

Contents

Ontology Versioning, Provenance & Collaborative Governance	1
A Technical Report	1
Table of Contents	1
1. The Problem — Versioning & Rollback in RDF	2
2. RDF Mechanisms for Provenance & Versioning	2
2.1 Named Graphs	2
2.2 RDF-star	3
2.3 PROV-O	3
2.4 How They Combine	4
3. PROV-O Use Cases & Patterns	4
3.1 Core Use Cases	4
3.2 Algorithm-Version Rollback Pattern	4
3.3 Deletion Handling	5
4. Triplestore Options with Versioning & Inference	6
4.1 TerminusDB	6
4.2 Kolibrie	6
4.3 The Inference–Versioning Gap	6
5. Integrating PROV-O with GitHub	7
5.1 Core Mapping: Git Commit = PROV-O Changeset	7
5.2 Changeset Generation Workflow	7
5.3 Rollback Strategy	8
6. Collaborative Ontology Governance	8
6.1 Modularisation by Team	8
6.2 Namespace Ownership	8
6.3 Conflict Types	9
6.4 CI/CD Pipeline	9
6.5 Deployment Pipeline	10
6.6 Governance Tiers	10
7. Real-World Examples & References	10
Projects using this approach	10
Key tools	11
Governance frameworks	11

Ontology Versioning, Provenance & Collaborative Governance

A Technical Report

Date: 2026-05-15

Context: Architecture discussion for collaborative ontology development with versioning, rollback, and CI/CD governance.

Table of Contents

1. The Problem — Versioning & Rollback in RDF

2. RDF Mechanisms for Provenance & Versioning
 - 2.1 Named Graphs
 - 2.2 RDF-star
 - 2.3 PROV-O
 - 2.4 How They Combine
 3. PROV-O Use Cases & Patterns
 - 3.1 Core Use Cases
 - 3.2 Algorithm-Version Rollback Pattern
 - 3.3 Deletion Handling
 4. Triplestore Options with Versioning & Inference
 - 4.1 TerminusDB
 - 4.2 Kolibrie
 - 4.3 The Inference–Versioning Gap
 5. Integrating PROV-O with GitHub
 - 5.1 Core Mapping: Git Commit = PROV-O Changeset
 - 5.2 Changeset Generation Workflow
 - 5.3 Rollback Strategy
 6. Collaborative Ontology Governance
 - 6.1 Modularisation by Team
 - 6.2 Namespace Ownership
 - 6.3 Conflict Types
 - 6.4 CI/CD Pipeline
 - 6.5 Deployment Pipeline
 - 6.6 Governance Tiers
 7. Real-World Examples & References
-

1. The Problem — Versioning & Rollback in RDF

The starting motivation was Wikibase’s ability to track the full history of changes to ontology classes, properties, and individuals — with rollback — something plain RDF does not provide natively.

The core challenge: **there is no single tool that gives you OWL inference + triple-level versioning with rollback out of the box.** You must either accept tradeoffs or combine tools.

Three RDF-native mechanisms can address parts of this problem: Named Graphs, RDF-star, and PROV-O. They are complementary, not alternatives.

2. RDF Mechanisms for Provenance & Versioning

2.1 Named Graphs

A fourth element (graph name) added to each triple, forming a quad. Lets you partition triples into named sets and make statements about the whole set.

```
# Triples inside a named graph
:changeset42 { :MyClass a owl:Class . }
```

```
# Metadata about the graph
:changeset42 dc:created "2026-05-15" ;
             dc:creator :Gaetan .
```

Granularity: graph-level only — you annotate the whole graph, not individual triples.

Use for: changeset snapshots, version grouping, rollback by graph deletion.

2.2 RDF-star

Part of RDF 1.2. Makes a triple itself the subject of another triple — enabling per-triple annotation without verbose reification.

```
<<:MyClass a owl:Class>> prov:wasAttributedTo :Gaetan ;
                             prov:generatedAtTime "2026-05-15"^^xsd:date .
```

Granularity: triple-level.

Use for: per-triple provenance, algorithm-source tagging.

Note: syntax mechanism only — does not define what the annotation *means*; requires a vocabulary (e.g. PROV-O) for that.

2.3 PROV-O

The W3C Provenance Ontology. A vocabulary of classes and properties describing *who did what, when, and how*. Not a structural mechanism — rides on top of Named Graphs or RDF-star.

Three core concepts:

Concept	Meaning
prov:Entity	A thing (a triple, a dataset, a class)
prov:Activity	Something that happened (an edit session, a pipeline run)
prov:Agent	Who or what was responsible (a person, an algorithm)

```
:EditSession1 a prov:Activity ;
  prov:startedAtTime "2026-05-15"^^xsd:dateTime ;
  prov:wasAssociatedWith :Gaetan .
```

```
:MyClass prov:wasGeneratedBy :EditSession1 .
```

Purpose: audit log — answers “*how did this come to exist?*”

Not designed for: rollback, diffing, or integrity constraints on its own.

2.4 How They Combine

Mechanism	Role	Granularity
Named Graphs	Groups triples into versioned changesets	Graph-level
RDF-star	Attaches provenance to individual triples	Triple-level
PROV-O	Vocabulary describing what the provenance means	Depends on carrier

Recommended combination: Named Graphs per changeset (Level 2) as the primary versioning mechanism, with RDF-star + PROV-O for algorithm-generated triples that need per-triple rollback.

3. PROV-O Use Cases & Patterns

3.1 Core Use Cases

PROV-O was designed for the following scenarios:

- **Scientific reproducibility** — tracking data pipeline steps from raw input to published result; used heavily in bioinformatics and earth sciences.
- **Linked Data trust assessment** — consumers assess freshness and source quality of RDF triples from external datasets.
- **Regulatory audit trails** — GDPR, financial, healthcare; immutable record of who handled data, when, and how.
- **Algorithm/ML output tracking** — tagging triples by the model version that generated them.
- **Data integration lineage** — tracing which source system a record came from through a multi-system pipeline.
- **Attribution and credit** — academic and cultural heritage contexts; who created what under which licence.
- **Workflow debugging** — tracing bad outputs back to the activity and inputs that caused them.

Common thread: PROV-O records causality retrospectively. It tells you what happened; it does not prevent or undo anything on its own.

3.2 Algorithm-Version Rollback Pattern

This is a strong PROV-O use case. Tag triples at write time with their source algorithm using RDF-star:

```
<<:MyClass skos:broader :Animal>> prov:wasGeneratedBy :ClassifierV2 .  
<<:MyClass owl:equivalentClass :Mammal>> prov:wasGeneratedBy :ClassifierV2 .
```

Revert all triples from that algorithm version with a single SPARQL Update:

```
DELETE { ?s ?p ?o }
WHERE {
  <<?s ?p ?o>> prov:wasGeneratedBy :ClassifierV2 .
}
```

Caveats:

- **Cascade problem** — downstream triples derived from :ClassifierV2 output are not caught unless also tagged with prov:wasDerivedFrom. Walk the provenance chain explicitly.
- **Overwrite problem** — if the triple also existed from another source, deleting it removes that assertion too. Requires multi-source tracking.
- **Audit trail loss** — deleting the triple removes its RDF-star annotation. Record a deletion activity separately before removing.

3.3 Deletion Handling

Deletion is PROV-O's structural weak spot: once a triple is gone, there is nothing left to annotate. The workaround is to record the deletion explicitly before removing the triple:

```
:deletion1 a prov:Activity ;
  prov:used :MyClass ;
  rdfs:comment "Removed – deprecated in v0.5" ;
  prov:endedAtTime "2026-05-15"^^xsd:dateTime .
```

With Named Graph changesets, use a dedicated deletions graph:

```
:changeset43_deletions {
  :deletion1 a prov:Activity ;
  prov:used :MyClass ;
  prov:endedAtTime "2026-05-15"^^xsd:dateTime .
}
```

Then physically remove the triple from the graph that originally asserted it.

Full history query after this:

```
# What added triples about :MyClass?
SELECT ?changeset ?time WHERE {
  GRAPH ?changeset { :MyClass ?p ?o }
  ?changeset prov:generatedAtTime ?time .
}
```

```
# What deleted triples about :MyClass?
SELECT ?changeset ?time WHERE {
  GRAPH ?changeset { ?deletion prov:used :MyClass . }
  ?changeset prov:generatedAtTime ?time .
}
```

4. Triplestore Options with Versioning & Inference

4.1 TerminusDB

A graph database with Git-like versioning built in (branch, diff, rollback at triple/document level). Has a Python client. Active commercial backing.

Feature	Status
Triple-level versioning	Yes — native
Rollback	Yes — native
OWL inference	No — schema is structural (like JSON Schema), not a description logic reasoner
SPARQL	Partial (also has its own WOQL query language)

Best for: when versioning is the primary requirement and OWL inference is not needed.

4.2 Kolibrie

A high-performance SPARQL engine and RDF stream processing engine built in Rust by KU Leuven's Stream Intelligence Lab. Actively maintained (last update March 2026).

Feature	Status
Performance	SIMD + parallel processing (Rayon), very fast
Inference	Built-in reasoner: forward/backward chaining, semi-naive evaluation
RDF stream processing	Yes — timestamped triples, sliding/tumbling windows
Triple-level versioning	No

Best for: high-throughput querying and rule-based inference. Does not close the versioning gap.

4.3 The Inference–Versioning Gap

No single tool currently provides both full OWL 2 inference and native triple-level versioning with rollback. The realistic combinations:

Need	Tool
Full OWL 2 reasoning	Apache Jena, GraphDB, Stardog
Fast RDFS/rule inference	Kolibrie, Oxigraph
Native versioning + rollback	TerminusDB
Git-backed SPARQL endpoint	Quit Store (check maintenance status)

Need	Tool
Provenance layer	PROV-O + Named Graphs or RDF-star

The practical recommendation: use **Git + PROV-O changesets** for versioning (described in Section 5) and a separate triplestore (Jena, GraphDB, or Kolibrie) for inference and querying.

5. Integrating PROV-O with GitHub

5.1 Core Mapping: Git Commit = PROV-O Changeset

Keep Git as the source of truth. The triplestore is a derived, queryable view. The Git commit SHA links the two systems.

```
repo/
  ontology/
    myontology.ttl          <- current state (what Git diffs)
  changesets/
    changeset_56c1577.ttl  <- PROV-O record for that commit
    changeset_e301a44.ttl
    changeset_56c1577_deletions.ttl
```

Each changeset TTL references its Git commit SHA:

```
<urn:git:56c1577> a prov:Entity, :OntologyChangeset ;
  prov:generatedAtTime "2026-05-15"^^xsd:dateTime ;
  prov:wasAssociatedWith :Gaetan ;
  rdfs:comment "OWL class label fix, OWL property ops" .
```

5.2 Changeset Generation Workflow

```
edit ontology.ttl
  ↓
pre-commit hook: diff HEAD~1 vs staged -> generate changeset_<sha>.ttl
  ↓
git commit (ontology + changeset together)
  ↓
GitHub Actions: load changeset TTL into triplestore
```

The diff is computed with rdflib in Python:

```
old = Graph().parse("myontology_prev.ttl")
new = Graph().parse("myontology.ttl")

added   = new - old   # -> named graph :changeset_<sha>
removed = old - new   # -> named graph :changeset_<sha>_deletions
```

Alternative: generate changesets in GitHub Actions (simpler local workflow, slight lag before changeset exists in repo).

Triplestore recovery: if the triplestore is ever lost, replay full history by loading all changeset TTLs from Git in order. Git is the durable recovery point.

5.3 Rollback Strategy

Two complementary rollback paths — both append a new corrective change, never rewriting history:

Path	How
File-level rollback	git revert <sha> -> new commit -> CI re-deploys previous T-Box
Semantic rollback	SPARQL DELETE triples from a specific changeset -> new corrective changeset recorded

6. Collaborative Ontology Governance

6.1 Modularisation by Team

Split the ontology using owl:imports. Each team owns one module file, preventing most Git conflicts by design.

```
ontology/  
  core/  
    core.ttl      <- shared upper ontology, owned by architecture team  
  domains/  
    clinical.ttl  <- Team A  
    genomics.ttl <- Team B  
    imaging.ttl   <- Team C  
  main.ttl       <- imports all modules
```

```
# main.ttl  
<http://example.org/ontology> a owl:Ontology ;  
  owl:imports <http://example.org/ontology/core/core> ;  
  owl:imports <http://example.org/ontology/domains/clinical> ;  
  owl:imports <http://example.org/ontology/domains/genomics> .
```

6.2 Namespace Ownership

Each team owns a namespace prefix. A CI check enforces that teams only define URIs within their namespace — preventing silent URI collisions.

```
@prefix core:      <http://example.org/ontology/core/> .
@prefix clinical: <http://example.org/ontology/clinical/> .
@prefix genomics: <http://example.org/ontology/genomics/> .
```

6.3 Conflict Types

Git merge conflicts (two teams edited the same file) are the easy case. The harder cases are **semantic conflicts** – no Git conflict, but the combined ontology is logically broken:

Conflict type	Example
Disjointness violation	Team A: <code>:Person owl:disjointWith :Org</code> / Team B: <code>:Employee a :Person, :Org</code>
Duplicate URI redefinition	Both teams define <code>:status</code> with different ranges
Breaking change to shared class	Team A renames a core class Team B's module depends on
Circular imports	Module A imports B which imports A

An OWL reasoner in CI catches semantic conflicts that Git cannot detect.

6.4 CI/CD Pipeline

```
PR opened
|
1. Syntax check          -- rdflib parse all TTL files
2. OWL consistency      -- run Hermit/ELK reasoner on merged ontology
3. SHACL governance     -- naming conventions, namespace ownership, required annotations
4. Breaking change     -- diff against main, flag removed/redefined public classes
5. PROV-O generation    -- compute changeset TTL (added + deleted triples)
|
All green --> PR can be merged
|
Merge to main --> auto-deploy to TEST triplestore (full T-Box + production A-Box)
|
Manual approval gate -- team explores results on TEST
|
Deploy to PROD triplestore
```

ROBOT is the standard tool for steps 1.–4. (ontology merging, reasoning, diff, SHACL validation).

6.5 Deployment Pipeline

```
feature/clinical-v2 --PR--> main --auto--> TEST --approval--> PROD
                                     |
                                     Full T-Box loaded against
                                     production A-Box -- catches
                                     schema changes that break
                                     existing individuals
```

TEST against the real A-Box is the critical gate: it reveals cases where a T-Box change (e.g. a removed class) breaks existing data that will not appear in unit tests.

6.6 Governance Tiers

Tier	Scope	Reviewers required
Core ontology	Upper classes, shared properties	Architecture committee, 2+ approvals
Domain module	Team's own namespace	Team lead only
Cross-domain dependency	One team references another team's class	Both team leads
Breaking change	Removing or redefining existing public classes	RFC issue + deprecation period + architecture sign-off

GitHub branch protection enforces: - No direct push to main — PRs only - CI must be green before merge - core/ changes require 2 architecture team approvals

7. Real-World Examples & References

Projects using this approach

Gene Ontology (GO)

World's largest gene function ontology. Distributed international consortium. Uses GitHub PRs, GitHub Actions on every commit, Jenkins for deployment. Multiple repos per domain. - geneontology/go-ontology - geneontology/pipeline

FIBO — Financial Industry Business Ontology

Multi-team ontology for the finance industry, contributed to by banks and financial institutions. GitHub-primary since 2020, Jenkins CI, automated hygiene tests on every commit, domain teams working in parallel (securities, derivatives, indices). Closest analogy to a multi-department enterprise ontology project. - edmcouncil/fibo - Infrastructure paper (ResearchGate)

OBO Foundry

Consortium of 100+ interoperable biomedical ontologies, each with its own GitHub repo, all using shared CI conventions. - obofoundry.org - OBO Academy — Project Ontology Development tutorial

Key tools

ROBOT — Standard build tool for ontology CI: reasoning, validation, format conversion, diff, modularisation. Peer-reviewed. - ROBOT paper — BMC Bioinformatics / PMC - robot.obolibrary.org

ODK (Ontology Development Kit) — Scaffolds a GitHub repo with ROBOT-based CI and GitHub Actions out of the box. - ODK paper (arXiv)

Kolibrie — High-performance Rust-based SPARQL engine with built-in rule reasoner and stream processing. - StreamIntelligenceLab/Kolibrie - Kolibrie website

TerminusDB — Graph database with native Git-like versioning. - terminusdb.com

Governance frameworks

BASF GOMO — Industrial governance model covering the full ontology lifecycle, roles, quality criteria, and change management. - BASF GOMO — Zenodo

Metaphacts Ontology Governance Guide - blog.metaphacts.com

W3C Collaborative Ontology Development - w3.org/wiki/Collaborative_Ontology_Development

Report generated from architecture discussion on 2026-05-15.