TopoGen

Seed-Deterministic Network Topology Generator — Technical Reference

Simon Knight

Independent Researcher, Adelaide, Australia

March 2026 • Version 1.0

Abstract. TopoGen is a seed-deterministic network topology generator implemented in Rust that produces structurally correct, vendor-aware, geographically realistic topologies across datacenter, WAN, and random-graph domains. This technical report serves as the definitive design rationale and reference document for TopoGen. It explains the architectural decisions behind the system, documents all nine generators and their configuration options, describes the composable validation pipeline, and provides a complete API reference for both the CLI and Python interfaces. The report includes worked examples of increasing complexity, from single-command datacenter fabric generation to multi-domain topology composition with traffic matrices and policy-based validation.

Version	Changes
1.0 (March 2026)	Initial release

Contents

1 Introduction

1.1 What TopoGen Is

TopoGen is a network topology generator that produces annotated graph structures for use in network testing, simulation, capacity planning, and lab automation. It generates topologies with structural correctness (fat-tree pod invariants, WAN tier connectivity), operational metadata (vendor-specific interface names, bandwidth, latency), and geographic realism (real city coordinates, fiber-distance-derived latency). All generation is seed-deterministic: the same seed produces bit-identical output across platforms, languages, and invocations.

TopoGen is implemented in Rust (~15,000 lines) and exposes three API surfaces: a CLI (`clap`), a Python module (PyO3/maturin), and a REST API (axum). It exports topologies in six formats: YAML, JSON, ContainerLab, AutoNetKit, Parquet, and GeoJSON.

1.2 Design Philosophy

Two principles guide every design decision in TopoGen:

1. **Structure-aware generation with composable post-hoc validation.** Generators are fast, stateless, and domain-specific. They know how to build a fat-tree or a hierarchical WAN, but they do not enforce policy constraints. Validators are independent predicates composed via conjunction. This separation means adding a new validation rule never requires modifying a generator, and generators can be freely combined via composition operators before validation runs.
2. **Seed determinism as a first-class property.** All randomness flows through a single `ChaCha8Rng` instance initialized from a user-supplied seed. No system entropy, no floating-point non-determinism, no platform-dependent behavior. This makes topologies reproducible across CI pipelines, team members, and Rust/Python/REST API surfaces.

1.3 Who Should Read This

This document is intended for:

- Engineers integrating TopoGen into network automation pipelines
- Researchers using TopoGen to generate topologies for simulation studies
- Contributors to the TopoGen codebase who need to understand design rationale

Familiarity with Python and basic graph theory is assumed. Prior experience with network topology concepts (fat-tree, leaf-spine, WAN tiers) is helpful but not required—each concept is introduced where first used.

1.4 How to Read This Document

- **Quick start:** Read §2 (Installation), §13 (Usage by Example), and §12 (API Reference).
- **Understanding the design:** Read §3 (Data Model), §4 (Architecture), then individual subsystem sections as needed.
- **Contributing:** Start with §4, then §5 for generator patterns.

2 Installation

2.1 Python Package

```
pip install topogen
```

Requires Python 3.9+. The package includes pre-built wheels for macOS (aarch64, x86_64) and Linux (x86_64, aarch64). PyYAML ≥6.0 is the only runtime dependency.

2.2 CLI Binary

```
cargo install topogen
```

Requires Rust 1.75+ (edition 2021). The binary is a single static executable with no runtime dependencies.

2.3 From Source

```
git clone https://github.com/sk2/ank-topogen.git
cd ank-topogen
cargo build --release          # CLI binary
maturin develop --release      # Python package (dev install)
```

3 Data Model

TopoGen models topologies as undirected graphs with rich node and edge annotations. This section defines the data model formally.

3.1 Core Types

Design Decision: Undirected Graphs

Problem: Should topologies be directed or undirected?
Options: (1) Directed graphs model asymmetric links but double the edge count for symmetric links. (2) Undirected graphs halve edge count for the common case of bidirectional links.
Decision: Undirected. Most physical network links are bidirectional. Traffic matrices model directional flows separately.
Consequence: Unidirectional links (e.g., monitoring taps) require workarounds using edge properties.

A topology is stored as a `petgraph::StableGraph<Node, Edge, Undirected>`. `StableGraph` provides O(1) node/edge access by index and index-stability across removals—critical for composition operators that merge graphs while maintaining references.

Node. Each node carries:

Field	Type	Description
`id`	`String`	Unique identifier within the topology
`label`	`String`	Human-readable display name
`node_type`	`NodeType`	One of: `Router`, `Switch`, `Host`, `IXP`, `Custom(String)`
`interfaces`	`Vec<Interface>`	Ordered list of network interfaces
`properties`	`HashMap<String,String>`	Extensible key-value metadata
`site`	`Option<String>`	Hierarchical containment: site
`pod`	`Option<String>`	Hierarchical containment: pod
`zone`	`Option<String>`	Hierarchical containment: zone
`asn`	`Option<u32>`	Autonomous System Number
`igp_area`	`Option<u32>`	IGP (OSPF/IS-IS) area identifier
`overlays`	`Vec<String>`	Overlay network memberships

Table 1. Node fields.

Interface. Each interface carries:

Field	Type	Description
`name`	`String`	Interface name (e.g., `Ethernet1/1`)
`bandwidth_gbps`	`Option<f64>`	Link capacity in Gbps
`latency_us`	`Option<f64>`	One-way propagation delay in μs
`jitter_us`	`Option<f64>`	Delay variation in μs
`packet_loss_pct`	`Option<f64>`	Packet loss percentage (0–100)
`properties`	`HashMap<String,String>`	Extensible metadata

Table 2. Interface fields.

Edge. Each edge carries source/target node IDs, source/target interface names, optional bandwidth, latency, jitter, packet loss, access technology, SRLG identifier, relationship type, overlay memberships, and an extensible property map.

3.2 Type System

`NodeType` is an enum with four fixed variants and one extensible variant:

```
pub enum NodeType {
    Router,    // L3 routing device
    Switch,    // L2 switching device
    Host,      // End-host / compute node
    Ixp,       // Internet Exchange Point
    Custom(String),  // User-defined type
}
```

The `Custom` variant allows users to model arbitrary node types (load balancers, firewalls, controllers) without modifying TopoGen's source code.

4 Architecture

TopoGen is organized into five layers. Each layer has a well-defined interface to adjacent layers; data flows downward from user interfaces through generators, validation, composition, and export.

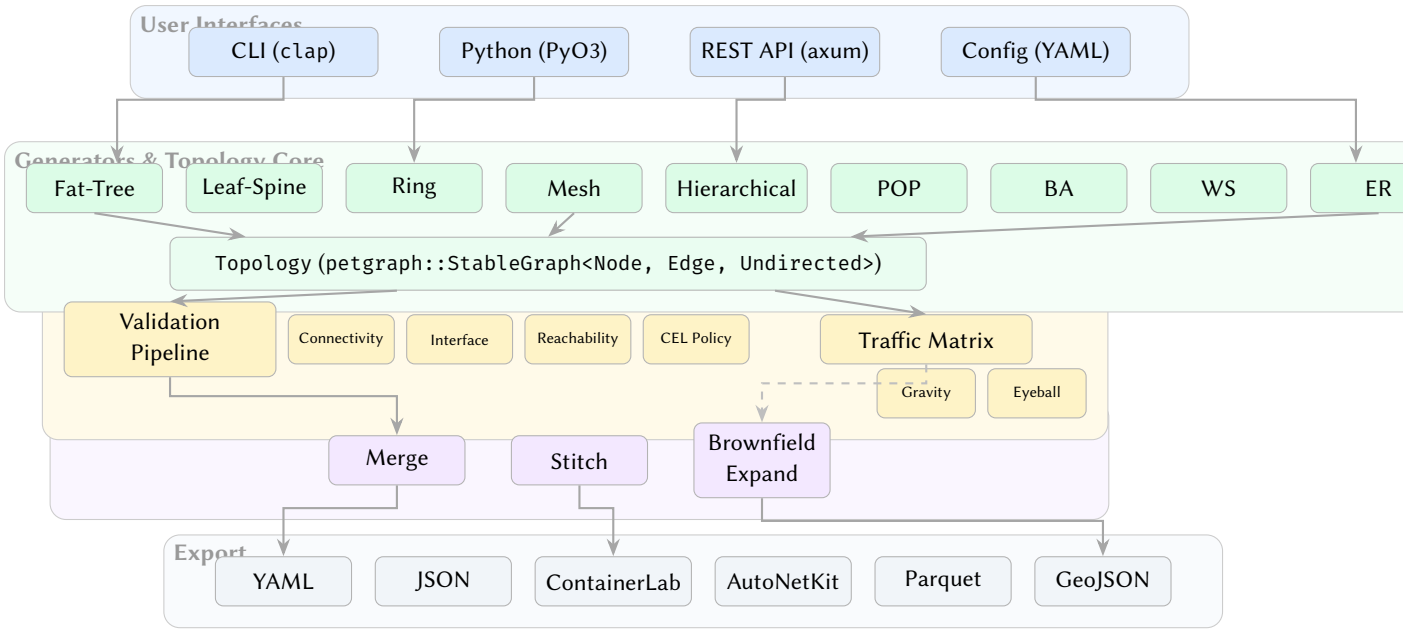

Figure 1. TopoGen system architecture. Blue: user interfaces. Green: generators and topology core. Amber: validation and traffic. Purple: composition operators. Grey: export formats.

4.1 Layer 1: User Interfaces

> **Design Decision: Three API Surfaces from One Codebase**
>
> **Problem:** How to serve CLI users, Python data scientists, and service integrations without maintaining three codebases?
> **Options:** (1) Separate implementations per surface. (2) Code generation from an IDL. (3) Thin wrappers over a shared Rust core.
> **Decision:** Thin wrappers. The CLI uses `clap` to parse arguments and call Rust functions directly. The Python module uses PyO3's `#[pyfunction]` to wrap the same functions. The REST API uses axum handlers that deserialize JSON into the same config types.
> **Consequence:** Near-zero API divergence, but Python/REST wrappers must manually convert between Rust types and Python/JSON types. PyO3's `#[pyclass]` and `#[pymethods]` provide getter/setter generation that keeps this manageable.

CLI. The CLI uses `clap` with derive macros for subcommand-driven operation:

```
topogen generate fat-tree -k 8 --seed 42 --vendor cisco -o dc.yaml
topogen generate hierarchical-wan --node-count 20 --seed 4242
topogen validate dc.yaml
topogen merge dc.yaml wan.yaml -o combined.yaml
topogen traffic combined.yaml --model gravity --total-gbps 100
topogen export combined.yaml --format containerlab -o lab.clab.yml
```

Python. The Python module exposes generator functions with keyword arguments:

```
import topogen

# Generate a fat-tree topology
topo = topogen.generate_fat_tree(
    name="dc-east", k=8, seed=42, vendor="cisco"
```

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

# Generate traffic matrix
tm = topogen.generate_traffic_matrix(
    topo, total_traffic_gbps=100.0, seed=42
)
print(f"Total traffic: {tm.total_traffic_gbps} Gbps")
```

REST API. The axum-based REST API accepts JSON POST requests:

```
curl -X POST http://localhost:3000/generate/fat-tree \
  -H "Content-Type: application/json" \
  -d '{"name": "dc-east", "k": 8, "seed": 42}'
```

4.2 Layer 2: Generator Registry

All generators implement the `TopologyGenerator` trait:

```
pub trait TopologyGenerator {
    type Config;
    fn generate(&self, base: &GeneratorConfig,
                config: &Self::Config) -> Result<Topology>;
    fn validators(&self) -> ValidationPipeline;
    fn generate_validated(&self, base: &GeneratorConfig,
                          config: &Self::Config) -> Result<Topology>;
}
```

Design Decision: Stateless Generators

Problem: Should generators maintain state across calls?
Decision: No. Generators are stateless; all inputs come via `GeneratorConfig` (shared) and `Self::Config` (generator-specific).
Rationale: Statelessness makes generators thread-safe, testable in isolation, and trivially parallelizable. Combined with seed determinism, it means the same inputs always produce the same output.

The `GeneratorRegistry` provides runtime discovery:

```
pub struct GeneratorConfig {
    pub seed: Option<u64>,      // ChaCha8Rng seed
    pub name: String,           // Topology name
}
```

4.3 Layer 3: Validation and Traffic

The validation pipeline and traffic matrix generators operate independently on the produced topology. Both are optional. See §6 and §8 for detailed treatment.

4.4 Layer 4: Composition

Merge, stitch, and brownfield expansion operators combine topologies. See §9.

4.5 Layer 5: Export

Six export formats serve different downstream consumers:

Format	Module	Use case
YAML	`export::writer`	Human-readable, version-controllable topology files
JSON	`export::writer`	Programmatic consumption, API responses
ContainerLab	`export::containerlab`	Direct deployment to ContainerLab network labs
AutoNetKit	`export::autonetkit`	Legacy AutoNetKit toolchain compatibility
Parquet	`export::parquet`	Columnar analytics via pandas, Polars, DuckDB
GeoJSON	`geo::geojson_io`	Geographic visualization in mapping tools

Table 3. Export formats and their use cases.

5 Generators

TopoGen provides nine generators across three domains.

5.1 Datacenter Generators

5.1.1 Fat-Tree

A k-ary fat-tree consists of:

- $(k/2)^2$ core switches
- k pods, each with $k/2$ aggregation and $k/2$ edge switches
- Optionally $k/2$ hosts per edge switch
- Total switches: $5k^2/4$, total hosts: $k^3/4$

Example: Fat-tree generation

CLI:

```
topogen generate fat-tree -k 4 --seed 42 --vendor cisco
```

Python:

```
topo = topogen.generate_fat_tree(
    name="dc", k=4, seed=42, vendor="cisco"
)
# Produces: 20 switches (4 core, 8 agg, 8 edge)
```

Configuration options:

Parameter	Type	Description	Default
`k`	`u32`	Fat-tree radix (must be even, ≥ 4)	Required
`seed`	`u64`	RNG seed	Random
`vendor`	`String`	Naming convention: `cisco`, `arista`, `juniper`	None
`hosts`	`bool`	Generate host nodes	`false`
`core_bw`	`f64`	Core link bandwidth (Gbps)	100.0
`agg_bw`	`f64`	Aggregation link bandwidth (Gbps)	40.0
`edge_bw`	`f64`	Edge link bandwidth (Gbps)	10.0

Table 4. Fat-tree configuration parameters.

5.1.2 Leaf-Spine

A 2-tier Clos fabric with L leaf switches and S spine switches, fully meshed between tiers ($L \times S$ inter-tier links).

Example: Leaf-spine generation

```
topo = topogen.generate_leaf_spine(
    name="spine", leaf_count=8, spine_count=4, seed=42
)
# Produces: 12 switches, 32 links
```

5.2 WAN Generators

5.2.1 Ring

Bidirectional ring of n nodes. With geographic mode, nodes are placed at cities and latencies derived from haversine distances.

Example: Geographic ring

```
topo = topogen.generate_ring(
    name="backbone", node_count=6, seed=42,
    region="NA"  # Place at North American cities
)
```

5.2.2 Full Mesh

Complete graph K_n with $n(n-1)/2$ edges. Used for core WAN backbones.

5.2.3 Hierarchical WAN

Multi-tier WAN topology reflecting real ISP structure. The tier assignment algorithm:

- $n \leq 2$: All nodes Tier-1 (core), fully meshed
- $3 \leq n \leq 6$: Top $\lceil n/3 \rceil$ nodes Tier-1, rest Tier-2
- $n > 6$: Up to 5 Tier-1, up to $n/2$ Tier-2, rest Tier-3

Connectivity rules depend on the `redundancy` parameter:

Redundancy Level	Tier-1 mesh	T2→T1 uplinks	T3→T2 uplinks
`minimal`	Full	1	1
`standard`	Full	2	1
`high`	Full	≥ 3	2

Table 5. Hierarchical WAN redundancy levels.

Example: Hierarchical WAN

```
topo = topogen.generate_hierarchical_wan(
    name="isp", node_count=20, seed=4242,
    redundancy="standard", region="EU"
)
```

5.2.4 POP (Point of Presence)

Generates PoP internal structure: core router pair with edge router fan-out.

5.2.5 Eyeball

ISP access network model: external content sources connect to subscriber sinks with configurable inbound-to-outbound ratio.

5.3 Random Graph Generators

5.3.1 Barabási-Albert

Preferential attachment model. Configuration: initial clique size m_0, attachment count m, total node count n.

```
topo = topogen.generate_barabasi_albert(
    name="ba", node_count=1000, m=3, seed=42
)
print(topo.graph_properties)
# GraphProperties(average_clustering=..., average_path_length=...)
```

5.3.2 Watts-Strogatz

Small-world model: ring lattice with k nearest-neighbor connections and rewiring probability β.

5.3.3 Erdős-Rényi

$G(n, p)$ model: n nodes with independent edge probability p.

5.4 Seed Determinism

Note

All generators use `ChaCha8Rng` from the `rand_chacha` crate. ChaCha8 produces identical byte streams across all platforms (x86_64, aarch64, WASM). Unlike Mersenne Twister, it has no platform-dependent floating-point behavior.
The same seed passed to the CLI, Python, or REST API produces byte-identical YAML output. This is tested in CI across macOS and Linux.

6 Validation Pipeline

6.1 Design Rationale

Design Decision: Independent Validators

Problem: How to validate topologies against diverse constraints (connectivity, structure, bandwidth, policy) without coupling validators to generators?
Options: (1) Validate inside each generator. (2) A monolithic validation function. (3) Independent validator predicates composed via pipeline.
Decision: Option 3. Each validator is a pure function V_i : Topology $\to$ Result. The pipeline composes them: Pipeline$(T) = \bigwedge_i V_i(T)$.
Rationale: Adding a new validator (e.g., a custom CEL policy) never requires touching existing validators or generators. Validators can be run selectively. Auto-detection can infer the right set from topology metadata.
Consequence: Cross-validator constraints (e.g., "bandwidth must be sufficient *given* the traffic matrix") require a validator that takes both topology and traffic matrix as input. Currently handled by the `traffic_fit` validator.

6.2 Built-in Validators

Validator	What it checks	Complexity
`Connectivity`	Single connected component	$O(V+E)$
`InterfaceConsistency`	Valid interface indices, non-negative values	$O(E)$
`Reachability`	Source-dest paths exist (optional hop bounds)	$O(V^2)$
`FatTreeStructure`	Pod count, tier connectivity invariants	$O(V+E)$
`Bandwidth`	Link capacities meet minimums	$O(E)$
`Redundancy`	Critical nodes have redundant paths	$O(V \cdot E)$
`CELPolicy`	User-defined CEL expression constraints	$O(V+E)$
`Cycles`	Detects or validates cycle presence	$O(V+E)$
`NodeCapacity`	Node degree/capacity limits	$O(V)$
`TrafficFit`	Traffic matrix fits within link capacities	$O(E)$

Table 6. Built-in validators.

6.3 Auto-Detection

When validators are not specified explicitly, TopoGen inspects the topology to select an appropriate set:

1. `Connectivity` and `InterfaceConsistency` always run.
2. If the topology has a `topology_type` property matching "fat-tree", add `FatTreeStructure`.
3. If edges have `bandwidth_gbps` annotations, add `Bandwidth`.
4. If nodes have geographic coordinates, add latency consistency checks.
5. If CEL policy annotations are present, add `CELPolicy`.

6.4 CEL Policy Expressions

The Common Expression Language (CEL) integration enables user-defined structural constraints without modifying TopoGen's source:

```
# All core nodes must have degree >= 4
# All edge nodes must have at least one uplink to aggregation tier
```

CEL expressions are evaluated per-node or per-edge, with the full node/edge property map available as context variables.

6.5 Usage

Example: Validation

CLI:

```
topogen validate topology.yaml                # auto-detect validators
topogen validate topology.yaml --validators connectivity,bandwidth
```

Python:

```
topo = topogen.generate_fat_tree(name="dc", k=8, seed=42,
                                 validate=True, auto_detect=True)
# Validation runs automatically; raises ValidationError on failure
```

7 Geographic Realism

7.1 City Database

TopoGen includes a curated database of ~150 cities organized by continent (NA, EU, AP, SA, AF, ME) and tier:

- **Tier-1:** Major internet hubs (New York, London, Frankfurt, Tokyo, Singapore). Up to 5 per continent.
- **Tier-2:** Regional hubs (Dallas, Madrid, Mumbai).
- **Tier-3:** Access/edge cities.

Warning

North American Tier-1 cities max out at 5. Do not request 6 or more Tier-1 nodes in the NA region—the generator will reuse cities or error depending on the generator.

Fuzzy matching. City names are matched using Levenshtein distance with a configurable threshold. "New York", "NewYork", and "NYC" all resolve to the same entry.

7.2 Latency Model

$$\text{latency}_{\text{ms}} = \frac{d_{\text{haversine}}(\text{src}, \text{dst}) \times r_{\text{fiber}}}{c_{\text{fiber}}} \times 10^3 \tag{1}$$

where:

- $d_{\text{haversine}}$: Great-circle distance (km) using WGS-84 radius
- $r_{\text{fiber}} = 1.4$ (terrestrial) or 1.6 (submarine): fiber-to-crow ratio
- $c_{\text{fiber}} = 2 \times 10^5$ km/s: speed of light in fiber ($\approx 2/3\, c$)

The constant `MS_PER_100KM = 0.5` provides a convenient shorthand: multiply fiber-km by 0.005 to get one-way latency in ms.

7.3 SRLG Inference

Shared Risk Link Groups (SRLGs) identify links that share physical infrastructure and may fail simultaneously. TopoGen infers SRLGs by:

1. Computing GeoJSON LineStrings for each link's geographic path.
2. Discretizing paths into H3 resolution-5 hexagons (~250 km^2).
3. Reporting links whose paths share any H3 cell as belonging to the same SRLG.

7.4 Fiber Map Import

TopoGen can import real fiber route data from GeoJSON files, overlaying actual cable paths onto generated topologies for more accurate latency computation.

```
fiber_map = topogen.import_fiber_geojson(geojson_str)
topo = topogen.generate_hierarchical_wan(
    name="isp", node_count=15, seed=42
)
geojson = topogen.export_geojson(topo, fiber_map=fiber_map)
```

8 Traffic Matrix Generation

8.1 Gravity Model

Design Decision: Gravity vs. Tomographic Models

Problem: How to generate realistic traffic matrices without real measurement data?
Options: (1) Uniform random (unrealistic). (2) Gravity model (captures spatial decay). (3) Tomographic inference (requires real link loads).
Decision: Gravity model as default, with eyeball model for ISP access networks. Tomographic inference is out of scope since TopoGen generates synthetic topologies without real link load data.
Consequence: Generated traffic matrices approximate real distributions but cannot capture application-specific patterns (e.g., CDN cache hit rates). This is documented as a known limitation.

The gravity model generates flows:

$$f(i,j) = C \times \frac{\text{out}(i) \times \text{in}(j)}{d(i,j)^{\alpha}} \tag{2}$$

Parameter	Type	Description	Default
`total_traffic_gbps`	`f64`	Target total traffic	100.0
`distance_exponent`	`f64`	Distance decay α	2.0
`include_self_traffic`	`bool`	Include intra-node flows	`false`
`asymmetry_ratio`	`f64`	Inbound/outbound ratio	1.0
`seed`	`u64`	RNG seed	Random

Table 7. Gravity model configuration.

Example: Traffic matrix generation

```
topo = topogen.generate_hierarchical_wan(
    name="isp", node_count=20, seed=42
```

```
)
tm = topogen.generate_traffic_matrix(
    topo, total_traffic_gbps=500.0,
    distance_exponent=1.5, seed=42
)
print(f"Total: {tm.total_traffic_gbps} Gbps")
print(f"Flows: {tm.flow_count}")
```

8.2 Eyeball Model

The eyeball model targets ISP access networks:

```
tm = topogen.generate_eyeball_traffic_matrix(
    topo,
    total_traffic_gbps=110.0,
    inbound_to_outbound=10.0,  # 10:1 download bias
    measurement="access",
    distribution="uniform"
)
```

8.3 QoS Auto-Inference

Traffic flows are automatically tagged with QoS classes:

- **CS6**: Control-plane traffic (core↔core)
- **EF**: Low-latency flows (short geographic distance)
- **AF**: Standard inter-site traffic
- **BE**: Default class

8.4 Temporal Traffic Series

TopoGen can generate time-varying traffic with diurnal and weekly patterns:

```
series = topogen.generate_traffic_time_series(
    baseline=tm,
    duration_hours=168,        # one week
    granularity_hours=1.0,
    diurnal_amplitude=0.3,     # +/- 30% daily swing
    weekly_amplitude=0.2       # +/- 20% weekly swing
)
```

9 Composition Operators

9.1 Merge

Combines two disjoint topologies with automatic node ID deduplication:

```
topogen merge dc.yaml wan.yaml -o combined.yaml --prefix-b wan-
```

If node IDs collide, the second topology's IDs are prefixed (default: topology name).

9.2 Stitch

Connects two topologies at designated boundary nodes, creating peering links:

```
stitch:
  connections:
    - source_node: dc-core-0
      target_node: wan-tier1-nyc
      bandwidth_gbps: 100
```

9.3 Brownfield Expansion

Augments an existing topology with new capacity while preserving interface numbering and prefix assignment continuity.

Example: End-to-end composition

```
# Generate components
dc = topogen.generate_fat_tree(name="dc", k=8, seed=42)
wan = topogen.generate_hierarchical_wan(
    name="wan", node_count=15, seed=4242
)

# Merge and validate
combined = topogen.merge_topologies(dc, wan)
# Stitch at boundary nodes
# Validate the composite
```

10 Graph Properties

TopoGen computes standard graph-theoretic properties for any generated topology:

```
topo = topogen.generate_barabasi_albert(
    name="ba", node_count=1000, m=3, seed=42
)
props = topo.graph_properties
print(f"Clustering coefficient: {props.average_clustering}")
print(f"Average path length: {props.average_path_length}")
print(f"Degree range: {props.min_degree} - {props.max_degree}")
print(f"Mean degree: {props.mean_degree}")
```

Warning

Python attribute names differ from what you might guess:

- `average_clustering` (not `avg_clustering_coefficient`)
- `average_path_length` (not `avg_path_length`)
- `min_degree`, `max_degree`, `mean_degree` (not `min`/`max`/`mean`)

Also: `TrafficMatrix` properties are attributes, not methods: `tm.total_traffic_gbps` not `tm.total_traffic_gbps()`.

11 Multi-Layer Topologies

TopoGen supports multi-layer topology generation for modeling optical transport, IP, and service layers. Layers are defined declaratively, with explicit inter-layer node mappings and link realization strategies.

```
ml_topo = topogen.generate_multi_layer(
    name="layered",
    layers=[
        {"name": "optical", "generator": "ring", "node_count": 6},
        {"name": "ip", "generator": "mesh", "node_count": 4},
    ],
    seed=42
)
```

12 Python API Reference

12.1 Generator Functions

Function	Description
`generate_fat_tree()`	Fat-tree DC fabric. Params: `name`, `k`, `seed`, `vendor`, `hosts`, bandwidth params
`generate_leaf_spine()`	Leaf-spine Clos. Params: `name`, `leaf_count`, `spine_count`, `seed`
`generate_ring()`	Bidirectional ring. Params: `name`, `node_count`, `seed`, `region`
`generate_mesh()`	Full mesh. Params: `name`, `node_count`, `seed`, `region`
`generate_hierarchical_wan()`	Multi-tier WAN. Params: `name`, `node_count`, `seed`, `redundancy`, `region`
`generate_pop()`	PoP internal structure. Params: `name`, `seed`, redundancy strategy
`generate_eyeball()`	ISP access network. Params: `name`, subscriber config
`generate_barabasi_albert()`	BA random graph. Params: `name`, `node_count`, `m`, `seed`
`generate_watts_strogatz()`	WS small-world. Params: `name`, `node_count`, `k`, `beta`, `seed`
`generate_erdos_renyi()`	ER random graph. Params: `name`, `node_count`, `p`, `seed`
`generate_multi_layer()`	Multi-layer topology. Params: `name`, `layers`, `seed`

12.2 Traffic Functions

Function	Description
`generate_traffic_matrix()`	Gravity model. Params: `topology`, `total_traffic_gbps`, `distance_exponent`, `seed`
`generate_eyeball_traffic_matrix()`	Eyeball model. Params: `topology`, `total_traffic_gbps`, `inbound_to_outbound`
`generate_traffic_time_series()`	Temporal series. Params: `baseline`, `duration_hours`, `granularity_hours`

12.3 Utility Functions

Function	Description
`create_topology()`	Create an empty topology by name
`create_node()`	Create a standalone node
`create_interface()`	Create a standalone interface
`haversine_distance_km()`	Great-circle distance between two coordinates
`calculate_latency_ms()`	Estimated fiber latency from distance
`import_fiber_geojson()`	Import fiber route data from GeoJSON
`export_geojson()`	Export topology as GeoJSON
`discover_roles()`	Infer node roles from topology structure
`infer_geometric_srlgs()`	Detect shared risk link groups
`interface_spec_json()`	Get the vendor interface specification

12.4 Classes

Class	Key attributes / methods
`Topology`	`.node_count`, `.edge_count`, `.nodes()`, `.edges()`, `.graph_properties`, `.to_yaml()`, `.add_node()`, `.add_edge()`
`Node`	`.id`, `.label`, `.properties`, `.site`, `.pod`, `.zone`, `.asn`, `.overlays`, `.add_interface()`, `.get_interfaces()`
`Edge`	`.source_id`, `.target_id`, `.bandwidth_gbps`, `.latency_us`, `.srlg`
`Interface`	`.name`, `.bandwidth_gbps`, `.latency_us`
`NodeType`	Static methods: `.router()`, `.switch()`, `.host()`, `.ixp()`, `.custom(name)`
`GraphProperties`	`.average_clustering`, `.average_path_length`, `.min_degree`, `.max_degree`, `.mean_degree`
`TrafficMatrix`	`.total_traffic_gbps` (attribute), `.flow_count`, `.flows()`
`CityDatabase`	`.lookup(name)`, `.cities_in_region(region)`

12.5 Error Handling

TopoGen defines three Python exception types, all inheriting from `TopoGenError`:

- `InvalidParameterError`: Invalid configuration values (e.g., odd k for fat-tree, negative bandwidth)
- `GenerationError`: Topology generation failures (e.g., insufficient cities for requested node count)
- `ValidationError`: Topology fails validation constraints

```python
try:
    topo = topogen.generate_fat_tree(name="dc", k=3, seed=42)
except topogen.InvalidParameterError as e:
    print(f"Bad config: {e}")  # k must be even
```

```
except topogen.GenerationError as e:
    print(f"Generation failed: {e}")
```

13 Usage by Example

Four examples of increasing complexity.

13.1 Example 1: Simple DC Fabric

"How do I get a testable DC topology in one command?"

```
topogen generate fat-tree -k 4 --seed 42 --vendor cisco -o dc.yaml
```

This produces a 20-switch fat-tree with Cisco NX-OS interface naming (`Ethernet1/1`, `Ethernet1/2`, etc.), saved as YAML.

13.2 Example 2: Multi-Region WAN with Realistic Latencies

"How do I get realistic inter-region latencies?"

```
topo = topogen.generate_hierarchical_wan(
    name="global-wan",
    node_count=20,
    seed=4242,
    redundancy="standard",
    region="NA"  # North American cities
)

# Check latencies
for edge in topo.edges():
    if edge.latency_us:
        print(f"{edge.source_id} -> {edge.target_id}: "
              f"{edge.latency_us / 1000:.1f} ms")
```

Latencies are derived from haversine distances between real city coordinates, multiplied by the fiber-to-crow ratio of 1.4.

13.3 Example 3: Composed DC + WAN with Validation

"How do I test end-to-end reachability across DC and WAN?"

```
# Generate components
dc = topogen.generate_fat_tree(
    name="dc-east", k=8, seed=42, vendor="arista"
)
wan = topogen.generate_hierarchical_wan(
    name="wan-backbone", node_count=15, seed=4242
)

# Merge topologies
combined = topogen.merge_topologies(dc, wan)

# Validate composite
# - Connectivity: is the merged graph connected?
# - Reachability: can edge switches reach WAN tier-1?
print(f"Combined: {combined.node_count} nodes, "
      f"{combined.edge_count} edges")
```

13.4 Example 4: Traffic Matrix with QoS Validation

"How do I verify capacity under realistic load?"

```
# Generate topology + traffic
topo = topogen.generate_hierarchical_wan(
    name="isp", node_count=20, seed=42
)
tm = topogen.generate_traffic_matrix(
    topo,
    total_traffic_gbps=500.0,
    distance_exponent=1.5,
    seed=42
)

# Inspect traffic
print(f"Total traffic: {tm.total_traffic_gbps} Gbps")
print(f"Number of flows: {tm.flow_count}")

# Generate time-varying traffic for weekly simulation
series = topogen.generate_traffic_time_series(
    baseline=tm,
    duration_hours=168,
    granularity_hours=1.0,
    diurnal_amplitude=0.3,
    weekly_amplitude=0.2
)
```

14 Limitations

14.1 Static Topology Snapshots Only

Description: TopoGen generates static topology snapshots. It does not model dynamic topology changes (link failures, node additions), routing protocol convergence, or time-varying topology state.

Workaround: Generate multiple snapshots with different configurations to simulate before/after states. Use composition operators to model capacity expansion.

Planned resolution: Temporal topology evolution (planned maintenance windows, failure injection) is under investigation.

14.2 Synthetic Traffic Matrices

Description: Traffic matrices use gravity and eyeball models, not learned from real traces. They approximate real distributions but cannot capture application-specific patterns (CDN cache hit rates, microservice call graphs).

Workaround: Use the gravity model as a baseline and manually adjust specific flows for known traffic patterns.

Planned resolution: Future work on traffic matrix learning from SNMP/NetFlow data.

14.3 Limited City Database

Description: The geographic city database contains ~150 cities. This covers all major internet hubs but may lack smaller cities relevant to specific regional deployments.

Workaround: Use `Custom` coordinates for cities not in the database.

Planned resolution: No plan to grow beyond ~200 cities. The database is intentionally curated, not comprehensive.

14.4 Three Vendor Naming Schemes

Description: Interface naming supports Cisco NX-OS, Arista EOS, and Juniper conventions. Other vendors (Nokia, Huawei, etc.) are not supported.

Workaround: Use generic interface names and post-process with vendor-specific tooling.

Planned resolution: Additional vendor schemes may be added based on community contributions.

14.5 No Routing Protocol Simulation

Description: TopoGen produces topology and traffic, not forwarding state. It does not simulate BGP, OSPF, IS-IS, or any routing protocol.

Workaround: Feed TopoGen output into Batfish, Mininet, or ContainerLab for routing simulation.

Planned resolution: Out of scope by design. TopoGen generates inputs for simulation tools, not simulation itself.

14.6 No Formal Verification

Description: Validation is heuristic and structural, not formally verified. The validators check necessary conditions (connectivity, interface consistency) but do not prove sufficiency for all possible operational requirements.

Workaround: Combine TopoGen's structural validation with domain-specific verification tools (Batfish for config correctness, Minesweeper for reachability proofs).

Planned resolution: No current plan for formal verification integration.

15 Future Work

- **Temporal topology evolution:** Model planned maintenance windows, link failure/recovery sequences, and capacity expansion timelines as topology diffs.
- **Learned traffic matrices:** Infer traffic patterns from real SNMP/NetFlow data to augment or replace the gravity model.
- **Routing simulation integration:** Generate BGP/OSPF configurations alongside topologies for direct deployment into simulation engines.
- **Additional vendor support:** Nokia SR OS, Huawei VRP interface naming conventions.
- **Topology diffing:** Compare two topologies to identify structural changes (added/removed nodes, changed links, modified properties).

A CLI Quick Reference

Command	Description
`topogen generate fat-tree`	Generate fat-tree DC fabric
`topogen generate leaf-spine`	Generate leaf-spine Clos
`topogen generate ring`	Generate bidirectional ring
`topogen generate mesh`	Generate full mesh
`topogen generate hierarchical-wan`	Generate multi-tier WAN
`topogen generate pop`	Generate PoP structure
`topogen generate eyeball`	Generate ISP access network
`topogen generate barabasi-albert`	Generate BA random graph
`topogen generate watts-strogatz`	Generate WS small-world
`topogen generate erdos-renyi`	Generate ER random graph
`topogen validate <file>`	Run validation pipeline
`topogen merge <a> <b>`	Merge two topologies
`topogen traffic <file>`	Generate traffic matrix
`topogen export <file>`	Export to specified format
`topogen geo <file>`	Geographic analysis
`topogen serve`	Start REST API server
`topogen completions <shell>`	Generate shell completions
`topogen diagnostics`	Show system diagnostics

Common flags (all generators):

Flag	Type	Description
`--seed <N>`	`u64`	Deterministic RNG seed
`--name <S>`	`String`	Topology name
`-o <file>`	`Path`	Output file (default: stdout)
`--format <F>`	`String`	Output format: yaml, json, containerlab
`--vendor <V>`	`String`	Interface naming: cisco, arista, juniper
`--validate`	flag	Run validation after generation
`--auto-detect`	flag	Auto-detect validators