NDS.LIVE TECHNICAL DAY · OCTOBER 2026

Ask for map features, not for storage

Towards a higher-level NDS.Live API - design ideas for discussion



HOW NDS.LIVE IS BUILT - AND WHY

Built to be delivered and updated efficiently

Tiles and levels

"...so the data can be stored, loaded, and updated in manageable chunks instead of all at once."

Tiles are self-contained: no references between tiles, so updates cause no avalanche.

Layers, versioned on their own

"Tiles do not have a version in NDS.Live. Instead, the data layers inside tiles are independently versioned."

A service "combines exactly the layers that a client asks for"; each is compressed and signed.

Attributes apart from features

"Attribute maps are a space-efficient way to store attributes because the same attribute can be assigned to multiple features."

Column-based lists of one type compress well.

Modules, versioned independently

"Modules are versioned independently, ensuring that updates to one module do not affect others."

Compatibility rules let a newer reader read older data.

Services and lifetimes

"...static, dynamic, and live data can coexist in one system while remaining updateable and interoperable."

Tiles, paths, objects; a registry as the entry point.

Why it pays off

The map never stands still: roadworks, temporary closures, new roads, removed roads, changed speed limits - to name a few.

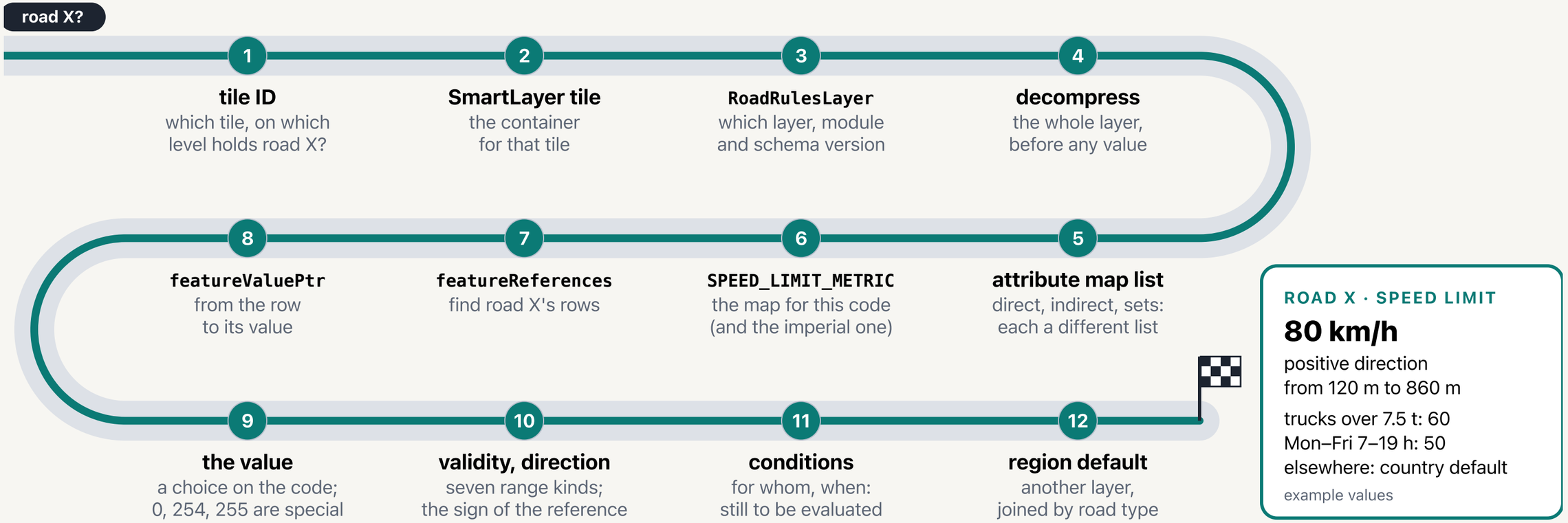
An update ships only the layers a change touches, as a delta. Short-lived changes such as closures come as dynamic data.

Quotes: NDS.Live documentation (documentation.nds.live, a4e0d43) - tiling_scheme.md:3-5, differences_classic_to_live.md:34-36, tiles.md:32-33, about_nds_live.md:53-54, 76-77, attribute_maps_and_sets.md:6-7, 13-14, system_model.md:33-34, data_lifetime.md:33, delta_updates.md.

THE PRICE OF EFFICIENCY

One question about a feature, twelve steps through storage

"What is the speed limit on road X?" - the storage answers questions about storage; this one is about a road.



ROAD X · SPEED LIMIT

80 km/h

positive direction
from 120 m to 860 m

trucks over 7.5 t: 60
Mon–Fri 7–19 h: 50
elsewhere: country default

example values

WHAT WE WANT IT TO BE

```
map.getRoad(roadId).getSpeedLimits();*
```

* well - almost. Levels, services and data versions add a line or two.

Types and codes: NDS.Live 2026.09 (rules, core). Simplified: indirect values and attribute sets add further paths.

THE IDEA

Two things users get

1 · The logical model specification

Normative and language-neutral: which features exist - Road, Intersection, LaneGroup, Lane, … - with their identity, geometry, attributes and relations, independent of how and in which version they are stored.

Generated, like the NDS.Live schema today, from a source that carries hand-written comments and documentation - so it cannot drift from the code. On top: concise, hand-written guides, like the architecture documentation of the NDS.Live schema.

2 · The generated APIs

```
map.getRoad(roadId)
   .getSpeedLimits();
```

Typed and generated per language from the same source - C++ first, Python next.

Built on the APIs generated from the NDS.Live schema today, not instead of them.

Higher-level API - how a program asks: typed getters, navigation, queries — **new**

Logical model - what the map says: features, attributes, relations — **new**

NDS.Live schema - how the map is stored: tiles, layers, attribute maps, versions per module — **today**

WHAT CHANGES FOR A MAP USER

What you can forget - and what stays yours

You never see

- tiles and tile borders
- SmartLayer containers, packaging, compression
- attribute maps, value pointers, raw codes
- seven kinds of validity
- artificial intersections, zero-length lanes
- schema versions

higher-level API

Still yours - it is about the map

Which detail level

the detailed network or an aggregated one

Which data version

static releases, dynamic or live data

The situation you ask about

time, vehicle, weather

Memory and loading are technical necessities, not map concepts - we come back to them.

THE API

What a user writes

```
auto map = nds::Map::connect(url);
auto roads = map.getRoads(nds::Circle{position, 200});

auto road = map.getRoad(roadId);
// will cover a non-existing road cleanly

for (auto& limit : road->getSpeedLimits()) {
    limit.value();       // 50 km/h · Unlimited · Variable
    limit.range();       // [0.0, 0.62] of the road
    limit.conditions();  // trucks, 22:00–06:00
    limit.sourceRef();   // tile, layer, …
}

for (auto next : road->getNextRoads(nds::Positive))
    …                    // across tile borders
```

connect
one data source; cache and loading behind it

query by extent
circle, box, corridor or tile - never "give me tile x"

look up by ID
tile + type + local ID, reversible to the stored road

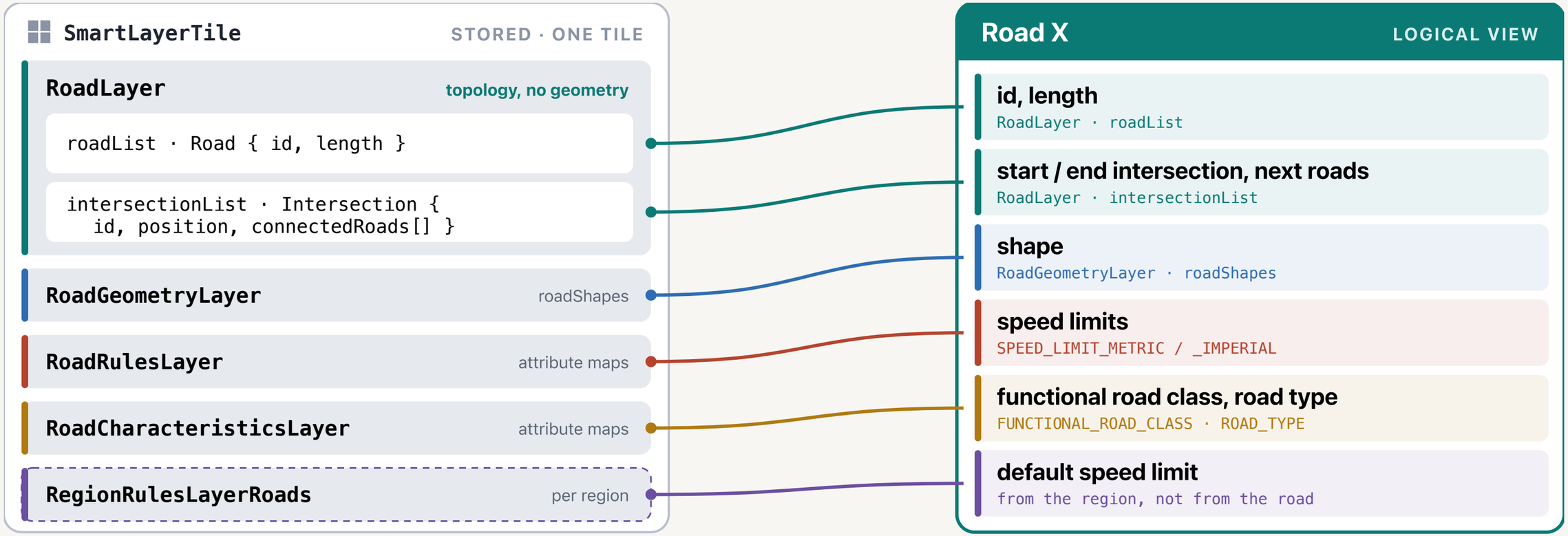
one getter per meaning
metric and imperial codes, maps and sets, direct and indirect - all in `getSpeedLimits()`; typed values

traceable
every value knows where it is stored

navigation
through intersections, across tile borders, without artificial intersections

THE LOGICAL MODEL

A feature is assembled from the layers that store it



The same pattern for every feature type.

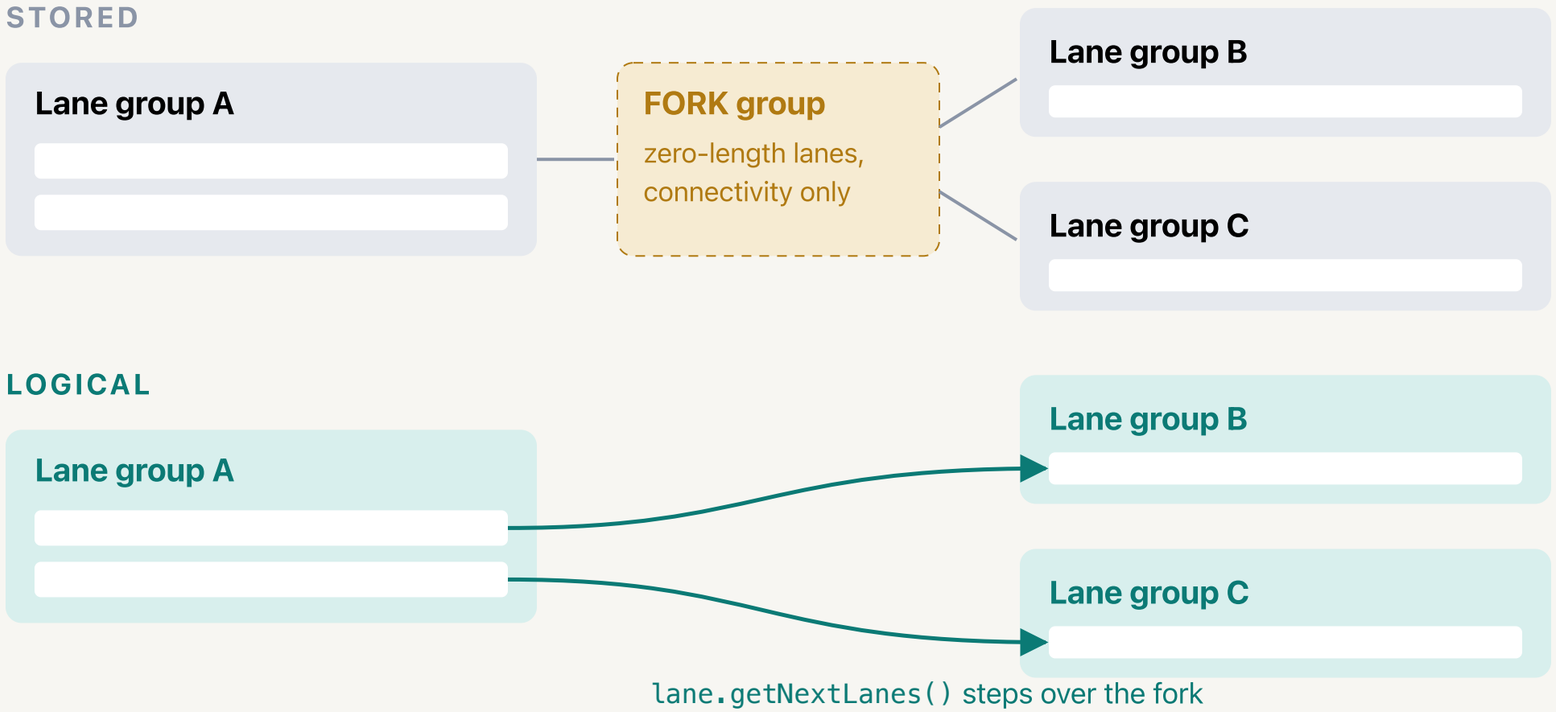
The logical model is defined by formalizing these relations - layer, list and code to property, per schema version.

Not always one to one: a lane's prohibited passage is stored on its lane group and handed to the lane through a lane mask.

Fact: smart 2026.09 tile.zs:77-87 (one tile, several data layers); road 2026.03 layer.zs:30-69, road.zs:97-132; rules 2026.09 layer.zs:29-37, 102, 224, attributes.zs:326-329; characteristics 2026.09 layer.zs:30, attributes.zs:201, 212.

ONLY REAL FEATURES

Constructs for tiling and connectivity become relations



Tile borders, the same way

A road crossing a border ends in an artificial intersection on each side.

`getNextRoads()` finds the twin in the neighbour tile by position and z-level: one neighbour, one load.

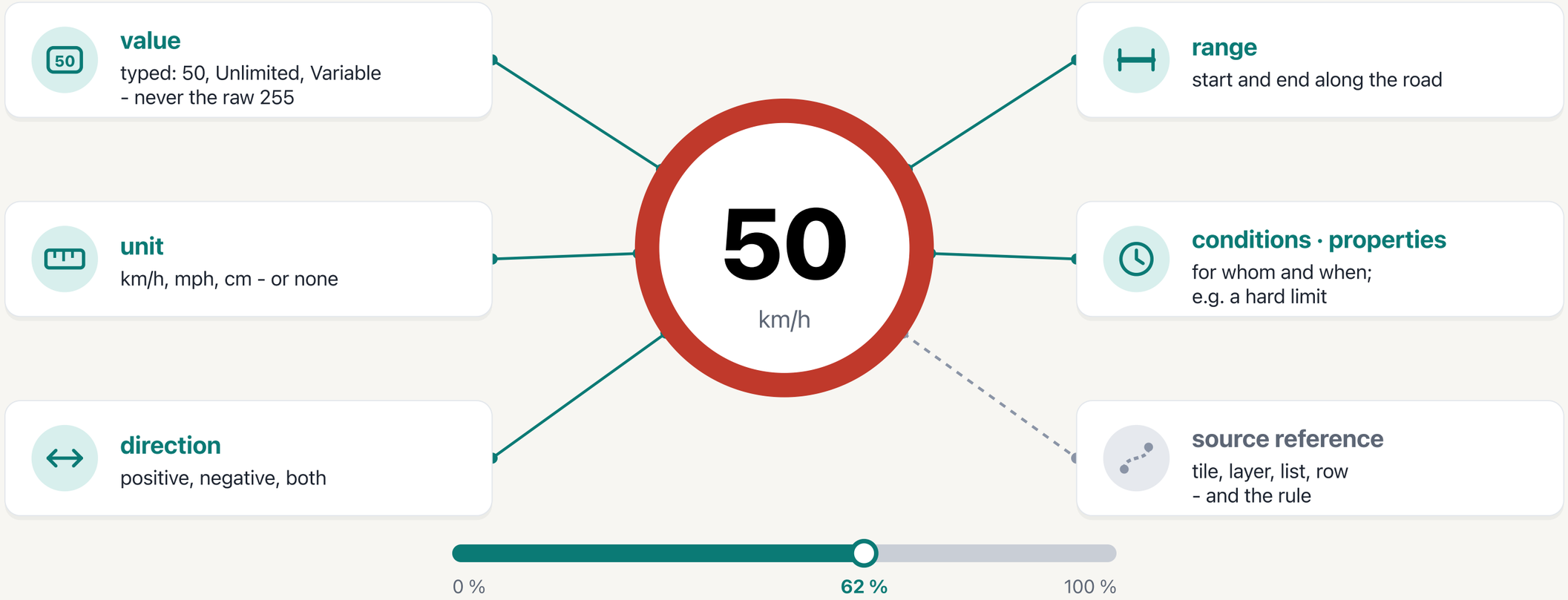
Not done: stitching a long highway into one feature - that would load every tile it crosses. Tile-cut roads stay separate features with separate IDs until stitched IDs exist.

Fact: BORDER and FORK lane groups are "artificially introduced", carry connectivity only (lane 2026.09, lane-7tcxtz, lane-rpxa5q).

A VALUE NEVER TRAVELS ALONE

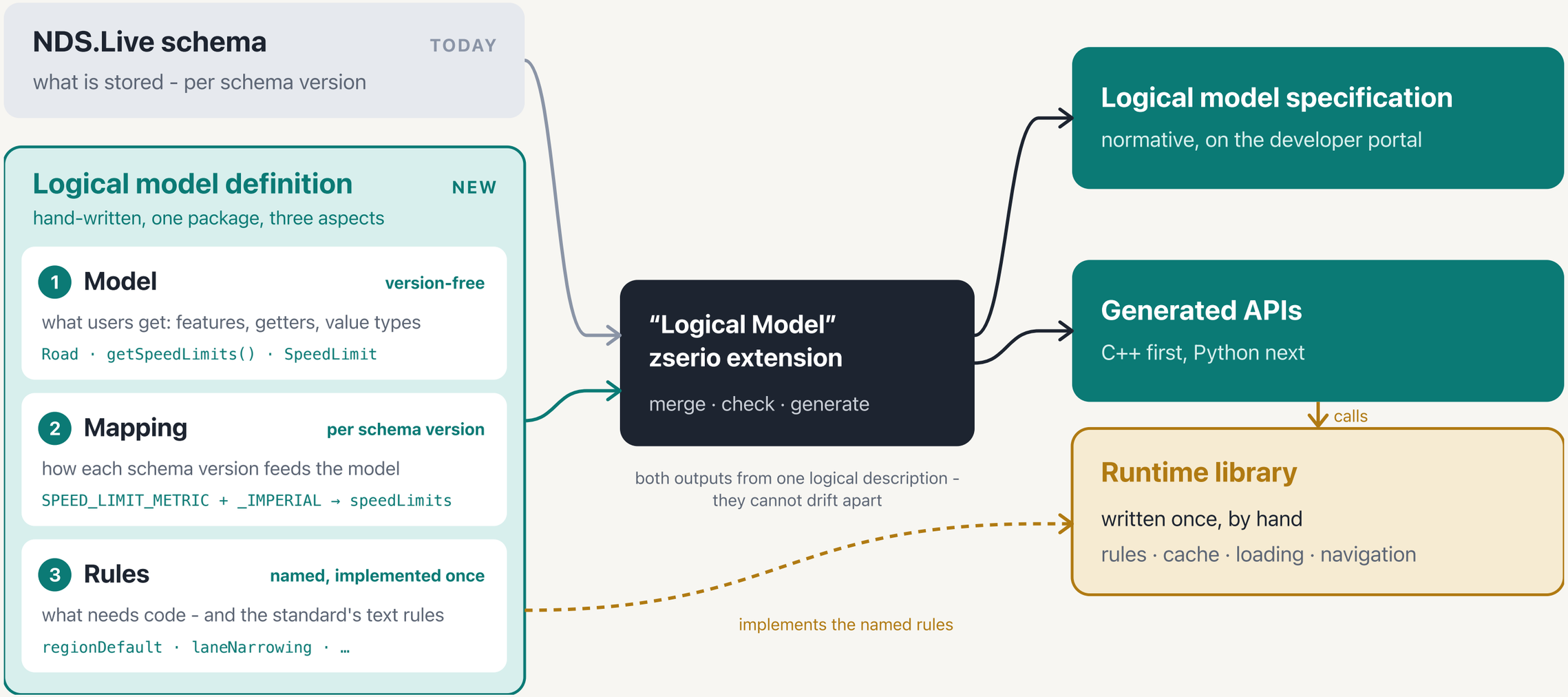
One speed limit of road X

`road.getSpeedLimits()` always returns **all** speed limits of road X - each with the full context needed to use it.



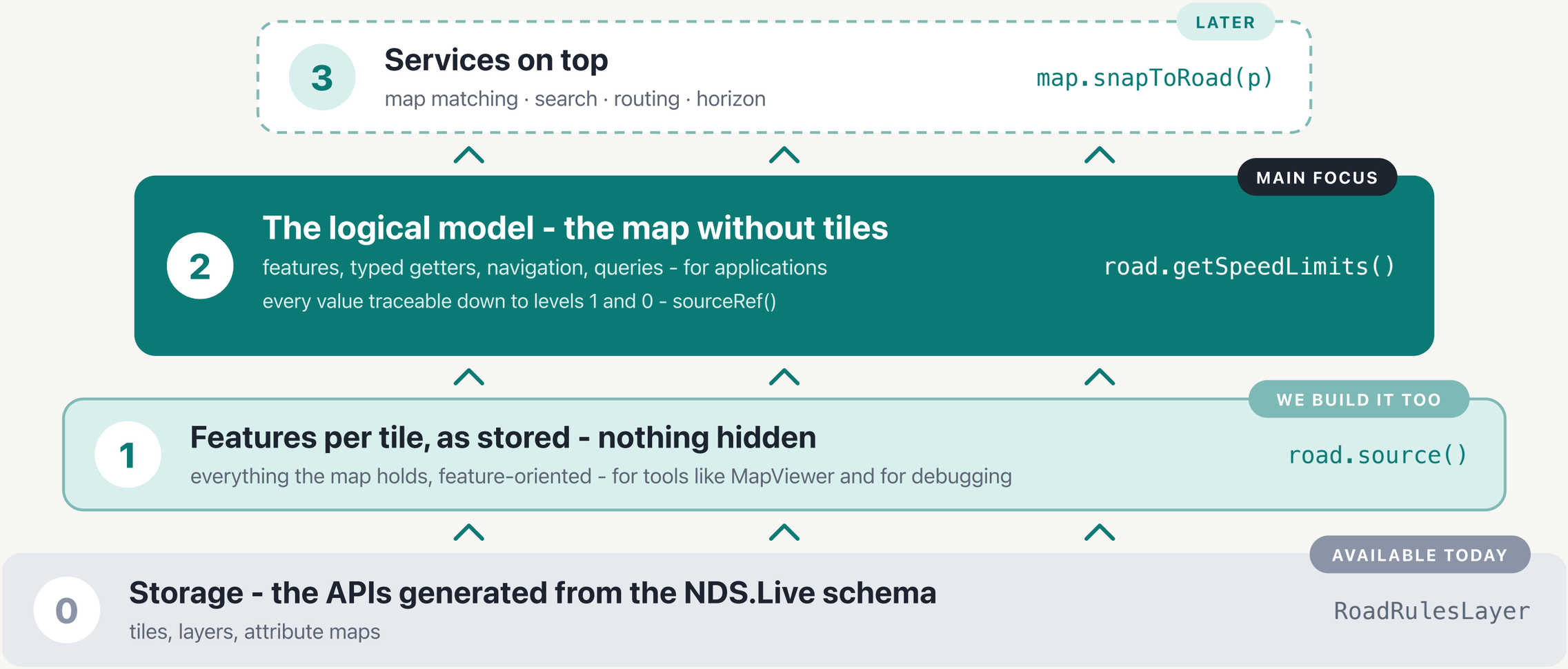
HOW IT IS BUILT

Three building blocks, two outputs - and a runtime



API LEVELS

Built up level by level - level 2 is our focus



Detail levels - roads aggregated on levels 11 to 13 - are another kind of level; they are on the next slide.

BEYOND THE ESSENTIALS

What we have to consider in addition



Detail levels

Roads aggregated on levels 11 to 13; a map reads one, up and down by relation.

designed · proof of concept: level 13



Conditions and effective value

“A car at 10:00, eastbound” - the standard's procedure, named policies where it leaves the choice.

rule register



Beyond roads

Every feature type, attribute module and reference kind - each kind answered once.

proof of concept: roads, lanes



Data versions

Tile versions need not be consistent; local IDs are not stable across versions.

solution open



Caching

Filled lazily by column, one byte budget - and invalidated when data changes.

invalidation open



Loading and memory

An explicit fetch policy, never a hidden load; one memory budget for the map.

designed



Tracing and export

Every value back to its stored row; constant-attribute edges for graphs and routing.

designed



Safety

No exceptions across the API, bounded memory, MISRA C++:2023.

designed · not yet checked

Not for 30 minutes - but each one has its place in the design, and all of them have to be solved.

INTO THE PANEL

Where we are and how to continue

Where we are

Design
the higher-level API and the logical model, in review

C++ proof of concept - ongoing
a tool to challenge and refine the approach, not yet for review;
evaluated with different maps: roads, lanes, several kinds of attribution

Next
refine it with you - C++ first, Python next

Let's discuss

1. Which questions should the API answer first for your tools?
2. Should the mapping - which codes form one meaning, units, what a reference points at - become annotations in the standard?
3. Who owns the rules and the policies where the standard leaves the choice?

Shape it with us.
Collaboration within the NDS developer community is essential to validate and shape the logical model and the higher-level API - join the discussion.