The Eyeling Playground: N3 rules in your browser

Patrick Hochstenbach[1,2,*], Jos De Roo[1], Wout Slabbinck[1], Beatriz Esteves[1] and Pieter Colpaert[1]

[1]Ghent University, IMEC-IDLab, Ghent, Belgium
[2]Ghent University, Ghent University Library, Ghent, Belgium

Abstract

Semantic Web applications increasingly run in client-side, interactive, and data-sovereign environments, where users need to evaluate rules close to the data rather than delegating every reasoning task to a server-side infrastructure. Existing Notation3 reasoners demonstrate the expressivity of rule-based reasoning on the Web, but they are often implemented for native or server-side execution environments, making browser deployment, optimization, integration, and interactive explanation more difficult. We present Eyeling, a JavaScript-based Notation3 reasoner designed from the ground up for execution in the browser, and demonstrate how it brings rule-based reasoning to Web applications through use cases such as ODRL access control policies. The accompanying Eyeling Playground allows users to select examples, load data and rules, execute reasoning locally, inspect derived facts, and explore proof-oriented explanations right from a browser.

Keywords

Notation3, RDF, Rules, Browser-based reasoning, ODRL, Access control, Neurosymbolic AI

1. Introduction

Semantic Web applications increasingly need lightweight reasoning where data is consumed. A Web client may need to decide whether a user is allowed to perform an action, explain why a policy grants or denies access, validate a local context, or combine user state with remote machine-readable knowledge. Such tasks are often small enough to run near the user interface, but expressive enough that ad-hoc JavaScript conditionals become difficult to inspect, share, and reuse.

This need becomes more urgent in emerging data and agent spaces [1], where data, services, policies, credentials, tools, and AI agents are combined across organizational boundaries. In such settings, trust cannot rely only on a remote platform or on a probabilistic agent's answer. Applications need explicit, inspectable rules that say how heterogeneous terms should be aligned, which assertions may be relied on in a given context, and which actions are permitted under the applicable policies. Semantic rules are a natural mechanism for this: they can express mappings and constraints between vocabularies, derive context-specific consequences, and provide a deterministic boundary.

Usage control is a representative case because data and services cannot be treated as legally or operationally reusable merely because that are technically possible. Whether open, shared, or protected, they should be accompanied by machine-interpretable usage conditions that state under which purposes, parties, actions, duties, temporal constraints, or contextual conditions they may be used. Vocabularies such as the Open Digital Rights Language (ODRL) [2, 3] describe permissions, prohibitions, duties, constraints, assets, parties, and actions as RDF. However, describing such conditions is only the first step: an application must still evaluate whether a concrete request complies with them. In practice, this requires combining policy data, request context, resource metadata, state-of-the-world information, and domain-specific rules [4]. Running such checks in the browser allows the decision to be made close to the interaction it governs, while keeping the rules and facts that justify the decision available for inspection. In an independent project [5], we evaluated N3 compliance against the latest W3C

SEMANTiCS 2026, September 15–17, 2026, Ghent, BE
*Corresponding author: Patrick Hochstenbach, Patrick.Hochstenbach@UGent.be.
Patrick.Hochstenbach@UGent.be (P. Hochstenbach)
0000-0001-8390-6171 (P. Hochstenbach); 0000-0001-8862-0666 (J. D. Roo); 0000-0002-3287-7312 (W. Slabbinck); 0000-0003-0259-7560 (B. Esteves); 0000-0001-6917-2167 (P. Colpaert)

```
@prefix dpv:      <https://w3id.org/dpv#> .
@prefix dpv-odrl: <https://w3id.org/dpv/mappings/odrl#> .
@prefix odrl:     <http://www.w3.org/ns/odrl/2/> .
@prefix ex:       <https://example.org/> .
@prefix eye:      <https://eyereasoner.github.io/eye/reasoning#> .
@prefix math:     <http://www.w3.org/2000/10/swap/math#> .
@prefix time:     <https://www.w3.org/TR/owl-time/> .
@prefix sotw:     <https://w3id.org/force/sotw#> .
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#> .

# Policy, request, and requester data
ex:policy a odrl:Policy ; odrl:uid ex:policy ; odrl:profile dpv-odrl: ; odrl:permission [
    odrl:target ex:d; odrl:action odrl:use; odrl:constraint [
    odrl:leftOperand dpv-odrl:Purpose; odrl:operator odrl:eq; odrl:rightOperand dpv:AcademicResearch ] , [
    odrl:leftOperand odrl:dateTime; odrl:operator odrl:lt;
    odrl:rightOperand "2027-01-01T00:00:00Z"^^xsd:dateTimeStamp ] ] .
ex:request a sotw:EvaluationRequest; sotw:evaluatedTarget ex:d; sotw:evaluatedAction odrl:use;
    sotw:evaluatedParty ex:alice ; sotw:requestParameter [ a sotw:RequestParameter ;
    sotw:value dpv:AcademicResearch ; sotw:describesFeature sotw:Purpose ] .
ex:sotw a sotw:SotW ; sotw:context [ a time:Instant ;
    time:inXSDDateTimeStamp "2026-05-30T00:00:00Z"^^xsd:dateTimeStamp  ] .

# Policy interpretation and output
{ ?p odrl:permission [ odrl:target ?d; odrl:action ?a;
    odrl:constraint [ odrl:leftOperand dpv-odrl:Purpose; odrl:operator odrl:eq ; odrl:rightOperand ?purpose ],
    [ odrl:leftOperand odrl:dateTime; odrl:operator odrl:lt; odrl:rightOperand ?pt]] .
  ?r sotw:evaluatedTarget ?d; sotw:evaluatedAction ?a; sotw:evaluatedParty _:b1 ;
    sotw:requestParameter ?rp .
  ?rp sotw:value ?purpose .
  ?s a sotw:SotW ; sotw:context ?c1 .
  ?c1 a time:Instant ; time:inXSDDateTimeStamp ?t .
  ?t math:lessThan ?pt . }
=> { ?r ex:decision ex:Permit .
     eye:result eye:output "Permit: purpose and time constraint satisfied." . } .
```

Figure 1: A compact N3 example illustrating an ODRL policy in which data (`ex:d`) may only be used for Academic Research before 2027, together with a request evaluation by Alice and a State of the World indicating the current time, an interpretation rule, and a decision-oriented reasoning output.

Community Group specification [6]. Eyeling is the first next-generation N3 reasoner to achieve full compliance. Although no standardized performance benchmarks currently exist, Eyeling performs on par with the EYE reasoner [7], which has been used in industrial settings for nearly 25 years.

Eyeling was built for this setting. It is a compact Notation3 (N3) reasoner implemented in JavaScript. In N3, RDF-style facts, quoted formulae, graph-to-graph rules, checks, and output statements can be kept in one artifact. An Eyeling program can therefore contain the policy, the request, the rules that interpret the policy, and statements that render a decision or explanation. Operationally, Eyeling parses N3 facts and rules, performs forward saturation, and uses a backward prover for rule bodies and built-ins. This blend works well for among others, policy-style examples, as illustrated in fig. 1: forward rules derive decisions, while backward reasoning and built-ins can satisfy rule bodies without materializing every intermediate fact.

The accompanying Eyeling Playground demonstrates this browser-native workflow. Users can select or load examples, edit local N3, combine it with background knowledge, run reasoning locally, inspect streamed results, enable proof comments, and share a complete configuration as a URL.

2. Related work

Rule-based reasoning for RDF and knowledge graphs has a long history, including N3 reasoners, RDF rule engines, Datalog systems, SPARQL engines, and RDF stream processors. These systems differ in expressivity, performance, language design, data-access model, and deployment model. A recently closely related system is **Nemo** [8]: a Datalog-oriented rule engine for RDF-compatible data. Nemo supports datatypes, value invention, negation, aggregation, RDF and CSV import/export, and

optimized access to SPARQL endpoints. Nemo Web executes reasoning locally in the browser through a WebAssembly and TypeScript build, and the Nemo Explain Visualizer provides interactive proof-tree explanations for derived results.

Kolibrie [9] then again is a Rust-based SPARQL database, RDF Stream Processing engine, and RDF toolkit with support for RDF parsing, Turtle/N3 input, SPARQL querying, stream/window operations, and rule-based inference. Its playground provides a browser interface for experimenting with RDF data, SPARQL queries, and rules, and uses WebAssembly to expose the Rust engine client-side. Kolibrie is thus close in deployment setting, but differs in focus: it primarily targets SPARQL-oriented querying, RDF stream processing, and Datalog-style rule evaluation rather than N3-native rule execution as an embeddable JavaScript library.

Comunica [10] provides a complementary JavaScript-native point in the design space. It is a modular SPARQL query framework for heterogeneous Web sources, with browser, command-line, server, and JavaScript application deployment modes, and strong RDF/JS interoperability. It provides a playground that works fully client-side[1]. A recent Comunica-based LTQP demo extends this setting with online schema alignment: mapping rules are discovered during traversal, scoped to subwebs, and applied by forward chaining so that aligned triples can contribute to traversal and join processing [11]. This illustrates how rule-like processing can be embedded in Web querying workflows, but its focus remains SPARQL link traversal and schema alignment rather than general N3 rule execution.

Whelk [12] is a relevant example of browser-capable Semantic Web reasoning outside the N3 rule-reasoning setting. It is an OWL EL+RL reasoner implemented in Scala, with support for OWL profile reasoning, A-box materialization, incremental reasoning, parallel query answering, and SWRL rules. Through Scala.js, Whelk can be used in browser-based JavaScript applications, and the authors provide an interactive Web version. Whelk therefore shows that ontology reasoning can be moved into client-side environments, but its focus is OWL profile reasoning rather than N3 rules, quoted formulae, Web-oriented built-ins, or embedding N3 rule execution as a JavaScript-native library.

One way to bring reasoners to browsers is to compile an existing reasoner or query engine to WebAssembly. This was used for **eye-js**, which brought the EYE reasoner [7] written in Prolog to JavaScript environments [13]; Nemo Web follows a similar route for the Nemo Datalog engine [8]; and Kolibrie's playground uses WebAssembly for its Rust-based RDF/SPARQL engine [9]. This approach reuses mature engines, but also inherits assumptions from the original runtime, such as memory layout, input/output conventions, build tooling, and extension mechanisms. Eyeling takes another route: it is implemented directly in JavaScript, exposes APIs such as `reasonStream(...)` and `reasonRdfJs(...)`, integrates with RDF/JS data structures, supports JavaScript-level custom built-ins, and can run in a Web Worker without adapting a foreign runtime model.

3. Eyeling

Eyeling is an LLM-driven synthesized N3 reasoner produced by the project's reasoning team using *GPT-5 Extended Thinking* (using versions 5.1, 5.2, 5.4 and 5.5) throughout the course of the project. The inputs used to develop the Eyeling reasoner comprised the three N3 specifications [6, 14, 15] and our `notation3tests` compliance test suite. Beyond compliance, the synthesis had to produce a mature implementation whose performance was on par with the EYE reasoner. This required thousands of prompts guided by human feedback. When the LLM was stuck, selected fragments of the EYE prolog source were provided as hints. Eyeling is therefore not a translation of EYE: the underlying V8 engines (Eyeling) and SWI-Prolog (EYE) differ in ways that materially affect implementation choices, clause indexing as the most consequential. As of April 29, 2026, this process had produced 294 chronological versions.[2]

Eyeling accepts one or more N3 documents as input, each parsed individually into an internal AST representation. A *Rule Normalization* step then applies blank node normalization to the AST, promoting

[1] https://query.comunica.dev

[2] This corresponds to approximately five months of human-assisted development time.

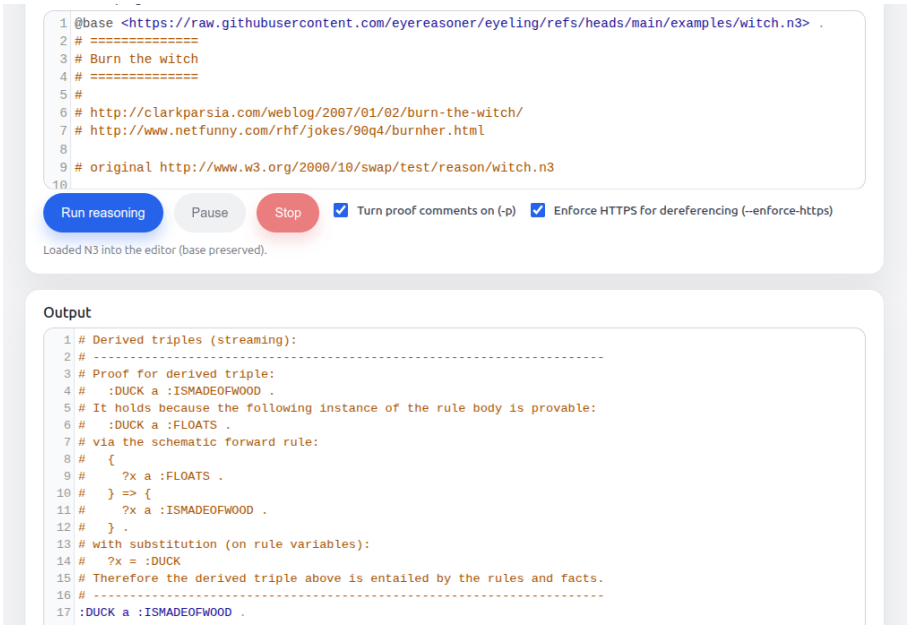

Figure 2: The Eyeling Playground lets developers edit N3 rules, execute them in the browser, inspect derived facts, and enable proof comments explaining why triples were derived.

blank nodes appearing in rule premises to universally quantified variables, as required by the N3 specification. This step also handles RDF collections, ensuring that both `rdf:first`/`rdf:rest` chains and native N3 list notation are treated uniformly throughout the reasoning pipeline. The normalized rules and initial facts are then passed to the reasoning engine, which iteratively applies forward and backward chaining until a fixpoint is reached. Derived solutions are emitted as N3 output in a streaming fashion.

Both forward and backward chaining are supported within a single N3 input document, in accordance with the N3Logic semantics. In addition, the synthesized reasoner implements complete coverage of the N3 Built-in Functions specification.

Despite its LLM-driven synthesis, Eyeling has no runtime dependencies on any language model: every execution consists entirely of deterministic symbolic reasoning steps. Once generated, Eyeling operates as a fully self-contained deterministic symbolic N3 reasoner. The implementation is written in pure JavaScript and has no external dependencies and no runtime requirements (e.g., no WebAssembly runtime). It can be executed as a command-line tool, consumed via an API, or embedded directly in a web browser (see the next section). Although implemented in JavaScript, the package ships with TypeScript type declarations, so it can be consumed as a fully typed API from TypeScript projects without additional configuration.

In addition to its score on the compliance suite, Eyeling was also the fastest of the four N3 implementations on the compliance-suite throughput. Completing the full suite took 2:25 (mm:ss) for Eyeling, compared to 4:42 for EYE, 9:15 for cwm, and 12:33 for jen3. All measurements were taken on a 2.3 GHz 8-Core Intel Core i9 MacBook Pro, macOS 26.3.1, with Node.js v22.14.0, SWI-Prolog 9.3.1 and Python version 2.7.18. These timings should be read with caution: the compliance suite is correctness-oriented and does not exercise the math-heavy or large-scale workloads that engines like EYE are typically used for.

4. The playground

The Eyeling Playground, available at https://eyereasoner.github.io/eyeling/playground, is the browser-based demonstrator. The playground was created in order to support teaching activities, debugging, live demonstrations, issue reports, and sharing reproducible examples.

At the center of the playground is an editable N3 program containing facts, rules, checks, requests, or reporting directives. N3 can also be loaded from a URL, either into the editor or as background knowledge. Background mode supports realistic demonstrations: a stable policy vocabulary, rule base, or dataset can remain fixed while the user edits only the local request, query, or explanation logic.

Reasoning runs locally in the browser and can be executed in a worker so the interface remains responsive. Results can be inspected as streamed derived triples or rendered as a textual report when the program derives `log:outputString` facts. The former is useful for debugging; the latter is useful for presenting a human-readable decision. Explainability is exposed through proof comments. When enabled, Eyeling annotates derived triples with compact justifications: the derived triple, the instantiated rule body that was provable, and the schematic rule that produced it. For policy examples, this lets users inspect not only whether a request was permitted or denied, but also which facts and rules led to that conclusion.

The playground also supports syntax-error reporting with line and column information, optional HTTPS dereferencing enforcement, and shareable state. A URL can encode the editor content, imported background source, proof-comment settings, and dereferencing settings. The examples this way can be shared in papers, tutorials, lectures, or issue reports. At SEMANTiCS, the demo will show this workflow by loading a small policy-oriented example, editing the request context, running the reasoner, enabling proof comments, switching to URL-loaded background knowledge, and sharing the final configuration as a URL[3].

5. Conclusion

Eyeling brings N3 reasoning into the browser as a JavaScript-native component for Semantic Web applications. Despite its current limitations (such as the maximum input data size, limited by the JavaScript `MAX_STRING_LENGTH` to 512 MB), Eyeling is useful for scenarios such as ODRL access-control policies, where RDF policy data, local request context, domain rules, and inspectable explanations need to be combined close to the user interaction. Furthermore, Eyeling supports a streaming mode that allows reasoning over RDF Messages [16] using a window of at most 512 MB. In the vision of *Agent Spaces* as introduced by Polleres et al. [1], such a rule engine could this way provide the symbolic policy guardrails for probabilistic AI systems before the data leaves the end-user device.

The Eyeling Playground demonstrates this direction through editable N3, URL-loaded background knowledge, local browser execution, streamed output, proof comments, and shareable configurations. Beyond the playground, Eyeling's main contribution is an N3-native integration surface for client-side Web applications, including RDF/JS interoperability, Web Worker execution, custom built-ins, and JavaScript APIs for embedding reasoning directly into applications. In future work, Eyeling will be the basis for, among others, supporting browser-level privacy controls, as proposed by the European Commission's Digital Omnibus Proposal in Article 88b[4].

To test whether the synthesis method generalizes beyond Eyeling's JavaScript target, we had *Claude Opus 4.7* synthesize a C++ N3 reasoner from the W3C specifications, the compliance test suite, and Eyeling's source, with minimal human input. It produced a spec-compliant reasoner in 29 commits and about 6 hours of wall-clock time. The result is *compliant* but does not match Eyeling's *maturity* or *performance*, and still needs human-in-the-loop refinement. Once a reference implementation exists, porting to a new platform by LLM synthesis is cheap.

Disclaimer

During the preparation of this work, the author(s) used ChatGPT to improve the writing style the paper. The author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

[3]Screencast of a demonstration of the capabilities of the Eyeling Playground: https://doi.org/10.5281/zenodo.20415011
[4]https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52025PC0837

Acknowledgments

This work was partially funded by the EU Horizon 2020 project HARNESS (Grant Agreement No. 101169409) and by COST Action CA23147 GOBLIN, supported by COST (European Cooperation in Science and Technology, https://www.cost.eu).

References

[1] A. Polleres, D. Rincon-Yanez, A. Anjomshoaa, D. Dobriy, E. Sallinger, From Data Spaces to Agent Spaces (2026). URL: https://dbis.rwth-aachen.de/SDS26/.

[2] R. Iannella, S. Villata, ODRL Information Model 2.2, 2018. URL: https://www.w3.org/TR/odrl-model/.

[3] W. Slabbinck, J. Rojas Meléndez, B. Esteves, P. Colpaert, R. Verborgh, Interoperable Interpretation and Evaluation of ODRL Policies, in: E. Curry, M. Acosta, M. Poveda-Villalón, M. van Erp, A. Ojo, K. Hose, C. Shimizu, P. Lisena (Eds.), The Semantic Web, Springer Nature Switzerland, Cham, 2025, pp. 192–209. doi:10.1007/978-3-031-94578-6_11.

[4] B. Esteves, W. Slabbinck, Y. Sellami, A. Cimmino, V. Rodríguez-Doncel, R. Verborgh, Capturing Requests and Context for ODRL-based Access and Usage Control, in: Joint Proceedings of the 16th Workshop on Ontology Design and Patterns and the 1st Workshop on Bridging Hybrid Intelligence and the Semantic Web (WOP-HAIBRIDGE 2025) co-located with the 24th International Semantic Web Conference (ISWC 2025), 2025. URL: https://ceur-ws.org/Vol-4093/paper5.pdf.

[5] P. Hochstenbach, notation3tests, https://codeberg.org/phochste/notation3tests, 2026. Accessed: 2026-05-16.

[6] W. Van Woensel, D. Arndt, P.-A. Champin, D. Tomaszuk, G. Kellogg, Notation3 Language, 2026. URL: https://w3c.github.io/N3/spec/.

[7] J. De Roo, eyereasoner/eye: Euler Yet Another Proof Engine, https://github.com/eyereasoner/eye, 2026. Accessed: 2026-05-15.

[8] R. Dachselt, L. Gerlach, P. Hanisch, A. Ivliev, M. Krötzsch, M. Marx, J. Méndez, Nemo at v0. 10: Explainable Web Rule Reasoning for RDF, SPARQL, and More (2026). URL: https://www.mt.inf.tu-dresden.de/cnt/uploads/nev-eswc.pdf.

[9] V. Kadzhaia, P. Bonte, Neuro-Symbolic Stream Reasoning with Kolibrie, in: The Semantic Web – ESWC 2026, volume 16550 of *Lecture Notes in Computer Science*, Springer, Cham, 2026, pp. 39–57. doi:10.1007/978-3-032-25159-6_3.

[10] R. Taelman, J. Van Herwegen, M. Vander Sande, R. Verborgh, Comunica: A Modular SPARQL Query Engine for the Web, in: The Semantic Web – ISWC 2018, volume 11137 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 239–255. doi:10.1007/978-3-030-00668-6_15.

[11] B.-E. Tam, P. Colpaert, R. Taelman, Demonstrating Online Schema Alignment in Decentralized Knowledge Graphs Querying, 2026. URL: https://arxiv.org/abs/2604.19205. arXiv:2604.19205.

[12] J. P. Balhoff, C. J. Mungall, Whelk: An OWL EL+RL reasoner enabling new use cases, Transactions on Graph Data and Knowledge 2 (2024) 7:1–7:17. doi:10.4230/TGDK.2.2.7.

[13] J. Wright, J. D. Roo, I. Smessaert, EYE JS: A client-side reasoning engine supporting Notation3, RDF Surfaces and RDF Lingua, in: Posters, Demos, and Industry Tracks: From Novel Ideas to Industrial Practice, co-located with 23nd International Semantic Web Conference (ISWC 2024)., 2024. URL: https://ceur-ws.org/Vol-3828/paper8.pdf.

[14] W. Van Woensel, P. Hochstenbach, Notation3 Builtin Functions, 2026. URL: https://w3c-cg.github.io/n3Builtins/.

[15] D. Arndt, P.-A. Champin, Notation3 Semantics, 2026. URL: https://w3c.github.io/N3/reports/20230703/semantics.html.

[16] P. Colpaert, P. Sowinski, It's Time to Standardize RDF Messages, 2026. URL: http://arxiv.org/abs/2604.22619. doi:10.48550/arXiv.2604.22619, arXiv:2604.22619 [cs.DB] version: 1.