Semantic Data Pipeline: A Zero-Install Browser Workbench for Visual Ontology Editing and Event-to-Graph Materialisation

Patrik Kompuš[1]

[1]*Prague University of Economics and Business, nám. W. Churchilla 1938/4, Prague, 130 67, Czech Republic*

Abstract

Ontology authoring and data integration are served by separate toolchains: desktop or server-hosted editors on one side, declarative mapping languages executed by command-line processors on the other. Practitioners who must do both pay for the gap in installation overhead, context switching, and hand-written mapping files that are only validated when a processor finally runs. We demonstrate *Semantic Data Pipeline*, a browser-resident workbench that closes the loop. It explores an OWL vocabulary as an interactive graph, edits that vocabulary in place with change review at triple granularity, binds event-payload fields to ontology traversal paths through an interface that offers only the hops the vocabulary actually declares, and materialises incoming payloads as ontology-conformant RDF written back to the store. The same interface runs over a remote SPARQL 1.1 endpoint or over an in-browser WebAssembly triplestore that needs no server at all, so the entire pipeline is demonstrable from a single browser tab.

Keywords

ontology engineering, visual ontology editing, knowledge graph construction, RDF mapping, SPARQL

1. Introduction

Building a knowledge graph from operational data requires two distinct activities. The first is *vocabulary work*: deciding which classes and properties exist, how they specialise one another, and what their domains and ranges are. The second is *mapping work*: deciding which field of which incoming record becomes which triple. In practice these are supported by disjoint tooling. Ontology editors such as Protégé [1] and its web successor WebProtégé [2] are excellent at the first and stop at the vocabulary boundary. Mapping languages such as R2RML [3], RML [4] and SPARQL-Generate [5] are expressive and principled at the second, but are text artefacts authored outside the editor and executed out of band.

The seam between them is where errors accumulate. A mapping document names classes and properties by IRI, and nothing checks those IRIs against the vocabulary while the mapping is written. A renamed property, or a path the ontology never declared, produces a rule that looks plausible and fails — or worse, silently produces nothing — only once a processor runs over real data. Validating the *output* with SHACL [6] catches the symptom after the fact; it does not stop the malformed rule from being written.

We demonstrate *Semantic Data Pipeline* (SDP), a single-page web application in which vocabulary work and mapping work happen against the same live graph, in the same tab, with no installation. Its contributions are:

C1 — Storage-agnostic, zero-install operation. Every module addresses storage through one narrow interface with `query` and `update` operations. Two implementations are interchangeable at runtime: a remote SPARQL 1.1 endpoint, and a triplestore compiled to WebAssembly that runs inside the page. The second needs no server, which is what makes the whole pipeline demonstrable from a browser alone.

The 25th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2026) - Satellite Events (Workshops, Tutorials, Demos, Posters).

qkomp00@vse.cz (P. Kompuš)
https://www.vse.cz/ (P. Kompuš)
0009-0001-7816-2179 (P. Kompuš)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-26

CEUR-WS.org/Vol-4340/paper14.pdf

C2 — In-place editing with triple-level review. The displayed graph becomes an editable canvas. Because the display projection is deliberately lossy, edits are diffed as *triples* rather than as diagram elements. Every added and removed triple is presented for individual approval before commit, deletion never cascades, and in a multi-ontology workspace only the ontology being edited is writable.

C3 — Mapping targets that cannot be malformed. A rule's target is built by guided traversal: from a start class, the interface offers exactly the object properties the ontology declares for that class and its ancestors, and terminates on a datatype property reachable the same way. A structurally unsatisfiable target is therefore not expressible. Materialisation assigns individuals stable IRIs derived from a designated key field, so repeated payloads about the same entity converge on one individual instead of accumulating duplicates.

2. Related Work

Editors and management platforms. Protégé [1] remains the reference desktop environment, and WebProtégé [2] brought collaborative authoring to the browser. Both are considerably more expressive than SDP as *editors* — they cover the OWL 2 language [7] and integrate reasoners — but neither carries the vocabulary forward into data materialisation, which is the step SDP exists to connect. Enterprise platforms such as metaphactory [8] and SciBite CENtree [9] supply the governance layer — versioning, release management, multi-user curation — that SDP only gestures at with per-ontology write locking, and are correspondingly heavier to deploy than a single browser tab.

Ontology visualisation. VOWL [10] defines a user-oriented visual notation for OWL, realised for the web in WebVOWL [11]; both target comprehension and are read-only. Chowlk [12] inverts the direction, compiling UML-style diagrams into OWL, but as a one-way batch conversion rather than live editing against a connected store.

Mapping languages. R2RML [3] standardised relational-to-RDF mapping; RML [4] generalised it to heterogeneous sources; SPARQL-Generate [5] extends SPARQL itself to emit RDF from arbitrary formats. All three are strictly more expressive than SDP's rule model, which has no transformation functions, no conditionals and no joins. SDP trades that expressivity for a property they cannot offer: a rule constructed by walking the loaded vocabulary is well-formed with respect to it by construction, and the loop is interactive rather than edit-run-inspect.

3. Architecture

SDP is a single-page TypeScript and React application of roughly 12 000 lines, with a further 2 000 of unit tests. All state — triples, edit sessions, event schemas, mapping rules — lives in the browser; there is no application server, only an optional triplestore.

Storage abstraction. Figure 1 shows the layering. The remote backend speaks SPARQL 1.1 Query and Update [13] over HTTP against a configurable namespace of a Blazegraph server, selectable at runtime from those that server exposes. The in-page backend is Oxigraph compiled to WebAssembly; at startup it ingests the files named in a manifest, with the serialisation inferred from the extension (RDF/XML, Turtle, N-Triples, N-Quads, TriG). Because both satisfy the same interface, the module layer is unaware of which is active, and a session developed against the in-page store can be pointed at a server unchanged.

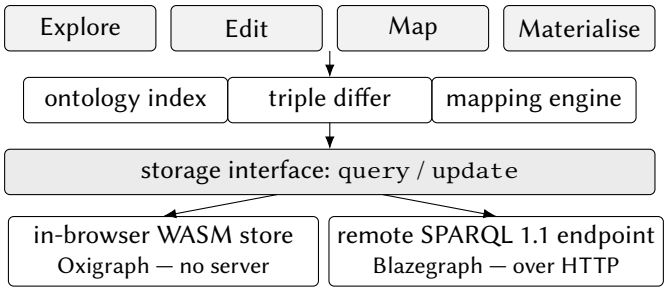


Figure 1: Modules address storage through one interface, so the two backends are interchangeable at runtime. Everything above the interface, and the WebAssembly store itself, runs inside the browser tab.

Multi-ontology workspace. The demonstration workspace bundles six vocabularies — core, contact, currency, organisation, person and wealth — opening on 83 classes, 49 object properties and 47 datatype properties. Each IRI is attributed to an ontology by longest-prefix match over the known ontology URIs. That one rule drives both rendering (per-ontology colouring and filtering) and permission: during an edit session, terms belonging to a non-active ontology are shown for context but locked against modification, so a session cannot silently rewrite a vocabulary it did not open.

Rendering. The graph is drawn with Cytoscape.js [14]. Projecting triples onto it is lossy on purpose — annotations and datatype properties are folded onto their subject node to keep the diagram legible — which is precisely why editing is not modelled at the diagram level.

4. Demonstration Walkthrough

The demonstration follows one path end to end: explore a vocabulary, extend it, bind an event field to it, and materialise a payload. A live instance runs at https://ontology-browser-zeta.vercel.app/, and the same walkthrough is built into the tool: its *Learning* view (https://ontology-browser-zeta.vercel.app/learning) carries a short lesson path ending in an annotated frame-by-frame clip of the sequence below, captured from that instance.

Step 1 — Explore. The workspace loads and renders as a graph, filterable by source ontology and by term category. Selecting a node reveals its labels, comments, superclasses, and declared properties.

Step 2 — Edit in place. Toggling edit mode turns the displayed graph into a canvas: classes can be added, renamed and deleted, object properties drawn between classes, datatype properties attached, and `subClassOf` links redrawn. On commit, the canvas is serialised to triples and diffed against the baseline serialised the same way. Diffing two serialisations of the same fragment — rather than against the raw store — guarantees the change set is closed over the constructs the editor understands, so a triple the editor does not model cannot be destroyed by a round-trip. It recognises `owl:Class`, `owl:ObjectProperty` and `owl:DatatypeProperty` declarations together with `rdfs:label`, `comment`, `subClassOf`, `domain` and `range`; everything else is passed over untouched.

The resulting change set is presented as a list of added and removed triples, each of which the user marks as approved, left pending for a later round, or discarded for the session. Approved triples become a single SPARQL update pairing `DELETE DATA` with `INSERT DATA`. Deletion removes only a term's own triples and never cascades: if a surviving class still names a deleted parent, or a surviving property still points at a deleted range, the commit reports a dangling-reference warning rather than refusing the edit or quietly repairing the graph.

Step 3 — Bind an event field to the vocabulary. A mapping rule joins a leaf field of an event schema — a JSON Schema document held in a library the workbench maintains — to a path through the

ontology. The path starts at a class, takes zero or more object-property hops, and ends on a datatype property; at each step the interface enumerates only what the vocabulary declares for the current class and its ancestors, with the option to narrow a hop to a subclass of the declared range. Listing 1 shows a stored rule. The rule set persists in the browser and round-trips as a single RDF/XML document, so it can be versioned alongside the ontologies it targets.

```
{ "eventName":  "UserCreated",
  "sourcePath": ["content", "internalUserId"],
  "rootClass":  "ex:Person",
  "hops":       [ { "viaProperty": "ex:hasIdentifier",
                    "toClass": "ex:Identifier" } ],
  "datatypeProperty": "ex:identifierValue" }
```

Listing 1: A mapping rule: an event field bound to a two-step ontology path.

Step 4 — Materialise a payload. A payload is pasted in and matched against the rules for its event type. Rules are grouped by start class, so one individual per class is produced per payload. Its IRI is derived from a user-designated key field, and each hop extends its parent's IRI with the traversed property, so the addresses in Listing 2 are stable: re-processing the same entity updates it in place rather than creating a second individual. Values are typed from the target property's declared range, and a property designated as holding a credential is hashed before emission so no plaintext secret reaches the store.

```
@prefix ex:   <http://example.org/core#> .
@prefix data: <http://example.org/data/> .

data:Person-u-4711  a  ex:Person ;
    ex:hasIdentifier  data:Person-u-4711-hasIdentifier .
data:Person-u-4711-hasIdentifier  a  ex:Identifier ;
    ex:identifierValue  "u-4711" .
```

Listing 2: Triples materialised from the rule in Listing 1 and a payload whose `content.internalUserId` is `u-4711`.

Nothing is written until the user has seen it. The result is previewed as Turtle alongside a summary of how many triples were produced, which rules applied, and — importantly for debugging a rule set — which rules were skipped and why, distinguishing an absent key field from an absent source field. Confirming writes the triples through the same storage interface as an edit commit, closing the loop.

5. Discussion and Limitations

SDP is deliberately a narrow tool, and its limits are worth stating plainly. The editable fragment of OWL is small: classes, object and datatype properties, subsumption, domain, range, labels and comments. Class expressions, cardinality and value restrictions, equivalence and disjointness are neither authored nor displayed; triples expressing them survive editing untouched, which makes the tool safe to point at a rich ontology, but not the only editor such an ontology ever sees. No reasoner is integrated: inherited properties are computed by walking asserted subsumption, and nothing detects an inconsistency the user introduces.

The rule model is likewise restricted. A target path must terminate on a datatype property; there are no transformation functions beyond credential hashing, no conditionals, and no joins across payloads. Rules guarantee only *structural* conformance — that every class and property named exists and the path is traversable — not that the resulting individuals satisfy any semantic constraint. Editing is single-writer with last-write-wins semantics, with no locking across concurrent sessions.

Finally, we report no empirical evaluation: there has been no user study, and performance has not been measured beyond the bundled workspace of a few thousand triples, where interaction is immediate. Whether the guided-traversal interface remains usable at the scale of a large vocabulary is an open question, and the obvious first thing to measure.

Future work follows from these gaps: a reasoner and SHACL validation, a wider editable OWL fragment, exporting rules as RML so they can run on established engines, and a controlled study of rule-authoring accuracy against hand-written mappings.

A synthetic data generator is the piece we would build first. The workbench already knows a vocabulary's shape — its classes, their subsumption, the domain and range of every property — so it can generate individuals that conform to it. Populating a store on demand would let a user exercise an ontology against realistic volume instead of a handful of hand-written examples, and would supply the data the scale question above cannot be answered without.

6. Conclusion

Semantic Data Pipeline shows that exploring a vocabulary, editing it, binding data fields to it, and materialising records against it can be one continuous activity in one browser tab, with the ontology as the interface at every step rather than a document consulted between them. Constraining the mapping model to paths the vocabulary declares buys the property we care about: a rule that is well-formed when written, not when finally run.

Declaration on Generative AI

During the preparation of this work, the author(s) used Claude (Anthropic) in order to: draft content, improve writing style, and assist with the literature search. After using this tool, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] M. A. Musen, The protégé project: a look back and a look forward, AI Matters 1 (2015) 4–12. doi:10.1145/2757001.2757003.

[2] T. Tudorache, C. Nyulas, N. F. Noy, M. A. Musen, WebProtégé: A collaborative ontology editor and knowledge acquisition tool for the Web, Semantic Web 4 (2013) 89–99. doi:10.3233/SW-2012-0057.

[3] S. Das, S. Sundara, R. Cyganiak, R2RML: RDB to RDF mapping language, W3C Recommendation, 2012. URL: https://www.w3.org/TR/r2rml/, 27 September 2012.

[4] 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), volume 1184 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2014. URL: https://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[5] M. Lefrançois, A. Zimmermann, N. Bakerally, A SPARQL extension for generating RDF from heterogeneous formats, in: The Semantic Web — ESWC 2017, volume 10249 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 35–50. doi:10.1007/978-3-319-58068-5_3.

[6] H. Knublauch, D. Kontokostas, Shapes constraint language (SHACL), W3C Recommendation, 2017. URL: https://www.w3.org/TR/shacl/, 20 July 2017.

[7] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, S. Rudolph, OWL 2 web ontology language primer (second edition), W3C Recommendation, 2012. URL: https://www.w3.org/TR/owl2-primer/, 11 December 2012.

[8] P. Haase, D. M. Herzig, A. Kozlov, A. Nikolov, J. Trame, metaphactory: A platform for knowledge graph management, Semantic Web 10 (2019) 1109–1125. doi:10.3233/SW-190360.

[9] SciBite, CENtree: Ontology management, Product documentation, ???? URL: https://scibite.com/solutions/ontology-management/centree/, accessed 2026-07-25.

[10] S. Lohmann, S. Negru, F. Haag, T. Ertl, Visualizing ontologies with VOWL, Semantic Web 7 (2016) 399–419. doi:10.3233/SW-150200.

[11] S. Lohmann, V. Link, E. Marbach, S. Negru, WebVOWL: Web-based visualization of ontologies, in: Knowledge Engineering and Knowledge Management: EKAW 2014 Satellite Events, volume 8982 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 154–158. doi:10.1007/978-3-319-17966-7_21.

[12] S. Chávez-Feria, R. García-Castro, M. Poveda-Villalón, Chowlk: from UML-based ontology conceptualizations to OWL, in: The Semantic Web — ESWC 2022, volume 13261 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 338–352. doi:10.1007/978-3-031-06981-9_20.

[13] P. Gearon, A. Passant, A. Polleres, SPARQL 1.1 update, W3C Recommendation, 2013. URL: https://www.w3.org/TR/sparql11-update/, 21 March 2013.

[14] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, G. D. Bader, Cytoscape.js: a graph theory library for visualisation and analysis, Bioinformatics 32 (2016) 309–311. doi:10.1093/bioinformatics/btv557.