NTE: Network Topology Engine

Technical Reference

Simon Knight

Adelaide, Australia

March 2026 • Version 0.3

Abstract. NTE (Network Topology Engine) is a Rust-based hybrid graph-relational engine for network topology modelling, exposed to Python via PyO3 bindings under the package name `ank_nte`. Its central design principle is a dual-write store: topology structure lives in a `petgraph::StableDiGraph` for $O(1)$-hop traversal, whilst node and edge properties are simultaneously maintained in Polars Arrow-backed DataFrames for SIMD-accelerated columnar scans, with a filter-first query planner that exploits both representations in a single query. This report is intended for network automation engineers and backend developers integrating NTE into larger systems, and covers the full surface area of the engine: the data model, storage architecture, query system (including a fluent Python query builder and an interactive NTE-QL REPL with visual EXPLAIN), Python API, ACID transactions, persistence, GPU acceleration, graph algorithm plugins, policy engine, reactive events, production diagnostics, and distributed deployment.

Contents

1 Introduction

1.1 What NTE Solves

Network automation tools routinely need to answer two very different classes of question about a topology:

1. **Relational / property questions** — "Which routers have AS number 64512 and are in the SYD PoP?" — that are best answered by scanning columns of structured data.
2. **Graph / structural questions** — "Is there a path between device *A* and device *B* using fewer than 4 hops?" — that are best answered by traversing adjacency lists.

Most systems optimise for one and bolt on the other as an afterthought. Relational databases support graph queries via recursive CTEs that are notoriously hard to optimise. Dedicated graph databases (Neo4j [6], KuzuDB [3]) support property lookups but typically store properties row-by-row and lack columnar scan performance.

NTE's answer is a *dual-write* architecture: every mutation is applied to both a petgraph `StableDiGraph` (for structural queries) and a set of Polars Arrow-backed DataFrames (for property scans). A filter-first query planner decides, at runtime, which representation to consult first. In the common case — filter on a property, then traverse — it runs the cheap columnar scan to get a candidate set of node IDs, then constrains the traversal to those candidates.

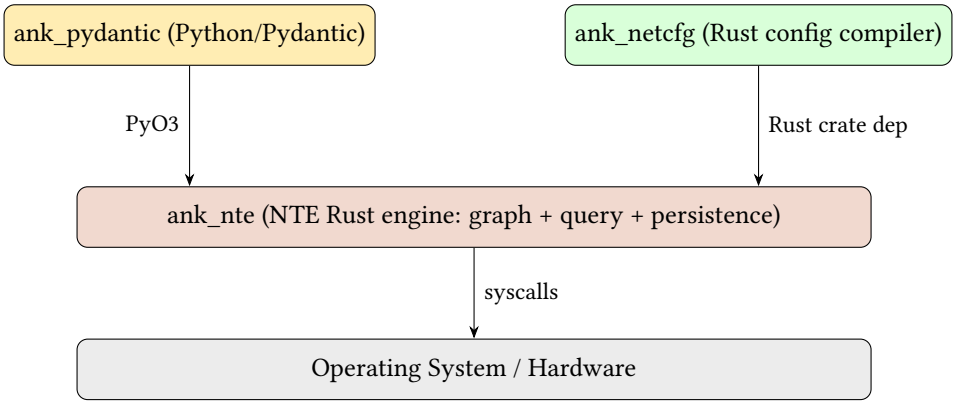
1.2 Ecosystem: ank_pydantic and ank_netcfg

NTE is a shared backend engine consumed by two alternative frontends:

`ank_pydantic` A Python library that provides high-level, Pydantic-validated network models. It owns the domain schema (what a *Router* looks like, what fields are mandatory) and calls NTE via PyO3 bindings. Suited to interactive exploration, Jupyter notebooks, and Python-centric automation pipelines.

`ank_netcfg` A pure-Rust deterministic network configuration compiler. It consumes `nte-topology` directly (no Python layer), reads declarative YAML blueprints, maps them through a DeviceIR intermediate representation, and renders vendor-neutral device configurations. Suited to CI/CD pipelines, large-scale config generation, and environments where a Python runtime is undesirable.

Both frontends use the same underlying NTE storage, query, and persistence layers. The choice between them is an ergonomic decision, not an architectural one.



All three repositories live side-by-side in the same parent directory.

1.3 Who Should Read This Document

- **Network automation engineers** using `ank_pydantic` or `ank_netcfg` who want to understand what the engine is doing, write performant queries, or interpret error messages from NTE's exception hierarchy.
- **Backend developers** integrating NTE directly (via the Rust API or the Python extension) into larger systems, needing to understand the crate workspace, build process, and extension points.
- **Contributors** wishing to extend the engine with new backends, query operators, or GPU kernels.

1.4 How to Read This Document

Readers wanting to get started quickly should read §2 (Data Model) and §5 (Python API), then refer to individual sections as needed.

§§2–3 cover the core data model and storage engine. §4 covers the query system: the filter-first planner, the fluent Python query builder, NTE-QL, and the interactive REPL. §5 provides a Python API walkthrough. §6 covers ACID transactions and conflict detection. §§7–13 cover advanced topics — persistence, GPU acceleration, graph algorithm plugins, policy engine (including shadow topologies), reactive events, production diagnostics, and Raft clustering — that can be read in any order.

2 Data Model

2.1 Formal Definition

A topology in NTE is a typed, directed, attributed multi-graph $G = (V, E, \ell_V, \ell_E, \pi_V, \pi_E, L, \lambda)$ where:

$$
\begin{aligned}
V &= V_{\text{node}} \cup V_{\text{endpoint}} \quad \text{(disjoint node and endpoint sets)} \\
E &\subseteq V \times V \quad \text{(directed edges)} \\
\ell_V &: V \to \Sigma_V \quad \text{(type label, e.g. “Router”, “Switch”)} \\
\ell_E &: E \to \{\text{Inter}, \text{Intra}, \text{Intranode}, \ldots\} \quad \text{(edge kind)} \\
\pi_V &: V \to (\text{String} \rightharpoonup \text{JSON}) \quad \text{(property maps)} \\
\pi_E &: E \to (\text{String} \rightharpoonup \text{JSON}) \quad \text{(edge property maps)} \\
L &\quad \text{(ordered set of named layers)} \\
\lambda &: V \to L \cup \{\bot\} \quad \text{(optional layer assignment)}
\end{aligned}
$$

Nodes are assigned integer IDs that are globally unique within a topology instance. NTE uses `i32` as the external ID type. IDs are chosen by the caller; NTE does not auto-assign them (though helpers exist for auto-increment).

2.2 Node Types

NTE recognises two top-level vertex categories:

Node A network *device* — a router, switch, server, firewall, optical transponder, etc. Nodes carry a string `node_type` and an optional layer membership. Property data is stored as an arbitrary JSON object, populated by `ank_pydantic` from Pydantic model fields.

Endpoint A connection point on a node — a physical or logical port, interface, or sub-interface. Endpoints are owned by exactly one node via an Intra edge. Like nodes, they carry a type label (e.g. “Endpoint”, “Interface”) and a property map.

A **LogicalNode** is a specialised node that *always* belongs to a layer and represents protocol-level or virtual topology objects (e.g. a BGP session state machine, a VLAN domain). Logical nodes are excluded from most query results by default.

> **Terminology note**
>
> NTE uses "node" specifically for *network devices* and "endpoint" for their connection points. This differs from graph-theory usage where "node" means any vertex. In this document, "vertex" is used when the graph-theoretic meaning is intended.

2.3 Edge Kinds

NTE defines three principal external edge kinds. Internally, the engine also uses several bookkeeping edge kinds (`Parent`, `Child`, `Root`, `Origin`) that are invisible to Python callers by default.

2.3.1 Inter (Connectivity)

An **Inter** edge connects two endpoints. It models a physical or logical link between two ports on different devices. Inter edges are bidirectional by convention: NTE stores them as a pair of directed edges. The Python method is `add_inter_edges`.

2.3.2 Intra (Structural / Ownership)

An **Intra** edge runs *from* an endpoint *to* the node that owns it. It expresses the "this port belongs to this device" relationship. A node may own any number of endpoints; each endpoint has exactly one owning node. The Python method is `add_intra_edges`.

2.3.3 Intranode

An **Intranode** edge connects two nodes within the same logical chassis. This is used to model devices that consist of multiple line cards or virtual routing instances that share a common management plane but appear as separate nodes in the topology. The Python method is `add_intranode_edges`.

2.3.4 Device Shortcut (DeviceLink)

When an Inter edge is created between two endpoints, NTE automatically adds a **DeviceLink** shortcut edge between the owning devices. This avoids the three-hop traversal (*DeviceA* → *PortA* → *PortB* → *DeviceB*) in the common case where callers only want device-to-device connectivity. The shortcut is maintained in `device_shortcuts: HashMap<(i32,i32), Vec<EdgeIndex>>` inside `NteGraph`.

2.4 The Three-Hop Connectivity Pattern

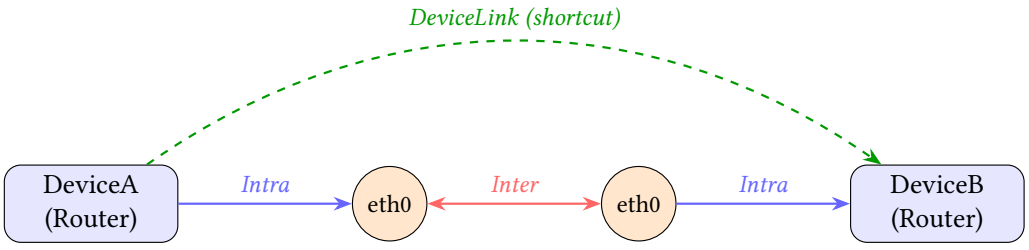


Figure 1. The three-hop connectivity pattern and the automatic DeviceLink shortcut. Intra edges run *from* endpoint to device; Inter edges are bidirectional.

Figure 1 shows the canonical pattern for inter-device connectivity. The DeviceLink shortcut means that "which devices are directly connected?" queries do not need to traverse six vertices; they can follow a single shortcut edge. However, if a query requires port-level detail (e.g. "which port on DeviceA connects to DeviceB?"), the full three-hop path is available in the graph.

2.5 Layer Hierarchy

Layers partition the topology into planes of abstraction. Common examples include a `physical` layer (hardware links), an `ip` layer (IP adjacencies), a `bgp` layer (BGP sessions), and an `mpls` layer. Layers are ordered and can declare dependencies: the BGP layer depends on the IP layer, which depends on physical.

Layers are stored in a `LayerDependencyGraph` (a DAG of layer names). NTE validates at edge-creation time that nodes on incompatible layers are not mixed (raising `LayerMismatchError` if so). A special `base` layer exists for cross-cutting nodes that appear in all layers.

2.6 Property Maps

Both nodes and edges carry arbitrary key-value properties stored as JSON objects. Properties are serialised from Pydantic model instances by `ank_pydantic` and stored in Polars DataFrames as individual columns (one column per field name). The Polars representation enables fast columnar scans, type-aware comparisons, and SIMD acceleration.

> **Warning**
>
> Property columns are created lazily as new field names appear. This means queries against fields that have never been set on any node return an empty result rather than an error. Check column existence before relying on absent-field semantics.

2.7 Workspace Layout

NTE is an 18-crate Cargo workspace. The crates and their roles:

Crate	Layer	Purpose
`src/` (root)	Bindings	PyO3 Python extension (`lib.rs`, `topology.rs`, `query.rs`, `transaction.rs`)
`nte-core`	Core	Core topology with domain bindings
`nte-domain`	Core	Pure domain types (no dependencies)
`nte-graph`	Core	`petgraph` wrapper, ID mapping, CSR, GPU kernels
`nte-topology`	Core	Topology struct, transactions, snapshots, diffing, shadow
`nte-query`	Query	Executor, planner, matcher, profiler, visualisation, algorithm plugins
`nte-cli`	Tooling	NTE-QL REPL, policy validator, TUI dashboard
`nte-lsp`	Tooling	Language Server Protocol for NTE-QL editing
`nte-policy`	Policy	Starlark engine, DeltaNet validator
`nte-backend`	Storage	Abstract `TopologyBackend` trait
`nte-datastore`	Storage	Feature-flag crate selecting the active backend
`nte-datastore-core`	Storage	Shared backend types
`nte-datastore-polars`	Storage	Polars Arrow-backed DataFrames (default)
`nte-datastore-duckdb`	Storage	DuckDB embedded OLAP (experimental)
`nte-datastore-lite`	Storage	Lightweight in-memory store (testing)
`nte-server`	Server	Axum HTTP/WebSocket, Raft consensus, diagnostics
`nte-monte-carlo`	Optional	Monte Carlo simulation
`ladybug_backend`	Experimental	KuzuDB graph-native backend exploration

Figure 2 shows how the crates relate. Data flows downward from the consumer frontends through the PyO3 binding layer into the core engine. The query planner sits between the consumer and the dual-write store, deciding whether to consult the graph or the columnar store first.

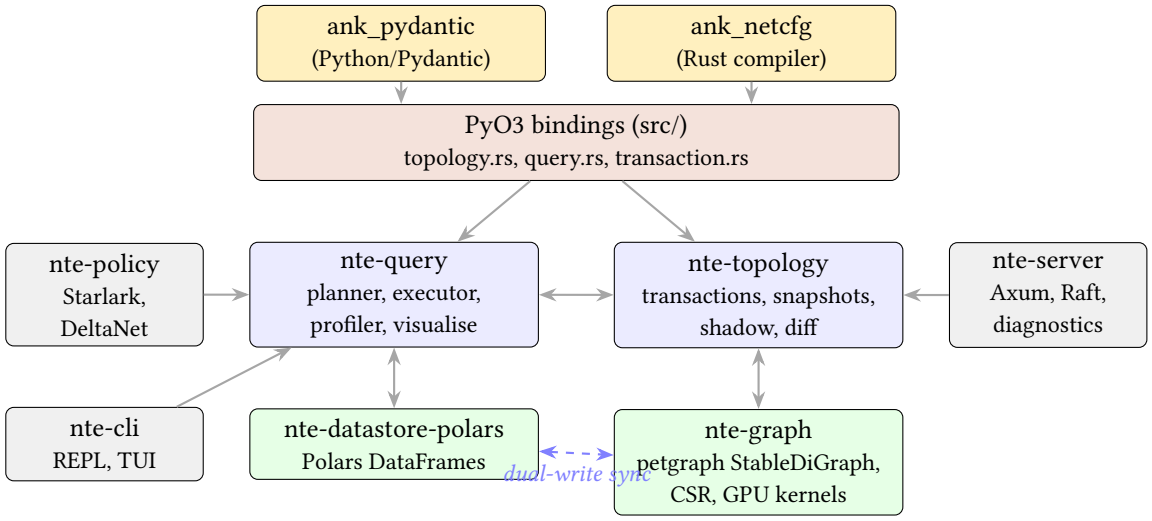


Figure 2. Crate architecture and data flow. Solid arrows show dependency/call direction; the dashed arrow represents the dual-write consistency protocol between the columnar and graph stores.

3 Storage Architecture

3.1 Rationale for Dual-Write

A graph database excels at traversal. A columnar store excels at predicate evaluation on properties. For network topology queries, you almost always need both. A query like "find all routers in the SYD PoP that have a BGP session to a peer with AS 65000" has two parts:

- *Property filter*: `pop == "SYD"`, best served by a columnar scan.
- *Structural filter*: connected via a BGP-layer edge to a peer with `as_number == 65000`, best served by graph traversal.

NTE's dual-write approach keeps both representations in sync and lets the query planner pick the optimal execution order.

3.2 petgraph StableDiGraph

The graph structure is stored in a `petgraph::StableDiGraph<GraphNode, GraphEdge>` [1]. `StableDiGraph` was chosen over the non-stable variant because:

- Node and edge indices remain valid after deletions. This is essential because NTE's ID mapping (`HashMap<i32, usize>` in both directions) must not be invalidated by unrelated mutations.
- It allows $O(1)$ lookup of node weights by `NodeIndex`, which is the primary graph-layer operation.

The `NteGraph` struct wraps `StableDiGraph` and adds:

- **Bidirectional ID mapping**: `index_map: HashMap<i32, usize>` (external → internal) and `reverse_index_map: HashMap<usize, i32>` (internal → external). All Python-facing APIs use external `i32` IDs.
- **Endpoint / node ID sets**: `HashSet<i32>` for $O(1)$ "is this ID an endpoint?" checks.
- **Layer dependency graph**: `LayerDependencyGraph` for topological ordering and layer-consistency checks.
- **Device shortcut map**: `HashMap<(i32,i32), Vec<EdgeIndex»` for fast device-to-device lookups.
- **CSR cache**: a thread-safe, lazily-computed Compressed Sparse Row representation for high-performance batch traversal (described in Section 7).

3.3 Polars Columnar Store

Node and edge properties are stored in Polars [10] DataFrames — one DataFrame per entity class (nodes, endpoints, links). Polars uses the Apache Arrow columnar format internally, which gives:

- SIMD-accelerated predicate evaluation (AVX2/AVX-512 on x86-64).
- Copy-on-Write (CoW) semantics for zero-copy snapshot cloning.
- Zero-copy transfer to Python via PyArrow (when the `arrow-ids` feature is enabled).
- Lazy evaluation via `LazyFrame` for filter push-down.

Properties are stored in wide columnar format: each unique property name becomes a column in the DataFrame. This is a deliberate trade-off. A narrow schema (one row per property) would be more flexible but would require an expensive "pivot" before columnar predicates could be applied. The wide schema means NTE scans efficiently but requires that all instances of a given node type have compatible types for a given field name.

3.4 Dual-Write Consistency Protocol

Every mutation method in the `Topology` struct applies changes to both the graph and the DataFrame store before returning. The protocol is:

Algorithm 1 Dual-write mutation protocol

1: Acquire write lock on topology
2: Validate inputs (type checks, layer compatibility, ID uniqueness)
3: Apply mutation to `NteGraph` (structural side)
4: Apply corresponding mutation to `DataFrameStore` (property side)
5: If either step fails, rollback and propagate error
6: Emit `TopologyEvent` to ring buffer
7: Invalidate CSR cache
8: Release write lock

> **Note**
>
> NTE implements copy-on-write transaction semantics: if step 4 fails, the graph mutation in step 3 is rolled back via a compensating operation and the DataFrame store is restored from its pre-mutation Arrow CoW clone in O(1). This ensures atomicity within a single process. However, NTE does not implement a write-ahead log (WAL): a process crash between steps 3 and 4 leaves the in-memory state inconsistent. For durable storage, use the snapshot/serialisation APIs (Section 7). Distributed deployments use Raft to provide crash consistency (Section 13).

The `Topology` struct is protected by a `RwLock`. Query operations acquire a read lock; mutations acquire a write lock. The PyO3 binding layer releases the Python GIL before acquiring Rust locks, so concurrent Python threads can make progress on other work while a write lock is held.

3.5 EventStore Ring Buffer

Every mutation emits a `TopologyEvent` variant into an in-memory ring buffer (a `VecDeque` managed by the `EventStore`). Events carry:

- A monotonically increasing `EventSequence` number (atomic `u64`).
- A UTC timestamp (`chrono::DateTime<Utc>`).
- Event-specific payload (node IDs, edge pairs, changed fields, etc.).

Events are serialisable to JSON (via `serde_json`) for external consumption. The ring buffer has a configurable capacity; once full, the oldest events are evicted.

3.6 Pluggable Backends

The storage layer is factored into three crates:

Crate	Description	Status
`nte-datastore-polars`	Polars Arrow-backed DataFrames	Default, production
`nte-datastore-duckdb`	DuckDB embedded OLAP database	Optional, experimental
`nte-datastore-lite`	Lightweight in-memory store	Development / testing

The `nte-datastore-core` crate defines the abstract `TopologyBackend` trait that all backends implement. Selecting a backend is a compile-time feature flag in `nte-datastore`.

4 Query System

4.1 Design Goals

NTE's query system was designed around three principles:

1. **Filter-first execution**: property predicates should run before graph traversal, not after, to eliminate candidates early.
2. **Composable, typed query objects**: queries should be buildable programmatically from Python, not only via string parsing, so that tooling can construct, inspect, and serialise query plans.
3. **Honest cost estimation**: the planner should rank predicates by selectivity rather than always applying them left-to-right.

4.2 Query Representations

The `nte-query` crate exposes two parallel query representations:

`QuerySpec` A flat set of node-level filters: type, layer, kind, ID set, field equality, and expression filters. Used for "give me all nodes matching these conditions" queries. Exposed as `QuerySpec` in Python.

`QueryPlan` / `PatternNode` A structural graph pattern AST. Describes a subgraph shape to match (e.g. a router connected to a switch via an Inter edge). Used for multi-hop traversal queries. Exposed as `QueryPlan` and `PatternNode` in Python.

4.3 Filter-First Execution Model

The `QueryExecutor` in `nte-query` compiles a `QuerySpec` into an ordered `FilterPlan` — a sequence of `FilterOp` variants sorted by `FilterRank`:

Rank	Filter type	Rationale
F1 (lowest cost)	ID filter	Exact hash lookup, $O(\|S\|)$
F2	System filters	Type, layer, kind — low-cardinality
F3	Field equality	Single-column Polars scan
F4	Range / list	Multi-column or `is_in` scan
F5 (highest cost)	Expression filters	Arbitrary Rust AST evaluation

Execution proceeds left-to-right. Each filter receives the candidate set from the previous filter and returns a (potentially smaller) subset. The Polars LazyFrame pipeline is constructed in-order and materialised with a single `collect()` call, benefiting from Polars' internal optimiser.

Filter plan example

Query: "All routers in the SYD PoP with AS number 64512"

1. F2: type = "Router" → scan `node_types` DataFrame
2. F3: `pop == "SYD"` → filter `data_pop` column
3. F3: `as_number == 64512` → filter `data_as_number` column

Polars executes steps 2 and 3 as a single lazy pipeline after step 1 narrows the candidate set.

4.4 Pattern Matching

The `PatternNode` AST describes the shape of a subgraph. Supported patterns:

`PatternNode::node(type)` Match a single vertex of the given type.

`PatternNode::binding(name, type)` Match a vertex and bind it to a name for later reference.

`PatternNode::chain(steps)` A sequence of alternating node-steps and edge-steps.

`PatternNode::union(patterns)` Union of multiple patterns.

Variable-length paths `HopSpec::exact(n)`, `HopSpec::range(min,max)`, or `HopSpec::any()`.

The pattern matcher performs a depth-first search over the graph, pruning branches when a node fails the pattern constraint. The maximum number of matches is capped at `QueryPlan::DEFAULT_MAX_MATCHES` (10,000) by default to prevent runaway traversals; the result carries a `truncated` flag when this cap is reached.

4.5 Selectivity Ranking in Detail

The five filter ranks correspond to the following `FilterOp` variants:

Listing 1. FilterRank enum from executor.rs

```
enum FilterRank {
    Id         = 0,  // HashSet<i32> lookup -- always first
    System     = 1,  // Type/layer/kind -- low-cardinality columns
    Equality   = 2,  // Single field equality
    Range      = 3,  // List / is_in / range
    Expression = 4,  // Arbitrary ExprNode AST -- always last
}
```

Field equality filters on list values are promoted from `Equality` to `Range` rank because they require a Polars `is_in` operation, which is costlier than a direct equality predicate.

4.6 Query Profiling

Setting `QueryHints{enable_profiling: true}` in a `QuerySpec` records timing data for each filter step. The profiler captures per-step candidate counts and wall-clock durations, enabling identification of expensive filters. The profile is returned alongside the result set.

4.7 Plan Cache

NTE maintains an LRU plan cache (capacity configurable, default 256 entries) for `QuerySpec` objects. The cache key is a `CacheKeyMaterial` struct containing sorted, canonicalised representations of all filter fields (to avoid cache misses from HashMap ordering non-determinism). Cache keys are computed lazily via `OnceLock` to avoid repeated sorting allocations.

> **Note**
>
> The plan cache caches compiled `FilterPlan` objects, not query results. Result caching (returning previously computed row sets) is available via `QueryHints{cache_results: true}` but is off by default, as topology mutations must invalidate the result cache.

4.8 NTE-QL String Interface

In addition to the programmatic query API, NTE supports a Cypher-inspired string query language called NTE-QL. Queries are parsed by a PEG grammar implemented with the `chumsky` parser combinator library and compiled into `QueryPlan` / `QuerySpec` objects before execution.

The grammar is documented in full in Appendix A. Here is a brief survey of supported query patterns:

Simple node scan

```
MATCH (r:Router)
WHERE r.pop = "SYD" AND r.as_number = 64512
RETURN r.hostname
```

Two-hop traversal: direct BGP peers

```
MATCH (r:Router)-[:Inter]->(s:Router)
WHERE r.hostname = "core-01"
RETURN r.hostname, s.hostname, s.as_number
```

Bounded-hop reachability: within 3 hops

```
MATCH (src:Router)-[*1..3]->(dst:Router)
WHERE src.hostname = "edge-01"
RETURN dst.hostname
```

Aggregation: node degree

```
MATCH (r:Router)-[:Inter]->(s:Router)
WHERE r.pop = "SYD"
RETURN r.hostname, COUNT(s) AS peer_count
ORDER BY peer_count DESC
```

NOT / exclusion: isolated routers

```
MATCH (r:Router)
WHERE NOT EXISTS {
    MATCH (r)-[:Inter]->(:Router)
}
RETURN r.hostname
```

Path finding: shortest path

```
MATCH p = shortestPath((src:Router)-[*]->(dst:Router))
WHERE src.hostname = "r1" AND dst.hostname = "r4"
RETURN [n IN nodes(p) | n.hostname] AS path
```

4.9 The ExprNode AST

Complex filter predicates are represented as an `ExprNode` abstract syntax tree. The full set of node types:

Variant	Description
`Field(name)`	Reference to a node/edge property
`BindingField(binding, name)`	Property access via a named binding
`Literal(value)`	Constant (null, bool, int, float, string, list)
`Not(expr)`	Logical negation
`Logical(op, left, right)`	AND / OR
`Comparison(op, left, right)`	$=, \neq, <, \leq, >, \geq$
`Arithmetic(op, left, right)`	$+, -, \times, \div$
`StringOp(op, field, pattern)`	contains, startsWith, endsWith, matches
`IsNull(expr)` / `IsNotNull`	Null checks
`IsIn(field, values)`	Membership test

4.10 Python Fluent Query Builder

In addition to the programmatic `QuerySpec` / `PatternNode` APIs and the NTE-QLstring interface, NTE provides a fluent Python query builder that does not require `ank_pydantic`. The builder is inspired by Polars' expression API and compiles down to the same `QuerySpec` objects used by the Rust executor.

Fluent query builder: node queries

```python
from ank_nte import Topology
from ank_nte.query import QueryNamespace, Expr

t = Topology()
# ... populate topology ...
q = QueryNamespace(t)

# Chainable node query
syd_routers = (
    q.nodes()
     .of_type("Router")
     .in_layer("physical")
     .filter(Expr.field("pop") == "SYD")
     .sort("hostname")
     .ids()
)

# Expression DSL supports operators
fast_ports = (
    q.nodes()
     .of_type("Endpoint")
     .filter(Expr.field("speed_gbps") > 40)
     .count()
)

# String operations
juniper_devices = (
    q.nodes()
     .of_type("Router")
     .filter(Expr.field("vendor").contains("Juniper"))
     .collect()
)
```

Fluent query builder: link queries

```
# Link queries for edge filtering
transit_links = (
    q.links()
     .of_type("Inter")
     .in_layer("physical")
     .filter(Expr.field("speed_gbps") >= 100)
     .ids()
)

# Between-nodes filtering
r1_links = (
    q.links()
     .between(source_ids=[1], target_ids=[2])
     .collect()
)
```

The expression DSL supports the full `ExprNode` AST via Python operator overloading: `==`, `!=`, `>`, `>=`, `<`, `<=` for comparisons; `&` (and), `|` (or), `~` (not) for logic; `+`, `-`, `*`, `/` for arithmetic; and methods `.contains()`, `.startswith()`, `.endswith()`, `.matches()` for string operations. Terminal methods include `.ids()`, `.collect()`, `.count()`, `.exists()`, `.first()`, and `.one()`.

4.11 Query Plan Visualisation

NTE can render query execution plans as Mermaid flowchart diagrams. This is useful for understanding how the planner decomposes a query and for identifying optimisation opportunities.

Listing 2. Mermaid DAG generation (from nte-query/src/visualize.rs)

```
pub fn generate_mermaid(plan: &ExecutionPlan) -> String {
    // Generates a Mermaid flowchart DAG from the execution plan.
    // Nodes represent operations: NodeScan, EdgeScan, Filter,
    // Join, WcojJoin (worst-case optimal), SemiJoin.
    // Edges represent data flow between operations.
}
```

The visualisation is accessible in two ways:

1. **NTE-QL REPL**: `EXPLAIN VISUAL <query>` renders the plan inline.
2. **HTTP API**: `GET /explain?query=...` returns an HTML page with the rendered Mermaid diagram (via the `nte-server` diagnostics endpoint).

4.12 NTE-QL Interactive REPL

The `nte-cli` crate provides a standalone command-line interface with three modes:

`nte-cli repl` An interactive NTE-QL shell (built on `reedline`) that parses queries, compiles them to execution plans, and displays results in formatted tables (via `comfy_table`). The `EXPLAIN` prefix shows the compiled plan; `EXPLAIN VISUAL` renders the Mermaid DAG.

`nte-cli validate` A policy validation workflow: loads a YAML policy file and a topology archive, evaluates all policies, and renders a compliance report.

`nte-cli dashboard` A TUI dashboard (built on `ratatui` and `crossterm`) showing real-time memory usage, CPU load, graph statistics (node/edge counts), and a 30-entry history sparkline.

NTE-QL REPL session

```
nte> MATCH (r:Router) WHERE r.pop = "SYD" RETURN r.hostname
+----------+
| hostname |
+----------+
| r-syd-01 |
| r-syd-02 |
+----------+
2 rows returned

nte> EXPLAIN VISUAL MATCH (r:Router)-[:Inter]->(s:Router) RETURN r, s
-- Mermaid DAG rendered --
```

5 Python API

5.1 Installation

NTE is distributed as a compiled Python extension wheel via PyO3 [2]. In development, it is built from source using `maturin`:

Listing 3. Building NTE from source

```
# Install Rust toolchain (https://rustup.rs) then:
pip install maturin uv

# Development build (fast iteration)
uv run maturin develop

# Or, faster compilation (less optimised):
uv run maturin develop --profile dev-fast

# Production build
uv run maturin build --release
```

5.2 Creating a Topology

Listing 4. Basic topology creation

```
from ank_nte import Topology

# Create an empty topology
t = Topology()

# Check initial state
print(t.node_count())   # 0
print(t.edge_count())   # 0
```

5.3 Adding Nodes

Nodes and endpoints are added in bulk via the `add_nodes` family of methods. Properties are passed as a list of dicts, one per node.

Listing 5. Adding nodes and endpoints

```
# Add device nodes (routers)
node_ids = t.add_nodes(
    ids=[1, 2, 3],
    node_types=["Router", "Router", "Switch"],
    layer="physical",
```

```
    data=[
        {"hostname": "r1", "as_number": 64512, "pop": "SYD", "vendor": "Cisco"},
        {"hostname": "r2", "as_number": 64512, "pop": "MEL", "vendor": "Juniper"},
        {"hostname": "sw1", "pop": "SYD", "vendor": "Arista"},
    ]
)

# Add endpoints (ports)
endpoint_ids = t.add_endpoints(
    ids=[10, 11, 12, 13, 14, 15],
    endpoint_types=["Endpoint"] * 6,
    layer="physical",
    data=[
        {"name": "eth0", "speed_gbps": 100},
        {"name": "eth1", "speed_gbps": 100},
        {"name": "eth0", "speed_gbps": 100},
        {"name": "eth1", "speed_gbps": 100},
        {"name": "eth0", "speed_gbps": 10},
        {"name": "eth1", "speed_gbps": 10},
    ]
)
```

5.4 Adding Edges

Listing 6. Adding ownership and connectivity edges

```
# Intra edges: endpoints -> owning device
# Endpoints 10,11 belong to router 1; 12,13 to router 2; 14,15 to switch sw1
t.add_intra_edges(
    endpoint_ids=[10, 11, 12, 13, 14, 15],
    node_ids   =[1,  1,  2,  2,  3,  3],
)

# Inter edges: bidirectional physical links between endpoints
# r1:eth0 -- r2:eth0 (100G transit)
# r1:eth1 -- sw1:eth0 (10G access)
t.add_inter_edges(
    sources     =[10, 11],
    destinations=[12, 14],
)

print(t.node_count())   # 9 (3 devices + 6 endpoints)
print(t.edge_count())   # edges: 6 intra + 4 inter (2 links * 2 directions)
                        # + 2 DeviceLink shortcuts
```

5.5 Removing Nodes and Edges

Listing 7. Removing topology elements

```
# Remove specific endpoints (must have no children)
t.remove_nodes(ids=[15])  # removes endpoint 15

# Cascade removal: remove a device and all its endpoints
removed_children = t.remove_node_cascade(node_id=3)
# removed_children = [14] (endpoint 14 was sw1's only remaining port)

# Remove specific edges
t.remove_edges(from_=[10], to=[12])  # remove r1:eth0 -- r2:eth0 link
```

5.6 Querying with QuerySpec

For property-based filtering without graph traversal, use `QuerySpec`:

Listing 8. QuerySpec examples

```python
from ank_nte import QuerySpec, ExprNode

# All routers in SYD
spec = QuerySpec(
    type_filter=["Router"],
    field_filters={"pop": "SYD"},
)
syd_routers = t.execute_query(spec)  # returns list[int] of node IDs

# Routers with bandwidth > 40 Gbps using expression filter
expr = ExprNode.gt_(ExprNode.field("speed_gbps"), ExprNode.int_(40))
spec = QuerySpec(
    type_filter=["Endpoint"],
    expr_filters=[expr],
)
fast_ports = t.execute_query(spec)

# Count query
count = t.execute_query_count(spec)

# Existence check (returns immediately on first match)
exists = t.execute_query_exists(spec)
```

5.7 Querying with PatternNode

For multi-hop structural queries, use `PatternNode` and `QueryPlan`:

Listing 9. PatternNode graph traversal

```python
from ank_nte import PatternNode, PatternStep, HopSpec, QueryPlan, MaterialiseMode

# Match: Router -> (any edge) -> Router (direct device-to-device)
pattern = (
    PatternNode.binding("r", "Router")
               .then_any_edge()
               .then_binding("s", "Router")
)
plan = QueryPlan(pattern).with_mode(MaterialiseMode.collect())

result = t.execute_pattern_query(plan)
for match_tuple in result.matches:
    r_ids = match_tuple.get_binding("r")
    s_ids = match_tuple.get_binding("s")
    print(f"Router {r_ids} connects to Router {s_ids}")

# Match: paths of 1-3 hops between routers
pattern = (
    PatternNode.binding("src", "Router")
               .then_variable_path(HopSpec.range(1, 3))
               .then_binding("dst", "Router")
)
plan = QueryPlan(pattern).with_max_matches(500)
result = t.execute_pattern_query(plan)
print(f"Found {len(result.matches)} paths (truncated={result.truncated})")
```

5.8 Snapshot Management

NTE supports named in-memory snapshots. Snapshots use Arrow's Copy-on-Write semantics: taking a snapshot is effectively $O(1)$ — it clones the topology struct but the underlying Arrow buffers are shared until a write forces a copy.

Listing 10. Snapshots

```
# Save current state as a named snapshot
t.snapshot("before_maintenance")

# Make some changes
t.remove_node_cascade(node_id=2)

# List available snapshots
snapshots = t.list_snapshots()  # ["before_maintenance"]

# Restore from snapshot
t.restore_snapshot("before_maintenance")
print(t.node_count())  # back to original count

# Delete a snapshot when no longer needed
t.delete_snapshot("before_maintenance")
```

5.9 Topology Diffing

After restoring or maintaining multiple versions, you can diff two states:

Listing 11. Topology diff

```
t.snapshot("v1")

# Make changes
t.add_nodes(ids=[99], node_types=["Router"], layer="physical",
            data=[{"hostname": "r-new"}])

delta = t.diff("v1")  # returns TopologyDelta

for nd in delta.node_deltas:
    print(f"Node {nd.node_id}: {nd.operation}")  # Added/Removed/Modified

for ed in delta.edge_deltas:
    print(f"Edge {ed.from_id}->{ed.to_id}: {ed.operation}")

for pc in delta.property_changes:
    print(f"Node {pc.node_id}.{pc.field}: {pc.old_value} -> {pc.new_value}")
```

5.10 Event Subscription

The reactive event bus allows Python callbacks to be invoked synchronously after each topology mutation:

Listing 12. Event subscription

```
from ank_nte import TopologyEvent

def on_topology_change(event: TopologyEvent) -> None:
    print(f"[{event.timestamp_iso}] #{event.sequence}: {event.event_type}")
    details = event.details()
    if event.event_type == "node_created":
        print(f"  New nodes: {details['node_ids']}")

# Subscribe (returns a handle for later unsubscription)
handle = t.subscribe(on_topology_change)

# Mutations now trigger the callback
t.add_nodes(ids=[50], node_types=["Router"], layer="physical",
            data=[{"hostname": "r50"}])
# Prints: [2026-03-04T...] #1: node_created
```

```
#           New nodes: [50]

# Unsubscribe when done
t.unsubscribe(handle)
```

5.11 Error Handling

NTE exposes a rich exception hierarchy. All exceptions are subclasses of Python's built-in `Exception` and carry structured fields for programmatic inspection:

Listing 13. Exception handling

```
from ank_nte import NodeNotFoundError, LayerMismatchError, InvariantViolationError

try:
    t.remove_nodes(ids=[999])  # node doesn't exist
except NodeNotFoundError as e:
    print(e.node_id)    # 999
    print(e.operation)  # "remove_nodes"

try:
    # Attempting to connect nodes from incompatible layers
    t.add_inter_edges(sources=[10], destinations=[20])  # 20 is in a different layer
except LayerMismatchError as e:
    print(e.source_layer, e.target_layer)

try:
    # Invariant violation: graph and store out of sync
    # (This should not normally occur; if it does, it indicates a bug)
    t.verify_state_parity()
except InvariantViolationError as e:
    print(f"Graph: {e.graph_count}, Store: {e.store_count}")
```

5.12 Thread Safety and the GIL

NTE is designed for concurrent use from multiple Python threads. The implementation:

1. Every `Topology` method releases the Python GIL before acquiring the topology's `RwLock`.
2. Read operations (`execute_query`, `get_nodes`, etc.) use `RwLock::read()`, allowing concurrent reads.
3. Write operations use `RwLock::write()`, serialising all mutations.

This means the Rust engine can execute queries in parallel across threads while Python code runs elsewhere, without GIL-induced serialisation.

> **Warning**
>
> The `subscribe` / callback mechanism invokes Python callables from within the write lock. Long-running callbacks will stall all topology mutations. Keep callbacks short; use a queue or thread pool for expensive work.

5.13 Integration with ank_pydantic and ank_netcfg

When using `ank_pydantic`, you typically do not interact with `ank_nte` directly. Pydantic models are serialised to property dicts and passed in bulk to NTE methods. The `ank_pydantic` layer handles:

- ID assignment and namespace management.
- Pydantic model validation before calling NTE.

- Deserialisation of query results back into Pydantic model instances.
- Layer management and dependency declaration.

When using `ank_netcfg`, NTE is consumed as a pure Rust crate dependency (`nte-topology`). There is no Python layer involved. The `ank_netcfg` compiler reads declarative YAML blueprints, populates an NTE topology, then walks the graph to generate device configurations via its DeviceIR mapping DSL. This path is suited to deterministic, reproducible config generation in CI/CD pipelines.

5.14 Worked End-to-End Example

The following example models a small two-PoP network, queries it, makes a change, validates a policy, and inspects the diff.

Listing 14. End-to-end worked example

```
from ank_nte import (
    Topology, QuerySpec, ExprNode, PatternNode, QueryPlan,
    MaterialiseMode, NodeNotFoundError
)

# ---- 1. Build the topology ----
t = Topology()

# Two routers: one in SYD, one in MEL
t.add_nodes(
    ids=[1, 2],
    node_types=["Router", "Router"],
    layer="physical",
    data=[
        {"hostname": "r-syd-01", "as_number": 64512, "pop": "SYD"},
        {"hostname": "r-mel-01", "as_number": 64512, "pop": "MEL"},
    ]
)

# Their uplink ports
t.add_endpoints(
    ids=[10, 11, 20, 21],
    endpoint_types=["Endpoint"] * 4,
    layer="physical",
    data=[
        {"name": "et-0/0/0", "speed_gbps": 100},
        {"name": "et-0/0/1", "speed_gbps": 100},
        {"name": "et-0/0/0", "speed_gbps": 100},
        {"name": "et-0/0/1", "speed_gbps": 100},
    ]
)

# Ownership: ports -> routers
t.add_intra_edges(
    endpoint_ids=[10, 11, 20, 21],
    node_ids    =[1,  1,  2,  2],
)

# Physical link: r-syd-01:et-0/0/0 <-> r-mel-01:et-0/0/0
t.add_inter_edges(sources=[10], destinations=[20])

# ---- 2. Query: find all directly-connected router pairs ----
pattern = (
    PatternNode.binding("a", "Router")
               .then_any_edge()
               .then_binding("b", "Router")
)
```

```
plan = QueryPlan(pattern).with_mode(MaterialiseMode.collect())
result = t.execute_pattern_query(plan)

print("Connected pairs:")
for m in result.matches:
    a = m.get_binding("a")[0]
    b = m.get_binding("b")[0]
    # Fetch properties via QuerySpec
    props_a = t.query_nodes_as_structs(
        QuerySpec(id_filter=[a])
    )[0]
    print(f"  {props_a.data.get('hostname')} <-> ...")

# ---- 3. Snapshot before maintenance ----
t.snapshot("pre-maintenance")

# ---- 4. Simulate link failure: remove the transit link ----
t.remove_edges(from_=[10], to=[20])

# ---- 5. Check reachability ----
reachable = t.is_reachable(source=1, target=2)
print(f"r-syd-01 can reach r-mel-01: {reachable}")  # False

# ---- 6. Restore ----
t.restore_snapshot("pre-maintenance")
assert t.is_reachable(source=1, target=2)

# ---- 7. Diff to verify restore was clean ----
delta = t.diff("pre-maintenance")
assert len(delta.edge_deltas) == 0  # no differences

# ---- 8. Persist to disk ----
t.save("/tmp/my_topology.nte")

# ---- 9. Reload ----
t2 = Topology.load("/tmp/my_topology.nte")
assert t2.node_count() == t.node_count()
```

6 Transactions

6.1 ACID Transaction Model

NTE implements copy-on-write ACID transactions for the dual-write architecture. Every mutation can be wrapped in a transaction scope that atomically commits or rolls back changes to both the graph and the DataFrame store.

The transaction model uses Arrow's reference-counted CoW buffers for $O(1)$ snapshot creation at transaction start and $O(1)$ rollback if the transaction is aborted. This avoids the need for a separate undo log.

6.2 Isolation Levels

Three isolation levels are supported, matching standard database semantics:

Level	Behaviour
`ReadCommitted`	Sees only committed data; each read within the transaction sees the latest committed state.
`RepeatableRead`	Reads within the transaction see a consistent snapshot taken at transaction start.
`Serializable`	Full conflict detection: transactions that read/write overlapping data are detected and the later transaction is aborted.

6.3 Transaction Snapshots

Each transaction creates a `TransactionSnapshot` at begin time, using persistent data structures (`im::HashMap`) for structural sharing:

Listing 15. TransactionSnapshot (from nte-topology/src/topology/snapshot.rs)

```rust
pub struct TransactionSnapshot {
    pub adjacency: im::HashMap<NodeId, Vector<NodeId>>,
    pub node_metadata: im::HashMap<NodeId, NodeMetadata>,
    pub dataframes: Arc<DataFrameStore>,
}
```

The use of `im::HashMap` (an immutable persistent hash map) means that the snapshot shares structure with the live topology. Only modified entries are copied, giving near-constant memory overhead for transactions that touch a small fraction of the graph.

6.4 Python Transaction API

The `PyTransaction` class exposes transactions as a Python context manager:

Listing 16. Transaction API

```python
from ank_nte import Topology, IsolationLevel

t = Topology()
# ... populate topology ...

# Context manager: auto-rollback on exception
with t.begin_transaction(IsolationLevel.Serializable) as txn:
    t.add_nodes(ids=[100], node_types=["Router"], layer="physical",
                data=[{"hostname": "r-new"}])
    t.add_inter_edges(sources=[10], destinations=[100])
    txn.commit()  # explicit commit

# If an exception occurs before commit(), the transaction
# is automatically rolled back (both graph and DataFrame store).

# Manual rollback
with t.begin_transaction() as txn:
    t.remove_node_cascade(node_id=1)
    if not t.is_reachable(source=2, target=3):
        txn.rollback()  # revert: removal would partition the graph
    else:
        txn.commit()
```

> **Warning**
>
> NTE transactions are in-memory only. There is no write-ahead log (WAL): a process crash during a transaction leaves the in-memory state inconsistent. For durable crash consistency, use periodic disk snapshots or the Raft distributed deployment (Section 13).

6.5 Conflict Detection

When multiple transactions execute concurrently, NTE detects conflicts using a `VersionTracker` that records per-node and per-edge version numbers. At commit time, the tracker compares each transaction's read and write sets against committed versions:

Write–write conflicts (all isolation levels): if a node or edge was modified by another transaction after the current transaction began, a `ConflictError` is raised.

Read–write conflicts (Serializable only): if a node that was *read* by the current transaction was subsequently *written* by another committed transaction, a `ConflictError` is raised. Read-your-own-writes are excluded from this check.

Conflict errors carry the transaction ID, the list of conflicting node/edge IDs, and the conflict type (`WriteWrite` or `ReadWrite`), allowing callers to implement application-level retry or merge logic.

7 Persistence and Snapshots

7.1 Named Snapshots

As shown in Section 5, NTE supports named in-memory snapshots managed by `SnapshotManager`. The key implementation detail is that Polars DataFrames use Apache Arrow's reference-counted buffers. Cloning a DataFrame shares the underlying Arrow buffers; a copy is only made when a buffer needs to be mutated (CoW semantics).

This means:

- Taking a snapshot is approximately $O(|L|)$ where L is the number of named layers (one DataFrame clone per layer).
- Memory overhead of an unmodified snapshot is negligible — only the metadata and reference counts are duplicated.
- Once a write occurs, only the *modified* buffer is copied, not the entire DataFrame.

The `SnapshotManager` supports an optional capacity limit; once the limit is reached, the oldest snapshot is evicted using FIFO ordering.

7.2 File Serialisation

NTE provides two file serialisation formats:

`.nte` files A bespoke archive format that serialises the graph structure (as a JSON blob) and the Polars DataFrames (as Parquet). Files are loaded with `Topology.load(path)` and saved with `t.save(path)`. This is the recommended format for operational use.

In-memory bytes `t.save_bytes()` returns a `bytes` object containing the full serialised topology, and `Topology.load_bytes(data)` deserialises it. Useful for network transfer or embedding in larger artefacts.

7.3 CSR Serialisation for Archival

For high-performance batch traversal and long-term archival, NTE can serialise the graph as a Compressed Sparse Row (CSR) structure using the `rkyv` [4] zero-copy serialisation framework:

Listing 17. CsrGraph structure (from nte-graph/src/csr.rs)

```rust
#[derive(Debug, rkyv::Archive, rkyv::Serialize, rkyv::Deserialize)]
pub struct CsrGraph {
    // Outgoing adjacency
    pub out_offsets:      Vec<u32>,  // CSR row offsets
    pub out_neighbors:    Vec<i32>,  // column indices (external IDs)
    pub out_edge_indices: Vec<u32>,  // petgraph EdgeIndex values
    pub out_kinds:        Vec<u8>,   // edge kind tag (0..9)
    // Incoming adjacency (for reverse traversal)
    pub in_offsets:       Vec<u32>,
    pub in_neighbors:     Vec<i32>,
    pub in_edge_indices:  Vec<u32>,
    pub in_kinds:         Vec<u8>,
}
```

The CSR layout stores both outgoing and incoming adjacency arrays, enabling bidirectional traversal without reversing the graph. The `rkyv` library serialises the structure to a flat byte buffer with no padding or relocation, allowing it to be memory-mapped directly without deserialisation cost.

7.4 Memory-Mapped Loading

The `MmapCsrGraph` type wraps a `memmap2::Mmap` and provides a zero-copy view onto an `ArchivedCsrGraph` stored on disk:

Listing 18. Loading a CSR graph via mmap

```rust
let graph = MmapCsrGraph::load_from_disk("/data/topology.csr")?;
// graph.view() returns &ArchivedCsrGraph with no copying
let node_count = graph.view().node_count();
```

Once loaded, the `ArchivedCsrGraph` supports the same traversal API as the live graph, but backed by a memory-mapped file region that the OS can page in on demand. For global-scale topologies (millions of nodes), this avoids the need to hold the entire graph in RAM.

7.5 Use Cases for Historical Querying

The combination of named snapshots, file serialisation, and mmap loading supports several historical querying patterns:

- **Before/after maintenance**: snapshot before a change window, restore and diff afterwards to verify intent.
- **Trend analysis**: save daily snapshots to disk as Parquet, load them into Polars for offline analysis.
- **Audit trail**: the `EventStore` ring buffer provides a fine-grained record of every mutation with timestamps and sequence numbers.
- **Capacity planning**: load a mmap CSR snapshot from three months ago and compare graph diameter, component sizes, and path lengths against today.

8 GPU Acceleration

8.1 Architecture Overview

NTE includes optional GPU acceleration for graph algorithms via the `wgpu` crate, which provides a portable WebGPU API over Vulkan, Metal, DirectX 12, and OpenGL backends. GPU compute shaders are written in WGSL (WebGPU Shading Language). Two algorithms are currently GPU-accelerated:

1. **Single-Source Shortest Path (SSSP)** — parallel Bellman-Ford relaxation.
2. **Connected Components** — label propagation algorithm.

8.2 Hardware Acceleration State

The `HardwareAccelerationState` enum captures the GPU availability:

Listing 19. GPU state detection

```rust
pub enum HardwareAccelerationState {
    Wgpu(Device, Queue),  // GPU available; wgpu Device and Queue
    CpuOnly,              // No compatible GPU; Rayon CPU fallback
}

impl HardwareAccelerationState {
    pub fn detect_hardware_acceleration() -> Self {
        // Requests a high-performance wgpu adapter.
        // Falls back to CpuOnly if none found or device init fails.
    }
}
```

`detect_hardware_acceleration()` is called once at engine initialisation. The result is stored and used to route algorithm calls to either the GPU kernel or the CPU fallback. This means the same binary runs on hardware with or without a GPU, with no conditional compilation.

8.3 SSSP Kernel

The SSSP kernel implements a parallel Bellman-Ford iteration. Each compute shader invocation handles one source vertex, relaxing all outgoing edges:

Listing 20. SSSP WGSL kernel (excerpt)

```
@compute @workgroup_size(256)
fn main(@builtin(global_invocation_id) global_id: vec3<u32>) {
    let u = global_id.x;
    if (u >= info.node_count) { return; }

    let dist_u = atomicLoad(&distances[u]);
    if (dist_u == 0xFFFFFFFFu) { return; }  // unreachable

    let start = out_offsets[u];
    let end   = out_offsets[u + 1u];

    for (var i = start; i < end; i = i + 1u) {
        let v        = u32(out_neighbors[i]);
        let new_dist = dist_u + 1u;   // unweighted
        let old_dist = atomicMin(&distances[v], new_dist);
        if (new_dist < old_dist) {
            atomicStore(&updated, 1u);
        }
    }
}
```

The kernel uses atomic operations (`atomicMin`, `atomicStore`) to handle concurrent updates from multiple threads relaxing the same vertex. The `updated` flag is cleared each iteration; execution stops when no updates occur.

8.4 Connected Components Kernel

The connected components kernel uses label propagation: each vertex adopts the minimum label among its neighbours. Convergence is detected when no label changes in an iteration:

Listing 21. Connected components WGSL kernel (excerpt)

```
@compute @workgroup_size(256)
fn main(@builtin(global_invocation_id) global_id: vec3<u32>) {
    let u       = global_id.x;
    let label_u = atomicLoad(&labels[u]);

    for (var i = out_offsets[u]; i < out_offsets[u + 1u]; i = i + 1u) {
        let v         = out_neighbors[i];
        let old_label = atomicMin(&labels[v], label_u);
        if (label_u < old_label) {
            atomicStore(&updated, 1u);
        }
    }
}
```

8.5 CPU Fallback

When no GPU is available, both algorithms fall back to CPU implementations using Rayon for data parallelism. The Rayon pool matches the number of logical CPU cores. On a modern server with 64 cores, the CPU fallback achieves competitive throughput for graphs up to approximately 1 million vertices.

8.6 When to Use GPU Acceleration

Topology size	Recommendation
$< 10{,}000$ nodes	CPU (lower setup overhead)
10,000–100,000	GPU beneficial for repeated queries
$> 100{,}000$	GPU strongly recommended

The GPU incurs a fixed setup cost: uploading the CSR adjacency arrays to GPU memory and compiling the WGSL shader. This is amortised across repeated queries on the same topology. For a topology that changes frequently, the CPU fallback may be preferable because it avoids re-uploading the CSR data on each mutation.

8.7 Invoking GPU Algorithms from Python

Listing 22. GPU-accelerated graph algorithms

```
# Build an AnalysisGraph (undirected view, optional port collapsing)
ag = t.build_undirected_graph(layer="physical", collapse_endpoints=True)

# Shortest path (routes automatically to GPU or CPU)
path = ag.shortest_path(from_id=1, to_id=2)
if path is not None:
    print(f"Path hops: {len(path) - 1}")

# Connected components
components = ag.connected_components()
```

```
print(f"Number of connected components: {len(components)}")
print(f"Largest component size: {max(len(c) for c in components)}")

# Check full connectivity
if not ag.is_connected():
    print("WARNING: topology is partitioned")

# Find single points of failure
bridges = ag.bridges()
art_pts = ag.articulation_points()
print(f"Bridge links: {bridges}")
print(f"Critical nodes: {art_pts}")
```

9 Graph Algorithm Plugins

9.1 Plugin Architecture

The `nte-query` crate includes an extensible algorithm plugin system. Each algorithm registers under a namespace (e.g. `algo.pathfinding.k_shortest`) and receives the topology graph and named arguments. Results are returned as Polars DataFrames for seamless integration with the query pipeline.

9.2 K-Shortest Paths

The `k_shortest` plugin implements Yen's algorithm (via `petgraph::algo`) to find the k shortest paths between two nodes. It registers as `algo.pathfinding.k_shortest` and accepts `source`, `target`, and `k` arguments. Results are returned as a Polars DataFrame with columns `path_index` and `cost`.

9.3 Single Point of Failure Detection

The `spof` plugin identifies articulation points — vertices whose removal would disconnect the graph. It registers as `algo.vulnerability.spof` and returns a DataFrame with a `node_id` column listing vulnerable nodes.

> **Note**
>
> Algorithm plugins are currently available at the Rust API level only. Python bindings for the plugin registry are planned. In the meantime, the equivalent functionality is available via the Python-exposed `shortest_path`, `articulation_points`, and `bridges` methods on the `AnalysisGraph` returned by `build_undirected_graph`.

These algorithms complement the GPU-accelerated SSSP and connected components (Section 8) and the pattern-based queries (Section 4).

10 Policy Engine

10.1 Design Rationale

Network topologies must satisfy design-rule constraints: every edge device must have at least two uplinks, no two devices in the same rack should be connected to the same spine, all transit links must have ≥ 100 Gbps speed. Encoding these constraints as ad-hoc Python checks is fragile and scattered across the codebase.

NTE externalises design-rule contracts into a declarative policy layer [5], separating *what the constraint is* from *when it is evaluated*. Policies can be evaluated on demand, before a mutation (what-if), or as a continuous validation step in a CI pipeline.

10.2 Policy File Format

Policies are defined in YAML files with the following schema:

Listing 23. Policy file format (YAML)

```yaml
version: 1
policies:
  - id: "P001"
    category: "attribute"     # or "structural"
    severity: "ERROR"         # or "WARNING"
    expr: "device.speed_gbps >= 100"
    message: "All devices must support 100G"
    repair_hints:
      - "Upgrade device hardware"
      - "Replace with 100G-capable model"

  - id: "P002"
    category: "structural"
    severity: "ERROR"
    expr: "transit_exit_count >= 2"
    message: "Every PoP must have at least 2 transit exits"
    repair_hints:
      - "Add a second transit provider"
```

Two policy categories are supported:

`attribute` Evaluated per node/edge. The expression is checked against each entity's property map independently.

`structural` Evaluated once against global topology metrics (counts, ratios, aggregate properties).

10.3 The Starlark Policy Engine

For policies requiring programmatic logic beyond simple comparisons, NTE includes a Starlark-based policy engine. Starlark is a Python-like scripting language designed for safe, deterministic execution in embedded environments. It was chosen because:

- **Sandboxed**: no filesystem access, no network access, no mutable global state.
- **Deterministic**: no random number generation, no timing functions.
- **Familiar**: syntax is a subset of Python, comfortable for network engineers.
- **Fast**: scripts are parsed and compiled to bytecode once, then executed against each context.

Listing 24. Starlark policy example: uplink redundancy

```python
# Policy: every leaf switch must have at least 2 uplinks
def validate_leaf_uplinks(node_id, uplink_count, required):
    if uplink_count < required:
        fail("Node {} has only {} uplinks, need {}".format(
            node_id, uplink_count, required
        ))
    return True

# Host function bound from Rust: check_uplinks(node_id, required)
# Returns True if the node has >= required uplinks
result = check_uplinks("leaf-01", 2)
result  # Starlark scripts return the value of their last expression
```

Host functions are registered via the `#[starlark_module]` macro in Rust:

Listing 25. Registering host functions in the Starlark engine

```rust
#[starlark_module]
pub fn nte_graph_funcs(builder: &mut GlobalsBuilder) {
    fn check_uplinks(node_id: &str, required: i32) -> anyhow::Result<bool> {
        // Performs a petgraph degree lookup against the current Topology context.
        // Returns true if the node has >= required uplinks.
        ...
    }
}
```

10.4 Policy Evaluation

The `PolicyEngine` struct pre-compiles all policy expressions at load time (using CEL — Common Expression Language — for attribute policies, Starlark for procedural policies). At evaluation time, only execution is performed:

1. Structural policies are evaluated once against a global `RuntimeContext` containing aggregate topology metrics.
2. Attribute policies are evaluated per node/edge, with the entity's property map injected as variables.
3. Each violation produces a `Finding` with: policy ID, severity, human-readable message, optional subject entity ID, and repair hints.

10.5 Shadow Topologies and DeltaNet Validation

NTE supports *what-if* analysis via `ShadowTopology`: a zero-copy clone of the live topology onto which a proposed `TopologyDelta` is applied without affecting the real state. Because Polars DataFrames use Arrow CoW buffers, creating a shadow is effectively $O(1)$ in memory.

The `DeltaNetValidator` (in `nte-policy`) combines shadow topologies with the Starlark policy engine to validate intent *before* committing a change:

1. The caller constructs a `TopologyDelta` describing the proposed change.
2. A `ShadowTopology` is created from the current topology plus the delta.
3. The validator runs all registered Starlark policies against the shadow.
4. If any policy fails, the change is rejected before it touches the live topology.

Listing 26. DeltaNet pre-commit validation

```rust
let shadow = ShadowTopology::new(&topology, &proposed_delta)?;
let validator = DeltaNetValidator::new();
validator.validate(&shadow, &["transit_redundancy.star",
                              "no_spof.star"])?;
// Only if all policies pass:
topology.apply(proposed_delta)?;
```

This pattern is particularly useful in CI/CD pipelines where a proposed topology change (encoded as a delta) must pass all design-rule constraints before being merged.

10.6 Worked Policy Examples

10.6.1 Transit Redundancy

Listing 27. Transit redundancy policy

```python
# Every PoP must have at least 2 transit exit links
def check_transit_redundancy(pop_name, exit_count):
    if exit_count < 2:
        fail("PoP {} has only {} transit exits (minimum 2)".format(
```

```
            pop_name, exit_count
        ))
    return True

check_transit_redundancy(pop_name, transit_exit_count(pop_name))
```

10.6.2 Peer Diversity

Listing 28. CEL attribute policy: peer diversity

```
# P003: No two BGP sessions should share the same peer AS (from a given router)
# Expressed as a structural policy on the aggregated adjacency matrix
version: 1
policies:
  - id: "P003"
    category: "structural"
    severity: "WARNING"
    expr: "max_sessions_per_peer_as <= 1"
    message: "Router has multiple BGP sessions to the same peer AS"
    repair_hints:
      - "Use route reflectors or consolidate peering"
```

10.6.3 No Single Point of Failure

Listing 29. No single point of failure policy

```
# All routers in the critical path must have >= 2 uplinks
def check_no_spof(critical_nodes):
    for node_id in critical_nodes:
        uplinks = check_uplinks(node_id, 2)
        if not uplinks:
            fail("Critical node {} is a single point of failure".format(node_id))
    return True

# critical_path_nodes() is a host function that returns the set of
# articulation points in the physical topology
check_no_spof(critical_path_nodes())
```

11 Reactive Events

11.1 Event Types

NTE emits a `TopologyEvent` for every topology mutation. The full event taxonomy:

Event	Trigger
`NodeCreated`	`add_nodes`
`EndpointCreated`	`add_endpoints`
`NodeRemoved`	`remove_nodes`, `remove_node_cascade`
`EndpointRemoved`	endpoint removal
`NodeUpdated`	`update_node_field`, `update_nodes_batch`
`InterEdgeCreated`	`add_inter_edges`
`IntraEdgeCreated`	`add_intra_edges`
`IntranodeEdgeCreated`	`add_intranode_edges`
`ParentEdgeCreated`	`add_parents`
`RootEdgeCreated`	`add_roots`
`InterEdgeRemoved` / …	corresponding removal methods
`EdgeUpdated`	`update_link_field`
`LayerCreated`	`create_layer`
`LayerRemoved`	`remove_layer`
`GraphTransform`	split, merge, explode, collapse

Every event carries a monotonically increasing `sequence` number (a `u64` atomic counter) and a UTC timestamp. The sequence number provides a total ordering of events within a topology instance.

11.2 Python Subscription API

Listing 30. Event subscription patterns

```python
from ank_nte import TopologyEvent
import logging

log = logging.getLogger("topology.events")

# Basic subscription
def audit_log(event: TopologyEvent) -> None:
    log.info("AUDIT: %s seq=%d ts=%s payload=%s",
             event.event_type, event.sequence,
             event.timestamp_iso, event.to_json())

handle = t.subscribe(audit_log)

# Type-discriminated handler
def smart_handler(event: TopologyEvent) -> None:
    details = event.details()
    match event.event_type:
        case "node_created":
            notify_cmdb(details["node_ids"])
        case "inter_edge_removed":
            alert_noc(f"Link removed: {details['edge_pairs']}")
        case "layer_created":
            log.info("New layer: %s", details["layer_name"])

# Unsubscribe
t.unsubscribe(handle)
```

11.3 Subscription Patterns

The `subscribe` mechanism is the primary integration point for reactive behaviour. Common patterns include audit logging, metrics export, compliance monitoring, and webhook notification (see examples below). For high-frequency mutations (e.g. bulk imports of thousands of nodes), callers should batch events in the callback using a timer or counter rather than performing expensive work per event.

11.4 Use Cases

Prometheus integration

```
import prometheus_client as prom

node_gauge = prom.Gauge("nte_node_count", "Total nodes in topology")
link_gauge = prom.Gauge("nte_link_count", "Total links in topology")

def update_metrics(event: TopologyEvent) -> None:
    node_gauge.set(t.node_count())
    link_gauge.set(t.link_count())

t.subscribe(update_metrics)
```

Webhook notification

```
import httpx

def webhook_notify(event: TopologyEvent) -> None:
    if event.event_type in ("node_created", "node_removed"):
        httpx.post("https://cmdb.example.com/webhook",
                   json={"event": event.to_json()},
                   timeout=2.0)

t.subscribe(webhook_notify)
```

Compliance monitoring

```
from ank_nte import PolicyEngine

policy_engine = PolicyEngine.load("policies/design-rules.yaml")

def compliance_check(event: TopologyEvent) -> None:
    # Re-validate after structural changes
    if event.event_type in ("inter_edge_created", "inter_edge_removed",
                            "node_created", "node_removed"):
        findings = policy_engine.validate(t)
        if findings:
            for f in findings:
                log.warning("POLICY VIOLATION [%s] %s: %s",
                            f.severity, f.policy_id, f.message)

t.subscribe(compliance_check)
```

12 Production Diagnostics

12.1 Health and Memory Endpoints

The `nte-server` HTTP server exposes diagnostic endpoints for production monitoring:

Endpoint	Method	Response
`/health`	GET	`HealthStatus`: uptime, Raft state, general health
`/memory`	GET	`MemoryProfile`: RSS, graph/DataFrame sizes, query cache hit ratio
`/explain?query=...`	GET	HTML page with the Mermaid-rendered query execution DAG

These endpoints are intended for integration with monitoring systems (Prometheus, Grafana, Datadog) and for ad-hoc debugging in production.

12.2 Chaos Engineering Endpoints

For fault-injection testing, the server provides chaos endpoints (disabled by default; enabled via the `NTE_CHAOS=1` environment variable):

Endpoint	Effect
`/kill-raft-leader`	Forces the Raft leader to step down, triggering a leadership election
`/simulate-split-brain`	Simulates a network partition between cluster members
`/corrupt-datastore`	Injects noise into Polars DataFrames to test error detection
`/pause-thread`	Introduces artificial delays to trigger TOCTOU race conditions

> **Warning**
>
> Chaos endpoints are destructive by design. They are intended solely for testing environments. The `NTE_CHAOS` flag should never be set in production.

13 Distributed Deployment

13.1 When to Use Clustering

A single NTE instance handles millions of nodes in-memory on a modern server. Clustering is appropriate when:

- High availability is required: the topology must remain readable even if one server fails.
- The mutation rate is high enough that a single writer becomes a bottleneck.
- Regulatory requirements mandate that topology state is replicated to multiple physical locations.

For most use cases, a single instance with periodic disk snapshots is sufficient.

13.2 Raft Consensus via OpenRaft

The `nte-server` crate provides a standalone HTTP/WebSocket server (built on Axum and Tokio) that includes an OpenRaft-based [9] distributed consensus layer [8]. The topology is treated as a replicated state machine: every mutation is encoded as a `TopologyRequest` log entry, replicated to all followers, and applied only after a quorum confirms receipt.

Listing 31. Raft request types (from nte-server/src/raft/types.rs)

```
#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum TopologyRequest {
    AddNode     { id: i32, node_type: String, layer: String },
    AddEdge     { from: i32, to: i32 },
    RemoveNode  { id: i32 },
    UpdateProperties { id: i32, json_payload: String },
}
```

The Raft type configuration uses:

- `NodeId = u64` — cluster member identifier.
- `D = TopologyRequest` — the command type replicated by Raft.
- `R = TopologyResponse` — the response returned to the client.
- `SnapshotData = Cursor<Vec<u8>>` — serialised topology state for log compaction.

13.3 Configuration

A three-node cluster is the recommended minimum for Raft:

Listing 32. Cluster startup (environment variables)

```
# Node 1 (leader candidate)
NTE_NODE_ID=1
NTE_RPC_ADDR=10.0.0.1:9001
NTE_API_ADDR=10.0.0.1:8080
NTE_PEERS=10.0.0.2:9001,10.0.0.3:9001

# Node 2
NTE_NODE_ID=2
NTE_RPC_ADDR=10.0.0.2:9001
NTE_API_ADDR=10.0.0.2:8080
NTE_PEERS=10.0.0.1:9001,10.0.0.3:9001
```

13.4 Leader and Follower Roles

Leader Accepts all write requests. Replicates log entries to followers. There is exactly one leader at any time.

Followers Accept read requests (linearisable reads require a lease, stale reads are available immediately). Apply log entries from the leader.

Candidate Transient state during leader election.

Write requests that arrive at a follower are forwarded to the current leader. Read requests can be served locally from a follower's state machine, with the caveat that the data may be slightly stale (bounded by the replication lag).

13.5 Snapshot Transfer

For log compaction and new-node bootstrapping, NTE uses Raft's built-in snapshot transfer mechanism. The state machine serialises the topology to a byte buffer (using the `save_bytes` format),

sends it to the new follower via the Raft snapshot API, and the follower installs it by calling `Topology.load_bytes`.

> **Warning**
>
> The distributed Raft implementation in `nte-server` is currently experimental. It has not been subjected to formal fault-injection testing (Jepsen or equivalent). Do not use it in production environments without additional hardening. See Section 14 for the current known gaps.

14 Limitations and Future Work

14.1 Current Limitations

Cyclic-pattern WCOJ. NTE now implements Worst-Case Optimal Join (WCOJ) [7] via the LeapFrog TrieJoin algorithm for cyclic query patterns (e.g. triangles, squares). This avoids the large intermediate results that a sequence of binary joins would produce. The implementation uses sorted iterator intersection and is integrated into the pattern matcher. Performance on dense cycle topologies (e.g. full-mesh data centre fabrics) is significantly improved over the earlier binary-join decomposition.

Raft not fully hardened. The OpenRaft integration passes basic correctness tests but has not been evaluated under adversarial network conditions (network partition, clock skew, message loss). Lease-read semantics (allowing followers to serve reads with a bounded staleness guarantee without contacting the leader) are not yet implemented.

ExprNode coverage. The filter executor now handles all expression variants that are meaningful in a flat `QuerySpec` context: `Comparison`, `Logical`, `Not`, `IsNull`, `IsNotNull`, `IsIn`, all four `StringOp` variants (Contains, StartsWith, EndsWith, Matches), and `Arithmetic` (+, −, *, /). Two variants—`BindingId` and `InQuery`—are inherently multi-stage operations that require subquery materialisation; these are supported only in pattern-query execution, not in flat `QuerySpec` filters. The KuzuDB backend integration would handle these natively via Cypher translation.

Transactional mutations (recently added). NTE now implements copy-on-write transaction semantics for the dual-write architecture. Mutations are wrapped in a transaction scope that automatically rolls back both the graph and DataFrame store if either side fails, using Arrow's reference-counted CoW buffers for O(1) rollback cost. State parity verification is enforced as an assertion in tests. However, NTE still does not implement a write-ahead log (WAL): a process crash *during* a mutation leaves the in-memory state inconsistent. For durable crash consistency, use periodic disk snapshots or the Raft distributed deployment (Section 13).

Property column proliferation. Because properties are stored as wide DataFrames (one column per field name), topologies with highly heterogeneous node types accumulate many columns, most of which are null for any given row. Polars handles this gracefully, but very wide DataFrames (>1000 columns) may show increased memory overhead.

GIL-released callback limitation. The event subscription callbacks are invoked from within the write lock, synchronously, before the Python GIL is reacquired. This means callbacks cannot safely call back into NTE (deadlock) and must complete quickly.

14.2 Planned Improvements

Recently completed.

- **WCOJ pattern matching** (*done*): LeapFrog TrieJoin is now integrated into the pattern matcher for cyclic query patterns.
- **Transaction semantics** (*done*): full ACID transactions with copy-on-write rollback, isolation levels (ReadCommitted, RepeatableRead, Serializable), conflict detection, and a Python context manager API (Section 6).
- **Query executor** (*done*): the executor is fully functional with ranked filter plans, hybrid graph/property execution, and Polars LazyFrame compilation.
- **Python fluent query builder** (*done*): chainable `NodeQuery`/`LinkQuery` API with expression DSL, independent of `ank_pydantic` (Section 4.10).
- **NTE-QL REPL and TUI** (*done*): interactive query shell with `EXPLAIN VISUAL` support and a real-time system dashboard (Section 4.12).
- **Query plan visualisation** (*done*): Mermaid DAG rendering of execution plans (Section 4.11).
- **Graph algorithm plugins** (*done*): K-shortest paths and SPOF detection as registered algorithm plugins (Section 9).
- **Production diagnostics** (*done*): health, memory, and explain HTTP endpoints plus chaos engineering endpoints for fault-injection testing (Section 12).
- **ClickHouse telemetry** (*scaffolded*): `MetricProvider` trait implementation for federated telemetry queries against ClickHouse TSDB, enabling dynamic edge weights in pathfinding.
- **RoaringBitmap candidate pruning** (*done*): `HashSet<i32>` replaced with `RoaringBitmap` in the query matcher for more compact, cache-friendly candidate sets.
- **Parallel path expansion** (*done*): `rayon`-parallelised multi-seed structural expansion in path queries, with validated thread safety.
- **GIL release audit** (*done*): all O(N) `PyTopology` methods now release the GIL via `py.allow_threads`.
- **Structural metadata mirroring** (*done*): lightweight node/edge metadata stored directly in petgraph weights for early pruning during traversals.

Remaining planned improvements.

- **Raft hardening**: Jepsen test suite, lease reads, poison-pill detection.
- **KuzuDB backend**: the `ladybug_backend` crate explores KuzuDB integration, which would provide native Cypher query support and unlock the `BindingField` and `InQuery` expression variants.
- **Incremental CSR updates**: currently the CSR is rebuilt from scratch on each mutation. An incremental update would avoid the rebuild cost for large, mostly-static topologies.
- **Zero-Copy Arrow FFI**: share Polars DataFrames with Python using Arrow C Data Interface pointers to eliminate serialisation overhead.
- **Z3 formal verification**: the `z3-verification` feature flag in `nte-query` scaffolds a Z3-based constraint solver for translating the topology into Datalog Horn clauses and mathematically proving invariants (VLAN isolation, loop-freedom). The module structure is in place but integration with the policy engine is not yet complete.
- **LSP support**: `nte-lsp` provides a Language Server Protocol implementation for NTE-QLquery editing with IDE autocomplete and diagnostics.
- **Write-ahead log**: an on-disk WAL would provide crash consistency without requiring the full Raft distributed deployment.

14.3 Known Performance Trade-offs

Table 1 summarises Criterion micro-benchmarks collected on an Apple M-series laptop (single-threaded, best of 100 samples). All times are wall-clock medians.

Table 1. Representative micro-benchmarks (Criterion, 100 samples).

Operation	Complexity	100	1 000	10 000
`topology_build`	$O(n)$	34 µs	305 µs	—
`add_nodes` (batch)	$O(n)$	8.4 µs	105 µs	1.6 ms
`query_execute_spec`	$O(n)$	174 µs	278 µs	387 µs
`topology_validate`	$O(n)$	160 µs	2.2 ms	—
`peers_lookup`	$O(1)$	25 ns	25 ns	25 ns
`is_endpoint_lookup`	$O(1)$	10 ns	10 ns	10 ns
`cache_warm` (5 000 nodes)				403 µs
`cache_cold` (5 000 nodes)				454 µs

Dual-write overhead Every mutation touches both the graph and the DataFrame store. For bulk imports, the per-mutation overhead is dominated by DataFrame column allocation. The batch APIs (`add_nodes`, `add_edges`) amortise this cost significantly.

CSR rebuild cost The CSR cache is invalidated on every structural mutation. For a topology of 100,000 nodes, a rebuild takes approximately 20–50 ms. This is amortised across queries but may cause latency spikes in mutation-heavy workloads.

Snapshot memory Named snapshots share Arrow buffers via CoW. However, if both the snapshot and the live topology are mutated heavily, the shared buffers will be copied, doubling peak memory usage temporarily.

A NTE-QL Grammar (EBNF)

The following EBNF grammar describes the NTE-QLquery language. Terminals are in `UPPERCASE` or quoted strings; non-terminals are in *italics*.

```
query ::= match_clause
          where_clause?
          return_clause
          order_clause?
          skip_clause?
          limit_clause?

match_clause    ::= MATCH pattern (',' pattern)*
                  | OPTIONAL MATCH pattern (',' pattern)*

pattern         ::= node_pattern ( rel_pattern node_pattern )*
                  | 'shortestPath' '(' node_pattern rel_pattern node_pattern ')'
                  | 'allShortestPaths' '(' node_pattern rel_pattern node_pattern ')'

node_pattern    ::= '(' binding? (':' label ('|' label)*)? props? ')'
binding         ::= IDENTIFIER
label           ::= IDENTIFIER
props           ::= '{' prop_pair (',' prop_pair)* '}'
prop_pair       ::= IDENTIFIER ':' literal

rel_pattern     ::= '-' '[' rel_detail? ']' '-'      (* undirected *)
                  | '-' '[' rel_detail? ']' '->'      (* directed right *)
                  | '<-' '[' rel_detail? ']' '-'      (* directed left *)
```

```
rel_detail      ::= binding? (':' rel_type ('|' rel_type)*)?
                    ('*' hop_spec)?
hop_spec        ::= INT                              (* exact N hops *)
                  | INT '..' INT                     (* min..max hops *)
                  | '..'                             (* any hops *)
rel_type        ::= IDENTIFIER

where_clause    ::= WHERE expr

return_clause   ::= RETURN DISTINCT? return_item (',' return_item)*
return_item     ::= '*'
                  | expr (AS IDENTIFIER)?

order_clause    ::= ORDER BY sort_item (',' sort_item)*
sort_item       ::= expr (ASC | DESC)?

skip_clause     ::= SKIP INT
limit_clause    ::= LIMIT INT

expr ::= or_expr

or_expr  ::= and_expr (OR and_expr)*
and_expr ::= not_expr (AND not_expr)*
not_expr ::= NOT not_expr
           | cmp_expr

cmp_expr ::= add_expr (cmp_op add_expr)?
           | add_expr IS NULL
           | add_expr IS NOT NULL
           | add_expr IN '[' literal_list ']'
           | EXISTS '{' match_clause where_clause? '}'

cmp_op ::= '=' | '<>' | '!=' | '<' | '<=' | '>' | '>='

add_expr ::= mul_expr (('+' | '-') mul_expr)*
mul_expr ::= unary_expr (('*' | '/') unary_expr)*
unary_expr ::= '-' unary_expr | atom

atom ::= literal
       | property_access
       | function_call
       | '(' expr ')'
       | binding_ref

property_access ::= IDENTIFIER '.' IDENTIFIER
binding_ref     ::= IDENTIFIER

function_call ::= IDENTIFIER '(' (expr (',' expr)*)? ')'
                (* Built-in functions: COUNT, SUM, AVG, MIN, MAX,
                   nodes(), relationships(), length(),
                   shortestPath(), allShortestPaths() *)

literal_list ::= literal (',' literal)*
literal      ::= INTEGER | FLOAT | QUOTED_STRING | TRUE | FALSE | NULL

IDENTIFIER   ::= [a-zA-Z_][a-zA-Z0-9_]*
INTEGER      ::= '-'? [0-9]+
FLOAT        ::= '-'? [0-9]+ '.' [0-9]+
QUOTED_STRING ::= '"' [^"]* '"' | "'" [^']* "'"
TRUE         ::= 'true' | 'TRUE'
FALSE        ::= 'false' | 'FALSE'
NULL         ::= 'null' | 'NULL'
```

B Python API Quick Reference

B.1 Topology Class

Method signature	Description
`node_count() -> int`	Total vertices (nodes + endpoints)
`edge_count() -> int`	Total edges (all kinds)
`get_nodes() -> list[int]`	All external node IDs
`add_nodes(ids, node_types, layer, data)`	Add device nodes in bulk
`add_endpoints(ids, endpoint_types, layer, data)`	Add endpoints in bulk
`remove_nodes(ids)`	Remove nodes (fails if has children)
`remove_node_cascade(node_id)`	Remove node and all descendants
`remove_node_reparent(node_id)`	Remove node, promote children
`remove_edges(from_, to)`	Remove edges by endpoint pairs
`add_inter_edges(sources, destinations)`	Add bidirectional connectivity links
`add_intra_edges(endpoint_ids, node_ids)`	Add ownership edges
`add_intranode_edges(sources, destinations)`	Add intra-chassis edges
`add_parents(children, parents)`	Add parent hierarchy edges
`peers(external_id) -> list[int]`	Get neighbouring nodes
`get_parent_nodes(node_ids)`	Get immediate parent of each node
`get_root_nodes(node_ids)`	Get root of each node's hierarchy
`get_ancestor_in_layer(node_id, layer)`	Walk up to find ancestor in layer
`get_descendant_in_layer(node_id, layer)`	Walk down to find descendant
`update_node_field(node_id, field, value)`	Update a single property
`update_nodes_batch(updates)`	Batch property updates
`execute_query(spec) -> list[int]`	Run a QuerySpec, return IDs
`execute_query_count(spec) -> int`	Count matching nodes
`execute_query_exists(spec) -> bool`	Check if any matches exist
`execute_pattern_query(plan)`	Run a PatternNode query
`execute_link_query(spec) -> list[int]`	Query links by LinkQuerySpec
`query_nodes_as_structs(spec)`	Return full RustNode structs
`shortest_path(src, dst, layer)`	BFS shortest path (hop count)
`shortest_path_weighted(src, dst, weight, layer)`	Dijkstra shortest path
`all_paths(src, dst, max_len, ...)`	All paths up to max length

B.2 QuerySpec Constructor

```
QuerySpec(
    type_filter:          list[str] | None = None,
    layer_filter:         list[str] | None = None,
    kind_filter:          list[str] | None = None,    # "device" or "endpoint"
    id_filter:            list[int] | None = None,
    field_filters:        dict[str, Any] | None = None,
    expr_filters:         list[ExprNode] | None = None,
    exclude_logical_nodes: bool = True,
    exclude_special_layers: bool = True,
    isolated_filter:      bool = False,
    connected_filter:     bool = False,
    sort_field:           str | None = None,
    sort_descending:      bool = False,
)
```

B.3 ExprNode Static Methods

Method	Returns
`ExprNode.field(name)`	Property reference
`ExprNode.int_(value)`	Integer literal
`ExprNode.float_(value)`	Float literal
`ExprNode.string(value)`	String literal
`ExprNode.bool_(value)`	Boolean literal
`ExprNode.null()`	Null literal
`ExprNode.eq_(left, right)`	Equality comparison
`ExprNode.ne_(left, right)`	Inequality
`ExprNode.gt_(left, right)`	Greater-than
`ExprNode.ge_(left, right)`	Greater-or-equal
`ExprNode.lt_(left, right)`	Less-than
`ExprNode.le_(left, right)`	Less-or-equal
`ExprNode.and_(left, right)`	Logical AND
`ExprNode.or_(left, right)`	Logical OR
`ExprNode.not_(expr)`	Logical NOT
`ExprNode.contains_(field, pattern)`	String contains
`ExprNode.starts_with_(field, pattern)`	String prefix
`ExprNode.ends_with_(field, pattern)`	String suffix
`ExprNode.matches_(field, pattern)`	Regex match
`ExprNode.is_null_(expr)`	Null check
`ExprNode.is_not_null_(expr)`	Non-null check
`ExprNode.is_in_ints(field, values)`	Integer membership
`ExprNode.is_in_strings(field, values)`	String membership
`ExprNode.add_(left, right)`	Addition
`ExprNode.sub_(left, right)`	Subtraction
`ExprNode.mul_(left, right)`	Multiplication
`ExprNode.div_(left, right)`	Division

B.4 PatternNode Builder Methods

Method	Returns
`PatternNode.any_node()`	Match any vertex
`PatternNode.node(type_name)`	Match typed vertex
`PatternNode.binding(name, type_name)`	Named typed vertex
`PatternNode.chain(steps)`	Sequential pattern
`PatternNode.union(patterns)`	Union of patterns
`p.in_layer(layer)`	Restrict to layer
`p.then_any_edge()`	Extend with any edge
`p.then_edge(edge_type)`	Extend with typed edge
`p.then_any_node()`	Extend with any node
`p.then_node(type_name)`	Extend with typed node
`p.then_binding(name, type_name)`	Extend with named node
`p.then_variable_path(hop_spec)`	Variable-length extension
`p.binding_names()`	List all binding names

B.5 Fluent Query Builder Quick Reference

NodeQuery method	**Effect**
`q.nodes()`	Start a node query (`q = QueryNamespace(t)`)
`.of_type(type_name)`	Filter by node type
`.in_layer(layer)`	Filter by layer
`.of_kind(kind)`	Filter by kind (“device” or “endpoint”)
`.with_ids(ids)`	Filter by ID set
`.filter(expr)`	Apply an Expr predicate
`.isolated()`	Only isolated nodes
`.connected()`	Only connected nodes
`.sort(field, desc=False)`	Sort results
`.ids()`	Terminal: return list[int]
`.collect()`	Terminal: return full node structs
`.count()`	Terminal: return count
`.exists()`	Terminal: return bool
`.first()`	Terminal: return first match or None
`.one()`	Terminal: return exactly one match (raises if $\neq 1$)

LinkQuery method	Effect
`q.links()`	Start a link query
`.of_type(edge_type)`	Filter by edge type
`.in_layer(layer)`	Filter by layer
`.between(src_ids, dst_ids)`	Filter by endpoint nodes
`.include_internal()`	Include NTE-internal edges
`.filter(expr)`	Apply an Expr predicate
`.ids() / .collect() / .count()`	Terminal methods

B.6 Exception Reference

Exception class	Raised when
`NodeNotFoundError`	Node ID not in graph
`EdgeNotFoundError`	Edge not found
`NotAnEndpointError`	Expected endpoint, got device node
`NotANodeError`	Expected device node, got endpoint
`LayerMismatchError`	Incompatible layer combination
`LayerCycleError`	Layer dependency would form a cycle
`LengthMismatchError`	Parallel lists have different lengths
`LinkHasDependentsError`	Deleting a link that others depend on
`LinkNotFoundError`	Link ID not found
`InvariantViolationError`	Graph/store count mismatch
`DatastoreError`	Polars/DataFrame operation failed
`SchemaError`	Data doesn't match expected schema
`ArchiveError`	File I/O error during save/load
`SerializationError`	JSON/rkyv serialisation error
`TypeMismatchError`	Wrong value type for field
`BaseLayerError`	Invalid base layer operation
`DataFrameNotFoundError`	Named DataFrame not found in store

References

[1] Bluss and the petgraph Contributors. petgraph: Graph data structure library for Rust. https://crates.io/crates/petgraph, 2024.

[2] David Hewitt and the PyO3 Contributors. PyO3: Rust bindings for Python. In *Proceedings of the Python Language Summit*, 2023.

[3] Guodong Jin et al. Kùzu: An in-process property graph database management system. https://kuzudb.com, 2023.

[4] David Kolber. rkyv: Zero-copy deserialization in Rust. In *Proceedings of the Systems Programming Workshop*, 2023.

[5] Meta Platforms and the starlark-rust Contributors. starlark-rust: Rust implementation of the Starlark language. https://github.com/facebookexperimental/starlark-rust, 2024.

[6] Neo4j, Inc. Neo4j graph database. https://neo4j.com, 2024.

[7] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. In *ACM SIGMOD Record*, volume 42, pages 5–16, 2014.

[8] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 305–320. USENIX, 2014.

[9] Xuanwo Tang. OpenRaft: An advanced Raft implementation in Rust. In *Proceedings of the Rust OSDI Workshop*. USENIX, 2023.

[10] Ritchie Vink and the Polars Contributors. Polars: Blazingly fast dataframes in Rust and Python. https://pola.rs, 2024.