NTE User Manual

Graph-Relational Topology Engine

Simon Knight

Adelaide, Australia

March 2026

About this manual. This manual covers installation, core workflows, the NTE-QL query language, CLI reference, configuration, troubleshooting, and integration with AutoNetKit, NetCfg, NetVis, and the Workbench. NTE (Network Topology Engine) is a Rust-based hybrid graph-relational engine exposed to Python via PyO3 bindings. It maintains topology structure in a `petgraph` directed graph for $O(1)$-hop traversal and node/edge properties in Polars Arrow-backed DataFrames for SIMD-accelerated columnar scans, combining both in a single filter-first query planner. Suitable for network automation engineers who need to build, query, validate, and diff large network topologies programmatically.

Contents

1 Quick Start 1

2 Installation & Prerequisites 2
2.1 Building from Source . . . 2
2.2 Verifying the Installation . . . 2
2.3 Workspace Layout . . . 2

3 Core Workflows 3
3.1 Building a Topology . . . 3
3.2 Querying the Topology . . . 4
3.2.1 Using QuerySpec . . . 4
3.2.2 Using PatternNode for Graph Traversal . . . 4
3.3 Snapshots & What-If Analysis . . . 5
3.4 Policy Validation . . . 6

4 NTE-QL Query Language 6
4.1 Basic SELECT Queries . . . 6
4.1.1 Node scan with property filters . . . 6
4.1.2 All nodes of a type, sorted . . . 6
4.1.3 Count nodes per PoP . . . 6
4.1.4 Return only the first ten results . . . 7
4.2 Pattern Matching Syntax . . . 7
4.2.1 Direct adjacency (one hop) . . . 7
4.2.2 Bounded-hop reachability . . . 7
4.2.3 Shortest path between two routers . . . 7
4.2.4 Negation — find isolated routers . . . 7
4.3 EXPLAIN and EXPLAIN VISUAL . . . 7

5 CLI Reference 8
5.1 nte-cli repl — Interactive NTE-QL Shell . . . 8
5.2 nte-cli validate — Policy Validation . . . 8
5.3 nte-cli dashboard — Real-Time TUI Dashboard . . . 8

6 Configuration 9
6.1 Environment Variables . . . 9
6.2 Policy YAML Format . . . 10
6.3 .nte Archive Format . . . 10

7 Troubleshooting 11

8 Integration with Other Tools 12
8.1 AutoNetKit (ank_pydantic) — Python Frontend . . . 12
8.2 NetCfg (ank_netcfg) — Rust Configuration Compiler . . . 12
8.3 netsim — Topology Validation and Simulation . . . 13
8.4 NetVis — Topology Visualisation . . . 13
8.5 Workbench — Orchestration and Automation . . . 13

9 Further Reading 13

1 Quick Start

Get a topology loaded and queried in under ten minutes.

Step 1. **Install NTE.** From the repository root, build and install the Python extension using `maturin`:

```
uv run maturin develop
```

This compiles the Rust workspace and installs `ank_nte` into the active virtual environment.

Step 2. **Create a topology and add nodes.**

```
from ank_nte import Topology

topo = Topology()
topo.add_nodes([
    {"id": 1, "node_type": "Router", "hostname": "r1", "pop": "SYD"},
    {"id": 2, "node_type": "Router", "hostname": "r2", "pop": "SYD"},
    {"id": 3, "node_type": "Switch", "hostname": "sw1", "pop": "SYD"},
])
```

Step 3. **Add endpoints and wire them together.**

```
# Add endpoints (ports) owned by each router
topo.add_nodes([
    {"id": 10, "node_type": "Endpoint", "iface": "eth0"},
    {"id": 11, "node_type": "Endpoint", "iface": "eth0"},
])
# Intra edges: endpoint -> owning device
topo.add_intra_edges([(10, 1), (11, 2)])
# Inter edge: port-to-port link (bidirectional)
topo.add_inter_edges([(10, 11)])
```

Step 4. **Run a basic query.**

```
from ank_nte import QuerySpec

spec = QuerySpec(type_filter="Router", field_filters={"pop": "SYD"})
results = topo.query(spec)
for node in results:
    print(node["hostname"])
```

Expected output:

```
r1
r2
```

Step 5. **Take a snapshot before making changes.**

```
snap_id = topo.snapshot()
# ... make experimental changes ...
topo.restore_snapshot(snap_id)   # roll back if needed
```

2 Installation & Prerequisites

▷ **Prerequisites**

- **Python 3.8+** — tested on CPython 3.8, 3.10, 3.12.
- **Rust toolchain** — stable channel, 1.75 or later. Install via `rustup` (`https://rustup.rs`).
- **maturin** — the PyO3 build tool (`pip install maturin`).
- **uv** (recommended) — fast Python package manager and virtual environment tool (`pip install uv`).
- **Git LFS** (optional) — required only if fetching benchmark topology fixtures from the repository.
- **CUDA 12.x** (optional) — for GPU-accelerated graph kernels; see §6 for the `NTE_ENABLE_GPU` flag.

2.1 Building from Source

Clone the repository and build the Python extension:

```
git clone https://github.com/example/ank_nte.git
cd ank_nte
uv venv
uv run maturin develop
```

For a release build (significantly faster at runtime):

```
uv run maturin build --release
pip install target/wheels/ank_nte-1.4*.whl
```

Faster development builds

Add `–profile dev-fast` to skip LTO and enable incremental compilation. Rebuild times drop from ~30 s to ~4 s for incremental changes:

```
uv run maturin develop --profile dev-fast
```

2.2 Verifying the Installation

```
python -c "import ank_nte; print(ank_nte.__version__)"
```

Expected output: `1.4.0` (or the installed patch version).

GPU acceleration is optional

NTE does not require a GPU. GPU kernels are used automatically when `NTE_ENABLE_GPU=1` is set *and* a CUDA-capable device is detected. On topologies with fewer than ~5,000 nodes, the GPU overhead exceeds the benefit; leave GPU disabled for small-to-medium workloads.

2.3 Workspace Layout

NTE is an 18-crate Cargo workspace. The crates most relevant to day-to-day use are:

Crate	Purpose
`src/` (root)	PyO3 extension entry point
`nte-topology`	Topology struct, transactions, snapshots, diff
`nte-query`	Query executor, planner, profiler, pattern matcher
`nte-policy`	Starlark policy engine and DeltaNet validator
`nte-cli`	NTE-QL REPL, policy validator, TUI dashboard
`nte-datastore-polars`	Default Polars Arrow-backed property store
`nte-server`	Axum HTTP/WebSocket server and Raft consensus

3 Core Workflows

3.1 Building a Topology

The canonical workflow is: create a `Topology`, add nodes in batch, add endpoints, wire endpoints together with intra/inter edges, then query or persist.

Use `add_nodes` in batch rather than one node at a time; it amortises the Polars DataFrame rebuild cost.

Step 1. Instantiate a topology.

```
from ank_nte import Topology
topo = Topology()
```

Step 2. Add network devices in batch. Pass a list of dicts. Each dict must contain `id` (`int`) and `node_type` (`str`); all other fields become searchable properties.

```
routers = [
    {"id": i, "node_type": "Router",
     "hostname": f"r{i}", "pop": "SYD", "as_number": 64512 + i}
    for i in range(1, 9)
]
topo.add_nodes(routers)
```

Step 3. Add endpoints (interfaces) for each device. Endpoint IDs must not overlap with device IDs. A common convention is to use a high-range ID block (e.g. 10000+):

```
endpoints = [
    {"id": 10000 + i, "node_type": "Endpoint",
     "iface": "eth0", "owner_id": i}
    for i in range(1, 9)
]
topo.add_nodes(endpoints)
```

Step 4. Create intra edges (endpoint → owning device).

```
intra_pairs = [(10000 + i, i) for i in range(1, 9)]
topo.add_intra_edges(intra_pairs)
```

Step 5. Create inter edges (port-to-port links). Inter edges are bidirectional; NTE stores two directed edges internally and automatically creates a DeviceLink shortcut between the owning devices:

```
# Wire r1-eth0 <-> r2-eth0, r2-eth0 <-> r3-eth0, etc.
inter_pairs = [(10001, 10002), (10002, 10003), (10003, 10004)]
topo.add_inter_edges(inter_pairs)
```

Step 6. Verify the topology.

```
print(f"Nodes: {topo.node_count()}")
print(f"Edges: {topo.edge_count()}")
```

Layer assignment

Assign nodes to named layers for multi-plane topologies. Pass `layer="physical"` (or `"ip"`, `"bgp"`, etc.) in the node dict. NTE validates layer compatibility at edge-creation time and raises `LayerMismatchError` if nodes on incompatible layers are connected.

3.2 Querying the Topology

Use `QuerySpec` for flat property filters; use `PatternNode` / `QueryPlan` for multi-hop graph traversals.

NTE exposes two complementary query objects:

`QuerySpec` Flat predicate on node properties: type, layer, kind, field equality or range, and arbitrary expression filters. Best for "give me all nodes matching these properties".

`QueryPlan` / `PatternNode` Structural subgraph pattern AST. Best for "find all routers connected to a switch within two hops".

3.2.1 Using QuerySpec

Step 1. Build a `QuerySpec`.

```
from ank_nte import QuerySpec

spec = QuerySpec(
    type_filter="Router",
    field_filters={"pop": "SYD", "as_number": 64512},
)
```

Step 2. Execute the query.

```
results = topo.query(spec)
for node in results:
    print(node["hostname"], node["as_number"])
```

Step 3. Enable profiling to diagnose slow queries.

```
from ank_nte import QueryHints

hints = QueryHints(enable_profiling=True)
results, profile = topo.query(spec, hints=hints)
print(profile)  # per-step timing and candidate counts
```

3.2.2 Using PatternNode for Graph Traversal

Step 1. Describe the pattern. Find all routers directly connected (via Inter edge) to routers in the MEL PoP:

```
from ank_nte import PatternNode, QueryPlan, EdgeKind

pattern = PatternNode.chain([
    PatternNode.binding("src", "Router"),
    EdgeKind.Inter,
    PatternNode.node("Router", field_filters={"pop": "MEL"}),
])
plan = QueryPlan(root=pattern)
```

```
```

Step 2. Execute the pattern query.

```
matches = topo.match(plan)
for m in matches:
    print(m.bindings["src"]["hostname"])
```

Step 3. Check for truncation. By default, results are capped at 10,000 matches. Check `matches.truncated` if you expect large result sets and increase `QueryPlan.max_matches` accordingly.

3.3 Snapshots & What-If Analysis

NTE's snapshot mechanism captures the full topology state (graph structure plus all property DataFrames) at a point in time. Snapshots are in-memory by default; use the persistence API to write them to disk.

Snapshots use Polars CoW semantics — cloning is O(1) and adds negligible memory overhead for small topologies.

Step 1. Take a named snapshot before a risky change.

```
snap_id = topo.snapshot(label="before-redistribution")
print(f"Snapshot ID: {snap_id}")
```

Step 2. Make experimental changes.

```
topo.add_inter_edges([(10005, 10006)])
topo.update_node(5, {"as_number": 65001})
```

Step 3. Diff the current state against the snapshot.

```
delta = topo.diff(snap_id)
print("Added edges:", delta.added_edges)
print("Changed nodes:", delta.changed_nodes)
```

Step 4. Restore the snapshot if the changes are unwanted.

```
topo.restore_snapshot(snap_id)
```

Step 5. List available snapshots.

```
for s in topo.list_snapshots():
    print(s.id, s.label, s.created_at)
```

Step 6. Persist a snapshot to a `.nte` archive.

```
topo.save("topology-v1.nte", snapshot_id=snap_id)
```

Shadow topology for what-if analysis

For complex what-if scenarios, use `topo.shadow()` to create an independent copy that shares no memory with the primary topology. The shadow is fully writable and can be queried, diffed, and discarded without touching the original.

3.4 Policy Validation

NTE's policy engine evaluates Starlark-based rules against a topology and reports violations. Policies are loaded from YAML files that declare checks using a declarative constraint vocabulary.

Policies are evaluated against a read-only view of the topology; they cannot modify it.

Step 1. Write a policy file. See §6.2 for the full YAML schema. A minimal policy checking AS number uniqueness:

```
policy:
  name: as-uniqueness
  version: "1.0"
  checks:
    - id: unique-as-numbers
      description: "Each router must have a unique AS number"
      type: uniqueness
      node_type: Router
      field: as_number
```

Step 2. Load and run the policy engine.

```
from ank_nte import PolicyEngine

engine = PolicyEngine.load("policies/as-uniqueness.yaml")
report = engine.validate(topo)
```

Step 3. Inspect the report.

```
if report.has_violations():
    for v in report.violations:
        print(f"[{v.check_id}] {v.message} (node {v.node_id})")
else:
    print("All checks passed.")
```

Step 4. Run via the CLI (see §5):

```
nte-cli validate --policy policies/as-uniqueness.yaml topology.nte
```

4 NTE-QL Query Language

NTE-QL is a Cypher-inspired string query language compiled by a PEG grammar into `QueryPlan` / `QuerySpec` objects before execution. It is available in the interactive REPL (§5.1) and via `topo.query_str("...")` in Python.

4.1 Basic SELECT Queries

4.1.1 Node scan with property filters

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

4.1.2 All nodes of a type, sorted

```
MATCH (r:Router)
RETURN r.hostname, r.pop, r.as_number
ORDER BY r.hostname ASC
```

4.1.3 Count nodes per PoP

```
MATCH (r:Router)
RETURN r.pop, COUNT(r) AS router_count
ORDER BY router_count DESC
```

4.1.4 Return only the first ten results

```
MATCH (r:Router)
RETURN r.hostname
LIMIT 10
```

4.2 Pattern Matching Syntax

4.2.1 Direct adjacency (one hop)

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

4.2.2 Bounded-hop reachability

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

4.2.3 Shortest path between two routers

```
MATCH p = shortestPath((src:Router)-[*]->(dst:Router))
WHERE src.hostname = "r1" AND dst.hostname = "r8"
RETURN [n IN nodes(p) | n.hostname] AS path
```

4.2.4 Negation — find isolated routers

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

4.3 EXPLAIN and EXPLAIN VISUAL

Use `EXPLAIN` to inspect the compiled `FilterPlan` without executing the query:

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

Output lists each `FilterOp` in execution order with its estimated selectivity rank (F1–F5) and the column it targets.

`EXPLAIN VISUAL` renders an ASCII-art plan diagram in the REPL showing the filter pipeline and estimated row counts at each step. This is the fastest way to identify whether an expensive expression filter (F5) can be replaced with a more selective field-equality filter (F3).

Improve query performance

If `EXPLAIN` shows an F5 expression filter executing before a cheaper F3 equality filter, restructure the `WHERE` clause so the equality test appears first. NTE's planner sorts filters by rank, but provides a hint mechanism: `QuerySpec(force_filter_order=True)` applies filters in the order you specify.

5 CLI Reference

The `nte-cli` binary is built as part of the workspace and installed to `$CARGO_HOME/bin/nte-cli` (or the `.venv/bin/` directory when using uv).

5.1 nte-cli repl — Interactive NTE-QL Shell

```
nte-cli repl [--topology <file.nte>] [--history <history-file>]
```

Opens an interactive NTE-QL shell with readline history, tab completion on keywords and node types, and inline query profiling. All NTE-QL syntax described in §4 is available.

```
nte> MATCH (r:Router) WHERE r.pop = "SYD" RETURN r.hostname LIMIT 5
r1
r2
r3
r4
r5
(5 rows, 0.4 ms)

nte> EXPLAIN MATCH (r:Router) WHERE r.as_number = 64512 RETURN r.hostname
FilterPlan:
  [F2] type_filter = "Router"        -> ~1200 candidates
  [F3] field_eq(as_number, 64512)    -> ~8 candidates
Total estimated cost: LOW

nte> .quit
```

Type `.help` at the REPL prompt for a list of meta-commands.

5.2 nte-cli validate — Policy Validation

```
nte-cli validate --policy <policy.yaml> [--format text|json] <topology.nte>
```

Loads a topology from a `.nte` archive, evaluates all checks in the named policy file, and prints a structured violation report. Use `–format json` for machine-readable output in CI pipelines.

```
Checking policy: as-uniqueness (v1.0)
  [PASS] unique-as-numbers
  [FAIL] bgp-peer-reachability: 2 violation(s)
         - Node 5 (r5): no BGP peer reachable within 2 hops
         - Node 7 (r7): no BGP peer reachable within 2 hops
Exit code: 1
```

5.3 nte-cli dashboard — Real-Time TUI Dashboard

```
nte-cli dashboard --server <host:port> [--refresh <ms>]
```

Connects to a running `nte-server` instance and renders a real-time terminal UI showing:

- Topology node and edge counts with live mutation counters.
- Event ring buffer: last 50 `TopologyEvent` entries.
- Active write-lock contention and query latency percentiles (p50/p95/p99).
- Snapshot inventory with creation times and sizes.
- Server uptime and Raft cluster state (if distributed mode is active).

Python API Quick Reference

Command	Description
`Topology()`	Create a new empty topology
`topo.add_nodes(list)`	Batch-add nodes or endpoints
`topo.add_intra_edges(pairs)`	Add structural (ownership) edges
`topo.add_inter_edges(pairs)`	Add connectivity (link) edges
`topo.add_intranode_edges(p)`	Add intranode (chassis) edges
`topo.update_node(id, props)`	Update node properties
`topo.remove_node(id)`	Remove a node and its edges
`topo.query(spec)`	Execute a QuerySpec
`topo.match(plan)`	Execute a PatternNode plan
`topo.query_str(nteql)`	Execute an NTE-QL string
`topo.snapshot(label)`	Create an in-memory snapshot
`topo.restore_snapshot(id)`	Restore to a named snapshot
`topo.diff(snap_id)`	Diff current state vs snapshot
`topo.shadow()`	Create an independent copy
`topo.save(path)`	Persist to a .nte archive
`Topology.load(path)`	Load from a .nte archive
`PolicyEngine.load(yaml)`	Load a policy file
`engine.validate(topo)`	Run all checks; return report

QuerySpec Constructor

Command	Description
`type_filter=str`	Match nodes of this type
`layer_filter=str`	Restrict to this layer
`id_set=list[int]`	Restrict to specific node IDs
`field_filters=dict`	Equality filters on properties
`force_filter_order=bool`	Apply filters in given order
`QueryHints(enable_profiling)`	Attach profiling to a query
`QueryHints(cache_results)`	Cache result set (off by default)

6 Configuration

6.1 Environment Variables

NTE is configured at runtime via environment variables. Variables are read on process startup; changes require a restart.

Variable	Description	Default
NTE_NODE_ID	Unique integer ID for this node in a Raft cluster. Required in distributed mode.	—
NTE_RPC_ADDR	host:port this server listens on for Raft RPC.	127.0.0.1:7000
NTE_HTTP_ADDR	host:port for the Axum HTTP/WebSocket API.	127.0.0.1:8080
NTE_PEERS	Comma-separated id=host:port list of cluster peers.	—
NTE_DATA_DIR	Directory for WAL segments and snapshot archives.	./nte-data
NTE_ENABLE_GPU	Set to 1 to enable CUDA graph kernels.	0
NTE_PLAN_CACHE_CAP	LRU capacity for the compiled query plan cache.	256
NTE_EVENT_BUF_CAP	Ring buffer capacity for TopologyEvent entries.	4096
NTE_LOG	Log level: error, warn, info, debug, trace.	info

6.2 Policy YAML Format

Policy files are YAML documents with a `policy` top-level key. Each entry in `checks` describes one constraint. Supported check types:

```yaml
policy:
  name: production-checks
  version: "1.0"
  checks:
    # Uniqueness: each Router must have a unique hostname
    - id: unique-hostnames
      type: uniqueness
      node_type: Router
      field: hostname

    # Reachability: every Router must reach at least one peer
    - id: bgp-peer-reachability
      type: reachability
      node_type: Router
      edge_kind: Inter
      min_peers: 1
      max_hops: 2

    # Field required: all routers must declare a PoP
    - id: pop-required
      type: required_field
      node_type: Router
      field: pop

    # Value in set: AS numbers must be in the private range
    - id: private-asn
      type: field_in_set
      node_type: Router
      field: as_number
      allowed_range: [64512, 65534]

    # Custom Starlark rule
    - id: no-stub-routes
      type: starlark
      script: |
        def check(node, topo):
            if node.get("stub", False) and node["pop"] == "CORE":
                return "Core nodes must not be stubs"
            return None
```

6.3 .nte Archive Format

A `.nte` file is a ZIP archive containing:

Entry	Contents
`manifest.json`	Format version, NTE version, creation timestamp, node/edge counts
`graph.bin`	Serialised `petgraph` adjacency structure (bincode format)
`nodes.arrow`	Arrow IPC stream for the node property DataFrame
`endpoints.arrow`	Arrow IPC stream for the endpoint property DataFrame
`edges.arrow`	Arrow IPC stream for the edge property DataFrame
`snapshots/`*`id`*`.bin`	One bincode-serialised snapshot per stored snapshot
`policies/`	Optional: copies of policy YAML files associated with this topology

Inspecting .nte archives

`.nte` files are standard ZIP archives. Use `unzip -l topology.nte` to list contents or `python -c "import pyarrow.ipc as ipc; print(ipc.open_stream(open('nodes.arrow','rb')).read_all())"` to inspect the property table directly.

7 Troubleshooting

Problem 1: Topology state appears inconsistent or partially mutated after a process crash

Cause: NTE does not implement a write-ahead log (WAL). A crash during a dual-write mutation (between updating the `petgraph` structure and updating the Polars DataFrames) leaves the in-memory state in an undefined condition. The on-disk archive, if not yet re-saved, reflects the pre-crash state.

Solution: Always call `topo.save()` after a series of mutations to persist a consistent snapshot. For crash-consistent operation, deploy NTE in distributed mode (§6) where Raft provides durability. On restart, load the last known-good `.nte` archive with `Topology.load(path)`.

Problem 2: Python callback registered via `topo.on_event()` hangs or deadlocks

Cause: The event callback is invoked while NTE holds the topology write lock. If the callback re-enters NTE (e.g. by calling `topo.query()` from inside the handler), a deadlock occurs because the query tries to acquire the read lock that is already held by the mutation path.

Solution: Never call topology methods from within an event callback. Instead, enqueue the event payload to a Python `queue.Queue` and process it from a separate thread. NTE's GIL release guarantee ensures the producer side is non-blocking.

Problem 3: Queries slow down significantly after many bulk insertions

Cause: NTE's Compressed Sparse Row (CSR) cache is invalidated on every structural mutation. After a large batch of `add_inter_edges` calls, the first traversal-heavy query triggers an expensive CSR rebuild that can take several hundred milliseconds for topologies with 50,000+ edges.

Solution: Rebuild the CSR cache explicitly *after* all batch mutations and *before* the first query by calling `topo.rebuild_csr()`. This amortises the rebuild cost into a single operation. Alternatively, wrap bulk insertions in a transaction (`topo.begin_transaction()`) — NTE defers CSR rebuild until the transaction commits.

Problem 4: GPU acceleration enabled but queries are slower than without GPU

Cause: For topologies with fewer than ~5,000 nodes, the overhead of marshalling data to GPU memory, executing CUDA kernels, and transferring results back exceeds the computational saving. This is expected behaviour, not a bug.

Solution: Set `NTE_ENABLE_GPU=0` (or leave it unset) for topologies below the 5,000-node threshold. Use `EXPLAIN VISUAL` in the REPL to inspect whether the query planner is routing work through the GPU path, indicated by `[GPU]` annotations in the plan output.

Problem 5: DataFrame operations raise `SchemaError: column 'X' not found`

Cause: Property columns in the Polars store are created lazily when a field name first appears. If you query a field that has never been set on any node, the column does not exist and Polars raises a schema error during filter execution.

Solution: Before filtering on a field, verify it exists with `topo.has_property_column("X")`. Alternatively, use `QuerySpec(field_filters={"X": value})` — NTE's query layer checks for column existence before constructing the Polars pipeline and returns an empty result set (rather than an error) when the column is absent.

8 Integration with Other Tools

NTE is the shared backend engine for the network automation ecosystem. The following tools use NTE directly or interoperate with it.

8.1 AutoNetKit (ank_pydantic) — Python Frontend

`ank_pydantic` is a Python library providing 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. It is the recommended entry point for interactive exploration, Jupyter notebooks, and Python-centric automation pipelines.

```
from ank_pydantic import Network, Router, Switch
import ank_nte

net = Network()
net.add(Router(id=1, hostname="r1", pop="SYD", as_number=64512))
net.add(Router(id=2, hostname="r2", pop="SYD", as_number=64513))
# AutoNetKit populates the NTE topology via the Python API
topo = net.to_nte_topology()
```

See also: *AutoNetKit User Manual* (ANK-UM-001) — topology modelling, Pydantic schema reference, and export formats.

8.2 NetCfg (ank_netcfg) — Rust Configuration Compiler

`ank_netcfg` is a pure-Rust deterministic configuration compiler. It consumes the `nte-topology` crate directly (no Python layer), reads declarative YAML blueprints, maps them through a DeviceIR intermediate representation, and renders vendor-neutral device configurations. Use NetCfg when a Python runtime is undesirable or when CI/CD pipeline throughput is critical.

```
# Export NTE topology to JSON, compile with NetCfg
nte-cli export topology.nte --format json > topology.json
netcfg compile topology.json --platform junos --output configs/
```

See also: *NetCfg User Manual* (NETCFG-UM-001) — blueprint YAML schema, platform targets, and DeviceIR reference.

8.3 netsim — Topology Validation and Simulation

`netsim` takes an NTE topology and runs dataplane simulations: reachability checks, MTU consistency, and routing convergence. It communicates with NTE via the HTTP API exposed by `nte-server`.

```
# Start the NTE server, then run netsim validation
nte-server --topology topology.nte &
netsim validate --nte http://localhost:8080 --checks reachability,mtu
```

See also: *netsim User Manual* (NETSIM-UM-001) — simulation scenarios, check definitions, and result interpretation.

8.4 NetVis — Topology Visualisation

`netvis` subscribes to the NTE WebSocket event stream and renders a live, force-directed graph of the topology in a browser. Node colour and size are driven by NTE properties; double-clicking a node opens a property inspector panel backed by a live `QuerySpec`.

```
# Serve the topology and open NetVis
nte-server --topology topology.nte &
netvis open --nte ws://localhost:8080/events
```

See also: *NetVis User Manual* (NETVIS-UM-001) — layout algorithms, property-driven styling, and export to SVG/PNG.

8.5 Workbench — Orchestration and Automation

The Workbench provides a graphical orchestration environment that chains NTE topology operations, NetCfg compilation runs, and netsim validation checks into reproducible pipelines. It uses the NTE Python API directly and persists pipeline state in `.nte` archives.

```
# workbench.yaml
pipeline:
  - step: load_topology
    source: topology.nte
  - step: apply_policy
    policy: policies/production.yaml
    on_fail: abort
  - step: compile_configs
    tool: netcfg
    platform: eos
  - step: simulate
    tool: netsim
    checks: [reachability, bgp_convergence]
```

```
workbench run workbench.yaml
```

See also: *Workbench User Manual* (WB-UM-001) — pipeline YAML schema, step library, and CI/CD integration.

9 Further Reading

- *NTE Technical Reference* (NTE-TR-001) — full architecture, data model, storage internals (dual-write protocol, Polars and petgraph integration), query system design, ACID transaction semantics, persistence formats, GPU acceleration, graph algorithm plugins, Starlark policy engine, reactive events, production diagnostics, and Raft-based distributed deployment. **Read this document to understand why NTE behaves as it does and how to extend it.**

- *AutoNetKit Technical Report* (ANK-TR-001) — Pydantic schema design, model validation pipeline, and integration with NTE's type system.
- *NetCfg Technical Report* (NETCFG-TR-001) — DeviceIR intermediate representation, Jinja2/Starlark template pipeline, and deterministic configuration generation.
- *Ecosystem Technical Report* (NETAUTO-TR-001) — cross-tool integration architecture, shared data formats, and design decisions for the overall network automation stack.