AutoNetKit

A Type-Safe Framework for Network Topology Modelling

Simon Knight

Independent Researcher, Adelaide, Australia

March 2026 • Version 0.1

Last updated: March 12, 2026

Abstract. AutoNetKit (current package name `ank_pydantic`) is a Python framework for type-safe network topology modelling built on Pydantic v2 and backed by NTE, a Rust graph engine documented in companion report NTE-TR-001. Its central design principle is that network models should be *statically typed end-to-end*: user data lives in generic Pydantic models parameterised over `BaseModel` subclasses, providing IDE autocompletion, runtime validation, and serialisation for free. Topologies are manipulated through a manager-first façade that wires eighteen domain-specific managers over a shared store, while a multi-layer architecture enables independent protocol views of the same physical network. A Polars-style fluent query API and a composable rule engine (the "network linter") round out the framework, enabling both exploratory analysis and declarative design validation. This report covers the data model, architecture, layer system, query API, rule engine, configuration generation pipeline, and an end-to-end tutorial.

Contents

1 Introduction

AutoNetKit is a Python framework for building, querying, validating, and compiling network topologies. It sits between the network engineer who thinks in terms of routers, interfaces, and protocols, and the NTE Rust engine that provides high-performance graph storage and columnar queries.

1.1 What AutoNetKit Is

At its core, AutoNetKit provides:

1. A **typed data model** where every node, edge, and endpoint carries a generic `data: T` field backed by a Pydantic `BaseModel`.
2. A **manager-first API** where operations like `topology.nodes.add()`, `topology.layers.get()`, and `topology.query.nodes()` are namespaced under domain-specific managers.
3. A **multi-layer architecture** where the same physical network can be viewed through independent protocol lenses (OSPF, BGP, ISIS, etc.).
4. A **fluent query API** inspired by Polars, with expression filtering, pattern matching, and Rust-accelerated execution.
5. A **rule engine** that validates topologies against composable, declarative design rules—the "network linter."
6. A **configuration generation pipeline** that compiles validated topologies into vendor-specific device configurations.

1.2 Design Principles

Five principles guide every design decision:

Type safety end-to-end. Generic Pydantic models ensure that user data is validated at the boundary and carries type information through to queries, rules, and configuration generation.

More layers are better than fewer. Each protocol gets its own layer rather than overloading a single graph. Layers are cheap—they are just filtered views over the shared store.

Managers own behaviour. The `Topology` class is a thin façade; all domain logic lives in managers that can be tested and extended independently.

Validate before you generate. The rule engine runs before configuration generation, catching design errors (single points of failure, disconnected components, naming violations) before they become operational incidents.

Rust for speed, Python for ergonomics. The NTE engine handles graph storage, traversal, and columnar queries in Rust; AutoNetKit provides the Pythonic API, Pydantic models, and domain logic.

1.3 Audience

This report targets three overlapping audiences:

- **Network engineers** building topology models for design validation and configuration generation.
- **SRE / NetDevOps engineers** integrating AutoNetKit into CI/CD pipelines for automated network testing.
- **Network architects** designing multi-layer topologies and custom design rules for organisational standards.

1.4 Document Map

Companion Document

This report covers the Python framework. For the Rust engine internals (graph storage, columnar queries, PyO3 bindings), see **NTE-TR-001: Network Topology Engine — Technical Reference**.

The package is currently published as `ank_pydantic`. A rename to `autonetkit` is planned before public launch.

Section 2 introduces network modelling concepts. Section 3 details the generic Pydantic data model. Section 4 describes the manager-first architecture. Section 5 covers the layer system. Section 6 presents the query API. Section 7 explains the rule engine. Section 8 outlines configuration generation. Section 9 walks through an end-to-end example. Sections 10 and 11 discuss limitations and future work. Appendices provide quick-reference tables.

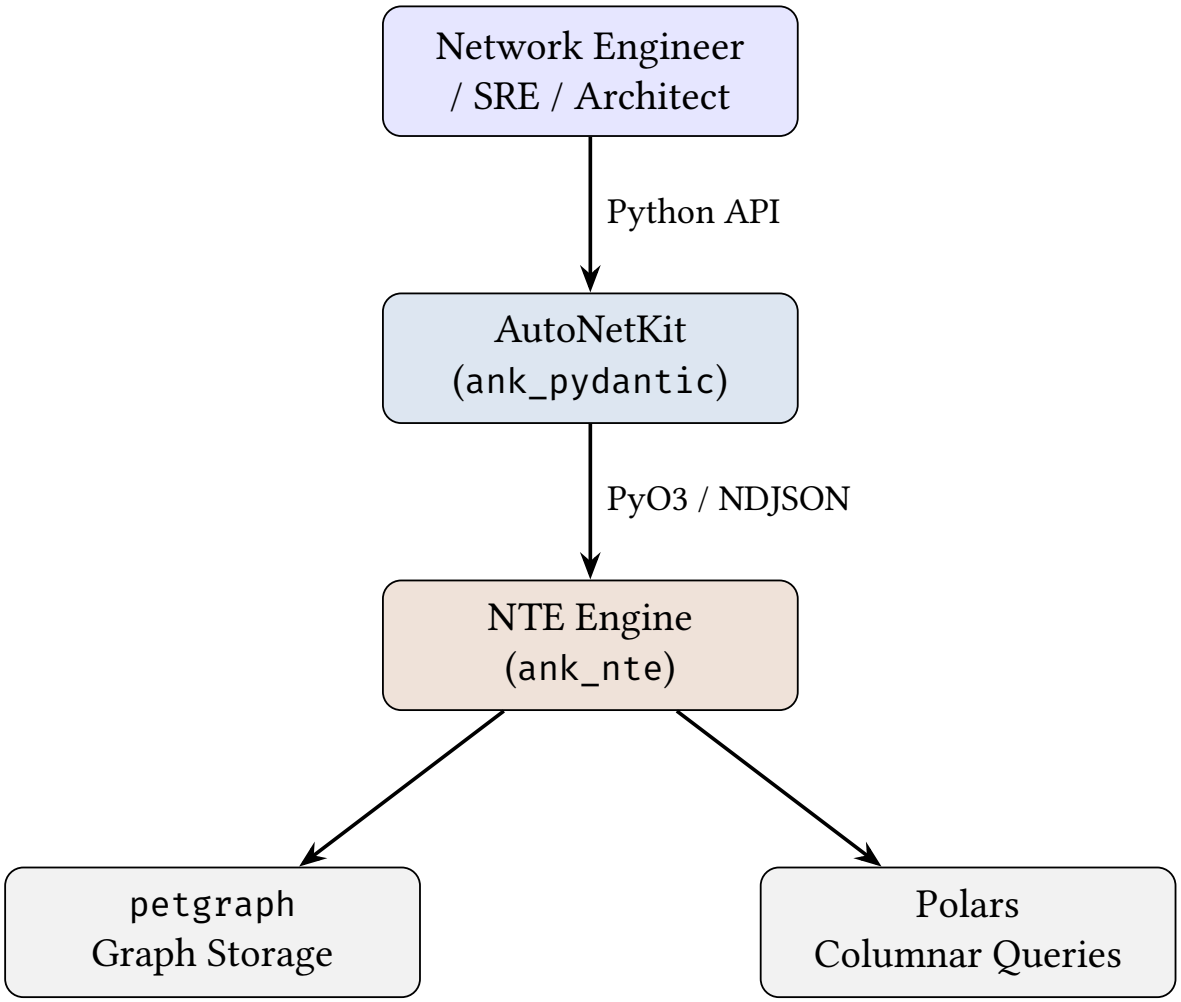

Figure 1. System context: AutoNetKit sits between the user and the NTE Rust engine.

2 Background — Network Modelling Concepts

This section provides a domain primer for software engineers who may not have a networking background. Experienced network engineers may skim or skip ahead to Section 3.

2.1 Nodes, Endpoints, and Edges

A network topology is a graph. The vertices represent *devices* (routers, switches, hosts) and the edges represent *connections* between them. In AutoNetKit, we distinguish three kinds of graph element:

Nodes represent devices or logical elements: routers, switches, VLANs, VRFs. They are first-class graph citizens that can be queried, filtered, and traversed.

Endpoints represent connection points on devices: physical interfaces (Ethernet0/0), loopbacks, subinterfaces. An endpoint belongs to exactly one node.

Edges represent relationships between elements. AutoNetKit defines three kinds of edge, each with strict connection rules (Section 3.4).

2.2 Layers

A single physical network supports multiple protocol views. An OSPF process sees only the routers within its area; a BGP session sees only the routers within its autonomous system. AutoNetKit models each view as a separate *layer*—a filtered projection of the topology graph.

Layers are *derived*, not duplicated. When a layer is created, nodes are *cloned* from a parent layer (typically the physical layer), and edges are selectively copied based on filtering rules (e.g., "keep edges where the source and destination share the same ASN" for an OSPF layer). Each derived node maintains a pointer back to its plan-layer counterpart, enabling cross-layer traversal.

2.3 Ownership

Not all relationships are peer connections. A router *owns* its interfaces; a VRF *belongs to* a router. AutoNetKit models these with *ownership edges*, which are structurally distinct from inter-device connections and intra-device relationships.

Running Example: 7-Router ISP Network

Throughout this report we use a small ISP network as a running example: two core routers (`Core1`, `Core2`) and four edge routers (`Edge1`–`Edge4`), plus a route server (`RS1`). Cores form a fully-connected backbone; each edge router peers with both cores. This topology is small enough to diagram yet rich enough to illustrate multi-layer derivation, queries, and design rules.

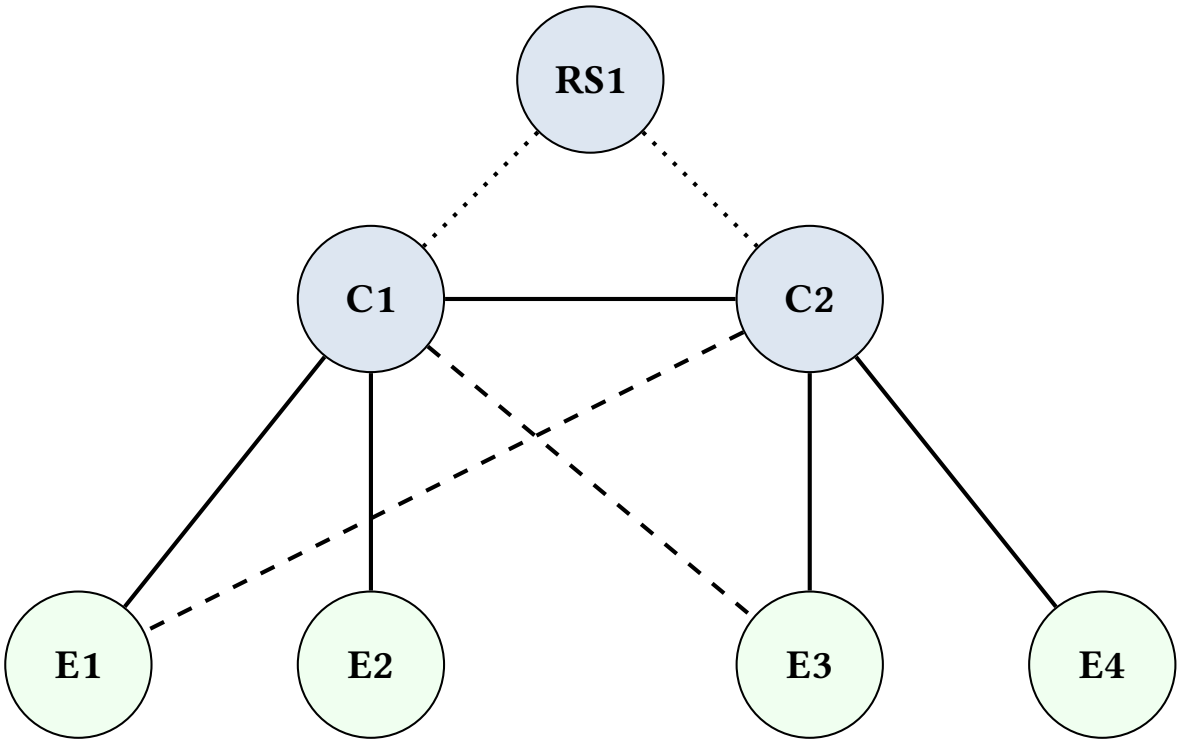

Figure 2. Running example: 7-router ISP network. Solid lines are primary links; dashed lines are backup paths; dotted lines connect the route server.

3 Data Model — Generic Pydantic Models

The data model is the foundation of AutoNetKit. Every node, edge, and endpoint carries a typed `data` field that holds user-defined attributes. This section explains the design decisions behind this pattern and walks through the class hierarchy.

3.1 Problem: Type Safety vs. Flexibility

Network modelling tools face a tension. On one hand, different users model different attributes: a datacenter engineer cares about spine/leaf roles and fabric bandwidth; an ISP engineer cares about ASN, OSPF area, and BGP communities. The data model must be *flexible* enough to accommodate any domain.

On the other hand, a stringly-typed dictionary of attributes provides no IDE support, no validation, and no documentation. Typos in field names (`bandwith` vs. `bandwidth`) silently corrupt data. The data model must be *type-safe*.

3.2 Decision: Generic Data Field

AutoNetKit resolves this tension with a generic `data: T` field, where `T` is a user-defined Pydantic `BaseModel`:

Listing 1. The generic data pattern.

```python
from pydantic import BaseModel
from ank_pydantic.core.models.base import BaseTopologyNode

class RouterData(BaseModel):
    label: str
    vendor: str = "Generic"
    asn: int = 0

class Router(BaseTopologyNode[RouterData]):
    pass

# Full type safety: IDE knows router.data.vendor is str
router = Router(layer="physical", data=RouterData(label="R1", asn=65001))
print(router.data.vendor)  # "Generic"
```

Consequences:

- Full IDE autocompletion on `data.*` fields.
- Runtime validation by Pydantic: invalid types raise `ValidationError`.
- Free serialisation (`model_dump()`, `model_dump_json()`).
- System fields (`id`, `layer`) are separated from user data.
- A `FlexibleData` default (with `extra="allow"`) supports untyped prototyping.

3.3 Node Hierarchy

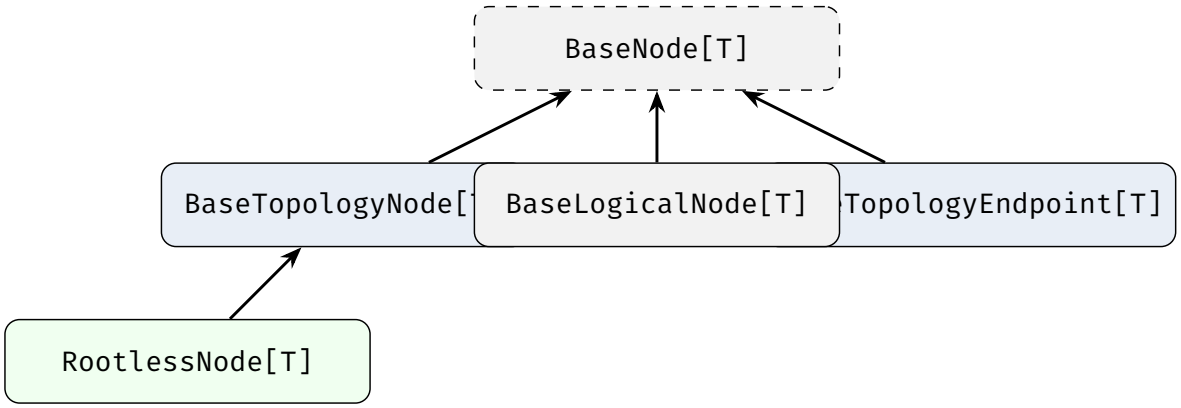

Figure 3. Node class hierarchy. All classes are generic over a data model `T`.

The node hierarchy (Figure 3) provides four concrete base classes:

`BaseTopologyNode[T]` — First-class graph elements: routers, switches, VLANs. Supports layer traversal (`in_layer()`, `in_plan()`), neighbour queries, and internal namespaces.

`BaseTopologyEndpoint[T]` — Connection points on devices: interfaces, ports. Connected via `connect_to()` which creates connection links.

`BaseLogicalNode[T]` — Internal structures within a device: VRFs, routing instances, line cards. Not first-class graph citizens; accessed via their parent node.

`RootlessNode[T]` — Derived-layer-only nodes with no plan-layer counterpart: collision domains, broadcast domains. Created by design functions.

All nodes share three system fields managed by the topology:

Field	Type	Description
`id`	`Optional[int]`	Assigned by topology on `add()`
`layer`	`str`	Layer this node belongs to
`data`	`T`	User-defined data model

3.4 Edge Hierarchy

Edges follow the same generic pattern but enforce strict connection rules:

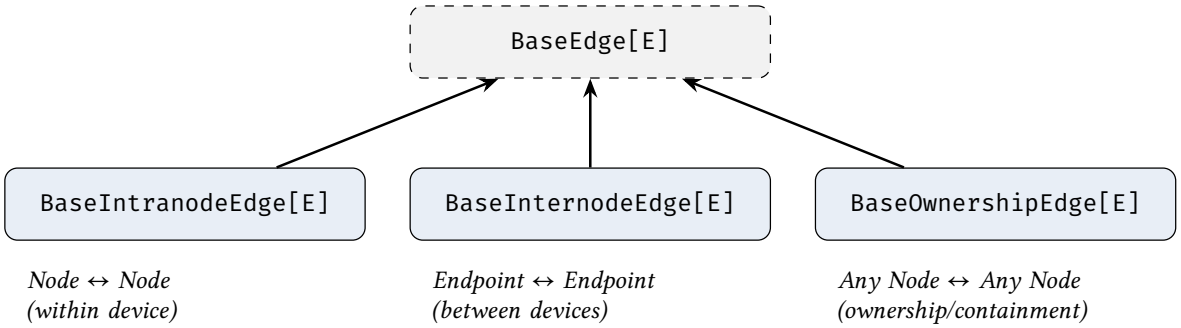

Figure 4. Edge class hierarchy with connection rules.

`BaseIntranodeEdge[E]` connects `BaseTopologyNode` to `BaseTopologyNode` within the same device (e.g., VLAN membership).

`BaseInternodeEdge[E]` connects `BaseTopologyEndpoint` to `BaseTopologyEndpoint` across devices (e.g., physical links).

`BaseOwnershipEdge[E]` connects any `BaseNode` to any `BaseNode` for containment (e.g., "router owns interface").

Edge system fields include `id`, `layer`, `type` (a string discriminator), `src_id`, `dst_id`, and `data: E`.

3.5 Worked Example: Defining a Datacenter Model

Datacenter Model Definition

The `batteries_included` module ships a leaf-spine datacenter topology. Here is how the models are defined:

```python
from pydantic import BaseModel
from ank_pydantic.core.models.base import BaseTopologyNode, BaseTopologyEndpoint
from ank_pydantic.core.models.edge import BaseInternodeEdge

class SpineRouterData(BaseModel):
    label: str
    role: str = "spine"
    platform: str = "arista_eos"

class SpineRouter(BaseTopologyNode[SpineRouterData]):
    pass

class LeafSwitchData(BaseModel):
    label: str
    role: str = "leaf"
    platform: str = "arista_eos"

class LeafSwitch(BaseTopologyNode[LeafSwitchData]):
    pass

class FabricEndpointData(BaseModel):
    label: str
    speed: str | None = None

class FabricEndpoint(BaseTopologyEndpoint[FabricEndpointData]):
    pass

class FabricLinkData(BaseModel):
    bandwidth: int | None = None

class FabricLink(BaseInternodeEdge[FabricLinkData]):
    type: str = "fabriclink"
```

Each model is fully typed: `SpineRouter.data` is always `SpineRouterData`, never an untyped dictionary.

4 Architecture — Manager-First Façade

4.1 Problem: God-Object vs. Scattered API

A topology object that exposes every operation directly becomes a god-object with hundreds of methods. Conversely, splitting operations across unrelated modules forces users to discover and import many classes. AutoNetKit needs a middle ground.

4.2 Decision: Topology as Thin Façade

The `Topology` class is a thin wiring container. Its `__init__` instantiates approximately eighteen managers, each responsible for a domain of functionality, and wires them together over a shared `TopologyStore`:

Listing 2. Topology manager wiring (simplified).

```python
class Topology:
    def __init__(self, layer="whiteboard", *, auto_freeze=True):
        self._store = TopologyStore()

        # Core managers
        self.nodes: NodeManager = NodeManager(self._store)
        self.edges: EdgeManager = EdgeManager(self._store)
        self.links: LinkManager = LinkManager(self._store)
        self.layers: LayerManager = LayerManager(self._store)
        self.ancestors: AncestorManager = AncestorManager(self._store)
        self.sync: SyncManager = SyncManager(self._store)
        self.io: IOManager = IOManager(self._store)

        # Analysis and design
        self.analysis: AnalysisManager = AnalysisManager(self._store)
        self.design: DesignManager = DesignManager(self._store)
        self.workflow: WorkflowManager = WorkflowManager(self._store)

        # Query API
        self.query: QueryManager = QueryManager(self)

        # Wire inter-manager dependencies
        self.layers.set_managers(self.nodes, self.ancestors)
        self.nodes.set_managers(self.endpoint_mgr, self.links)
        # ... (further wiring omitted)
```

Consequences:

- Users discover functionality through dot-completion: `topology.` → `nodes`, `edges`, `layers`, `query`, etc.
- Each manager can be unit-tested with a mock store.
- New domains (e.g., `LayoutManager`) are added by creating a manager class and wiring it into `Topology.__init__`.
- The shared `TopologyStore` holds the NTE engine handle and the identity map, avoiding global state.

> **NTE Rust Engine**
>
> The `TopologyStore` wraps the NTE Rust engine, which provides the actual graph storage (`petgraph::StableDiGraph`) and columnar property tables (Polars DataFrames). Communication between Python and Rust uses NDJSON serialisation for bulk operations and direct PyO3 calls for single-element access. See NTE-TR-001 for engine internals.

4.3 Data Flow: Adding a Node

When the user calls `topology.nodes.add(router)`, the following occurs:

1. **Pydantic validation**: the `Router` model validates all fields (including `data`).
2. **NodeManager** serialises the node to NDJSON and calls `store.nte.add_node()`.
3. **NTE engine** inserts a vertex into the `petgraph` graph and a row into the Polars DataFrame.
4. **ID assignment**: NTE returns the assigned node ID, which is set on the Python model (`router.id = 42`).
5. **Identity map**: the node is registered in the identity map so that future mutations to `router.data.vendor` write through to Rust storage.

5 Layer Architecture

Layers are AutoNetKit's mechanism for maintaining multiple independent views of the same network. This section covers the motivation, mechanics, and usage patterns.

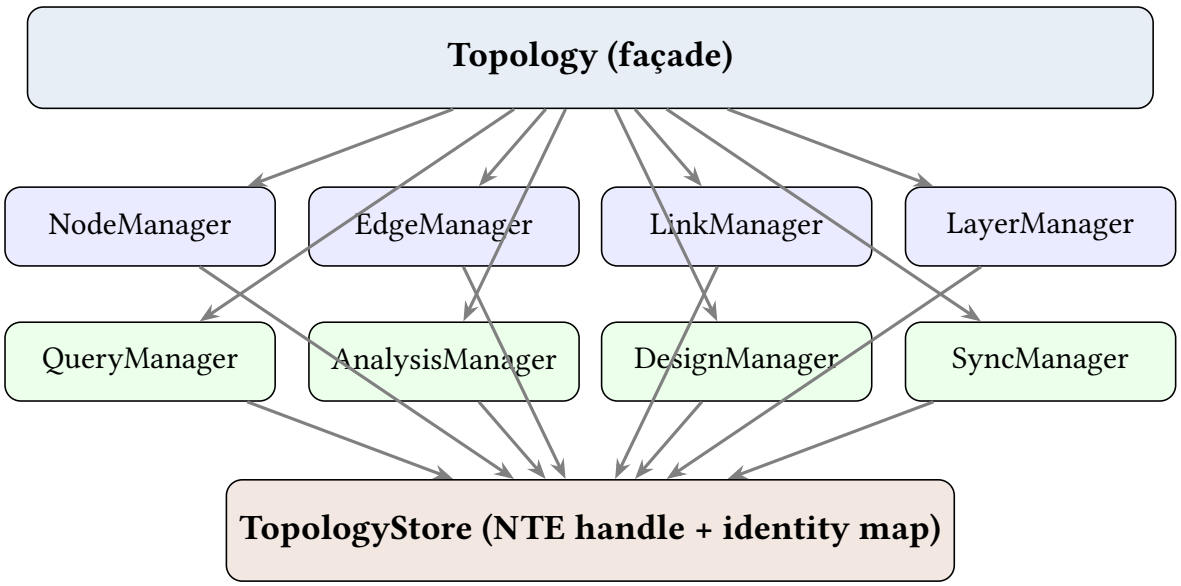

Figure 5. Container diagram: `Topology` wires managers over a shared store.

5.1 Motivation: More Layers > Fewer

A naïve approach stores all protocol information on a single node: a router object carries OSPF area, BGP ASN, ISIS NET, and MPLS label range as top-level fields. This conflates concerns and makes it impossible to answer questions like "which routers participate in OSPF area 0?" without filtering by a magic field.

AutoNetKit's principle is *more layers are better than fewer*. Each protocol view is a separate layer with its own nodes and edges:

- The **whiteboard** layer is the initial design input.
- The **plan** layer is the formalised device structure.
- The **physical** layer represents the physical topology.
- Protocol layers (**ospf**, **bgp**, **isis**, etc.) are derived from the physical layer using filtering rules.

5.2 Layer Mechanics

Three classes implement the layer system:

`LayerManager` (`topology.layers`) — the central orchestrator. Manages layer creation, querying, cloning, freezing, and configuration.

`LayerHandle` — a fluent accessor returned by `topology.layers.get("ospf")`. Provides type-safe querying and configuration for a specific layer.

`LayerBuilder` — handles declarative layer configuration, validating and building layers from configuration dictionaries.

5.2.1 Cloning Nodes to Derive a Layer

Layer derivation works by *cloning* nodes from a parent layer:

Listing 3. Cloning nodes to a derived layer.

```
# Get source nodes
source_nodes = topology.layers.get("physical").nodes().models()

# Clone to new OSPF layer
result = topology.layers.clone_to_layer(
    nodes=source_nodes,
    target_layer="ospf",
)

# result.source_to_target maps physical IDs to OSPF IDs
# result.target_ids lists all created OSPF node IDs
```

Each cloned node gets a *ParentNode* edge back to its source, and a *PlanOrigin* edge back to the plan layer. This enables cross-layer traversal: given an OSPF router, call `router.in_plan(topology)` to get its plan-layer counterpart, or `router.in_layer("bgp", topology)` to find it in the BGP layer.

5.2.2 Declarative Layer Building

For common patterns, layers can be built declaratively:

Listing 4. Declarative layer configuration.

```
LAYERS = {
    "physical": {
        "parent": "whiteboard",
    },
    "ospf": {
        "parent": "physical",
        "keep_edges_where_same": "asn",  # OSPF within same AS
    },
    "ebgp": {
        "parent": "physical",
        "keep_edges_where_different": "asn",  # BGP across AS boundaries
    },
    "ibgp": {
        "parent": "physical",
        "copy_edges": False,
        "group_by": "asn",
        "create": "full_mesh",  # iBGP full mesh within each AS
    },
}

errors = topology.layers.validate(LAYERS)
if not errors:
    topology.layers.build(LAYERS)
```

The builder resolves dependencies (e.g., `ospf` depends on `physical`), validates the configuration, and builds layers in topological order.

5.3 Layer Handles

A `LayerHandle` is a fluent interface for querying and configuring a specific layer:

```
layer = topology.layers.get("physical")

# Query nodes in this layer
routers = layer.nodes(Router).where(asn=65001).models()

# Query links
links = layer.links().collect()

# Set default types for this layer
layer.set_defaults(node=Router, endpoint=Interface, link=PhysicalLink)

# Create simplified device-to-device view
simplified = layer.to_simplified(promote={"bandwidth": "max"})
```

5.4 Freeze Semantics

Layers can be *frozen* to prevent accidental modification:

```
topology.layers.freeze("whiteboard")
assert topology.layers.frozen("whiteboard") == True
# Any mutation to a frozen layer raises an error
```

This is particularly important for the whiteboard and plan layers, which serve as the "source of truth" for derived layers.

5.5 Design Functions

A `DesignFunction` encapsulates reusable layer configurations:

```
from ank_pydantic.core.designs.base import DesignFunction

class SmallInternetDesign(DesignFunction):
    name = "small_internet"
    description = "Small internet topology with BGP and OSPF"
    version = "1.0"

    layers = {
        "physical": {"parent": "whiteboard"},
        "ospf": {
            "parent": "physical",
```

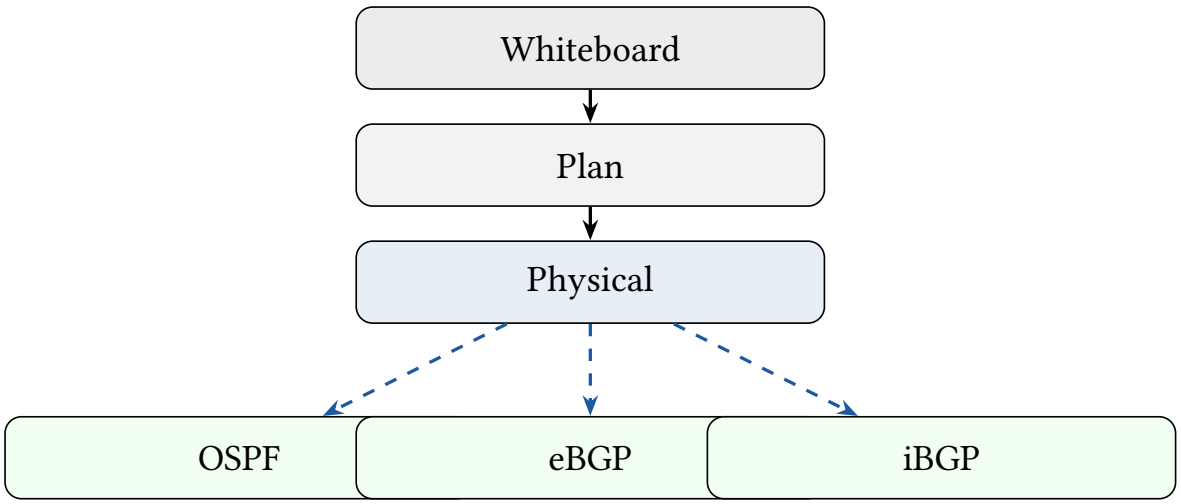

Figure 6. Multi-layer stack: protocol layers derived from the physical layer. Solid arrows show parent relationships; dashed arrows show derivation.

```
            "keep_edges_where_same": "asn",
        },
        "ebgp": {
            "parent": "physical",
            "keep_edges_where_different": "asn",
        },
    }

# Apply to any topology
design = SmallInternetDesign()
design.apply(topology)
```

6 Query API

AutoNetKit provides a Polars-style fluent query API for exploring and filtering topology data. Queries are lazy builders: filtering methods return new query objects, and terminal methods (`models()`, `ids()`, `count()`) execute against the Rust engine.

6.1 Entry Points

All queries start from `topology.query`:

```
# Node queries
routers = topology.query.nodes().of_type(Router).models()

# Link queries
links = topology.query.links().in_layer("physical").collect()

# Edge queries
edges = topology.query.edges().of_type(FabricLink).dataframe()

# Path queries
path = topology.query.paths().from_node(r1.id).to_node(r2.id).shortest()
```

6.2 NodeQuery Chaining

`NodeQuery` is generic over the model type, enabling type-safe results:

```
# Type narrowing: NodeQuery[Router] -> List[Router]
cisco_routers = (
    topology.query.nodes()
    .of_type(Router)           # NodeQuery[Router]
    .in_layer("physical")       # Still NodeQuery[Router]
    .where(vendor="Cisco")      # Still NodeQuery[Router]
    .models()                   # List[Router]
)
```

Key filtering methods:

Method	Description
`of_type(Model)`	Filter by Pydantic model type (includes subtypes)
`in_layer(name)`	Filter by layer
`where(**fields)`	Equality filter on data fields (Rust-optimised)
`where(predicate)`	Python-side predicate filter
`filter(*exprs)`	Expression-based filtering
`to_plan()`	Navigate to plan-layer counterparts
`to_layer(name)`	Navigate to another layer

Terminal methods:

Method	Returns
`models()`	`List[T]` — hydrated Pydantic models
`ids()`	`List[int]` — node IDs only
`count()`	`int` — match count (no materialisation)
`first()`	`Optional[T]` — first match or None
`one()`	`T` — exactly one match (raises otherwise)
`exists()`	`bool` — any match?
`dataframe()`	`pl.DataFrame` — Polars DataFrame

6.3 Expression Filtering

The `q` module provides Polars-style expression building:

Listing 5. Expression filtering with the `q` module.

```python
from ank_pydantic import q

# Field comparisons
high_bw = topology.query.nodes().of_type(Router).filter(
    q.field('bandwidth') > 1000
).models()

# Logical composition
cisco_fast = topology.query.nodes().of_type(Router).filter(
    (q.field('vendor') == 'Cisco') & (q.field('bandwidth') > 1000)
).models()

# String operations
core_routers = topology.query.nodes().of_type(Router).filter(
    q.field('label').contains('Core')
).models()

# Endpoint subqueries
routers_with_fast_ports = topology.query.nodes().of_type(Router).filter(
    q.endpoints().any(q.field('speed') > 10000)
).models()
```

Expressions support the full set of comparison operators (`==`, `!=`, `>`, `>=`, `<`, `<=`), logical operators (`&`, `|`, `~`), string methods (`contains`, `startswith`, `matches`), and membership tests (`is_in`, `is_null`).

The `Model.fields` accessor provides an alternative syntax:

```python
# Using Model.fields (equivalent to q.field())
high_bw = topology.query.nodes().of_type(Router).filter(
    Router.fields.bandwidth > 1000
).models()
```

6.4 Pattern Matching

For graph-structural queries, AutoNetKit supports pattern matching with the » operator:

```python
from ank_pydantic.core.query.pattern import Pattern, P, HopSpec

# Simple pattern: Router connected to Switch via any link
pattern = Router >> Link >> Switch
```

```python
# Named bindings with filter
pattern = P("r", Router) >> Link >> P("s", Switch)

# Variable-length paths
pattern = P("r", Router) >> HopSpec.range(1, 3) >> Switch
```

Named bindings allow filtering on pattern elements:

```python
# Filter by binding fields
expr = P["r"].fields.vendor == P["s"].fields.vendor
```

6.5 Worked Example: Finding Under-Provisioned Links

Query: Under-Provisioned Links

Find all physical links where the bandwidth is below 1 Gbps and both endpoints have "core" in their label:

```python
from ank_pydantic import q

slow_core_links = (
    topology.query.links()
    .in_layer("physical")
    .filter(
        q.field('bandwidth') < 1000,
        q.field('src_label').contains('Core'),
        q.field('dst_label').contains('Core'),
    )
    .collect()
)

for link in slow_core_links:
    print(f"Under-provisioned: {link.src_label} <-> {link.dst_label}"
          f" ({link.data.bandwidth} Mbps)")
```

7 Design Rules — The Network Linter

The rule engine is AutoNetKit's headline feature: a composable, declarative system for validating network designs before configuration generation. It acts as a "linter for network designs," catching errors that would otherwise become operational incidents.

7.1 Problem: Imperative Checks vs. Declarative Rules

Imperative validation (ad-hoc `if/else` checks scattered through code) is hard to compose, test, and report on. AutoNetKit needs a structured framework where rules are:

- Independently testable.
- Composable via logical operators.
- Self-documenting (ID, name, description, tags, severity).
- Executable in dependency order.
- Reportable in multiple formats (text, JSON, DataFrame).

7.2 Core Framework

7.2.1 Severity

Three levels of severity:

```python
class Severity(str, Enum):
    INFO = "info"
    WARNING = "warning"
    ERROR = "error"
```

7.2.2 Rule ABC

Every rule inherits from `Rule` and implements `check()`:

Listing 6. The Rule abstract base class.

```python
class Rule(ABC):
    id: ClassVar[str]           # e.g., "validation.isolated_nodes"
    name: ClassVar[str]
    description: ClassVar[str]
    tags: ClassVar[Set[str]] = set()
```

```
    default_severity: ClassVar[Severity] = Severity.ERROR
    depends_on: ClassVar[List[type["Rule"]]] = []

    @abstractmethod
    def check(self, ctx: AnalysisContext) -> List[RuleResult]:
        """Implement rule logic here."""
        pass
```

7.2.3 RuleResult and RuleFinding

Each rule execution produces a `RuleResult` with optional granular `RuleFinding` entries:

```
@dataclass
class RuleResult:
    rule_id: str
    passed: bool
    severity: Severity
    message: str
    affected_nodes: List[int] = field(default_factory=list)
    affected_edges: List[int] = field(default_factory=list)
    findings: List[RuleFinding] = field(default_factory=list)
    tags: Set[str] = field(default_factory=set)

@dataclass
class RuleFinding:
    entity_id: int
    entity_type: Literal["node", "edge", "port"]
    message: str
    severity: Severity = Severity.ERROR
    context: dict[str, Any] = field(default_factory=dict)
```

7.3 Rule Composition

Five composition operators build complex logical rules from simple ones:

Operator	Semantics
`AllOf(r1, r2, ...)`	Passes only if *all* children pass. Failure severity is the maximum of child failures.
`AnyOf(r1, r2, ...)`	Passes if *at least one* child passes.
`Not(rule)`	Inverts the child result.
`IfThen(cond, conseq)`	Evaluates consequence only if condition passes. If condition fails, the overall rule passes (condition not met).
`AtLeast(n, r1, r2, ...)`	Passes if at least n of the children pass.

Listing 7. Composing rules with logical operators.

```
from ank_pydantic.core.analysis import AllOf, AnyOf, IfThen

# All mandatory checks must pass
mandatory = AllOf(
    IsolatedNodesRule(),
    MinConnectionsRule(Router, min_count=2),
    rule_id="mandatory_checks",
)

# Core routers need extra redundancy (conditional)
core_redundancy = IfThen(
    condition=NodeTypeRule(CoreRouter),
    consequence=DualHomedRule(CoreRouter, min_upstreams=3),
    rule_id="core_router_redundancy",
)
```

7.4 Built-In Rules Catalogue

AutoNetKit ships with rules across five categories:

Category	Rule	Purpose
Validation	`IsolatedNodesRule`	Find degree-0 nodes
	`RequiredFieldRule`	Ensure a field is set
	`MinConnectionsRule`	Minimum connections per node
Anomaly	`SinglePointOfFailureRule`	Find articulation points
	`DisconnectedComponentsRule`	Detect network partitions
Pattern	`HubAndSpokePattern`	Detect star topologies
	`FullMeshPattern`	Find fully-connected cliques
	`HighDegreeNodesRule`	Flag potential bottlenecks
Resilience	`DualHomedRule`	Multi-upstream redundancy
	`PathDiversityRule`	Edge-disjoint paths
	`LinkRedundancyRule`	Redundant links
Operational	`DisconnectedDevicesRule`	Find offline devices
	`AsymmetricLinksRule`	Directional mismatches
	`MissingAttributeRule`	Verify attribute presence

Additionally, the `blueprints/design_rules` module provides 27 protocol-specific rules for OSPF, ISIS, BGP, MPLS, SR, EVPN, L3VPN, BFD, and IXP configurations.

7.5 RuleSet and Execution

A `RuleSet` collects rules and executes them with dependency resolution:

Listing 8. Running a RuleSet.

```python
from ank_pydantic.core.analysis import RuleSet, standard_ruleset

# Pre-built standard ruleset
report = standard_ruleset().run(topology)
print(report.to_text())

# Custom ruleset
ruleset = RuleSet("network_design")
ruleset.add(IsolatedNodesRule())
ruleset.add(MinConnectionsRule(Router, min_count=2))
ruleset.add(DualHomedRule(Router, min_upstreams=2))

report = ruleset.run(topology)
print(f"Pass rate: {report.pass_rate:.0%}")
print(f"Errors: {report.error_count}, Warnings: {report.warning_count}")
```

The `standard_ruleset()` factory returns a pre-configured set containing `IsolatedNodesRule`, `DisconnectedComponentsRule`, `SinglePointOfFailureRule`, and `HighDegreeNodesRule`.

7.6 AnalysisReport

The report object provides multiple output formats:

```python
# Summary statistics
report.passed        # True if no error-level violations
report.error_count   # Count of ERROR severity failures
report.pass_rate     # Ratio of passed rules (0.0-1.0)

# Filtering
failed = report.filter_failed()
errors = report.filter_by_severity(Severity.ERROR)

# Export
text = report.to_text(verbose=True)
json_str = report.to_json(indent=2)
df = report.to_dataframe()  # Polars DataFrame
by_tag = report.results_by_tag()
```

7.7 Worked Example: Writing a Custom Rule

Custom Rule: Minimum Interfaces per Router

```python
from ank_pydantic.core.analysis.base import Rule, RuleResult, Severity

class MinInterfacesRule(Rule):
    id = "custom.min_interfaces"
    name = "Minimum Interfaces"
    description = "Each router must have at least N interfaces"
    tags = {"validation", "connectivity"}
    default_severity = Severity.ERROR

    def __init__(self, min_count: int = 2):
        self.min_count = min_count

    def check(self, ctx):
        results = []
        for router in ctx.nodes_of_type(Router):
            degree = ctx.get_degree(router.id)
            result = RuleResult(
                rule_id=self.id,
                passed=degree >= self.min_count,
                severity=self.default_severity,
                message=(f"{router.data.label} has {degree} interfaces"
                         f" (minimum: {self.min_count})"),
                affected_nodes=[router.id] if degree < self.min_count else [],
            )
            results.append(result)
        return results

# Use it
ruleset = RuleSet("custom")
ruleset.add(MinInterfacesRule(min_count=2))
report = ruleset.run(topology)
```

8 Configuration Generation

AutoNetKit compiles validated topologies into vendor-specific device configurations using a registry-based compiler pipeline.

8.1 Compiler Pipeline

The pipeline has four stages:

1. **Platform lookup**: the registry maps a `platform` string (e.g., `"ios"`, `"eos"`) to a compiler class.
2. **Data extraction**: the compiler extracts protocol configuration from the topology (OSPF areas, BGP neighbours, interface addresses).
3. **Template rendering**: Jinja2 templates produce vendor-specific CLI configuration.
4. **Output**: configuration files are written to disk, optionally wrapped in deployment environments (Containerlab, Ansible, netsim).

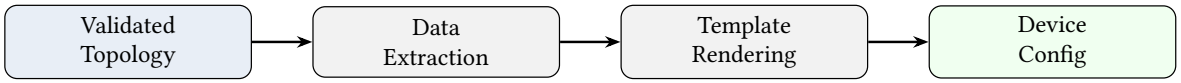

Figure 7. Compiler pipeline: topology to device configuration.

8.2 Supported Platforms

Vendor	Platform ID	Compiler Class
Cisco	`ios`, `iosxr`, `nxos`	IOSCompiler, IOSXRCompiler, NXOSCompiler
Juniper	`junos`	JunOSCompiler
Arista	`eos`	EOSCompiler
Nokia	`sros`, `srlinux`	SROSCompiler, SRLinuxCompiler
Other	`frr`, `sonic`, `vyos`, `cumulus`	FRRCompiler, etc.

8.3 Usage

```python
from ank_pydantic.blueprints.compilers.registry import get_compiler

# Get compiler for a platform
compiler = get_compiler("ios")

# Compile device configuration
config = compiler.compile(topology, device_id=42)

# Render to CLI text
cli_config = compiler.render_config(config)
print(cli_config)
```

8.4 Design Rules

The `blueprints/design_rules` module provides 27 protocol-specific rules organised in three tiers:

Tier 1 (Critical) — Empty topology check. Short-circuits on failure.
Tier 2 (Topology) — OSPF area 0 connectivity, iBGP full mesh or route reflector requirement.
Tier 3 (Protocol) — Per-protocol configuration consistency: OSPF router IDs, ISIS NET format, BGP ASN consistency, MPLS label ranges, SR prefix SID uniqueness, EVPN RD/RT, L3VPN VRF, BFD timers, IXP members.

```python
from ank_pydantic.blueprints.validation import validate_design

failures = validate_design(topology)
for f in failures:
    print(f.format_summary())
```

8.5 Environment Generators

Output can be wrapped for deployment environments:

Environment	Purpose	Output
Containerlab	Container-based emulation	`topology.clab.yml`
Ansible	Configuration management	Playbooks + inventory
netsim	Network simulation	`topology.yml` for netlab

9 End-to-End Tutorial

This section walks through a complete workflow using the ISP running example from Section 2.

9.1 Step 1: Install

```
pip install ank_pydantic
```

AutoNetKit requires the NTE Rust engine (`ank_nte`), which is installed automatically as a dependency.

9.2 Step 2: Define Models

```python
from pydantic import BaseModel
from ank_pydantic.core.models.base import (
    BaseTopologyNode, BaseTopologyEndpoint
)
from ank_pydantic.core.models.edge import BaseInternodeEdge

class RouterData(BaseModel):
    label: str
    role: str = "core"
    asn: int = 65000
    platform: str = "cisco_ios"

class Router(BaseTopologyNode[RouterData]):
    pass

class InterfaceData(BaseModel):
    label: str
    speed: str | None = None

class Interface(BaseTopologyEndpoint[InterfaceData]):
```

```
    pass

class LinkData(BaseModel):
    bandwidth: int | None = None

class Link(BaseInternodeEdge[LinkData]):
    type: str = "link"
```

9.3 Step 3: Build Physical Topology

```
from ank_pydantic.core.topology.topology import Topology

topo = Topology()
topo.nodes.register_models([Router, Interface])
topo.edges.register_models([Link])

# Create core routers
core1 = Router(layer="physical", data=RouterData(label="Core1"))
core2 = Router(layer="physical", data=RouterData(label="Core2"))
topo.nodes.add([core1, core2])

# Create edge routers
edges = []
for i in range(1, 5):
    er = Router(
        layer="physical",
        data=RouterData(label=f"Edge{i}", role="edge"),
    )
    topo.nodes.add([er])
    edges.append(er)

# Create interfaces and connect them
for src, dst in [(core1, core2)] + [(e, core1) for e in edges]:
    src_if = topo.endpoint_mgr.add(
        Interface, layer="physical",
        data=InterfaceData(label="Te0/0"),
        parent_id=src.id,
    )
    dst_if = topo.endpoint_mgr.add(
        Interface, layer="physical",
        data=InterfaceData(label="Te0/0"),
        parent_id=dst.id,
    )
    topo.edges.add(Link(data=LinkData(bandwidth=10000), src=src_if, dst=dst_if))
```

9.4 Step 4: Derive Protocol Layers

```
LAYERS = {
    "ospf": {
        "parent": "physical",
        "keep_edges_where_same": "asn",
    },
    "ebgp": {
        "parent": "physical",
        "keep_edges_where_different": "asn",
    },
}
topo.layers.build(LAYERS)
```

9.5 Step 5: Query

```
# How many routers per layer?
for layer_name in topo.layers.names():
    count = topo.query.nodes().in_layer(layer_name).count()
    print(f"{layer_name}: {count} nodes")

# Find core routers
cores = (
    topo.query.nodes()
    .of_type(Router)
```

```
    .where(role="core")
    .models()
)
print(f"Core routers: {[r.data.label for r in cores]}")
```

9.6 Step 6: Validate

```
from ank_pydantic.core.analysis import standard_ruleset

report = standard_ruleset().run(topo)
print(report.to_text())

if not report.passed:
    print("Design validation FAILED")
    for result in report.filter_failed():
        print(f"  [{result.severity.value}] {result.message}")
```

9.7 Step 7: Generate Configuration

```
from ank_pydantic.blueprints.compilers.registry import get_compiler

for router in topo.query.nodes().of_type(Router).models():
    compiler = get_compiler(router.data.platform)
    config = compiler.compile(topo, router.id)
    cli_text = compiler.render_config(config)
    print(f"--- {router.data.label} ---")
    print(cli_text)
```

9.8 Step 8: batteries_included Shortcut

For quick prototyping, the `batteries_included` module provides pre-built topologies:

```
from ank_pydantic.batteries_included import isp_topology

topo = isp_topology()
print(f"Nodes: {topo.query.nodes().count()}")
print(f"Links: {topo.query.links().count()}")
```

Four topologies are available: `datacenter_topology()` (leaf-spine), `isp_topology()` (core/edge), `campus_topology()` (3-tier), and `wan_topology()` (hub-and-spoke).

10 Limitations

Limitation	Workaround	Planned
NTE compilation requirement — the Rust engine must be compiled for the target platform, which requires a Rust toolchain.	Pre-built wheels for common platforms (Linux x86_64, macOS ARM).	CI-built wheels for PyPI.
NDJSON serialisation overhead — bulk operations between Python and Rust use NDJSON, which is slower than binary formats.	Use batch operations (`add([list])`) rather than single-element calls. Direct PyO3 calls for single-element access.	Arrow IPC or zero-copy serialisation.
Mutable ID assignment — node IDs are assigned on `add()` and are mutable integers, not stable UUIDs.	Use labels for stable identification. Avoid storing raw IDs across serialisation boundaries.	UUID-based stable identifiers.
Batch-only layer derivation — layers are derived in bulk; there is no reactive update when the parent layer changes.	Re-derive layers after parent mod	Reactive layer updates (Section 11).

11 Future Work

Reactive layer updates. When a node or edge in a parent layer changes, derived layers should update automatically. This requires an event system (partially implemented via `EventManager`) and incremental re-derivation.

Distributed topology. For very large networks (10,000+ devices), the NTE engine could be deployed as a server with AutoNetKit as a client. NTE-TR-001 discusses the distributed architecture.

Plugin maturation. The compiler and environment registries support third-party plugins, but the plugin API is not yet stable or documented.

Additional design patterns. Connection patterns beyond full-mesh, ring, chain, star, and hub-spoke (e.g., partial mesh, dual-homed leaf).

Visualisation. A web-based topology visualiser backed by the NTE engine's layout capabilities.

Rename to autonetkit. The package will be renamed from `ank_pydantic` to `autonetkit` before public launch.

A Model Quick Reference

Base Class	Generic	System Fields
`BaseNode[T]`	`T: BaseModel`	`id`, `layer`, `data`
`BaseTopologyNode[T]`	`T: BaseModel`	Inherits + layer traversal methods
`BaseTopologyEndpoint[T]`	`T: BaseModel`	Inherits + `connect_to()`
`BaseLogicalNode[T]`	`T: BaseModel`	Inherits (not first-class)
`RootlessNode[T]`	`T: BaseModel`	Inherits + `_rootless = True`
`BaseEdge[E]`	`E: BaseModel`	`id`, `layer`, `type`, `src_id`, `dst_id`, `data`
`BaseIntranodeEdge[E]`	`E: BaseModel`	`src`: Node, `dst`: Node
`BaseInternodeEdge[E]`	`E: BaseModel`	`src`: Endpoint, `dst`: Endpoint
`BaseOwnershipEdge[E]`	`E: BaseModel`	`src`: Any Node, `dst`: Any Node

B Built-In Rules Catalogue

Rule ID	Description	Tags
`validation.isolated_nodes`	Find degree-0 nodes	validation, connectivity
`validation.required_field`	Ensure a field is set on all nodes	validation, data_quality
`validation.min_connections`	Minimum connections per node type	validation, redundancy
`anomaly.spof`	Single points of failure (articulation points)	anomaly, reliability
`anomaly.disconnected`	Disconnected graph components	anomaly, connectivity
`pattern.hub_spoke`	Detect star topologies	pattern, topology
`pattern.full_mesh`	Detect fully-connected cliques	pattern, topology
`pattern.high_degree`	Flag high-degree bottleneck nodes	pattern, topology
`resilience.dual_homed`	Multi-upstream redundancy check	resilience, redundancy
`resilience.path_diversity`	Edge-disjoint path check	resilience, redundancy
`resilience.link_redundancy`	Redundant link check	resilience, redundancy
`operational.disconnected`	Find offline devices	operational, connectivity
`operational.high_cost`	Flag expensive links	operational, cost
`operational.asymmetric`	Detect directional mismatches	operational, connectivity
`operational.missing_attr`	Verify attribute presence	operational, data_quality
`naming.convention`	Enforce naming patterns (regex)	naming
`addressing.unique`	No duplicate IP addresses	addressing
`addressing.subnet_overlap`	No overlapping subnets	addressing
`addressing.consistent_asn`	BGP ASN consistency	addressing

C Supported Compiler Platforms

Vendor	Platform ID	Template	Notes
Cisco	`ios`	`cisco/ios/device.j2`	IOS and IOS-XE
Cisco	`iosxr`	`cisco/iosxr/device.j2`	IOS-XR (+ protocol templates)
Cisco	`nxos`	—	NX-OS
Juniper	`junos`	`juniper/junos/device.j2`	Junos (MX, EX, QFX, SRX)
Arista	`eos`	`arista/eos/device.j2`	EOS
Nokia	`sros`	`nokia/sros/device.j2`	SR OS
Nokia	`srlinux`	—	SR Linux
FRRouting	`frr`	—	Open-source routing
SONiC	`sonic`	—	Open-source NOS
VyOS	`vyos`	—	Open-source router
Cumulus	`cumulus`	—	Cumulus Linux

D batteries_included Topologies

Factory	Pattern	Nodes	Description
`datacenter_topology()`	Leaf-Spine	2 Spines, 4 Leaves	Non-blocking Clos fabric; Arista EOS
`isp_topology()`	Core/Edge	2 Cores, 4 Edges	Redundant backbone with dual-homed edges; Cisco IOS
`campus_topology()`	3-Tier	1 Core, 2 Dist, 4 Access	Enterprise campus hierarchy; Cisco IOS
`wan_topology()`	Hub-Spoke	1 Hub, 4 Branches	Centralised WAN with serial links; Cisco IOS

Each topology exports four model types: node data, node class, endpoint class, and link class. All models follow the generic `data: T` pattern and can be used as starting points for custom topologies.

List of Figures

List of Tables

List of Design Decisions & Rules