Lambda BER Schema

Technical Deep Dive

LinkML Schema Development and Repository Architecture

Agenda

1. LinkML Schema Architecture
2. Schema Structure and Design
3. Code Generation and Compilation
4. Repository Structure
5. Development Workflow
6. Contributing to the Repository
7. Testing and Validation
8. Advanced Topics

Part 1: LinkML Architecture

What is LinkML?

Linked Data Modeling Language - A framework for defining data models:

- **Human-readable**: YAML-based schema definitions
- **Machine-actionable**: Generates code in multiple languages
- **Interoperable**: Bridges JSON, RDF, and relational models
- **Extensible**: Plugin architecture for custom generators

Think of it as "schema definition as code"

LinkML Metamodel

Every LinkML schema is itself an instance of the LinkML metamodel:

```yaml
classes:
  Sample:
    description: A biological sample
    attributes:
      sample_code:
        range: string
        required: true
      sample_type:
        range: SampleTypeEnum
        required: true
```

Metamodel concepts: `classes`, `slots`, `types`, `enums`

Why LinkML for Structural Biology?

- **Multi-format support**: YAML, JSON, RDF/TTL all from one schema
- **Rich semantics**: Links to ontologies (GO, ChEBI, UniProt)
- **Validation**: Built-in constraints and validation rules
- **Documentation**: Auto-generated docs in multiple formats
- **Python integration**: Native dataclasses for easy coding
- **Tooling ecosystem**: Validators, converters, visualizers

LinkML vs Other Approaches

Approach	Pros	Cons
JSON Schema	Simple, widely adopted	Limited semantics, manual docs
OWL/RDF	Rich semantics	Complex, steep learning curve
Protocol Buffers	Fast serialization	Limited to specific languages
LinkML	Best of all worlds	Newer, smaller ecosystem

Part 2: Schema Structure

lambda-ber-schema Architecture

Located at: `src/lambda_ber_schema/schema/lambda-ber-schema.yaml`

```yaml
id: https://w3id.org/lambda-ber/lambda-ber-schema
name: lambda-ber-schema
description: Schema for structural biology imaging data

prefixes:
  linkml: https://w3id.org/linkml/
  schema: http://schema.org/

imports:
  - linkml:types

classes:
  # Core classes defined here...
```

Core Class Hierarchy

Container Classes:

- `Dataset` (root) → contains `Study` objects
- `Study` → contains all experimental data

Entity Classes:

- `Sample`, `SamplePreparation`
- `Instrument` (+ subclasses for each technique)
- `ExperimentRun`, `WorkflowRun`
- `DataFile`, `Image` (+ specialized image types)

Supporting Classes:

- `MolecularComposition`, `BufferComposition`

Slot Definitions

Slots are reusable attributes defined separately:

```yaml
slots:
  sample_code:
    description: Unique identifier for the sample
    range: string
    required: true
    identifier: true

  temperature:
    description: Temperature in Kelvin
    range: float
    unit:
      ucum_code: K
```

Slots can be inherited and reused across classes.

Enumerations (Controlled Vocabularies)

```yaml
enums:
  TechniqueEnum:
    permissible_values:
      cryoem:
        description: Cryo-electron microscopy
      xray:
        description: X-ray crystallography
      saxs:
        description: Small angle X-ray scattering
      waxs:
        description: Wide angle X-ray scattering
      sans:
        description: Small angle neutron scattering
```

Provides controlled vocabularies throughout the schema.

Design Pattern: Inlined Collections

```yaml
classes:
  Study:
    attributes:
      samples:
        range: Sample
        multivalued: true
        inlined: true
        inlined_as_list: true
```

- `multivalued: true` → array/list
- `inlined: true` → embed objects directly
- `inlined_as_list: true` → JSON/YAML array format

Design Pattern: Identifiers and References

```yaml
classes:
  Sample:
    attributes:
      sample_code:
        identifier: true  # Primary key

  ExperimentRun:
    attributes:
      sample_id:
        range: Sample  # Foreign key reference
        required: true
```

Supports both embedding and referencing.

Design Decision: Date/Time Handling

Problem: Different systems use different datetime formats

Solution: Use `string` type with ISO 8601 recommendation

```
slots:
  collection_date:
    range: string  # Not 'date' type
    description: Date in ISO 8601 format (YYYY-MM-DD)
```

More forgiving for real-world data integration.

Design Decision: Scientific Notation

Problem: YAML scientific notation (2.0e12) causes JSON Schema issues

Solution: Document that numbers should be written out fully

```yaml
# Avoid:
particle_count: 2.0e12

# Use instead:
particle_count: 2000000000000
```

Part 3: Code Generation

The gen-project Command

Core compilation command:

```
uv run gen-project \
  --config-file config.yaml \
  src/lambda_ber_schema/schema/lambda-ber-schema.yaml \
  -d assets
```

Generates all downstream artifacts from the schema.

What Gets Generated?

From a single YAML schema:

1. **Python**: Dataclasses (`assets/lambda_ber_schema.py`)
2. **JSON Schema**: Validators (`assets/jsonschema/`)
3. **Documentation**: HTML/Markdown (`assets/docs/`)
4. **GraphQL**: API schema (`assets/graphql/`)
5. **OWL**: Ontology (`assets/owl/`)
6. **SHACL**: RDF shapes (`assets/shacl/`)
7. **JSON-LD Context**: Linked data (`assets/jsonld/`)
8. **SQL DDL**: Database schema (`assets/sqlschema/`)

Generation Configuration

`config.yaml` controls what gets generated:

```yaml
generators:
  - python
  - json_schema
  - owl
  - graphql
  - shacl
  - markdown_docs

generator_args:
  python:
    package: lambda_ber_schema

  markdown_docs:
    directory: docs
```

Python Dataclasses Example

Generated from schema:

```python
from dataclasses import dataclass
from typing import Optional, List

@dataclass
class Sample:
    sample_code: str
    sample_type: str
    sample_name: Optional[str] = None
    molecular_composition: Optional[MolecularComposition] = None
    # ... more fields
```

Ready to use in Python code!

Using Generated Python Classes

```python
from lambda_ber_schema import Sample, MolecularComposition

# Create a sample
sample = Sample(
    sample_code="sample-001",
    sample_type="protein",
    sample_name="My Protein",
    molecular_composition=MolecularComposition(
        proteins=["UniProt:P12345"]
    )
)

# Serialize to dict
sample_dict = asdict(sample)

# Write to YAML/JSON
import yaml
with open('sample.yaml', 'w') as f:
    yaml.dump(sample_dict, f)
```

JSON Schema Validation

Generated JSON Schema can validate data:

```
# Using linkml-validate
uv run linkml-validate \
  -s src/lambda_ber_schema/schema/lambda-ber-schema.yaml \
  tests/data/valid/Sample-protein.yaml

# Or using JSON Schema directly
jsonschema -i data.json assets/jsonschema/lambda-ber-schema.json
```

Auto-Generated Documentation

Schema docs generated at `assets/docs/`:

- **Index**: Overview of all classes
- **Class pages**: Detailed documentation per class
- **Enum pages**: Controlled vocabulary definitions
- **Type pages**: Type definitions and constraints

Formatted as markdown and/or HTML.

Part 4: Repository Structure

Directory Layout

```
lambda-ber-schema/
├── src/lambda_ber_schema/
│   ├── schema/
│   │   └── lambda-ber-schema.yaml    # Source schema
│   └── datamodel/
├── assets/                           # Generated outputs
├── tests/data/valid/                 # Example data files
├── docs/                             # Documentation
│   ├── slides/                       # Presentation slides
│   ├── spec.md                       # Specification
│   └── background/                   # Research docs
├── config.yaml                       # Generation config
├── pyproject.toml                    # Python project config
└── justfile / project.justfile       # Build automation
```

Source Schema Location

Primary source of truth:

`src/lambda_ber_schema/schema/lambda-ber-schema.yaml`

All other files are generated from this schema.

Never edit generated files directly!

Edit the schema, then regenerate.

Assets Directory

Auto-generated, do not edit:

```
assets/
├── lambda-ber-schema.py                # Python dataclasses
├── jsonschema/
│   └── lambda-ber-schema.json
├── docs/                               # Generated documentation
├── graphql/
│   └── lambda-ber-schema.graphql
├── owl/
│   └── lambda-ber-schema.owl.ttl
└── ... more formats
```

Test Data Organization

```
tests/data/valid/
├── Sample-protein.yaml
├── Sample-hetBGL.yaml
├── ExperimentRun-cryoet.yaml
├── WorkflowRun-3dclass.yaml
├── Dataset-berkeley-tfiid.yaml
└── ... more examples
```

Each file is:

1. A valid instance of the schema
2. Used for automated validation testing
3. Documentation by example

Documentation Structure

```
docs/
├── spec.md                          # Complete specification
├── background/                      # Research and context
│   ├── nexus.md
│   ├── mmcif.md
│   ├── empiar.md
│   └── ... more
├── slides/                          # Presentations
│   ├── overview.md
│   └── technical-overview.md
└── examples/                        # Analyzed examples
```

Part 5: Development Workflow

Setting Up Development Environment

```
# Clone the repository
git clone https://github.com/lambda-ber/lambda-ber-schema.git
cd lambda-ber-schema

# Install dependencies using uv
just install
# or directly:
uv sync --group dev

# Verify installation
uv run linkml-lint --version
```

Requires: Python 3.9+, uv, just (optional but recommended)

The Development Cycle

1. **Edit schema**: Modify `src/lambda_ber_schema/schema/lambda-ber-schema.yaml`
2. **Generate artifacts**: Run `just gen-project`
3. **Create/update examples**: Add test data to `tests/data/valid/`
4. **Validate**: Run `just test-examples`
5. **Test**: Run `just test` (pytest, mypy, formatting)
6. **Document**: Update docs as needed
7. **Commit**: Commit both schema and regenerated artifacts

Key justfile Targets

```
# Install dependencies
just install

# Generate all artifacts from schema
just gen-project

# Validate all example files
just test-examples

# Run full test suite
just test

# Generate documentation
just gendoc

# Serve docs locally
just serve
```

Schema Modification Example

Task: Add a new field to Sample class

```yaml
classes:
  Sample:
    attributes:
      # Existing fields...
      ph_value:  # NEW FIELD
        range: float
        description: pH of the sample buffer
        minimum_value: 0
        maximum_value: 14
```

Then: `just gen-project` to regenerate all artifacts

Adding a New Class

```yaml
classes:
  CryoGridPreparation:  # NEW CLASS
    is_a: SamplePreparation
    description: Details of cryo-EM grid preparation
    attributes:
      grid_type:
        range: GridTypeEnum
      blot_time:
        range: float
        unit:
          ucum_code: s
      blot_force:
        range: integer
```

Inheritance via `is_a` reuses parent class attributes.

Adding an Enumeration

```yaml
enums:
  GridTypeEnum:  # NEW ENUM
    permissible_values:
      quantifoil_r1.2_1.3:
        description: Quantifoil R1.2/1.3 holey carbon grids
      c_flat_1.2_1.3:
        description: C-flat 1.2/1.3 holey carbon grids
      ultrathin_carbon:
        description: Ultrathin continuous carbon grids
      graphene_oxide:
        description: Graphene oxide grids
```

Validation Rules

LinkML supports sophisticated validation:

```yaml
classes:
  Sample:
    attributes:
      concentration_value:
        range: float
        minimum_value: 0  # Must be non-negative

    rules:
      - preconditions:
          slot_conditions:
            concentration_value:
              required: true
        postconditions:
          slot_conditions:
            concentration_unit:
              required: true
        description: If concentration_value is provided, unit is required
```

Creating Example Data

Start with minimal required fields:

```yaml
# tests/data/valid/Sample-minimal.yaml
sample_code: "sample-min-001"
sample_type: "protein"
```

Then expand with optional fields for richer examples:

```yaml
sample_code: "sample-full-001"
sample_type: "protein_complex"
sample_name: "TFIID Complex"
molecular_composition:
  proteins:
    - "UniProt:P12345"
    - "UniProt:P67890"
buffer_composition:
  components:
    - name: "Tris-HCl"
      concentration_value: 50
```

Part 6: Contributing

Contribution Workflow

1. **Fork** the repository on GitHub
2. **Clone** your fork locally
3. **Create a branch** for your feature: `git checkout -b add-feature-x`
4. **Make changes** to the schema
5. **Regenerate**: `just gen-project`
6. **Test**: `just test`
7. **Commit** with descriptive message
8. **Push** to your fork
9. **Open a Pull Request**

What to Contribute

Schema Enhancements:

- New classes for additional techniques
- Additional attributes for existing classes
- Enhanced enumerations/vocabularies
- Validation rules and constraints

Examples:

- Real-world datasets
- Edge cases and variations
- Multi-technique integrative examples

Documentation:

Best Practices for Schema Changes

1. **Start small**: Incremental changes are easier to review
2. **Add examples**: Include test data demonstrating new features
3. **Document thoroughly**: Use `description` fields extensively
4. **Follow conventions**: Match existing naming patterns
5. **Test thoroughly**: Ensure all tests pass
6. **Explain the why**: PR description should explain motivation

Naming Conventions

Classes: `PascalCase` (e.g., `SamplePreparation`)

Slots: `snake_case` (e.g., `sample_code`)

Enums: `PascalCaseEnum` (e.g., `TechniqueEnum`)

Enum values: `snake_case` (e.g., `protein_complex`)

Files: `kebab-case` (e.g., `lambda-ber-schema.yaml`)

Code Review Process

Pull requests are reviewed for:

- **Schema validity**: Does it compile without errors?
- **Semantic correctness**: Do the definitions make sense?
- **Consistency**: Does it follow existing patterns?
- **Documentation**: Are descriptions clear and complete?
- **Examples**: Are there test cases?
- **Breaking changes**: Is backwards compatibility maintained?

Semantic Versioning

Schema versions follow semver (MAJOR.MINOR.PATCH):

- **PATCH**: Bug fixes, documentation improvements
- **MINOR**: New classes, attributes (backwards compatible)
- **MAJOR**: Breaking changes (renamed/removed elements)

Incremented in schema `version` field:

```
version: "1.2.0"
```

Part 7: Testing and Validation

Testing Strategy

Multiple layers of testing:

1. **Schema validation**: LinkML lints the schema itself
2. **Example validation**: Test data validates against schema
3. **Unit tests**: Python tests for custom logic
4. **Type checking**: mypy for Python type safety
5. **Format checking**: ruff for code style

All run via: `just test`

Schema Linting

Validate the schema against LinkML metamodel:

```
uv run linkml-lint \
  src/lambda_ber_schema/schema/lambda-ber-schema.yaml
```

Checks for:

- Syntax errors
- Invalid metamodel usage
- Dangling references
- Type mismatches

Example Validation

`linkml-run-examples` validates and converts example data:

```
uv run linkml-run-examples \
  -t yaml -t json -t ttl \
  -s src/lambda_ber_schema/schema/lambda-ber-schema.yaml \
  -e tests/data/valid \
  -d examples
```

For each example:

1. Validates against schema
2. Converts to multiple formats (YAML, JSON, RDF/TTL)
3. Outputs to `examples/` directory

Manual Validation

Validate individual files:

```
uv run linkml-validate \
  -s src/lambda_ber_schema/schema/lambda-ber-schema.yaml \
  tests/data/valid/Sample-protein.yaml
```

Useful for debugging specific instances.

Python Testing

pytest tests in `tests/` directory:

```python
import pytest
from lambda_ber_schema import Sample

def test_sample_creation():
    """Test creating a Sample instance."""
    sample = Sample(
        sample_code="test-001",
        sample_type="protein"
    )
    assert sample.sample_code == "test-001"
    assert sample.sample_type == "protein"
```

Run with: `just pytest`

Type Checking

mypy ensures type safety in Python code:

```
just mypy
# or directly:
uv run mypy src tests
```

Catches type errors before runtime.

Continuous Integration

GitHub Actions runs on every PR:

```
# .github/workflows/test.yml
- name: Run tests
  run: |
    uv sync --group dev
    just gen-project
    just test-examples
    just test
```

Ensures all contributions pass tests.

Part 8: Advanced Topics

Linking to External Ontologies

```yaml
classes:
  MolecularComposition:
    attributes:
      proteins:
        range: uriorcurie
        description: UniProt identifiers
        pattern: "^UniProt:\\w+$"
        multivalued: true
```

Enables semantic integration with external resources.

Units and Measurements

Using UCUM (Unified Code for Units of Measure):

```
slots:
  temperature:
    range: float
    unit:
      ucum_code: K

  wavelength:
    range: float
    unit:
      ucum_code: nm
```

Mixins for Reusable Patterns

```
classes:
  Timestamped:  # Mixin
    mixin: true
    attributes:
      created_at:
        range: string
      updated_at:
        range: string

  Sample:
    mixins:
      - Timestamped  # Inherits timestamp attributes
```

Slot Usage vs Definition

Define once, use many times:

```yaml
slots:
  sample_id:  # Slot definition
    range: Sample

classes:
  ExperimentRun:
    slot_usage:
      sample_id:  # Customize for this class
        required: true
        description: Sample used in this experiment
```

Conditional Validation Rules

```yaml
classes:
  ExperimentRun:
    rules:
      - preconditions:
          slot_conditions:
            technique:
              equals_string: "cryoem"
        postconditions:
          slot_conditions:
            instrument_id:
              range: CryoEMInstrument
```

Technique-specific validation.

Performance Considerations

For large datasets:

- **Use references** instead of inlining for large collections
- **Lazy loading**: Load objects on demand
- **Streaming validation**: Validate incrementally
- **Database backends**: Use SQL schema generation for persistence

Working with RDF/Linked Data

```
# Generate RDF from YAML instance
uv run linkml-convert \
  -s schema.yaml \
  -t rdf \
  input.yaml -o output.ttl

# Query with SPARQL
# Load into triple store (e.g., Apache Jena)
```

Bridge to semantic web ecosystem.

Schema Evolution Strategies

Adding fields: Always backwards compatible

Renaming fields: Use `deprecated` and `aliases`

```yaml
slots:
  sample_id:
    aliases:
      - sample_identifier  # Old name
    deprecated: "Use sample_id instead"
```

Removing fields: Deprecate first, remove in major version

Custom Generators

Extend LinkML with custom generators:

```
from linkml.generators import Generator

class CustomGenerator(Generator):
    def serialize(self):
        # Your custom output format
        pass
```

Or use Python scripts to post-process generated artifacts.

Integration Patterns

API Integration:

- Generate JSON Schema → OpenAPI spec
- Use GraphQL schema for API

Database Integration:

- Generate SQL DDL
- ORM mapping with generated Python classes

Data Lake Integration:

- Validate incoming data
- Schema-on-read with LinkML

Documentation Best Practices

Every class and slot should have:

```yaml
classes:
  Sample:
    description: >
      A biological sample that is the subject of study.
      This could be a purified protein, protein complex,
      nucleic acid, or other biological material.
    comments:
      - Samples should have unique identifiers within a study
    see_also:
      - https://example.org/sample-preparation-guide
```

Future LinkML Features

Upcoming capabilities:

- **Conditional logic**: More sophisticated rules engine
- **Computed fields**: Derived attributes
- **Multi-schema imports**: Compose from multiple schemas
- **Enhanced validation**: Custom validators
- **Performance**: Faster generation and validation

Resources and Community

LinkML Resources:

- Documentation: https://linkml.io
- GitHub: https://github.com/linkml
- Slack: linkml.slack.com

This Project:

- GitHub: https://github.com/lambda-ber/lambda-ber-schema
- Issues: Report bugs and request features
- Discussions: Ask questions and share ideas

Getting Help

Schema Questions:

- Open a GitHub issue
- Check existing examples
- Review LinkML documentation

LinkML Questions:

- LinkML Slack channel
- LinkML GitHub discussions
- Stack Overflow (tag: linkml)

Recap: Key Takeaways

1. **LinkML**: Powerful schema framework for biomedical data
2. **Single source**: One YAML schema → many formats
3. **Validation**: Built-in data quality checks
4. **Generation**: Automated code and documentation
5. **Testing**: Multi-layered validation strategy
6. **Contributing**: Fork, modify, test, PR
7. **Community**: Open and collaborative

Next Steps

For Users:

1. Explore examples in `tests/data/valid/`
2. Validate your own data
3. Provide feedback on schema coverage

For Contributors:

1. Read CONTRIBUTING.md (if exists)
2. Join GitHub discussions
3. Start with documentation improvements
4. Progress to schema enhancements

Thank You!

lambda-ber-schema: Structured metadata for structural biology

Questions? Open a GitHub issue or discussion

Want to contribute? Fork and submit a PR

Let's build better data standards together!