

Chapter 7 evaluates the platform through the Mobility Pipeline Control Center, the Brussels Fast Map, and two corridor studies. It connects the implementation back to the research questions by showing what the integrated platform changes for operators and analysts.

Chapter 8 closes the thesis by consolidating the research-question answers, stating the applied contribution, and identifying future work for MobilityDB generalisation, production hardening, correction-aware loading, linkage quality, and additional sources.

Chapter 2

Background and Related Work

This chapter positions the platform against the research and practitioner work that makes the thesis technically legible. It focuses on the bodies of literature needed to assess the contribution: data engineering lifecycles, ETL reliability, geospatial databases, spatio-temporal extensions, smart mobility integration, source-specific traffic data, and public-sector governance.

2.1 Data Engineering Foundations

2.1.1 The Data Engineering Lifecycle

Reis and Housley organise data engineering work around a five-part lifecycle, using the labels generation, ingestion, storage, transformation, and serving [38]. I use that model here as a design checklist rather than as a literal architecture diagram. Weakness in one part of the lifecycle tends to reappear elsewhere: brittle ingestion increases transformation complexity, and inflexible storage constrains serving. In this thesis, source ingestors cover generation and ingestion concerns, file-state directories provide storage checkpoints, ETL modules reshape the data, and PostgreSQL plus the Brussels Fast Map server expose the serving layer.

The same lifecycle view also emphasises *undercurrents*: concerns that cut across stages instead of belonging to only one step. Reis and Housley’s undercurrents are security, data management, DataOps, data architecture, orchestration, and software engineering. In a public-sector mobility context, governance and auditability play a similar cross-cutting role: they shape choices from credential handling during ingestion to provenance tracking when data is served.

2.1.2 ETL Process Models and Formal Foundations

Vassiliadis surveyed Extract-Transform-Load technology in 2009, mapping the field’s conceptual models, logical representations, and optimisation strategies into a unified taxonomy [46]. His survey identified three persistent challenges: (1) the semantic gap between source schemas and target schemas, which forces transformation logic to encode domain-specific knowledge that resists automation; (2) the brittleness of ETL workflows under schema drift, where upstream changes propagate failures through tightly coupled pipelines; and (3) the tension between performance optimisation (e.g., pipelining, parallelism) and maintainability. These three challenges remain active fifteen years later and shape the design decisions in this thesis.

Kimball and Caserta codified practitioner-oriented ETL patterns in *The Data Warehouse ETL Toolkit*, covering surrogate key generation, slowly changing dimension handling, error row diversion, and audit metadata attachment [25]. Their patterns assume a dimensional warehouse target. The

platform described here does not use dimensional modelling, but several Kimball-Caserta patterns transfer: deterministic key construction (applied here via content hashing), conflict-safe insertion (via PostgreSQL’s `ON CONFLICT`), and audit column propagation (`fetch_at`, `ingested_at` timestamps on every record).

2.1.3 Architecture Patterns: Warehouse, Lake, Lakehouse

Three architecture patterns dominate data platform design, each encoding a different bet on where to place schema enforcement in the data lifecycle.

The **data warehouse** pattern, formalised by Kimball and Inmon, enforces schema at write time [22, 24]. Sources pass through ETL before landing in star or snowflake models. SQL-based analytics and governance benefit from this rigidity, but the upfront schema commitment creates friction when external providers modify payload shapes without coordinated releases.

The **data lake** pattern inverts the approach: store raw files first and defer schema enforcement until read or analysis time [3]. Lakes improve auditability and replay support, but without meta-data catalogs, access controls, and documented transformation lineage they risk becoming what practitioners call a “data swamp” [3].

Armbrust et al. proposed the **lakehouse** as a synthesis: open table formats (Delta Lake, Apache Iceberg, Apache Hudi) add ACID transactions, schema enforcement, and time-travel queries on top of file-based object storage [3]. The lakehouse preserves raw data immutability while supporting warehouse-style SQL analytics. The *medallion architecture*, popularised by Databricks, structures lakehouse data into Bronze (raw), Silver (cleaned), and Gold (business-ready) layers [8].

The layered directory model in this thesis (`raw` $\rightarrow$ `processing` $\rightarrow$ `staged` $\rightarrow$ `loaded` $\rightarrow$ `processed`) mirrors the medallion architecture at the filesystem level. Raw files correspond to Bronze, staged files to Silver, and loaded database tables to Gold. The implementation substitutes local directories for cloud object storage and PostgreSQL transactions for Delta Lake commits, an adaptation driven by the deployment constraints of Bruxelles Mobilite’s infrastructure.

2.1.4 ELT, the Modern Data Stack, and Data Mesh

The shift from ETL to ELT reflects a change in where transformation happens. In ELT, raw data lands in a warehouse or lakehouse first; transformation runs inside the target engine using SQL, coordinated by tools like dbt (data build tool). dbt treats SQL transformations as software artifacts: version-controlled, tested, documented, and composed into DAGs. This approach works well when the target system has sufficient compute capacity to run transformations at query time.

The platform in this thesis uses ETL rather than ELT. Transformation runs outside the database, in Python, producing staged JSON files before loading. This choice reflects two constraints: (1) the target PostgreSQL instance runs on moderate hardware, making heavy in-database transformation less practical; and (2) the file-based intermediaries provide natural audit checkpoints that would be harder to achieve with in-database transformation alone.

Dehghani proposed *data mesh* as a decentralised alternative to centralised data platform architectures [9]. Data mesh treats each domain (in the Bruxelles Mobilite context, each source provider) as a data product owner responsible for quality, discoverability, and schema stability. The thesis does not implement data mesh, but it borrows two of its principles: domain-specific ingestion boundaries (each source adapter owns its schema knowledge) and explicit data contracts (the canonical staged format serves as the interface between source volatility and downstream stability).

2.1.5 Data Contracts and Schema Evolution

Brahmia et al. reviewed schema evolution research across relational, NoSQL, and graph databases, cataloguing strategies for handling structural changes without breaking downstream consumers [6]. Their taxonomy distinguishes between *schema modification operations* (adding, removing, or renaming attributes) and *schema versioning* (maintaining multiple schema versions simultaneously). In multi-source integration, schema evolution pressure comes from external providers who control their own release cycles. The platform handles this by isolating schema knowledge in source-specific transform modules: when TomTom changes a field name, only the TomTom transformer requires modification.

The concept of *data contracts* formalises the boundary between data producers and consumers. A data contract specifies expected schema, semantics, freshness, and quality guarantees. In this thesis, the canonical staged JSON format acts as a data contract: transform modules guarantee a documented structure, and loaders depend on that structure without inspecting raw source payloads. This separation means provider-side schema drift cannot propagate past the transform boundary.

2.1.6 Pipeline Reliability: Idempotency, Delivery Guarantees, and Error Containment

Kleppmann identified idempotency as a core requirement for reliable data systems: an operation is idempotent when executing it multiple times produces the same result as executing it once [26]. In ETL pipelines, idempotency enables safe retries after partial failures. Without it, a crashed-and-restarted pipeline risks duplicating records or corrupting aggregations. The loaders in this thesis achieve idempotency through PostgreSQL’s `INSERT ... ON CONFLICT DO NOTHING` on deterministic primary keys, ensuring that re-processing the same staged files produces no duplicate rows.

Kleppmann also distinguished three delivery semantics in distributed systems: at-most-once (messages may be lost), at-least-once (messages may be duplicated), and exactly-once (each message processed once). True exactly-once delivery requires transactional coordination between producer and consumer. In practice, most systems combine at-least-once delivery with idempotent consumers, a pattern Kleppmann calls “effectively once.” The ANPR ingestion path in this thesis follows this pattern: the Kafka consumer may re-deliver messages after a crash, but the deterministic key strategy ensures duplicates are absorbed at load time.

Mohammad’s review of Apache Kafka event-streaming systems is relevant because it treats Kafka applications as recurring architectural choices rather than as broker deployments alone [32]. The patterns discussed include transactional or exactly-once processing, change-data-capture pipelines, sagas for distributed workflows, compacted logs, tiered retention, and CQRS-style separation between write and read models. For this thesis, the important point is that robust Kafka systems often pair possible message redelivery with idempotent downstream writes, which is the strategy used by the ANPR ingestion path.

The *commit-before-move* invariant used in this thesis (database commit completes before staged files move to the loaded directory) implements a form of two-phase coordination between the database and the filesystem. If the move fails, data remains committed in the database and the staged files remain available for re-processing. If the commit fails, the files stay in staged and the loader retries on the next run. This ordering prevents the silent data loss that occurs when files are moved before their contents are persisted.

2.1.7 Data Quality and Observability

Ehrlinger and Wöß began from an inventory of 667 data quality tools and then compared 13 representative systems in detail [10]. Their analysis is useful here because it separates measurement dimensions, such as completeness, accuracy, consistency, and timeliness, from monitoring techniques such as rules, statistical checks, and machine-learning anomaly detection. In that comparison, open-source examples such as Great Expectations, Deequ, and TDDA are strongest for validation and profiling, whereas commercial systems more often include lineage and incident-management features.

The platform described in this thesis implements data quality enforcement at two levels: structural validation during transformation (rejecting malformed records) and referential integrity enforcement at load time (foreign key constraints in PostgreSQL). It does not yet implement statistical profiling or anomaly detection, a limitation discussed in section 7.6.

2.2 Streaming and Ingestion Architectures

2.2.1 Apache Kafka as an Integration Backbone

Kreps, Narkhede, and Rao designed Kafka at LinkedIn as a distributed commit log for high-throughput message processing [28]. Kafka’s architecture partitions topics across brokers, allowing consumer groups to parallelise reads while maintaining per-partition ordering guarantees. These properties made Kafka the standard infrastructure for event-driven data architectures in the decade following its release.

In this thesis, the ANPR data source uses Kafka as its delivery mechanism: an external provider publishes vehicle-count aggregations to a Kafka topic, and the platform’s ANPR ingestor consumes messages via a dedicated consumer group. Kafka’s at-least-once delivery guarantee, combined with the idempotent loader described in Section 2.1, ensures that consumer restarts do not produce duplicate records in the database.

The platform uses Kafka as a *source interface*, not as an internal communication bus. The ETL modules communicate through filesystem state (directory transitions), not through Kafka topics. This design isolates streaming complexity to the single source that requires it, while keeping the internal pipeline accessible to inspection and debugging through standard filesystem tools.

2.2.2 Batch, Micro-Batch, and Streaming Tradeoffs

Akidau et al. formalised the dataflow model for unbounded data processing, introducing event-time versus processing-time semantics and windowing strategies that enable correct results over late-arriving data [1]. Carbone et al. implemented these concepts in Apache Flink, demonstrating that a single engine can handle both batch and streaming workloads through a unified dataflow abstraction [7].

Three processing models occupy different points on the latency-complexity tradeoff:

Batch processing runs on complete, bounded datasets at scheduled intervals (e.g., nightly). Batch systems are the simplest to debug and test, but they impose latency floors equal to the scheduling interval.

Streaming processing operates on unbounded event streams with per-event or micro-window granularity. Streaming delivers the lowest latency but requires stateful operators, checkpoint management, and expertise in watermark semantics to handle out-of-order events.

Micro-batch processing partitions unbounded streams into small, bounded batches processed at short intervals (seconds to minutes). Spark Structured Streaming popularised this model, trading

a small latency increase for simpler state management and closer alignment with batch semantics [47].

The platform in this thesis operates as a file-state micro-batch system. Ingestors write raw files at polling intervals (typically 60 seconds); transformers process accumulated files in batches; loaders commit batches to PostgreSQL. End-to-end latency from source event to database availability ranges from one to five minutes depending on source polling frequency and batch accumulation. This latency is acceptable for the operational monitoring and retrospective analysis use cases that Bruxelles Mobilite prioritises, and it avoids the operational overhead of maintaining a distributed stream processing cluster.

2.3 Geospatial Database Foundations

PostgreSQL with PostGIS provides geometry storage, metric operations, coordinate reference system (CRS) transformation, and spatial indexing. PostGIS implements the OGC Simple Features specification and provides GiST (Generalised Search Tree) and BRIN index types for spatial queries [34, 36]. GiST indexes implement R-tree structures on bounding boxes, supporting range queries, intersection tests, and nearest-neighbour searches that the geometry linking strategy in this thesis requires.

For mobility-specific queries, spatial joins between point observations (e.g., ANPR camera locations, Waze alerts) and line reference networks (road segments) rely on two families of PostGIS functions. The platform uses `ST_DWithin` for proximity-based point matching, and an intersection-length ranking for line matching. Without GiST indexes, these spatial joins degrade to sequential scans that are impractical for operational datasets [36].

Nguyen et al. evaluated spatial indexing performance in PostGIS, demonstrating GiST index benefits for trajectory and road network queries while noting that optimal index strategy depends on data characteristics such as overlap degree and spatial skew [33]. Rese et al. extended this analysis to moving object data, showing that the interaction between data format, index type, and dataset characteristics affects query performance in ways that resist one-size-fits-all recommendations [39]. These findings motivate the explicit index provisioning in the implemented schema: a GiST index on `global_geometry.geom` and B-tree indexes on temporal and foreign key columns.

The shared geometry registry in this thesis, a single table centralising all source geometries with foreign key links to source-specific observations, applies a known concept (common reference networks) in an uncommon way: as a cross-provider bridge at the observation level rather than at the network topology level. Smarzaro et al. proposed a similar common conceptual schema for integrating authoritative and crowdsourced spatial data into multimodal transportation networks [43], though their approach operates at a coarser granularity than the per-observation linking used here.

2.3.1 Map Matching and Segment Linking

Map matching, the process of associating GPS traces or point observations with road network segments, is a well-studied problem in transportation research. Millard-Ball et al. presented a PostGIS-based map-matching algorithm for poor-quality GPS traces in urban environments [31]. Saki and Hagen built a scalable cloud-based framework using PostGIS for spatial assembly and Valhalla for routing-aware matching, processing about 18 million trips with more than 5 billion GPS observations [40].

The geometry linking strategy in this thesis uses simpler heuristics than probabilistic map matching: nearest segment for points within a distance threshold, maximum intersection length for lines.

This simplification is appropriate because the input geometries are provider-reported locations (camera coordinates, Waze polylines, TomTom incident geometries) rather than noisy GPS traces. The risk of false matches in dense network areas motivates future work on confidence scoring and manual validation sampling.

2.3.2 Vector Tiles for Spatial Data Serving

PostGIS 3.0 introduced server-side Mapbox Vector Tile (MVT) generation through the `ST_AsMVT` aggregate function, enabling lightweight tile servers that translate map tile requests into SQL queries executed by the database engine [36]. Performance improvements in PostGIS 3.0 include parallelisable MVT generation and integration of the Wagyu clipping algorithm for reliable geometry clipping at tile boundaries.

Rechsteiner’s vector-tile benchmark compares six open-source servers backed by PostGIS: BBOX, Ldproxy, Martin, pg_tileserv, Tegola, and TiPg [37]. The study is useful for this thesis because it evaluates tile-response behaviour under repeated request scenarios rather than treating tile generation as a purely theoretical database feature. Its results reinforce two implementation choices used later in this thesis: spatial indexes matter for latency, and simplified geometries reduce the cost of serving lower-zoom tiles.

The Brussels Fast Map server described in Chapter 6 generates MVT tiles through direct PostGIS `ST_AsMVT` queries, applying Visvalingam-Whyatt simplification (`ST_SimplifyVW`) to reduce tile payload size at lower zoom levels. This approach avoids external tile server dependencies while achieving interactive-grade response times for the Brussels road network.

2.4 Spatio-Temporal Database Extensions and MobilityDB

Güting et al. established the theoretical foundations of moving object databases by formalising abstract data types for representing moving points and regions as functions from time to space [19]. HERMES provided trajectory query capabilities on top of Oracle Spatial [35], and MovingPandas introduced efficient trajectory structures for Python-based analysis [18].

MobilityDB, developed at the Université libre de Bruxelles, extends PostgreSQL/PostGIS with temporal geometry points (`tgeompoint`), temporal floats, and temporal booleans [49, 50]. The system implements lifting algorithms that extend static PostGIS spatial operations to the temporal domain. Zimányi et al. presented MobilityDB’s type system, indexing framework, and query optimiser, demonstrating competitive performance against the BerlinMOD benchmark [50]. Bakli et al. extended MobilityDB to distributed settings through Citus integration for scalable trajectory processing [4]. Sakr, Vaisman, and Zimányi provide the most current treatment of the full mobility data pipeline in their textbook on Mobility Data Science [41].

The current implementation does not use MobilityDB’s temporal types. The PostgreSQL/PostGIS schema preserves temporal semantics (ingestion time, event intervals, publication timestamps) in all tables, and the PostGIS geometry foundation means MobilityDB extensions can be layered onto the existing schema without redesign. This sequencing is intentional: the project establishes reliable integration contracts on the stable relational/spatial foundation first, deferring temporal analytics adoption until governance and observability maturity support it.

2.5 Smart Mobility Data Integration

2.5.1 Multi-Source Urban Traffic Platforms

Scrocca et al. present TANGENT as a Semantic Web and NGSI-LD approach for integrating heterogeneous mobility data across Athens, Lisbon, Manchester, and Rennes [42]. Their work frames interoperability as a chain of tasks around discovering relevant sources, obtaining the data, harmonising semantics, combining datasets, and making the result usable by downstream applications. The four-city validation is important for this thesis because it shows that the Brussels integration problem is part of a broader European pattern. TANGENT, however, depends on ontology-driven middleware; that design brings semantic expressiveness, but also engineering and governance demands that may be difficult for a public agency without dedicated ontology staff.

Lemonde et al., working with the City of Lisbon through the ILU project, analysed opportunities and challenges of integrating multimodal traffic data from heterogeneous sources [29]. Their work documented a tension central to this thesis: the gap between diverse data stream *availability* and *operational readiness* to exploit them jointly. They found that fragmented institutional responsibilities compound technical fragmentation, a pattern also present at Bruxelles Mobilité, where source ownership, contracts, and technical interfaces distribute across multiple actors.

Zhang et al. tackled multi-source urban infrastructure data integration in Shenzhen, formalising the problem as heterogeneous model integration under data sparsity and absent ground truth [48]. Their UrbanCPS system improved inference accuracy by 21% over single-source baselines, providing empirical evidence that integration yields measurable value. This thesis aims to support a similar claim through segment-level cross-source queryability.

Bonnemaizon et al.’s European traffic dataset is a useful comparator because it shows harmonisation at a much broader geographic scale: 36 cities are represented, and the source data are attached to road segments with annual traffic measures plus variables such as truck share and speed [5]. Covering observations from 2015–2024, the dataset demonstrates that city-scale traffic harmonisation can produce reusable research infrastructure. Its per-segment, multi-attribute structure parallels the per-fid cross-source structure in this thesis, although the temporal resolution differs substantially: annual summaries in their dataset versus sub-hourly operational observations here.

2.5.2 FIWARE and Smart City Middleware

The FIWARE ecosystem represents the most mature open-source framework for smart city data integration. FIWARE’s Context Broker architecture uses the NGSI-LD API and standardised Smart Data Models to build a context information management layer holding a digital-twin representation of the city [16, 17]. Cities including Santander, Valencia, and Takamatsu have deployed FIWARE-based platforms for mobility, environmental monitoring, and disaster management [27]. CityPulse demonstrated real-time smart-city data fusion in heterogeneous environments [11].

FIWARE provides a robust reference architecture, but adoption requires middleware investment and organisational commitment to ontology governance. The approach in this thesis substitutes a lightweight relational integration layer for the full FIWARE stack, preserving the file-first philosophy suited to the current institutional context. The canonical staging entities serve an analogous purpose to FIWARE’s Smart Data Models: defining a documented, stable interface between raw source data and analytical consumption.

2.5.3 Real-Time Traffic Visualization Platforms

Jung et al. describe an open-source stack that moves traffic-simulation outputs into Elasticsearch and exposes them through a browser dashboard [23]. Their contribution is relevant here as evidence that city-scale traffic layers can be rendered interactively with current web-mapping components. The Triangulum City Dashboard addressed a different visualization problem: interactive monitoring of smart-city performance indicators, combining aggregated data with trend views across several urban categories [15].

The Brussels Fast Map server described in Chapter 6 differs from these systems in one respect: it renders *integrated* multi-source operational data rather than single-source simulation output or aggregated indicators. The multi-layer overlay of TomTom flow, Waze jams, Brussels camera counts, and ANPR observations on a single interactive map is, to the author’s knowledge, the first such implementation for the Brussels road network.

2.6 Data Source Characterisation

2.6.1 ANPR Data in Traffic Analytics

Automatic Number Plate Recognition systems have become standard infrastructure for urban traffic monitoring. Hadavi et al.’s Belgian case study is relevant here because it treats ANPR camera records as a transport-analysis dataset, supporting flow analysis, travel-time inference, and comparisons between vehicle categories [20]. Tang et al. provided a systematic review of ANPR technology in smart city applications, covering technical advancements, sensing performance factors, and cross-jurisdictional policy comparisons [44]. Lubna, Mufti, and Shah surveyed recognition pipelines, identifying a trend toward machine learning and hybrid models for improved accuracy [30]. These reviews address recognition technology but give less attention to data *integration* and downstream *analytics pipelines*, a gap this thesis partially fills.

From an integration perspective, ANPR feeds present specific characteristics: near-real-time streams (via Kafka), hierarchical identifier structures (source/resource/aggregation), and mixed point and line geometries. Chapter 5 documents the mapping of these structures to the unified relational model.

2.6.2 Crowdsourced Traffic Data: Waze

In an Iowa ATMS comparison, Amin-Naseri et al. report two headline findings for Waze: the feed matched 43.2% of crash and congestion records, and its incident reports arrived 9.8 minutes earlier on average than a probe-based alternative [2]. Hoseinzadeh et al. validated Waze speed data quality on surface streets against Bluetooth ground truth, finding similar patterns with accuracy varying by traffic conditions and segment characteristics [21].

Report redundancy is a persistent challenge: multiple users can describe the same incident, so downstream systems need either clustering or another deduplication mechanism. The same Amin-Naseri et al. study handles this by grouping reports in space and time before estimating reliability [2]. The platform in this thesis takes a narrower engineering approach: deterministic hashing of Waze entity attributes and `ON CONFLICT DO NOTHING` insertion prevent duplicate persistence, but they do not attempt probabilistic event merging.

2.6.3 Commercial Traffic Intelligence: TomTom

TomTom provides commercial traffic intelligence through incident feeds exposed as HTTP JSON responses and flow data exposed as vector-tile payloads [45]. Integration requires solving three problems that are distinct from other sources: binary tile decoding from Protocol Buffer format, coordinate transformation from tile-local pixel space to geographic coordinates via inverse Web Mercator projection, and geometry deduplication across overlapping tiles. These challenges receive less attention in open academic benchmarking than ANPR or Waze, but they are operationally significant due to the richness of TomTom’s event metadata and its broad road coverage.

2.6.4 Magnetometer and Inductive Loop Detection

Magnetometer-based vehicle detection systems (e.g., Sensys Networks) use wireless in-pavement sensors that detect magnetic field changes caused by passing vehicles. These sensors provide lane-level statistics (volume, occupancy, speed) with minimal infrastructure compared to camera-based systems. The SNAPS platform aggregates sensor data through an HTTP XML API delivering periodic statistical summaries per lane and intersection approach.¹

From an integration perspective, magnetometer feeds have regular temporal structure (fixed measurement periods aligned to clock boundaries), lane-level spatial granularity, and XML payload format, distinguishing them from the JSON-centric feeds of other sources. Integration requires parsing hierarchical XML structures, handling deduplication across polling cycles, and normalising location group identifiers for spatial linkage with the shared geometry registry.

2.6.5 Traffic Signal Controllers and VISU Systems

Traffic signal control systems generate high-resolution event data capturing phase transitions (green, amber, red), detector activations, and public transport priority requests. The VISU DB264 platform, deployed across Belgian traffic management, exposes this data through a time-windowed HTTP API returning JSON payloads for specified intervals and installations.²

Signal controller data differs from other sources in temporal resolution (sub-second events versus multi-minute aggregations) and operational semantics (discrete state transitions versus continuous measurements). Integrating signal controller data with the unified platform enables correlation between signal timing patterns and observed traffic conditions from other sources: for example, relating green phase durations at an intersection to upstream congestion detected by Waze or downstream flow changes measured by TomTom.

2.7 Public-Sector Governance and Compliance Context

The institutional context of this thesis introduces governance requirements beyond technical performance. Public-sector data platforms must balance operational effectiveness with transparency, auditability, and privacy protection.

The EU’s GDPR framework imposes obligations on public mobility agencies regarding data publication, access control, and personal data processing [13]. EU open-data policy encourages interoperable publication and reuse, reinforcing the need for clear provenance and stable data contracts [14]. The INSPIRE directive mandates spatial data infrastructure standards for public bodies

¹The SNAPS API is proprietary; its interface documentation is not publicly available and was accessed under agreement with Bruxelles Mobilité.

²The VISU DB264 API is proprietary; its interface documentation is not publicly available and was accessed under agreement with Bruxelles Mobilité.

[12]. The architectural choices in this thesis, including aggregated rather than individual-level data processing, file-based lifecycle for auditability, and separation of credentials from code, reflect these constraints without constituting formal legal compliance certification.

2.8 Positioning of this Thesis within Related Work

This thesis is an **applied engineering research artifact** that synthesises data engineering, geospatial, and ETL reliability principles into a reproducible platform for public-sector mobility data integration. It does not introduce a new database model or spatial algorithm. It demonstrates how practitioners can operationalise established patterns in an institutional context characterised by heterogeneous providers, moderate infrastructure, and governance requirements.

The differentiation from existing work lies in the **integrated scope**: source-specific ingestion across six heterogeneous providers (JSON, XML, binary, streaming transports), canonical staging, shared geometry linking, relational persistence, and interactive vector tile visualization treated as one coherent architecture rather than disconnected scripts or isolated proofs of concept. Unlike ontology-driven approaches (TANGENT, FIWARE), the thesis favours a lightweight relational integration strategy adoptable incrementally. Unlike cloud-native lakehouse architectures, it operates within local-first deployment constraints while preserving progressive data refinement principles. Compared with trajectory-centric MobilityDB research, it focuses on integration and linkage foundations on PostgreSQL/PostGIS, with MobilityDB compatibility as an explicit forward-looking design constraint.

The closest comparator in the literature remains the ILU project in Lisbon [29], which also tackles multi-source urban traffic data integration in a European city. ILU focuses on analytics and machine learning applications downstream of integration, while this thesis concentrates on the integration and persistence layer itself: the engineering foundation that enables such downstream applications. The data engineering foundations reviewed in Section 2.1, particularly the ETL lifecycle model from Reis and Housley, the reliability patterns from Kleppmann, and the schema evolution analysis from Brahmia et al., provide the theoretical grounding for the specific design choices documented in the subsequent chapters.

2.9 Chapter Summary

This literature review establishes the boundary of the thesis contribution. The platform does not claim novelty at the level of database theory, spatial indexing, map matching, or smart-city ontology design. Its contribution is the applied synthesis of those established ideas into an operational pattern for a public mobility agency: source-specific ingestion remains isolated, canonical contracts stabilise the downstream pipeline, PostgreSQL/PostGIS provides the shared spatial substrate, and the resulting database supports cross-source analysis without requiring a heavyweight semantic middleware stack.

Chapter 3

Data Ingestion

This chapter presents the data ingestion layer of the platform: the set of components responsible for acquiring raw traffic and mobility data from external providers and persisting it locally in a format suitable for downstream transformation. Ingestion is the first operational boundary of the system and its design directly determines the platform’s ability to absorb heterogeneous sources without coupling provider-specific details to analytical logic.

The chapter is organized around transport protocols rather than per-source repetition. Section 3.1 describes the architectural pattern shared by all ingestors, including the multi-threaded runner, file rotation, and envelope conventions. Section 3.2 covers the Kafka-based ANPR connector. Section 3.3 consolidates the HTTP JSON polling pattern shared by Waze, TomTom, and Brussels, including TomTom’s unique dual-mode and vector tile decoding. Section 3.4 describes the HTTP XML magnetometer connector, and Section 3.5 covers the multi-installation VISU polling model. Section 3.6 addresses operational concerns such as cadence, idempotency, and traceability.

3.1 Ingestion Architecture

3.1.1 Configuration-Driven Multi-Source Runner

All ingestors are orchestrated by a central runner that reads a JSON configuration file specifying one entry per source. Each entry declares the ingestor type, output path, polling interval, and source-specific parameters such as API keys, URLs, and query parameters. At startup, the runner iterates over the configuration list, instantiates the appropriate ingestor class, and launches it in a dedicated `threading.Thread` set as a daemon. The main thread then blocks on a join loop, allowing all ingestors to run concurrently and independently until a keyboard interrupt triggers graceful shutdown.

Listing 3.1: Simplified runner dispatch loop.

```
for entry in cfg:
    t, ingestor = start_ingestor(entry)
    runners.append((t, ingestor))

while any(t.is_alive() for t, _ in runners):
    for t, _ in runners:
        t.join(timeout=0.5)
```

This design decouples ingestor lifecycles: a transient failure in one source (for example, a TomTom API timeout) does not interrupt ingestion from other sources. The threading model was chosen

over multiprocessing because ingestors are I/O-bound (network requests and file writes) rather than CPU-bound, making thread-level concurrency sufficient.

The runner also handles environment variable resolution for configuration files that reference secrets. A `resolve_properties()` function scans Java-style properties files for `${VAR}` placeholders and substitutes them with values from the process environment, writing the resolved output to a temporary file. This mechanism allows credentials (Kafka SASL passwords, API keys) to be injected at deployment time without storing them in version-controlled configuration.

3.1.2 Shared Ingestor Contract

Despite their differences in transport protocol and data format, all ingestors follow a common behavioral contract:

1. **Polling loop:** each ingestor runs an infinite loop that sleeps for a configurable interval between iterations. The interval is source-specific: fast-changing feeds such as Waze and TomTom flow use intervals of 120 and 300 seconds respectively, while the ANPR consumer uses a continuous Kafka poll with a per-call timeout.
2. **Envelope wrapping:** every raw payload is wrapped in an envelope that includes a `fetch_at` timestamp (ISO 8601) and the source payload itself. This envelope is the interface contract between ingestion and transformation: downstream stages always receive records of the form `{"fetch_at": ..., "payload": ...}`.
3. **File-based persistence:** payloads are appended to JSON files under `data/raw/<source>/`. This ensures that ingested data survives process restarts, is human-inspectable for debugging, and can be replayed through the ETL pipeline without re-contacting the upstream API.
4. **Timed file rotation:** when the polling interval is positive, ingestors rotate output files by appending a timestamp to the base filename (for example, `feed_2025-15-01_14-30.json`). Each rotation window accumulates records in an in-memory list, flushes to disk after every append, and starts a fresh file when the interval elapses. This prevents unbounded file growth and provides natural batch boundaries for the transformation stage.

3.1.3 Graceful Shutdown

The runner supports graceful shutdown through a two-phase mechanism. On `KeyboardInterrupt`, it first invokes `stop()` on ingestors that expose such a method—notably the ANPR Kafka consumer, which must commit offsets and leave the consumer group cleanly—then joins the corresponding threads with a five-second timeout. Ingestors that rely on HTTP polling (Brussels, Waze, TomTom) terminate naturally when their daemon threads are interrupted.

3.2 Kafka-Based Ingestion: ANPR

The ANPR (Automatic Number Plate Recognition) source provides aggregated traffic measurements from camera infrastructure. Unlike the other sources, ANPR data is delivered through Apache Kafka rather than an HTTP API. The entity types extracted from ANPR payloads are detailed in Section 4.4.1.

3.2.1 Transport: Kafka Consumer

The ANPR ingestor wraps a `confluent_kafka.Consumer` configured for SASL/SSL authentication. Configuration is loaded from a Java-style properties file that specifies bootstrap servers, security protocol, and SASL credentials. The consumer subscribes to a single topic and polls in a tight loop with a configurable timeout (default: 1 second).

Each consumed message is enriched with Kafka transport metadata before being appended to the output buffer:

Listing 3.2: ANPR message envelope with Kafka metadata.

```
record = {
    "topic": msg.topic(),
    "partition": msg.partition(),
    "offset": msg.offset(),
    "key": msg.key().decode("utf-8") if msg.key() else None,
    "value": item,
}
self._current_records.append(record)
```

Preserving the topic, partition, and offset alongside the business payload enables replay from a specific offset if reprocessing is required, and supports forensic analysis when diagnosing data quality issues. The consumer operates with `auto.offset.reset=earliest` by default, ensuring that a fresh consumer group receives the full history of the topic rather than starting from the latest message.

3.2.2 File Rotation and Flush Behavior

When the `interval` parameter is positive, the ANPR ingestor rotates output files at the specified cadence. If no records arrive during a rotation window, the window is silently extended rather than writing an empty file. After each message, the entire in-memory record list is serialized to disk via `json.dumps`. This eager-flush strategy was chosen because ANPR data volume is moderate (typically a few thousand records per rotation window) and the priority is to minimize data loss on unexpected shutdown, even at the cost of repeated serialization overhead.

3.2.3 Shutdown Protocol

The ANPR ingestor exposes a `stop()` method that sets a `threading.Event` flag and calls `consumer.wakeup()` to interrupt the blocking `poll()` call from another thread. This ensures that the consumer commits its current offset and cleanly leaves the consumer group, preventing rebalance delays on the next startup. The pattern is essential in a multi-threaded runner where the stop signal originates from the main thread.

3.3 HTTP JSON Polling

Three sources—Waze, TomTom, and Brussels—share a common HTTP JSON polling pattern. Each ingestor uses an `httpx` client with a configurable timeout to issue periodic GET requests, wraps the response in the platform's standard `{fetched_at, payload}` envelope, and appends the result to the current rotation file. A time-window plus lag offset mechanism ensures that requests target finalized data rather than in-progress aggregations. Non-JSON responses are handled gracefully: Waze falls back to storing raw text with its `content_type` header; Brussels wraps unexpected responses in a `{"raw": ...}` structure.

3.3.1 Source Parameters

The polling intervals for each HTTP JSON source are summarized in Table 3.1. Source-specific payload structure details are documented in the transformation adapters (Sections 4.4.2, 4.4.3, and 4.4.4). Each source uses a configurable HTTP timeout (Waze: 20s, Brussels: 15s) and wraps responses in the platform’s standard envelope. Non-JSON responses are handled gracefully: Waze falls back to storing raw text with its `content_type` header; Brussels wraps unexpected responses in a `{"raw": ...}` structure; TomTom isolates per-tile errors inline.

3.3.2 TomTom: Dual-Mode and Vector Tile Decoding

The TomTom ingestor supports two acquisition modes within a single class, selected by the presence or absence of `tile_urls` in the configuration:

Single-endpoint mode polls one URL (typically the incident details endpoint) and decodes the response as JSON. The TomTom API key is injected as a query parameter alongside configurable base parameters (default geographic coordinates, speed units, and display options).

Multi-tile mode iterates over a list of tile URLs per polling cycle, issues individual HTTP requests for each, and aggregates the results into a single snapshot with a `tiles` array. Per-tile errors are captured inline rather than aborting the entire batch, ensuring that a single unavailable tile does not prevent ingestion of the remaining tiles:

Listing 3.3: Tile-level error isolation in TomTom multi-tile ingestion.

```
for tile_url in self.tile_urls:
    try:
        resp = self.client.request(self.method, tile_url, **kwargs)
        resp.raise_for_status()
        tiles.append({"url": tile_url, "payload": payload})
    except Exception as exc:
        tiles.append({"url": tile_url, "error": str(exc)})
```

When the `response_mode` is set to `vector_tile_geojson`, the ingestor decodes Protocol Buffer payloads inline using the `mapbox_vector_tile` library and converts tile-local pixel coordinates to WGS 84 geographic coordinates. The transformation applies the standard inverse Web Mercator projection. Given a tile at zoom level z with indices (x_t, y_t) and a feature point at tile-local coordinates (x_l, y_l) within an extent E (typically 4096), the geographic coordinates are:

$$\lambda = \left(\frac{x_t + x_l/E}{2^z} \times 360 \right) - 180, \quad \varphi = \frac{180}{\pi} \arctan \left(\sinh \left(\pi \left(1 - \frac{2(y_t + y_l/E)}{2^z} \right) \right) \right) \quad (3.1)$$

where λ is longitude and φ is latitude in degrees. The tile indices (z, x_t, y_t) are parsed from the request URL using a regular expression matching the `/z/x/y.pbf` path pattern.

The decoded output is structured as a GeoJSON `FeatureCollection` with each feature retaining its original MVT properties, augmented with a `_layer` field identifying the source vector tile layer name.

3.3.3 Alternative Offline Decoding Pipeline

For deployments where the `mapbox_vector_tile` dependency is unavailable at ingestion time, the ingestor can operate in `binary_base64` mode, storing raw Protocol Buffer bytes as Base64-encoded strings alongside size and content-type metadata. Two companion tools provide offline batch decoding:

1. `decode_vector_tile.py` extracts Base64-encoded payloads from ingestor output files and writes individual `.pbf` files to a designated directory.
2. `pbf_to_geojson.py` decodes each `.pbf` file into a GeoJSON `FeatureCollection` with the same coordinate transformation described in Equation 3.1, reading tile indices either from command-line arguments or from a `z{z}_x{x}_y{y}` naming convention in the filename.

This two-stage workflow decouples the real-time ingestion process from the computationally heavier tile decoding step, providing flexibility in resource-constrained environments.

3.4 HTTP XML Ingestion: Magnetometer

The Sensys magnetometer infrastructure provides lane-level traffic statistics through the SNAPS statistics endpoint. Unlike all other sources in the platform, the SNAPS API returns data in XML format rather than JSON. The XML field structure is detailed in Section 4.4.5.

3.4.1 Transport: HTTP Polling with XML Responses

The magnetometer ingestor uses the `requests` library with configurable timeout and SSL verification settings. Each polling cycle iterates over a configured list of location groups, issuing one HTTP GET request per group. The response XML is stored as a raw string within a structured JSON envelope that preserves full request metadata:

Listing 3.4: Magnetometer record envelope structure.

```
record = {
    "fetched_at": datetime.now().isoformat(),
    "source": {
        "system": "SNAPS",
        "service": "statistics",
        "location_group": location_group,
    },
    "interval": {
        "start": interval_start.isoformat(),
        "end": interval_end.isoformat(),
        "duration_seconds": self.period_seconds,
        "start_time_unix": start_time_unix,
    },
    "status_code": response.status_code,
    "raw_xml": raw_xml,
}
```

This design preserves the raw XML for downstream parsing while embedding it within the platform's standard JSON envelope convention, maintaining consistency with other ingestors.

3.4.2 Temporal Alignment and Deduplication

The magnetometer ingestor implements two mechanisms that are absent from other ingestors. First, **period-aligned timestamps**: when the `align_to_period` flag is set (default: `true`), the computed `startTime` is rounded down to the nearest period boundary. For a 60-second period, this ensures that successive polls request statistics for the same canonical time windows regardless of polling jitter, which simplifies downstream temporal joins.

Second, **digest-based deduplication**: after each successful response, the ingestor computes a hash digest of the XML payload and compares it against the last known digest for that location group. If the digest matches, the response is discarded as a duplicate. This mechanism prevents storage of redundant data when the API returns identical statistics across consecutive polling cycles—a common occurrence when the polling interval is shorter than the statistics period.

The ingestor also supports a configurable `lag_seconds` parameter that shifts the requested time window into the past, avoiding requests for periods where the SNAPS server has not yet aggregated sensor data.

3.4.3 Location Group Normalization

Brussels magnetometer location groups follow a naming convention where each group is prefixed with “BXL ” followed by an identifier. The ingestor normalizes group names through a configurable prefix mechanism: if a group identifier does not already start with the expected prefix, the prefix is prepended automatically. This ensures consistent group naming regardless of whether the configuration uses short identifiers (e.g., “101”) or fully qualified names (e.g., “BXL 101”).

3.5 VISU Traffic Light Ingestion

The VISU DB264 system exposes raw traffic signal data through a time-windowed HTTP API supporting three endpoint types (detectors, signalgroups, ptTelegrams). For the three endpoint types and their payload structure, see Section 4.5.

3.5.1 Transport: Multi-Installation HTTP Polling

The VISU ingestor uses `httpx` with a 20-second timeout and follows the same envelope pattern as other HTTP-based ingestors. Each polling cycle computes a time window based on the current time, a configurable lookback window (default: 3600 seconds), and a lag offset (default: 300 seconds) to avoid requesting data that the VISU system has not yet finalized.

The ingestor iterates over a Cartesian product of configured installations and endpoints, issuing one HTTP request per combination:

Listing 3.5: VISU multi-installation polling loop (simplified).

```
for installation in self.installations:
    for endpoint in self.endpoints:
        url = f"{self.base_url}/{endpoint}/{installation}"
        resp = self.client.get(url, params=params)
        record = {
            "fetched_at": datetime.now().isoformat(),
            "endpoint": endpoint,
            "installation": installation,
            "window": {"begin_ms": begin_ms, "end_ms": end_ms},
            "payload": self._parse_payload(resp),
        }
```

This multi-dimensional polling model distinguishes VISU from other ingestors, which typically poll a single endpoint per cycle. The installation list is configured per deployment, allowing operators to selectively monitor specific intersections based on operational priority.

3.5.2 Time Window Computation

The VISU ingestor supports two time-window modes. In **sliding window** mode (default), the window is computed dynamically: **end** is set to the current time minus the lag offset, and **begin** is set to **end** minus the lookback window. In **fixed window** mode, explicit **begin_ms** and **end_ms** values are used, enabling historical data retrieval for backfilling or forensic analysis. Timestamp normalization automatically converts second-precision values to millisecond-precision when needed, preventing common unit-mismatch errors.

3.6 Operational Specifications

3.6.1 Cadence and Refresh Frequencies

Table 3.1 summarizes the default polling intervals and transport mechanisms for each source. All values are configurable per deployment through the runner’s JSON configuration file, allowing operators to adjust cadence based on API rate limits and data freshness requirements.

Table 3.1: Default ingestion cadence and transport by source.

Source	Transport	Default Interval	Output Format
ANPR	Kafka consumer	1 s poll timeout	JSON
Waze	HTTP GET (JSON)	120 s	JSON
TomTom	HTTP GET (+ MVT)	300 s	JSON
Brussels	HTTP GET (JSON)	120 s	JSON
Magnetometer	HTTP GET (XML)	60 s	JSON
VISU	HTTP GET (JSON)	60 s	JSON

3.6.2 Reliability and Fault Tolerance

Each ingestor wraps its polling logic in a try/except block that catches all non-keyboard exceptions, logs the error message, and continues to the next iteration. This design ensures that transient network failures, API rate limiting responses, or malformed payloads do not terminate the ingestion process. The ANPR consumer additionally handles Kafka-specific exceptions (broker disconnections, consumer group rebalance events) through its poll error handler, and distinguishes shutdown-induced exceptions from genuine errors using the stop event flag.

3.6.3 Idempotency Considerations

At the ingestion layer, strict deduplication is intentionally not enforced: re-polling the same source within its refresh window may produce overlapping records in the raw files. This is a deliberate design decision. Deduplication is deferred to the transformation stage, where content-addressable hashing (SHA-1 over canonical JSON representations) and primary key constraints at the database level provide the actual idempotency guarantee. Keeping the ingestors stateless and append-only simplifies their implementation and avoids the need for local state management, bloom filters, or deduplication caches at the ingestion edge.

3.6.4 Traceability

Every ingested record carries a `fetch_at` timestamp that captures the moment the platform acquired the data, distinct from any event-time timestamp within the source payload. This duality enables downstream consumers to distinguish between “when did this event occur in the real world?” and “when did the platform learn about it?”, a distinction that is essential for detecting ingestion delays, assessing data freshness, computing end-to-end pipeline latency, and diagnosing temporal ordering anomalies in cross-source analysis.

For the ANPR source, the Kafka metadata (topic, partition, offset) provides an additional axis of traceability, enabling precise identification of the message position within the Kafka log for replay and audit purposes.

3.7 Chapter Summary

This chapter establishes the platform’s first guarantee: heterogeneous upstream systems become durable, timestamped, replayable raw files before any analytical interpretation is attempted. Kafka messages, JSON polling responses, XML statistics, vector tiles, and VISU event windows enter through different technical paths, but all leave ingestion with explicit provenance and a consistent filesystem boundary. That boundary is what allows the following ETL chapter to treat provider volatility as an input condition rather than a source of architectural coupling.

Chapter 4

ETL Pipeline

This chapter describes the Extract–Transform–Load (ETL) pipeline that converts raw ingested payloads into normalized, deduplicated records ready for relational persistence. The pipeline operates as a continuously looping micro-batch process that detects new raw files, transforms them through source-specific adapters with configurable per-source file caps, loads the results into PostgreSQL, and cleans up transient artifacts. A single-instance lock prevents concurrent pipeline runs from corrupting file-state transitions.

The chapter first establishes the directory convention and file-state model that governs data flow (Section 4.1), then details the orchestration logic (Section 4.2). Section 4.3 extracts the cross-cutting transformation mechanisms shared by all sources, and Section 4.4 documents the source-specific adapter logic. Section 4.5 covers the independent VISU traffic light pipeline. Section 4.6 covers the shared loading layer, and Section 4.7 addresses operational behavior including idempotency, batching, performance metrics, and cleanup.

4.1 Directory Convention and File-State Model

The ETL pipeline manages data flow through a set of well-defined directory states under **data/**. Files move through directories that encode their current processing status:

1. **data/raw/<source>/** — files written by the ingestion layer, not yet claimed by the pipeline.
2. **data/processing/<source>/** — a single file snapshot has been moved here atomically, indicating it is currently being transformed. The atomic move prevents concurrent pipeline runs from processing the same file.
3. **data/staged/<source>/<entity>/** — individual entity records produced by transformation, written as one JSON file per record (e.g., one file per Waze alert, one per ANPR count).
4. **data/loaded/<source>/<entity>/** — transient handoff location for staged files that have been successfully committed to the database and moved here by the loader.
5. **data/processed/<source>/** — transient handoff location for the original raw snapshot after transformation completes.

This convention provides several operational properties. First, the directory a file resides in *is* its state: no separate state database or metadata store is required. Second, the snapshot move from **raw** to **processing** is atomic at the filesystem level (**shutil.move**), preventing double-processing

even under concurrent execution. Third, the filesystem state remains inspectable during each cycle while the database remains the durable system of record after loading. Replay after artifact cleanup depends on reintroducing the source payload from upstream retention or external backups, not on **processed/** as a permanent archive. The lifecycle has two coupled transitions: raw snapshots move **raw** → **processing** → **processed**, while transformed entity files move **staged** → **loaded** after the database commit.

4.2 Pipeline Orchestration

The pipeline orchestrator implements a two-phase execution model.

4.2.1 Phase 1: Transform (Micro-Batch)

The orchestrator checks for the presence of raw files across all source directories (ANPR, Brussels, Waze, TomTom, Magnetometer). The pipeline supports two execution modes: a **micro-batch mode** (default, enabled via the `ETL_MICROBATCH_ENABLED` environment variable) and a legacy drain-loop mode for backward compatibility. In micro-batch mode, each cycle processes up to a configurable number of files per source, governed by per-source cap environment variables:

Listing 4.1: Micro-batch transform phase with per-source file caps (simplified).

```
caps = {
    "anpr":      int_env("ETL_MAX_FILES_ANPR_PER_CYCLE", 5),
    "brussels": int_env("ETL_MAX_FILES_BRUSSELS_PER_CYCLE", 20),
    "waze":      int_env("ETL_MAX_FILES_WAZE_PER_CYCLE", 20),
    "tomtom":    int_env("ETL_MAX_FILES_TOMTOM_PER_CYCLE", 20),
    "magneto":   int_env("ETL_MAX_FILES_MAGNETO_PER_CYCLE", 10),
}

anpr_files = snapshot_anpr_files(max_files=caps["anpr"])
for f in anpr_files:
    process_counts(f)
    process_events(f)

brussels_files = snapshot_brussels_files(max_files=caps["brussels"])
for f in brussels_files:
    process_brussels(f)

process_waze(max_files=caps["waze"])
process_tomtom(max_files=caps["tomtom"])
process_magneto(max_files=caps["magneto"])
```

The per-source file caps provide explicit backlog control: operators can tune each source independently based on its typical file size, transformation cost, and acceptable pipeline latency. For example, ANPR files tend to be larger and more expensive to transform, so a lower cap (default: 5) prevents a single source from dominating cycle time. The micro-batch design also enables backlog tracking: before and after each cycle, the pipeline records the number of pending raw files per source, and these metrics are emitted as structured JSON log lines prefixed with `[PERF]` for operational monitoring.

A noteworthy detail is that the ANPR source is processed via two separate function calls on the *same* snapshot file: `process_counts()` and `process_events()` both read the same snapshot

but extract different entity families (counts with source geometries versus events with measure geometries). This avoids reading and parsing the same large JSON file twice and ensures that both extraction paths see a consistent view of the data.

4.2.2 Phase 2: Load and Cleanup

Once the current capped transform micro-batch completes, the orchestrator invokes the load pipeline (`run_load_pipeline()`) to persist all staged records to PostgreSQL, followed by an artifact cleanup step:

Listing 4.2: Load and cleanup phase.

```
if empty_dirs >= 1:
    run_load_pipeline()
    removed = cleanup_etl_artifacts(BASE_DIR)
```

The load phase is guarded by a condition ensuring that at least one transform cycle executed, preventing unnecessary database connections when no new data is available.

4.2.3 Continuous Execution and Single-Instance Lock

When executed as a standalone process (`__main__`), the pipeline acquires a single-instance lock (`.etl_single_instance.lock`) to prevent concurrent pipeline runs from corrupting file-state transitions. The lock file contains the process PID, and stale locks from crashed processes are automatically detected and removed by checking whether the recorded PID still exists.

The pipeline then enters a continuous loop with configurable cycle sleep time (default: 15 seconds, governed by `ETL_CYCLE_SLEEP_SECONDS`). When no raw files are pending, the pipeline enters an idle wait state with a longer polling interval (default: 5 seconds, governed by `ETL_IDLE_WAIT_SECONDS`), reducing CPU usage during quiet periods. Each cycle invokes the full transform-then-load sequence. Keyboard interrupts terminate the loop cleanly, and unhandled exceptions are caught and logged without crashing, ensuring that transient issues (e.g., a temporary database outage) do not permanently halt the pipeline.

4.3 Shared Transformation Mechanisms

Before describing individual source adapters, this section extracts three cross-cutting mechanisms that are shared by all or most transformers. Centralizing these patterns here avoids repeating identical logic descriptions in each adapter section.

4.3.1 Content-Addressable Geometry Identification

The geometry hashing mechanism is central to the platform’s deduplication strategy and is shared across all source transformers. Rather than relying on provider-assigned identifiers (which may be absent or inconsistent), the system computes a deterministic hash over the canonical JSON representation of the geometry:

Listing 4.3: Content-addressable geometry hashing.

```
def geometry_hash(obj):
    return hashlib.sha1(
        json.dumps(obj, sort_keys=True).encode()
    ).hexdigest()
```

The use of `sort_keys=True` ensures that key ordering does not affect the hash, and the SHA-1 output provides a 40-character hexadecimal identifier with negligible collision probability for the expected data volumes. This approach guarantees that identical geometries from different records, different batches, or even different sources map to the same `geometry_id`, enabling natural deduplication in the global geometry table without cross-source identifier coordination. This mechanism is referenced by all source adapters in Section 4.4 and by the global geometry registry (Section 5.6).

4.3.2 Deterministic Record Identification

Every source adapter derives its primary key from the natural identifier of each observation, ensuring that replaying the same raw file produces identical keys and that `ON CONFLICT DO NOTHING` silently skips already-loaded rows. The key strategies vary by source but follow a common principle of deterministic, content-derived identification:

- **ANPR**: uses the provider-assigned `aggregation_id`, which is stable across Kafka replays.
- **Waze**: uses the `uuid` field with a fallback to a content hash when the UUID is absent.
- **TomTom incidents**: uses `props.get("id")` with a geometry-hash fallback for incidents lacking a provider ID.
- **TomTom flow**: computes a SHA-1 hash over (`geometry_id`, `fetches_at`, tile coordinates, feature index) to produce a composite `observation_id`.
- **Brussels**: constructs a composite key from `camera_id`, `period`, `slot`, and `start_ts`.
- **Magnetometer**: computes a SHA-1 hash over (`tss_id`, `lane_name`, `stats_ts`, `period`).

In all cases the key is derived from the observation's natural identifier, ensuring that replaying the same raw file produces identical primary keys and that `ON CONFLICT DO NOTHING` silently skips already-loaded rows.

4.3.3 Snapshot Selection and Oldest-First Strategy

The `snapshot_raw_file()` function, shared by ANPR, Waze, TomTom, and Brussels adapters, selects the oldest file (by modification time) matching a source-specific glob pattern, moves it atomically to `data/processing/<source>/`, and returns the new path. Empty files are detected and skipped. The oldest-first strategy ensures chronological processing order and prevents starvation of older batches when new files arrive continuously. Magnetometer files already carry timestamps in their names and follow the same move-then-process pattern.

4.4 Source Adapters

Each source adapter applies the shared mechanisms of Section 4.3 to its payload structure. This section documents only the adapter-specific logic: entity types produced, payload traversal, and source quirks.

4.4.1 ANPR

ANPR transformation is split across two modules that process the same raw snapshot file but extract different entity types. This dual-function pattern is unique to ANPR and avoids parsing the same large JSON file twice.

The `process_counts()` function extracts three entity types per message: sources (camera identifiers with multilingual names), counts (per-camera traffic volume measurements), and source geometries (camera Point positions, deduplicated via geometry hashing). The `process_events()` function extracts the complementary entity family from the same snapshot: resources (measurement path identifiers), events (speed/duration aggregates with speed-class distributions as JSONB), and measure geometries (LineStrings constructed from the `measures` array). The structural distinction between counts (referencing `source`) and events (referencing `resource`) reflects the upstream data model where a single camera participates in multiple measurement corridors.

4.4.2 Waze

The Waze transformer produces three entity types: geometries (Points for alerts, LineStrings for jams), alerts, and jams. The critical unique behavior is batch-level geometry deduplication via an in-memory `seen_geoms` set: because alerts and jams share a geometry pool within the same feed snapshot, the same geometry may appear in multiple records. The set ensures each geometry is written to `staged/waze/geometries/` at most once per batch, preventing redundant files and reducing loader overhead.

4.4.3 TomTom

The TomTom transformer handles two structurally different payload types within the same raw file, dispatching based on the presence of `incidents` or `tiles` keys.

For incidents, the transformer flattens properties from the nested `properties` block into a canonical record structure:

Listing 4.4: TomTom incident property flattening (excerpt).

```
incident_payload = {
    "incident_id": props.get("id") or geometry_hash(incident),
    "geometry_id": geometry_id,
    "fetched_at": fetched_at,
    "icon_category": props.get("iconCategory"),
    "magnitude_of_delay": props.get("magnitudeOfDelay"),
    "start_ts": props.get("startTime"),
    "end_ts": props.get("endTime"),
    "from_desc": props.get("from"),
    "to_desc": props.get("to"),
    "road_numbers": props.get("roadNumbers"),    # preserved as JSON
    "events": props.get("events"),               # preserved as JSON
    "tmc": props.get("tmc"),                     # preserved as JSON
}
```

Complex nested structures (`roadNumbers`, `events`, `tmc`) are preserved as-is for storage as PostgreSQL JSONB columns, avoiding lossy flattening of deeply nested provider-specific metadata.

For flow observations from vector tiles, the composite `observation_id` design ensures that: the same road segment observed at different times produces distinct records (temporal dimension via `fetched_at`); the same feature from the same tile at the same time produces exactly one record

(deduplication on exact replay); and different features on the same tile are distinguished by their index position. The flow record captures `road_type`, `traffic_level`, `traffic_road_coverage`, `road_closure`, and the `source_layer` from the original MVT encoding.

4.4.4 Brussels Traffic

The Brussels transformer handles the deeply nested API response structure, iterating over three nesting levels: camera identifier, measurement period, and time slot. For each valid measurement slot—defined as one where `count` is not null and timestamps are neither null nor the sentinel value `"-"`—the transformer constructs a deterministic `event_id`:

Listing 4.5: Deterministic Brussels event ID construction.

```
start_id = start_ts.replace("/", "").replace(" ", "").replace(":", "")
event_id = f"brussels_{camera_id}_{period}_{slot}_{start_id}"
```

Unlike the other sources, Brussels traffic responses do not carry explicit spatial geometry. Camera positions are instead obtained from a separate `?request=devices` endpoint and resolved at load time (see Section 4.6): the transformer produces event records keyed by `camera_id`, and the loader resolves each to a Point geometry in `global_geometry`.

4.4.5 Magnetometer

The magnetometer transformer processes XML-embedded JSON files. Unlike other transformers that work with JSON payloads, it must parse XML content extracted from the JSON envelope's `raw_xml` field. A four-level validation ladder distinguishes genuine data from error responses and empty windows:

1. **XML well-formedness**: malformed XML triggers a `skipped_malformed_xml` audit entry.
2. **Error response detection**: if the root element is `<error>`, the response is logged as `skipped_error_xml`.
3. **Root element validation**: the root must be `<statistics>`; unexpected root tags are rejected.
4. **Lane presence**: records with no `<lane>` elements produce a `skipped_no_lanes` audit entry.

This multi-level validation is more elaborate than the JSON-based transformers because the SNAPS API occasionally returns error responses or empty statistical windows.

For valid documents, the transformer produces three output types per file: statistics rows (per-lane measurements with deterministic `stat_id`), lane map rows (deduplicated TSS-to-lane mappings), and audit rows (per-record processing outcomes). All three are bundled into a single staged batch file, preserving the one-file-per-raw-snapshot correspondence.

4.5 VISU Traffic Light Pipeline

The VISU traffic light pipeline is a self-contained processing pipeline that ingests, transforms, stores, and analyses phase transition data from the Brussels DB264 signal controller network. It operates independently from the main file-based ETL pipeline described in the preceding sections, maintaining its own data directories, database schema, and orchestration logic. The pipeline requires

the MobilityDB PostgreSQL extension for temporal type support and exposes six composable stages orchestrated by a single pipeline runner:

1. **Bootstrap**: polls the VISU HTTP API and writes raw JSON files.
2. **Transform**: parses raw files into staged dimension, event, and geometry artefacts.
3. **Load**: persists staged artefacts to the VISU schema in PostgreSQL.
4. **Infer modes**: analyses the event history to detect and track signal timing modes.
5. **Views**: creates or refreshes analytical SQL views over the VISU schema.
6. **Report**: generates a validation report summarising data quality and coverage.

All six stages can be executed sequentially via `-mode full`, or invoked individually for replay and debugging.

4.5.1 Bootstrap: API Polling

The bootstrap stage reuses the VISU ingestion pattern described in Section 3.5, polling the `DB264/signalgroups` endpoint for each configured TSS installation. All responses are consolidated into a single raw JSON file per pipeline run. Each polling attempt is recorded in the `poll_audit` table with its HTTP status code, response payload count, duration, and any error message, providing a complete audit trail. An alternative replay mode allows the pipeline to skip API calls and reprocess the most recently acquired raw file, which is useful for re-processing after algorithm changes without additional API load.

4.5.2 Transform: Signal Event Extraction

The transform stage parses the consolidated raw JSON file and produces five categories of staged artefacts:

TSS rows. One entry per traffic signal site (`tss_id`), derived from the installation identifiers present in the raw payload.

Signal group dimension rows. One entry per (TSS, signal group number) pair, with the movement name resolved from a static metadata file.

Phase transition events. The core output. Each item in the VISU API payload carries a `timeline.Timestamp` anchor and an array of `EventsOffsets` entries, each associating a phase code (`Value`) with a millisecond `offset` from the anchor. The transformer reconstructs absolute event timestamps as:

$$t_{\text{event}} = t_{\text{timeline}} + \Delta t_{\text{offset}} \quad (4.1)$$

and maps integer phase codes to canonical text states (green: 48, amber: 12, red: 3, off: 0). A content-addressable `event_id` is computed as a SHA-1 hash of the (TSS, signal group, timestamp, value) tuple, ensuring idempotent loading. Within-batch natural-key deduplication on this tuple prevents duplicate events from overlapping API time windows.

Geometry mapping rows. Signal group spatial positions are sourced from a static GeoJSON reference file (the Mouvram dataset). Features are matched by their `Key` (TSS identifier) and `ID2` (signal group number) properties. A SHA-1 geometry hash is computed for each matched feature, and the mapping is written to `signalgroup_geometry_map`. Multiple geometries per signal group are supported, with the first flagged as primary.

Geometry rows. The deduplicated set of spatial geometries referenced by the mapping, written separately to permit incremental insertion.

4.5.3 Load: Staging to Database

The load stage persists all five staged artefact categories to the VISU PostgreSQL schema using `INSERT ... ON CONFLICT DO NOTHING` for dimension and event tables. In addition to these idempotent inserts, the stage performs an upsert on the `signalgroup_latest` table: for each (TSS, signal group) pair, it records the most recent phase transition event observed in the current batch. This materialised “current state” row enables real-time map rendering without scanning the full event history.

4.5.4 Mode Inference

The mode inference stage is the analytical core of the VISU pipeline. Its purpose is to detect the recurring signal timing *mode* active at each intersection and to track controller mode switches over time. A mode is defined as a characteristic combination of per-signal-group phase durations (green, amber, red, off) and inter-group phase offsets that repeats with sufficient regularity across observed cycles.

Cycle Extraction

For each TSS, phase transition events are fetched from `signalgroup_event` within a configurable lookback window (default: 3600 seconds). Events are normalised—filtered to known phase codes, sorted chronologically, and deduplicated for simultaneous or repeated values—before cycle boundaries are identified. A cycle begins at each green onset (transition to value 48). For each complete cycle between two consecutive green onsets, the algorithm accumulates the phase durations:

$$\phi_{\text{phase}} = \sum_{\text{segments}} \Delta t_{\text{segment}} \cdot \mathbf{1}[\text{segment.value} = \text{phase}], \quad \text{cycle_ms} = t_{\text{green}_{i+1}} - t_{\text{green}_i} \quad (4.2)$$

Mode Candidate Construction

Across all eligible signal groups (those with at least `min_cycles` complete cycles, default: 3), the algorithm computes the median green, amber, red, off, and total cycle durations. Phase offsets are computed relative to a reference signal group (the lowest-numbered eligible group) by measuring the median time difference between paired green onsets across cycles. All duration values are quantised to the nearest `tolerance_ms` (default: 500 ms) to absorb jitter from polling resolution and controller clock drift.

The quantised per-group parameters are serialised as a canonical JSON array sorted by signal group number and hashed with SHA-1 to produce a `signature_hash`. This hash is the stable identity of the mode: two independent observations with identical quantised parameters will match the same mode regardless of when they were collected. A quality score is also computed per mode:

$$Q = 1 - \frac{\text{MAD}(\text{cycle durations})}{\text{median}(\text{cycle durations})} \quad (4.3)$$

where MAD denotes the median absolute deviation. A score near 1.0 indicates highly regular cycles; lower scores flag controller variability or polling noise.

Mode Tracking and Interval Management

The inferred candidate is compared against the previously recorded modes for the same TSS. Four outcomes are possible, each logged in `tss_mode_inference_audit`:

- **Insufficient evidence:** fewer than `min_cycles` cycles observed; no database change.
- **Match/extend:** the candidate hash matches the currently active mode; the open interval's temporal extent is updated.
- **Create mode:** the candidate hash is novel; a new record is written to `tss_mode` and `tss_mode_cycle`, and a new open interval is created in `tss_mode_interval`.
- **Switch mode:** the candidate hash differs from the active mode after `mode_switch_mismatches` consecutive mismatches (default: 2). The current interval is closed with reason `'switch'` and a new interval is opened for the incoming mode.

A partial unique index ON `tss_mode_interval(tss_id)` WHERE `status = 'open'` enforces the invariant that at most one mode interval is open per TSS at any time. The MobilityDB `TTEXT` column in `tss_mode_temporal` stores the complete temporal sequence of mode identifiers as a native temporal object, enabling MobilityDB's interval predicates for mode-at-time queries.

4.5.5 Analytical Views and Validation Report

The views stage creates three SQL views over the VISU schema:

- **v_signalgroup_latest_map:** joins `signalgroup_latest`, `signalgroup_dim`, and the geometry tables to expose the current phase state and spatial location of each signal group in GeoJSON format, used by the Brussels Fast Map live signal layer.
- **v_signalgroup_last_hour_events:** a time-filtered view over `signalgroup_event` for the trailing hour, used by the capacity analysis API endpoint.
- **v_signalgroup_coverage:** aggregates per-TSS statistics on total, mapped, and unmapped signal groups and joins with recent event counts to quantify both spatial and temporal data coverage.

The validation report stage queries these views and the underlying tables to generate a coverage and quality summary per pipeline run, reporting events ingested, modes inferred, and signal groups without resolved geometry.

4.5.6 Integration Scope

Architecturally, the pipeline is fully decoupled from the main ETL system: it does not share the file-state directory model, is not invoked by the main ETL orchestrator, and maintains its own isolated PostgreSQL schema (Section 5.5). It runs as a separate periodic process, with its inferred mode and current-cycle data consumed by the Brussels Fast Map server's traffic light API endpoints (Section 6.4). The pipeline depends on the MobilityDB PostgreSQL extension for temporal type support, and the monitored TSS list is configured per deployment rather than discovered dynamically from the signal network.

4.6 Load Pipeline

4.6.1 Execution Order and Dependency Awareness

The load pipeline executes loaders in a fixed order that respects foreign key dependencies:

1. **Schema bootstrap:** `create_tables()` issues `CREATE TABLE IF NOT EXISTS` and `CREATE INDEX IF NOT EXISTS` for all tables and indexes, making the schema self-initializing on first run. Table creation is performed only on the first cycle and cached thereafter to avoid unnecessary DDL overhead.
2. **ANPR dimensions:** `resources` → `events` → `measures` → `sources` → `counts` → source geometries (satisfying FK from `count` to `source`, from `event` to `resource`).
3. **Waze:** `geometries` → `alerts` → `jams` (satisfying FK from `alerts/jams` to `global_geometry`).
4. **TomTom:** `geometries` → `incidents` → `flow` (same FK pattern).
5. **Magnetometer:** lane statistics are loaded in a single step, as the magneto staged batch files contain pre-joined statistics, lane map, and audit rows.
6. **Brussels:** device geometries (fetched from the Brussels Mobility devices API, inserted into `source` and `global_geometry`) → `events` (referencing camera positions via `camera_id`). Brussels geometry refresh is scheduled on a configurable interval (default: 60 minutes) rather than every cycle, reducing API load.
7. **Geometry linking:** cross-source segment assignment via `link_sources_to_troncons()`, detailed in Chapter 5. The linking step is triggered conditionally: it runs when new geometries were inserted during the current cycle, or periodically every N cycles (configurable via `ETL_GEOLINK_EVERY_N_CYCLES`, default: 20) as a safety net.
8. **Traffic light linking:** after geometry linking, the pipeline synchronizes traffic light signal group positions to the road segment network via `link_traffic_lights_to_segments()`, enabling cross-referencing between signal timing data and traffic observations on the same segments.

4.6.2 Commit-Before-Move Safety Pattern

All loaders implement the commit-before-move pattern through the shared `commit_and_move_batch()` function. Staged files are processed in configurable batches (default: 1000 files, governed by the `ETL_DB_BATCH_SIZE` environment variable). After each batch of `INSERT` statements, the database transaction is committed *before* the corresponding staged files are moved to `loaded/`:

Listing 4.6: Commit-before-move pattern.

```
def commit_and_move_batch(conn, pending_paths, staged_root, loaded_root):
    if not pending_paths:
        return 0
    conn.commit()           # database durable first
    for path in pending_paths:
        move_to_loaded(path, staged_root, loaded_root)
    pending_paths.clear()
```

This ordering is safety-critical. If the process crashes after `commit` but before `move`, the files remain in `staged/` and will be reprocessed. Because all inserts use `ON CONFLICT DO NOTHING`, reprocessing is idempotent—the database silently skips already-committed rows. The reverse ordering (`move` then `commit`) would risk silent data loss: a crash after moving but before committing would leave records in `loaded/` without database rows, and the pipeline would never retry them.

4.6.3 Conflict-Safe Insert Pattern

Every loader uses parameterized `INSERT ... ON CONFLICT (primary_key) DO NOTHING` statements. This provides idempotency at the database level: the same staged file can be loaded arbitrarily many times without duplicating rows. The tradeoff is that late corrections to existing records are silently ignored rather than applied. For sources where late updates are expected, an `ON CONFLICT DO UPDATE SET ...` policy can be introduced on a per-table basis, but the current design prioritizes insertion safety and operational simplicity.

4.6.4 JSONB Column Handling

For tables with JSONB columns (`speed_classes` in `event`, `road_numbers/events/tmc` in `tomtom_incident`), the loaders serialize Python dictionaries or lists via `json.dumps()` before passing them as bind parameters. This explicit serialization ensures that the PostgreSQL driver receives a valid JSON string rather than a Python object, avoiding type coercion issues.

4.7 Operational Behavior

4.7.1 Incremental Execution

The pipeline processes only newly arrived raw files per cycle. Files moved out of `raw/` are not reconsidered by the next cycle unless they are deliberately reintroduced from an upstream replay source or external backup. This incremental model keeps per-cycle runtime proportional to the configured micro-batch size and the volume of new data rather than the total historical volume, enabling sustained long-term operation.

4.7.2 Artifact Cleanup

After a successful load cycle, the cleanup step removes all transient files from the `staged/`, `processed/`, `loaded/`, and `processing/` trees. Directory structures are preserved to avoid breaking subsequent runs.

4.7.3 Error Isolation

Errors during transformation are isolated per source: a malformed Waze payload does not prevent ANPR or TomTom processing. At the load level, errors trigger a per-loader transaction rollback, but subsequent loaders still execute. The entire load phase is wrapped in a top-level exception handler that logs errors without crashing the continuous loop, ensuring that transient database outages do not permanently halt the pipeline.

4.7.4 Observability and Performance Metrics

Each pipeline stage produces console log messages with source-tagged prefixes (e.g., `[ANPR]`, `[LOAD]`, `[GEOLINK]`) that support basic log filtering and operational monitoring in deployment environments.

Additionally, each micro-batch cycle emits a structured JSON performance log line prefixed with `[PERF]` that captures comprehensive cycle metrics:

Listing 4.7: Structure of per-cycle performance metrics.

```
perf = {  
    "cycle": cycle_index,
```

```
"mode": "microbatch",
"duration_s": duration_s,
"backlog_before": {"anpr": 3, "brussels": 12, ...},
"backlog_after": {"anpr": 0, "brussels": 0, ...},
"caps": {"anpr": 5, "brussels": 20, ...},
"transform": { ... per-source transform stats ... },
"load_success": True,
"cleanup_removed": 47,
"load": { ... per-loader insertion metrics ... },
}
```

The backlog tracking (files pending before and after each cycle) enables operators to detect sources that are accumulating faster than the pipeline can process them, providing early warning for capacity planning. The per-source file caps in the `caps` field make the cycle's processing bounds explicit, supporting reproducible performance analysis.

4.8 Chapter Summary

The chapter shows how the platform converts raw-source heterogeneity into stable persistence-ready records without depending on a heavyweight orchestration stack. Its main result is the combination of three operational properties: deterministic transformations make replays converge on the same identifiers, commit-before-move loading prevents silent data loss between the filesystem and PostgreSQL, and per-source isolation keeps malformed or delayed feeds from stopping the full pipeline. Together these properties turn the ingestion outputs from the previous chapter into a repeatable data-production process that can be monitored, retried, and extended source by source.

Chapter 5

Database Structure and Geometry Construction

This chapter defines the persistence layer that receives the ETL outputs and makes them queryable across sources. It explains how PostgreSQL, PostGIS, source-specific fact tables, a global geometry registry, and road-segment linking combine into the common analytical model used by the Fast Map and the evaluation chapter.

5.1 Design Goals and Technology Choice

The platform’s persistence layer must satisfy three competing requirements: it must preserve source-native information without premature abstraction, it must support incremental and idempotent loading from a continuous ETL pipeline, and it must enable spatial joins across all providers through a shared geometry reference system. These requirements led to the selection of PostgreSQL with the PostGIS spatial extension as the database stack, deployed under the database name `mobilitydb` to reflect compatibility with MobilityDB temporal types should they be introduced in a future iteration.

PostgreSQL was chosen over document-oriented or columnar alternatives for several reasons. The relational model enforces referential integrity between fact tables and their geometry references, which is essential when six independent sources must converge onto a shared spatial index. PostGIS provides mature implementations of spatial predicates (`ST_DWithin`, `ST_Intersects`), transformations (`ST_Transform`), and measurements (`ST_Distance`, `ST_Length`) required by the geometry linking algorithm. The JSONB column type additionally allows preserving complex nested provider metadata (TMC codes, speed class distributions, event lists) without requiring full relational normalization of every upstream field.

The connection layer uses `psycopg` (v3) and connects to PostgreSQL with configurable credentials. In production, these parameters are externalized to environment variables or a secrets manager.

5.2 Schema Architecture

The current deployment separates the thesis implementation across three PostgreSQL schemas: `rt` for runtime observations, `static` for shared reference and spatial bridge tables, and `tlc` for the VISU traffic-light pipeline. Within those schemas, the model is organized into four logical tiers:

1. **Reference tables** (`static.source`, `static.resource`): dimension-like entities that represent the physical or logical origins of traffic measurements. These are loaded first and referenced by foreign keys from fact tables.
2. **Traffic fact tables**: the `rt` schema stores one table per source–entity combination for ANPR, Waze, TomTom, Brussels traffic counters, and Sensys magnetometers. Each table has a text-typed primary key derived from the source provider’s identifier or from a content-addressable hash computed during transformation.
3. **Spatial bridge tables** (`static.global_geometry`, `static.geolink_lignes`): the geometry registry and the reference road network. These tables enable cross-source spatial analysis by linking heterogeneous source geometries to a common set of road segment identifiers.
4. **Traffic signal tables**: the VISU/traffic light schema, `tlc`, stores signal-site dimensions, signal-group events, current-state rows, geometry mappings, and inferred mode histories. Static bridge tables connect these signal groups to road segments and to Sensys magnetometer lanes.

The resulting fact-to-geometry relationships are summarized in Figure 5.1.

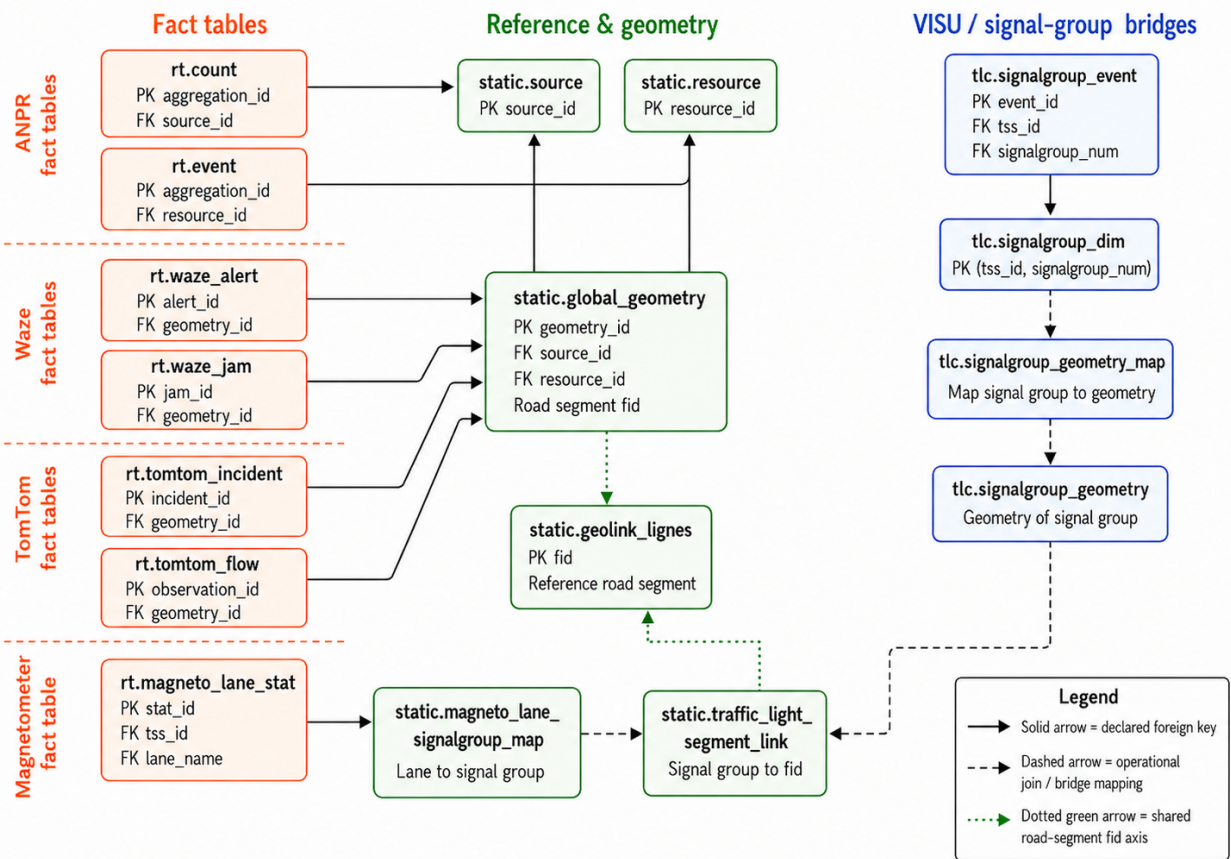


Figure 5.1: Fact-table links to the global geometry register and the road-segment bridge. Solid arrows denote declared foreign keys, dashed arrows denote operational bridge joins, and dotted green arrows denote the shared `fid` axis.

5.3 Reference Tables

5.3.1 source

The **source** table serves as the master registry of data-producing entities across all providers. For ANPR, each camera is registered as an individual source; for Waze and TomTom, a single provider-level entry (`'waze'`, `'tomtom'`) is seeded at schema creation time; for Brussels, each traffic-counting camera is registered as a separate source during the device-position loading step. Each entry records a `source_id` (used as the primary key), an `external_name` (extracted from the English translation for ANPR, or from the device description for Brussels cameras), and a `translations` JSONB column to preserve all available language variants without schema expansion.

5.3.2 resource

The **resource** table parallels **source** but represents measurement paths (the logical corridor between two ANPR cameras over which speed and duration are aggregated). The `resource_id` primary key, `external_name` (French translation), and `translations` JSONB column follow the same pattern

as **source**. Separating sources and resources into distinct tables reflects the upstream data model, where a single counting source may participate in multiple measurement resources.

5.4 Traffic Fact Tables

5.4.1 ANPR Tables: count and event

The **count** table stores per-camera traffic volume measurements. Each row is keyed by **aggregation_id** (a provider-assigned unique identifier) and references **source(source_id)** via a foreign key. Temporal bounds are stored as **TIMESTAMPTZ** columns (**start_ts**, **end_ts**) alongside provenance timestamps (**creation_ts**, **update_ts**). The measurement itself comprises **count** (integer vehicle count) and **vehicle_type** (text classification).

The **event** table stores per-resource speed and duration aggregates. Its structure mirrors **count** but references **resource(resource_id)** and adds **median_speed** (integer, km/h), **median_duration** (integer, seconds), and **speed_classes** (JSONB mapping speed brackets to observation counts). The JSONB storage for speed classes avoids the need for a separate bridge table while preserving the full distribution information provided by the source.

A composite index on (**source_id**, **start_ts**) accelerates the most common analytical query pattern: time-series rollups filtered by camera.

5.4.2 Waze Tables: waze_alert and waze_jam

The **waze_alert** table stores point-based traffic event reports. Each alert is keyed by **alert_id** (the Waze UUID or a fallback content hash) and references **global_geometry(geometry_id)** for its spatial position. Alert metadata includes **alert_type** and **subtype** (Waze’s hierarchical event classification), **reliability** and **confidence** (integer quality scores), **street** and **city** context, and **report_ts** (the publication timestamp, converted from epoch milliseconds during loading).

The **waze_jam** table stores linestring-based congestion segments. Each jam is keyed by **jam_id** and references the global geometry table. Traffic attributes include **speed** (m/s), **speed_kmh**, **delay** (seconds of additional travel time), **length** (meters), and **level** (0–5 severity scale). Temporal context is captured via **start_ts/end_ts** (feed-level bounds) and **pubMillis** (jam-level publication epoch).

5.4.3 TomTom Tables: tomtom_incident and tomtom_flow

The **tomtom_incident** table stores traffic incidents with the richest metadata of any source in the platform. Beyond the standard **incident_id**, **geometry_id**, and temporal bounds, each row carries **icon_category** (TomTom’s numeric incident type code), **magnitude_of_delay** (0–4 scale), **delay** and **length** measurements, **from_desc/to_desc** (human-readable road descriptions), and three JSONB columns preserving complex nested structures:

- **road_numbers**: an array of road identifiers associated with the incident.
- **events**: a list of sub-events with their own codes and descriptions.
- **tmc**: Traffic Message Channel location codes and related metadata for integration with TMC-based navigation systems.

These JSONB columns were chosen over full relational decomposition because the TomTom metadata schema is complex, partially documented, and subject to change. Storing it as JSONB preserves the full payload for future analysis while keeping the core relational schema stable.

The `tomtom_flow` table stores per-feature traffic flow observations from vector tiles. Each observation is keyed by a content-addressable `observation_id` (hash of geometry, timestamp, and tile coordinates) and references the global geometry. Tile provenance is captured via `tile_z/tile_x/tile_y` coordinates, and traffic state via `road_type`, `traffic_level` (continuous congestion indicator), `traffic_road_coverage`, and `road_closure` (boolean). Composite indexes on (`geometry_id`, `fetches_at` DESC) and (`fetches_at`) support time-series queries and spatial filtering respectively.

5.4.4 Brussels Table: `brussels_traffic_event`

The `brussels_traffic_event` table stores camera-based traffic measurements from the Brussels Mobility API. Each row is keyed by a deterministic `event_id` (constructed during transformation, see Section 4.4.4). The table includes `fetches_at` (platform acquisition time), `request_date` (API-reported time), `period` (measurement granularity label), temporal bounds, and three measurement columns: `count` (integer), `speed` (double precision, km/h), and `occupancy` (double precision, fraction). The `camera_id` column serves as the spatial join key: each `camera_id` corresponds to a `source_id` in `global_geometry`, where the camera’s Point position is stored after being fetched from the Brussels Mobility devices endpoint at load time.

5.4.5 Magnetometer Table: `magneto_lane_stats`

The `magneto_lane_stats` table stores per-lane traffic statistics from Sensys magnetometer sensors. Each row is keyed by a deterministic `stat_id` (SHA-1 hash of TSS identifier, lane name, statistics timestamp, and period; see Section 4.4.5). The table carries `tss_id` (the traffic signal site or location group identifier), `lane_name` (individual lane designation), temporal bounds (`interval_start`, `interval_end` as `TIMESTAMPZ`), `period_seconds` (measurement duration), and four measurement columns: `volume` (integer vehicle count), `occupancy` (double precision, 0.0–1.0), `speed_average` (double precision, km/h), and `speed_median` (double precision, km/h). A `fetches_at` timestamp records platform acquisition time, and `raw_file_name` preserves the provenance link to the original ingested file.

An associated `magneto_lane_map` table provides a dimensional reference mapping TSS identifiers to their constituent lane names, supporting analytical queries that aggregate across lanes or filter by specific intersection approaches.

The `magneto_poll_audit` table records the processing outcome for each raw record: `parsed`, `skipped_missing_raw_xml`, `skipped_malformed_xml`, `skipped_error_xml`, or `skipped_no_lanes`. This audit trail enables systematic monitoring of ingestion quality and supports root-cause analysis when data gaps are observed.

5.4.6 Traffic Light Signal Group Mapping

The `traffic_light_segment_link` table links traffic signal group identifiers to road segment `fid` values in the `geolink_lignes` reference network. Each row records a `signalgroup_id`, `signalgroup_num`, optional `geometry_id`, and matched `fid`, enabling SQL joins between signal timing data from the VISU pipeline and traffic observations from other sources on the same road segment. This mapping is refreshed by the `link_traffic_lights_to_segments()` step in the load pipeline, which uses spatial proximity matching between signal group positions and reference road segments.

5.5 VISU Schema

The VISU pipeline (Section 4.5) maintains its own isolated PostgreSQL schema, `tlc`, separate from the `rt` runtime tables and the `static` reference tables used by the other sources. The schema name remains configurable at pipeline invocation time to support multiple parallel deployments or schema versioning. The schema requires the MobilityDB extension and is bootstrapped automatically by `create_tables()` with advisory locking to prevent concurrent initialisation races.

5.5.1 Dimension Tables: `tss` and `signalgroup_dim`

The `tss` table is the root dimension, with one row per traffic signal site (`tss_id`, text primary key). An `is_active` flag and `created_at/updated_at` timestamps support lifecycle management as the monitored site inventory evolves.

The `signalgroup_dim` table stores one row per (TSS, signal group number) pair, with `tss_id` and `signalgroup_num` forming the composite primary key. Each row records the provider-assigned `signalgroup_id` string, a human-readable `movement_name` (e.g., “pedestrian north”), and a `geometry_status` flag (`'mapped'` or `'missing'`) indicating whether a spatial geometry has been resolved for this signal group.

5.5.2 Event Table: `signalgroup_event`

The `signalgroup_event` table is the primary fact table, storing individual phase transition observations. Each row is keyed by a content-addressable `event_id` (SHA-1 hash of TSS, group number, timestamp, and phase value). The core observation columns are:

- `event_ts`: the absolute `TIMESTAMPTZ` of the phase transition, reconstructed from the API timeline anchor and offset.
- `value_int`: the integer phase code (green: 48, amber: 12, red: 3, off: 0).
- `state_text`: the canonical text label derived from `value_int`.
- `offset_ms`: the millisecond offset from `timeline_ts` used to reconstruct the event.

A natural-key unique constraint on (`tss_id`, `signalgroup_num`, `event_ts`, `value_int`) provides database-level deduplication. The table carries three B-tree indexes: a composite (`tss_id`, `event_ts` DESC) index for time-series queries per TSS, a standalone `event_ts` index for global temporal scans, and a (`value_int`, `event_ts`) index supporting phase-specific lookups.

5.5.3 Current State Table: `signalgroup_latest`

The `signalgroup_latest` table maintains a single row per (TSS, signal group) pair recording the most recently observed phase. This materialised view pattern avoids full event history scans in latency-sensitive contexts such as live map rendering. The primary key is the same (`tss_id`, `signalgroup_num`) composite as `signalgroup_dim`, and the table is upserted on every load cycle.

5.5.4 Geometry Tables: `signalgroup_geometry` and `signalgroup_geometry_map`

The `signalgroup_geometry` table stores the spatial footprints of signal groups as PostGIS `GEOMETRY` objects, with a `geometry_id` (SHA-1 hash of the geometry payload) as the primary key. A GIST

spatial index supports proximity and intersection queries. Geometries are populated from the static Mouvram GeoJSON reference during the transform stage.

The `signalgroup_geometry_map` bridge table links (TSS, signal group number) pairs to geometry identifiers, supporting the case where a single signal group has multiple associated geometries (e.g., stop line and detector loop). The `is_primary` flag designates the canonical geometry, and the `match_method` column records the provenance of the mapping (default: `'key_id2'`, indicating a match on the Mouvram Key/ID2 property pair).

5.5.5 Mode Tables: `tss_mode`, `tss_mode_cycle`, and `tss_mode_interval`

These three tables together store the output of the mode inference algorithm (Section 4.5.4).

The `tss_mode` table defines the vocabulary of known signal timing modes. Each mode is identified by a `mode_id` (text primary key) and carries a `signature_hash` (SHA-1 over quantised per-group cycle parameters), a sequential `mode_seq` per TSS, and the inference parameters used to create it (`tolerance_ms`, `min_cycles_evidence`). Unique constraints on (`tss_id`, `mode_seq`) and (`tss_id`, `signature_hash`) ensure that each mode is registered at most once per TSS.

The `tss_mode_cycle` table stores the per-signal-group phase duration parameters for each mode. Each row records median `green_ms`, `amber_ms`, `red_ms`, `off_ms`, `total_cycle_ms`, and `phase_offset_ms` relative to the reference signal group, along with the `sample_cycles_count` and a `quality_score` (0.0–1.0) derived from cycle-to-cycle regularity.

The `tss_mode_interval` table records when each mode was active. Each row captures the TSS, the active mode, a `valid_from` timestamp, and an optional `valid_to` (null while the mode is still active). The `status` column is either `'open'` (current) or `'closed'`, and a `close_reason` records whether the interval ended due to a mode `'switch'`, an extended period without new data (`'extended_no_change'`), or a forced recomputation (`'recompute'`). A partial unique index ON `tss_mode_interval(tss_id)` WHERE `status = 'open'` guarantees that at most one interval is open per TSS at any time.

5.5.6 Temporal Mode Column: `tss_mode_temporal`

The `tss_mode_temporal` table stores one row per TSS with a MobilityDB TTEXT column (`mode_t`) that encodes the complete sequence of mode identifiers as a native temporal text object. This enables MobilityDB’s temporal predicates—such as mode-at-time lookups or temporal intersection with external event windows—on the full mode history without joining across `tss_mode_interval` rows.

5.5.7 Audit Tables: `poll_audit` and `tss_mode_inference_audit`

The `poll_audit` table records the outcome of every API polling attempt during the bootstrap stage: HTTP status code, payload size, duration, and any error message. It is indexed on `run_id` to support per-run diagnostics.

The `tss_mode_inference_audit` table records one row per TSS per inference run, capturing the analysis window, event and cycle counts, the candidate signature hash, the matched or created mode identifier, and the action taken (`match_existing`, `create_mode`, `extend_mode`, `switch_mode`, or `insufficient_evidence`). A `details` JSONB column preserves additional diagnostic information. This table is the primary observability surface for validating and tuning the mode inference algorithm over time.

5.6 Spatial Bridge: Global Geometry and Segment Linking

5.6.1 The `global_geometry` Table

The `global_geometry` table is the central spatial registry of the platform. Every geometry produced by any source—ANPR camera points, ANPR measure corridors, Waze alert points, Waze jam lines, TomTom incident geometries, TomTom flow segment geometries, and Brussels camera positions—is stored here with a content-addressable `geometry_id` as primary key. The table carries:

- **geom**: a PostGIS `GEOMETRY` column storing the actual spatial object (Point, LineString, or MultiLineString).
- **geom_type**: a text label recording the geometry type, used by the linking algorithm to dispatch between point-matching and line-matching strategies.
- **source_id**: foreign key to **source**. Set to the ANPR camera identifier for ANPR geometries, to 'waze' or 'tomtom' for the corresponding provider geometries, and to the Brussels `camera_id` for Brussels device positions.
- **resource_id**: foreign key to **resource**. Populated only for ANPR measure geometries (the LineString corridors between two cameras); null for all other sources.
- **fid**: an integer foreign key to the reference road segment table, populated by the geometry linking algorithm after initial loading.

The table is indexed with a GIST spatial index on **geom**, a B-tree index on **source_id**, a B-tree index on **fid**, and a composite index on (**geom_type**, **fid**) to support the linking algorithm's type-filtered scans.

5.6.2 CRS Inference and Storage Convention

Source geometries arrive in different coordinate reference systems: Waze and TomTom provide WGS 84 (EPSG:4326) coordinates, while ANPR data may use the Belgian Lambert 72 projected CRS (EPSG:31370). Rather than requiring explicit CRS metadata from each source, the geometry loading module implements a heuristic: if any coordinate falls outside the valid WGS 84 range ($-180 \leq \lambda \leq 180$, $-90 \leq \varphi \leq 90$), the geometry is assumed to be in EPSG:31370; otherwise, it is treated as EPSG:4326 and transformed to 31370 using `ST_Transform` at insertion time. All geometries are stored in EPSG:31370, which is a metric projected CRS suitable for distance and area computations in the Belgian geographic context. The Python implementation of `infer_srid()` applies this heuristic by checking coordinate ranges before issuing the appropriate insertion query.

The SQL insertion adapts based on the inferred SRID:

Listing 5.1: Conditional CRS transformation at INSERT time.

```
-- For WGS 84 input:
INSERT INTO global_geometry (geometry_id, geom, geom_type)
VALUES (%s, ST_Transform(ST_GeomFromText(%s, 4326), 31370), %s)
ON CONFLICT (geometry_id) DO NOTHING;

-- For Lambert 72 input:
INSERT INTO global_geometry (geometry_id, geom, geom_type)
VALUES (%s, ST_GeomFromText(%s, 31370), %s)
ON CONFLICT (geometry_id) DO NOTHING;
```

5.6.3 WKT Construction

Geometry objects are converted from their JSON coordinate representation to Well-Known Text (WKT) format for insertion via `ST_GeomFromText`. The `build_wkt()` function handles three geometry types:

- **Point**: `POINT(x y)` from a two-element coordinate array.
- **LineString**: `LINESTRING(x1 y1, x2 y2, ...)` from a list of coordinate pairs.
- **MultiLineString**: `MULTILINESTRING((x1 y1, ...), (x2 y2, ...))` from a nested list of coordinate lists.

Unsupported geometry types raise a `ValueError`, ensuring that unexpected spatial data is flagged rather than silently dropped.

5.6.4 The `geolink_lignes` Table

The `geolink_lignes` table contains the reference road network against which all source geometries are linked. Each row represents a road segment with an integer `fid` primary key and a `GEOMETRY(LineString, 31370)` column storing the segment geometry in Belgian Lambert 72. A GIST spatial index on the geometry column supports the proximity and intersection queries required by the linking algorithm.

In production, this table would be populated from an authoritative road network dataset (e.g., OpenStreetMap or the regional Urbis network). The implementation includes a bootstrap fallback: if `geolink_lignes` does not exist or is empty, it is automatically created and populated from existing `LineString` geometries in `global_geometry`. This bootstrap mechanism ensures that the linking algorithm can operate even before an external network dataset is loaded, using the geometries already present in the system as a provisional reference.

5.7 Geometry Linking Algorithm

The geometry linking step assigns each geometry in `global_geometry` to its nearest reference road segment by writing the segment's `fid` into `global_geometry.fid`. This linkage creates the shared spatial axis that enables cross-source comparison at the road segment level.

5.7.1 Point Matching

For geometries with `geom_type = 'point'`, the algorithm applies a distance-based two-stage criterion:

Listing 5.2: Point-to-segment matching query.

```
UPDATE global_geometry gg
SET fid = (
    SELECT gl.fid
    FROM geolink_lignes gl
    WHERE ST_DWithin(gl.geom, gg.geom, 15) -- candidate filter
    ORDER BY ST_Distance(gl.geom, gg.geom) -- nearest selection
    LIMIT 1
)
WHERE lower(gg.geom_type) = 'point'
AND gg.fid IS NULL;
```

The query uses a 15-metre candidate radius with nearest-distance selection, processing only unlinked geometries (`fid IS NULL`) to ensure incremental operation. The design rationale, including the radius choice and ambiguity handling, is discussed in the system architecture chapter.

5.7.2 Line Matching

For geometries with `geom_type` in `{'linestring', 'multilinestring'}`, the algorithm uses geometric intersection rather than proximity:

Listing 5.3: Line-to-segment matching query.

```
UPDATE global_geometry gg
SET fid = (
    SELECT gl.fid
    FROM geolink_lignes gl
    WHERE ST_Intersects(gl.geom, gg.geom)
    ORDER BY ST_Length(ST_Intersection(gl.geom, gg.geom)) DESC
    LIMIT 1
)
WHERE lower(gg.geom_type) IN ('linestring', 'multilinestring')
AND gg.fid IS NULL;
```

The query selects the candidate segment with the longest geometric overlap, appropriate for linear traffic features that follow road alignments. The design rationale, including tie resolution and ambiguity handling, is discussed in the system architecture chapter.

5.7.3 Ambiguity and Tie Resolution

When multiple candidate segments are equidistant (for points) or have equal overlap length (for lines), the deterministic SQL ordering resolves ties by database row order. This behavior is reproducible across runs on the same dataset but is not semantically optimal in dense intersection zones. The current system intentionally favors deterministic execution and simplicity over confidence-scored matching, acknowledging this as a known limitation documented in the project backlog.

5.7.4 Bootstrap Behavior

The `_ensure_geolink_lignes()` function verifies that the reference network table exists and is populated before linking proceeds. If the table is absent, it is created with the appropriate schema and GIST index. If the table exists but is empty, the function bootstraps it by copying all `LineString` geometries from `global_geometry` into `geolink_lignes` with auto-generated integer `fid` values. This fallback ensures algorithm continuity during development but should be replaced by an authoritative curated network for production use.

5.7.5 Coverage Reporting

After linking, the algorithm queries `global_geometry` to report the number of geometries with and without a `fid` assignment, providing an immediate measure of linking coverage. Low coverage may indicate that the reference network is incomplete or that the search radius is too restrictive for the data's spatial distribution.

5.8 Indexing Strategy

The indexing strategy is designed to support four primary access patterns: spatial queries (map tile rendering, proximity searches, geometry linking), temporal queries (time-series rollups and latest-observation retrieval), relational joins (foreign key lookups during loading and cross-source analysis), and constraint enforcement (uniqueness guarantees on operational state). The strategy deliberately selects different index types—GiST, BRIN, B-tree, and partial unique indexes—based on the access characteristics of each column.

5.8.1 Spatial Indexes (GiST)

Every geometry column in the schema carries a GiST (Generalised Search Tree) index. GiST is the standard PostGIS index structure for spatial data: it organises geometries into a bounding-box hierarchy that supports efficient evaluation of predicates such as `ST_DWithin`, `ST_Intersects`, and `ST_Contains` without requiring sequential scans.

The platform maintains five GiST indexes:

- `global_geometry.geom` (EPSG:31370): used by the geometry linking algorithm for point proximity matching (`ST_DWithin`) and line intersection matching (`ST_Intersects`), and by all cross-source spatial joins that operate in the metric Belgian Lambert projection.
- `global_geometry.geom_3857` (EPSG:3857): a pre-projected Web Mercator copy of each geometry, used directly by the tile generation pipeline to avoid per-request `ST_Transform` calls during MVT rendering. This denormalisation trades storage for query latency: the tile endpoint can apply `ST_TileEnvelope` and `ST_AsMVTGeom` against a geometry column that is already in the target projection.
- `geolink_lignes.geom`: supports the candidate filtering step in both point matching and line matching during segment linking.
- `signalgroup_geometry.geom` (VISU schema): supports proximity queries when linking signal groups to road segments via the traffic light segment mapping step.

5.8.2 Temporal Indexes (BRIN)

For the three largest append-only fact tables—`tomtom_flow`, `brussels_traffic_event`, and `tomtom_incident`—the platform uses BRIN (Block Range Index) indexes on the `fetches_at` timestamp column rather than conventional B-tree indexes:

Listing 5.4: BRIN index on an append-only temporal column.

```
CREATE INDEX IF NOT EXISTS idx_tomtom_flow_fetches_at_brin
ON tomtom_flow USING BRIN (fetches_at)
WITH (pages_per_range = 32);
```

BRIN indexes exploit the physical correlation between insertion order and timestamp value that naturally arises in append-only tables. Instead of indexing every row individually (as B-tree does), BRIN stores summary information—minimum and maximum values—for each contiguous block of `pages_per_range` heap pages. This yields indexes that are orders of magnitude smaller than their B-tree equivalents, reducing I/O pressure on time-range scans while remaining effective for the coarse temporal filtering patterns used by the tile generation and statistics APIs (e.g., “all TomTom flow observations in the last 2 hours”).

The `pages_per_range = 32` parameter controls the granularity of the block summaries. A smaller value produces finer-grained summaries and more precise filtering at the cost of a larger index; 32 was chosen as a pragmatic balance given the expected table growth rate and the typical query selectivity.

BRIN is not used for tables with frequent updates or non-sequential inserts (e.g., `count`, `event`), where the physical-to-logical correlation would not hold and B-tree remains the appropriate choice.

5.8.3 Conventional B-Tree Indexes

B-tree indexes are used for three categories of columns: foreign key join columns, composite query-support columns, and single-column temporal filters where BRIN is not applicable.

Table 5.1: B-tree index inventory by table and purpose.

Table	Columns	Purpose
<code>count</code>	<code>(source_id, start_ts)</code>	Time-series rollups by camera
<code>count</code>	<code>(source_id, end_ts)</code>	Interval boundary queries
<code>global_geometry</code>	<code>source_id</code>	FK lookups during loading
<code>global_geometry</code>	<code>fid</code>	Segment-level joins
<code>tomtom_flow</code>	<code>geometry_id</code>	Geometry-based joins
<code>waze_jam</code>	<code>geometry_id</code>	Geometry-based joins
<code>waze_alert</code>	<code>geometry_id</code>	Geometry-based joins
<code>waze_alert</code>	<code>report_ts</code>	Temporal filtering
<code>tomtom_incident</code>	<code>geometry_id</code>	Geometry-based joins
<code>brussels_traffic_event</code>	<code>camera_id</code>	Spatial join to <code>global_geometry</code>
<code>magneto_lane_stat</code>	<code>(tss_id, interval_start)</code>	Time-series by sensor site
<code>magneto_lane_stat</code>	<code>fetches_at</code>	Ingestion recency queries
<code>magneto_parse_audit</code>	<code>(status, created_at)</code>	Quality monitoring dashboards
<code>magneto_parse_audit</code>	<code>(tss_id, created_at)</code>	Per-site audit trail
<code>traffic_light_segment_link</code>	<code>fid</code>	Segment-to-signal joins
<code>traffic_light_segment_link</code>	<code>geometry_id</code>	Geometry-to-signal joins

The composite indexes on `count(source_id, start_ts)` and `magneto_lane_stat(tss_id, interval_start)` follow the standard pattern for time-series tables: the leading column filters by entity, and the trailing timestamp column supports ordered scans and range predicates within that entity.

5.8.4 Expression Index

The `waze_jam` table presents a modelling challenge: the upstream Waze feed does not always populate both `start_ts` and `end_ts`. Temporal queries must therefore use `COALESCE(end_ts, start_ts)` to obtain a single effective timestamp. Rather than computing this expression at query time and preventing index use, the platform defines an expression index:

Listing 5.5: Expression index on a `COALESCE` pattern.

```
CREATE INDEX IF NOT EXISTS idx_waze_jam_effective_ts
ON waze_jam ((COALESCE(end_ts, start_ts)));
```


PostgreSQL can use this index when the query’s **WHERE** or **ORDER BY** clause contains the identical **COALESCE(end_ts, start_ts)** expression, enabling efficient time-range filtering on Waze jam records without requiring the application to know which timestamp column is populated.

5.8.5 Partial Unique Index

The VISU mode interval table (**tss_mode_interval**) requires the invariant that at most one interval per TSS is in the ‘open’ state at any time. This is enforced via a partial unique index:

Listing 5.6: Partial unique index enforcing a single open interval per TSS.

```
CREATE UNIQUE INDEX IF NOT EXISTS
    idx_tss_mode_interval_one_open
    ON tss_mode_interval(tss_id)
    WHERE status = 'open';
```

A partial unique index is more expressive than a standard unique constraint because it applies uniqueness only to the subset of rows matching the **WHERE** predicate. This allows multiple closed intervals per TSS while guaranteeing that the mode inference algorithm never accidentally opens a second concurrent interval. The database enforces this invariant at the storage level, independent of application logic.

5.8.6 VISU Schema Indexes

The VISU schema defines its own set of indexes optimised for the traffic light analysis workload. Table 5.2 lists the key indexes; all are parameterised by the schema name to support parallel deployments.

Table 5.2: VISU schema index inventory.

Table	Columns	Purpose
signalgroup_event	(tss_id, event_ts DESC)	Time-series queries per TSS
signalgroup_event	event_ts	Global temporal scans
signalgroup_event	(value_int, event_ts)	Phase-specific lookups
signalgroup_latest	current_value_int	Current-state filtering
poll_audit	run_id	Per-run diagnostics
tss_mode	(tss_id, mode_seq DESC)	Latest mode retrieval
tss_mode_cycle	(tss_id, signalgroup_num)	Per-group cycle lookups
tss_mode_interval	(tss_id, valid_from DESC)	Active interval queries
tss_mode_interval	(tss_id, status, valid_to)	Open-interval filtering
tss_mode_inference_audit	(run_id, tss_id, created_at DESC)	Audit trail retrieval
signalgroup_geometry_map	geometry_id	Geometry resolution

5.8.7 Design Rationale Summary

The choice of index type is driven by the data’s physical characteristics rather than a uniform default:

- **GiST** for geometry columns, because spatial predicates require bounding-box decomposition that B-tree cannot provide.
- **BRIN** for `fetched_at` on large append-only tables, because the insertion-order correlation with timestamp makes BRIN compact and effective, while a B-tree on millions of rows would consume significantly more storage for marginal query-time improvement on typical time-range filters.
- **B-tree** for foreign keys, composite query-support columns, and temporal columns on smaller or update-heavy tables where physical-order correlation is not guaranteed.
- **Expression index** for the Waze **COALESCE** pattern, because the alternative—forcing the application to choose between two nullable columns—would leak data-quality concerns into every consumer query.
- **Partial unique index** for the open-interval invariant, because it encodes a domain constraint directly in the storage layer where it cannot be bypassed by application bugs.

5.9 Chapter Summary

This chapter established the storage layer that makes cross-source mobility analysis possible. The PostgreSQL/PostGIS schema keeps each provider’s observations in source-specific fact tables, but routes their geometries through `global_geometry` and the `fid` segment link. That design preserves the semantics of ANPR counts, Waze reports, TomTom flow and incidents, Brussels camera events, magnetometer lane statistics, and VISU signal data while giving them a common spatial axis.

The practical outcome is that formerly siloed observations can be queried at road-segment level: ANPR counts can be compared with Waze congestion, TomTom incidents can be correlated with Brussels speed drops, magnetometer measurements can be read alongside traffic signal phases, and coverage maps can distinguish multi-source corridors from single-source blind spots. The indexes and linking rules described in the chapter are therefore not implementation detail alone; they are the mechanism that turns integrated storage into usable spatial evidence.

Chapter 6

Brussels Fast Map: Visualization and Results

The preceding chapters described how heterogeneous traffic data is ingested, transformed, and persisted into a unified relational-spatial database. However, the operational value of this unified model remains abstract until analysts and operators can *see* the integrated data, explore it interactively, and derive actionable insights from the spatial and temporal patterns that emerge when multiple sources are rendered simultaneously on a single map.

The Brussels Fast Map server was developed to bridge this gap. It is a lightweight, self-contained HTTP server that generates Mapbox Vector Tiles (MVT) directly from the PostgreSQL/PostGIS database and serves an interactive web dashboard built on MapLibre GL JS. The server enables real-time multi-layer visualization of all integrated data sources, time-range filtering, segment-level statistical inspection, and traffic light capacity analysis—capabilities that were impossible when each source existed in isolation.

This chapter describes the server architecture (Section 6.1), the vector tile generation pipeline (Section 6.2), the interactive dashboard features (Section 6.3), the API endpoints for detailed analysis (Section 6.4), and the results achieved through this visualization approach (Section 6.5).

6.1 Server Architecture

The Brussels Fast Map server is implemented as a single-file Python HTTP server using the standard library’s `ThreadingHTTPServer`. This design choice prioritizes deployment simplicity and eliminates external web framework dependencies while still supporting concurrent tile requests through threading.

The server connects to the same PostgreSQL/PostGIS database used by the ETL pipeline, using a `psycopg` connection pool (`psycopg_pool.ConnectionPool`) to manage database connections efficiently. Connection pooling is essential because each tile request triggers a database query, and concurrent map interactions can generate dozens of tile requests per second during panning and zooming.

The server exposes four categories of endpoints:

1. **Dashboard endpoint** (`/`): serves the interactive HTML/JavaScript map application.
2. **Tile endpoints**: generate and serve Mapbox Vector Tiles for each data layer.

3. **Statistics API**: returns segment-level aggregated statistics and geometry details for a selected `fid`.
4. **Traffic light API**: returns signal capacity curves and current cycle state for a selected signal group.

6.2 Vector Tile Generation Pipeline

6.2.1 Tile Request Processing

When the client requests a tile at coordinates (z, x, y) for a specific layer, the server executes a layer-specific SQL query that performs the full pipeline from raw data to MVT output within a single database round-trip. Each query follows a common pattern:

1. **Bounds computation**: `ST_TileEnvelope(z, x, y)` computes the tile’s bounding box in Web Mercator (EPSG:3857), which is then transformed to Belgian Lambert 72 (EPSG:31370) for spatial filtering against the stored geometries.
2. **Data aggregation**: source-specific aggregation is applied within the time window (configurable lookback period, default: 1 hour). For example, TomTom flow observations are averaged per geometry, Brussels counts are averaged per camera, and ANPR counts are summed per source.
3. **Geometry transformation and simplification**: stored geometries (in EPSG:31370) are transformed to Web Mercator and simplified using the Visvalingam–Whyatt algorithm (`ST_SimplifyVW`) to reduce tile payload size at lower zoom levels.
4. **MVT encoding**: `ST_AsMVTGeom` clips and transforms geometries into tile-local coordinates, and `ST_AsMVT` assembles the final Protocol Buffer tile payload.

6.2.2 Supported Layers

The server supports six data layers, each backed by a dedicated SQL query:

Table 6.1: Brussels Fast Map vector tile layers.

Layer	Source Table(s)	Aggregation
<code>tomtom_flow</code>	<code>tomtom_flow</code> , <code>global_geometry</code>	AVG traffic_level per geometry
<code>tomtom_incidents</code>	<code>tomtom_incident</code> , <code>global_geometry</code>	Raw incidents with metadata
<code>waze_jams</code>	<code>waze_jam</code> , <code>global_geometry</code>	Raw jams with severity/speed
<code>waze_alerts</code>	<code>waze_alert</code> , <code>global_geometry</code>	Raw alerts with type/subtype
<code>brussels_counts</code>	<code>brussels_traffic_event</code> , <code>global_geometry</code>	AVG count/speed/occupancy per camera
<code>anpr_counts</code>	<code>count</code> , <code>global_geometry</code>	SUM total/sample count per source

Each layer query includes a time-range filter that accepts either absolute timestamps or a relative lookback period in hours, enabling both real-time monitoring (“show the last hour”) and historical investigation (“show data from 8:00 to 9:00 yesterday”).

6.2.3 Spatial Indexing and Query Performance

The tile generation pipeline depends critically on the spatial indexes provisioned in the database schema (Chapter 5). The GIST index on `global_geometry.geom` enables the bounding-box filter (`geom && bounds`) to use index-accelerated intersection tests rather than sequential scans. Combined with B-tree indexes on temporal columns (`fetch_at`, `report_ts`), each tile query typically completes in tens of milliseconds for moderate data volumes, enabling smooth interactive panning and zooming.

The Visvalingam–Whyatt simplification (`ST_SimplifyVW` with a tolerance of 6 pixels at zoom level 10) reduces the number of coordinate points in each geometry, decreasing tile payload size and improving client-side rendering performance. This simplification is applied per-tile during generation, preserving the full-resolution geometries in the database for accurate spatial analysis.

6.3 Interactive Dashboard

The server’s root endpoint serves a self-contained HTML5 application that combines MapLibre GL JS for map rendering with a custom glassmorphism-styled user interface. The dashboard provides several analytical capabilities:

6.3.1 Multi-Layer Visualization

The dashboard renders all six data layers simultaneously on a single map, with independent visibility toggles and color-coded styling. Each layer uses data-driven styling: TomTom flow segments are colored on a green-to-red gradient based on congestion score (computed as $1 - \text{traffic_level}$, with road closures highlighted); Waze jams are colored by severity level (0–5 scale); Brussels camera counts are sized proportionally to average vehicle count; and ANPR observations are rendered with count-proportional markers.

This multi-layer overlay is the most direct demonstration of the platform’s integration value: for the first time, operators can visually correlate TomTom’s commercial traffic intelligence with Waze’s crowdsourced reports and Bruxelles Mobilite’s own camera infrastructure on a single interactive map.

6.3.2 Time Range Selection

The dashboard provides a time range picker that supports both relative lookback periods (e.g., “last 1 hour”, “last 4 hours”) and absolute time ranges with date-time inputs. When the time range changes, all layers are automatically refreshed by adding temporal query parameters to tile requests. This enables rapid temporal exploration: operators can “rewind” the map to examine conditions at a specific time, compare morning and evening peak patterns, or track the evolution of an incident over hours.

6.3.3 Segment-Level Inspection

Clicking on any feature in the map triggers an API call to retrieve detailed segment-level statistics. The statistics panel displays aggregated metrics for the selected road segment (identified by its `fid`), including traffic flow averages, incident counts, camera measurements, and temporal distributions. This drill-down capability enables operators to move from a city-wide overview to detailed segment-level analysis without leaving the dashboard.

6.3.4 Traffic Light Analysis

The dashboard includes a dedicated traffic light analysis panel that provides two complementary views:

Capacity curves show the relationship between green time utilisation and traffic flow for a selected signal group, enabling assessment of whether an intersection is operating near capacity. The capacity curve data is retrieved from the `/api/traffic-light/{id}/{num}/capacity` endpoint, which computes flow-versus-green-time distributions from the VISU pipeline data.

Latest cycle state displays the current or most recent signal cycle for a selected signal group, showing phase durations (green, amber, red) and transition times. This information is retrieved from the `/api/traffic-light/{id}/{num}/latest-cycle` endpoint and enables real-time monitoring of signal controller behavior.

6.4 API Endpoints for Detailed Analysis

Beyond tile serving, the Brussels Fast Map server exposes REST API endpoints that support programmatic access to the unified data model:

6.4.1 Segment Statistics (`/api/fid/{fid}/stats`)

Returns aggregated traffic statistics for a specific road segment, including average traffic levels from TomTom flow, vehicle counts from ANPR cameras, speed and occupancy from Brussels counters, and Waze jam severity—all filtered by the current time range. This endpoint demonstrates the cross-source analytical value of the `fid`-based segment linking: a single API call retrieves correlated data from all providers monitoring the same road segment.

6.4.2 Segment Geometries (`/api/fid/{fid}/geometries`)

Returns all source geometries linked to a specific road segment, including their original geometry types, source identifiers, and geometry hashes. This endpoint supports spatial debugging and coverage analysis, enabling users to understand which sources contribute observations for a given segment.

6.4.3 Traffic Light Capacity (`/api/traffic-light/{id}/{num}/capacity`)

Returns capacity utilisation data for a specific traffic signal group, computed from the VISU pipeline. The response includes green time distributions, flow measurements during each phase, and derived capacity metrics that support intersection performance assessment.

6.4.4 Traffic Light Latest Cycle (`/api/traffic-light/{id}/{num}/latest-cycle`)

Returns the most recent complete signal cycle for a specific signal group, including phase transition timestamps, phase durations, and cycle length. This endpoint enables real-time monitoring of signal controller behavior and supports detection of anomalous cycle patterns.

6.4.5 Waze Jam Segment Matching

The server implements a refined spatial matching strategy for Waze jams that goes beyond the basic geometry linking described in Chapter 5. For each Waze jam geometry, the server computes

overlap with candidate road segments using a buffer-based intersection approach: segments within a configurable distance threshold (default: 20 meters) are identified, and the segment with the highest overlap ratio (minimum: 25% of the jam’s length) is selected. This stricter matching reduces false positives in dense urban areas where multiple road segments run in close proximity.

6.5 Results and Operational Value

The Brussels Fast Map server demonstrates several concrete outcomes that validate the platform’s integration architecture.

6.5.1 Cross-Source Spatial Correlation

The multi-layer visualization enables direct visual correlation of observations from different providers on the same road segments. For example, during a morning peak period, an operator can observe: TomTom flow segments turning red (high congestion) on a specific corridor, Waze jam polylines appearing on the same segments with increasing severity levels, Brussels camera counts spiking on the approaches to the congested area, and ANPR measurements showing elevated vehicle counts on upstream feeder routes. This simultaneous multi-source view was the original motivation for the platform and was impossible before the integration architecture was implemented.

6.5.2 Temporal Pattern Analysis

The time-range filtering capability enables systematic comparison of traffic patterns across different periods. By switching between morning peak, afternoon peak, and off-peak time windows, operators can identify segments where congestion patterns differ significantly between providers—potentially indicating calibration issues, coverage gaps, or genuine differences in what each source measures.

6.5.3 Traffic Signal Performance Assessment

The integration of VISU traffic light data with flow and count observations enables a new category of analysis: correlating signal timing with observed traffic conditions. The capacity curve endpoint provides evidence of whether an intersection’s signal timing is well-matched to demand, while the latest-cycle endpoint enables real-time monitoring of signal controller behavior. When combined with the flow and count layers, this creates a feedback loop where signal performance can be assessed against actual traffic conditions measured by independent sources.

6.5.4 Coverage and Data Quality Assessment

The multi-layer map provides an immediate visual indicator of monitoring coverage: segments where multiple layers overlap represent well-monitored corridors, while segments with sparse coverage may indicate infrastructure gaps. The segment-level statistics API quantifies this coverage by reporting which sources contribute data for each `fid`, enabling systematic identification of monitoring blind spots.

6.5.5 Performance Characteristics

The server achieves interactive-grade performance for the Brussels road network: typical tile generation times are in the tens-of-milliseconds range for moderate zoom levels, supported by the

spatial indexing infrastructure described in Chapter 5. The connection pool prevents connection-establishment overhead from becoming a bottleneck under concurrent requests. The Visvalingam–Whyatt simplification keeps tile payloads compact (typically under 100 KB per tile at city-wide zoom levels), enabling smooth client-side rendering even on standard office hardware.

6.5.6 Limitations and Future Directions

Several limitations of the current Brussels Fast Map implementation should be noted: the server uses HTTP rather than HTTPS, authentication is not implemented, and the single-file architecture limits extensibility. The traffic light capacity analysis depends on the VISU pipeline data, which is currently available only for a subset of monitored intersections. Future work could extend the server with WebSocket-based live updates, user authentication, persistent dashboard configurations, and integration with Bruxelles Mobilite’s operational alerting infrastructure.

6.6 Chapter Summary

This chapter demonstrates the serving value of the platform. The Fast Map is not a separate visualization prototype: it is the visible surface of the same ingestion, ETL, schema, and segment-linking decisions described earlier. By rendering TomTom, Waze, Brussels, ANPR, VISU, and reference-segment layers through one time-aware interface, it makes the unified database inspectable by operators and analysts who do not need to write SQL for every corridor question. The next chapter uses that interface, together with the Control Center, as evidence for the research-question answers.

Chapter 7

Evaluation and Results

Chapter 1–Chapter 6 describe the machinery: the ingestors, the staged contracts, the geometry registry, the segment-linking scheme, and the Fast Map server. This chapter shows what Bruxelles Mobilite gets from that machinery in day-to-day use.

Two tools carry the argument. The Mobility Pipeline Control Center exposes the ingestion and loading pipeline as a live, inspectable system that an operator can trust to run without a developer sitting in front of it. The Brussels Fast Map turns the unified database into an interactive surface where an analyst can pose cross-source questions that previously required custom scripts and several hours of manual reconciliation.

Both tools are visible evidence of the integration work. The Control Center shows that the pipeline behind the data is observable and recoverable. The Fast Map shows that the data itself, once integrated, answers questions that the six source feeds could not answer on their own. The rest of the chapter walks through each tool with its actual screenshots, drills into two real corridors where cross-source correlation becomes visible, and closes with a compact mapping back to the three research questions.

Table 7.1: What each section of this chapter delivers.

Section	Primary RQ	What it shows
Pipeline Control Center (Section 7.1)	RQ2	The ETL pipeline runs as a live, monitored system. An operator can diagnose it without reading logs.
Fast Map at a glance (Section 7.2)	RQ1	The Fast Map renders the shared segment model and layers every source on one map.
Rue Scailquin: ANPR vs TomTom (Section 7.3)	RQ1 / RQ3	Two independent sources agree on the same afternoon event when pinned to a single segment.
Avenue Charles-Quint: magnetometer, signal, TomTom (Section 7.4)	RQ3	Signal capacity, observed demand, and commercial flow align on one intersection.
Value Delivered (Section 7.5)	RQ3	Concrete change in capability before and after the platform.
Known Limits (Section 7.6)	RQ3	The boundary of the current delivery, stated without softening.

7.1 The Mobility Pipeline Control Center

7.1.1 What the Operator Sees

The Control Center is a single browser page that an on-call engineer opens to answer one question: is the platform healthy right now? The page consolidates the file-based ETL described in Chapter 4 into four blocks, visible together in Figure 7.1.

The top strip holds the configuration controls: the path to the active `general_config.json`, an *Apply config path* button, a *Reload config* button, and six start/stop toggles for the three daemons (ingestion, ETL, and the traffic-light analytics worker labelled TLC). The status badges next to the header confirm at a glance which daemons hold the single-instance lock. On the capture, *ingestion: running*, *ETL: running*, and *TLC: running* all appear in green, and the configuration shows that seven sources are loaded from `general_config.json`.

The *Ingestion Health* panel on the left lists each of the six source families with a dedicated card. Each card carries a green *Healthy* badge when the last payload landed within the expected interval, and shows the freshness (time since last payload), the last update timestamp, the rolling file and event counters, and the error counter. On the capture, ANPR, Brussels, Waze, TomTom, Magneto, and TLC are all green, with freshness values ranging from 37 seconds for Magneto to 875 seconds for ANPR, all within the configured polling cadence for each source.

The *ETL Progress* panel on the right tracks the transformation and loading worker. The current stage banner at the top reads *Stage: Load TomTom 1/1*, and three progress bars underneath report global progress, transform progress, and load progress for the active cycle. Below the bars, a fixed block reports the last cycle end time, the last cycle duration (27:12), the average cycle duration (47:14), the current backlog (stable, 0 files, 0 B), and the session error counter. A correct reading of this panel is that the ETL finishes its cycles, does not accumulate backlog, and has not escalated an error during the session.



Figure 7.1: The Mobility Pipeline Control Center on the deployment server. The page consolidates configuration, start/stop controls, per-source ingestion health, ETL progress, and live logs into one browser view.

The *Artifact Volumes* row under the ETL panel tracks the file-lifecycle counters: raw, staged, processing, processed, and loaded. On the capture, loaded sits at 2000 files and 1.3 MB, processing and staged are at zero, and the raw counter holds six files at 1.3 MB. That distribution is what the commit-before-move semantics predict at the moment of capture: raw holds the short head of inbound payloads, processing and staged clear in near real time, and files reach the `loaded/` handoff only after their rows have been committed. The counter is an operational artifact-state snapshot, not a durable filesystem archive; successful cycles may subsequently remove transient files during cleanup.

The *Live Logs* panel at the bottom shows the most recent event lines across three tabs (ingestion, ETL, TLC). The capture shows a stream of `[tomtom] stored payload bytes=...`, `[brussels] stored payload`, `[waze] stored payload`, and one ETL line reading `cycle groups=4 stored=6 duplicates=0 error_payloads=1 payload_bytes_total=4327`. That single error payload, surfaced without crashing the cycle, is exactly the per-source isolation behaviour described in Chapter 4: a malformed payload is quarantined and counted, the cycle completes, and the next cycle continues.

7.1.2 Evidence of Live Operation

The Control Center is not a deployment marketing screenshot. It functions as a point-in-time operational proof that the pipeline was running end-to-end at the moment the capture was taken. Three properties in the figure support that claim:

1. Every source family shows a green *Healthy* badge and a freshness value inside its expected interval. That rules out a silent ingestion stall.
2. The *loaded* counter stands at 2000 files while *processing* and *staged* stand at zero at the moment of capture. The important signal is not long-term file retention, but that files reached the post-commit handoff state and the intermediate states were clear under the commit-before-move rule.
3. The live log stream shows one error payload on the most recent ETL cycle, isolated at source level, with the cycle still producing stored rows and zero duplicates. That is the error-containment contract working as specified, not hiding a crash.

The capture does not claim a formal uptime series. It establishes something narrower and more defensible: the system was a live monitored pipeline at that point, not a replayed dataset inside a notebook.

7.1.3 What the Tool Changes Operationally

Before the Control Center existed, a diagnosis started in the log directory. An operator read recent lines from six ingestor logs, checked the size of each state directory through `ls`, correlated timestamps by hand, and then formed a guess about what was stuck. The Control Center replaces that sequence with one page. An issue that used to take fifteen minutes to locate now takes seconds, and the diagnostic trail is the same for everyone on the team rather than reconstructed in private each time.

The same page also reduces the cost of a handover. A new engineer learns the shape of the pipeline from the Control Center layout before opening a source file, because the panels mirror the ETL lifecycle stages of Chapter 4 one-to-one.

7.2 The Brussels Fast Map at a Glance

7.2.1 Dashboard Layout

Figure 7.2 shows the Fast Map in its baseline state: the shared segment network from the geometry registry, rendered as black polylines on top of a light basemap, with every data layer switched off. The purpose of the screenshot is to make the controls and panels visible before any data is added, so the reader can read the dashboard structure independently of the data it carries.

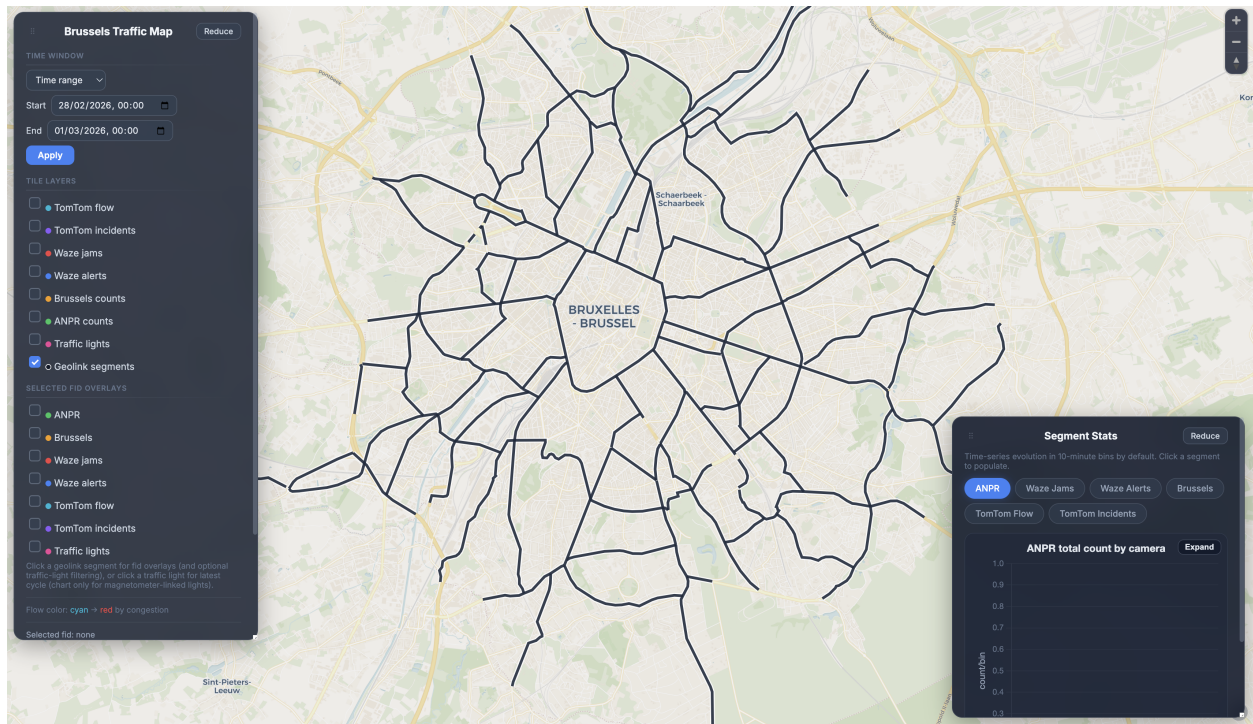


Figure 7.2: Brussels Fast Map in its baseline state, with only the *Geolink segments* layer enabled. The screenshot exposes the three functional regions of the dashboard: time-range control on the top-left, layer toggles and legend in the left sidebar, and the segment-statistics side panel on the right.

Three regions of the dashboard are visible:

- **Time range (top-left).** A *Time range* selector with explicit *Start* and *End* datetime inputs and an *Apply* button. The screenshot shows the range set from 28/02/2026 00:00 to 01/03/2026 00:00, which is the 24-hour window used in the rest of the chapter. The control accepts either absolute timestamps or a relative lookback through the server-side query parameters.
- **Layers and legend (left sidebar).** The upper block toggles each data layer: *TomTom flow*, *TomTom incidents*, *Waze jams*, *Waze alerts*, *Brussels counts*, *ANPR counts*, *Traffic lights*, and *Geolink segments* (the shared network rendered here). The lower block is the legend, colour-coded by source family. Toggling a layer fetches the corresponding data without changing the map view.

- **Segment Stats panel (right).** A tabbed panel with one tab per source (*ANPR*, *Waze Jams*, *Waze Alerts*, *Brussels*, *TomTom Flow*, *TomTom Incidents*). In Figure 7.2 the panel shows its empty state, because no segment has been clicked. Clicking any segment on the map populates every tab with the data that source contributed to the selected *fid*.

The black-segment map is itself part of the evidence. Every line rendered here comes from the shared *geolink_lignes* segment reference keyed by *fid*; source geometries in *global_geometry* are linked to this reference network for cross-source queries. The visible density across the region shows that the segment reference and geometry-linking strategy cover the Brussels road network at the granularity the later analyses depend on.

7.2.2 TomTom Flow Over the Region

Figure 7.3 switches the *TomTom flow* layer on over the same time range. Each segment is coloured by the TomTom congestion score (computed as $1 - \text{traffic_level}$), on a green-to-red gradient with road closures in grey.

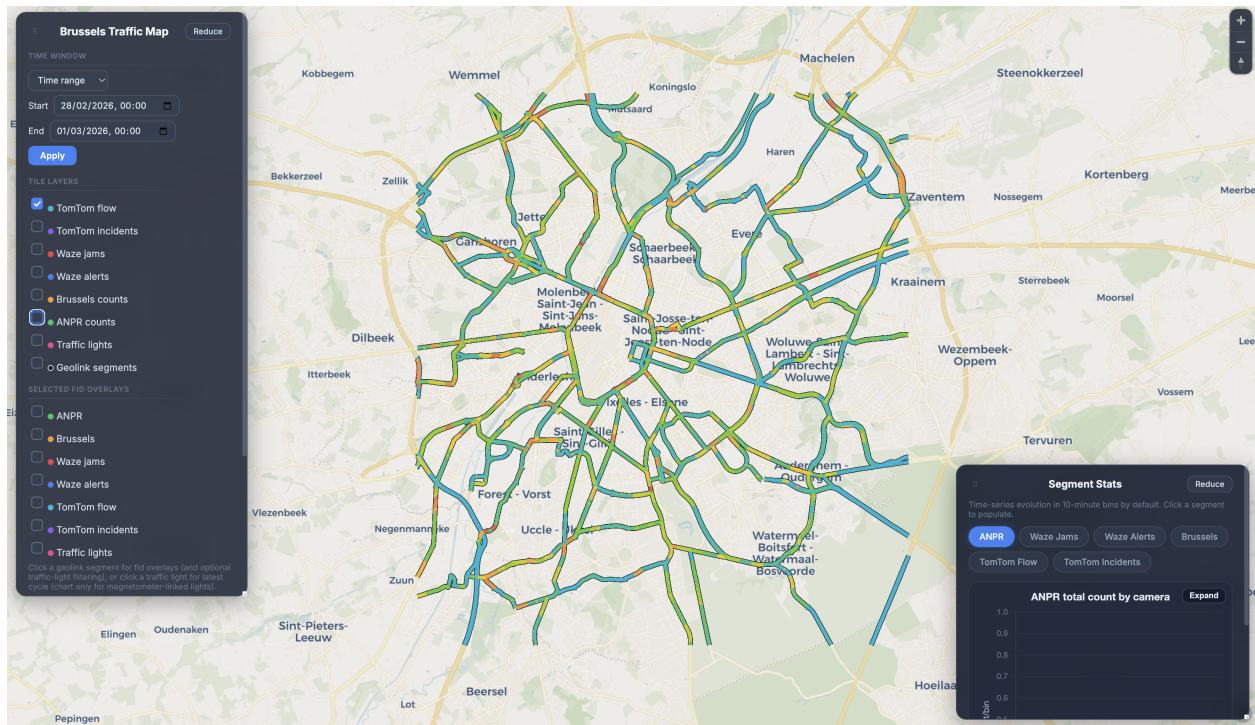


Figure 7.3: Brussels Fast Map with the *TomTom flow* layer enabled over the same 24-hour window. Green segments carry free-flowing traffic; yellow and red segments carry congestion or incidents. The figure is the direct output of the Fast Map tile pipeline against a single provider.

Read from the map, the TomTom flow view is a city-scale traffic snapshot: the inner ring, the main commuter axes in and out of the centre, and the motorway approaches into Brussels are visible as continuous coloured lines. Green dominates on lower-order streets; red and yellow concentrate on the ring and on the axes around Saint-Josse, Etterbeek, and Ixelles. The payoff of the Fast Map is not the heatmap itself (TomTom exposes its own native visualisation), but the fact that this layer sits in exactly the same segment frame as ANPR, Waze, Brussels counts, magnetometers, and traffic lights. That shared frame is the precondition for the two corridor analyses that follow.

7.3 Rue Scailquin: ANPR and TomTom Agree on the Same Event

The strongest value claim of the integrated platform is that two independent sensing systems, collected through different protocols, produce the same time signal when projected onto the same road segment. The Rue Scailquin corridor in Saint-Josse-ten-Noode is the cleanest example, because both an ANPR camera array and TomTom flow observations are linked to the same `fid` in the geometry registry.

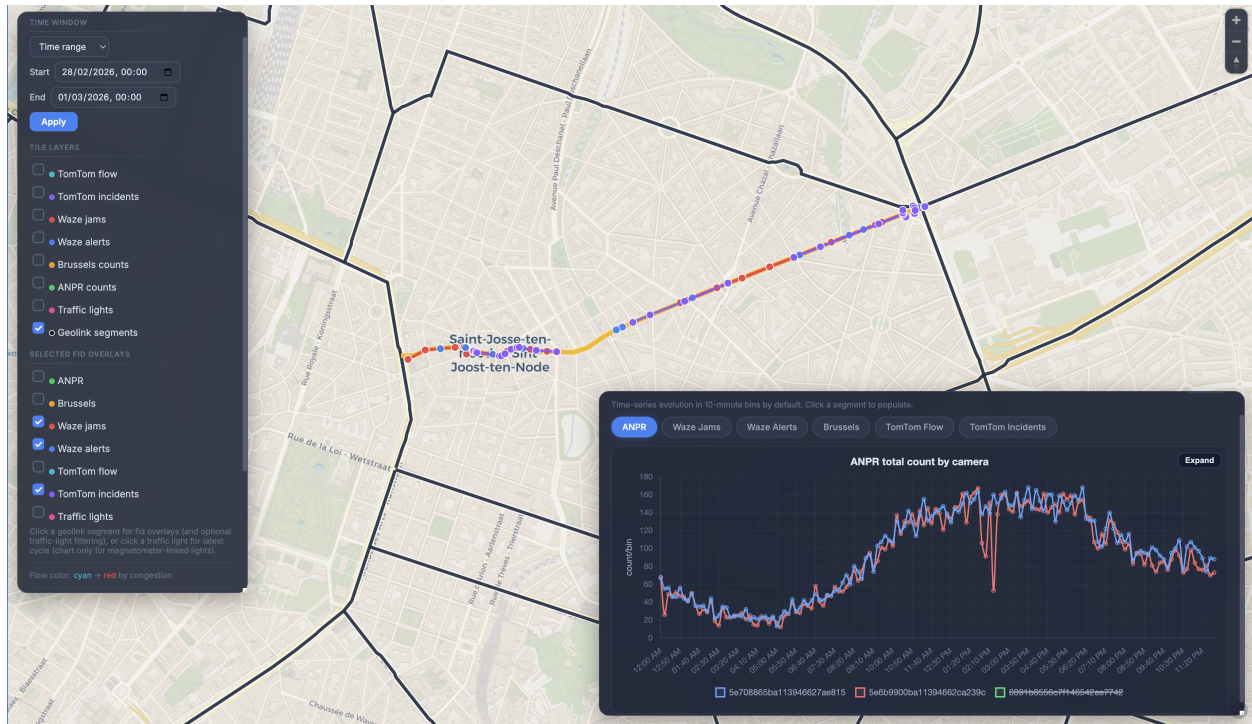


Figure 7.4: Rue Scailquin segment, *ANPR* tab of the Segment Stats panel. The rainbow-coloured points on the corridor are the ANPR cameras linked to that `fid`. The panel plots *ANPR total count by camera* across the 24-hour window, with one line per camera. The afternoon bump between roughly 14:00 and 17:00 is the event of interest.

The analyst workflow is identical in both figures: click the corridor once, switch the tab on the right panel. Everything else is provided by the platform. The map stays on the same `fid`. The time window stays constant. Each tab reads from the same unified relational-spatial schema described in Chapter 5.

The interpretation is direct. In Figure 7.4, ANPR counts climb from the late-morning baseline into an afternoon peak and then fall again in the evening. In Figure 7.5, TomTom relative speed stays near the free-flow reference across the morning, drops sharply in the afternoon, and recovers toward the end of the window. The two signals are produced by different sensing modalities: ANPR is a physical camera count, TomTom flow is a commercial traffic-level index computed on TomTom's own infrastructure. When both are projected onto the same segment through the `fid` join, they describe the same event with the same timing.

That agreement has two uses:

1. **Cross-validation at zero marginal cost.** Once the platform exists, the analyst does not run an ad-hoc spatial alignment script. The correlation is readable in two clicks on a browser

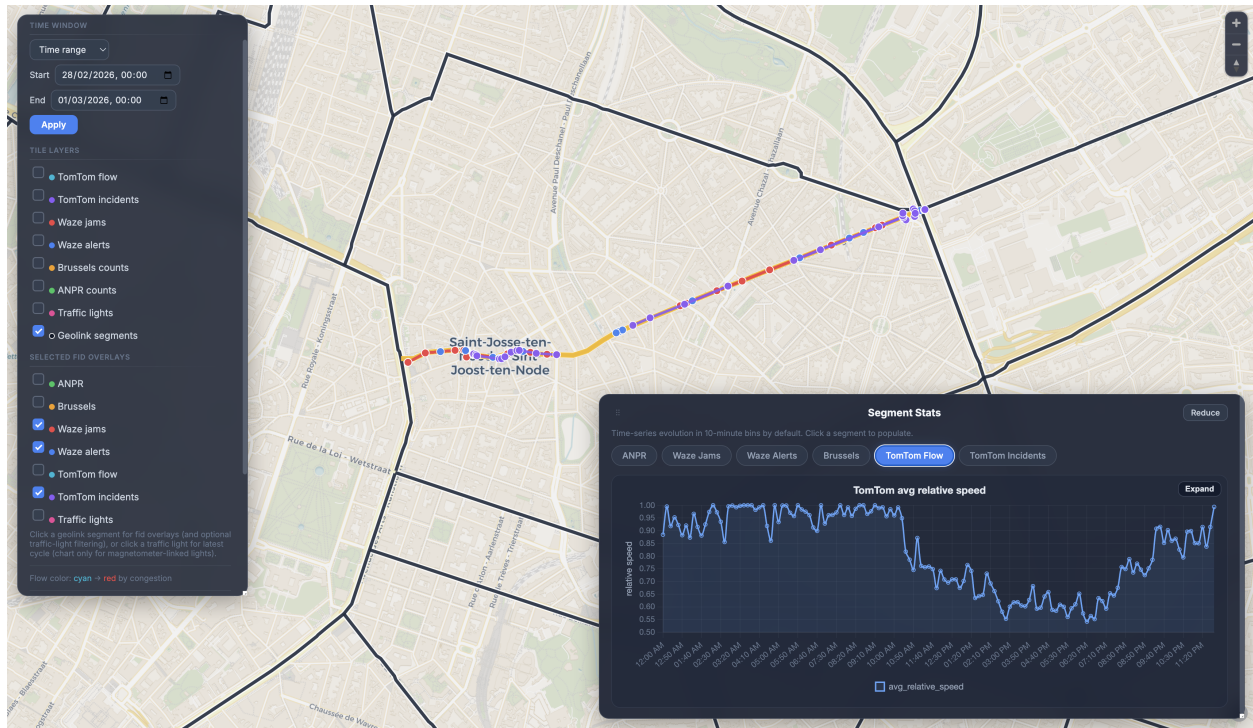


Figure 7.5: The same Rue Scailquin segment, *TomTom Flow* tab of the same Segment Stats panel. The panel plots *TomTom avg relative speed* across the same 24-hour window. The speed line dips between roughly 14:00 and 17:00, at the same window where ANPR counts climb in Figure 7.4.

tab.

2. **Confidence calibration.** When two feeds agree, the composite signal is stronger than either one alone. When they disagree, the disagreement itself becomes a diagnostic: a missing ANPR camera, a calibration issue on the TomTom side, or a genuine divergence worth investigating.

7.4 Avenue Charles-Quint: Signal Capacity Meets Observed Demand

The second corridor adds two more source families to the picture: the Sensys magnetometer counts and the VISU traffic-light controller state. Figure 7.6 shows the Fast Map centred on Avenue Charles-Quint, with the traffic-light layer, the geometry segments, and the TomTom flow panel all visible in one frame.

The panels in the figure carry three linked signals that came from three different systems:

- **Traffic-light capacity.** The *Traffic Light Stats* chart plots three series for signal group SAB01/3, direction 3, three lanes: the theoretical maximum traffic (capacity if the signal ran at 100% green, dashed line), the observed magnetometer traffic in vehicles per ten minutes (orange line), and the light capacity ratio as a percentage (green line). The panel is computed against the VISU cycles described in Section 4.5.4.
- **TomTom flow.** The *Segment Stats* panel in the lower right shows the TomTom average relative speed over the same 24-hour window on the same corridor.



Figure 7.6: Avenue Charles-Quint segment with the traffic-light layer enabled. The *Traffic Light Stats* panel on the right plots capacity for the selected signal group SAB01/3 against theoretical maximum traffic and against observed magnetometer traffic. The *Segment Stats* panel below carries the TomTom relative-speed curve for the same corridor in the same window.

Read together, the three series describe one intersection from three independent angles. The magnetometer line tracks the vehicles that physically crossed the detector loops. The light-capacity line tracks how much of the theoretical throughput the signal programme offered. The TomTom speed line tracks what commercial floating-car data saw as traffic conditions on the same corridor. They rise and fall together in the same time window, which is the same correlation result the Rue Scailquin section showed with a different pair of sources.

This is the analysis the agency could not produce before the platform existed. Signal-state data lived in a VISU-specific tool. Magnetometer counts lived in the SNAPS XML interface. TomTom flow lived in a third-party dashboard. Putting the three on one page required a unified segment identifier, a unified time axis, and a unified query endpoint. The Fast Map is the composition of those three things into a working interface.

7.5 Value Delivered

Chapter 1 argued that the pre-project state of the agency was abundant raw signals and scarce joint analysis. The Control Center and the Fast Map, together with the integrated database behind them, change that state on the capabilities that matter for the agency.

Table 7.2: Before the platform versus after the platform, on capabilities the agency cares about.

Capability	Before	After
Answering a cross-source corridor question	Manual per-source pulls, ad-hoc spatial alignment, one-off notebooks. Reproducibility near zero.	One time range, one click on a segment, one response listing every source’s contribution to the segment. Rue Scailquin (Section 7.3) is the demonstrator.
Monitoring pipeline health	Tail six log files, inspect each state directory through <code>ls</code> , correlate timestamps by hand.	One page with per-source freshness, backlog, ETL progress, and live logs. Figure 7.1 is the surface.
Recovering from a partial failure	Reconstruct which files loaded, guess what to rerun, risk double-inserts.	Lifecycle counters and the Live Logs panel show exactly what is stuck. Loading is idempotent through <code>ON CONFLICT DO NOTHING</code> .
Adding a new source	Rewrite downstream joins because identifier and geometry conventions differ.	Implement a source-specific ingestor and transformer. Geometry linkage plugs into the registry; the Fast Map gains a layer by adding a tile query.
Signal-versus-demand analysis	Two disconnected tools: a VISU signal-state viewer and a separate traffic viewer.	Capacity curves, observed magnetometer traffic, and TomTom speed on one screen (Figure 7.6).
Data-quality introspection	Each analyst defines their own notion of usable data.	The ETL and Control Center expose source freshness, error payloads, backlog, and artifact counts through one operational surface.

The shift matters for three groups. For on-call engineers, the Control Center collapses the triage loop from minutes to seconds. For analysts, the Fast Map eliminates the per-question integration effort that used to gate every cross-source study, as the two corridor sections show. For the agency as an institution, the pair makes the data asset transferable: a new team member can read both the pipeline shape and the data shape from two pages, without opening source code first.

7.6 Known Limits

Stating the limits is part of the value claim, not a retreat from it.

- **Point-in-time captures.** The Control Center and the Fast Map are presented through screenshots. The chapter argues that the platform runs live; it does not claim a long-horizon

uptime study.

- **Two corridors, not a network study.** The cross-source agreement shown on Rue Scailquin and Avenue Charles-Quint is evidence that the platform supports segment-level correlation on real data. It is not a statistical claim about the network as a whole.
- **Source-semantic asymmetry.** The Fast Map overlays sources that measure different things: crowdsourced reports, commercial flow scores, physical counts, signal state. A visual overlap is an analytical cue, not a claim of equivalence.
- **Geometry-linkage edges.** The `fid` join works at scale but does not eliminate edge cases in dense urban topologies. A segment click returns what the registry says is there; confidence scoring and manual validation remain production hardening work.
- **Deployment surface.** The Fast Map runs on HTTP with no authentication, and the Control Center has no alerting. Both are acceptable for internal use during the thesis phase and flagged as production hardening work in Chapter 8.

7.7 Research Question Answers

Table 7.3: Research question evidence in this chapter.

RQ	Claim	Evidence in this chapter
RQ1	Heterogeneous feeds integrate into one queryable spatio-temporal model.	The Fast Map renders every source on one shared network (Figures 7.2 and 7.3) and drills down on a single <code>fid</code> across all six families (Figures 7.4, 7.5, 7.6).
RQ2	The pipeline operates as a live, monitored, replay-safe system.	The Control Center (Figure 7.1) surfaces process state, source freshness, ETL progress, artifact volumes, and live logs in one page. The capture shows all six sources healthy and an isolated error payload handled without crashing the cycle.
RQ3	Integration delivers concrete operational value.	Rue Scailquin shows ANPR and TomTom agreeing on the same afternoon event on one segment. Avenue Charles-Quint shows magnetometer counts, signal capacity, and TomTom speed aligned on one intersection. The before/after table names the capability changes.

The chapter closes on the same claim it opened with, now backed by six screenshots and two corridor analyses: the integrated platform has moved Bruxelles Mobilité from six isolated feeds to one monitored pipeline and one interactive surface where cross-source mobility questions become one interaction each. The evidence is deliberately practical rather than theoretical. It shows that the architecture is observable while it runs, that independent data sources can be inspected on the same segment and time window, and that the resulting interface changes the amount of work required to answer an operational corridor question.

Chapter 8

Conclusion and Future Work

This final chapter consolidates the answer to the research question rather than introducing a new technical subsystem. It restates what the implemented platform delivered, connects the evidence from the evaluation chapter to the three research questions, and separates the thesis contribution from the production work that remains.

8.1 What the Thesis Delivered

The thesis set out to answer a single engineering question for Bruxelles Mobilité: can six operational data feeds, collected under different protocols, schemas, and temporal conventions, be integrated into one coherent relational-spatial database that preserves source semantics and stays maintainable by a small team? The answer is yes. Chapter 3–Chapter 7 document the architecture, the implementation, the data model, the visualisation layer, and the evidence that the resulting platform runs as a live monitored system and produces the analyses that the pre-project state could not.

Three artefacts carry the claim. The ETL pipeline, built on a file-state lifecycle with commit-before-move semantics and conflict-safe inserts, handles Kafka, HTTP JSON, HTTP XML, and binary vector tile payloads through a single canonical contract. The PostgreSQL/PostGIS schema stores every source under deterministic primary keys and links them to a shared geometry registry, so a single `fid` identifier joins ANPR, Brussels counts, Waze, TomTom, magnetometers, and VISU signal state on the same road segment. The Brussels Fast Map turns that integrated database into an interactive surface where an analyst resolves a cross-source corridor question in two clicks instead of a morning of bespoke scripts, as the Rue Scailquin and Avenue Charles-Quint sections of Chapter 7 show.

8.2 Research Questions, Settled

8.2.1 RQ1: Coherent Spatio-Temporal Integration

The project shows that source heterogeneity can be absorbed when integration is organised around explicit canonical contracts rather than direct source-to-table coupling. Each source keeps its own ingester and its own transformation rules. Everything downstream of the canonical staging layer depends on a stable shape, which is the property that made the schema survive six different providers without rewriting the queries every time a new one arrived. The result is a schema that preserves source semantics while enabling common querying patterns across providers, and that exposes the shared spatial frame through one segment identifier.

8.2.2 RQ2: Robust Architecture Under Heterogeneous Updates

The combination of file-state boundaries, deterministic load ordering, and conflict-safe inserts gives the pipeline strong operational resilience for repeated runs and restart scenarios. Chapter 7 makes this visible in the Control Center capture: all six sources healthy, the loaded counter advancing, the intermediate states empty, and a single isolated error payload counted without crashing the cycle. The practical reliability result is that pipeline continuity holds without a high-complexity orchestration stack. A small team runs this system from one browser page.

8.2.3 RQ3: Operational Value

The platform establishes the technical precondition for segment-level, cross-source mobility analysis in a single database. Chapter 7 shows the precondition paying off on two real corridors. Rue Scailquin exhibits ANPR counts and TomTom relative speed moving as one signal across the same afternoon window. Avenue Charles-Quint shows magnetometer counts, signal capacity, and TomTom speed aligned on one intersection. The integration bottleneck that used to gate every retrospective study of a Brussels corridor is gone.

8.3 Contribution Statement

The contribution is applied and architectural. Its reusable output is an integration pattern for public mobility organisations, which the thesis documents as four concrete commitments:

1. isolate source volatility at ingestion boundaries, so a provider change does not ripple into the database layer;
2. enforce canonical staged contracts, so downstream code does not bind to any one source's schema;
3. persist with deterministic keys and replay-safe loading, so repeated runs and partial failures converge to the same state;
4. unify cross-source analysis through a shared geometric reference, so a single SQL join replaces a per-question reconciliation script.

The output is stronger than a prototype script collection, and more useful to the agency than a new theoretical data model. It is code that runs, schemas that hold, and a dashboard that exposes both.

8.4 Future Work

The roadmap below groups the next steps by risk reduction first and feature breadth second. One item stands out because it is already moving: the MobilityDB migration has started since this thesis was submitted, which is the reason the first subsection gets more space than the others.

8.4.1 MobilityDB: From Temporal-Aware to Temporal-Native

The schema presented in Chapter 5 stores time as conventional timestamp columns and lets the application layer reconstruct trajectories on demand. This works for the six original source families,

which publish aggregated observations on fixed windows. It does not scale to moving-object data, where the natural unit is a continuous trajectory rather than a row per time bucket.

MobilityDB [50] is the PostgreSQL/PostGIS extension that addresses this. It adds temporal types such as `tgeompoint` and `tfloat`, represented as instants, sequences, and sequence sets over continuous time, along with SQL operators for spatiotemporal joins, range queries, and nearest-neighbour searches. For Bruxelles Mobilité, the immediate benefits are three:

1. **Trajectories become first-class.** A vehicle’s path over a day is a single row with a `tgeompoint` sequence value, not a sorted batch of location samples.
2. **Time-window queries collapse to operators.** “Which segments did bus 27 traverse between 08:15 and 08:30?” is one SQL call against the temporal type, not a join over thousands of point rows.
3. **Historical replay shares the query path.** The same operator answers the live question and the retrospective one, which matches the “rewind the map” interaction of the Fast Map.

Since the end of the thesis, colleagues on the Bruxelles Mobilité team have continued the project and applied MobilityDB to a new source family: the STIB real-time feed for buses, trams, and metro. The feed streams position updates for the operating fleet, and the extended platform persists those updates as MobilityDB temporal types on the shared segment model. The Fast Map exposes the result through a dedicated *Transports* panel, shown in Figure 8.1.

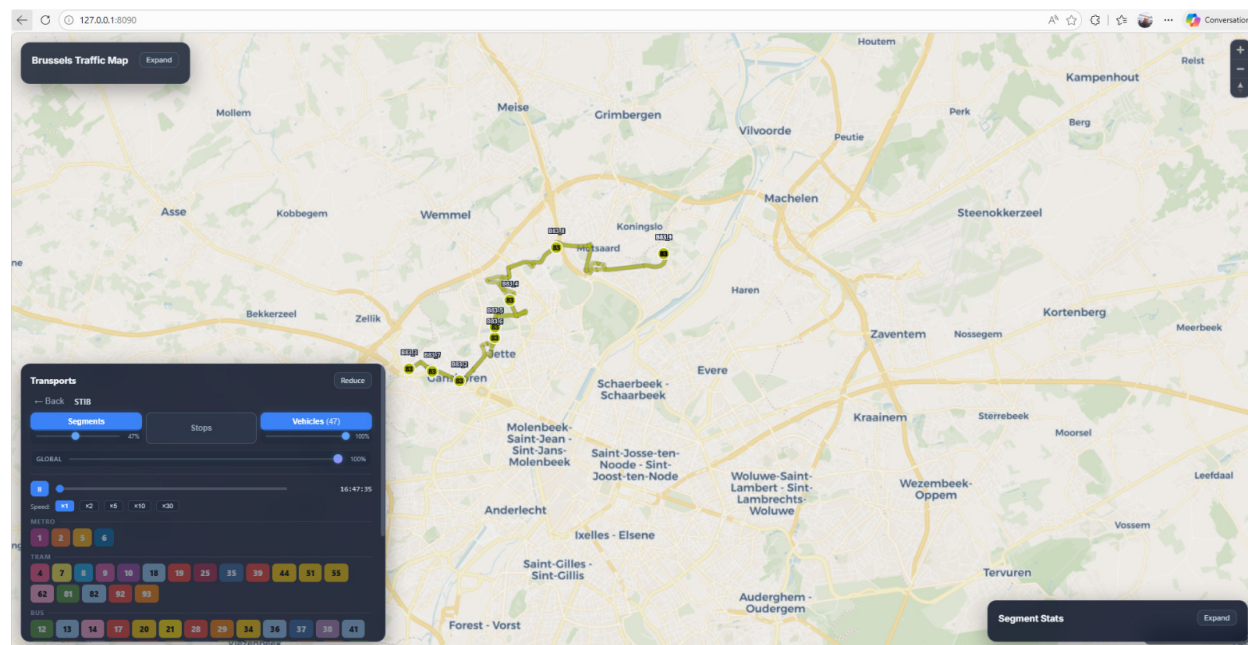


Figure 8.1: The Brussels Fast Map extended with real-time STIB data, after the MobilityDB migration started by the Bruxelles Mobilité team. The *Transports* panel on the left carries tabs for *Segments*, *Stops*, and *Vehicles* (47), a global coverage slider, a time slider with speed multipliers (x1, x2, x5, x10, x30), and per-line toggles for the metro, tram, and bus networks. The yellow-green trace on the map is one bus line replayed through its recorded trajectory between vehicle stops labelled B83/5, B83/12, and neighbouring IDs.

The *Transports* panel in Figure 8.1 is an operational proof that the MobilityDB direction was the right one. The panel renders a trajectory as a single continuous path, accepts a time slider that moves every vehicle together, and allows per-line filtering across metro, tram, and bus, all against the same shared segment model that Chapter 5 introduced. The architectural assumption that source volatility ends at the ingestion boundary held: adding STIB required a new ingestor and a new set of canonical staging rules, not a redesign of the schema.

The MobilityDB direction matters for the rest of the platform in three ways:

1. **Existing sources benefit without rewriting ingestors.** TomTom flow segments, Waze jam polylines, and VISU signal transitions all carry temporal semantics. Migrating them to temporal types compresses the storage, removes the application-side reassembly of time series, and lets a single SQL call answer “what was this corridor like during yesterday’s peak”.
2. **Retrospective analysis becomes cheap.** Historical comparisons across months or seasons are today bounded by query cost on wide fact tables. Temporal types with MobilityDB’s spatiotemporal indexes open retrospective studies that the current schema discourages.
3. **New moving-object feeds plug in.** Shared bikes, scooters, and freight vehicles share STIB’s structure. The ingestion pattern validated on STIB is reusable.

The work required to generalise the STIB success is documented rather than mystical: extend the geometry registry loader to emit temporal types for the sources where it makes sense, update the Fast Map tile queries to call MobilityDB operators, and replace the Python-side trajectory assembly with database-side temporal constructors. None of that invalidates the architecture of Chapter 3–Chapter 6. It sharpens it.

8.4.2 Production Hardening

Priority on the operations side is governance and reliability. Three concrete items belong here. Secret management moves from configuration files to a vault, because the current setup is acceptable for a thesis deployment and not for a production one. Observability moves from the Control Center dashboard and log tails to structured metrics with alerting thresholds on source freshness, backlog growth, and error-payload rate. Retention policies align with GDPR: ANPR and camera-count records carry personal-data implications that require documented retention windows and an audit trail for deletions.

8.4.3 Correction-Aware Loading

The current load path uses `ON CONFLICT DO NOTHING`, which is the correct choice for replay safety on append-only measurement data. It is not the correct choice for sources where a record can be revised after publication: TomTom incidents get re-classified, Waze jams get re-evaluated by crowdsourcing, VISU cycles get late corrections. A second load mode that resolves conflicts with a provenance-aware upsert, tracked in a small history table, covers those cases without losing the replay-safe default for everything else.

8.4.4 Linkage Quality Instrumentation

The next step for the segment-linking layer is confidence-weighted linkage: attach a match score to each geometry-to-segment assignment, expose it as an optional filter in the Fast Map, and sample a small share of the assignments for manual validation. That turns the silent tail of edge cases in dense urban topologies into a measurable quality signal.

8.4.5 Further Sources and Retrospective Analytics

Two source extensions would compound with the MobilityDB direction. The KMI weather feed gives an exogenous signal for congestion drivers beyond the road network. The SNCB/NMBS suburban rail timetable adds the regional rail dimension to the multimodal picture. Both are HTTP JSON feeds that fit the existing ingestor pattern.

On the analytics side, the platform today is tuned to recent time windows. Seasonal comparisons, year-over-year peak analyses, and incident after-action reviews need efficient retrospective queries over months of data. Those are the workloads MobilityDB targets, which is why the two directions reinforce each other.

8.5 Final Perspective

The main result of this thesis is an architecturally legible platform. Assumptions are stated. Failure modes are identifiable. Extension points are marked on the plan rather than discovered under pressure. The STIB extension in Figure 8.1 is the simplest evidence that those properties hold in practice: a new team, a new source, a new technology stack (MobilityDB), and the same segment model and the same Fast Map surface as before. Bruxelles Mobilite now has a foundation on which the next feeds, the next analyses, and the next operational tools can be built without rebuilding the house.

Bibliography

- [1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015. doi: 10.14778/2824032.2824076.
- [2] M. Amin-Naseri, P. Chakraborty, A. Sharma, S. B. Gilbert, and M. Hong. Evaluating the reliability, coverage, and added value of crowdsourced traffic incident reports from waze. *Transportation Research Record*, 2672(43):34–43, 2018. doi: 10.1177/0361198118790619.
- [3] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, 2021.
- [4] Mohamed Bakli, Mahmoud Sakr, Esteban Zimányi, Nils Dijk, and Marco Slot. Distributed mobilitydb: A scalable moving object database management system. *ACM Transactions on Spatial Algorithms and Systems*, 11(2):1–39, 2025. doi: 10.1145/3719202.
- [5] Xavier Bonnemaizon, Philippe Ciais, Chuanlong Zhou, Qinren Shi, Rohith Teja Mittakola, Clément Goldmann, Simon Ben Arous, Nicolas Megel, et al. Harmonized annual averaged traffic data at street segment level for european cities. *Scientific Data*, 12(1):1365, 2025. doi: 10.1038/s41597-025-05698-y.
- [6] Zouhaier Brahmia, Fabio Grandi, and Barbara Oliboni. A literature review on schema evolution in databases. *Computing Open*, 2:2430001, 2024. doi: 10.1142/S2972370124300012.
- [7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.
- [8] Databricks. What is a medallion architecture? <https://www.databricks.com/glossary/medallion-architecture>, 2024. Accessed 2026-02-10.
- [9] Zhamak Dehghani. *Data Mesh: Delivering Data-Driven Value at Scale*. O’Reilly Media, 2022.
- [10] Lisa Ehrlinger and Wolfram Wölk. A survey of data quality measurement and monitoring tools. *Frontiers in Big Data*, 5:850611, 2022. doi: 10.3389/fdata.2022.850611.
- [11] European Commission. Real-time iot stream processing and large-scale data analytics for smart city applications (citypulse). <https://cordis.europa.eu/project/id/609035>, 2017. CORDIS project fact sheet.

- [12] European Parliament and Council of the European Union. Directive 2007/2/ec establishing an infrastructure for spatial information in the european community (inspire). <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32007L0002>, 2007.
- [13] European Parliament and Council of the European Union. Regulation (eu) 2016/679 (general data protection regulation). <https://eur-lex.europa.eu/eli/reg/2016/679/oj>, 2016.
- [14] European Parliament and Council of the European Union. Directive (eu) 2019/1024 on open data and the re-use of public sector information. <https://eur-lex.europa.eu/eli/dir/2019/1024/oj>, 2019.
- [15] Mina Farmanbar and Chunming Rong. Triangulum city dashboard: An interactive data analytic platform for visualizing smart city performance. *Processes*, 8(2):250, 2020. doi: 10.3390/pr8020250.
- [16] FIWARE Data Models Community. Smart data models. <https://smartdatamodels.org/>, 2025. Accessed 2026-02-10.
- [17] FIWARE Foundation. Fiware: Open source platform for smart solutions. <https://www.fiware.org/>, 2025. Accessed 2026-02-10.
- [18] Anita Graser. Movingpandas: Efficient structures for movement data in python. *GI_Forum, Journal of Geographic Information Science*, 7(1):54–68, 2019. doi: 10.1553/giscience2019_01_s54.
- [19] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000. doi: 10.1145/352958.352963.
- [20] Sheida Hadavi, Heleen Buldeo Rai, Sara Verlinde, He Huang, Cathy Macharis, and Tias Guns. Analyzing passenger and freight vehicle movements from automatic-number plate recognition camera data. *European Transport Research Review*, 12(1):37, 2020. doi: 10.1186/s12544-020-00405-x.
- [21] Nima Hoseinzadeh, Yuandong Liu, Lee D. Han, Candace Brakewood, and Amin Mohammad-nazar. Quality of location-based crowdsourced speed data on surface streets: A case study of waze and bluetooth speed data in sevierville, tn. *Computers, Environment and Urban Systems*, 83:101518, 2020. doi: 10.1016/j.compenvurbsys.2020.101518.
- [22] William H. Inmon. *Building the Data Warehouse*. Wiley, 4 edition, 2005.
- [23] Jaeun Jung, Taeho Oh, Inhi Kim, and Shinhyoung Park. Open-sourced real-time visualization platform for traffic simulation. *Procedia Computer Science*, 220:243–250, 2023. doi: 10.1016/j.procs.2023.03.033.
- [24] Ralph Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. Wiley, 1996.
- [25] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, 2004.

- [26] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.
- [27] Hiroaki Kondoh, Tetsuya Aoyama, and Naoki Seo. Building a common smart city platform utilizing fiware (case study of takamatsu city). *NEC Technical Journal*, 13(1):36–40, 2018.
- [28] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. <https://kafka.apache.org/documentation/>, 2011. Accessed 2026-02-10.
- [29] Carlos Lemonde, Elisabete Arsénio, and Rui Henriques. Integrative analysis of multimodal traffic data: Addressing open challenges using big data analytics in the city of lisbon. *European Transport Research Review*, 13(1):64, 2021. doi: 10.1186/s12544-021-00520-3.
- [30] Lubna, Naveed Mufti, and Syed Afaq Ali Shah. Automatic number plate recognition: A detailed survey of relevant algorithms. *Sensors*, 21(9):3028, 2021. doi: 10.3390/s21093028.
- [31] Adam Millard-Ball, Robert C. Hampshire, and Rachel R. Weinberger. Map-matching poor-quality gps data in urban environments: the pgmapmatch package. *Transportation Planning and Technology*, 42(6):539–553, 2019. doi: 10.1080/03081060.2019.1622249.
- [32] Muzeeb Mohammad. Analysis of design patterns and benchmark practices in apache kafka event-streaming systems. *arXiv preprint arXiv:2512.16146*, 2025. doi: 10.48550/arXiv.2512.16146.
- [33] T. T. Nguyen et al. Indexing postgis databases and spatial query performance evaluations. *International Journal of Geoinformatics*, 5(3):1–9, 2009.
- [34] Open Geospatial Consortium. Opendata simple feature access – part 1: Common architecture. <https://www.ogc.org/standards/sfa/>, 2011.
- [35] Nikos Pelekis, Elias Frentzos, Nikos Giatrakos, and Yannis Theodoridis. Hermes: A trajectory db engine for mobility-centric applications. *International Journal of Knowledge-Based Organizations*, 5(2):19–41, 2015. doi: 10.4018/ijkbo.2015040102.
- [36] PostGIS Project Steering Committee. Postgis documentation. <https://postgis.net/documentation/>, 2025. Accessed 2026-02-10.
- [37] Fabian Rechsteiner. Performance comparison of open source vector tiles server solutions for providing geodata from postgis databases. <https://github.com/FabianRechsteiner/vector-tiles-benchmark>, 2024. Master's thesis benchmark repository.
- [38] Joe Reis and Matt Housley. *Fundamentals of Data Engineering: Plan and Build Robust Systems*. O'Reilly Media, 2022.
- [39] Tim C. Rese, Alexandra Kapp, and David Bermbach. Evaluating the impact of spatial features of mobility data and index choice on database performance. *arXiv preprint arXiv:2505.14466*, 2025.
- [40] Siavash Saki and Tobias Hagen. A practical guide to an open-source map-matching approach for big gps data. *SN Computer Science*, 3(5):415, 2022. doi: 10.1007/s42979-022-01340-5.
- [41] Mahmoud Sakr, Alejandro Vaisman, and Esteban Zimányi. *Mobility Data Science: From Data to Insights*. Springer, 2025. doi: 10.1007/978-3-031-82636-8.

- [42] Mario Scrocca, Marco Grassi, Marco Comerio, Valentina Anita Carriero, Tiago Delgado Dias, Ana Vieira Da Silva, and Irene Celino. Intelligent urban traffic management via semantic interoperability across multiple heterogeneous mobility data sources. In *The Semantic Web – ISWC 2024*, Lecture Notes in Computer Science, pages 218–235. Springer, 2024. doi: 10.1007/978-3-031-77847-6_12.
- [43] Rodrigo Smarzaro, Clodoveu A. Davis, and José Alberto Quintanilha. Creation of a multi-modal urban transportation network through spatial data integration from authoritative and crowdsourced data. *ISPRS International Journal of Geo-Information*, 10(7):470, 2021. doi: 10.3390/ijgi10070470.
- [44] Junqing Tang, Li Wan, Jennifer Schooling, Pengjun Zhao, Jun Chen, and Shufen Wei. Automatic number plate recognition (anpr) in smart cities: A systematic review on technological advancements and application cases. *Cities*, 129:103833, 2022. doi: 10.1016/j.cities.2022.103833.
- [45] TomTom. Traffic api documentation: Incident details and vector flow tiles. <https://developer.tomtom.com/traffic-api/documentation/tomtom-maps/v1/traffic-incidents/incident-details>; <https://developer.tomtom.com/traffic-api/documentation/tomtom-maps/v1/traffic-flow/vector-flow-tiles>.
- [46] Panos Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining*, 5(3):1–27, 2009. doi: 10.4018/jdwm.2009070101.
- [47] Matei Zaharia, Reynold Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael Franklin, et al. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016. doi: 10.1145/2934664.
- [48] Desheng Zhang, Juanjuan Zhao, Fan Zhang, Tian He, Haengju Lee, and Sang H. Son. Heterogeneous model integration for multi-source urban infrastructure data. *ACM Transactions on Cyber-Physical Systems*, 1(1):1–26, 2017. doi: 10.1145/2967503.
- [49] Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. Mobilitydb: A mainstream moving object database system. In *Proceedings of the 16th International Symposium on Spatial and Temporal Databases*, pages 206–209, 2019. doi: 10.1145/3340964.3340991.
- [50] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. Mobilitydb: A mobility database based on postgresql and postgis. *ACM Transactions on Database Systems*, 45(4):19:1–19:42, 2020. doi: 10.1145/3406534.