Building and Publishing a Railway Infrastructure Knowledge Graph: Lessons Learned from SPARQL Anything, GeoSPARQL, SHACL and RINF Publication

Mathias Vanden Auweele[1], Linnea Catharina Olsen[2], Mia Pin Berge[2] and Johan Korsmo[2]

[1]Matdata, Belgium
[2]Bane NOR, Norway

Abstract

Railway infrastructure data is fragmented across topology models, asset systems, engineering systems, geographic information systems and regulatory datasets; the same railway is represented differently in each. A knowledge graph offers a semantic integration layer that unifies these silos around a shared model. We report on the construction and publication of a production railway infrastructure knowledge graph for the Norwegian infrastructure manager, comprising roughly 13 million triples across seventeen ontologies and covering more than five thousand kilometres of track. Unlike many geospatial domains, railway infrastructure is fundamentally *topology-based*: assets are positioned through linear referencing and route-based coordinates rather than absolute geometry. We describe a pragmatic ingestion pipeline built on SPARQL Anything, the evolution from post-processing SPARQL queries towards declarative SHACL rules, and the generation of a European Register of Infrastructure (RINF) export directly from the graph. Our central finding is that, while SPARQL Anything, SHACL and GeoSPARQL enable much of the required functionality, our experience demonstrates that railway infrastructure knowledge graphs require stronger support for topology-based modelling, routing and linear referencing than is currently available in mainstream Semantic Web tooling. As a consequence, a significant amount of spatial and topological logic had to remain in SpatiaLite SQL or in custom Python scripts outside the graph – an observation we believe is relevant to the wider community.

Keywords

Knowledge Graph, Railway Infrastructure, SPARQL Anything, GeoSPARQL, SHACL, Linear Referencing, Topology, RINF, GIS

1. Introduction

1.1. Problem

A railway infrastructure manager is responsible for far more than running trains on time. The infrastructure itself – thousands of kilometres of track, together with signals, switches, bridges, tunnels and overhead lines – must be built, maintained, renewed and planned decades ahead. A wide range of business domains depend on accurate and consistent infrastructure data: asset maintenance (track, catenary, signalling, telecoms, structures), infrastructure projects (ERTMS rollout, track renewal, station upgrades), operations (traffic management, capacity planning, timetabling, simulation), stakeholder communication (train operators, regulators, the public) and corporate functions (finance, contracts, procurement).

Historically, each of these use cases evolved its own application, its own data model and its own copy of the infrastructure. The result is dozens of overlapping, inconsistent representations of the *same* railway – sometimes so divergent that they appear to describe a different railway entirely. Changes made in one system do not propagate to the others; reconciliation is manual and error-prone. Railway infrastructure data thus exists simultaneously in topology models, asset systems, engineering systems, geographic information systems (GIS) and regulatory datasets, each with its own identifiers, granularity and assumptions.

A knowledge graph offers a semantic integration layer over these silos: a single, authoritative model of the infrastructure that can serve reference data to all use cases from one source of truth, and that grounds transactional data exchange in solid, shared reference data. This is the vision pursued by the *Digital Infrastructure Model* (DIM) of the Norwegian infrastructure manager Bane NOR [1]: consolidating data from multiple sources into one connected model that gives all user groups access to traffic-oriented information.

1.2. Challenge

Realising this vision surfaces difficulties that reach well beyond the geometric concerns of most GIS applications. Managing railway asset and network data involves three fundamental, generic and often neglected complexities [2]:

- **Location** – an asset must be positioned along several complementary dimensions: *topological* (where the asset sits in the network graph, and thus how it affects train routing – the most critical dimension), *geographic* (latitude and longitude), *linear* (a measure along a line) and *address-based* (kilometrage).
- **Time** – both the time of *knowledge* (what we already know today about a future state of the network) and the time of *validity* (when a change actually becomes operational) must be represented.
- **Identity** – the identity of an entity, of its lifecycle phase (a distinct identifier per phase plus one canonical, phase-less identity) and of its aspect (for example, equipment versus function).

A production infrastructure model must ultimately address all three to become a stable product, and a knowledge graph is well suited to doing so: it allows independent metadata to be attached to each individual property, so that a complex model can still expose a simple snapshot of the network through a few rules. *In this paper, however, we deliberately limit the discussion to the first complexity, and within it to the topological and spatial dimensions of location.*

Sem4Tra 2026: Seventh International Workshop on Semantics for Transport and Logistics, co-located with the SEMANTiCS Conference 2026, September 15, 2026, Ghent, Belgium
*Corresponding author.
mathias@matdata.eu (Mathias Vanden Auweele); linnea.catharina.olsen@banenor.no (L. C. Olsen); mia.pin.berge@banenor.no (M. P. Berge); johan.korsmo@banenor.no (J. Korsmo)
0009-0004-4838-7029 (Mathias Vanden Auweele)

Time and identity, though equally important, are beyond our scope here.

Even so restricted, the problem is hard, because these dimensions are only partially supported by current Semantic Web standards. Where cadastral, utility or generic GIS applications are primarily concerned with *geometry* – points, lines and polygons in a coordinate reference system – railway infrastructure relies heavily on:

- **topology** – the network as a graph of nodes and edges, with explicit navigability;
- **linear referencing** – positioning assets along a track by a measure (e.g. kilometrage or an intrinsic coordinate) rather than by an absolute point;
- **route-based positioning** – locations that only make sense along a path through the network.

GeoSPARQL [3] is geometry-centric; it offers rich handling of geometries, spatial predicates and coordinate reference systems, but it has no native notion of linear referencing, kilometrage, routing or topology traversal. As we shall show, this mismatch has practical consequences for any railway knowledge graph built on today's tooling.

1.3. Contribution

This paper presents an experience report of a production railway knowledge graph. Concretely, we contribute:

1. a description of a *production* railway KG construction pipeline, including its pragmatic workarounds (Section 3);
2. lessons learned using *SPARQL Anything* [4] and the Facade-X [5] approach for ingesting heterogeneous railway source formats;
3. the evolution from SPARQL post-processing towards *SHACL*-based transformation [6, 7] (Section 4);
4. the limitations we encountered with *GeoSPARQL* for topology-centric and linear-reference-centric data (Section 6);
5. the generation of a *RINF* publication directly from the graph (Section 5).

Our aim is not to claim a novel algorithm, but to distil what worked, what did not, and where the Semantic Web stack falls short for a topology-driven domain.

2. Railway Topology as the Foundation

The single most important insight from our work is that railway infrastructure is fundamentally topology-based. Before describing the pipeline, we explain what this means and why it matters.

2.1. Many Topology Models Coexist

Unfortunately, there is no single railway topology model. Several standards exist and are in active use at the same time, often within one organisation:

- **RailTopoModel (RTM)** [8] – the UIC standard, published as IRS 30100, that first established a topology-based foundation for positioning on the railway network. It models the network with net elements and net relations expressing navigability, at configurable levels of detail (micro, meso, macro). Version 1.1 is the most widely shared baseline; the next three models below can largely be understood as forks of it.
- **railML 3.x** [9] – the XML serialisation that adopted RTM for positioning and then continued to develop the topology model under the railML umbrella (railML 3.1/3.2/3.3 tracking RTM 1.3/1.4/1.5). The earlier and still widely deployed **railML 2.x** generation is track-and-line oriented and does *not* carry a full topology.
- **RSM (Rail System Model)** [10] – UIC's own continuation of RTM (effectively RTM 1.2), developed after UIC and railML diverged.
- **ERA ontology** [11, 12] – the European regulatory model behind RINF. It began as a macro-level model of operational points and sections of line; more recently, RTM concepts were introduced to support micro topology, but with renamed classes (for example `SpotLocation` becomes `NetPointReference`). Its topology is therefore *inspired by* RTM but not identical to it – a divergence that later causes concrete conversion problems (Section 5).
- **OpenTNF** [13] – an open topology network format built as an extension of GeoPackage [14] (itself an extension of SQLite). Nodes represent switches and ends of track; edges represent track segments.
- **Proprietary formats** – many operational and engineering systems carry their own topology and asset-positioning schemes. The authors have semantic-mapping experience with, among others, Hacon's TPS, TrafIT's railOscope and Infrabel's InfraNET/InfraGIS [15, 16, 17], each with its own take on topology and positioning.
- **OpenStreetMap** [18] – a crowdsourced geometric view with partial and inconsistent topology.

Each model draws the network boundary differently, chooses a different granularity, and uses different identifiers. Integrating source systems is, therefore, fundamentally an act of mapping their base topologies onto one unified topology. Once a shared topology exists, assets from different source systems can coexist and relate to one another.

2.2. Graph-Based Topology

At its core a topology models the railway as a graph (Figure 1): a set of *nodes* connected by *edges*. [1]

The models also differ in how much structure they guarantee. In RTM the network is described by *net elements* connected by *net relations*, and navigability – which movements through a node are physically possible – is expressed explicitly as a property of those relations, which makes RTM a good integration target. Not every model is so disciplined: some topology models do not represent navigability at all, and some do not restrict nodes to switches and ends of track but instead attach signals and other assets directly

[1] A crucial caveat is that this *topology graph* is not the same thing as the *knowledge graph* into which it is eventually encoded. In a topology graph, nodes and edges denote physical network primitives – connection points and track segments. Once serialised as RDF, those primitives become resources, and the RDF edges (triples) instead express relationships *about* them. In other words, nodes and edges carry a different meaning in the two graphs, and conflating them is a common source of confusion.

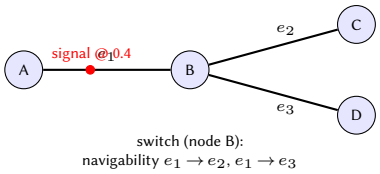

Figure 1: A railway topology as a graph. Node B is a switch connecting edges e_1, e_2 and e_3. The signal is positioned by an intrinsic coordinate (0.4) along edge e_1, not by an absolute point.

into the topology graph. The latter choice severely complicates topology maintenance, because moving or renewing an asset then perturbs the network structure itself.

OpenTNF illustrates the practical gaps. Its nodes are switches and ends of track and its edges are track segments, but it has no native concept of navigability; in practice this has to be reconstructed in a proprietary manner. OpenTNF also uses a `link_sequence` construct – a kind of "super-edge" that spans several smaller edges – so a necessary first ingestion step is to decompose each link sequence back into the most basic edges before the topology can be mapped onto the unified model.

Once navigability is present, this graph structure yields three properties that plain geometry cannot:

- **Unambiguous positioning** – every asset has a precise, unique location on the network.
- **Clear navigability** – the graph can be traversed to find routes, adjacencies and reachability.
- **Things, not strings** – positioning does not rely on human-readable names or kilometre posts, which are ambiguous and change over time; it relies on the structure itself.

2.3. Asset Positioning

Assets in source systems are still typically recorded by their absolute geometric position (or worse.. with a kilometric position...), but what matters for railway operations is their *topological* position, because that is what determines how an asset affects train traffic. The distinction is not academic: a signal may physically stand closer to a neighbouring track than to the track it actually governs, so a purely geometric "nearest track" assignment would attach it to the wrong edge. Only an explicit topological position captures the intended meaning.

When positioning assets against a topology, three complementary methods appear throughout the data:

- **Intrinsic coordinates** – the asset is bound to one or more net elements through an associated-net-element reference, with an intrinsic coordinate (a value between 0 and 1) giving its relative position along an edge. A signal "at 0.4 on edge e_1" is unambiguous regardless of the edge's absolute geometry. This is the most important method, being the only one intrinsically tied to the topology.
- **Geometric coordinates** – ordinary spatial coordinates, but used as a reference to the topological position rather than to the true physical location of the asset. Keeping a geometry that follows the track makes visual representation easy while the position remains topologically meaningful.
- **Linear coordinates** – a measure along a linear positioning system. Depending on that system, the measure may be relative to an anchor (such as a kilometric post) or relative to the origin of the positioning system; the latter makes distance calculations between positions straightforward along that specific axis.

Kilometrage – the traditional railway "km + offset" notation – deserves a specific warning. Although it superficially resembles a linear referencing system [19], in practice it is so riddled with misuse and hidden issues (it resets at boundaries, contains gaps and overlaps, and changes when a line is re-surveyed) that it is safest to treat it as a mere *addressing* system rather than a reliable measure to compute with.

3. Knowledge Graph Construction Pipeline

We now describe how the graph is built. To keep the focus on the semantic lessons, we present a simplified view here and defer the operational plumbing (orchestration, scheduling, clustering) and performance figures to Section 7. The pipeline is mature and runs in production: it currently produces roughly 13 million triples across seventeen ontologies, covering more than five thousand kilometres of track [1].

3.1. The Vision Versus the Reality

We set out to build a simple three-step pipeline: multiple heterogeneous source systems feed a single knowledge graph, from which railML files are generated for downstream use cases. The railML files act as a bridge, so that existing tools keep working without modification while use cases gradually move towards consuming RDF directly. Reality required considerably more machinery, summarised in Figure 2.

3.2. Choosing a Base Railway Ontology

No single railway ontology covers all of our requirements, so the choice of a base vocabulary was a deliberate trade-off. We selected the railML ontology [20] as the backbone for railway-specific concepts. It is derived from the railML schema, a mature and proven exchange format that already covers a broad catalogue of documented railway use cases [21], and it inherits the solid RTM foundation for topology (Section 2). Although the *ontology* is still under active development, the underlying *schema* is very stable, which gives us confidence in the modelling. By contrast, the ERA ontology [11], while mandated for regulatory publication, does not yet have a comparable track record of proven use cases; we therefore treat it as an export target (Section 5) rather than as our internal backbone.

It is worth being precise about *what* is being compared. The railway standards of Section 2 fall into two families. RTM [8], RSM [10] and OpenTNF [13] are essentially *topology, positioning and entity-defining* schemas: they establish how the network graph and locations along it are described, but they deliberately say little about the assets themselves. railML 3.x [9] and the ERA ontology [11], by contrast, are *asset-describing* models that build on such a topological foundation to catalogue the physical and functional infrastructure. A meaningful requirements comparison therefore only makes sense *within* a family; pitting an asset ontology

against a pure topology format such as RTM would not be fair. For that reason we restrict the comparison in Table 1 to the two asset-describing models that are genuine candidates for our backbone: railML 3.x and the ERA ontology.

The single respect in which ERA is clearly ahead is that it is a *native* ontology with first-class RINF compatibility: it is expressed in OWL/RDF and is the regulatory target for European infrastructure reporting, whereas railML is primarily an XML exchange format whose ontology is a more recent, still-maturing derivative and whose data must be *mapped* to satisfy RINF (Section 5). Both models are open and community-governed, although railML carries some licensing restrictions on its schema. On virtually every other axis, however, railML is currently the more complete choice for an internal backbone. It is a widely used exchange format with proven industrial adoption and a detailed asset catalogue, and it supports a number of capabilities that the ERA ontology does not yet express at all – among them:

- railML distinguishes over- and undercrossings, where ERA offers only generic bridges and tunnels;
- railML supports network levels, element aggregation and the description of multiple coexisting networks;
- railML can convert between topological (intrinsic), linear (kilometric – across several linear positioning systems) and geographic coordinates, which ERA cannot;
- railML describes left/right and through/diverging branches at a switch, absent in ERA;
- railML models signaling panels (speed indicators, pantograph and traction controls, and so on) that have no ERA counterpart;
- railML treats overhead contact line, etcs areas, speed sections, gradients and curves as first-class positioned entities, whereas ERA links them to a track;
- railML covers interlocking (routes, locks, signal aspects, work zones), schematic visualisations and timetabling (itineraries, trains, connections, rosterings) – none of which the ERA ontology currently addresses.

None of this is a criticism of the ERA ontology's trajectory. It is a young, regulatory-driven model, and we have every confidence that it will evolve to overcome these limitations; the authors indeed intend to contribute, in their own capacity, to that evolution. For the specific role of an *internal backbone* that must already express the full richness of the infrastructure today, however, railML remains the pragmatic choice, and we treat the ERA ontology as an export target (Section 5).

For the many generic, non-railway-specific aspects we reuse established W3C and community ontologies rather than reinventing them: OWL-Time for temporal entities, QUDT for quantities and units, SKOS for code lists and concept schemes, GeoSPARQL for geometry, schema.org and Dublin Core Terms for descriptive metadata, and PROV for provenance. This keeps the railway-specific vocabulary small and lets standard tooling operate on the generic parts of the graph.

3.3. Data Ingestion with SPARQL Anything

Our team was new to knowledge graph construction and evaluated several mapping approaches (R2RML [22], RML [23], custom code). The decisive factor was that we already had to learn SPARQL for querying the graph – so why learn yet another mapping language? SPARQL Anything [4] lets us write CONSTRUCT queries that map source data into RDF through the Facade-X lens [5], using a single language for both mapping and querying. Our primary source formats are XML, CSV and the OpenTNF/GeoPackage database.

The authoritative source turned out not to be raw railML 2.x XML (which was incomplete and carried no topology) but OpenTNF, whose spatial foundation is a topology. Because SPARQL Anything operates on files rather than databases, we built a *Source Ingestor* that exports the relevant SQL tables to CSV. The ingestor later took on additional responsibilities (Sections 4 and 6). Each exported CSV is asset-based (one file per asset type – switches, signals, balises, speed sections, and so on), and each is mapped by one CONSTRUCT query. The mapping layer covers roughly forty asset types. Listing 1 shows the recurring pattern.

Listing 1: Asset mapping with SPARQL Anything: a SERVICE clause reads a CSV through Facade-X; the CONSTRUCT builds railML/RTM triples and positions the asset by intrinsic coordinate.

```
CONSTRUCT {
  ?balise a railml3:Balise, rtm:NetEntity ;
    rdfs:label ?label ;
    rtm:hasSpotLocation ?loc .
  ?loc a rtm:SpotLocation ;
    rtm:onNetElement ?netelement ;
    rtm:hasIntrinsicCoordinate ?coord .
}
WHERE {
  SERVICE <x-sparql-anything:location=
      balises.csv,csv.headers=true> {
    ?s xyz:atb_banedata_id ?id ;
       xyz:name ?label ;
       xyz:measure_on_link ?coord .
  }
  BIND(IRI(CONCAT(STR(bnd:),
      "_balise_", STR(?id))) AS ?balise)
}
```

3.4. Loading, Post-Processing and Export

The generated Turtle is loaded into an Apache Jena TDB2 dataset and served through Fuseki [24] as a SPARQL endpoint. Because a single pass of CONSTRUCT mapping is not enough (see Section 4), a series of post-processing queries runs against the freshly loaded graph, followed by fifteen sequential *splitting* queries that extract subnetworks for railML export. Crucially, all of this post-processing takes place on a *staging* TDB2 dataset; only once post-processing has completed successfully do we *hot-swap* the finished dataset behind the stable endpoint name, so that the graph can be updated without ever exposing consumers to a partially processed state.

4. Semantic Transformation with SHACL

A recurring theme in our work is *where* semantic transformation should happen. This section traces that evolution, which we consider more instructive than the mapping queries themselves.

4.1. Initial Approach: SPARQL CONSTRUCT

Initially, all mapping was performed with CONSTRUCT queries at ingestion time (Listing 1). This works well for single-file, asset-by-asset mappings and gives a low entry barrier. It

Table 1
Requirements matrix comparing the two asset-describing candidate ontologies for the internal backbone. See the note below for the rating scale.

Requirement	railML 3.x	ERA ontology
Native ontology	○	●
Open and community-governed	◕	●
RINF compatibility	◑	●
Widely used exchange format	●	◑
Proven industrial adoption	●	◑
Detailed railway asset catalog	◕	◔
Topology management	◔	○
Schematic track plan	◕	◔
Network statement	◕	◑
Route time calculation	◕	○
Timetable management	◕	○
ETCS data exchange	◕	○

Rating scale (Harvey balls, more filled is stronger): ● full / native support, ◕ good support, ◑ emerging or partial support, ◔ limited support, ○ none or not (yet) available.

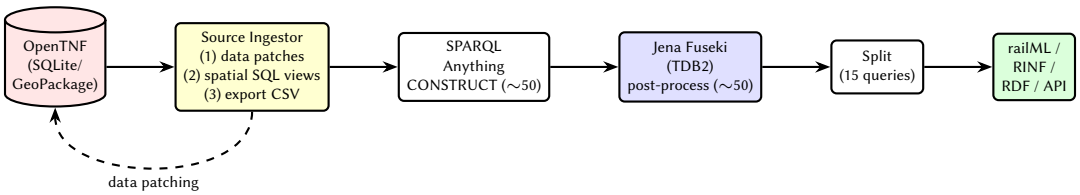

Figure 2: Simplified production pipeline. The Source Ingestor performs data patches and spatial SQL before export; SPARQL Anything maps CSV to RDF; Jena Fuseki hosts the graph and runs post-processing; splitting queries extract subnetworks for railML and RINF export. Orchestration is omitted for clarity.

breaks down, however, as soon as relationships span multiple source files. Inferring where a track begins and ends, for example, requires resolving how tracks connect through switches and buffer stops across several asset files – awkward to express with nested SERVICE clauses in SPARQL Anything, which reads one file at a time.

4.2. Evolution: Post-Processing Queries

We therefore moved cross-cutting logic into post-processing INSERT/DELETE queries that run against the fully loaded graph. This solved two problems: (i) cross-file relationships, now expressible over the assembled graph; and (ii) generalised actions applied uniformly to many entity types – for example completing or normalising values that individual mapping queries left implicit. Listing 2 shows a representative case: deriving the missing `time:inXSDDate` of a time instant from its `time:inXSDDateTimeStamp`, applied uniformly to every instant in the graph.

Listing 2: Representative post-processing: a generalised INSERT that completes the missing date on any `time:Instant` from its timestamp, applied uniformly across the whole graph.

```
INSERT {
  ?instant time:inXSDDate ?date .
}
WHERE {
  ?instant a time:Instant ;
    time:inXSDDateTimeStamp ?dateTimeStamp .
  FILTER NOT EXISTS {
    ?instant time:inXSDDate ?d . }
  BIND(STRDT(SUBSTR(STR(?dateTimeStamp), 1, 10),
    xsd:date) AS ?date)
}
```

The cost of this approach becomes apparent at scale. We accumulated roughly fifty post-processing queries that must run in a specific order, because some steps depend on the output of earlier ones. The dependencies are encoded only by a numeric filename prefix and comments; there is no explicit dependency graph, no validation that a step's preconditions hold, and no lineage recording which source triples produced which derived triples. In effect, the post-processing layer is an imperative program written in SPARQL.

Earlier iterations of the pipeline also performed multi-ontology mapping in this layer, adding ERA types alongside railML ones in the same post-processing step. We have since moved away from that: mixing regulatory ERA mapping into general post-processing meant many different semantic actions happened at once, which increased complexity. Conversion to the ERA ontology is now done *exclusively* in the RINF export through SHACL-AF rules (Section 5), keeping the post-processing layer focused on graph-internal completion.

4.3. Future Direction: SHACL Rules

The natural next step is to replace ad-hoc post-processing with declarative SHACL rules [6, 7]: first ingest a raw, faithful graph that mirrors the source structure, triplified according to Facade-X [5][2], then apply SHACL rules to perform the semantic mapping to the target ontologies. In our pipeline this is already in use for the regulatory RINF export, where target properties are derived through SHACL-AF rules and validated against ERA's official shapes before each export.

Listing 3 shows a representative rule from that export. It targets the railML-centred graph and mints the corresponding ERA entity: every operational point carrying a primary-

[2]We will deviate slightly from the default Facade-X metamodel towards a "one-eyed graph" [25] which will improve SHACL rule readability.

location-code (PLC) designator becomes an `era:PrimaryLocation` in the RINF-oriented view.

Listing 3: A SHACL-AF rule mapping the railML-centred graph towards the RINF/ERA view: each operational point with a PLC designator yields an `era:PrimaryLocation`. The rule is attached to a node shape whose target selects the currently valid operational points.

```
bnd:_rule_mint_primary_location
  a sh:SPARQLRule ;
  sh:prefixes bnd:_prefixes ;
  sh:construct """
    CONSTRUCT {
      ?pl a era:PrimaryLocation ;
        era:primaryLocationCode ?plc ;
        era:primaryLocationName ?label .
    }
    WHERE {
      $this rdfs:label ?label ;
        railml3:hasDesignator ?d .
      ?d railml3:hasEntry ?plc ;
         railml3:hasRegister "PLC" .
      BIND(IRI(CONCAT(STR(bnd:),
        "_primaryLocation_", ?plc)) AS ?pl)
    }
  """ .
```

The advantages align well with current Semantic Web discussions:

- **Separation of concerns** – ingestion is decoupled from semantic mapping, so each can evolve independently.
- **Maintainability** – rules are declarative shapes rather than an ordered pile of imperative queries. Their execution order is not hand-coded but determined by the rules engine, which builds a dependency graph over the rules and typically applies them repeatedly until a fixpoint is reached and no new triples are inferred.
- **Lineage** – the rules are themselves triples in the graph and can therefore be annotated, which makes it natural to record which rule and which source triples justify each derived triple.
- **Validation and transformation together** – the same shape language expresses both what is valid and how to derive missing facts, and can drive API generation and dereferencing.

The main obstacle is tooling maturity: we return to this in Section 8.

5. RINF Publication

The European Register of Infrastructure (RINF) is a common register maintained by the European Union Agency for Railways, into which every infrastructure manager must report a defined set of infrastructure properties [12, 26]. RINF exists for *interoperability*: it enables cross-border reporting, route compatibility checks and Europe-wide digital services.

Our approach is to generate the RINF export directly from the knowledge graph rather than maintaining it as a separate dataset. Rather than pre-computing ERA-ontology views, *all* mapping from the railML-centred graph to the ERA vocabulary [11] happens during SHACL-AF rule execution at export time: the mandatory RINF element types – tracks, signals, tunnels, bridges and train detection systems – and their required properties are derived by the rules and validated against ERA's official shapes before submission [1]. Coverage is still growing: at the time of writing roughly half of the mandatory RINF properties are mapped, with the remainder in active development.

The benefits are partly organisational. Generating RINF from the graph avoids duplicate maintenance: there is a single source of truth, and regulatory reporting becomes a *view* of the infrastructure model rather than a parallel dataset that can drift out of sync. Beyond that, performing the transformation in SHACL brings a documentation benefit: because the rules are declarative and are themselves part of the graph, they document the data lineage of the export far better than an opaque procedural exporter would (Section 4). The value is not limited to the export product, either: by adding the inferred ERA triples back into the DIM graph, data consumers of the underlying graph gain new perspectives and additional data, so the regulatory mapping doubles as an enrichment of the model for all users.

The mapping also exposed a genuine difficulty: RTM and ERA take fundamentally different approaches to asset topology locations. A lean RTM `LinearLocation` that simply references its associated net elements must be expanded into ERA's `NetLinearReference`, an ordered RDF-list structure whose generating SHACL rules run to many hundreds of lines for what is conceptually the same information. This conversion cost is a direct consequence of the two models having diverged (Section 2); a single, shared approach to topology and positioning across the industry would remove an entire class of such complicated conversions.

Notably, this approach enabled the first submission of RINF data at the micro (topology) level of detail [1], made possible precisely because the underlying graph is topology-based. This section demonstrates impact rather than novelty: it shows that a topology-centred KG can satisfy a real regulatory obligation as a by-product of good modelling.

6. GeoSPARQL and Railway Infrastructure: Lessons Learned

We consider this the most important section for the community. GeoSPARQL is the natural candidate for spatial reasoning in an RDF graph, and we adopted it expecting to perform our spatial operations inside SPARQL. In practice, a large share of the spatial and topological logic had to remain in SQL [27]. This section explains why.

6.1. Railway Spatial Requirements

Our pipeline needs the following spatial and topological operations:

- **Linear referencing** – converting between intrinsic coordinates, measures and absolute points along an edge (projecting an asset onto a track, or locating a measured position).
- **Kilometrage** – handling the discontinuous "km + offset" reference system.
- **Routing** – finding paths through the topology respecting navigability at switches.
- **Topology traversal** – following net relations to compute adjacency and reachability.
- **Subnet extraction** – selecting the subgraph for a line or operational area together with all assets positioned on it.

- **Coordinate reference system (CRS) transformation** – converting geometries between the coordinate reference systems used by the various source systems and by the published outputs.

6.2. What GeoSPARQL Supports Well

GeoSPARQL [3] is strong at what it was designed for. It offers a well-defined geometry model (WKT/GML literals), a comprehensive set of spatial predicates (topological relations such as `sfIntersects`, `sfWithin`, and distance/buffer functions) and proper coordinate reference system handling. For classic "does this asset fall within this region" or "how far apart are these two points" questions, it is entirely adequate – provided the spatial operations are implemented correctly by the SPARQL engine, which is not always a given – and we use it where geometry is genuinely the right abstraction.

6.3. What Required Workarounds

The requirements above, however, are not geometry questions – they are linear referencing, routing and topology questions, and GeoSPARQL has no native vocabulary or functions for them. We therefore fell back to SpatiaLite SQL, which does provide the necessary primitives (`ST_ClosestPoint`, `ST_Line_Locate_Point`, `ST_Azimuth`, `Line_Substring`, `ST_Reverse`, `ST_AddMeasure`, `ST_TrajectoryInterpolatePoint`). Concrete examples include:

- **Switch branch determination** – deciding whether a branch turns left or right by comparing the azimuths of the track segments meeting at a switch node (Listing 4). This requires extracting a short substring of each edge near the node and computing its bearing.
- **Line substring calculations** – cutting an edge between two measures to obtain the geometry of a linear location.
- **Route extraction** – assembling an ordered path through several edges.
- **Asset projection** – projecting an asset's absolute point onto the nearest track to obtain its measure.
- **Trajectory calculations** – interpolating a position along a measured trajectory.
- **Ordering of segments** – placing linearly connected segments in the correct sequence, so that a chain of edges reads as a continuous route rather than an unordered set.

Most striking of all, GeoSPARQL lacks even the most elementary accessors: there is no standard function to extract the X, Y or Z ordinate of a geometry (no `ST_X`, `ST_Y` or `ST_Z` equivalent). Obtaining a raw coordinate value – a trivial one-liner in any spatial SQL dialect – therefore already forces a detour outside the graph or a very ugly regex.

Listing 4: Switch branch determination in SpatiaLite: the direction of a branch is derived from the azimuth of a short substring of each track segment near the switch node – an operation with no GeoSPARQL equivalent.

```
SELECT node_oid,
  ST_Azimuth(
    ST_StartPoint(seg),
    ST_EndPoint(seg)) AS bearing
FROM (
  SELECT Line_Substring(
    centreline_geom,
    0,
    MIN(20 / ST_Length(centreline_geom), 1)
  ) AS seg, node_oid
  FROM track_segments
);
```

6.4. Consequence

The consequence is that a significant amount of logic that conceptually belongs to the graph had to remain in SpatiaLite SQL, executed *before* data reaches SPARQL Anything. This is not merely inconvenient: it splits the model across two paradigms, forces spatial pre-computation ahead of semantic mapping, and means that the graph consumer cannot ask topological or linear-referencing questions directly. We regard this as an important observation for the community: the gap is not in the expressiveness of RDF or SPARQL as such, but in the absence of standardised, well-implemented linear-referencing and topology functions in the geospatial layer of the stack.

Topology-based traversal in plain SPARQL is possible but painful. The subnet extraction queries initially relied on many nested `OPTIONAL` clauses with predicates bound to variables – effectively path-finding through the graph. We ultimately had to decompose them into fifteen sequential queries to make them tractable, an outcome that again reflects the absence of first-class topology support.

7. Technical Implementation and Performance

The pipeline of Section 3 deliberately hid the operational plumbing. We make it concrete here and report execution times, since reproducibility and performance are legitimate concerns for a production knowledge graph and were raised in review.

7.1. Deployment

The pipeline runs on a managed Kubernetes cluster (Azure Kubernetes Service) as part of the DIM platform [1]. Orchestration is provided by Apache Airflow 3.2, deployed as the usual set of components – API server, scheduler, DAG processor, triggerer and a Celery worker – backed by a PostgreSQL metadata database and a Redis (Valkey) broker, with all pipeline state held on a shared Azure Files volume mounted into every pod. The actual data processing runs in a single Celery worker pod, which is by far the largest workload: it is provisioned with 1–4 vCPU and 28–32 GiB of memory. The worker image bundles the full toolchain – a Java 21 runtime, Apache Jena 5.6 (TDB2 and Fuseki) [24], SPARQL Anything [4], SpatiaLite for the spatial pre-processing [27], and both SHACL engines, TopBraid SHACL [28] and maplib [29]. SPARQL Anything is run with a 16 GB Java heap; splitting and post-processing over the full graph are the main drivers of the worker's large memory envelope.

7.2. Orchestration and Reliability

The pipeline is decomposed into a sequence of Airflow DAGs – source ingestion, SPARQL Anything mapping, post-processing, splitting, railML generation and RINF genera-

Table 2
Per-stage wall-clock execution time of the production pipeline (Figure 2), measured on a single Celery worker pod (1–4 vCPU, 28–32 GiB) on AKS. The two RINF rows are *alternative* engines for the same step, not additive.

Pipeline stage	Time
Source ingestion (SpatiaLite SQL, CSV export)[a]	3.5 min
SPARQL Anything mapping (∼50 CONSTRUCT)	10 min
Post-processing (∼50 SPARQL UPDATE, Fuseki)	45 min
Splitting (15 sequential SPARQL queries)	1 h 30
railML generation (split subnetworks)	15 min
RINF generation – maplib SHACL	10 min
RINF generation – TopBraid SHACL	2 h

[a]Assumes a warmed-up OpenTNF input whose SpatiaLite spatial views are already precomputed.

tion – chained through Airflow's data-aware scheduling: each stage declares the dataset it produces, and the next stage is triggered automatically when that dataset is updated. All post-processing runs against a *staging* TDB2 dataset that is only hot-swapped behind the stable Fuseki endpoint once the stage has completed successfully (Section 3). Individual tasks are configured to retry up to three times with exponential backoff, so that transient failures – dominated by Azure Files I/O latency and occasional memory pressure during the splitting stage – are absorbed automatically, and the hot-swap guarantees that consumers never observe a partially processed graph even when a stage fails and is re-run.

7.3. Execution Times

Table 2 reports the per-stage wall-clock time on the hardware above, for the full production dataset (roughly 13 million triples, more than five thousand kilometres of track). Two observations stand out. First, the *splitting* stage (1 h 30) and *post-processing* (45 min) dominate the end-to-end run; both are pure SPARQL over the whole graph in Fuseki, and the former is precisely the fifteen sequential topology-traversal queries whose cost we attribute to the absence of first-class topology support (Section 6). Second, the choice of SHACL engine for the RINF export matters by an order of magnitude: the same rule set that maplib executes in about 10 minutes takes TopBraid SHACL roughly 2 hours, which quantifies the qualitative claim of Section 8. Excluding the slower TopBraid alternative, an end-to-end run completes in under three hours.

8. Discussion

We synthesise the lessons per technology.

8.1. SPARQL Anything

Strengths. A low entry barrier for a team new to RDF; rapid development; and, crucially, the direct use of SPARQL for both mapping and querying, avoiding a second mapping language. For file-based, asset-by-asset mapping it is highly effective, and Facade-X [5] provides a uniform lens over heterogeneous formats.

Weaknesses. It operates on files, not databases, which forced us to build the Source Ingestor to export CSV. Cross-file relationships are hard to express with nested SERVICE clauses – though this weakness is largely overcome by using SPARQL Anything (and its Facade-X metamodel) purely to triplify the sources into a faithful one-eyed graph first, and then expressing the cross-file semantic transformation as SHACL-AF rules over the assembled graph rather than inside the mapping queries. Left unchecked, the large mapping surface (∼40 asset types plus ∼50 post-processing queries) otherwise becomes an imperative program that is difficult to maintain, order and reason about.

8.2. SHACL

Strengths. Declarative mappings via rules; validation and transformation in one language; better maintainability than ordered SPARQL scripts; natural lineage and impact analysis; and the potential to drive API generation and URI dereferencing from the same shapes. In our RINF export, SHACL already both derives and validates properties [1].

Weaknesses. Tooling maturity – though this has improved markedly. At least two engines can execute SHACL rules at production scale: TopBraid SHACL [28] and maplib [29]. We adopted the latter: on our workload it is exponentially faster than TopBraid and employs a strong dependency-graph algorithm to order rule execution automatically. What is still missing is first-class data-lineage, geosparql with linear referencing and impact-analysis support, together with richer validation tooling that lets us inspect and manage results – summaries by type, trends over time, and links to work items – rather than merely producing a conformance report.

8.3. GeoSPARQL

Main finding. Current GeoSPARQL implementations are *geometry-centric*, whereas railway infrastructure modelling is largely *topology-centric* and *linear-reference-centric*. GeoSPARQL handles geometries, spatial predicates and coordinate reference systems relatively well, but it offers no native support for linear referencing, kilometrage, routing or topology traversal. As a result, spatial logic that ought to live in the graph was pushed down into SpatiaLite SQL. We believe this is the crux of the mismatch between the Semantic Web stack and the railway domain, and the strongest conclusion we can offer.

9. Future Work

Our lessons point to a clearer target architecture and a concrete wish list for the community.

Facade-X + SHACL-AF rules. We intend to fully adopt the two-stage approach: SPARQL Anything ingests a raw, faithful "one-eyed" graph, and SHACL rules perform the semantic mapping to railML and ERA views, with lineage recorded throughout. Figure 3 contrasts this with today's pipeline.

SHACL-first transformations. Replacing the ∼50 ordered post-processing queries with declarative SHACL rules, supported by a performant rules engine and lineage/impact-analysis tooling.

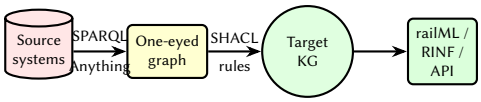

Figure 3: Target architecture: faithful ingestion into a one-eyed graph, followed by declarative SHACL-based semantic mapping with lineage. Spatial pre-processing and ad-hoc post-processing queries are eliminated once the tooling gaps are closed.

Linear referencing support in GeoSPARQL. The single change that would most simplify our work is native linear referencing [19] in GeoSPARQL, properly implemented in a triple store such as Jena [24] or an RDF library such as maplib [29]. This would let us move asset projection, line-substring and measure conversions from SQL pre-processing into SPARQL.

Topology-aware query functions. Complementary to linear referencing, standardised functions for routing, topology traversal and subnet extraction would make the topology a first-class citizen of the query layer, removing the need to decompose traversals into long sequential query chains.

If these gaps were closed, the split-paradigm pipeline of Section 3 could collapse into the clean architecture of Figure 3. The DIM portal [1] serves as a lighthouse for this direction: it already demonstrates infrastructure data consumed directly from the triple store, and the first micro-level RINF submission shows that a topology-based knowledge graph can meet real regulatory and interoperability needs today.

Acknowledgments

We thank the Bane NOR Digital Infrastructure Model (DIM) team and the wider railway and Semantic Web communities for the discussions that shaped this experience report.

Declaration on Generative AI

During the preparation of this work, the author(s) used generative AI tools for grammar and spelling checking and for improving the readability of the text. After using these tools, the author(s) reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] Bane NOR, DIM – digital infrastructure model web portal, https://dim.apps.banenor.no/, 2026. Accessed: 2026-07-15.

[2] M. Vanden Auweele, A vision on "future infrastructure": Location, time and identity in the digital infrastructure model, Position paper, Matdata / Bane NOR, 2026. Unpublished.

[3] Open Geospatial Consortium, OGC GeoSPARQL – A Geographic Query Language for RDF Data, Version 1.1, Technical Report OGC 22-047r1, Open Geospatial Consortium, 2024. URL: https://docs.ogc.org/is/22-047r1/22-047r1.html.

[4] L. Asprino, E. Daga, J. Dowdy, P. Mulholland, M. Ridolfi, SPARQL Anything: A system for a-posteriori integration of data using SPARQL, https://sparql-anything.cc/, 2023. Accessed: 2026-07-15.

[5] E. Daga, L. Asprino, P. Mulholland, A. Gangemi, Facade-x: An opinionated approach to SPARQL anything, in: Further with Knowledge Graphs (SEMANTiCS 2021), volume 53 of *Studies on the Semantic Web*, IOS Press, 2021, pp. 58–73. doi:10.3233/SSW210035.

[6] H. Knublauch, D. Kontokostas, Shapes Constraint Language (SHACL), W3C Recommendation, W3C, 2017. URL: https://www.w3.org/TR/shacl/.

[7] H. Knublauch, SHACL Advanced Features (SHACL-AF), W3C Working Group Note, W3C, 2017. URL: https://www.w3.org/TR/shacl-af/.

[8] UIC, railML.org, RailTopoModel (IRS 30100) – railway infrastructure topological model, https://www.railtopomodel.org/, 2016. Accessed: 2026-07-15.

[9] railML.org, railML 3 – the railway markup language, https://www.railml.org/, 2024. Accessed: 2026-07-15.

[10] UIC, RSM – rail system model, version 1.2, https://rsm.uic.org/, 2023. Accessed: 2026-07-15.

[11] European Union Agency for Railways, The ERA vocabulary – ontology for the Register of Infrastructure (RINF), https://data-interop.era.europa.eu/era-vocabulary/, 2024. Accessed: 2026-07-15.

[12] European Union Agency for Railways, Register of infrastructure (RINF), https://rinf.era.europa.eu/, 2024. Accessed: 2026-07-15.

[13] Bane NOR, OpenTNF – open topology network format, https://github.com/OpenTNF/OpenTNF, 2024. Accessed: 2026-07-15.

[14] Open Geospatial Consortium, OGC GeoPackage Encoding Standard, Version 1.4, Technical Report, Open Geospatial Consortium, 2024. URL: https://www.geopackage.org/.

[15] Hacon, TPS – train planning system, https://www.hacon.de/, 2024. Accessed: 2026-07-15.

[16] TrafIT Solutions, railOscope, https://www.trafit.ch/, 2024. Accessed: 2026-07-15.

[17] Infrabel, InfraNET and InfraGIS – infrastructure asset and geographic information systems, https://www.infrabel.be/, 2024. Accessed: 2026-07-15.

[18] OpenStreetMap contributors, OpenStreetMap railway data, https://www.openstreetmap.org/, 2024. Accessed: 2026-07-15.

[19] ISO, ISO 19148:2021 geographic information – linear referencing, International Organization for Standardization (2021). URL: https://www.iso.org/standard/75150.html.

[20] railML.org, The railML ontology, https://ontology.railml.org/, 2024. Accessed: 2026-07-15.

[21] railML.org, railML use cases, https://www.railml.org/en/usecases, 2024. Accessed: 2026-07-15.

[22] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF mapping language, W3C Recommendation (2012). URL: https://www.w3.org/TR/r2rml/.

[23] A. Dimou, M. V. Sande, P. Colpaert, R. Verborgh, E. Mannens, R. V. de Walle, RML: A generic language for integrated RDF mappings of heterogeneous data, in: Proceedings of the 7th Workshop on Linked Data on the Web (LDOW), 2014.

[24] Apache Software Foundation, Apache jena and fuseki, https://jena.apache.org/, 2024. Accessed: 2026-07-15.

[25] W3C Facade-X Community Group, The one-eyed graph: Facade-x as a faithful triplification of source structure, https://github.com/w3c-facade-x/meeti

ngs/tree/main/meetings/2026-01-12, 2026. Accessed: 2026-07-15.

[26] European Commission, Commission implementing regulation (EU) 2019/777 on the common specifications for the register of railway infrastructure (RINF), https://eur-lex.europa.eu/eli/reg_impl/2019/777, 2019. Accessed: 2026-07-15.

[27] A. Furieri, SpatiaLite – a spatial extension to SQLite, https://www.gaia-gis.it/fossil/libspatialite/, 2024. Accessed: 2026-07-15.

[28] TopQuadrant, TopBraid SHACL API, https://github.com/TopQuadrant/shacl, 2024. Accessed: 2026-07-15.

[29] DataTreehouse, maplib: A knowledge graph library with SHACL rules and validation, https://github.com/DataTreehouse/maplib, 2024. Accessed: 2026-07-15.