Network Engineers v2.1.0

AutoNetKit User Manual

Type-Safe Network Topology Modelling

Simon Knight

Adelaide, Australia

March 2026

> **About this manual.** This manual covers installing AutoNetKit, building typed network topologies with Pydantic models, deriving multi-layer protocol views, querying topology data with the fluent API, running design-rule validation before configuration generation, and compiling validated designs into vendor-specific device configurations. Chapters on the layer system, the query DSL, built-in design rules, supported output platforms, and ready-made `batteries_included` topologies make this the primary practitioner reference for AutoNetKit 2.1.0.

Contents

1 Quick Start

Go from zero to a running, validated topology in under ten minutes.

Step 1. Install AutoNetKit (the NTE Rust engine installs automatically):

```
pip install autonetkit
```

Step 2. Create and populate a topology:

```python
from pydantic import BaseModel
from autonetkit import Topology
from autonetkit.models.base import BaseTopologyNode, BaseTopologyEndpoint
from autonetkit.models.edge import BaseInternodeEdge

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

class Router(BaseTopologyNode[RouterData]):
    pass

class InterfaceData(BaseModel):
    label: str

class Interface(BaseTopologyEndpoint[InterfaceData]):
    pass

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

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

Step 3. Build the physical topology:

```python
topo = Topology()
r1 = Router(layer="physical", data=RouterData(label="R1"))
r2 = Router(layer="physical", data=RouterData(label="R2"))
topo.nodes.add([r1, r2])

# Add interfaces and connect them
i1 = topo.endpoint_mgr.add(
    Interface, layer="physical",
    data=InterfaceData(label="Eth0/0"), parent_id=r1.id,
)
i2 = topo.endpoint_mgr.add(
    Interface, layer="physical",
    data=InterfaceData(label="Eth0/0"), parent_id=r2.id,
)
topo.edges.add(Link(data=LinkData(bandwidth=10000), src=i1, dst=i2))
```

Step 4. Derive protocol layers, validate, and compile:

```python
# Derive an OSPF layer
```

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

# Validate the design
from autonetkit.analysis import standard_ruleset
report = standard_ruleset().run(topo)
print(report.to_text())

# Compile to IOS configuration
from autonetkit.compilers.registry import get_compiler
compiler = get_compiler("ios")
config = compiler.compile(topo, r1.id)
print(compiler.render_config(config))
```

Step 5. Query node counts per layer:

```
# In a Python REPL or script
for name in topo.layers.names():
    print(name, topo.query.nodes().in_layer(name).count())
```

2 Installation and Prerequisites

2.1 Requirements

▷ **Prerequisites**

- **Python 3.9 or later** (3.11+ recommended for best performance).
- **Pydantic v2** — installed automatically as a dependency.
- **NTE engine** (`ank_nte`) — the Rust graph-storage backend; installed automatically as a dependency.
- **Polars** — columnar query execution; installed automatically.
- An active Python virtual environment is strongly recommended.

2.2 Installing from PyPI

```
pip install autonetkit
```

Tip

Use `uv pip install autonetkit` for significantly faster dependency resolution in projects with large dependency trees. `uv` resolves the Rust wheel for your platform in a single round-trip rather than iterating through pip's backtracking solver.

Caution

The NTE engine (`ank_nte`) ships pre-built wheels for Linux (x86_64, aarch64), macOS (Intel, Apple Silicon), and Windows (x86_64). If you are on an unsupported platform, `pip` will attempt a source build. This requires a stable Rust toolchain (`rustup`) and the `maturin` build tool. Install Rust first with:

`curl –proto '=https' –tlsv1.2 -sSf https://sh.rustup.rs | sh`

2.3 Verifying the Installation

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

Both commands should print `2.1.0` and the NTE version string respectively.

2.4 Upgrading

```
pip install --upgrade autonetkit
```

> **Caution**
>
> AutoNetKit 2.x uses a different NDJSON serialisation protocol to 1.x. Topology objects serialised under 1.x cannot be loaded by 2.x directly. Use `autonetkit.io.migrate_v1(path)` to convert legacy NDJSON exports before loading them into a 2.x `Topology`.

3 Core Concepts

AutoNetKit organises a network topology into three kinds of element:

Nodes represent devices or logical network elements: routers, switches, hosts, VLANs, VRFs. Nodes are first-class graph citizens.

Endpoints represent connection points on devices: physical interfaces, loopbacks, sub-interfaces. Each endpoint belongs to exactly one node via an ownership edge.

Edges represent relationships between elements. Three edge kinds enforce strict connection rules: *internode* (endpoint to endpoint across devices), *intranode* (node to node within a device), and *ownership* (any node to any node, containment).

Every element carries a generic `data: T` field backed by a user-defined Pydantic `BaseModel`. This gives full IDE autocompletion, runtime validation, and free serialisation — with no stringly-typed dictionaries.

The `Topology` façade exposes managers for every domain:

The `Topology` class is a thin façade wiring approximately eighteen domain managers over a shared NTE store. Use dot-completion to explore the API.

Command Reference

Command	Description
`topology.nodes`	Add, remove, and register node models
`topology.endpoint_mgr`	Add and query endpoints (interfaces)
`topology.edges`	Add, remove, and query edges
`topology.links`	Simplified link-level (node-to-node) view
`topology.layers`	Create, derive, freeze, and query layers
`topology.query`	Fluent query API over nodes, edges, links, paths
`topology.io`	Serialise / deserialise (NDJSON, JSON, GraphML)
`topology.analysis`	Run rule sets and retrieve reports

4 Core Workflows

4.1 Defining Custom Models

Every domain-specific element type inherits from one of AutoNetKit's base classes parameterised by a data model:

Inherit `BaseTopologyNode` for devices and `BaseTopologyEndpoi` for interfaces. Both are generic over a Pydantic `BaseModel` subclass.

Step 1. Define a Pydantic data class for each element kind:

```
from pydantic import BaseModel

class RouterData(BaseModel):
    label: str
```

```
    role: str = "core"        # "core" | "edge" | "spine" | "leaf"
    asn: int = 65000
    platform: str = "cisco_ios"
    loopback: str | None = None

class InterfaceData(BaseModel):
    label: str
    speed: str | None = None  # e.g. "10G", "100G"
    ip: str | None = None

class LinkData(BaseModel):
    bandwidth: int | None = None   # Mbps
    metric: int = 1
```

Step 2. Inherit the appropriate base class, parameterised by your data model:

```
from autonetkit.models.base import (
    BaseTopologyNode, BaseTopologyEndpoint,
)
from autonetkit.models.edge import BaseInternodeEdge

class Router(BaseTopologyNode[RouterData]):
    pass   # inherits id, layer, data; extend for domain logic

class Interface(BaseTopologyEndpoint[InterfaceData]):
    pass

class Link(BaseInternodeEdge[LinkData]):
    type: str = "link"   # discriminator used by registry
```

Step 3. Register models with the topology so the identity map can hydrate them:

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

> **Tip**
>
> Use `FlexibleData` (from `autonetkit.models.base`) as the data type during rapid prototyping. It sets `extra="allow"` so any field is accepted without a formal schema. Migrate to a typed model before running design rules or generating configurations.

4.2 Building the Physical Topology

Step 1. Instantiate a `Topology` and add nodes:

```
from autonetkit import Topology

topo = Topology()

core1 = Router(layer="physical",
               data=RouterData(label="Core1", role="core"))
core2 = Router(layer="physical",
               data=RouterData(label="Core2", role="core"))
topo.nodes.add([core1, core2])
# IDs are assigned by NTE after add(); router.id is now set
```

Step 2. Add endpoints (interfaces) owned by each node:

```
# endpoint_mgr.add() creates the endpoint and the ownership edge
c1_eth0 = topo.endpoint_mgr.add(
    Interface, layer="physical",
    data=InterfaceData(label="Eth0/0", speed="10G"),
    parent_id=core1.id,
)
c2_eth0 = topo.endpoint_mgr.add(
    Interface, layer="physical",
    data=InterfaceData(label="Eth0/0", speed="10G"),
    parent_id=core2.id,
)
```

Step 3. Connect endpoints with an internode edge (physical link):

```
link = Link(
    data=LinkData(bandwidth=10000, metric=1),
    src=c1_eth0,
    dst=c2_eth0,
)
topo.edges.add(link)
```

Step 4. Verify the topology:

```
print(topo.query.nodes().in_layer("physical").count())  # 2
print(topo.query.edges().in_layer("physical").count())  # 1
```

> **Caution**
>
> Node IDs are assigned by NTE *after* calling `topo.nodes.add()`. Do not capture `node.id` before the node has been added — it will be `None`. Batch-add is more efficient than adding nodes one by one for topologies with more than a few hundred elements.

4.3 Deriving Protocol Layers

Layer derivation is declarative, not reactive. Call `layers.build()` after all physical nodes and links are in place.

Protocol views are derived from the physical layer using a configuration dictionary:

Step 1. Define the layer configuration:

```
LAYERS = {
    "ospf": {
        "parent": "physical",
        # Clone edges only where both endpoints share the same ASN
        "keep_edges_where_same": "asn",
    },
    "ebgp": {
        "parent": "physical",
        # Clone edges only where endpoints cross AS boundaries
        "keep_edges_where_different": "asn",
    },
    "ibgp": {
        "parent": "physical",
        "copy_edges": False,
        # Group nodes by ASN; create a full mesh within each group
        "group_by": "asn",
        "create": "full_mesh",
    },
}
```

```
```

Step 2. Validate and build:

```
errors = topo.layers.validate(LAYERS)
if errors:
    for e in errors:
        print("Layer config error:", e)
else:
    topo.layers.build(LAYERS)
```

Step 3. Optionally freeze source layers to prevent accidental mutation:

```
topo.layers.freeze("physical")
assert topo.layers.frozen("physical")
```

4.4 Querying Topology Data

AutoNetKit's query API is fluent and type-safe:

Queries are lazy builders. Terminal methods (`.models()`, `.ids()`, `.count()`) execute against Rust.

Step 1. Query all nodes of a type in a layer:

```
# List[Router] -- fully hydrated Pydantic models
core_routers = (
    topo.query.nodes()
    .of_type(Router)
    .in_layer("physical")
    .where(role="core")
    .models()
)
```

Step 2. Use expression filters for complex predicates:

```
from autonetkit import q

# Field comparison
high_bw_links = (
    topo.query.links()
    .in_layer("physical")
    .filter(q.field("bandwidth") >= 10000)
    .collect()
)

# String match + logical AND
cisco_core = (
    topo.query.nodes()
    .of_type(Router)
    .filter(
        (q.field("platform").contains("cisco")) &
        (q.field("role") == "core")
    )
    .models()
)
```

Step 3. Use terminal methods suited to the task:

```
count   = topo.query.nodes().of_type(Router).count()
exists  = topo.query.nodes().where(role="spine").exists()
```

```
first   = topo.query.nodes().of_type(Router).first()
one     = topo.query.nodes().where(label="Core1").one()
df      = topo.query.nodes().of_type(Router).dataframe()
```

Step 4. Navigate across layers:

```
# Find the plan-layer counterpart of a physical router
plan_router = physical_router.in_plan(topo)

# Find the same router in the OSPF layer
ospf_router = physical_router.in_layer("ospf", topo)
```

4.5 Running Design Rules

Step 1. Run the built-in standard ruleset:

```
from autonetkit.analysis import standard_ruleset

report = standard_ruleset().run(topo)
print(report.to_text(verbose=True))
```

Step 2. Check the result and act on failures:

```
if not report.passed:
    print(f"Errors: {report.error_count}, "
          f"Warnings: {report.warning_count}")
    for r in report.filter_failed():
        print(f"  [{r.severity.value.upper()}] {r.message}")
    raise SystemExit(1)
```

Step 3. Export the report as a Polars DataFrame for CI integration:

```
df = report.to_dataframe()
df.write_csv("validation_report.csv")
```

4.6 Generating Device Configurations

Step 1. Look up the compiler for a platform:

```
from autonetkit.compilers.registry import get_compiler

compiler = get_compiler("ios")          # Cisco IOS
# or: get_compiler("eos"), get_compiler("junos"), ...
```

Step 2. Compile and render for each device:

```
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)
    out_path = f"output/{router.data.label}.cfg"
    with open(out_path, "w") as f:
        f.write(cli_text)
```

Step 3. Export for a deployment environment:

```python
from autonetkit.export import ContainerlabExporter

exporter = ContainerlabExporter()
exporter.export(topo, output_dir="lab/")
# Produces lab/topology.clab.yml and per-device config files
```

5 Layer Configuration Reference

A layer configuration dictionary maps layer names to parameter objects. The build system resolves dependencies, validates the configuration, and builds layers in topological order.

5.1 Parameters

Layer Parameters

Command	Description
`parent`	Name of the parent layer to clone nodes from
`keep_edges_where_same`	Clone edges only where `data.<field>` matches on both ends
`keep_edges_where_different`	Clone edges only where `data.<field>` differs
`copy_edges`	Boolean; if `False`, no edges are copied (default: `True`)
`group_by`	Field name; groups nodes by field value for topology creation
`create`	Topology pattern within each group: `full_mesh`, `hub_spoke`, `ring`

5.2 Common Patterns

5.2.1 Full Mesh (iBGP)

Nodes grouped by ASN; a full mesh of virtual links is created within each group. This models iBGP sessions without requiring explicit physical links.

```yaml
ibgp:
  parent: physical
  copy_edges: false
  group_by: asn
  create: full_mesh
```

5.2.2 Same-AS OSPF

Only physical links between routers in the same AS are retained.

```yaml
ospf:
  parent: physical
  keep_edges_where_same: asn
```

5.2.3 Cross-AS eBGP

Only physical links crossing AS boundaries are retained.

```yaml
ebgp:
  parent: physical
  keep_edges_where_different: asn
```

5.2.4 Ring Topology

Group nodes by a field and connect them in a ring (useful for modelling ring access networks).

```
access_ring:
  parent: physical
  copy_edges: false
  group_by: region
  create: ring
```

5.2.5 Hub and Spoke

Group nodes by a field; the node with the highest degree within each group becomes the hub.

```
hub_spoke:
  parent: physical
  copy_edges: false
  group_by: pop
  create: hub_spoke
```

5.3 Cross-Layer Traversal

After derivation, every cloned node carries pointers back to its parent:

```
# Get the physical counterpart of an OSPF node
ospf_router = topo.query.nodes().of_type(Router) \
                  .in_layer("ospf").first()

physical_router = ospf_router.in_plan(topo)       # -> plan layer
ibgp_router     = ospf_router.in_layer("ibgp", topo)  # -> iBGP layer
```

6 Python API Quick Reference

AutoNetKit is primarily a library. The following tables summarise the most commonly used manager methods.

6.1 Node and Endpoint Management

NodeManager — `topology.nodes`

Command	Description
`.add(nodes)`	Add a list of nodes; assigns IDs in place
`.remove(node_id)`	Remove node and all its edges
`.get(node_id)`	Retrieve a node by ID
`.register_models(types)`	Register model classes with the identity map
`.count()`	Total node count across all layers

EndpointManager — `topology.endpoint_mgr`

Command	Description
`.add(cls, layer, data, parent_id)`	Create endpoint and ownership edge
`.get(ep_id)`	Retrieve endpoint by ID
`.of_node(node_id)`	List all endpoints owned by a node

6.2 Edge and Link Management

EdgeManager — `topology.edges`

Command	Description
`.add(edge)`	Add a typed edge; assigns ID in place
`.remove(edge_id)`	Remove edge
`.get(edge_id)`	Retrieve edge by ID
`.count()`	Total edge count

LinkManager — `topology.links`

Command	Description
`.between(a_id, b_id)`	Find links between two nodes
`.neighbours(node_id)`	List neighbour node IDs
`.degree(node_id)`	Connection degree of a node

6.3 Layer Management

LayerManager — `topology.layers`

Command	Description
`.build(config)`	Build layers from config dict
`.validate(config)`	Validate config dict; returns error list
`.get(name)`	Return a `LayerHandle`
`.names()`	List all layer names
`.freeze(name)`	Prevent further mutation
`.frozen(name)`	Check frozen status
`.clone_to_layer(...)`	Clone a node set to a new layer

6.4 Query API

QueryManager — `topology.query`

Command	Description
`.nodes()`	Start a `NodeQuery`
`.edges()`	Start an `EdgeQuery`
`.links()`	Start a `LinkQuery`
`.paths()`	Start a `PathQuery`

NodeQuery chaining

Command	Description
`.of_type(Model)`	Type-narrowing filter
`.in_layer(name)`	Layer filter
`.where(**fields)`	Equality filter on `data.*` fields
`.filter(*exprs)`	Expression-based filter using `q.*`
`.to_plan()`	Navigate to plan-layer counterparts
`.to_layer(name)`	Navigate to another layer

NodeQuery terminals

Command	Description
`.models()`	`List[T]` of hydrated Pydantic models
`.ids()`	`List[int]` of node IDs
`.count()`	`int` (no model materialisation)
`.first()`	`Optional[T]` — first match or `None`
`.one()`	`T` — exactly one match (raises if 0 or 2+)
`.exists()`	`bool`
`.dataframe()`	`pl.DataFrame` (Polars)

6.5 IO and Serialisation

IOManager — `topology.io`

Command	Description
`.save(path)`	Serialise topology to NDJSON
`.load(path)`	Deserialise from NDJSON
`.to_json()`	Return JSON string
`.to_graphml(path)`	Export to GraphML (for visualisation)
`.migrate_v1(path)`	Convert 1.x NDJSON to 2.x format

7 Design Rules Catalogue

The rule engine is AutoNetKit's "network linter": composable, declarative validation that runs before configuration generation, catching design errors before they become operational incidents.

7.1 Built-In Rules

AutoNetKit ships 27 built-in rules across five categories. The `standard_ruleset()` factory runs the four most critical.

Category	Rule ID	Purpose	Severity
Validation	`validation.isolated_nodes`	Nodes with no connections	ERROR
	`validation.required_field`	Mandatory field is set	ERROR
	`validation.min_connections`	Minimum degree per node type	ERROR
	`validation.unique_labels`	No duplicate node labels	WARNING
	`validation.no_self_loops`	Edges do not connect a node to itself	ERROR
Anomaly	`anomaly.spof`	Articulation points (single PoF)	WARNING
	`anomaly.disconnected`	Partitioned topology components	ERROR
	`anomaly.high_degree`	Nodes above degree threshold	WARNING
	`anomaly.orphaned_endpoint`	Endpoints not connected to any link	WARNING
	`anomaly.stub_asn`	ASN with only one external link	INFO
	`anomaly.duplicate_link`	Parallel links between same pair	INFO
Pattern	`pattern.hub_spoke`	Star topology detected	INFO
	`pattern.full_mesh`	Fully connected clique	INFO
	`pattern.ring`	Ring topology detected	INFO
	`pattern.asymmetric_degree`	Unbalanced degree distribution	WARNING
	`pattern.triangle_free`	No triangles in physical layer	INFO
Resilience	`resilience.dual_homed`	Nodes with ≥ 2 upstreams	WARNING
	`resilience.path_diversity`	Edge-disjoint path count	WARNING
	`resilience.link_redundancy`	Redundant links per pair	INFO
	`resilience.min_upstreams`	Minimum upstream count	ERROR
	`resilience.max_radius`	Diameter bound	WARNING
	`resilience.ecmp_capacity`	Equal-cost path count	INFO
Operational	`operational.missing_attr`	Required attribute absent	ERROR
	`operational.asymmetric_link`	Speed mismatch across a link	WARNING
	`operational.platform_support`	Platform has a registered compiler	ERROR
	`operational.loopback_missing`	Routers without a loopback address	WARNING
	`operational.asn_conflict`	Same ASN used by unrelated routers	ERROR

7.2 Rule Composition Operators

Five operators build complex rules from simple leaf rules:

```
from autonetkit.analysis import AllOf, AnyOf, IfThen, Not, AtLeast

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

# Conditional: only check dual-homing on core routers
core_check = IfThen(
    condition=NodeTypeRule(CoreRouter),
    consequence=DualHomedRule(CoreRouter, min_upstreams=3),
    rule_id="core_redundancy",
)

# At least 2 of 3 resilience rules must pass
resilience = AtLeast(2,
    PathDiversityRule(min_paths=2),
    LinkRedundancyRule(),
```

```
    DualHomedRule(Router, min_upstreams=2),
    rule_id="resilience_floor",
)
```

7.3 Writing a Custom Rule

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

# Register and run
from autonetkit.analysis import RuleSet
ruleset = RuleSet("custom_checks")
ruleset.add(MinInterfacesRule(min_count=2))
report = ruleset.run(topo)
```

Tip

Tag rules with category strings (`"validation"`, `"resilience"`, etc.) and use `report.results_by_tag()` to group the report output by category in CI dashboards.

8 Configuration Generation

8.1 Supported Platforms

Vendor	Platform ID	Compiler Class	Protocols
Cisco	`ios`	`IOSCompiler`	OSPF, BGP, ISIS, MPLS
Cisco	`iosxr`	`IOSXRCompiler`	OSPF, BGP, ISIS, SR, BFD
Cisco	`nxos`	`NXOSCompiler`	OSPF, BGP, EVPN, VXLAN
Juniper	`junos`	`JunOSCompiler`	OSPF, BGP, ISIS, MPLS, SR
Arista	`eos`	`EOSCompiler`	OSPF, BGP, EVPN, VXLAN
Nokia	`sros`	`SROSCompiler`	OSPF, BGP, ISIS, SR, L3VPN
Nokia	`srlinux`	`SRLinuxCompiler`	OSPF, BGP, EVPN
FRRouting	`frr`	`FRRCompiler`	OSPF, BGP, ISIS
SONiC	`sonic`	`SONiCCompiler`	BGP, EVPN
VyOS	`vyos`	`VyOSCompiler`	OSPF, BGP
Cumulus	`cumulus`	`CumulusCompiler`	OSPF, BGP, EVPN

8.2 Compiler Usage

```
from autonetkit.compilers.registry import get_compiler, list_platforms

# Discover available platforms
print(list_platforms())

# Compile a single device
compiler  = get_compiler("iosxr")
config    = compiler.compile(topo, device_id=core1.id)
cli_text  = compiler.render_config(config)

# Compile all devices, routing by platform field
for router in topo.query.nodes().of_type(Router).models():
    c   = get_compiler(router.data.platform)
    txt = c.render_config(c.compile(topo, router.id))
    print(f"=== {router.data.label} ===\n{txt}")
```

8.3 Jinja2 Template Customisation

Each compiler resolves templates from a search path. Drop overrides into a local `templates/<platform>/` directory and set the search path:

```
compiler = get_compiler("ios")
compiler.add_template_dir("templates/ios")
# Local templates take precedence over built-in ones
```

Templates receive the compiled `config` object as context. Variables available in all templates:

```
# Available in every Jinja2 template:
hostname:      "Core1"
platform:      "cisco_ios"
interfaces:    [ {label: "Eth0/0", ip: "10.0.0.1/30", speed: "10G"} ]
ospf_areas:    [ {area: 0, interfaces: ["Eth0/0"]} ]
bgp_neighbors: [ {peer_ip: "10.0.0.2", remote_asn: 65002} ]
```

Caution

Not every compiler covers every protocol. If a router's topology data includes BGP configuration but the target compiler does not have a BGP template, the rendered configuration will silently omit BGP. Run the `operational.platform_support` design rule before compiling to catch coverage gaps early.

8.4 Deployment Environment Export

```
# Containerlab (container-based emulation)
from autonetkit.export import ContainerlabExporter
ContainerlabExporter().export(topo, output_dir="lab/")
# Produces: lab/topology.clab.yml

# Ansible (configuration management)
from autonetkit.export import AnsibleExporter
AnsibleExporter().export(topo, output_dir="ansible/")
# Produces: ansible/inventory.yml + playbooks/

# netsim / netlab (network simulation)
from autonetkit.export import NetsimExporter
NetsimExporter().export(topo, output_dir="sim/")
# Produces: sim/topology.yml
```

9 batteries_included Topologies

The `autonetkit.batteries_included` module ships four ready-made topology factories for common use cases. Each factory returns a fully populated `Topology` with typed models, physical links, and derived protocol layers.

9.1 Available Factories

```
from autonetkit.batteries_included import (
    datacenter_topology,
    isp_topology,
    campus_topology,
    wan_topology,
)
```

9.1.1 Leaf-Spine Datacenter

```
topo = datacenter_topology(
    spines=2,       # spine router count
    leaves=4,       # leaf switch count
    asn=65100,
    platform="arista_eos",
)
# Layers: physical, evpn (full mesh between leaves via spines)
```

9.1.2 ISP Internet Exchange

```
topo = isp_topology(
    cores=2,
    edges=4,
    route_servers=1,
    asn_base=65000,     # ASN assigned sequentially
    platform="cisco_iosxr",
)
# Layers: physical, ospf (same-ASN), ebgp (cross-ASN), ibgp (full mesh)
```

9.1.3 Enterprise Campus

```
topo = campus_topology(
    core_switches=2,
    distribution_switches=4,
    access_switches=8,
    platform="cisco_ios",
)
# Layers: physical, stp, ospf
```

9.1.4 WAN Mesh

```
topo = wan_topology(
```

```
    nodes=8,
    seed=42,              # deterministic layout
    topology="partial_mesh",  # or "ring", "hub_spoke"
    platform="junos",
)
# Layers: physical, ospf, ibgp (full mesh)
```

Tip

All battery-included factories accept a `seed` parameter for deterministic topology generation. Use the same seed in CI to ensure reproducible test scenarios.

9.2 Extending a Factory Topology

Factory topologies are ordinary `Topology` objects; you can add nodes, edges, or new layers after construction:

```
topo = isp_topology(cores=2, edges=4, asn_base=65000)

# Add a route reflector
rr = Router(layer="physical",
            data=RouterData(label="RR1", role="rr", asn=65000))
topo.nodes.add([rr])

# Re-derive the iBGP layer to include the new node
topo.layers.build({"ibgp": {"parent": "physical",
                            "copy_edges": False,
                            "group_by": "asn",
                            "create": "full_mesh"}})
```

10 Troubleshooting

Problem 1: NTE compilation error: `NteSerialiseError: invalid NDJSON`

Cause: A Pydantic model contains a field type that the NDJSON codec does not support (e.g., a nested model with circular references, or a custom type without a registered serialiser).

Solution: Add a `model_serializer` to the offending Pydantic class, or convert the field to a JSON-safe type (`str`, `int`, `dict`). Run `model.model_dump_json()` outside of AutoNetKit first to verify the model serialises correctly.

Problem 2: Slow performance when adding thousands of nodes one by one

Cause: Each individual `topo.nodes.add([node])` call crosses the Python–Rust boundary via NDJSON serialisation. For large topologies this overhead dominates.

Solution: Batch all nodes into a single list and call `topo.nodes.add(all_nodes)` once. Batch insertion serialises the entire list in a single NDJSON stream, reducing round-trips by an order of magnitude. Similarly, batch-add edges after all nodes are inserted.

Problem 3: `AttributeError: 'NoneType' object has no attribute 'id'` when reading a node ID immediately after creating it

Cause: The node ID is assigned by NTE *after* `topo.nodes.add()` is called. Capturing `node.id` before the node has been added returns `None`.

Solution: Call `topo.nodes.add([node])` before reading `node.id`. For batch adds, node IDs are set in place on all objects in the list by the time `add()` returns.

Problem 4: Layer derivation does not reflect nodes added after `layers.build()`

Cause: Layer derivation is not reactive. Nodes added to the parent layer after `build()` is called are not automatically cloned to derived layers.

Solution: Re-run `topo.layers.build(LAYERS)` after any structural change to the parent layer. Freeze the parent layer before building to prevent accidental post-build mutations.

Problem 5: Generated configuration is empty or missing protocol sections

Cause: The compiler for the selected platform does not have a Jinja2 template for the required protocol (e.g., no IS-IS template for `cisco_ios`).

Solution: Run the `operational.platform_support` design rule to list missing template coverage. Either add a custom template under `templates/<platform>/` or switch to a platform compiler that supports the required protocol. See Section 8 for the per-platform protocol coverage table.

Problem 6: `LayerValidationError: unknown parent layer 'physical'` when calling `layers.build()`

Cause: The LAYERS dictionary references `"physical"` as a parent, but no nodes have been added to a layer named `"physical"` yet.

Solution: Add nodes with `layer="physical"` before calling `layers.build()`. Alternatively, if using the whiteboard workflow, add a `"physical": {"parent": "whiteboard"}` entry to the LAYERS dictionary to derive the physical layer first.

11 Integration with the Ecosystem

AutoNetKit sits between the network engineer and a broader automation ecosystem. The following tools interoperate directly.

11.1 NTE — Graph Storage and Queries

NTE (`ank_nte`) is the Rust backend that AutoNetKit delegates to for all graph storage and columnar queries. It is installed automatically as a dependency; you do not need to interact with it directly.

For advanced use cases (custom Polars queries, direct graph traversal, petgraph algorithms), import the NTE handle from the topology store:

```
nte_handle = topo._store.nte  # Rust NTE object (PyO3)
df = nte_handle.node_dataframe(layer="physical")  # pl.DataFrame
```

See also: *NTE Technical Reference* (NTE-TR-001) for engine internals: graph storage (`petgraph::StableDiGraph`), Polars columnar tables, and PyO3 binding details.

11.2 NetCfg — Rust-Native Configuration Compiler

NetCfg is an alternative, high-performance configuration compiler written in Rust. Export from AutoNetKit and pass to NetCfg:

```
python -c "
import autonetkit; t = autonetkit.io.load('topology.ndjson')
autonetkit.io.export_json(t, 'topology.json')
"
netcfg compile topology.json --platform junos --output out/
```

See also: *NetCfg User Manual* (NETCFG-UM-001) for compilation targets, platform coverage, and template authoring.

11.3 TopoGen — Seed-Deterministic Topology Generation

TopoGen generates large, parameterised topologies that AutoNetKit can ingest as a starting point for design validation and config generation:

```
from topogen import generate, TopologySpec
from autonetkit.io import from_topogen

spec = TopologySpec(nodes=128, seed=42, topology="partial_mesh")
gen  = generate(spec)
topo = from_topogen(gen)   # Returns a populated Topology object
```

See also: *TopoGen User Manual* (TOPOGEN-UM-001) for seed parameters, topology templates, and large-scale generation strategies.

11.4 netsim / netlab — Network Simulation

AutoNetKit exports Containerlab and netsim topology files directly. Use the `NetsimExporter` to generate simulation input and then run netlab to instantiate the lab:

```
python -c "
from autonetkit.batteries_included import isp_topology
from autonetkit.export import NetsimExporter
t = isp_topology(cores=2, edges=4, asn_base=65000)
NetsimExporter().export(t, 'sim/')
"
cd sim && netlab up
```

See also: *netsim User Manual* for node image configuration and supported platform capabilities in simulation.

11.5 NetVis — Topology Visualisation

AutoNetKit exports GraphML for immediate ingestion by NetVis:

```
topo.io.to_graphml("topology.graphml")
# Open in NetVis or: netvis render topology.graphml
```

See also: *NetVis User Manual* (NETVIS-UM-001) for layout algorithms, layer overlays, and export formats.

11.6 Workbench — Orchestration

Workbench is the orchestration layer that drives the full design-to-deployment pipeline: it calls AutoNetKit for modelling, NetCfg for compilation, and netsim or Ansible for deployment. Individual AutoNetKit scripts can be wrapped as Workbench tasks:

```
# workbench.yml
tasks:
  - name: build_topology
    type: autonetkit
    script: build_isp.py
    output: topology.ndjson

  - name: compile_configs
    type: netcfg
    input: topology.ndjson
    platform: iosxr
    output: configs/
```

See also: *Workbench User Manual* (WB-UM-001) for task graph configuration, secrets management, and CI/CD integration.

12 Further Reading

- **ANK-TR-001** — *AutoNetKit: A Type-Safe Framework for Network Topology Modelling.* Architecture deep-dive: generic Pydantic data model, manager-first façade, layer mechanics, query expression engine, rule composition operators, and the configuration generation pipeline. Required reading before extending AutoNetKit with custom managers or compilers.
- **NTE-TR-001** — *NTE: Network Topology Engine Technical Reference.* Rust engine internals: `petgraph::StableDiGraph` storage, Polars columnar property tables, NDJSON serialisation protocol, and PyO3 binding design.
- **NETCFG-UM-001** — *NetCfg User Manual.* Rust-native configuration compiler: platform registry, Jinja2 template authoring, and batch compilation strategies.
- **TOPOGEN-UM-001** — *TopoGen User Manual.* Seed- deterministic topology generation: parameter reference and integration with AutoNetKit.
- **NETVIS-UM-001** — *NetVis User Manual.* Topology visualisation: GraphML import, interactive layer overlays, and SVG/PNG export.
- **WB-UM-001** — *Workbench User Manual.* End-to-end orchestration: task graphs, AutoNetKit integration, and deployment pipelines.