Network Automation Ecosystem

Architecture, Data Contracts, and Tool Integration

Simon Knight

Independent Researcher, Adelaide, Australia

March 2026 • Version 1.0

Abstract. The Network Automation Ecosystem is a collection of ten specialized tools—written primarily in Rust and Python—for network topology generation, modeling, protocol simulation, traffic analysis, advanced analysis, visualization, configuration parsing, device validation, and orchestration. This technical report documents the ecosystem architecture, the data contracts (RFC-01, RFC-02, RFC-03) that bind the tools together, the integration patterns that enable end-to-end workflows, and the intelligence layer (netassure) that extends the ecosystem from deterministic simulation into formal verification, graph analytics, failure modeling, and machine learning. The report is intended for engineers evaluating the ecosystem for network automation, contributors to the codebase, and researchers interested in composable network toolchain design.

Version	Date	Changes
1.0	March 2026	Initial release

Contents

1 Introduction

The Network Automation Ecosystem is a unified collection of ten Rust and Python tools for network topology generation, modeling, protocol simulation, traffic analysis, advanced analysis, visualization, configuration parsing, device validation, and orchestration. The ecosystem follows the Unix philosophy: each tool does one thing well, communicates through well-defined data contracts, and composes into workflows that span the full network lifecycle—from whiteboard sketch to production validation.

The ecosystem addresses a gap in the network automation landscape: no existing tool or platform provides an integrated, open-source pipeline from topology generation through protocol simulation, traffic analysis, formal verification, and visualization. Individual capabilities exist (ContainerLab for lab deployment, Batfish for configuration analysis, NetworkX for graph algorithms), but the "glue" between them is manual file conversion, ad-hoc scripting, and implicit assumptions about data formats.

Four architectural tenets guide every design decision in the ecosystem:

1. **Standard-Agnostic Core.** The ecosystem does not force a single schema or vendor format. The Network Topology Engine (NTE) acts as a universal translator, ingesting data from NetBox, ContainerLab, or custom sources via a mapping-driven interface.
2. **Hyper-Composable Entry Points.** There is no fixed beginning or end. Users can import from a Source of Truth, generate from scratch, or capture live state—entering the pipeline at whichever tool matches their starting point.
3. **Logical-to-Physical Decoupling (RFC-01).** The Physical Topology (`.topo.yaml`) is split from the Logical Intent (`.design.yaml`). This decouples engineering logic from specific vendor hardware, enabling vendor migration without redesigning protocol configurations.
4. **Contract-First Integration (ADR-0001).** Tools interoperate through versioned file-based contracts, not through tight in-process coupling. This preserves modularity, enables independent evolution, and makes every integration point debuggable with standard text tools.

This document is intended for three audiences:

- **Network engineers** evaluating the ecosystem for pre-deployment validation, design exploration, or CI/CD integration.
- **Contributors** to any of the ten tool repositories who need to understand cross-project integration decisions.
- **Researchers** interested in composable network toolchain design, graph-based network modeling, or GNN applications in networking.

Familiarity with IP routing protocols (OSPF, BGP) and basic understanding of network topology concepts is assumed. Experience with specific tools (ContainerLab, Batfish, PyATS) is helpful but not required.

Section 2 provides an ecosystem overview with the ten-tool table, technology stack, and maturity assessment. Section 3 defines the data contracts (RFC-01, RFC-02, RFC-03) that form the interoperability backbone. Section 4 describes the integration architecture: pipeline stages, orchestration, and cross-cutting concerns. Section 5 surveys each tool using a consistent five-point template. Section 6 covers the intelligence layer (netassure) in depth: five analysis paradigms, multi-source analysis, and GNN architecture. Section 7 walks through four end-to-end workflows with gap analysis. Section 8 positions the ecosystem against the competitive landscape. Section 9 summarizes the five Architecture Decision Records. Section 10 documents known limitations honestly. Section 11 outlines the implementation roadmap and future work.

Readers wanting a quick overview should read sections 2 and 3, then refer to individual sections as needed.

2 Ecosystem at a Glance

2.1 The Ten Tools

The ecosystem comprises ten specialized tools spanning six architectural layers: generation, modeling, simulation, analysis, parsing, and presentation. Table 1 summarizes each tool's role, implementation language, and maturity.

Table 1. Ecosystem tools, languages, and maturity (as of March 2026).

Tool	Language	Purpose	Status
topogen	Rust + PyO3	Generate realistic network topologies	v1.5
autonetkit	Python + Rust	Model, query, validate, and configure networks	Active
NTE	Rust + PyO3	High-performance graph engine	Stable
netsim	Rust	Deterministic protocol simulation	v2.1
netflowsim	Rust	Flow-based traffic analysis	Active
netassure	Rust + Python	Advanced analysis (5 paradigms)	v0.1
netvis	Rust + PyO3 + WASM	Topology visualization and layout	v1.9
configparsing	Python	LLM/RAG-based config parsing and analysis	v1.0
Workbench	Python + React	Orchestration platform	v4.2
deviceinteraction	Rust	Device testing and validation	v1.1

2.2 Maturity Assessment

The tools span a wide maturity range (table 2). The most mature tools are the most standalone; the tools that *bridge* the ecosystem are the least mature. This is the classic integration problem: individual components get polished while the seams between them lag behind.

Table 2. Maturity tiers with quantitative indicators.

Tier	Tool	Version	Tests	LOC	Key Indicator
Mature	netsim	v2.1	2,616	125K Rust	Enterprise protocols shipped
Mature	netvis	v1.9	1,535	92K Rust	Scale & export milestone
Mature	NTE	v0.2	—	125K Rust	16 crates, dual-write hardened
Mid	topogen	v1.5	869	49K Rust	Intent-based overlays
Mid	Workbench	v4.2	430	34K Py+React	Designer split planned
Mid	autonetkit	—	—	Py+Rust	Batteries-included module
Mid	configparsing	v1.0	—	Python	v2.0 production layer
Mid	deviceinteraction	v1.1	—	Rust	Verification framework
Early	netflowsim	v0.1	—	15K Rust	Profiling phase
Early	netassure	v0.1	—	—	Architecture phase

2.3 Shared Technology Stack

The ecosystem converges on a shared technology foundation (table 3). All Rust projects independently adopted the same libraries—petgraph for graphs, serde for serialization, clap for CLI, trait-based extensibility for plugins. This convergence was organic, not mandated, validating the choices.

Table 3. Shared technology stack across the ecosystem.

Technology	Used By	Purpose
petgraph	topogen, NTE, netflowsim, netvis	Graph data structure
PyO3 + maturin	topogen, NTE, netvis, netassure	Python–Rust FFI
serde + serde_yaml	all Rust tools	Serialization
clap	topogen, netsim, netvis, netflowsim	CLI parsing
Polars	NTE	Columnar DataFrame
Pydantic v2	autonetkit, Workbench, configparsing	Type-safe validation
FastAPI	autonetkit, Workbench	REST API
LangChain/RAG	configparsing	LLM-based extraction

3 Data Contracts

The data contracts are the ecosystem's interoperability backbone. Without them, each tool would define its own ad-hoc format, and integration would devolve into brittle format-conversion scripts. Three RFCs define the contract system; ADR-0001 establishes the architectural principle that contracts, not in-process coupling, are the primary integration boundary.

3.1 The Multi-File Sidecar Strategy (RFC-01)

RFC-01 defines a layered, file-based approach where each file has a single owner and a specific role. The key insight is **logical-to-physical decoupling**: protocol designs reference abstract interface IDs (`p1`, `p2`), not vendor-specific names (`GigabitEthernet0/1`). If the hardware changes from Cisco to Juniper, only the topology file's vendor mapping changes; the design file remains untouched.

3.1.1 The Topology File (`*.topo.yaml`)

- **Owner:** topogen or manual entry
- **Role:** Defines the "Physical World"—nodes, physical links, interface mapping, geographic coordinates, hardware metadata
- **Key Feature:** Logical-to-physical interface mapping table (abstract ID → vendor name)

Listing 1. Topology file with logical-to-physical mapping.

```
nodes:
  - id: lon-spine-01
    interfaces:
      - id: p1
        vendor_name: GigabitEthernet0/1
        speed: 100G
```

3.1.2 The Design Sidecar (`*.design.yaml`)

- **Owner:** autonetkit-config or manual entry
- **Role:** Defines the "Logical Intent"—IP addressing, protocol configs (OSPF, BGP, IS-IS), VRFs, ACLs
- **Pointer:** References a base topology via `base_topology`

Listing 2. Design sidecar referencing logical IDs.

```
schema: netauto/design/v2.0
base_topology: ./backbone.topo.yaml
protocols:
  ospf:
    area: 0
    interfaces:
      - node: lon-spine-01:p1   # Logical ID
        cost: 10
```

Design Decision: Why Sidecar Files?

The ecosystem could have used a single monolithic file combining topology, design, and results. The sidecar approach was chosen because:

1. **Single ownership:** Each file has exactly one tool that writes it. No merge conflicts between tools.
2. **Non-destructive outputs:** Simulation results never mutate the source topology or design files.
3. **Git-friendly:** Each sidecar has a clear diff history. Design changes are visible in `.design.yaml` diffs without noise from simulation output.
4. **Selective consumption:** netvis can render a topology without loading simulation results. netsim can simulate without traffic matrices.

3.1.3 The Result Sidecar (`*.results.json`)

- **Owner:** netsim, netflowsim
- **Role:** Defines "Execution Output"—converged routing tables, link utilization, convergence timing, error logs
- **Stability:** References Node/Interface IDs from the Topology/Design files, never invents new IDs

3.1.4 The OperationalTopology Snapshot (`*.operational.json`)

- **Owner:** Normalizers and operational sources (autonetkit-analysis, ank-parse, SoT importers)
- **Role:** Defines "Operational State" as a versioned snapshot aligned to RFC-01 identity conventions
- **Contract ID:** `netauto/operational-topology/v1.0`
- **Schema:** JSON Schema with validation, field ownership documentation, and sample fixtures

Figure 1 illustrates the relationships between sidecar files.

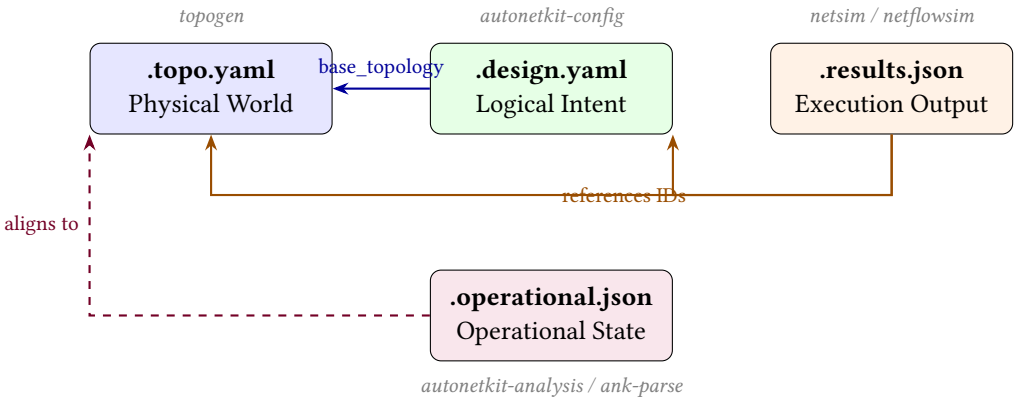

Figure 1. Sidecar file relationships. Solid arrows indicate direct references; dashed arrows indicate identity alignment.

3.2 ID Stability and Naming Conventions

Stable, human-predictable identifiers are the linchpin of the sidecar strategy. If re-running topogen produces different IDs, every downstream sidecar breaks.

Table 4. ID conventions across the ecosystem.

Entity	Pattern	Rules
Node ID	`[site]-[role]-[index]`	Deterministic from seed; e.g., `nyc-leaf-01`
Interface ID (Logical)	`p1`, `p2`, ...	Abstract; decouples design from hardware
Link ID (Derived)	`node_a:pX–node_b:pY`	Never manually named; derived from endpoints

3.3 The Project Manifest (RFC-02)

While RFC-01 defines the data *artifacts*, RFC-02 defines how artifacts are *associated and orchestrated*. The project manifest (`netauto.project`) is a lightweight YAML file—a "Makefile" for network engineering—that links sidecars together, tracks data sources, and coordinates tool execution.

Listing 3. Project manifest linking layers, artifacts, and hooks.

```yaml
project_name: "London-DC-Migration"
version: "2.0"

layers:
  topology:
    provider: "file"
    path: "./backbone.topo.yaml"
  design:
    provider: "git"
    url: "https://github.com/org/net-designs.git"
    ref: "main"
    path: "sites/london/ospf.design.yaml"

artifacts:
```

```
  fib: "./results/london.fib.json"
  analysis: "./results/utilization.results.json"

hooks:
  telemetry:
    type: "websocket"
    url: "ws://telemetry.local:8080/stream/london"
    target: "netvis"
```

3.3.1 Provider Architecture

The manifest abstracts data sources through **providers**, supporting multiple entry points into the ecosystem:

Table 5. RFC-02 provider types.

Provider	Use Case
`file`	Local development, rapid prototyping
`git`	Version-controlled designs, CI/CD pipelines
`http`	Pulling traffic matrices from monitoring APIs
`netbox`	Direct Source of Truth integration
`live`	Real-time network state for live analysis

3.3.2 Live Overlay Stream Contract

The `hooks.telemetry` WebSocket hook is pinned to the Live Overlay Stream v1.0 contract (`netauto/live-overlay-stream/v1.0`). Key semantics:

- **URL scoping:** Stream path implies topology scope (`/stream/{topology_id}`)
- **Delivery:** Best-effort with no ordering guarantees; consumers deduplicate by `event_id`
- **Compatibility:** Consumers ignore unknown fields; no breaking changes within v1.x

3.4 Interface Representation (RFC-03)

RFC-03 resolves OQ-02 (Open Question 2): how should the Network Topology Engine represent interfaces in the graph model? The NTE currently models endpoints as graph nodes, which preserves endpoint identity and supports unconnected ports, but creates friction for consumers that think in devices.

Design Decision: Endpoints-as-Nodes + Device-Centric API

Decision: Keep the current endpoints-as-nodes representation as the underlying storage model. Make device-centric abstraction APIs (Option D) the primary consumer-facing surface.
Evidence: Phase 7 benchmarks showed the baseline is fast for bounded structural queries. Option D materially improves ergonomics without forcing a structural migration. Option C (dual-graph) remains a validated prototype with defined trigger conditions for revisiting.
Trigger conditions for dual-graph: Memory becomes a hard blocker, build time prevents interactive use, or intelligence layer inference requires a device-level graph for feasible latency.

3.5 Tool Requirements (The "Unix" Contract)

Each tool has specific obligations under the contract system:

Table 6. Per-tool contract obligations.

Tool	Obligation
topogen	Export logical IDs (`p1`) and vendor_name mapping
autonetkit-config	Produce `.design.yaml` referencing logical IDs
netsim	Accept `.design.yaml`, resolve IDs via base `.topo.yaml`
netvis	Support layered rendering (base topology + design/result overlays)
Workbench	Manage file associations and relative paths between sidecars

4 Integration Architecture

4.1 Data Flow Pipeline

The tools form a natural pipeline (fig. 2). Data flows through five stages: Generate, Model, Simulate, Analyze, and Visualize. The pipeline is not strictly linear—tools can be entered at any stage, and results flow back for iteration.

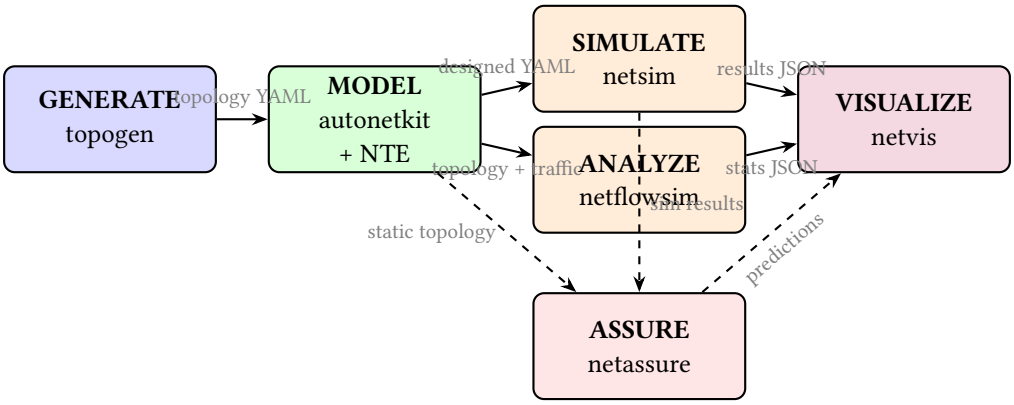

Figure 2. Data flow pipeline. Solid arrows are implemented; dashed arrows are planned or partially implemented.

4.2 Integration Interfaces

The current integration between tools uses file-based exchange as the primary mechanism, with in-process FFI where performance demands it (table 7).

Table 7. Tool-to-tool integration mechanisms.

From	To	Mechanism	Format
topogen	autonetkit	File (YAML)	Native or ANK YAML
topogen	netsim	File (YAML)	netsim topology YAML
topogen	netflowsim	File (JSON)	GeoJSON + traffic CSV
autonetkit	NTE	In-process (PyO3)	Rust FFI
autonetkit	netsim	File (YAML)	`export_netsim()`
autonetkit	netvis	File (JSON)	Graph JSON
netsim	Workbench	Subprocess + gRPC	UDS + protobuf
netvis	Workbench	Subprocess	JSON in, SVG out
deviceinteraction	live devices	SSH/Telnet/REST	CLI commands
deviceinteraction	autonetkit-analysis	File (JSON)	OperationalTopology
configparsing	autonetkit-analysis	File (JSON)	OperationalTopology
configparsing	deviceinteraction	Pipeline	CLI output → parsed config

4.3 Orchestration Architecture

ADR-0003 established that orchestration is a shared layer, not owned by any single tool. The project manifest (RFC-02) provides the coordination surface. Three entry points consume the same orchestration semantics:

1. **CLI (`ank`):** Unix-style pipes and filters for power users and CI/CD pipelines
2. **TUI (`ank-tui`):** Interactive terminal interface (Textual) for rapid design iteration
3. **Workbench (Web):** Full visual IDE (React/TypeScript) for multi-layer visualization and interactive simulation

All three are manifest-driven: they read and write to the same `netauto.project` file, allowing engineers to switch between them during a project lifecycle.

The orchestrator provides: provider resolution (fetching data from file/git/http/netbox/live sources), artifact dependency tracking (flagging stale outputs when inputs change), and tool runners (invoking binaries with correct arguments and parsing output).

4.4 Cross-Cutting Concerns

4.4.1 Determinism

Determinism is a deliberate cross-cutting architectural decision:

- **topogen:** ChaCha8Rng seeding, BTreeMap property ordering, sorted node IDs
- **netsim:** BTreeMap device iteration, explicit tie-breaking in best-path selection, convergence audits
- **netflowsim:** Per-thread seeded RNG for Monte Carlo
- **netvis:** DeterministicMode for reproducible layouts

This enables CI/CD regression testing of network designs—a capability no competitor offers.

4.4.2 Graph Representation

Each tool optimizes its graph representation for its use case (table 8).

Table 8. Graph representations across the ecosystem.

Tool	Graph Type	Direction	Reason
topogen	petgraph Undirected	Undirected	Links are bidirectional
NTE	petgraph StableDiGraph	Directed	Parent/child hierarchy
netsim	Custom BTreeMap	Directed	Deterministic iteration
netflowsim	petgraph StableGraph	Undirected	Stable indices for failure sim
netvis	Custom NetVisGraph	Undirected	Layout algorithms

4.4.3 Trait-Based Extensibility

Every Rust project uses the same compositional pattern: define a trait, implement per-variant, compose. `TopologyGenerator`, `TopologyConverter`, `PlacementStrategy`, `QueueingModel`, `LayoutAlgorithm`—all follow this shape. This is the right foundation for an ecosystem that needs to grow without constant refactoring.

5 The Tools

Each tool is documented using a consistent five-point template: (1) one-paragraph summary, (2) architecture sketch, (3) design decision boxes, (4) integration points, and (5) cross-reference to per-tool technical report. Tool-internal depth (algorithms, full API, benchmarks) is deferred to per-tool TRs.

5.1 Topology Generator (topogen)

Repository: `topogen/` **Language:** Rust + PyO3 **Status:** v1.5 (Intent-Based Overlays), 49K LOC, 869 tests

topogen generates deterministic, realistic network topologies with vendor-specific interface naming, geographic placement, and traffic matrix generation. It provides nine generators spanning data center (Fat-Tree, Leaf-Spine), WAN (Ring, Mesh, Hierarchical, POP), and random graph (Barabási-Albert, Watts-Strogatz, Erdős-Rényi) categories. All generators implement the `TopologyGenerator` trait and produce seed-deterministic output via ChaCha8Rng.

Architecture: Stateless generators with a unified `TopologyGenerator` trait. `GeneratorConfig` (base) composes with topology-specific configs (e.g., `FatTreeConfig`). Interfaces are stored on nodes, not edges. A validation pipeline with pluggable validators runs post-generation. Output formats include Native YAML (round-trip safe), ContainerLab YAML, and AutoNetKit YAML. Traffic matrices use a gravity model with configurable distance exponent, asymmetry policies, and temporal dynamics. The v1.5 milestone adds intent-based overlays: multi-overlay tagging (BGP, OSPF, Physical), automatic role assignment (`spine`/`leaf`, `p`/`pe`/`rr`), schematic grouping (Site, Pod, Zone hierarchies), and semantic edge relationships (`peering`, `upstream`, `transit`, SRLG fate-sharing).

> **Design Decision: Interfaces on Nodes**
>
> topogen stores interface definitions on nodes rather than on edges. This matches the physical reality (a router *has* interfaces; a link *connects* interfaces) and aligns with RFC-01's logical-to-physical mapping. The trade-off is that link metadata requires looking up both endpoint nodes, but this is a one-time cost during export.

Integration Points

Produces:	`.topo.yaml` (RFC-01), ContainerLab YAML, netsim YAML, traffic matrices (CSV/JSON)
Consumes:	Nothing (entry point tool)
Contracts:	RFC-01 (topology file owner)

Cross-reference: TOPOGEN-TR-001 (per-tool technical report).

5.2 Network Modeling and Configuration Library (autonetkit)

Repository: `ank_pydantic/` **Language:** Python + Rust (NTE) **Status:** v1.8

autonetkit is a type-safe Python library for declarative network topology modeling, multi-layer transformation, design validation, and multi-vendor configuration generation. It is positioned as "a linter for network designs"—the only tool that models topologies with typed Python, queries them with a Rust-backed API, and validates engineering correctness at design time.

Architecture: Three-package split following data flow direction: `autonetkit` (core: topology model, query API, rule engine, NTE integration), `autonetkit-config` (design → deploy: IP allocation, vendor compilers, Jinja2 templates, environment exporters), `autonetkit-analysis` (reality → model: NSOT importers, config normalization, operational rules). The Facade pattern delegates to 15+ specialized managers. The `TopologyInterface` ABC enables both in-process (`InProcessTopology`) and remote (`RemoteTopology`) operation. A "batteries included" module provides pre-built topology scenarios for rapid prototyping.

Alternative: ank-netcfg (Rust compiler). `ank-netcfg` is a Rust-native alternative to the entire autonetkit modeling and configuration stack. Where autonetkit (ank_pydantic) is a Python library with Rust acceleration via NTE, ank-netcfg is a pure Rust CLI compiler that takes declarative YAML blueprints through: blueprint parsing → topology transformation → DeviceIR generation → MiniJinja template rendering → traceable `.cfg` emission. Currently at v1.2 (front and back ends shipped), with v1.3 targeting advanced primitives (route reflectors, edge cloning), state validation mode, and benchmarking. The two tools serve different audiences: autonetkit for interactive Python workflows and library consumers, ank-netcfg for deterministic, auditable CI/CD pipelines where a single compiled binary is preferred.

Design Decision: Three-Package Split by Data Direction

The split follows **data flow direction**, not pipeline stage. Config generation flows *outward* from the design model; analysis flows *inward* from external sources. The core library has no opinion about what you do with a topology. This prevents the "kitchen sink" problem where every feature lands in one package, and isolates heavy dependencies (NAPALM, pynetbox in analysis; Jinja2 in config) from the lightweight core.

Integration Points

Produces:	`.design.yaml` (RFC-01), vendor configs (11 platforms), netsim YAML, netvis JSON, Operational-Topology
Consumes:	`.topo.yaml` (from topogen), NetBox/Nautobot data, legacy configs
Contracts:	RFC-01 (design file owner), RFC-02 (manifest consumer)

Cross-reference: No per-tool TR exists yet. Design validation covers 31 rules across 8 categories (graph structure, resilience, addressing, cross-endpoint consistency, routing protocol, hierarchy, addressing discipline).

5.3 Network Topology Engine (NTE)

Repository: `ank_nte/` **Language:** Rust + PyO3 **Status:** v0.2, 16-crate workspace, 125K LOC

The Rust engine that powers autonetkit's graph operations. Extracted into its own repository as scope grew beyond a backing store into a full-featured graph database with query engine, pluggable datastores, and event sourcing. The dual-write architecture has been hardened with rollback safety.

Architecture: 16-crate workspace: `nte-graph` (petgraph wrapper with bidirectional ID mapping), `nte-topology` (high-level API), `nte-domain` (pure domain types), `nte-query` (Cypher-inspired lazy query engine with arithmetic expression compilation), `nte-datastore-*` (pluggable backends: Polars, DuckDB, Lite), `nte-server` (Axum HTTP/WebSocket), `nte-monte-carlo` (Monte Carlo engine), `nte-policy` (policy evaluation and reporting).

Key design: bidirectional ID mapping (external IDs ↔ internal petgraph NodeIndex), pluggable datastore via feature flags, lazy query execution (patterns compiled to plans, executed at materialization), optional event sourcing via ring-buffer EventStore, and graph transformations (`split_edge`, `merge_nodes`, `explode_node`).

Integration Points

Produces:	Graph operations, query results, DataFrames
Consumes:	Topology data (in-process from autonetkit via PyO3)
Contracts:	Internal API; RFC-03 governs representation decisions

Cross-reference: NTE-TR-001 (per-tool technical report).

5.4 Network Simulator (netsim)

Repository: `network-simulator/` **Language:** Rust **Status:** v2.1 (Enterprise & Campus shipped), 125K LOC, 2,616 tests

netsim is a deterministic, tick-based network simulator that bridges the gap between algorithmic configuration analysis and container-based emulation. It provides protocol-accurate simulation of OSPF, IS-IS, BGP (including EVPN/VXLAN and L3VPN), MPLS/LDP, SR-MPLS, RSVP-TE, BFD, GRE, LACP, and LLDP with deterministic convergence detection.

Architecture: Two core principles: (1) convergence is a first-class time coordinate (not a side effect of "running long enough"), and (2) determinism over fidelity (RFC behavior, not vendor quirks). The simulation engine uses a three-phase tick: `tick_control()` → `tick_forwarding()` → `tick_finalize()`. All devices process in lockstep using BTreeMap iteration order. Convergence detection is a precisely defined predicate: simultaneous stability of 11 protocol states over a rolling window.

Daemon mode provides gRPC access via Unix Domain Sockets with Cisco IOS-style CLI commands and prefix abbreviation. Telemetry exports include BMP (real-time BGP state), PCAP (PCAPNG format), and NetFlow.

Design Decision: Convergence as Time Coordinate

Unlike container-based emulators that rely on "wait long enough," netsim defines convergence as an observable, addressable predicate. Events, assertions, and chaos engineering reference convergence as a time anchor (`at: converged + 500ms`). This makes simulation results reproducible and scriptable—essential for CI/CD pipelines.

Integration Points

Produces:	`.results.json` (RFC-01), BMP telemetry, PCAP captures, routing tables
Consumes:	Topology YAML (from topogen or autonetkit)
Contracts:	RFC-01 (result file producer)

Cross-reference: NETSIM-TR-001 (per-tool technical report, 45+ pages).

5.5 Network Traffic Simulator (netflowsim)

Repository: `netflowsim/` **Language:** Rust **Status:** v0.1, Phase 3/5

netflowsim performs flow-based network performance analysis using analytic queuing models and Monte Carlo simulation. It validates topologies at massive scale (1M+ flows/second on 10K+ node topologies).

Architecture: Three stages: Topology + FIBs + Traffic → Routing Matrix Generation → Monte Carlo Simulation → Statistics. Queuing models are trait-based (M/M/1, M/D/1, M/M/c, M/G/1), all O(1) per link using direct formulas. The Monte Carlo engine uses Rayon `par_iter` across iterations with lock-free, thread-local RNG. Routing matrix generation builds IP-to-node mapping and LPM tables from per-device FIBs, then traces paths with ECMP support.

Integration Points

Produces:	Link utilization statistics (JSON), GeoJSON with link overlays
Consumes:	GeoJSON topology + traffic CSV (from topogen), FIBs (from netsim, planned)
Contracts:	RFC-01 (result file producer)

Cross-reference: No per-tool TR exists yet.

5.6 Advanced Analysis Engine (netassure)

Repository: `netassure/` **Language:** Rust + Python (PyO3) **Status:** v0.1, architectural phase

netassure is a multi-paradigm computational analysis engine combining formal verification, graph algorithms, failure cascade modeling, machine learning (GNN), and optimization. It is the ninth tool in the ecosystem, filling the architectural gap between lightweight validation (autonetkit-analysis) and full simulation (netsim/netflowsim). See section 6 for in-depth coverage.

Integration Points

Produces:	Verification results, predictions, optimization suggestions, Live Hook overlays
Consumes:	`.topo.yaml`, `.design.yaml` (static), `.results.json` (simulation), Prometheus/BMP/NetFlow (telemetry)
Contracts:	RFC-01 (all artifact types as consumer)

Cross-reference: No per-tool TR exists yet. ADR-0005.

5.7 Network Visualization Engine (netvis)

Repository: `netvis/` **Language:** Rust + PyO3 + WASM **Status:** v1.9 (Scale & Export milestone), 92K LOC, 1,535 tests

netvis renders complex multi-layer topologies with five layout algorithms (force-directed, hierarchical, radial tree, isometric multi-layer, geographic), advanced edge bundling and routing, and high-quality static output (SVG, PNG via resvg, PDF via svg2pdf).

Architecture: Rendering pipeline: Layout Algorithm → Scene (intermediate representation) → SVG Renderer → Export. Multi-layer support via LayerRegistry with z-ordering (Physical, L2, IP, OSPF, iBGP, eBGP) and GroupRegistry for containment (AS, Datacenter, VLAN, Region). Seven themes with WCAG contrast compliance. Three output targets: native Rust library/CLI, Python module (PyO3), and WebAssembly (wasm-bindgen).

Design Decision: Renderer, Not Editor (ADR-0004)

netvis focuses on rendering and overlays; editing and authoring live elsewhere. This prevents netvis from becoming a monolith and keeps topology authoring in the Workbench (UX) or CLI/TUI. netvis consumes RFC-01 artifacts; it does not become the primary editor or project manager.

Integration Points

Produces:	SVG, PNG, PDF renders; WASM embeddable viewer
Consumes:	Topology JSON (from autonetkit), results JSON (from netsim/netflowsim), Live Hook telemetry stream
Contracts:	RFC-02 (Live Overlay Stream consumer)

Cross-reference: NETVIS-TR-001 (per-tool technical report).

5.8 Network Automation Workbench

Repository: `ank_workbench/` **Language:** Python (FastAPI) + React/TypeScript **Status:** v4.2 (UX Polish shipped), 34K LOC, 430 tests

The Workbench is the orchestration platform integrating all ecosystem tools into a single workflow. It provides a YAML editor with live validation, drag-and-drop topology editing (including drag-and-drop link creation), workflow stepper with real-time WebSocket progress, interactive device terminal (xterm.js → netsim daemon), large-scale visualization (Sigma.js WebGL), command palette (Cmd+K), toast notifications, and RBAC with strict typing.

Architecture: Backend: FastAPI with API routers, Workflow Coordinator (async state machine), Validation Engine (27 rules across 6 categories), tool adapters (subprocess → netsim/netvis), FIB bridge for traffic analysis, real-time EventBus with WebSocket bridge, SQLite persistence (WAL mode). Workflow state machine: CREATED → VALIDATING → SIMULATING → VISUALIZING → COMPLETE (or FAILED/CANCELLED). A v5.0 designer split is planned to separate the topology designer into a standalone component.

Tool integration is file-based subprocess invocation: write topology YAML to temp file, invoke binary, parse JSON/file output, clean up. The FIB bridge (v4.2) enables netsim → netflowsim traffic analysis within the Workbench workflow.

Integration Points

Produces:	Orchestrated workflows, Web UI, API
Consumes:	All tool outputs via subprocess; topology YAML, simulation results, visualizations
Contracts:	RFC-02 (manifest consumer and orchestrator)

Cross-reference: ANK-WB-TR-001 (per-tool technical report).

5.9 Device Interaction Framework (deviceinteraction)

Repository: `deviceinteraction/` **Language:** Rust **Status:** v1.1 (Complete Testing Stack milestone)

deviceinteraction is a PyATS/Genie-inspired testing and validation framework that closes the loop between design and operational reality. It connects to live network devices via SSH/Telnet/REST, executes test actions (triggers), parses CLI output into structured data, and validates device state against expected design specifications.

Architecture: Three layers: Core (testbed YAML, connection pooling via tokio, command execution, events), Parser (CLI output parsers, vendor-specific implementations, OS-agnostic data models), Harness (triggers, verifications with fluent assertion API, test macros). Four crates: `di-core`, `di-parsers`, `di-harness`, `di-cli`. Data-driven testing supports YAML test definitions for common scenarios. The v1.1 milestone adds a code-first verification framework with composable logic and typed results, plus a user-facing CLI tool for interactive and one-shot commands.

Integration Points

Produces:	Operational state → OperationalTopology (RFC-01), test results, pass/fail metrics
Consumes:	Expected state from autonetkit-config, routing baselines from netsim, testbed YAML
Contracts:	RFC-01 (OperationalTopology producer)

Cross-reference: No per-tool TR exists yet.

5.10 Configuration Parsing Framework (configparsing)

Repository: `configparsing/` **Language:** Python **Status:** v1.0 shipped, v2.0 in progress

configparsing is an LLM/RAG-powered framework that decouples network configuration from vendor-specific syntax. It uses retrieval-augmented generation to extract network-level intent and topology relationships from vendor documentation and CLI configurations, normalizing them into a vendor-neutral topology graph model aligned with autonetkit.

Architecture: The framework uses a RAG pipeline: vendor CLI configurations and documentation are chunked and embedded, then retrieved at extraction time to provide context for LLM-based parsing. The output is a normalized topology graph with protocol adjacencies, link roles, and VLAN membership—enabling cross-vendor configuration generation and validation through semantic simulation.

The v1.0 milestone established the core extraction pipeline. The v2.0 milestone targets production-grade operation: broader vendor coverage (Juniper, Nokia, F5, Palo Alto), additional protocols (MPLS, SR, SRv6), higher extraction accuracy with reduced human-in-the-loop burden, ecosystem integration (RFC-01 compliance, autonetkit interop), and production hardening (observability, batch processing).

Design Decision: LLM/RAG vs. Template Parsing

Traditional config parsers (TextFSM, TTP, PyATS Genie) rely on hand-written templates per vendor per command. This approach scales linearly with vendor count and breaks on firmware updates. configparsing uses LLM extraction with RAG context, trading compute cost for maintenance cost. The LLM generalizes across vendor syntax variations; the RAG pipeline grounds extraction in vendor documentation to reduce hallucination.

Integration Points

Produces:	Vendor-neutral topology graphs, normalized protocol configurations, OperationalTopology snapshots
Consumes:	Vendor CLI configurations, vendor documentation, device output (from deviceinteraction)
Contracts:	RFC-01 (OperationalTopology producer)

Cross-reference: No per-tool TR exists yet.

6 Intelligence Layer: netassure

netassure transforms the ecosystem from deterministic simulation into intelligent prediction and formal verification. Unlike autonetkit-analysis (which handles lightweight validation: config parsing, drift detection, 31 design rules), netassure provides computationally intensive analysis across five paradigms.

This section covers the architecture in depth; per-use-case algorithmic detail is deferred to a future netassure technical report.

6.1 Why a Standalone Tool?

The original plan (INTELLIGENCE-LAYER.md) proposed embedding GNN capabilities into `autonetkit-analysis[ml]` as an optional extra. ADR-0005 rejected this approach for five reasons:

1. **Conceptual mismatch:** autonetkit-analysis handles deterministic tasks; ML/GNN predictions are probabilistic.
2. **Dependency isolation:** ML stack (PyTorch, PyTorch Geometric, MLflow) adds ~2.5 GB vs. autonetkit-analysis's ~200 MB.
3. **Persona conflict:** Config parsing targets network operators; model training targets ML engineers.
4. **Evolution speed:** ML advances rapidly; config parsing is stable. Different release cadences.
5. **Expanded scope:** GNN-only focus missed formal verification (Z3), graph algorithms (rustworkx), and cascade modeling (percolation theory).

netassure is conceptually a sibling to netsim and netflowsim—a compute engine that takes topology as input and produces insights as output.

6.2 The Five Analysis Paradigms

Figure 3 illustrates the five paradigms as layers of increasing analytical sophistication.

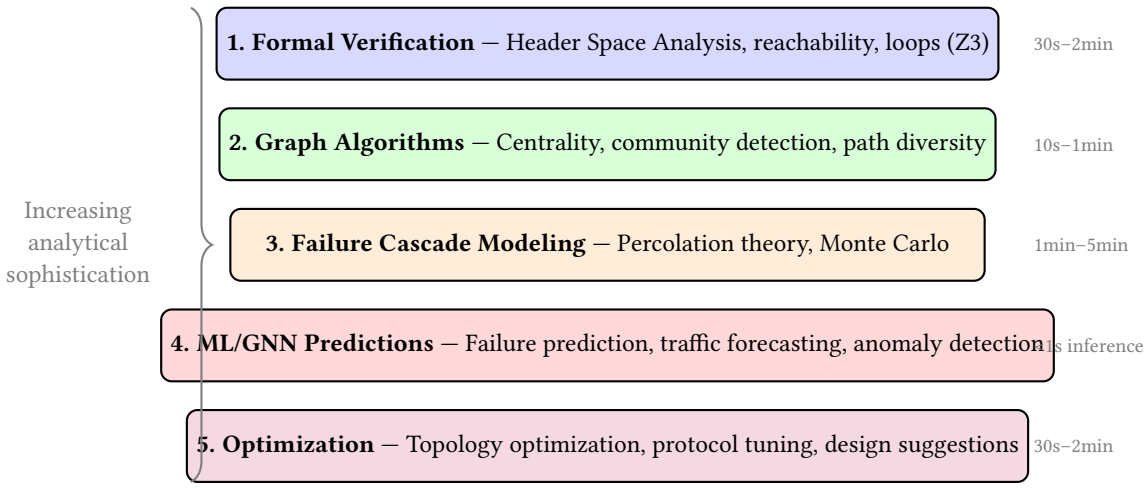

Figure 3. The five analysis paradigms of netassure, with typical compute times.

6.2.1 Paradigm 1: Formal Verification

Exhaustive guarantees about network behavior using Header Space Analysis (modeling all possible packets as geometric regions) and SMT solving (Z3). Capabilities include reachability proofs ("can VLAN 10 ever reach VLAN 20?"), forwarding loop detection, traffic isolation verification (VRF/VPN leak detection), and configuration equivalence checking. Complexity is $O(N \times R^2)$ where N is the number of rules and R is the rule space; typical compute time is 30 seconds to 2 minutes.

6.2.2 Paradigm 2: Graph Algorithms

Structural analysis of the topology graph using centrality measures (betweenness, closeness, eigenvector), community detection (Louvain/Leiden), path diversity (k-disjoint paths, MPLS TE), algebraic connectivity (Fiedler eigenvalue as a robustness metric), and min-cut analysis. Implemented in Rust using petgraph and rustworkx. Complexity ranges from $O(V \times E)$ to $O(V^2)$.

6.2.3 Paradigm 3: Failure Cascade Modeling

Probabilistic risk assessment using percolation theory (fragmentation threshold under random/targeted failure), Monte Carlo cascade simulation (expected impact of k-failure scenarios), load redistribution modeling (does link-1 failure overload link-2?), and network-of-networks interdependency analysis. The Monte Carlo engine is parallelized with Rayon. Typical compute time is 1–5 minutes for 1,000 iterations.

6.2.4 Paradigm 4: ML/GNN Predictions

Learned predictions from Graph Neural Networks trained on simulation data. Eight use cases are defined:

1. **Failure prediction:** Node classification (24h horizon)
2. **Traffic forecasting:** Edge regression (1h ahead)
3. **Anomaly detection:** Multi-class node classification
4. **Design optimization:** Graph generation
5. **Config validation:** Graph classification (safe/unsafe)
6. **Root cause analysis:** Node attribution (blame scores)
7. **Protocol tuning:** Reinforcement learning (policy gradient)
8. **Intent-to-topology:** Conditional graph generation

Training uses PyTorch Geometric with models registered in MLflow. Inference is sub-second per prediction. The training data pipeline leverages netsim: run 10K+ simulation scenarios with varied topologies, failures, and traffic, then extract labeled feature vectors.

6.2.5 Paradigm 5: Optimization

Design improvement using heuristic search and multi-objective optimization. Capabilities include topology optimization (where to add links for maximum resilience), protocol parameter tuning (OSPF costs, BGP local-preference via reinforcement learning), and AI-driven design suggestions. Optimization objectives can be combined: redundancy vs. cost vs. latency.

6.3 Multi-Source Analysis

The key innovation of netassure is operating on three distinct data sources, enabling cross-validation and comprehensive assurance (fig. 4).

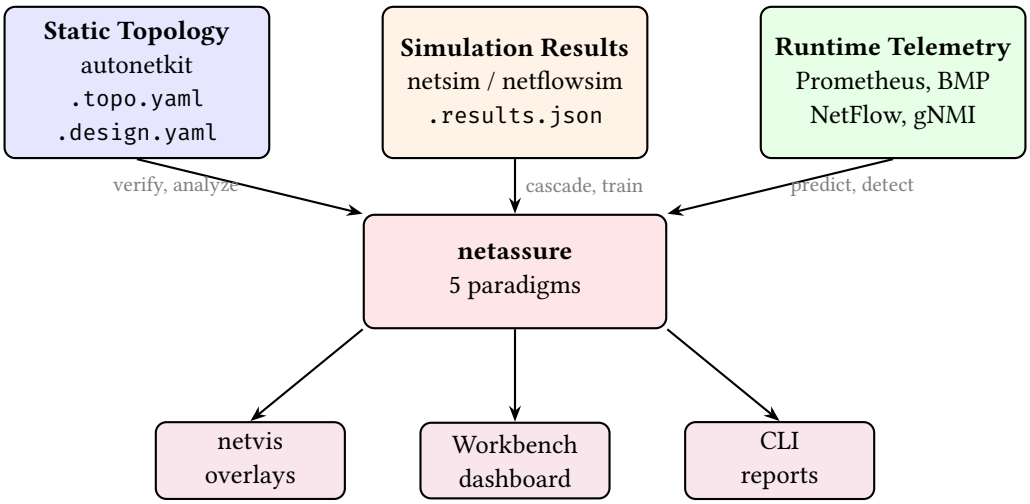

Figure 4. Multi-source analysis data flow: three data sources feed netassure, which produces insights for three consumer types.

Source 1: Static Topology (Design Intent). Inputs are `.topo.yaml` and `.design.yaml` from autonetkit. Analyses include formal verification ("Is this design correct?"), graph structure ("Which routers are critical?"), and design optimization ("Should we add this link?"). Philosophy: assure the design *before* deployment.

Source 2: Simulation Results (Behavioral Analysis). Inputs are `.results.json` from netsim and netflowsim. Analyses include failure cascades ("How does the network fail?"), protocol behavior ("What is the convergence pattern?"), and GNN training data generation (label 10K scenarios). Philosophy: understand how the design *behaves*.

Source 3: Runtime Telemetry (Predictive Operations). Inputs are time-series from Prometheus, BMP collectors, NetFlow, and gNMI. Analyses include GNN predictions ("What will fail in 24h?"), statistical anomaly detection, and design drift detection. Philosophy: predict what will *happen next*.

6.4 Technology Stack

- **Rust core:** petgraph, rustworkx (graph algorithms), z3-rs (SMT solver), rayon (parallelism). Crates: `netassure-verify`, `netassure-analyze`, `netassure-cascade`.
- **Python layer (via PyO3):** PyTorch + PyTorch Geometric (GNN training/inference), MLflow (model registry), Prometheus client (telemetry ingestion), scikit-learn (classical ML baselines).
- **CLI:** Five subcommands (`verify`, `analyze`, `cascade`, `predict`, `optimize`) for Unix-style pipeline integration.
- **Deployment:** Embedded library (MVP) → Microservice (FastAPI/gRPC, production scale).

6.5 GNN Training Pipeline

The training pipeline leverages the ecosystem's unique advantage: unlimited labeled simulation data. The flow is:

1. **Scenario generation:** topogen creates varied topologies (different sizes, structures, vendors).
2. **Simulation:** netsim runs each topology through failure scenarios, recording telemetry (BMP, NetFlow) and outcomes (convergence time, reachability loss, FIB state).
3. **Feature extraction:** Convert topology structure (node/edge features) and telemetry time-series into PyTorch Geometric `Data` objects.
4. **Training:** Train GNN models (GraphSAGE, GAT, GIN) with task-specific heads (classification, regression, generation).
5. **Registration:** Store trained models in MLflow with version tracking, performance metrics, and lineage.
6. **Inference:** Deploy models for sub-second prediction on live telemetry or new topologies.

This "train on simulation, predict on production" approach is unique in the network analysis space—competitors rely on hand-labeled production data, which is scarce, noisy, and privacy-constrained.

7 End-to-End Workflows

The ecosystem supports four distinct end-to-end workflows plus a fifth closed-loop assurance workflow enabled by netassure. Each workflow describes how tools connect, what data passes between them, and where gaps remain.

7.1 Workflow 1: Rapid Prototyping

"Generate a topology, simulate it, see if it works."

The fast inner loop for testing network designs: generate a structure, validate that routing converges, visualize the result.

$$\text{topogen} \xrightarrow{\text{topology YAML}} \text{netsim} \xrightarrow{\text{results JSON}} \text{netvis}$$

Status: Operational. topogen exports directly to netsim format (Phase 25). The gap—no `topogen export -format netsim` path—is closed.

Users: Network engineers testing "does this design converge?" before committing to container labs.

7.2 Workflow 2: Design-to-Deploy

"Model declaratively, generate vendor configs, validate, deploy."

autonetkit's primary workflow: whiteboard → plan → design layers → vendor compilers (Jinja2) → ContainerLab/Ansible deployment. netsim validates before deployment via `export_netsim()`.

Gap: One-way data flow. Simulation results don't propagate back to the design. If simulation reveals a problem (SPOF, slow convergence), there is no automated feedback mechanism. Engineers must manually diagnose and fix.

Target state: Simulation results annotate the autonetkit topology model. Analysis rules consume annotations and suggest design improvements (research-level work, uniquely differentiated).

Users: Network automation engineers building production configs.

7.3 Workflow 3: Traffic Engineering

"Analyze capacity and performance at scale."

Generate or import a topology with traffic demands, compute forwarding paths, run Monte Carlo simulation across millions of flow iterations, identify bottlenecks.

Two entry points:

- topogen → netflowsim (via GeoJSON + traffic CSV) — works today
- netsim → netflowsim (via FIB export) — planned but not implemented

Gap: netsim FIB export to netflowsim format. This is the bridge between "protocol simulation proves convergence" and "traffic analysis proves capacity."

Users: Capacity planners, traffic engineers evaluating "what happens if traffic doubles?"

7.4 Workflow 4: Interactive Exploration

"Single interface to edit, simulate, explore, and visualize."

The Workbench provides a unified web UI: YAML editor → validation (27 rules) → simulation (netsim subprocess) → visualization (netvis subprocess) → interactive terminal (netsim daemon gRPC).

Gap: The Workbench treats each tool as a black box. It validates YAML structure but cannot suggest protocol configurations, detect design anti-patterns, or offer topology-aware assistance.

Target state: The Workbench consumes NTE or autonetkit as a library and can reason about topology structure. Alternatively, the ecosystem MCP server provides the intelligence layer.

Users: Anyone wanting the full workflow without switching between CLI tools.

7.5 Workflow 5: Closed-Loop Assurance (Future)

"Design, deploy, monitor, predict, remediate."

This workflow extends Workflow 2 with netassure and deviceinteraction:

$$\text{autonetkit} \xrightarrow{\text{design}} \text{netsim} \xrightarrow{\text{validate}} \text{deploy} \xrightarrow[\text{autonetkit}]{\text{deviceinteraction}} \text{operational state} \xrightarrow{\text{netassure}} \text{predictions/drift} \xrightarrow{\text{feedback}}$$

netassure monitors runtime telemetry, predicts failures, detects drift from design intent, and feeds recommendations back to the design layer. deviceinteraction validates that deployed devices match design specifications. This closed loop is the long-term vision; it requires all three data sources (section 6.3) operating simultaneously.

8 Competitive Positioning

The ecosystem's strategy is "Complement, Don't Compete": provide the high-performance logic (simulation, design, analysis) that other tools lack, while integrating seamlessly with established workflows.

8.1 Landscape

Table 9. Competitive landscape and ecosystem positioning.

Category	Tools	Ecosystem Role
Sources of Truth	NetBox, Nautobot, Infrahub	"Active Intelligence Layer" — ingest SoT data, run design logic, push validated designs back
Lab Platforms	ContainerLab, CML, GNS3	"Scenario Generator" — produce fully-formed lab topologies with configs
Verification	Batfish, SuzieQ, Forward Networks	"Multi-Paradigm Analysis" — formal verification + simulation + ML prediction (lifecycle-spanning)
Orchestration	Ansible, Terraform, Nornir	"Structured Data Engine" — handle the thinking; Ansible handles the moving
Monitoring	Telegraf, Prometheus, gNMI	"Intelligent Contextual Visualizer" — topology-aware dashboards with predictive overlays

8.2 Unique Differentiators

Three capabilities that competitors cannot easily replicate:

1. **Simulation feedback loop.** Generate topology, simulate protocols, analyze traffic, discover issues, iterate. This requires owning both the generation and simulation layers.
2. **Deterministic reproducibility.** Seed-based output across generation, simulation, and visualization enables CI/CD regression testing of network designs. No competitor offers this.
3. **Vertical data contracts.** Topology metadata (type, tier, role, region, pod) flows from topogen through autonetkit to simulation without manual annotation. This "metadata flows downstream" property enables automation that format conversion cannot.

8.3 The Market Gap

The gap in the market is: "I want to go from a high-level intent ('ISP backbone with 5 POPs, OSPF, redundant core') to a validated, visualized, traffic-analyzed network design in minutes." No single tool or two-tool combination does this today. The ecosystem's integrated pipeline fills this gap.

8.4 Multi-Domain Stitching

The ecosystem supports ingesting and reconciling topology from multiple heterogeneous sources into a unified RFC-01 OperationalTopology view. Capabilities include identity resolution across sources (NetBox, CLI scrape, cloud APIs), conflict detection with audit trails, provenance tracking, and confidence scoring for fuzzy matches. Supported sources: NetBox/Nautobot (pynetbox), CLI scrape (8 vendors), cloud APIs (AWS, Azure, GCP), and fabric controllers (Cisco ACI, VMware NSX).

9 Architecture Decision Records

Five ADRs capture the ecosystem's key architectural decisions. Each follows the standard format: Context, Decision, Consequences, Alternatives.

9.1 ADR-0001: Contract-First Architecture

Decision: Adopt RFC-01 layered artifacts as the primary interoperability boundary and RFC-02 `netauto.project` as the orchestration boundary. Tools remain modular; orchestration and UX are built around the contracts.

Consequence: Requires schema validation and compliance testing. Encourages sidecar outputs. Enables parity across CLI/TUI/Web/agent entry points.

9.2 ADR-0002: Interface Representation (Deferred)

Decision: Defer a breaking structural change to NTE's graph model. In the near term, add device-centric APIs and views; benchmark with large topologies; prototype dual-view if benchmarks show need.

Consequence: Improves ergonomics immediately while collecting evidence. RFC-03 captures the detailed decision with migration path and trigger conditions.

9.3 ADR-0003: Shared Orchestration Layer

Decision: Treat orchestration as a shared layer (not Workbench-owned). Implement provider resolution, artifact dependency tracking, and tool runners in a shared library. Workbench consumes the library.

Consequence: CLI, TUI, Workbench, and AI agents share the same orchestration semantics. Prevents second-class entry points.

9.4 ADR-0004: netvis Scope (Renderer, Not Editor)

Decision: Define netvis scope as rendering, overlays, and export. Keep topology authoring and orchestration outside netvis. Editing belongs to the Workbench or CLI/TUI.

Consequence: Prevents netvis from becoming a monolith. Keeps clear ownership boundaries aligned with RFC-01 (netvis consumes artifacts, does not author them).

9.5 ADR-0005: netassure as Standalone Engine

Decision: Create netassure as a standalone tool (#9 in the ecosystem), not embedded in autonetkit-analysis. Five paradigms: formal verification, graph algorithms, cascade modeling, ML/GNN, optimization.

Consequence: Isolates 2.5 GB ML dependencies from 200 MB parsing stack. Enables independent evolution. Fills the architectural gap between validation and simulation.

10 Limitations and Known Gaps

This section documents known limitations honestly—both technical constraints and integration gaps.

10.1 Integration Gaps

1. **No simulation feedback loop (Workflow 2).** Simulation results do not propagate back to the design layer. The "detect → suggest → fix" loop is entirely manual.
2. **FIB bridge (Workflow 3).** The Workbench v4.2 FIB bridge enables netsim → netflowsim conversion within the orchestrated workflow. Direct CLI-level FIB export (without the Workbench) is not yet available.
3. **Workbench is a black box orchestrator (Workflow 4).** The Workbench validates YAML structure but cannot reason about topology semantics.

10.2 Maturity Imbalance

The most mature tools (netsim, netvis) are standalone. The tools that enable the most valuable cross-tool workflows (NTE, netflowsim) are the least mature. NTE has a known dual-write data integrity issue (petgraph + DataFrameStore can desync without rollback). This is the highest-risk item in the ecosystem.

10.3 Technical Limitations

- **NTE dual-write (mitigated):** The dual-write architecture (petgraph + DataFrameStore) has been hardened with rollback safety. Desync risk is substantially reduced but not formally eliminated.
- **netflowsim under-investment:** No competitive analysis against WAE/MATE, no research documentation. Solid queuing model foundation but thin integration story.
- **netassure is architecture-only:** All five paradigms are designed but implementation has not begun. Roadmap targets Phase 14–18.
- **deviceinteraction is early:** Core crate implemented with verification framework; CLI tool in planning (Phase 8).
- **Scale:** NTE has not been validated beyond 100K-device synthetic benchmarks. Production networks may exceed this.

10.4 Documentation Gaps

- No per-tool technical report for autonetkit, netflowsim, netassure, configparsing, or deviceinteraction.
- User persona analysis is thin—architecture mentions "network engineers, researchers, internal tools" without detailed needs analysis.
- No formal schema versioning/migration policy for RFC-01/RFC-02.

11 Roadmap and Future Work

11.1 Milestone History

- **v1.0 (shipped 2026-02-21):** Initial architecture definition—architecture spine, requirements traceability, RFC contracts, fixture projects (Phases 1–6).
- **v1.1 (shipped 2026-02-28):** Architecture evolution—OQ-02 investigation, sub-project deep-dives, intelligence layer, netassure decomposition (Phases 7–12).
- **v2.0 (shipped 2026-03-01):** Advanced capabilities—topology management, verification tooling, intelligence layer extensions, multi-domain interoperability, visualization at scale (Phases 13–17).
- **v3.0 (in progress):** Implementation and developer enablement—API documentation, code examples, onboarding (Phase 18+).

11.2 Near-Term Priorities

1. **v3.0 implementation enablement:** API documentation, code examples, and onboarding guides for all tools (in progress).
2. **configparsing v2.0:** Broader vendor coverage, higher accuracy, RFC-01 compliance, and autonetkit interop.
3. **netvis scale:** Barnes-Hut $O(n \log n)$ force repulsion and interactive HTML export (v1.9 milestone).
4. **deviceinteraction CLI:** User-facing tool for interactive and batch device testing (Phase 8).

11.3 Medium-Term Priorities

1. **netassure Phase 14–16:** Formal verification, graph algorithms, and cascade modeling (Q2–Q4 2026).
2. **Live-Hook visualization:** WebSocket ingestion for real-time netvis updates.
3. **Workbench topology awareness:** Consume NTE/autonetkit as a library for intelligent design assistance.

11.4 Long-Term Vision

1. **netassure Phase 17–18:** GNN integration and optimization (Q1–Q2 2027).
2. **Closed-loop assurance:** Full Workflow 5 with design → deploy → monitor → predict → remediate.
3. **Platform convergence:** Full `netauto.project` manifest integration across all entry points.
4. **AI-native orchestration:** MCP-driven autonomous control loops using netassure predictions.

References

[1] S. Knight, "RFC-01: Canonical Data Contract v2.0," Network Automation Ecosystem, Feb. 2026. `RFC-01.md`

[2] S. Knight, "RFC-02: Project Manifest (netauto.project)," Network Automation Ecosystem, Feb. 2026. `RFC-02.md`

[3] S. Knight, "RFC-03: Interface Representation in the Topology Engine (OQ-02)," Network Automation Ecosystem, Feb. 2026. `RFC-03.md`

[4] S. Knight, "netsim: Convergence-First Network Simulator — Technical Reference (NETSIM-TR-001)," Mar. 2026.

[5] S. Knight, "Network Automation Ecosystem — Architecture Overview," `README.md`, Mar. 2026.

[6] S. Knight, "Ecosystem Interoperability Strategy," `ECOSYSTEM_INTEROP.md`, Feb. 2026.

[7] S. Knight, "Network Automation Ecosystem — Critical Review," `REVIEW.md`, Feb. 2026.

[8] S. Knight, "Network Automation Ecosystem — Data Flow Map," `DATAFLOWS.md`, Feb. 2026.

[9] S. Knight, "Network Automation Ecosystem: Strategic Roadmap," `STRATEGY.md`, Feb. 2026.

[10] S. Knight, "netassure: Advanced Network Analysis Engine — Architectural Decomposition," `ANK-INTEL-DECOMPOSITION.md`, Feb. 2026.

[11] S. Knight, "GNN Use Cases Architecture," `use-cases-architecture.md`, Feb. 2026.

A Schema Reference

Table 10 lists all versioned schemas and contracts in the ecosystem.

Table 10. Versioned schemas and contracts.

Contract ID	Version	Location
`netauto/design/v2.0`	2.0	RFC-01 §2.2
`netauto/operational-topology/v1.0`	1.0	`rfc/rfc-01/.../schema.json`
`netauto/live-overlay-stream/v1.0`	1.0	`rfc/rfc-02/.../schema.json`
`netauto/project/v2.0`	2.0	RFC-02 §2

B Technical Report Cross-Reference

Table 11 maps each tool to its per-tool technical report, where available.

Table 11. Per-tool technical reports.

Tool	TR Number	Status	Pages
NTE	NTE-TR-001	Published	—
netsim	NETSIM-TR-001	Published	45+
netvis	NETVIS-TR-001	Published	—
topogen	TOPOGEN-TR-001	Published	—
Workbench	ANK-WB-TR-001	Published	—
ank-netcfg	NETCFG-TR-001	Published	—
autonetkit	—	Not started	—
netflowsim	—	Not started	—
netassure	—	Not started	—
configparsing	—	Not started	—
deviceinteraction	—	Not started	—
Ecosystem	**NETAUTO-TR-001**	**This document**	**~50**

C Ecosystem Overview Diagram

Figure 5 shows the full ecosystem with all ten tools arranged in architectural layers.

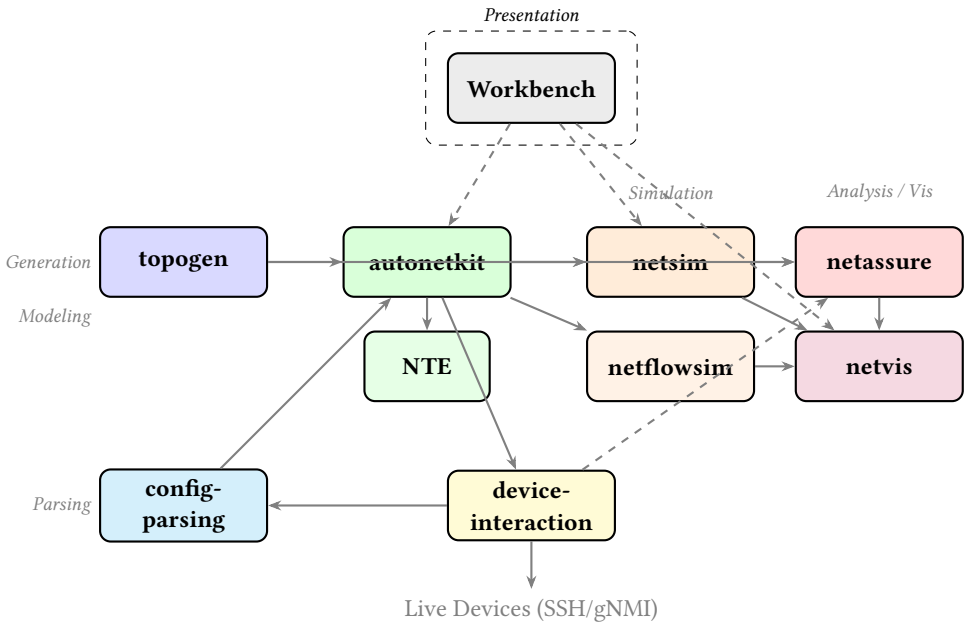

Figure 5. Ecosystem overview: ten tools in architectural layers with data flow arrows. Solid arrows are implemented; dashed arrows are planned.