

Ontologies for Developers and AI Engineers

Version OWL 2 · SHACL 1.0 · SPARQL 1.1 · RDF 1.1 · verified 2026-11-21

Ontologies as a shared, machine-checkable model of meaning — RDF, OWL, SHACL, SPARQL, grounding LLMs, and how to pick tools.

Contents

Mental model 3

The semantic spectrum 3

Triples, IRIs and Turtle 5

Classes, properties and inference 7

OWL and the open world 8

Validating with SHACL 10

Querying with SPARQL 12

Designing an ontology 13

Ontologies in AI engineering 15
Choosing tools 17
Further reading 18

Mental model

An ontology is a shared, explicit, machine-readable model of what things exist in a domain and how they relate — written down separately from any one database or codebase. A schema tells a machine what data is *shaped* like; an ontology tells it what data *means*, so it can infer facts nobody stored. The twist that surprises developers: ontology languages assume an open world, where a missing fact is unknown rather than false — so validation is a separate tool.

The semantic spectrum

Artifact	Adds	Standard
Glossary	Agreed terms and definitions	—
Taxonomy	A broader/narrower hierarchy	SKOS

Artifact	Adds	Standard
Thesaurus	Synonyms and related terms	SKOS
Schema	Shape of records	JSON Schema, SQL DDL
Ontology	Typed relations, rules, inference	RDFS, OWL

Most teams need less than a full ontology. A product catalog or tag system is a taxonomy: concepts with labels and a hierarchy, which SKOS expresses without any logic. Reach for an ontology when several systems, or an LLM, must agree on what relationships mean.

```
PREFIX skos: <http://www.w3.org/2004/02/skos/
core#>
PREFIX ex: <https://example.com/cat#>

ex:laptops a skos:Concept ;
  skos:prefLabel "Laptops"@en ;
```

```
  skos:altLabel  "Notebooks"@en ;
  skos:broader   ex:computers .
```

> **Tip:** Start with SKOS. A taxonomy can later grow into an ontology; untangling a premature one is harder.

Triples, IRIs and Turtle

Term	Meaning
Triple	subject – predicate – object
IRI	Global identifier for a thing or relation
Literal	A typed value, such as `"99.50"^^xsd:decimal`
Graph	A set of triples
Turtle	The readable text syntax for RDF

RDF stores every fact as a triple, and every name is an IRI, so two datasets that use the same IRI are talking about the same thing — that is how merging works with no join keys. The examples below all build on this one file, `shop.ttl`.

```
PREFIX ex: <https://example.com/shop#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ex:Order    a owl:Class .
ex:Customer a owl:Class .
ex:VipCustomer rdfs:subClassOf ex:Customer .
ex:placedBy a owl:ObjectProperty ;
  rdfs:domain ex:Order ;
  rdfs:range  ex:Customer .
ex:o42 a ex:Order ;
  ex:placedBy ex:ada ;
  ex:total "99.50"^^xsd:decimal .
```

> **Warning:** Mint IRIs on a domain you control and never reuse them for something else. They are as permanent as public URLs.

Classes, properties and inference

Axiom	A reasoner concludes
`A rdfs:subClassOf B`	Every `A` is also a `B`
`p rdfs:domain C`	Any subject of `p` is a `C`
`p rdfs:range C`	Any object of `p` is a `C`
`p rdfs:subPropertyOf q`	`x p y` implies `x q y`

The ontology part of `shop.ttl` is a set of axioms. A reasoner applies them to the data and adds the triples they entail — `ex:ada` never said she is a customer, but the range of `ex:placedBy` makes her one.

```
from rdflib import Graph, Namespace, RDF
from owlrl import DeductiveClosure, RDFS_Semantics

EX = Namespace("https://example.com/shop#")
g = Graph().parse("shop.ttl")
g.add((EX.o7, EX.placedBy, EX.widget))  # typo
DeductiveClosure(RDFS_Semantics).expand(g)
```

```
print((EX.ada, RDF.type, EX.Customer) in g)
# True
print((EX.widget, RDF.type, EX.Customer) in g)
# True — the typo became a customer
```

> **Gotcha:** `rdfs:domain` and `rdfs:range` are not constraints. They never reject data; they classify whatever they are given.

OWL and the open world

Assumption	Databases	OWL
Missing fact	False	Unknown
Two names	Two things	Maybe the same thing
Contradiction	Rejected write	Inconsistent ontology

OWL 2 adds richer axioms — disjoint classes, cardinality, inverse, transitive and functional properties. Its profiles

(EL, QL, RL) trade expressiveness for fast reasoning; RL is the one rule engines and triple stores usually implement.

```
from rdflib import Graph, Namespace, OWL, RDF
from owlrl import (
  DeductiveClosure, OWLRL_Semantics)

EX = Namespace("https://example.com/shop#")
g = Graph().parse("shop.ttl")
g.add((EX.placedBy, RDF.type,
       OWL.FunctionalProperty))
g.add((EX.o42, EX.placedBy, EX.bob))
DeductiveClosure(OWLRL_Semantics).expand(g)

print((EX.ada, OWL.sameAs, EX.bob) in g)
# True — not an error
```

> **Gotcha:** “At most one customer” does not reject a second one. Without a unique-name assumption, OWL concludes the two customers are the same individual.

Validating with SHACL

Constraint	Checks
`sh:targetClass`	Which nodes the shape applies to
`sh:minCount` / `sh:maxCount`	How many values
`sh:datatype`	Literal type
`sh:class`	Type of the linked node
`sh:minInclusive`	Numeric range

SHACL is the closed-world counterpart: shapes describe what valid data looks like and a validator returns a report. Use OWL to state meaning and SHACL to gate writes, the way you use types and runtime checks.

```
PREFIX ex: <https://example.com/shop#>
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

ex:OrderShape a sh:NodeShape ;
  sh:targetClass ex:Order ;
  sh:property [
    sh:path ex:placedBy ;
    sh:minCount 1 ; sh:maxCount 1 ;
    sh:class ex:Customer ] ;
  sh:property [
    sh:path ex:total ;
    sh:datatype xsd:decimal ;
    sh:minInclusive 0 ] .
```

> **Gotcha:** SHACL does no inference. Validating `shop.ttl` fails with "Value does not have class ex:Customer", because `ex:ada` is a customer only by inference. Type data explicitly or run pySHACL with `inference="rdfs"`.

Querying with SPARQL

Pattern	Does
`?s ex:p ?o .`	Match triples
`;`	Same subject, next predicate
`p*`	Zero or more hops
`FILTER (...)`	Keep matching rows
`OPTIONAL { }`	Left join

SPARQL matches graph patterns: every triple in `WHERE` must hold for the same variable bindings. Property paths let a query walk a class hierarchy itself, which covers many cases where you would otherwise run a reasoner first.

```
PREFIX ex:   <https://example.com/shop#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
schema#>

SELECT ?order ?who ?total WHERE {
  ?type rdfs:subClassOf* ex:Customer .
  ?who  a ?type .
  ?order ex:placedBy ?who ;
         ex:total ?total .
  FILTER (?total > 50)
}
```

With `ex:ada a ex:VipCustomer` added, this returns `ex:o42`, `ex:ada`, `99.50`.

Designing an ontology

Step	Practice
Scope	Write competency questions it must answer
Reuse	Import existing vocabularies before minting terms

Step	Practice
Model	Classes, then properties, then constraints
Test	Each competency question becomes a SPARQL test
Version	`owl:versionIRI` and changelog; deprecate, never delete

Competency questions — “Which VIP customers ordered over 50 this month?” — keep scope honest: a term no question needs does not belong yet. Reuse shared vocabularies such as schema.org, Dublin Core or your industry’s standard so partners already understand your terms.

> **Gotcha:** Modeling every noun as a class is the classic beginner mistake. “Shipped” is a status value, not a subclass of Order.

Ontologies in AI engineering

Use	How the ontology helps
LLM extraction	Allowed types and predicates bound the output
GraphRAG	Typed relations make graph hops meaningful
Agent tools	One vocabulary for tool arguments and results
Data contracts	One model generates code-level types

An LLM turns unstructured text into triples well only when it knows the allowed terms. Derive the structured-output schema from the ontology, so the model cannot invent a predicate, then run SHACL on the result. LinkML takes the opposite route: one YAML model generates Pydantic classes, JSON Schema, SHACL and OWL.

```
from rdflib import Graph, RDF, OWL

g = Graph().parse("shop.ttl")
preds = sorted(
  g.qname(p) for p in
  g.subjects(RDF.type, OWL.ObjectProperty))

triple = {
  "type": "object",
  "properties": {
    "s": {"type": "string"},
    "p": {"enum": preds},  # ["ex:placedBy"]
    "o": {"type": "string"}},
  "required": ["s", "p", "o"]}
```

> **Warning:** An LLM-drafted ontology looks plausible and is often subtly wrong. Treat it as a draft for a domain expert, not a source of truth.

Choosing tools

Need	Tools
Author OWL	Protégé (desktop), WebProtégé (shared)
Govern taxonomies	PoolParty, TopBraid EDG
Store and query	GraphDB, Stardog, Apache Jena Fuseki
Code	rdflib, pySHACL, OWL-RL, Apache Jena
Schema-first	LinkML
Property graph	Neo4j with neosemantics

Pick by job, not by feature list. Editors model; governance platforms add workflow, approval and stewardship for business users; triple stores query and reason at scale. Data catalogs such as OvalEdge govern glossaries and lineage and pair with a separate editor for deep OWL modeling.

> **Note:** Vendor “best tools” roundups rank their own product first. Shortlist by the job above, then prototype on your own data.

Further reading

- OWL 2 Primer — classes, properties and the open world, with examples
- SHACL specification — every constraint component
- SPARQL 1.1 Query Language — property paths and aggregates
- Ontology Development 101 — the classic design method
- LinkML — one schema to Pydantic, JSON Schema and SHACL
- OvalEdge — Ontology management tools — a vendor comparison of tools