OntologyLM / AI Ontology Engineer

Master Commercial Build Prompt for Claude

Prepared from complete product requirements and system architecture

Table of Contents

1 MASTER BUILD PROMPT FOR CLAUDE

1.1 OntologyLM / AI Ontology Engineer - Commercial Multi-Domain Enterprise Platform

Purpose: This prompt is intended to be given directly to Claude (Claude Code / Claude CLI / Claude-compatible coding environment) to research, plan, implement, test, validate, containerize, document, and harden a commercial application that can generate standards-compliant ontologies from business-domain documentation across major industries.

---

2 0. YOUR ROLE

You are acting simultaneously as:

- Principal Enterprise Architect
- Principal Ontology Engineer
- Knowledge Graph Architect
- AI/LLM Architect
- Senior Python/FastAPI Engineer
- Senior TypeScript/React Engineer
- ML/LLM Training Engineer
- DevSecOps / Platform Engineer
- Semantic Web Standards Expert
- QA / Test Automation Architect
- Enterprise Product Architect

You have deep practical experience with RDF, RDFS, OWL 2, SHACL, SKOS, SPARQL, PROV-O, JSON-LD, ontology design patterns, knowledge graphs, LLMs, RAG, LangGraph-style orchestration, PostgreSQL, vector search, Docker, Kubernetes, CI/CD, identity, auditability, and enterprise governance.

Your mission is to build **OntologyLM / AI Ontology Engineer**, a commercial-grade, multi-tenant, scalable semantic engineering platform.

Do not build a toy application. Do not build a single prompt that emits Turtle. Do not shortcut ontology engineering quality controls.

---

3 1. PRODUCT VISION

Build an enterprise product that transforms business/domain knowledge into governed, validated, traceable semantic models.

The product must support a user providing a Markdown description such as:

insurance.md
banking.md
pharma.md
healthcare.md
manufacturing.md
supply-chain.md
energy.md
telecommunications.md
retail.md
public-sector.md

and produce a complete ontology engineering package including:

requirements/
ontology/
constraints/
taxonomies/
mappings/
competency-questions/
queries/
examples/
documentation/
provenance/
validation/
release/

The long-term product must behave like an **AI Ontology Engineer**, not an ontology text generator.

The operating philosophy is:

Business Knowledge
↓
Requirements Understanding
↓
Competency Questions
↓
Conceptual Modeling
↓
Vocabulary Reuse
↓
Ontology Architecture
↓
Typed Ontology Intermediate Representation
↓
Deterministic Semantic Compilation
↓
OWL / SHACL / SKOS / SPARQL / PROV-O
↓
Reasoning + Validation
↓
Automated Repair Proposal
↓
Full Revalidation
↓
Human Governance
↓
Versioned Release
↓
Knowledge Graph / GraphRAG / Enterprise AI

---

4 2. CRITICAL ARCHITECTURAL PRINCIPLE

The most important rule in the entire system is:

The LLM is not the semantic authority.

The LLM may analyze, plan, propose, explain, classify, map, and repair.

The LLM must NOT directly determine that an ontology is correct, production-ready, or publishable.

Use the following architecture:

LLM / Agent
↓
Structured Proposal
↓
Typed Ontology IR
↓
Pydantic / JSON Schema Validation
↓
Cross-Reference Validation
↓
Deterministic Compiler
↓
OWL / SHACL / SKOS / SPARQL / PROV-O
↓
External Semantic Validation
↓
Release Policy
↓
Human Review / Approval

The same validated OIR + same compiler version must generate the same semantic output.

Do not make free-form Turtle the primary agent output.

---

5 3. STARTING POINT / EXISTING REPOSITORY

If an existing OntologyLM starter repository is supplied, **inspect and extend it rather than replacing it unnecessarily**.

The expected existing system may already contain:

- FastAPI API
- nine-agent workflow
- typed Pydantic OIR
- deterministic OWL compiler
- deterministic SHACL compiler
- PROV-O compiler
- validation interfaces
- insurance-domain E2E fixture
- PostgreSQL persistence
- Redis worker queue
- bounded repair/revalidation loop

- release policy
- release certificate
- governance state transitions
- lightweight Ontology Studio
- Docker Compose
- unit/integration tests

Before modifying code:

1. Read the repository completely.
2. Run all existing tests.
3. Map implemented features against this specification.
4. Produce a gap analysis.
5. Preserve working functionality.
6. Refactor only when it creates a clear architectural improvement.
7. Never delete tests to make the build pass.

---

6 4. REQUIRED DEVELOPMENT METHOD

You MUST follow this sequence.

6.1 Phase A - Research

Research the current implementation guidance and standards for:

- RDF / RDFS
- OWL 2
- OWL profiles: EL, QL, RL, DL
- SHACL
- SKOS
- SPARQL
- PROV-O
- JSON-LD
- ontology design patterns
- ontology modularity
- ontology versioning
- semantic diffing
- ontology reasoning
- HermiT
- ELK
- OWLAPI
- Apache Jena / Fuseki

- pySHACL
- ROBOT ontology workflows
- RDFLib
- LangGraph or equivalent stateful orchestration
- model serving using vLLM or equivalent
- fine-tuning using PEFT / LoRA / QLoRA
- post-training / preference optimization
- enterprise multi-tenancy
- OIDC / SAML / RBAC
- audit logging
- Kubernetes scaling
- OpenTelemetry

Document important technical decisions before implementation.

6.2 Phase B - Architecture

Produce:

1. Context diagram
2. Container diagram
3. component architecture
4. workflow graph
5. deployment architecture
6. database model
7. OIR schema architecture
8. agent contracts
9. validation architecture
10. training-data architecture
11. domain-pack architecture
12. security architecture
13. multi-tenancy architecture
14. observability architecture
15. release/governance state machine

Use Mermaid for diagrams and keep diagrams clean with no overlapping arrows.

6.3 Phase C - Implementation Plan

Create:

- epics
- milestones
- dependency graph
- prioritized backlog

- acceptance criteria
- test strategy
- migration strategy
- risk register

Only after this should implementation begin.

6.4 Phase D - Implementation

Implement feature-by-feature with tests.

6.5 Phase E - Validation

Run:

- unit tests
- API tests
- integration tests
- E2E semantic tests
- negative tests
- ontology reasoner tests
- SHACL tests
- CQ tests
- security tests
- lint/type checking
- Docker build
- migration tests
- regression tests

6.6 Phase F - Delivery

Deliver:

- source repository
- Docker Compose
- Kubernetes deployment configuration
- API docs
- architecture docs
- user guide
- administrator guide
- semantic methodology guide
- domain-pack authoring guide
- model-training guide
- production-readiness checklist

7 5. CORE FUNCTIONAL REQUIREMENTS

The platform must support:

1. Business/domain Markdown ingestion.
2. Optional structured Markdown templates.
3. Domain scope extraction.
4. Explicit exclusions.
5. glossary extraction.
6. synonym and acronym identification.
7. business entity extraction.
8. actor/role extraction.
9. event extraction.
10. process extraction.
11. business-rule extraction.
12. lifecycle-state extraction.
13. temporal concept extraction.
14. classification/taxonomy extraction.
15. competency-question generation.
16. existing ontology/vocabulary reuse.
17. semantic mapping decisions.
18. conceptual ontology modeling.
19. OWL axiom generation.
20. SHACL constraint generation.
21. SKOS concept-scheme generation.
22. SPARQL CQ generation.
23. sample RDF generation.
24. PROV-O provenance.
25. deterministic compilation.
26. RDF parsing.
27. OWL profile checks.
28. logical reasoning.
29. unsatisfiable-class detection.
30. SHACL validation.
31. CQ execution.
32. ontology-quality checks.
33. semantic duplicate detection.
34. naming/IRI policy validation.
35. documentation coverage validation.
36. vocabulary reuse scoring.
37. provenance coverage scoring.
38. ontology versioning.

39. semantic diff.
40. repair proposals.
41. bounded repair loop.
42. human ontology-engineer review.
43. SME review.
44. approval and publication lifecycle.
45. immutable release package.
46. release certificate.
47. audit trail.
48. training-data capture.
49. preference-data capture.
50. multi-domain domain packs.

---

8 6. NINE-AGENT SEMANTIC WORKFLOW

Implement exactly these nine logical roles. They may run as LangGraph nodes or equivalent deterministic workflow stages.

8.1 Agent 1 - Requirements Interpreter

Responsibilities:

- parse business Markdown
- extract explicit requirements
- assign stable requirement IDs
- retain source evidence
- distinguish explicit statements from inferences
- identify ambiguity
- identify unresolved terminology
- assign confidence

Must NOT design ontology classes yet.

Required output fields:

requirement_id
statement
category
source_document
source_section
source_quote_or_hash
explicit
confidence

ambiguities
notes

System instruction:

> Never invent a business fact. Any inferred fact must be explicitly marked as inferred.

8.2 Agent 2 - Domain Analyst

Identify:

- entity
- role
- event
- state
- process
- relationship
- attribute
- identifier
- value object
- classification
- taxonomy concept

Prevent common modeling errors:

- role modeled as permanent type without justification
- identifier modeled as entity
- relation modeled as subclassing
- synonym duplication
- database tables copied as ontology classes without semantic justification

Output conceptual candidates only.

8.3 Agent 3 - Competency Question Engineer

Generate measurable questions traceable to business requirements.

Each CQ must have:

cq_id
question
requirement_ids
category
expected_answer_type

involved_concepts
pass_condition

Categories may include:

taxonomy
relationship
temporal
cardinality
aggregation
traversal
classification
provenance

Reject CQs with no business traceability.

8.4 Agent 4 - Vocabulary Reuse Engineer

For every candidate concept, return one of:

REUSE
EXTEND
MAP
CREATE_NEW

Search:

- customer enterprise ontology
- approved internal vocabulary registry
- domain pack
- W3C vocabularies
- approved public ontologies
- licensed industry ontologies
- ontology design patterns

Do not use lexical similarity alone for semantic equivalence.

owl:equivalentClass and owl:equivalentProperty require strong semantic equivalence.

Track:

source ontology
version
license
IRI
mapping type

confidence
rationale

8.5 Agent 5 - Ontology Architect

Create conceptual ontology architecture.

Rules:

- true is-a only for subclassing
- object properties for entity relationships
- datatype properties for literal values
- separate roles from entities where appropriate
- distinguish events from states
- avoid overly deep taxonomies
- avoid over-modeling
- use approved terms when possible
- identify possible disjointness carefully
- preserve requirement traceability

The agent must output Ontology IR candidates, NOT Turtle.

8.6 Agent 6 - OWL Semantic Modeler

Convert approved conceptual architecture into formal OWL-oriented OIR.

Support:

- subclass axioms
- equivalent classes
- disjoint classes
- object properties
- datatype properties
- inverse properties
- functional / inverse functional properties
- symmetric / asymmetric
- transitive
- reflexive / irreflexive
- domain/range
- property hierarchy
- property chain axioms
- existential restrictions

- universal restrictions
- hasValue
- minimum / maximum / exact cardinality
- qualified cardinality

Every axiom must include:

rationale
requirement evidence
confidence
explicit_or_inferred

Never impose closed-world assumptions through OWL.

Recommend OWL profile:

OWL2_EL
OWL2_QL
OWL2_RL
OWL2_DL

8.7 Agent 7 - SHACL & Data Quality Engineer

Generate operational RDF constraints.

Support:

- minCount
- maxCount
- datatype
- node kind
- value range
- regex pattern
- enumeration
- class membership
- qualified value constraints
- closed shapes only when justified

Every constraint requires:

business requirement
shape target
path
severity

message
rationale

Severities:

INFO
WARNING
VIOLATION
CRITICAL

Critical rule:

> OWL semantics and SHACL validation are different concerns. Do not automatically convert every OWL restriction to SHACL or every SHACL constraint to OWL.

8.8 Agent 8 - Semantic Validation Engineer

This agent coordinates and interprets deterministic validator results but must NOT alter the ontology.

Validate:

- OIR schema
- OIR cross references
- RDF syntax
- IRI policy
- OWL profile
- logical consistency
- unsatisfiable classes
- SHACL
- competency questions
- documentation coverage
- provenance coverage
- duplicate concepts
- ontology pitfalls
- regression / semantic diff

Finding levels:

BLOCKER
CRITICAL
MAJOR
MINOR
INFO

Production release is impossible when:

- RDF cannot parse
- ontology is logically inconsistent
- named required classes are unsatisfiable
- critical SHACL checks fail
- required competency questions fail
- mandatory provenance fails

8.9 Agent 9 - Repair & Governance Engineer

Propose the smallest safe patch.

For every repair provide:

finding_id
affected_terms
affected_requirements
semantic_impact
backward_compatibility
OIR_patch
confidence
human_approval_required

Repair must follow:

failed candidate
↓
repair proposal
↓
new typed OIR
↓
complete validation again

Never skip full revalidation.

Bound retries to a configurable maximum, e.g. 2-3 attempts.

Never allow this agent to publish an ontology.

---

9 7. ONTOLOGY INTERMEDIATE REPRESENTATION (OIR)

Create a versioned, typed OIR and treat it as the internal semantic contract.

Use Pydantic models and generate JSON Schema.

Required top-level structures:

metadata
namespaces
imports
classes
object_properties
data_properties
annotation_properties
individuals
restrictions
property_axioms
shapes
concept_schemes
competency_questions
external_mappings
modeling_decisions
provenance

Minimum metadata:

ontology_id
title
description
base_iri
version
version_iri
language
owl_profile
domain_pack
created_at
compiler_version

Every semantic entity must support evidence:

requirement_id
document_id
source_section
source_text_hash
explicit
confidence
agent
model_version

Implement cross-reference validation so the following fail before compilation:

- undefined superclass
- undefined property
- undefined domain/range class
- duplicate IRI
- duplicate entity ID
- invalid inverse property
- invalid restriction target
- invalid SHACL path
- invalid CQ references
- invalid namespace prefix

---

10 8. DETERMINISTIC SEMANTIC COMPILERS

Build deterministic compilers for:

OIR → Turtle / OWL
OIR → SHACL
OIR → SKOS
OIR → SPARQL
OIR → JSON-LD context/artifact
OIR → PROV-O
OIR → Markdown documentation

Do not call an LLM inside these compilers.

Compiler requirements:

- deterministic output ordering where practical
- stable identifiers
- clear namespaces
- semantic annotations
- definitions via skos:definition
- preferred labels
- alternative labels
- provenance references
- version metadata

Support all OWL constructs represented by the OIR.

Never silently ignore an OIR construct. If a construct is unsupported, fail explicitly.

---

11 9. SEMANTIC VALIDATION PIPELINE

Build the following ordered release-gate pipeline:

OIR schema validation
↓
OIR reference validation
↓
compile
↓
RDF parse
↓
OWL profile check
↓
OWL reasoner
↓
SHACL validation
↓
CQ execution
↓
quality checks
↓
semantic regression
↓
quality score
↓
release policy

11.1 Required external semantic validation

Production mode must use real standards engines.

Recommended adapters:

RDFLib - parsing and lightweight RDF operations
OWLAPI - OWL manipulation/profile analysis
ROBOT - ontology automation workflows
HermiT - OWL 2 DL reasoning
ELK - OWL 2 EL reasoning
pySHACL - SHACL execution
Fuseki / Jena - SPARQL and RDF store

The system must NEVER report a skipped external reasoner as a pass.

Development mode may allow specific validators to be unavailable, but production mode must fail closed.

---

12 10. COMPETENCY QUESTION ENGINE

Every approved CQ must generate executable SPARQL.

Support:

- ASK
- SELECT
- expected result type
- positive fixture
- optional negative fixture

Record metrics:

total_cq
executable_cq
passing_cq
coverage
failed_cq

Target production CQ coverage >= 95% unless configured differently by customer policy.

---

13 11. POSITIVE AND NEGATIVE SEMANTIC FIXTURES

For each generated domain ontology, generate test fixtures.

Positive fixture:

- should satisfy required SHACL constraints
- should answer required CQs

Negative fixture:

- deliberately violates selected constraints
- must be rejected by SHACL

Example insurance requirement:

Every Claim has exactly one claim identifier.

Positive:

Claim C1 claimIdentifier "CLM-001"

Negative:

Claim C2 with no claimIdentifier

A test is only meaningful if the negative example actually fails.

---

14 12. QUALITY SCORING AND RELEASE POLICY

Implement both hard gates and a quality score.

Suggested quality components:

logical quality
CQ coverage
SHACL quality
terminology quality
provenance coverage
documentation coverage
reuse quality
human review quality

Example weighted score:

logical quality	20%
CQ coverage	20%
SHACL correctness	15%
terminology	10%
provenance	10%
documentation	10%
reuse	7.5%
human review	7.5%

Hard gates override score.

A score of 99 must still fail when the ontology is inconsistent.

Implement at least two policies:

14.0.1 Development policy

Allows unavailable optional external tools but clearly marks them SKIPPED.

14.0.2 Production policy

Requires:

- RDF parse PASS

- OIR PASS
- OWL reasoner executed and PASS
- unsatisfiable required classes = 0
- pySHACL executed and PASS
- critical findings = 0
- blockers = 0
- CQ coverage >= configured threshold
- provenance >= configured threshold
- quality score >= configured threshold

---

15 13. RELEASE CERTIFICATE

Generate a machine-readable certificate containing:

ontology ID
version
OWL profile
source hash
OIR hash
compiler version
model version
prompt version
validation engine versions
quality score
reasoner status
SHACL status
CQ metrics
provenance coverage
findings summary
human approvals
release policy
release decision
release timestamp
artifact hashes

Output JSON and optionally human-readable HTML/PDF later.

---

16 14. GOVERNANCE WORKFLOW

Implement a formal ontology lifecycle.

Suggested state machine:

DRAFT
↓
AI_GENERATED
↓
VALIDATING
↓
VALIDATED
↓
READY_FOR_REVIEW
↓
ONTOLOGIST_REVIEW
↓
SME_REVIEW
↓
APPROVED
↓
PUBLISHED
↓
DEPRECATED

Failure branches:

VALIDATION_FAILED
REPAIR_REQUIRED
REJECTED

Disallow invalid jumps such as:

READY_FOR_REVIEW → PUBLISHED

All transitions must record:

actor
actor_type
from_state
to_state
comment
timestamp
request_id

Only approved users can publish.

AI agents can never grant final approval.

---

17 15. MULTI-DOMAIN ARCHITECTURE

The platform must scale beyond insurance.

Implement domain packs.

Base structure:

```
domain_packs/
  core/
  insurance/
  finance/
  healthcare/
  pharma/
  manufacturing/
  supply-chain/
  energy/
  telecommunications/
  retail/
  public-sector/
```

Each domain pack should support:

```
manifest.yaml
terminology/
reference_ontologies/
patterns/
mappings/
validators/
examples/
cq_templates/
modeling_guidelines/
training_examples/
licenses/
```

Each domain pack must declare:

```
pack_id
version
domain
license
compatible_platform_version
approved_vocabularies
forbidden_vocabularies
recommended_patterns
custom_quality_rules
```

The core platform must remain domain-neutral.

---

18 16. INITIAL DOMAIN PACKS

Implement insurance first as the canonical E2E demonstration.

Then prepare skeleton packs for:

- Financial Services
- Healthcare
- Pharmaceutical / Life Sciences
- Manufacturing
- Supply Chain / Logistics
- Energy
- Telecommunications
- Retail
- Public Sector

Insurance first E2E example must include:

Policy
Policyholder
InsuredParty
Coverage
Risk
Claim
ClaimPayment
Broker

and demonstrate that:

Claim is NOT subclassOf Policy

Instead:

Claim --againstPolicy--> Policy

This test must be encoded in the automated suite.

---

19 17. STANDARDS RAG

Create a curated standards and semantic-reuse retrieval layer.

Indexes:

semantic-web standards
ontology design patterns
approved public ontologies
customer enterprise ontology
domain pack vocabulary
historical modeling decisions
organization glossary

Retrieval record must include:

IRI
label
definition
source
source_version
license
authority score
embedding score
semantic mapping status

Vector similarity is only a discovery signal.

Never convert embedding similarity directly to owl:equivalentClass.

---

20 18. SEMANTIC DUPLICATE DETECTION

Implement multi-signal duplicate detection using:

- preferred-label similarity
- alternative-label similarity
- embedding similarity
- definition similarity
- neighborhood similarity
- hierarchy position
- shared mappings
- human decision history

Return:

POSSIBLE_DUPLICATE
LIKELY_DUPLICATE

NOT_DUPLICATE
NEEDS_REVIEW

Only humans or explicit deterministic policies can approve destructive merges.

---

21 19. SEMANTIC DIFF

Implement ontology-version comparison.

Report:

added classes
removed classes
renamed classes
changed definitions
changed superclass
changed property domain/range
added/removed restrictions
changed SHACL constraints
changed mappings
changed annotations
CQ regression
new unsatisfiable classes
changed entailments
breaking vs non-breaking change

Do not rely on line diff alone.

---

22 20. BACKEND ARCHITECTURE

Preferred stack:

Python 3.12+
FastAPI
Pydantic v2
SQLAlchemy 2 / asyncpg
Alembic
PostgreSQL
pgvector
Redis
RDFLib

OWLAPI service where appropriate
pySHACL
ROBOT
HermiT / ELK adapters
Apache Jena Fuseki
S3-compatible object storage

Use clear service boundaries but do not create unnecessary microservices prematurely.

Suggested modules:

apps/api
apps/studio
services/orchestrator
services/worker
services/model_gateway
services/repository
services/vocabulary
services/ontology_registry
services/reasoner
services/quality
services/export
ontology_ir
compiler
validators
agents
domain_packs
training
evaluation
infrastructure

---

23 21. ASYNCHRONOUS JOB ARCHITECTURE

Commercial workloads must be asynchronous.

Pattern:

POST /jobs
  ↓
create durable run in PostgreSQL
  ↓
enqueue run_id in Redis / queue
  ↓

worker loads request from PostgreSQL
↓
workflow executes
↓
progress persisted
↓
result persisted
↓
artifacts stored in object storage

Redis is not the authoritative record of the request.

Store run lifecycle:

QUEUED
RUNNING
VALIDATING
REPAIRING
READY_FOR_REVIEW
FAILED

Support cancellation.

Support idempotency keys.

Support retry for infrastructure failures separately from semantic repair attempts.

---

24 22. DATABASE MODEL

Implement migrations for at least:

organizations
users
roles
organization_users
projects
project_members
project_settings
source_documents
source_document_versions
requirements
requirement_sources
business_terms
business_rules

ontology_projects
ontology_versions
ontology_ir_versions
ontology_entities
ontology_axioms
competency_questions
cq_test_runs
validation_runs
validation_findings
repair_proposals
review_tasks
review_comments
approvals
domain_packs
domain_pack_versions
external_vocabularies
vocabulary_terms
mapping_decisions
model_runs
model_versions
prompt_versions
training_examples
preference_examples
release_artifacts
generation_runs
audit_events
ontology_state_transitions

Every multi-tenant table must be scoped appropriately by organization/project.

---

25 23. API REQUIREMENTS

Use /api/v1.

Minimum endpoints:

25.1 Projects

POST /projects
GET /projects
GET /projects/{id}
PATCH /projects/{id}
DELETE /projects/{id}

25.2 Documents / requirements

POST /projects/{id}/documents
POST /projects/{id}/requirements/analyze
GET /projects/{id}/requirements

25.3 Generation

POST /projects/{id}/ontology/jobs
GET /runs/{run_id}
POST /runs/{run_id}/cancel

25.4 OIR

GET /ontology-versions/{id}/ir
PATCH /ontology-versions/{id}/ir
POST /ontology-versions/{id}/ir/validate

25.5 Validation

POST /ontology-versions/{id}/validate
GET /validation-runs/{id}
GET /validation-runs/{id}/findings

25.6 Repair

POST /ontology-versions/{id}/repair
GET /repair-proposals/{id}
POST /repair-proposals/{id}/accept
POST /repair-proposals/{id}/reject

25.7 Governance

POST /ontology-versions/{id}/review
POST /review-tasks/{id}/approve
POST /review-tasks/{id}/reject
POST /ontology-versions/{id}/publish

25.8 Export

GET /ontology-versions/{id}/export?format=ttl
GET /ontology-versions/{id}/export?format=owl
GET /ontology-versions/{id}/export?format=jsonld
GET /ontology-versions/{id}/export?format=package

25.9 Semantic diff

GET /ontology-versions/{left}/diff/{right}

25.10 Chat / explanation

POST /ontology-versions/{id}/chat

Chat answers must cite internal requirement IDs / modeling decisions.

26 24. FRONTEND - ONTOLOGY STUDIO

Build a professional web application using React / Next.js / TypeScript.

The interface must be suitable for ontology engineers, architects, SMEs, and reviewers.

Required screens:

26.1 1. Organization / Project Dashboard

Display:

- projects
- domains
- ontology versions
- quality scores
- validation status
- pending review
- latest generation jobs
- published releases

26.2 2. Requirements Studio

Split view:

Markdown editor | Extracted requirements
| Concepts
| Business rules
| Ambiguities
| Glossary

Users must approve/reject/edit extracted requirements.

26.3 3. Competency Question Studio

Display:

- CQ text
- linked requirements
- category
- involved terms
- generated SPARQL
- fixture result
- approval status

26.4 4. Ontology Graph Studio

Interactive graph with:

- class hierarchy
- object-property edges
- optional property types
- zoom/filter/search
- click-to-inspect
- domain/range visualization
- module filter

Use Cytoscape.js or equivalent robust graph library.

26.5 5. Term Inspector

Tabs:

Definition
Hierarchy
Relations
OWL Axioms
SHACL
Mappings
Evidence
Competency Questions
History
Comments

26.6 6. AI Ontology Copilot

Examples:

Why is Broker not modeled as Employee?
What requirement created hasCoverage?
Show all axioms derived from REQ-104.
Propose a less restrictive model for ClaimPayment.
Compare Policyholder and InsuredParty.

Copilot may propose changes but cannot apply production changes silently.

26.7 7. Quality Center

Show:

logical consistency
unsatisfiable classes

SHACL critical violations
CQ coverage
definition coverage
IRI compliance
provenance coverage
duplicate candidates
quality score
release gate

26.8 8. Semantic Diff

Visualize changes between versions.

26.9 9. Governance Center

Show lifecycle state, pending approvals, reviewer comments, publish/deprecate operations.

26.10 10. Domain Pack Registry

Allow admins to inspect installed domain packs and versions.

---

27 25. AUTHENTICATION, AUTHORIZATION, SECURITY

For commercial readiness implement:

- OIDC
- SAML option for enterprise deployments
- RBAC
- organization isolation
- project-level access
- audit logging
- encryption in transit
- encryption at rest
- secrets via secret manager / Kubernetes secrets
- secure headers
- CSRF protection if applicable
- strong CORS policy
- rate limiting
- request IDs
- secure file upload limits
- MIME validation
- malicious-file scanning hook
- dependency scanning

- SAST
- SBOM generation
- container vulnerability scanning

Do not rely on a single static API key for final commercial production authentication, although an API key may remain available for development/integration use.

Training data requires explicit consent.

Every source document should have:

training_allowed = false by default

Customer-private content must never enter another tenant's retrieval index.

---

28 26. MULTI-TENANCY

Design all platform data around:

organization
project
workspace/domain

Guarantee isolation at:

- SQL query level
- vector index metadata filter
- artifact storage prefix/bucket
- RDF dataset / named graph
- cache key
- model/RAG context

Add tests specifically designed to ensure tenant A cannot retrieve tenant B content.

---

29 27. KNOWLEDGE GRAPH STORAGE / PUBLICATION

RDF/OWL is the canonical semantic representation.

Support publication to:

- Apache Jena Fuseki
- optionally GraphDB/Stardog adapters later

Use named graphs and versioned graph IRIs.

Provide publication API with dry-run mode.

Never overwrite a published ontology without versioning.

Neo4j may be supported as a projection/export target, but property graph representation must not become the ontology source of truth.

---

30 28. RELEASE PACKAGE

Every approved ontology version should generate:

```
release-X.Y.Z/
  ontology.ttl
  ontology.owl
  ontology.jsonld
  shapes.ttl
  taxonomy.ttl
  provenance.ttl
  ontology-ir.json
  queries/
  examples/
  documentation/
  validation/
  release-certificate.json
  manifest.json
```

manifest.json must contain SHA-256 hashes of artifacts.

Later add signing support.

---

31 29. TRAINING DATA ARCHITECTURE

The proprietary model is a later stage, not the first dependency.

Capture high-quality expert-approved examples.

Training families:

```
requirements → normalized requirements
requirements → concepts
requirements → competency questions
concepts → vocabulary reuse decisions
requirements + CQ → ontology architecture
```

architecture → OWL OIR
business rules → SHACL OIR
CQ → SPARQL
bad ontology → findings
findings → repaired ontology
ontology → documentation
version A + B → semantic diff report

Each record must retain:

source provenance
license
training permission
domain
expert reviewer
validation result
quality score
model/prompt version

---

32 30. MODEL TRAINING PIPELINE

Use stages:

Stage 0: Foundation model + tools + RAG
Stage 1: SFT OntologyLM
Stage 2: Preference optimization
Stage 3: Semantic-validator reinforcement learning

Do not train a foundation model from scratch initially.

Use LoRA/QLoRA where appropriate.

Keep model provider abstract.

Support:

- external frontier LLM
- self-hosted model
- vLLM
- future proprietary OntologyLM adapter

Never couple core product behavior to one provider.

---

33 31. SFT RECORD SCHEMA

Implement typed records similar to:


```
{
  "record_id": "SFT-000001",
  "task_type": "requirements_to_ontology_ir",
  "input": {
    "business_markdown": "...",
    "domain": "insurance",
    "domain_pack": "insurance-v1"
  },
  "context": {
    "approved_terms": [],
    "retrieved_standards": []
  },
  "output": {
    "ontology_ir": {}
  },
  "validation": {
    "schema_valid": true,
    "logical_consistent": true,
    "shacl_valid": true,
    "cq_coverage": 1.0
  },
  "provenance": {
    "reviewed": true,
    "license": "internal-training"
  }
}
```


---

34 32. PREFERENCE RECORDS

Example:


```
{
  "input": {
    "requirement": "A broker sells policies for insurers."
  },
  "chosen": {
    "relation": "sellsPolicy"
  },
```

```
  "rejected": {
    "subClassOf": "Policy"
  },
  "reason": [
    "Broker is not a type of Policy.",
    "The requirement expresses a relationship."
  ]
}
```

Capture accepted/rejected repair proposals as preference data when legally permitted.

---

35 33. SEMANTIC REWARD FUNCTION

Implement modular reward calculation.

Suggested components:

syntax validity	0.10
logical consistency	0.20
CQ coverage	0.20
SHACL correctness	0.15
pattern quality	0.10
provenance	0.05
documentation	0.05
vocabulary reuse	0.05
expert semantic rating	0.10

Suggested penalties:

RDF syntax error	-1.00
logical inconsistency	-1.00
unsatisfiable required class	-0.50
unsupported hallucinated term	-0.20
duplicate concept	-0.10
untraceable axiom	-0.10
bad equivalence assertion	-0.30
failed required CQ	-0.20

Hard cap:

```
if not rdf_parse:
    reward = -1.0
if not logical_consistent:
    reward = -1.0
```

36 34. EVALUATION / BENCHMARK SUITE

Build a proprietary benchmark early.

Categories:

- subclass correctness
- relationship correctness
- role modeling
- event modeling
- OWL restriction correctness
- taxonomy vs ontology distinction
- SHACL correctness
- competency-question coverage
- vocabulary reuse
- duplicate detection
- repair
- semantic diff
- provenance
- documentation

Each benchmark case should include:

business input
gold OIR
gold ontology
acceptable alternatives
known-invalid alternatives
expected validation outcomes
expert explanation

Do not split near-identical ontology versions between training and test sets.

Split evaluation by project/domain/customer boundary to minimize leakage.

---

37 35. OBSERVABILITY

Instrument every generation run.

Use OpenTelemetry.

Capture:

trace ID
run ID
organization ID
project ID
agent start/end
model name/model version
prompt version
retrieved context IDs
token usage
latency
validation timings
repair count
quality score
human edits
publication status

Do not persist hidden chain-of-thought.

Persist structured decisions, rationales, evidence, and tool results instead.

---

38 36. MODEL / PROMPT VERSIONING

Track:

model_id
base_model
adapter
training_dataset_version
training_code_version
prompt_version
benchmark_results
release_date
license
approved_use

Prompts must be version-controlled files, not hard-coded anonymous strings across the application.

---

39 37. INFRASTRUCTURE

Provide Docker Compose for local development.

Minimum services:

web/studio
api
worker
postgres
redis
fuseki
minio
reasoner-service if separated
model-server optional GPU profile
observability stack optional profile

Kubernetes production topology must allow independent scaling of:

- API
- worker
- GPU model serving
- semantic validation/reasoning

Use Helm or Kustomize plus Terraform where appropriate.

---

40 38. CI/CD

Every pull request must run:

Python lint
TypeScript lint
Python type checks
TypeScript type checks
unit tests
API tests
OIR schema generation consistency
sample OIR compilation
RDF parse
SHACL fixture tests
reasoning smoke test when tool available
security scan
container build

Main branch additionally:

integration tests
semantic benchmark subset

DB migration tests
container scan
SBOM generation

Release branch:

full semantic benchmark
load/performance tests
backup/restore validation
migration rehearsal
signed/reproducible release artifacts

Never disable a test to force CI green.

---

41 39. TESTING REQUIREMENTS

Create:

41.1 Unit tests

- OIR validation
- compiler output
- namespace handling
- property-chain compilation
- restriction compilation
- release policy
- reward function
- state transitions

41.2 Negative OIR tests

- missing superclass
- unknown property
- duplicate IRI
- invalid inverse
- invalid SHACL path
- malformed agent JSON

41.3 Semantic tests

- expected entailment
- forbidden entailment
- consistency
- unsatisfiable-class test
- positive SHACL fixture

- negative SHACL fixture
- CQ ASK/SELECT

41.4 Security tests

- unauthorized project access
- tenant isolation
- invalid file upload
- authorization on publish

41.5 E2E

Markdown
→ requirements
→ domain analysis
→ CQs
→ reuse
→ conceptual architecture
→ OWL OIR
→ SHACL OIR
→ compile
→ reason
→ SHACL
→ CQ
→ quality score
→ review-ready release

---

42 40. FIRST REQUIRED E2E TEST - INSURANCE CLAIMS

Input:

Domain
Insurance Claims

Business Rules
Every Claim relates to a Policy.
A Claim may contain multiple Claim Payments.
Every Claim has exactly one claim identifier.
Claim Payment must contain a monetary amount.

Expected concepts:

Claim
Policy
ClaimPayment

Expected object properties:

Claim --againstPolicy--> Policy
Claim --hasPayment--> ClaimPayment

Expected datatype properties:

Claim --claimIdentifier--> xsd:string
ClaimPayment --paymentAmount--> xsd:decimal

Critical forbidden result:

Claim rdfs:subClassOf Policy

Add automated assertion that this incorrect subclass relation never appears.

Generate positive and negative fixtures and execute CQs.

---

43 41. CODE QUALITY RULES

Code must be:

- modular
- typed
- documented
- lint clean
- testable
- dependency-injected where useful
- configuration-driven
- secure by default
- failure-explicit

Do not:

- swallow exceptions
- silently ignore unsupported ontology constructs
- hide failing validators
- hard-code tenant IDs
- hard-code secrets
- embed provider-specific assumptions everywhere
- use huge monolithic files

- use uncontrolled Any types for semantic contracts

---

44 42. ERROR HANDLING

Define structured error codes, e.g.:

OIR_SCHEMA_INVALID
OIR_REFERENCE_INVALID
RDF_PARSE_FAILED
OWL_PROFILE_VIOLATION
ONTOLOGY_INCONSISTENT
UNSATISFIABLE_CLASS
SHACL_VIOLATION
CQ_FAILED
REPAIR_EXHAUSTED
INVALID_STATE_TRANSITION
PUBLISH_NOT_AUTHORIZED
DOMAIN_PACK_NOT_FOUND
MODEL_RESPONSE_INVALID

Return correlation/request IDs.

---

45 43. DOCUMENTATION

Generate and maintain:

docs/architecture.md
docs/semantic-methodology.md
docs/ontology-ir.md
docs/agents.md
docs/domain-packs.md
docs/validation.md
docs/governance.md
docs/training.md
docs/deployment.md
docs/security.md
docs/production-readiness.md
docs/api.md
docs/user-guide.md

Architecture diagrams must be included.

46 44. PRODUCTION READINESS

Before declaring v1 release-ready, verify:

- real OWL reasoner is installed and required
- real SHACL engine is installed and required
- external ontology licenses are tracked
- customer content is isolated
- RBAC works
- audit events work
- publication is human-controlled
- backups work
- restore works
- secrets are externalized
- TLS enabled
- monitoring enabled
- alerting enabled
- resource limits configured
- migrations tested
- image versions pinned
- vulnerability scans pass
- SBOM generated
- tenant isolation tests pass

Do not label development fallback validators as production semantic validation.

---

47 45. PRODUCT SUCCESS METRICS

Track:

RDF syntax success
OIR validity
logical consistency
unsatisfiable classes
CQ coverage
SHACL success
provenance coverage
definition coverage
expert acceptance rate

repair success rate
hallucination rate
expert editing time saved
ontology release cycle time

The primary business KPI should be:

> **How much ontology-engineer and SME time does the platform save while maintaining or improving semantic quality?**

---

48 46. INITIAL DELIVERY ROADMAP

Use the following as a baseline but refine it after repository inspection.

48.1 Milestone 1 - Semantic Vertical Slice

- Markdown ingestion
- Agents 1-7
- OIR
- deterministic compiler
- insurance E2E
- RDF parse
- CQ execution

48.2 Milestone 2 - Formal Validation

- HermiT / ELK
- pySHACL
- OWL profile validator
- positive/negative fixtures
- release policies

48.3 Milestone 3 - Product UX

- project dashboard
- Requirements Studio
- CQ Studio
- Ontology Graph Studio
- Term Inspector
- Quality Center

48.4 Milestone 4 - Governance

- users/organizations
- RBAC
- reviews

- semantic diff
- versioning
- release package
- audit

48.5 Milestone 5 - Multi-Domain

- domain-pack framework
- finance skeleton
- healthcare skeleton
- pharma skeleton
- manufacturing skeleton
- supply-chain skeleton

48.6 Milestone 6 - Proprietary Intelligence

- training-data pipeline
- expert correction capture
- benchmark suite
- SFT pipeline
- preference pipeline
- semantic reward environment

48.7 Milestone 7 - Enterprise Hardening

- Kubernetes
- OIDC/SAML
- observability
- backup/recovery
- security scans
- VPC/on-prem deployment

---

49 47. REQUIRED CLAUDE WORK STYLE

Claude must not immediately start writing hundreds of files blindly.

Use this operational sequence:

1. Inspect
2. Research
3. Produce gap analysis
4. Produce implementation plan
5. Implement one vertical slice
6. Run tests
7. Fix failures

8. Continue incrementally
9. Keep architecture documentation current
10. Produce final verification report

At each milestone:

- show files changed
- show tests added
- show tests run
- show failures and fixes
- show remaining risks

Never claim success without executing verification commands.

---

50 48. ACCEPTANCE CRITERIA FOR COMMERCIAL V1

Commercial v1 is considered ready only when a user can provide a domain Markdown file and complete this journey:

Upload business-domain Markdown
↓
AI extracts requirements
↓
Human can correct/approve requirements
↓
AI creates competency questions
↓
AI identifies reusable vocabularies
↓
AI produces conceptual ontology architecture
↓
AI produces typed OIR
↓
Deterministic compilers generate formal artifacts
↓
Real reasoner validates ontology
↓
Real SHACL validates fixtures/data
↓
CQs execute
↓
Quality score calculated

↓
Repair loop runs when necessary
↓
Ontology engineer reviews
↓
Domain SME approves
↓
Release package generated
↓
Ontology published to versioned RDF store

The release package must be reproducible and traceable back to business requirements.

---

51 49. DEFINITION OF DONE

The implementation is NOT done because the UI looks good or because the LLM produces plausible ontology text.

It is done when:

- tests pass
- negative tests prove bad semantics fail
- reasoner is executed
- SHACL is executed
- business requirements trace to ontology entities
- CQs execute
- release policy blocks invalid ontology
- invalid governance transitions fail
- AI cannot self-publish
- domain packs can be added without modifying core code
- multi-tenant isolation is tested
- release artifacts are versioned and hashed
- model outputs are typed and fail closed
- production deployment is documented

---

52 50. FINAL INSTRUCTION TO CLAUDE

Build this system as if it will be sold to regulated enterprises.

Favor:

formal semantics over plausible text
traceability over hidden reasoning
determinism over randomness
validation over confidence scores
modularity over model lock-in
versioning over overwrite
human governance over autonomous publishing
standards over proprietary shortcuts

The product's long-term competitive moat is NOT access to a foundation model.

The moat is:

expert-reviewed ontology dataset
+ Ontology IR
+ semantic compiler
+ validation engine
+ repair dataset
+ domain packs
+ benchmark suite
+ governance workflow
+ proprietary OntologyLM training pipeline

Do not skip any requirement in this master specification.

When constraints conflict, preserve semantic correctness, auditability, security, maintainability, and enterprise governance in that order.

At completion, provide:

1. final repository tree
2. architecture summary
3. implementation status by requirement
4. test results
5. benchmark results
6. security status
7. known limitations
8. production deployment steps
9. roadmap for the next release
10. a demonstration using the insurance-domain E2E example

Begin by inspecting the repository and producing the research + gap-analysis + implementation plan. Do not begin major implementation until that plan is complete.