LHCb VELO Toy Model

Repository Restructuring & Data-Structure Design

George William Scriven

Maastricht University

UHasselt

Nikhef

February 2026

Outline

Why Restructure?

Old repository pain-points

- Flat directory — 8 files, no sub-packages
- `StateEventGenerator` is a *god class* (config + propagation + noise + data store)
- Visualisation baked into data classes
- Wildcard imports (`from ... import *`)
- Tight coupling: Hamiltonian typed to accept the *generator*, not a data object
- No type aliases, no protocols, no `py.typed`

Goals for the new layout

- Clear separation of concerns: *generation* → *solvers* → *analysis*
- Immutable-by-convention dataclasses with ID cross-references
- Geometry as a pluggable ABC
- Segments computed *on-demand*, not stored
- Full type coverage (`py.typed` marker)
- Ready for quantum solver extensions (HHL)

Old vs. New: Directory Tree

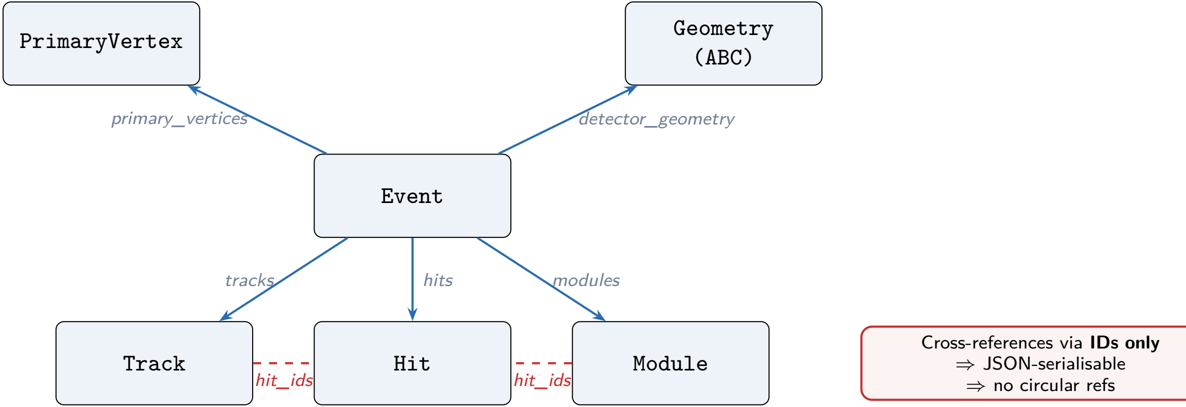
Old — flat

```
LHCB_Velo_Toy_Models/
  __init__.py
  state_event_model.py
  state_event_generator.py
  hamiltonian.py
  simple_hamiltonian.py
  simple_hamiltonian_fast.py
  simple_hamiltonian_cpp.py
  toy_validator.py
  lhcb_tracking_plots.py
```

New — modular

```
src/lhcb_velo_toy/
  core/            types.py
  generation/
    geometry/      base.py, plane.py,
                   rectangular_void.py
    entities/      hit.py, track.py,
                   module.py, event.py, ...
    generators/    state_event.py
  solvers/
    hamiltonians/ base.py, simple.py, fast.py
    classical/     solvers.py
    quantum/       hhl.py, one_bit_hhl.py
    reconstruction/
                   segment.py, track_finder.py
  analysis/
    validation/    match.py, validator.py
    plotting/      event_display.py,
                   performance.py
```

Core Data Classes (Generation Layer)



Key change: old code stored *object references* (e.g. `Track.hits: list[Hit]`). New code stores **ID lists** (e.g. `Track.hit_ids: list[int]`). Lookup is via `Event.get_hit(hit_id)`.

Dataclass Details

Hit

```python
@dataclass
class Hit:
    hit_id:    HitID
    x:         float
    y:         float
    z:         float
    module_id: ModuleID
    track_id: TrackID = -1
```

Track

```python
@dataclass
class Track:
    track_id: TrackID
    pv_id:    PVID = 0
    hit_ids: list[HitID]
```

Module

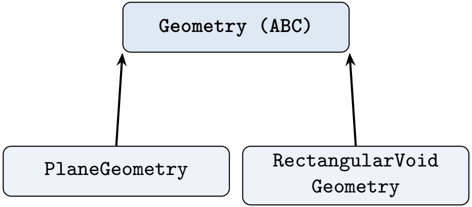
```python
@dataclass
class Module:
    module_id: ModuleID
    z:         float
    lx:        float   # half-width x
    ly:        float   # half-width y
    hit_ids:   list[int]
```

Segment (reconstruction layer)

```python
@dataclass
class Segment:
    hit_start: Hit
    hit_end:   Hit
    segment_id: SegmentID
    track_id:  TrackID = -1
    pv_id:     PVID  = -1
```

All type aliases (HitID, ModuleID, . . .) are defined in core/types.py alongside the SupportsPosition protocol.

Geometry Hierarchy

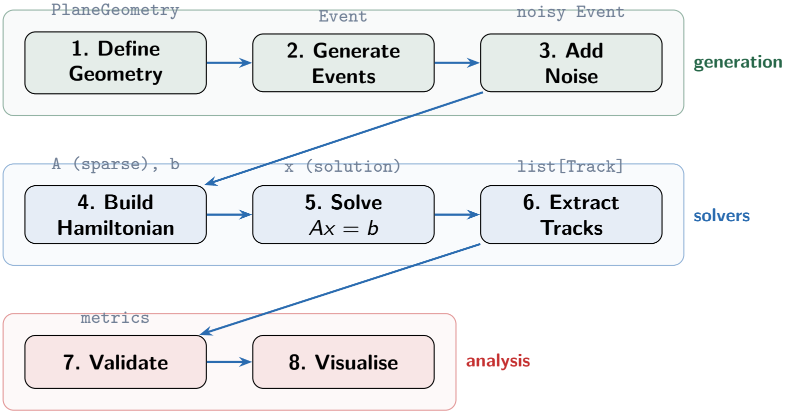


`PlaneGeometry`: simple rectangular planes.
`RectVoidGeometry`: planes with a beam-pipe hole.

ABC contract

```
class Geometry(ABC):
    @abstractmethod
    def __getitem__(idx) -> tuple
    @abstractmethod
    def __len__() -> int
    @abstractmethod
    def point_on_bulk(pos) -> bool
    @abstractmethod
    def get_z_positions() -> list
```

End-to-End Pipeline



Each stage only depends on the **data objects** produced by the previous stage — never on the *generator* class itself.

Pipeline — Code Sketch

```
# 1. Geometry
geo = PlaneGeometry(n_modules=26, z_first=0, z_spacing=55,
                   lx=50.0, ly=50.0)
# 2-3. Generation + noise
gen = StateEventGenerator(detector_geometry=geo,
       theta_max=0.40, phi_max=0.30, sigma_ms=0.01,
       sigma_noise=0.02, n_events=1, n_tracks=8)
event      = gen.events[0]
noisy_event = gen.make_noisy_event(event, efficiency=0.95,
                                ghost_rate=0.10)
# 4. Hamiltonian
ham = SimpleHamiltonian(epsilon=0.02, gamma=1.5,
                       delta=1.0, theta_d=0.01)
A, b = ham.construct_hamiltonian(gen, convolution=False)

# 5. Solve
x = ham.solve_classicaly() # or solve_conjugate_gradient(A, b)

# 6. Reconstruct
reco_tracks = get_tracks(ham, x, gen, threshold=0.5)

# 7-8. Validate & plot
validator = EventValidator(event, reco_tracks)
_, metrics = validator.match_tracks(purity_min=0.75)
fig = plot_event_3d(event, show_modules=True)
```

Old vs. New: At a Glance

Aspect	**Old**	**New**
Layout	Flat — 8 files, one folder	4-layer tree: `core / generation / solvers / analysis`
Generator	God class — factory & data store	Pure factory; returns `Event`; no mutable state
Cross-refs	Object refs (`Track.hits`)	ID lists (`Track.hit_ids`)
Segments	Stored on `Event` & `Track`	Computed on-demand in `reconstruction/`
Typing	Minimal; `dict` states; wildcard imports	`types.py` aliases, protocols, `py.typed`
Visualisation	Embedded in `Event.plot_segments()`	Standalone in `analysis/plotting/`
Geometry	Defined alongside data classes	Separate sub-package with ABC contract
Quantum	Not scaffolded	`solvers/quantum/` ready (HHL, 1-bit)

Design Principles Applied

1. **Single Responsibility**
   Each class does one thing: `Hit` stores a measurement, `Geometry` defines acceptance, `EventValidator` computes metrics.
2. **Dependency Inversion**
   Solvers depend on the `Geometry` ABC — not on a concrete `PlaneGeometry`.
3. **Open–Closed**
   Adding a new geometry (e.g. `PixelGeometry`) means subclassing `Geometry` and implementing four methods — no changes to downstream code.
4. **Separation of Concerns**
   Plotting lives in `analysis/plotting/`, data lives in `generation/entities/`, solving lives in `solvers/`.
5. **Serialisability**
   All cross-references use integer IDs so the full `Event` round-trips through JSON.

Why Segments Are Not Stored

Old approach

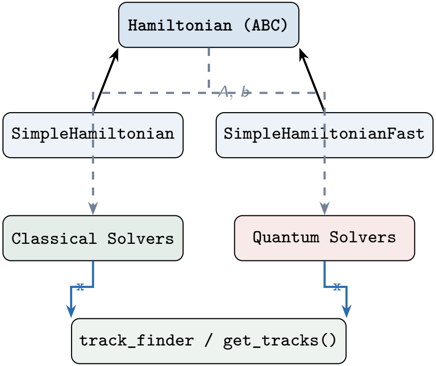
- `Event.segments` and `Track.segments` stored pre-computed segments
- Duplicated data (same info derivable from hits)
- Complicated serialisation
- Tight coupling to the generation phase

New approach

- Segments created *on-demand* by `Hamiltonian.construct_hamiltonian()`
- Live only in the reconstruction layer
- No duplication, no serialisation burden
- Different Hamiltonians can define different segment-building strategies

```
# Segment built from two consecutive hits
seg = Segment(
    hit_start=hit_a,
    hit_end=hit_b,
    segment_id=next_id,
    track_id=-1  # unknown until reco
)
```

Hamiltonian & Solver Hierarchy



Classical: `solve_direct` (dense), `solve_conjugate_gradient` (sparse, iterative), `select_solver` (auto-dispatch).
Quantum: HHL and 1-bit HHL — ready for Qiskit integration.

Non-Greedy Track Matching

Typical (greedy) approach

- Match reconstructed tracks to truth tracks in arbitrary order
- First match "wins" — later, better matches may be blocked
- Can over-count clones and under-report efficiency

New non-greedy matching

- Rank *all* candidate matches by a composite quality score: `purity` × `hit_efficiency`
- Assign the globally best match first, then remove both tracks from the pool
- Repeat until no candidates exceed the `purity_min` threshold
- Remaining unmatched reco tracks → ghosts; unmatched truth tracks → missed

Result: metrics are globally optimal — no match ordering artefacts. Implemented in `analysis/validation/validator.py` (`EventValidator.match_tracks`).

Summary

1. **Modular package layout** with clear layer boundaries: `generation` → `solvers` → `analysis`
2. **Clean dataclasses** (`Hit`, `Track`, `Module`, `Event`) with ID-based cross-references and JSON round-tripping
3. **Pluggable geometry** via an ABC; two implementations shipped
4. **On-demand segments** — computed in the solver, not pre-stored
5. **Non-greedy track matching** — globally optimal assignment, reliable efficiency and ghost-rate metrics
6. **Quantum solvers** implemented and verified: HHL (cosine sim > 0.99) and 1-BQF ($3.4\times$ shallower circuit)
7. **Full type coverage** with aliases, protocols, and `py.typed`

A Common Framework — Let's Collaborate

The goal of this work is the **start of a common framework**
for quantum track-reconstruction research.

- **Shared codebase**: consistent event generation, Hamiltonian construction, and validation across groups
- **Easy to extend**: plug in new geometries, solvers, or analysis modules without touching the core
- **Reproducible benchmarks**: same metrics and matching for fair comparison of classical vs. quantum approaches
- **Open and pip-installable**: `pip install lhcb-velo-toy`

We would love to **work together** on this —
contributions, feedback, and use cases are very welcome!

Thank you

Questions?

https://github.com/GeorgeWilliam1999/LHCb_VeLo_Toy_Model