DUNE Data Model in the Phlex Era

Toward a *lingua franca* for our algorithms

Brett Viren

May 20, 2026

Topics

- Why DUNE needs a shared data model in Phlex
- Lessons from the past: nativist vs. generalist strategies
- The problem space: what “data model” actually means
- A concrete but partial proposal: decide **format** now, defer **schema** choices for later
- What we may decide “today”; what work remains

The coordination problem

Phlex executes algorithms from many independent packages:

Wire-Cell	drift, det sim, signal processing, imaging, clustering
Pandora	multi-algorithm pattern recognition
SPINE	deep-learning reconstruction
…	and many individuals' efforts

Each package has its own data model, some directly use LArSoft's.
Without a shared model, every type of producer-consumer pairing needs a converter:

$$N_{\text{producers}} \times M_{\text{consumers}} \text{ converters}$$

Data model lessons from art/LArSoft

What worked

- Expressive, common “vocabulary” of “nouns” (data products) and “verbs” (art modules).
- Made data products that are independent from art (I think generally true?).
- Reduced but did not eliminate need for data product converters.

What did not work

- Data products do not provide the right “shape” for all needs
  - WCT’s 3D imaging and clustering have no match in LArSoft.
- C++ structures with no direct file I/O support

Key lessons

- Keep data model independent from any particular framework.
- Do not lock-in too many specifics data features from the start.
- Extensibility as first-class feature and have an eye open for file I/O.

Two approaches to interoperability

Nativist: wrap native types at the exchange boundary

```
struct Frame { WireCell::IFrame::pointer ptr; };  // thin wrapper
```

Good perfect fidelity and efficiency (locally), easy to add new algorithm.

Bad consumers must depend on the producer's library or $N \times M$ converter problem.

Generalist: define shared types, any converts are an algorithm's problem

Good common vocabulary, decouples inter-package dependencies

Bad wrong choices marginalize packages; still need some converters.

For best results we need both and the key is where to draw the lines

Major packages will not adopt a new internal data model.

- Some nativism is inevitable and some converters are always needed.

A **shared data model** collapses the combinatoric:

$$N \times M \longrightarrow N + M \text{ converters}$$

The shared model must be **expressive enough** not to marginalize any package.

- But, still lock down enough details to provide a useful *lingua franca*.

The choice of shared model is therefore a high-stakes design decision.

The full problem is large. Let's break it into two layers.

Format

The low-level **in-memory representation** — how bits are arranged in memory Concrete, technology-bounded, implementable today.

Schema

The **semantic layer** — field names, units, relationships, conventions. Best to gain consensus between developers of producer/consumer algorithm pairs with eye toward future novel work.

The full problem space is rather larger

Concern	Key questions
Format	What in-memory representations are **sufficient**?
Schema	What fields, units, relationships describe our data?
File I/O	HDF5? ROOT? Zarr? Custom scripts?
CPU/GPU	Can the same object live on device and host memory?
C++/Python	Zero-copy interop or a serialization boundary?
Phlex layers	How does data fit Phlex's **DFP graph** and **data-layer hierarchy**?
Versioning	How do schema evolve without breaking existing readers?
Provenance	What metadata must travel with every data product?

This talk addresses Format. We try to keep in mind the larger problems but defer them.

The key move: decouple format from schema

Most data model designs conflate the two.

- **Schema** changes often: new detector, new algorithm, new physics.
- **Format** can be stable: it is a small vocabulary of representations.

Decoupling means:

- Schema translation becomes *reinterpretation of the same memory* — not data copying.
- Adding a new schema does not require a new format library.
- Some problems can be solved at the simpler format level (eg, file I/O).

Proposal: standardize on a small set of general formats now; let schema emerge from algorithm community discussions.

Format primitive 1: **structured objects**

JSON-like nested key-value objects with typed scalars and arrays.

- Naturally expresses heterogeneous, hierarchical data.
- **Use for:** metadata, geometry, configuration, provenance records.
- C++ candidates: `nlohmann/json`, `boost::json`; Python: native `dict/json`.
- Easy transfer between C++ and Python via serialization. Some candidates allow direct sharing of object memory.

Format primitive 2: **rectangular arrays**

Typed, shaped, strided dense arrays — the Numpy array model.

- **Use for:** ADC waveforms, deposition clouds, images, feature tensors, point clouds.
- Zero-copy C++/Python transfer via the buffer protocol and DLPack standard.
  - `Eigen::Map`, `torch::from_blob` map existing memory without copying.
- Candidate solving other problems: `libtorch` (PyTorch C++ API).
  - CPU/GPU transparent: same type, same code, different device.
  - Already required for ML inference and GPU-accelerated Wire-Cell (SPNG).
  - True, it is a large dependency — but one we carry regardless.
- More discussion, research and testing before a final decision.

Format primitive 3: **sparse and irregular arrays**

Variable-length, nested, irregular structure — the “awkward” array model.

- **Use for:** hit lists, cluster contents, track segments, signal-processed wires.
  - WCT signal processing output is inherently sparse.
  - Graphs fit naturally as nested irregular arrays.
- **Leading candidate:** the Scikit-HEP `awkward` library.
  - Python-native with `awkward-cpp` for zero-copy C++ interop.
  - Columnar layout: efficient for bulk operations over collections.
  - Commonly used in the HEP Python ecosystem, less so on C++ side.

Note

Arguably, awkward array types are a combination of structured object and rectangular array types. It could be emitted though it provides representations that are closer to working memory and may avoid the need for some conversions. The cost/benefit needs discussion.

Every DUNE data type fits these three primitives

Data type	Format	Notes
ADC waveforms, images	Rectangular	(channels $\times$ ticks)
Simulated depositions	Rectangular	(N deposits $\times$ 7+4 fields)
Signal-processed signals	Sparse/awkward	zero-suppressed waveform fragments
Hit/cluster lists	Sparse/awkward	variable hits per cluster
Track segments	Sparse/awkward	variable points per track
ML feature tensors	Rectangular	CPU or GPU, same type
Run/event/geometry metadata	Structured object	JSON-like, hierarchical
Provenance records	Structured object	attached to any data product

Again, sparse/awkward types can reduce to rectangular array and/or structured objects.

Proposed package architecture

A layered factoring keeps format independent of framework and application.

1. **Format libraries** (no Phlex, no application dependency)
   - `dune-data-model-cpp` — C++ **format types** and utilities
   - `dune-data-model-python` — Python **format** equivalent, zero-copy interop (goal)
   - `dune-data-model-transfer` — cross-language transfer (depends on both)
2. **Phlex integration layer** `phlex-dune-data`
   - Thin wrappers to expose **format types** to the Phlex graph.
   - Depends on format libs + Phlex. **Does not** depend on any application lib.
3. **Application converters** (one per major package)
   - `{wire-cell,pandora,spine}-dune-data-model`
   - Each depends only on its native library + one format lib. No cross-dependencies.
4. **Schema libraries** (future) overlay specific structure types on format types.
   - Some *lingua franca*, some package-specific.

What remains open

Specifying *format* is necessary but not sufficient. Three large open problems:

Schema

What fields, units, and relationships constitute, e.g., a “hit” or a “cluster”? Needs a working group with representatives from WCT, Pandora, SPINE, and the ND. Wrong early commitments here replicate the LArSoft mistake.

File I/O and serialization

HDF5 (`HighFive`)? ROOT `RNTuple`? Zarr? The format choice must be I/O-friendly. “Official” files and “custom” analysis scripts need different strategies.

Versioning and provenance

How do schema evolve without breaking existing analysis code? What provenance metadata must every data product carry? These policies must be defined before the first schema is frozen.

Summary

We propose to decide “now” at least in principle

- Use a **three-primitive format vocabulary**: structured objects, rectangular arrays, sparse/awkward arrays.
- Adopt a **layered package factoring** that isolates format from framework and application.
- Straw-person choice of format-level technology candidates: `libtorch` / `awkward` / JSON library.

We defer, pending community input

- **Schema** definitions for all data types (needs a working group).
- **File I/O** strategy: format choice, ROOT vs.\ HDF5 vs.\ Zarr.
- **Versioning policy** and provenance metadata requirements.
- **Phlex data-layer** specifics (frames, products, graph hierarchy).

FIN

Backups

libtorch as the rectangular array candidate

Pros

- CPU and GPU tensors share the same type: `torch::Tensor`.
- Mature, well-maintained, large community (PyTorch).
- Already needed for ML inference and Wire-Cell SPNG.
- DLPack interop with NumPy, JAX, CuPy.

Cons

- Large (but required) dependency.
- C++ API is less ergonomic than Python; documentation lags (but LLMs know it).

Alternative: decouple GPU from the format layer — use `xtensor` / `Eigen` on CPU and defer GPU to algorithm-internal tensors. Simpler but loses CPU/GPU transparency at the exchange boundary.

Graph and point cloud representations

Graphs and point clouds are often treated as special cases, but both reduce to arrays.

Point cloud

A rectangular array of shape (N, d) where N is the number of points and d is the number of spatial (or feature) dimensions. No special format needed.

Graph

Two representations:

- **Adjacency matrix**: rectangular array, shape (N, N). Dense; impractical for sparse graphs.
- **Edge list**: rectangular array of shape $(2, E)$ for E edges (COO format). Sparse; standard in PyTorch Geometric and DGL.

Node and edge **features** are additional rectangular arrays aligned by index. Variable-length neighbor lists fit naturally as awkward arrays.