# TopoGen User Manual

## Seed-Deterministic Topology Generator

Simon Knight

Adelaide, Australia

March 2026

> **About this manual.** This manual covers generating datacenter, WAN, and random-graph topologies with deterministic seeds, geographic realism, traffic matrices, and validation. It describes all ten built-in generators, common flags, export formats, geographic features, and integration with AutoNetKit, ContainerLab, NetVis, and NetCfg. Manual ID: TOPOGEN-UM-001.

# Contents

# 1 Quick Start

Get a deterministic datacenter topology and a WAN in under three minutes.

**Step 1.** Install TopoGen (PyPI or Cargo):

```
pip install topogen          # Python library + bundled CLI
# -- or, for a standalone binary --
cargo install topogen
```

**Step 2.** Generate a $k$=4 fat-tree with a fixed seed:

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

TopoGen writes `dc.yaml` containing 20 nodes (4 core, 8 aggregate, 8 edge) and 32 links. The same command always produces byte-identical output for the same `--seed` value.

**Step 3.** Generate a 20-node hierarchical WAN for the North America region:

```
topogen generate hierarchical-wan \
    --node-count 20 --seed 4242 --region NA -o wan.yaml
```

**Step 4.** Validate the generated topology:

```
topogen validate dc.yaml
```

A summary of connectivity, bandwidth, and structural checks is printed to stdout. Exit code 0 indicates all checks passed.

**Step 5.** Export to ContainerLab format for lab instantiation:

```
topogen export dc.yaml --format containerlab
```

This writes `dc.clab.yaml` ready for `containerlab deploy`.

# 2 Installation & Prerequisites

▷ **Prerequisites**

- **Python 3.9 or later** with `pip` (for the Python package)
- **PyYAML** (`pip install pyyaml`) — used for YAML input/output
- **Rust 1.70+** toolchain (for `cargo install` only)
- (Optional) **H3-py** (`pip install h3`) — required for SRLG inference from geographic data
- (Optional) **Shapely** — required for GeoJSON polygon export

## 2.1 Installing the Python Package

The Python package provides both the `topogen` CLI and the importable `topogen` library:

```
pip install topogen
```

**Pre-built wheels**

PyPI distributes pre-built wheels for Linux (x86_64, aarch64), macOS (x86_64, arm64), and Windows (x86_64). The Rust core is compiled into the wheel via PyO3, so no local Rust toolchain is required for normal installation. Use `pip install --only-binary :all: topogen` to enforce wheel-only installation and fail fast if no wheel is available for your platform.

### 2.2 Installing the Standalone Binary

For environments where Python is unavailable or undesirable, build a native binary with Cargo:

```
cargo install topogen
```

The binary is placed in `$HOME/.cargo/bin/topogen`. Ensure that directory is on your `PATH`.

### 2.3 Verifying the Installation

```
topogen --version
```

Expected output: `topogen 1.0.0`. If the command is not found, check that the install location is on your `PATH`.

**Fat-tree degree constraint**

The fat-tree generator requires that $k$ is an **even integer** and $k \geq 4$. Odd values or $k < 4$ are rejected at parse time with an error message indicating the violated constraint. The number of nodes scales as $\frac{5k^2}{4}$, so choose $k$ carefully for large deployments.

## 3 Core Concepts

### 3.1 Seed Determinism

Every TopoGen generator accepts a `--seed` flag that initialises a ChaCha8 pseudo-random number generator (PRNG) before any topology decisions are made. Given the same seed and the same generator arguments, TopoGen always produces *byte-identical* output. This guarantee holds across platforms and across minor releases within the same major version.

Seed determinism enables:

- **Reproducible tests** — CI pipelines can assert exact topology properties without fragile golden-file management.
- **Shared scenarios** — teams can exchange a single integer to describe a topology rather than committing large YAML files.
- **Sweep studies** — iterate over a range of seeds to sample the distribution of a random-graph model without sacrificing reproducibility.

**Choosing seeds**

Use well-known seeds (`0`, `42`, `1234`) for shared reference topologies. Use random seeds (from `/dev/urandom` or `secrets.randbelow(2**32)`) for Monte Carlo sweeps, and record the seed alongside your results.

### 3.2 Topology Model

TopoGen represents topologies as an attributed graph stored in YAML or JSON. The top-level structure is:

```yaml
topology:
  name: dc-fat-tree-k4
  generator: fat-tree
  seed: 42
  nodes:
    - id: core-0
      role: core
      vendor: cisco
      platform: nexus9k
      # ...
  links:
    - src: core-0
      dst: agg-0
      capacity_gbps: 100
      latency_us: 1
      # ...
  metadata:
    generated_at: "2026-03-06T12:00:00Z"
    topogen_version: "1.0.0"
```

All fields are stable across patch releases. Minor releases may add optional fields; major releases may restructure the schema with a migration command.

### 3.3 Vendor Naming

The `--vendor` flag controls the naming scheme applied to nodes and platforms. Supported values are `cisco`, `juniper`, `arista`, `nokia`, and `generic`. The vendor flag affects node identifiers, platform strings, and the set of interface names generated. For example, `cisco` uses `GigabitEthernet0/0/0` naming, while `juniper` uses `et-0/0/0`.

## 4 Core Workflows

### 4.1 Datacenter Fabric Generation

#### 4.1.1 Fat-Tree

Fat-tree is the canonical multi-rooted datacenter topology. For a given degree $k$, TopoGen generates $\frac{k^2}{4}$ core switches, $\frac{k^2}{2}$ aggregate switches (grouped into $k/2$ pods), and $\frac{k^2}{2}$ edge (top-of-rack) switches.

*$k = 4$: 20 nodes; $k = 8$: 80 nodes; $k = 16$: 320 nodes.*

**Step 1.** Choose the degree $k$ (even, $\geq 4$) and a seed.

**Step 2.** Generate the topology:

```
topogen generate fat-tree \
    -k 8 --seed 100 --vendor juniper \
    --name dc-fat-tree --validate -o dc.yaml
```

**Step 3.** Inspect the structural summary:

```
topogen show dc.yaml
```

The summary reports node counts by role, link counts, bisection bandwidth, and oversubscription ratio.

**Step 4.** (Optional) Attach a traffic matrix:

```
topogen traffic gravity dc.yaml -o dc-with-tm.yaml
```

Key parameters for `fat-tree`:

| Flag | Description |
|---|---|
| `-k <int>` | Pod degree (even, ≥ 4; default: 4) |
| `--vendor <name>` | Node naming vendor: `cisco`, `juniper`, `arista`, `nokia`, `generic` |
| `--capacity <gbps>` | Per-link capacity in Gbps (default: 100) |
| `--oversubscribe` | Allow edge-to-aggregate oversubscription ratio > 1 |
| `--ecmp` | Annotate links with ECMP group membership |

#### 4.1.2 Leaf-Spine

Leaf-spine is a simpler two-tier fabric. All leaf switches connect to all spine switches, giving a non-blocking bisection bandwidth.

**Step 1.** Decide on the number of spine and leaf switches.

**Step 2.** Generate:

```
topogen generate leaf-spine \
    --spine-count 4 --leaf-count 16 \
    --seed 200 --vendor arista -o ls.yaml
```

**Step 3.** Validate and inspect:

```
topogen validate ls.yaml
topogen show ls.yaml --format table
```

> **Leaf-spine vs. fat-tree**
>
> Use leaf-spine when the number of leaf switches is fixed by rack count. Use fat-tree when you need a well-studied bisection bandwidth guarantee and a canonical benchmark topology.

### 4.2 WAN Generation

#### 4.2.1 Hierarchical WAN

The `hierarchical-wan` generator builds a three-tier WAN with backbone (tier-1), regional hub (tier-2), and access (tier-3) nodes. Nodes are placed in real cities drawn from TopoGen's built-in city database; link latencies are computed from the haversine great-circle distance plus a fiber-ratio correction.

*Use `--region` to constrain placement to a continent.*

**Step 1.** Choose the node count, seed, and region.

**Step 2.** Generate:

```
topogen generate hierarchical-wan \
    --node-count 30 --seed 4242 \
    --region NA --redundancy 2 \
    --vendor cisco -o wan.yaml
```

`--redundancy 2` ensures at least two geographically distinct paths between any tier-1 pair.

**Step 3.** Inspect geographic placement:

```
topogen export wan.yaml --format geojson -o wan.geojson
```

Open `wan.geojson` in NetVis or QGIS to visualise city placement.

**Step 4.** Validate connectivity:

```
topogen validate wan.yaml --check reachability --check srlg
```

Key parameters for `hierarchical-wan`:

| Flag | Description |
|---|---|
| `--node-count <n>` | Total nodes across all tiers (default: 20) |
| `--region <code>` | Geographic region filter (see section 6.2) |
| `--redundancy <n>` | Minimum path redundancy between tier-1 nodes (default: 1) |
| `--tier-ratio <a:b:c>` | Tier size ratio, e.g. `1:2:4` (default: `1:3:6`) |
| `--latency-model` | `haversine` (default) or `flat` |

## 4.3 Random Graph Topologies

TopoGen includes three classical random-graph models. All share the `--seed` and `--node-count` flags.

### 4.3.1 Barabási–Albert Preferential Attachment

Generates scale-free graphs with a power-law degree distribution, modelling the Internet AS-level topology.

**Step 1.** Generate with $n = 50$ nodes, attachment parameter $m = 2$:

```
topogen generate barabasi-albert \
    --node-count 50 --attachment 2 --seed 1 -o ba.yaml
```

**Step 2.** Inspect degree distribution:

```
topogen show ba.yaml --stat degree-distribution
```

### 4.3.2 Watts–Strogatz Small-World

Generates graphs with high clustering and short average path length, modelling metropolitan or regional access networks.

```
topogen generate watts-strogatz \
    --node-count 40 --k-nearest 4 --beta 0.3 \
    --seed 77 -o ws.yaml
```

> **Tip**
>
> `--beta 0` produces a regular ring lattice; `--beta 1` produces an Erdős–Rényi random graph. Values between 0.1 and 0.4 give small-world characteristics.

### 4.3.3 Erdős–Rényi

Generates a random graph where each edge exists independently with probability $p$.

```
topogen generate erdos-renyi \
    --node-count 30 --edge-probability 0.15 \
    --seed 999 -o er.yaml
```

> **Caution**
>
> For large $n$ and moderate $p$, the graph is very likely connected, but for small $p$ (below the connectivity threshold $\ln n/n$), isolated nodes or components are possible. Always run `topogen validate --check connectivity` after generation.

## 4.4 Topology Composition

TopoGen can combine multiple topologies into a single graph using the `compose` subcommand.

### 4.4.1 Merge

Merging combines two topology files into one without adding inter-topology links:

```
topogen compose merge dc.yaml wan.yaml -o merged.yaml
```

Node IDs are namespaced with the source topology name to avoid collisions.

### 4.4.2 Stitch

Stitching connects two topologies by adding links between nominated nodes:

**Step 1.** Generate the datacenter and WAN topologies separately.

**Step 2.** Stitch the datacenter edge layer to a WAN access node:

```
topogen compose stitch dc.yaml wan.yaml \
    --dc-attach "edge-*" --wan-attach "access-*" \
    --link-capacity 10 --seed 55 -o full.yaml
```

The `--dc-attach` and `--wan-attach` flags accept glob patterns over node identifiers.

**Step 3.** Validate the stitched result:

```
topogen validate full.yaml --check reachability
```

### 4.4.3 Brownfield Expansion

To expand an existing topology with new nodes rather than generating from scratch, use `compose expand`:

```
topogen compose expand existing.yaml \
    --add-nodes 4 --role edge --seed 300 \
    -o expanded.yaml
```

New nodes are connected following the same generator rules as the existing topology (detected from the embedded `generator` metadata field).

## 4.5 Traffic Matrix Generation

TopoGen can attach a synthetic traffic matrix to any topology, encoding per-node-pair demand in Gbps.

### 4.5.1 Gravity Model

The gravity model estimates demand between node pairs proportional to the product of their "mass" (here, node degree or an explicit capacity attribute):

**Step 1.** Generate a topology and attach a gravity traffic matrix:

```
topogen traffic gravity dc.yaml \
    --total-demand 1000 --seed 42 \
    -o dc-tm.yaml
```

`--total-demand` sets the aggregate offered load in Gbps.

**Step 2.** Inspect the top-10 traffic pairs:

```
topogen show dc-tm.yaml --stat traffic-top10
```

### 4.5.2 Eyeball Model

The eyeball model concentrates demand from many access nodes toward a small number of content nodes, matching observed Internet traffic patterns:

```
topogen traffic eyeball wan.yaml \
    --content-nodes "tier1-*" \
    --eyeball-nodes "access-*" \
    --asymmetry 5.0 --seed 88 \
    -o wan-tm.yaml
```

`--asymmetry` sets the downstream-to-upstream ratio (default: 4.0).

> **Synthetic traffic limitations**
>
> Traffic matrices generated by TopoGen are synthetic approximations. They are useful for stress-testing routing algorithms and load-balancing heuristics, but should not be treated as accurate forecasts of production traffic. Calibrate against real measurements before using in capacity planning.

## 4.6 Validation

TopoGen ships four built-in validators invoked by `topogen validate`:

**connectivity** Checks that the topology graph is connected (i.e. no isolated nodes or partitions).

**reachability** Runs a BFS/DFS from every node and asserts full reachability. Detects directed-graph asymmetry when links have directional attributes.

**fat-tree-structure** Verifies pod membership, core–aggregate link regularity, and the $k$-ary property for fat-tree topologies.

**bandwidth** Checks that every link has a positive capacity and that aggregate capacity at each node meets a configurable threshold.

Run all validators at once:

```
topogen validate topology.yaml
```

Run specific validators:

```
topogen validate topology.yaml \
    --check connectivity --check bandwidth \
    --bandwidth-threshold 40
```

Exit code 0 means all selected checks passed. Exit code 1 means at least one check failed; failures are printed to stderr in structured JSON for pipeline consumption.

# 5 Generator Reference

## 5.1 Available Generators

### Datacenter Generators

| Command | Description |
|---|---|
| `fat-tree` | Multi-rooted CLOS fat-tree ($k$-ary) |
| `leaf-spine` | Two-tier leaf-spine fabric |

### WAN Generators

| Command | Description |
|---|---|
| `hierarchical-wan` | Three-tier WAN with geographic city placement |
| `ring` | Unidirectional or bidirectional ring topology |
| `hub-and-spoke` | Star topology with optional backup spokes |

### Random Graph Generators

| Command | Description |
|---|---|
| `barabasi-albert` | Preferential attachment, scale-free degree distribution |
| `watts-strogatz` | Small-world with rewiring probability $\beta$ |
| `erdos-renyi` | Classical $G(n, p)$ random graph |

### Composition Commands

| Command | Description |
|---|---|
| `compose merge` | Combine two topology files without inter-links |
| `compose stitch` | Connect two topologies with new links |
| `compose expand` | Add nodes to an existing topology |

## 5.2 Common Flags

All generators and most subcommands accept the following common flags:

### Universal Flags

| Command | Description |
|---|---|
| `--seed <uint64>` | PRNG seed for deterministic generation |
| `--name <string>` | Topology name written into metadata |
| `-o, --output <file>` | Output file path (default: stdout) |
| `--format <fmt>` | Output format: `yaml` (default), `json` |
| `--vendor <name>` | Node naming vendor scheme |
| `--validate` | Run all validators immediately after generation |
| `--quiet` | Suppress informational output; print only errors |
| `--verbose` | Print generation trace for debugging |
| `-h, --help` | Print help for the current subcommand |

### 5.3 Generator: `fat-tree`

```
topogen generate fat-tree [OPTIONS]
```

| Flag | Description |
|---|---|
| `-k <int>` | Pod degree (even, $\geq 4$; default: 4) |
| `--capacity <gbps>` | Link capacity in Gbps (default: 100) |
| `--oversubscribe` | Allow edge:agg oversubscription $> 1$ |
| `--ecmp` | Annotate links with ECMP group IDs |
| `--vendor <name>` | Naming scheme (see section 3.3) |
| `--platform <str>` | Override platform string for all nodes |

### 5.4 Generator: `hierarchical-wan`

```
topogen generate hierarchical-wan [OPTIONS]
```

| Flag | Description |
|---|---|
| `--node-count <n>` | Total nodes (default: 20) |
| `--region <code>` | Region filter: `NA`, `EU`, `APAC`, `SA`, `AF` |
| `--redundancy <n>` | Minimum redundant paths between tier-1 nodes |
| `--tier-ratio <a:b:c>` | Tier size ratio (default: `1:3:6`) |
| `--latency-model` | `haversine` (default) or `flat` |
| `--fiber-ratio <f>` | Multiplier over great-circle distance (default: 1.4) |

### 5.5 Generator: `barabasi-albert`

```
topogen generate barabasi-albert [OPTIONS]
```

| Flag | Description |
|---|---|
| `--node-count <n>` | Number of nodes (default: 20) |
| `--attachment <m>` | Edges added per new node, $m \geq 1$ (default: 2) |
| `--directed` | Generate a directed graph |

### 5.6 Generator: `watts-strogatz`

```
topogen generate watts-strogatz [OPTIONS]
```

| Flag | Description |
|---|---|
| `--node-count <n>` | Number of nodes (default: 20) |
| `--k-nearest <k>` | Each node connects to $k$ nearest neighbours on ring (default: 4) |
| `--beta <float>` | Rewiring probability $\beta \in [0, 1]$ (default: 0.2) |

### 5.7 Generator: `erdos-renyi`

```
topogen generate erdos-renyi [OPTIONS]
```

| Flag | Description |
|---|---|
| `--node-count <n>` | Number of nodes (default: 20) |
| `--edge-probability <p>` | Edge probability $p \in (0, 1]$ (default: 0.15) |
| `--connected` | Retry until result is connected (uses seed sequence) |

# 6 Geographic Features

## 6.1 City Database

TopoGen ships a curated database of approximately 150 cities, each with:

- Name and ISO country code
- Latitude and longitude (WGS84)
- Population tier (used as weight in city selection)
- Region code

Cities are selected deterministically from the database using the generator seed, so topology placement is fully reproducible. The database is embedded in the binary and does not require network access.

To list all available cities in a region:

```
topogen geo list-cities --region EU
```

To show the city assigned to a specific node:

```
topogen show wan.yaml --node-info
```

## 6.2 Region Codes

| Code | Region | Coverage |
|---|---|---|
| `NA` | North America | USA, Canada, Mexico |
| `EU` | Europe | Western and Eastern Europe, UK, Scandinavia |
| `APAC` | Asia-Pacific | East Asia, Southeast Asia, Oceania |
| `SA` | South America | Brazil, Argentina, Colombia, Chile |
| `AF` | Africa | Sub-Saharan and North Africa |
| `ME` | Middle East | Gulf states, Levant, Turkey |
| `GLOB` | Global | All cities (default when no region is specified) |

> **Insufficient cities**
>
> If `--node-count` exceeds the number of cities in the chosen region, TopoGen will raise an error: `insufficient cities in region`. Either reduce `--node-count`, choose a broader region, or allow `--allow-city-reuse` (multiple nodes co-located in the same city).

## 6.3 Latency Model

Link latencies in WAN topologies are computed from geographic coordinates using the haversine formula augmented by a fiber-ratio correction:

$$\ell = \frac{d_{\text{haversine}} \times r_{\text{fiber}}}{c}$$

where $d_{\text{haversine}}$ is the great-circle distance in kilometres, $r_{\text{fiber}}$ is the fiber-ratio multiplier (default 1.4, accounting for cable routing not following straight lines), and $c$ is the speed of light in fiber ($\approx$ 200,000 km/s). The result is expressed in microseconds.

The flat latency model assigns a uniform 1 ms per hop and ignores geography. Use it for unit tests where geographic realism is not required.

### 6.4 SRLG Inference via H3 Hexagons

Shared Risk Link Groups (SRLGs) capture the risk of correlated failures, such as links sharing a physical conduit or geographic region. TopoGen infers SRLGs automatically from geographic coordinates using the H3 hexagonal hierarchical geospatial indexing system:

**Step 1.** Ensure `h3` is installed (`pip install h3`).

**Step 2.** Generate a WAN topology with SRLG annotation:

```
topogen generate hierarchical-wan \
    --node-count 20 --seed 42 --region EU \
    --srlg --srlg-resolution 4 -o wan-srlg.yaml
```

`--srlg-resolution` sets the H3 resolution (0–15; default 4 corresponds to hexagons of roughly 200 km across).

**Step 3.** Validate SRLG diversity:

```
topogen validate wan-srlg.yaml --check srlg
```

Links sharing an H3 cell at the chosen resolution are placed into the same SRLG and annotated with a `srlg_id` field in the output YAML. The SRLG IDs are stable across runs with the same seed because they are derived from geographic coordinates, not from the PRNG.

### 6.5 GeoJSON Export

Any topology with geographic coordinates can be exported to GeoJSON for visualisation:

```
topogen export wan.yaml --format geojson -o wan.geojson
```

The GeoJSON output contains:

- `Point` features for each node, with properties including `id`, `role`, `city`, and `vendor`.
- `LineString` features for each link, with properties including `src`, `dst`, `capacity_gbps`, and `latency_us`.
- (If SRLG is present) `Polygon` features for each H3 SRLG cell.

Open the GeoJSON file directly in NetVis, QGIS, Mapbox, or any GeoJSON viewer.

## 7 Export Formats

TopoGen can export a topology to several formats using the `topogen export` subcommand:

```
topogen export <topology.yaml> --format <fmt> [-o <output>]
```

| Format | Extension | Description |
|---|---|---|
| `yaml` | `.yaml` | Native TopoGen YAML (lossless, default) |
| `json` | `.json` | Native TopoGen JSON (lossless) |
| `containerlab` | `.clab.yaml` | ContainerLab topology definition |
| `autonetkit` | `.ank.yaml` | AutoNetKit topology model |
| `parquet` | `.parquet` | Columnar format for analytics pipelines |
| `geojson` | `.geojson` | GeoJSON for geographic visualisation |
| `graphml` | `.graphml` | GraphML for graph analysis tools |

### 7.1 YAML and JSON

The native YAML and JSON formats are lossless: all TopoGen attributes, metadata, and annotations round-trip without loss. Use these formats when passing topologies between TopoGen subcommands or storing topologies in version control.

### 7.2 ContainerLab

The ContainerLab export produces a `.clab.yaml` file ready for `containerlab deploy`:

```
topogen export dc.yaml --format containerlab -o dc.clab.yaml
containerlab deploy -t dc.clab.yaml
```

Node images are selected based on the `vendor` and `platform` fields. Use `--image-map` to override the default image selection:

```
topogen export dc.yaml --format containerlab \
    --image-map cisco/nexus9k=vrnetlab/vr-n9kv:9.3.13 \
    -o dc.clab.yaml
```

### 7.3 AutoNetKit

The AutoNetKit export produces a `.ank.yaml` topology model suitable for input to the AutoNetKit IP addressing and configuration pipeline:

```
topogen export dc.yaml --format autonetkit -o dc.ank.yaml
ank build dc.ank.yaml --validate
```

**See also:** *AutoNetKit User Manual* (ANK-UM-001) — Building and validating topology models.

### 7.4 Parquet

The Parquet export writes two files: `nodes.parquet` and `links.parquet`, suitable for analytics with Pandas, Polars, or Apache Spark:

```
topogen export dc.yaml --format parquet --output-dir ./data/
```

This is useful for large-scale sweep studies where topology statistics must be aggregated across hundreds or thousands of generated topologies.

## 8 Python API

TopoGen exposes a Python API for programmatic topology generation and manipulation. The API mirrors the CLI interface with Python-idiomatic argument names.

### 8.1 Generating a Topology

```python
from topogen import generate, validate

# Generate a fat-tree with k=4, seed=42
topo = generate("fat-tree", k=4, seed=42, vendor="cisco")

# Validate connectivity and structure
results = validate(topo, checks=["connectivity", "fat-tree-structure"])
for check, passed in results.items():
    print(f"{check}: {'PASS' if passed else 'FAIL'}")
```

## 8.2 Exporting from Python

```python
from topogen import generate, export

topo = generate("hierarchical-wan", node_count=20, seed=4242, region="NA")

# Export to ContainerLab
export(topo, format="containerlab", output="wan.clab.yaml")

# Export to GeoJSON
export(topo, format="geojson", output="wan.geojson")
```

## 8.3 Composing Topologies

```python
from topogen import generate, compose

dc   = generate("fat-tree", k=4, seed=42)
wan  = generate("hierarchical-wan", node_count=20, seed=4242)

# Stitch datacenter edge nodes to WAN access nodes
full = compose.stitch(
    dc, wan,
    dc_attach="edge-*",
    wan_attach="access-*",
    link_capacity=10,
    seed=55,
)
```

**Python API stability**

The Python API follows semantic versioning. The public interface of `topogen.generate`, `topogen.validate`, `topogen.export`, and `topogen.compose` is stable within major versions. Internal modules prefixed with _ are private and may change without notice.

# 9 Troubleshooting

**Problem 1:** `error: k must be even and >= 4` when running `fat-tree`

**Cause:** The `-k` value supplied is either odd or less than 4. Fat-tree topology construction requires an even degree to partition switches evenly into pods.

**Solution:** Use an even value of 4 or greater: `-k 4`, `-k 8`, `-k 16`, etc. If you need a smaller topology, consider `leaf-spine` which has no such constraint.

**Problem 2:** `error: insufficient cities in region 'EU' for 60 nodes`

**Cause:** The requested node count exceeds the number of distinct cities in the chosen region in TopoGen's built-in city database. The EU region contains approximately 45 cities.

**Solution:** Reduce `--node-count` to fit within the city count for the region, choose a broader region such as `GLOB`, or pass `--allow-city-reuse` to permit multiple nodes in the same city. Run `topogen geo list-cities --region EU` to see the exact count.

**Problem 3:** Validation reports `reachability: FAIL` immediately after generation

**Cause:** The Erdős–Rényi generator with low edge probability $p$ can produce disconnected graphs. For $p < \ln n/n$, disconnection is likely.

**Solution:** Increase `--edge-probability`, or pass `--connected` to instruct the generator to retry with successive seeds until a connected graph is found. The retry count is printed when `--verbose` is set.

**Problem 4:** No routing simulation: traffic matrix validation always passes

**Cause:** TopoGen does not simulate routing protocols. Traffic matrix validators check structural properties (node reachability, SRLG diversity) but cannot verify that a routing protocol would distribute traffic as modelled.

**Solution:** For routing simulation, export the topology to AutoNetKit or netsim and run a routing protocol emulation. Use TopoGen traffic matrices as offered-load input, not as a routing verification tool.

**Problem 5:** Vendor naming scheme mismatch: ContainerLab images not found

**Cause:** The node `platform` field derived from `--vendor` does not match any image in ContainerLab's image registry, or the image is not pulled locally.

**Solution:** Use `--image-map` during export to override the default platform-to-image mapping. Run `docker images` to confirm which images are available. Alternatively, set `--vendor generic` to use the `linux` ContainerLab image, which always resolves.

**Problem 6:** Parquet export fails with `ModuleNotFoundError: pyarrow`

**Cause:** The Parquet export depends on `pyarrow`, which is an optional dependency not installed by the base `pip install topogen`.

**Solution:** Install the analytics extras: `pip install topogen[analytics]`. This pulls in `pyarrow` and `pandas`.

# 10 Integration with the Automation Ecosystem

TopoGen is designed to serve as the topology source for a broader network automation pipeline. The following integrations are directly supported.

## 10.1 AutoNetKit — IP Addressing and Configuration Generation

AutoNetKit takes a topology model and generates IP addressing plans and vendor-specific device configurations. Feed TopoGen output into AutoNetKit using the `autonetkit` export format:

**Step 1.** Generate and export:

```
topogen generate fat-tree -k 4 --seed 42 --vendor cisco \
    -o dc.yaml
topogen export dc.yaml --format autonetkit -o dc.ank.yaml
```

**Step 2.** Build configurations with AutoNetKit:

```
ank build dc.ank.yaml --validate
```

**Step 3.** Compile to vendor-specific configs with NetCfg:

```
netcfg compile output/ --platform ios-xe
```

**See also:** *AutoNetKit User Manual* (ANK-UM-001) — IP addressing models and configuration generation.

### 10.2 ContainerLab — Direct Lab Deployment

ContainerLab can deploy a virtual topology directly from a TopoGen-generated `.clab.yaml` file:

```
topogen generate fat-tree -k 4 --seed 42 --vendor arista \
    -o dc.yaml
topogen export dc.yaml --format containerlab -o dc.clab.yaml
containerlab deploy -t dc.clab.yaml
```

**See also:** ContainerLab documentation at `https://containerlab.dev` — Supported node kinds and image requirements.

### 10.3 netsim — Network Simulation

For higher-fidelity routing protocol simulation, export to netsim:

```
topogen export wan.yaml --format yaml \
    | netsim import --topology-format topogen
```

**See also:** *netsim User Manual* (NETSIM-UM-001) — Importing external topology definitions.

### 10.4 NetVis — Geographic Visualisation

NetVis renders interactive geographic topology maps from GeoJSON. Export a WAN topology and open it in NetVis:

**Step 1.** Export to GeoJSON:

```
topogen export wan.yaml --format geojson -o wan.geojson
```

**Step 2.** Open in NetVis:

```
netvis open wan.geojson
```

NetVis renders node positions on a world map, colours nodes by role, and draws link arcs with thickness proportional to capacity.

**See also:** *NetVis User Manual* (NETVIS-UM-001) — Layer configuration, filtering, and export options.

### 10.5 NetCfg — Configuration Compilation

NetCfg compiles abstract device models (produced by AutoNetKit from TopoGen topologies) to platform-specific configuration files:

```
topogen generate fat-tree -k 8 --seed 1 --vendor juniper -o dc.yaml
topogen export dc.yaml --format autonetkit -o dc.ank.yaml
ank build dc.ank.yaml
netcfg compile output/ --platform junos
```

**See also:** *NetCfg User Manual* (NETCFG-UM-001) — Compilation targets, template customisation, and platform support matrix.

## 11 Configuration File Reference

TopoGen accepts a TOML configuration file for setting defaults that would otherwise need to be passed on every invocation:

```toml
# ~/.topogen/config.toml

[defaults]
seed   = 42
vendor = "cisco"
format = "yaml"

[geo]
fiber_ratio      = 1.4
srlg_resolution  = 4

[export.containerlab]
default_image = "ceos:4.32.0F"

[logging]
level = "info"   # trace | debug | info | warn | error
```

TopoGen searches for the configuration file in the following order:

1. Path set by `TOPOGEN_CONFIG` environment variable
2. `./topogen.toml` (current directory)
3. `$HOME/.topogen/config.toml`

CLI flags always override configuration file values.

## 12 Environment Variables

| Variable | Description |
|---|---|
| `TOPOGEN_CONFIG` | Path to a TOML configuration file |
| `TOPOGEN_SEED` | Default seed (overridden by `--seed`) |
| `TOPOGEN_LOG` | Log level: `trace`, `debug`, `info`, `warn`, `error` |
| `TOPOGEN_CACHE_DIR` | Directory for the city database cache (default: `$HOME/.cache/topogen`) |
| `NO_COLOR` | Set to any value to disable coloured terminal output |

## 13 Further Reading

- *TopoGen Technical Report* (TOPOGEN-TR-001) — Architecture, seed-determinism design, generator algorithms, geographic database methodology, and performance benchmarks.
- *AutoNetKit User Manual* (ANK-UM-001) — IP addressing models, configuration generation workflows, and strict type-checking.
- *NetCfg User Manual* (NETCFG-UM-001) — Compilation targets, template customisation, and platform support matrix.
- *NetVis User Manual* (NETVIS-UM-001) — Interactive geographic topology visualisation and layer configuration.

- *Ecosystem Technical Report* (NETAUTO-TR-001) — Cross-tool integration architecture and reference pipeline designs.