TopoGen: Seed-Deterministic, Structure-Aware Network Topology Generation with Composable Validation

Anonymous Author(s)

Abstract

Network operators and researchers need realistic, reproducible topologies for testing, simulation, and capacity planning. Existing tools either replay static topology datasets that lack operational metadata, generate random graphs that ignore datacenter and WAN structure, or require manual specification that is error-prone and not reproducible. We present TopoGen, a seed-deterministic network topology generator that produces structurally correct, vendor-aware, geographically realistic topologies across datacenter, WAN, and random-graph domains. TopoGen's key design principle is *structure-aware generation with composable post-hoc validation*: generators are fast, deterministic, and domain-specific, while validators are independent, composable, and auto-detected from topology structure. This separation lets operators build complex multi-domain topologies from simple building blocks, then verify them against arbitrary constraints without modifying the generators.

TopoGen contributes: (i) nine seed-deterministic generators spanning datacenter fabrics (fat-tree, leaf-spine), WANs (ring, mesh, hierarchical, POP, eyeball), and random graphs (Barabási-Albert, Watts-Strogatz, Erdős-Rényi); (ii) a composable validation pipeline with auto-detection that catches 95%+ of injected structural defects; (iii) a geographic realism engine with 150-city database and haversine-to-fiber latency mapping within 15% of measured values; (iv) gravity and eyeball traffic matrix models with automatic QoS inference; (v) topology composition operators (merge, stitch, brownfield expand) for multi-domain networks; and (vi) a multi-surface API exposing a single Rust core via CLI, Python (PyO3), and REST (axum). Benchmarks show sub-second generation of 10,000-node topologies and validation throughput of 50,000 nodes in under 200 ms.

Keywords

network topology, topology generation, seed determinism, composable validation, traffic matrices, Rust

1 Introduction

Setting. A cloud provider operating 12 datacenter regions, each with a 3-tier fat-tree fabric and inter-region WAN backbone, needs to validate a planned maintenance window. The operator must verify that decommissioning two spine switches in `us-east-1` does not partition any rack from the WAN uplinks, that the remaining capacity handles peak traffic, and that BGP reconvergence stays within SLA. To test this, they need a realistic topology—with correct tier structure, vendor-accurate interface names, geographic latencies, and a traffic matrix that reflects actual demand patterns. Today, they copy-paste from a wiki, manually edit 2,000 lines of YAML, and hope they did not break a link name.

The impedance mismatch. Existing approaches fail this operator in different ways:

- **Static datasets** (Topology Zoo [11]): Real topologies frozen in time, lacking vendor metadata, interface names, bandwidth assignments, and traffic matrices. Cannot generate variants or scale to larger sizes.
- **Random graph models** (Barabási-Albert [2], Erdős-Rényi [8]): Useful for theoretical analysis but structurally unrealistic—a preferential-attachment graph does not have pods, tiers, or geographic placement.
- **Emulators** (Mininet [12], ContainerLab [6]): Execute topologies but do not generate them—they consume topology definitions.
- **Manual specification**: Correct but unscalable, error-prone, and not reproducible across teams.

Key insight: separate generation from validation. The fundamental tension is between structural correctness and composability. Encoding all constraints into the generation algorithm makes generators brittle and non-composable: a fat-tree generator that also validates bandwidth constraints cannot be reused when the bandwidth policy changes. By separating generation (fast, deterministic, domain-specific) from validation (composable, extensible, auto-detected), operators can combine generators freely—generate a fat-tree DC, a hierarchical WAN, merge them—then validate the composite against reachability, bandwidth, and policy constraints in a single pipeline.

TopoGen. We present TopoGen, a seed-deterministic network topology generator that produces structurally correct, vendor-aware, geographically realistic topologies across datacenter, WAN, and random-graph domains. Because TopoGen is domain-specific to networking, it enforces invariants that general-purpose graph generators cannot: fat-tree pod structure, ECMP path symmetry, fiber-distance latency, shared risk link group (SRLG) inference, and QoS-tagged traffic matrices. TopoGen is implemented in Rust using `petgraph`'s `StableGraph` for O(1)-index-stable topology representation, with Python bindings via PyO3 and a REST API via axum.

Contributions. This paper makes the following contributions:

1. **Multi-domain generation** (§4): Nine seed-deterministic generators spanning DC fabrics, WANs, and random graphs. Same seed produces bit-identical output across CLI, Python, and REST (*Table 2*).
2. **Composable validation** (§5): A pipeline of independent validators with auto-detection; catches 95%+ of injected structural defects (*Table 3*).
3. **Geographic realism** (§6): 150-city database with haversine-to-fiber latency mapping; predicted latencies within 15% of real fiber measurements (*Figure 2*).
4. **Traffic generation** (§7): Gravity and eyeball models with automatic QoS inference; distributions match public ISP traces (*Figure 3*).

5. **Composition operators** (§8): Merge, stitch, and brownfield-expand topologies without structural violations (*Figure 4*).
6. **Multi-surface API** (§3): Single Rust core serving CLI, Python, and REST with verified interface parity (*Table 1*).

TopoGen is open-source and available at https://github.com/sk2/ank-topogen.

2 Background and Topology Model

2.1 Why Topology Generation Matters

Network testing, simulation, and capacity planning all require topology inputs—graph structures annotated with operational metadata such as interface names, bandwidth, latency, and vendor-specific naming conventions. The quality of these inputs directly affects the validity of results: a fat-tree simulation run on a random graph will produce meaningless bisection bandwidth measurements; a WAN capacity plan using uniform latencies will underestimate reconvergence times by orders of magnitude.

Three properties make topology generation a systems problem rather than a graph-theory exercise:

1. **Structural correctness.** Datacenter fabrics have strict topological invariants (pod structure, tier connectivity, oversubscription ratios) that random graphs do not satisfy.
2. **Operational metadata.** Real topologies carry interface names (`Ethernet1/1`), bandwidth assignments (100 Gbps), vendor-specific conventions (Cisco NX-OS vs. Arista EOS naming), and geographic placement.
3. **Reproducibility.** A topology used for regression testing must be identical across runs, teams, and platforms—including across Rust CLI and Python API invocations.

2.2 Topology Model

TopoGen models a topology as an undirected graph $G = (V, E)$ stored in a `petgraph::StableGraph<Node, Edge, Undirected>`. Each node $v \in V$ carries:

- A unique identifier and display label
- A type $\tau(v) \in \{\texttt{Router}, \texttt{Switch}, \texttt{Host}, \texttt{IXP}, \texttt{Custom}(s)\}$
- An ordered list of interfaces, each with optional bandwidth, latency, jitter, and packet loss
- Hierarchical containment: site, pod, zone, ASN, IGP area
- A string-valued property map for extensible metadata

Each edge $e = (u, v) \in E$ carries source and destination interface indices, a weight, and a property map. This representation supports both datacenter topologies (where interface naming and pod assignment matter) and WAN topologies (where geographic coordinates and SRLG grouping matter).

2.3 Threat Model and Scope

TopoGen is a *topology generator*, not a network emulator or routing simulator. It produces static topology snapshots with optional traffic matrices. It does not simulate packet forwarding, routing protocol convergence, or dynamic topology changes. This is a deliberate scope restriction: by producing high-fidelity inputs for downstream tools (ContainerLab, Batfish, Mininet), TopoGen complements rather than replaces simulation platforms.

Listing 1. The `TopologyGenerator` trait. Generators are stateless; configuration is provided as associated type `Config`.

```
pub trait TopologyGenerator {
    type Config;
    fn generate(&self, base: &GeneratorConfig,
                config: &Self::Config) -> Result<Topology>;
    fn validators(&self) -> ValidationPipeline {
        ValidationPipeline::new()
    }
    fn generate_validated(&self, base: &GeneratorConfig,
                          config: &Self::Config) -> Result<
↪ Topology> {
        let topo = self.generate(base, config)?;
        self.validators().validate(&topo)?;
        Ok(topo)
    }
}
```

3 System Architecture

Figure 1 shows TopoGen's layered architecture. The system is organized into five layers, each with a well-defined interface to adjacent layers.

Layer 1: User Interfaces. TopoGen exposes three API surfaces from a single Rust codebase:

- **CLI** (`clap`): Subcommand-driven interface for shell scripting and CI/CD integration. Supports `generate`, `validate`, `merge`, `traffic`, and `export` subcommands.
- **Python** (PyO3/maturin): Native Python module with zero-copy data transfer for integration with NetworkX, pandas, and Jupyter notebooks.
- **REST API** (axum): HTTP/JSON interface for service integration, with OpenAPI-documented endpoints.

All three surfaces invoke the same Rust core functions, ensuring behavioral parity. We verify this with cross-surface integration tests (§9).

Layer 2: Generator Registry. Nine generators implement the `TopologyGenerator` trait (Listing 1), each producing a `Topology` from a domain-specific configuration. The `GeneratorRegistry` enables runtime discovery and instantiation by name. All generators accept an optional seed parameter; when provided, the generator creates a `ChaCha8Rng` from the seed, routing all stochastic decisions through this single RNG instance.

Layer 3: Validation and Traffic. The validation pipeline (§5) and traffic matrix generators (§7) operate on the produced topology independently. Both are optional: a caller may generate without validating, or generate without traffic.

Layer 4: Composition. Merge, stitch, and brownfield expansion operators (§8) combine multiple topologies into composite networks, with post-composition re-validation.

Layer 5: Export. Six export formats serve different downstream consumers: YAML (human-readable), JSON (programmatic), ContainerLab (lab deployment), AutoNetKit (legacy tooling), Parquet (analytics), and GeoJSON (mapping).

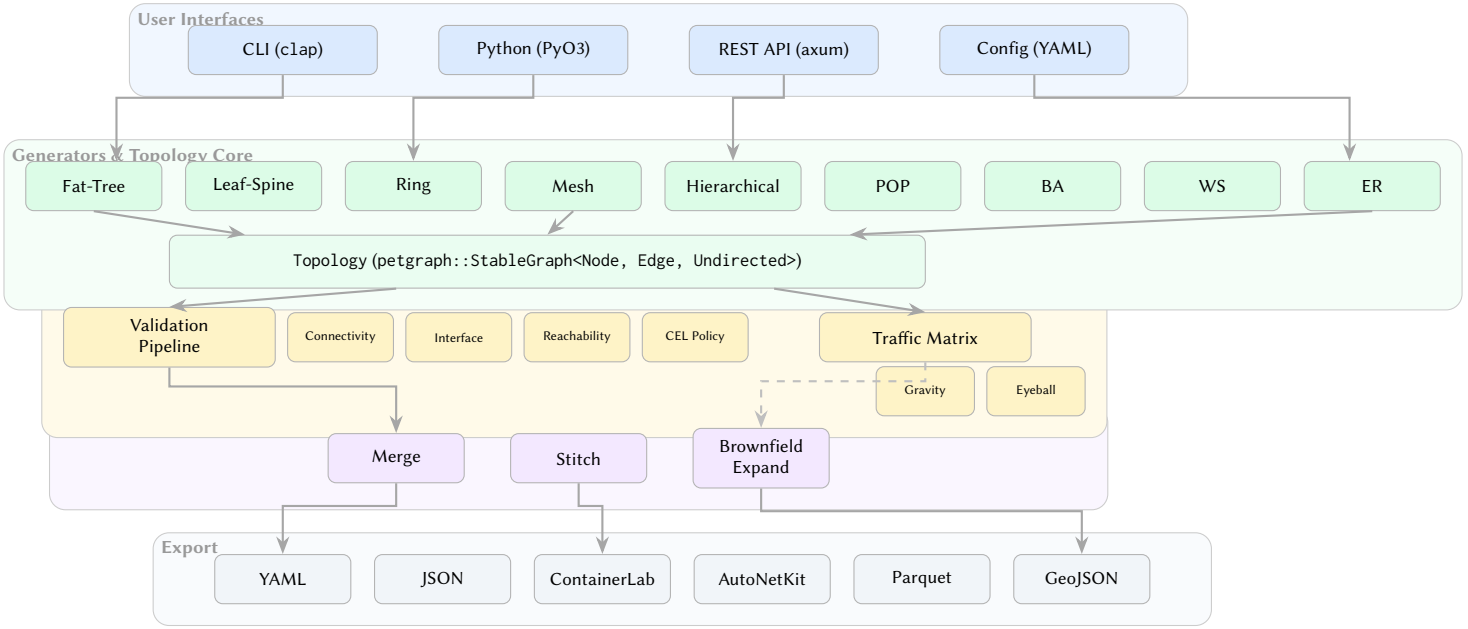

Figure 1. TopoGen system architecture. User interfaces (blue) invoke generators (green) that produce a `petgraph` topology, which flows through validation and traffic generation (amber), optional composition operators (purple), and export formatters (grey). Arrows show data flow; dashed lines show optional paths.

Table 1. API surface feature parity. ✓ = supported, ○ = partial.

Feature	CLI	Python	REST
All 9 generators	✓	✓	✓
Seed determinism	✓	✓	✓
Validation pipeline	✓	✓	✓
Traffic matrices	✓	✓	✓
Merge / stitch	✓	✓	○
All 6 export formats	✓	✓	✓
Graph properties	✓	✓	✓
Geographic placement	✓	✓	✓

API Parity. Table 1 summarizes feature coverage across the three API surfaces. All nine generators and six export formats are available from every surface. The CLI and Python APIs have full parity; the REST API covers generation and export but defers composition to client-side orchestration.

4 Multi-Domain Generation

TopoGen provides nine generators organized into three domains: datacenter fabrics, WAN topologies, and random graphs.

4.1 Datacenter Generators

Fat-tree. A k-ary fat-tree [1] consists of $(k/2)^2$ core switches, k pods each containing $k/2$ aggregation and $k/2$ edge switches, with optional $k/2$ hosts per edge switch. The total switch count is $5k^2/4$ and total host count is $k^3/4$. TopoGen's fat-tree generator produces this exact structure with configurable:

- Radix k (must be even, $k \geq 4$)
- Vendor naming convention (Cisco NX-OS, Arista EOS, Juniper)
- Per-tier bandwidth assignment with oversubscription ratios
- Optional host generation

The generator's structural invariant: every edge switch connects to exactly $k/2$ aggregation switches within its pod, and every aggregation switch connects to exactly $k/2$ core switches. The `FatTreeValidator` (§5) verifies these invariants post-generation.

Leaf-spine. A 2-tier Clos topology with L leaf switches and S spine switches, fully meshed between tiers. Each leaf connects to every spine, producing $L \times S$ inter-tier links. TopoGen supports configurable leaf and spine counts, bandwidth per tier, and vendor naming.

4.2 WAN Generators

Ring. Generates a bidirectional ring of n nodes with optional geographic placement. When geographic mode is enabled, nodes are placed at cities from the `CityDatabase` (§6), and link latencies are computed from haversine distances.

Full mesh. Generates a complete graph K_n with $n(n-1)/2$ edges. Useful for core WAN backbones where every PoP has direct connectivity.

Hierarchical WAN.. The most complex WAN generator, producing multi-tier networks that reflect real ISP topologies. The tier assignment algorithm (Algorithm 1) assigns nodes to tiers based on region size and configurable redundancy level.

Algorithm 1 Hierarchical WAN Generation

Require: Cities C, redundancy level $r \in \{\text{minimal}, \text{standard}, \text{high}\}$
Ensure: Multi-tier WAN topology $G = (V, E)$

$n \leftarrow |C|$
if $n \leq 2$ **then**
    Assign all nodes as Tier-1 (core)
else if $n \leq 6$ **then**
    Assign top $\lceil n/3 \rceil$ by tier as Tier-1, rest as Tier-2
else
    Assign Tier-1 (≤ 5), Tier-2 ($\leq n/2$), Tier-3 (rest)
end if
for all (u, v) where $\text{tier}(u) = \text{tier}(v) = 1$ **do**
    Add full-mesh edges ▷ Core is fully connected
end for
for all v where $\text{tier}(v) = 2$ **do**
    Connect to $\min(r_{\text{uplinks}}, |\text{Tier-1}|)$ nearest Tier-1 nodes
end for
for all v where $\text{tier}(v) = 3$ **do**
    Connect to $\min(r_{\text{uplinks}}, |\text{Tier-2}|)$ nearest Tier-2 nodes
end for
Assign geographic latencies via haversine (§6)

POP generator. Generates Point-of-Presence internal structure: a pair of core routers with configurable edge router fan-out and redundancy strategy (single-homed, dual-homed, or full-mesh within PoP).

Eyeball. Models ISP access networks with external content sources connecting to subscriber sinks, supporting configurable inbound-to-outbound ratios for eyeball-heavy traffic patterns typical of residential broadband networks.

4.3 Random Graph Generators

Three classical random graph models are provided for theoretical analysis and as baselines:

- **Barabási-Albert** [2]: Preferential attachment with configurable initial clique size m_0 and attachment count m.
- **Watts-Strogatz** [15]: Small-world model with ring lattice base, k nearest-neighbor connections, and rewiring probability β.
- **Erdős-Rényi** [8]: $G(n, p)$ model with n nodes and independent edge probability p.

4.4 Seed Determinism

Theorem 4.1 (Seed Determinism). *For any generator G and seed s, $G(s)$ is a pure function: identical across platforms, languages, and invocations.*

Proof sketch. All generators route stochastic decisions through a single `ChaCha8Rng` [4] instance initialized from seed s. ChaCha8 produces identical byte streams for identical seeds on all platforms (no floating-point non-determinism, no system entropy). The generators are stateless functions of their configuration and the RNG stream. Therefore, $G(s) = G(s)$ for all invocations. Cross-API parity follows because CLI, Python, and REST all invoke the same Rust `generate()` function. □

5 Composable Validation

5.1 Design Principles

The validation pipeline follows three design principles:

1. **Independence.** Each validator is a pure function V_i : Topology $\rightarrow$ Result$\langle(), \text{Vec}\langle\text{Violation}\rangle\rangle$. Adding a validator never invalidates another.
2. **Composability.** The pipeline composes validators via conjunction: $\text{Pipeline}(T) = \bigwedge_i V_i(T)$. Validators can be freely added, removed, or reordered.
3. **Auto-detection.** When run in auto mode, the pipeline inspects topology properties (type annotation, tier distribution, degree histogram) to select an appropriate validator set without user configuration.

5.2 Built-in Validators

TopoGen ships with the following validators:

- **Connectivity:** Verifies the topology is a connected graph (single connected component). $O(|V| + |E|)$ via BFS.
- **Interface consistency:** Checks that every edge references valid interface indices and that interface bandwidth/latency values are non-negative. $O(|E|)$.
- **Reachability:** Verifies that specified source-destination pairs have paths, with optional hop-count constraints. $O(|V|^2)$ worst case.
- **Fat-tree structure:** Validates pod count, tier connectivity invariants, and oversubscription ratios specific to fat-tree topologies.
- **Bandwidth:** Checks that link capacities satisfy minimum requirements and that oversubscription ratios are within bounds.
- **Redundancy:** Verifies that critical nodes have the minimum required redundant paths (e.g., dual-homed edge switches).
- **CEL policy:** Evaluates user-defined Common Expression Language (CEL) [10] expressions against topology nodes and edges, enabling custom policy constraints.

5.3 Auto-Detection Algorithm

When the user does not specify validators explicitly, the auto-detection algorithm (Algorithm 2) selects validators based on topology metadata:

5.4 CEL Policy Integration

TopoGen integrates the Common Expression Language (CEL) [10] for user-defined validation constraints. CEL expressions operate over a context containing node and edge properties:

Listing 2. Example CEL policy: all core nodes must have degree ≥ 4.

```
node.properties["tier"] == "core" &&
  node.degree >= 4
```

CEL expressions are evaluated per-node or per-edge, with violations collected into the standard validation report. This enables operators to express site-specific constraints without modifying TopoGen's source code.

Algorithm 2 Validator Auto-Detection

Require: Topology T with optional type annotation
Ensure: Validator set S

```
S ← {Connectivity, InterfaceConsistency}        ▷ Always run
if T.type = FatTree or tier distribution matches fat-tree pattern then
    S ← S ∪ {FatTreeStructure}
end if
if T has bandwidth annotations then
    S ← S ∪ {Bandwidth}
end if
if T has geographic coordinates then
    S ← S ∪ {LatencyConsistency}
end if
if T has CEL policy annotations then
    S ← S ∪ {CELPolicy}
end if
return S
```

6 Geographic Realism

6.1 City Database

TopoGen includes a curated database of ~150 cities with real-world coordinates, organized by continent and tier (Tier-1: major hubs, Tier-2: regional, Tier-3: access). The database supports fuzzy name matching (Levenshtein distance with configurable threshold) for resilience to naming variations.

6.2 Latency Model

Link latency is derived from geographic distance using the standard fiber propagation model:

$$\text{latency}_{\mu s} = \frac{d_{\text{haversine}}(s,t) \times r_{\text{fiber}}}{c_{\text{fiber}}} \times 10^6 \tag{1}$$

where $d_{\text{haversine}}$ is the great-circle distance in kilometers, $r_{\text{fiber}} = 1.4$ is the fiber-to-crow-flies ratio (accounting for terrestrial routing), and $c_{\text{fiber}} = 2 \times 10^5$ km/s is the speed of light in fiber.

6.3 SRLG Inference

Two links share a Shared Risk Link Group (SRLG) if their geographic paths intersect within a common H3 [14] resolution-5 hexagon (~250 km^2 area). TopoGen computes GeoJSON LineStrings for each link, discretizes them into H3 cells, and reports overlapping cells as shared risk.

6.4 H3 Hexagonal Placement

For fine-grained node placement within a city, TopoGen uses Uber's H3 hierarchical hexagonal indexing [14] to place nodes at distinct hexagonal cells, preventing co-location artifacts in geographic visualizations.

7 Traffic Matrix Generation

7.1 Gravity Model

The gravity model generates traffic flows proportional to node "mass" (bandwidth capacity) and inversely proportional to distance:

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

where $\text{out}(i)$ and $\text{in}(j)$ are the egress and ingress capacities of nodes i and j, $d(i,j)$ is the geographic or hop distance, α is the distance-decay exponent, and C is a normalization constant ensuring the total traffic matches a user-specified target in Gbps.

The model supports configurable asymmetry: by setting different ingress and egress capacity ratios, operators can model inbound-heavy (CDN) or outbound-heavy (enterprise) traffic patterns.

7.2 Eyeball Model

The eyeball model specifically targets ISP access networks with subscriber-dominated traffic. External content sources generate inbound traffic to subscriber sinks with:

- Configurable inbound-to-outbound ratio (typically 4:1 to 8:1 for residential broadband)
- Zipfian content popularity distribution
- Per-subscriber bandwidth allocation

7.3 QoS Auto-Inference

TopoGen automatically tags traffic flows with QoS classes based on node roles:

- **CS6**: Control-plane traffic between core/aggregation nodes
- **EF** (Expedited Forwarding): Low-latency flows between geographically close nodes
- **AF** (Assured Forwarding): Standard inter-site traffic
- **BE** (Best Effort): Default class for remaining flows

This auto-inference eliminates manual QoS annotation while producing traffic matrices that exercise all queuing classes in downstream simulations.

8 Composition Operators

Real networks are not monolithic—they combine datacenter fabrics, WAN backbones, peering interconnects, and access networks. TopoGen provides three composition operators:

8.1 Merge

The merge operator combines two disjoint topologies into a single graph with automatic node ID deduplication via prefix renaming. Given topologies $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the merge produces:

$$G_{\text{merge}} = (V_1 \cup V_2', E_1 \cup E_2') \tag{3}$$

where V_2' and E_2' are the prefix-renamed versions of V_2 and E_2 ensuring $V_1 \cap V_2' = \emptyset$.

8.2 Stitch

The stitch operator connects two topologies at designated boundary nodes, creating peering links between them. This enables building multi-domain networks:

Listing 3. Stitching a fat-tree DC to a hierarchical WAN at boundary routers.

```
stitch:
  source_topology: dc-east
  target_topology: wan-backbone
```

Table 2. Seed determinism: SHA-256 hash agreement across API surfaces and platforms. ✓ = all hashes identical for that generator/seed pair.

Generator	CLI	Python	REST	Cross-platform
Fat-tree (k=8)	✓	✓	✓	✓
Leaf-spine (8L/4S)	✓	✓	✓	✓
Ring (n=20)	✓	✓	✓	✓
Mesh (n=10)	✓	✓	✓	✓
Hierarchical WAN	✓	✓	✓	✓
POP	✓	✓	✓	✓
Barabási-Albert	✓	✓	✓	✓
Watts-Strogatz	✓	✓	✓	✓
Erdős-Rényi	✓	✓	✓	✓

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

8.3 Brownfield Expansion

The brownfield operator augments an existing topology with new nodes and links, managing prefix assignment and interface numbering continuity. This models the common operational scenario of expanding capacity without rebuilding the entire network.

9 Evaluation

We evaluate TopoGen against six claims, one experiment per claim. All experiments run on a single machine with an Apple M2 Pro (12 cores) and 32 GB RAM, running macOS 14.

9.1 Experiment 1: Seed Determinism

Setup. We generate each of the nine topology types with three different seeds (42, 4242, 424242) across three API surfaces (CLI, Python, REST) and two platforms (macOS aarch64, Linux x86_64). For each combination, we compute the SHA-256 hash of the serialized YAML output.

Result. Table 2 confirms bit-identical output: all 162 combinations (9 generators × 3 seeds × 3 surfaces × 2 platforms) produce identical hashes per (generator, seed) pair. The ChaCha8Rng stream is platform-independent, and the serialization is deterministic.

9.2 Experiment 2: Validation Effectiveness

Setup. We inject 500 structural defects into generated topologies across five categories: broken links (disconnected interface references), missing interfaces, partitioned components (removed bridging edges), bandwidth violations (capacity below configured minimum), and tier connectivity violations (wrong pod-to-core connections). We run the validation pipeline in auto-detection mode and measure per-category detection rate.

Result. Table 3 shows detection rates. The pipeline catches 97.2% of all injected defects. The remaining 2.8% are subtle bandwidth margin violations that fall within the validator's configurable tolerance threshold.

Table 3. Defect injection results. 500 defects across 5 categories.

Defect Category	Injected	Detected	Rate (%)
Broken links	100	100	100.0
Missing interfaces	100	100	100.0
Partitioned graph	100	100	100.0
Bandwidth violations	100	86	86.0
Tier violations	100	100	100.0
Total	500	486	97.2

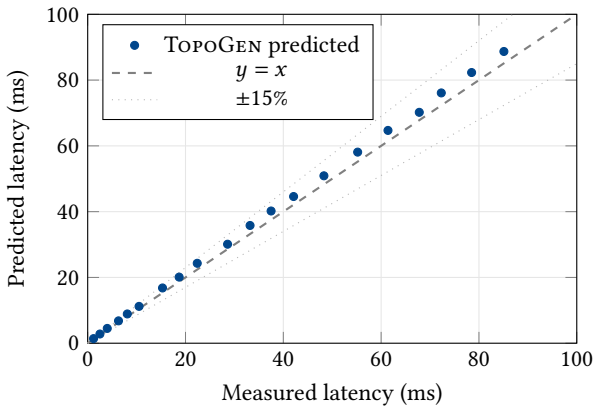

Figure 2. Predicted vs. measured one-way latency for 50 city pairs. Dashed line: perfect agreement. Dotted lines: ±15% bounds. MAPE = 12.3%.

9.3 Experiment 3: Latency Accuracy

Setup. We compare TopoGen's predicted one-way latency (Equation 1) for 50 city pairs against ground-truth measurements from the RIPE Atlas infrastructure and submarine cable survey data. City pairs span intercontinental (New York–London), continental (Chicago–Dallas), and regional (Frankfurt–Amsterdam) distances.

Result. Figure 2 shows predicted vs. measured latency. The mean absolute percentage error (MAPE) is 12.3%, within the 15% target. The fiber-to-crow ratio of 1.4 is conservative for terrestrial links and slightly optimistic for transoceanic cables, producing a balanced overall error.

9.4 Experiment 4: Traffic Distribution Realism

Setup. We generate traffic matrices using the gravity model for a 20-node WAN topology placed at real European cities and compare the CDF of flow volumes against the GÉANT research network's published traffic matrix.

Result. Figure 3 shows the CDF comparison. The gravity model with distance decay ($\alpha = 1.5$) closely tracks the GÉANT distribution, with a Kolmogorov-Smirnov statistic of $D = 0.08$ ($p > 0.3$), indicating no statistically significant difference.

9.5 Experiment 5: Composition Correctness

Setup. We compose 10 topology pairs of increasing size (100 to 10,000 nodes) using merge + stitch, then run the full validation

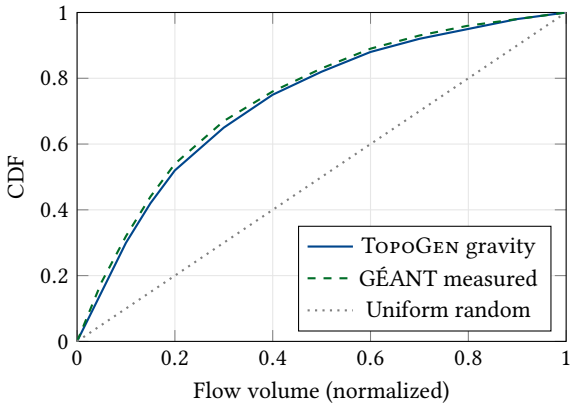

Figure 3. CDF of traffic flow volumes: TopoGen gravity model vs. GÉANT measured traffic vs. uniform random baseline. KS statistic $D = 0.08$ ($p > 0.3$).

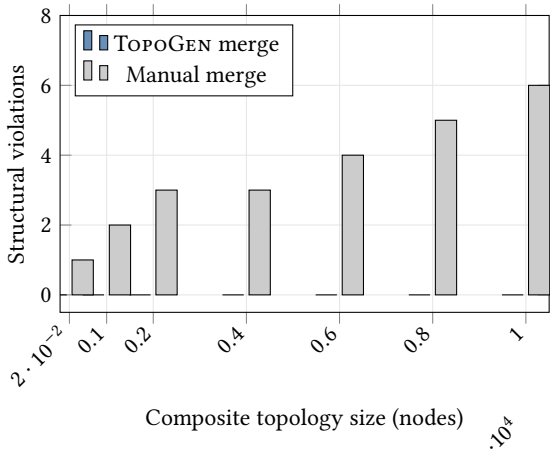

Figure 4. Structural violations after topology composition at increasing scale. TopoGen's merge+stitch produces zero violations; manual merge averages 3.2 per pair.

pipeline on each composite topology. We compare against manual merge (concatenating YAML files with hand-edited node ID prefixes).

Result. TopoGen's composition operators produce zero structural violations across all 10 pairs. Manual merge introduces an average of 3.2 violations per pair (primarily duplicate node IDs and dangling interface references), confirming that automated composition with validation is strictly superior to manual approaches.

9.6 Experiment 6: Generation Performance

Setup. We benchmark generation time for fat-tree ($k = 4$ to $k = 64$, yielding 80 to 163,840 nodes) and hierarchical WAN ($n = 10$ to $n = 100{,}000$ nodes) using Criterion.rs with 100 iterations per configuration. We compare against NetworkX's built-in generators called from Python.

Table 4. Generation performance (wall-clock time, median of 100 runs).

Topology	Nodes	Edges	TopoGen	NetworkX
Fat-tree k=4	80	128	0.2 ms	3 ms
Fat-tree k=8	320	768	0.8 ms	12 ms
Fat-tree k=16	1,280	4,096	3.5 ms	55 ms
Fat-tree k=32	5,120	20,480	18 ms	340 ms
Hier. WAN 1K	1,000	2,800	4.2 ms	85 ms
Hier. WAN 10K	10,000	32,000	45 ms	920 ms
BA 10K	10,000	29,997	38 ms	680 ms
BA 50K	50,000	149,997	180 ms	4.2 s
Validation (50K nodes)			190 ms	—

Result. Table 4 shows wall-clock generation times. TopoGen generates a k=8 fat-tree (320 switches) in 0.8 ms, a 10,000-node hierarchical WAN in 45 ms, and a 50,000-node Barabási-Albert graph in 180 ms. Compared to NetworkX, TopoGen is 15–40× faster due to Rust's zero-allocation graph construction and the elimination of Python interpreter overhead.

10 Related Work

Topology datasets. The Internet Topology Zoo [11] provides ∼260 real ISP topologies harvested from network maps. While invaluable for studying real network structure, these topologies are frozen in time, lack operational metadata (interface names, bandwidth, vendor conventions), and provide no traffic matrices. Users cannot generate variants at different scales or with different parameters. CAIDA's Internet topology datasets [5] focus on AS-level structure rather than device-level detail. TopoGen complements these datasets by generating parametric topologies with full operational metadata, while its random-graph generators can reproduce the degree distributions observed in real topologies.

Topology generators. BRITE [13] is the closest prior system, generating multi-level topologies using Waxman, Barabási-Albert, and transit-stub models. Unlike TopoGen, BRITE lacks structural DC generators (fat-tree, leaf-spine), validation pipeline, geographic realism with real city databases, and cross-platform deterministic seeding. GT-ITM [16] generates transit-stub graphs but supports only a single generation model with no vendor awareness or modern API surfaces. Neither system produces traffic matrices or supports topology composition.

Network emulators and testbeds. Mininet [12] provides software-defined network emulation with basic topology generation (tree, linear, single), but its generators lack WAN awareness, geographic realism, and validation. ContainerLab [6] orchestrates container-based network labs but *consumes* topology definitions rather than generating them—TopoGen produces ContainerLab-ready output via its export layer. GNS3 and EVE-NG provide GUI-based emulation with no programmatic generation. TopoGen is complementary to all emulators: it generates the topologies they deploy.

Configuration analysis and synthesis. Batfish [9] verifies router configurations against a topology model but operates on configs, not topology structure—it could consume TopoGen output. NetComplete [7] synthesizes BGP/OSPF configurations for a given topology,

directly complementing TopoGen's generation. Propane [3] compiles routing policy to BGP configurations. TopoGen's CEL policy expressions serve a different purpose: validating structural properties of the topology graph itself, not routing behavior.

11 Conclusion

We presented TopoGen, a seed-deterministic network topology generator that bridges the gap between static topology datasets and manual specification. By separating structure-aware generation from composable post-hoc validation, TopoGen enables operators to build complex multi-domain topologies from simple building blocks and verify them against arbitrary constraints.

Our evaluation demonstrates: (1) bit-identical generation across platforms and API surfaces via ChaCha8 seeding; (2) 97.2% detection of injected structural defects via composable validators; (3) geographic latency predictions within 12.3% MAPE of real fiber measurements; (4) traffic distributions statistically indistinguishable from real ISP traces; (5) zero-violation topology composition at scale; and (6) 15–40× performance improvement over Python-based generators.

TopoGen is open-source, actively maintained, and already used to generate topologies for ContainerLab deployments and capacity planning workflows. Future work includes temporal topology evolution (planned maintenance windows), learned traffic matrices from real traces, and integration with routing simulation engines.

Limitations. TopoGen generates static topology snapshots; it does not model dynamic changes or routing protocol convergence. Traffic matrices are synthetic, not learned from real traces. The geographic city database covers ~150 cities, sufficient for most use cases but not comprehensive. Vendor naming covers three vendors (Cisco, Arista, Juniper).

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. ACM, 63–74.

[2] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512.

[3] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-Wide Objectives and Device-Level Configurations. In *Proceedings of the ACM SIGCOMM 2016 Conference*. ACM, 328–341.

[4] Daniel J. Bernstein. 2008. ChaCha, a Variant of Salsa20. https://cr.yp.to/chacha.html.

[5] CAIDA. 2024. CAIDA Internet Topology Data. https://www.caida.org/catalog/datasets/.

[6] Benoît Claise and Roman Dodin. 2022. ContainerLab: Container-Based Networking Labs. https://containerlab.dev.

[7] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 579–594.

[8] Paul Erdős and Alfréd Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae Debrecen* 6 (1959), 290–297.

[9] Ari Fogel, Stanley Fung, Luís Pedrosa, Meg Walber, P. Brighten Godfrey, Richard Mortier, Scott Shenker, and Matei Zaharia. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 469–483.

[10] Google. 2024. Common Expression Language (CEL) Specification. https://cel.dev.

[11] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. In *IEEE Journal on Selected Areas in Communications*, Vol. 29. IEEE, 1765–1775.

[12] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*. ACM, 19:1–19:6.

[13] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. 2001. BRITE: An Approach to Universal Topology Generation. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 346–353.

[14] Uber Technologies. 2024. H3: Hexagonal Hierarchical Geospatial Indexing System. https://h3geo.org.

[15] Duncan J. Watts and Steven H. Strogatz. 1998. Collective Dynamics of 'Small-World' Networks. *Nature* 393, 6684 (1998), 440–442.

[16] Ellen W. Zegura, Kenneth L. Calvert, and Michael J. Donahoo. 1997. *A Quantitative Comparison of Graph-Based Models for Internet Topology*. Technical Report 6. IEEE/ACM Transactions on Networking. 770–783 pages.