netsim: Convergence-First Network Simulator

Technical Reference

Simon Knight

March 2026

Abstract. netsim is a tick-based network simulator written in Rust that bridges the gap between algorithmic configuration analysis and container-based network emulation. It provides protocol-accurate simulation of OSPF (v2/v3), IS-IS, BGP (including EVPN/VXLAN and L3VPN), MPLS/LDP, RSVP-TE, BFD, LACP, LLDP, RIPv2/RIPng, STP, VRRP v2/v3, and DHCP Relay with deterministic convergence detection at 486-device scale. This technical report documents the system's architecture, design rationale, data model, and usage. It is intended for engineers evaluating netsim for network validation, contributors to the codebase, and researchers interested in the convergence-first simulation model.

Version	Date	Changes
1.0	March 2026	Initial release
1.1	March 2026	Add RIP, STP, VRRP, DHCP Relay, path tracing, traceroute, dig, iperf, CoPP, E2E hardening

Contents

1 Introduction

netsim is a network simulator that models routing protocol behaviour at packet granularity with deterministic, reproducible results. It accepts a single YAML file describing a network topology—devices, links, protocol configurations, traffic generators, failure patterns, and assertions—and simulates the network tick by tick until all routing protocols converge. It implements 15 protocol families spanning link-state routing (OSPF v2/v3, IS-IS), distance-vector routing (RIPv2/RIPng), path-vector routing (BGP-4 with EVPN), label switching (MPLS/LDP, RSVP-TE), L2 infrastructure (STP, LACP, LLDP, VXLAN), first-hop redundancy (VRRP v2/v3), address management (DHCP Relay), and fast failure detection (BFD). It replaces the need for container-based emulation [4] in scenarios where determinism and speed matter more than vendor-specific NOS fidelity.

Two core design principles explain every subsequent decision in this document:

1. **Convergence is a first-class time coordinate.** Routing convergence is not a side effect of "running long enough." It is a precisely defined predicate (the simultaneous stability of 11 protocol states) that is observable, addressable, and scriptable. Events, assertions, and chaos engineering all reference convergence as a time anchor (`at: converged + 500ms`).
2. **Determinism over fidelity.** netsim models RFC behaviour, not vendor-specific implementation quirks. All timers are integer ticks, all RNGs are seeded, and all protocol synchronisation uses snapshot-then-apply rather than per-message queuing. This trades the last mile of realism for bit-identical reproducibility across runs.

This document is intended for three audiences:

- **Network engineers** evaluating netsim for pre-deployment validation in CI/CD pipelines.
- **Contributors** to the netsim codebase who need to understand architectural decisions before modifying the system.
- **Researchers** interested in the convergence-first simulation model and its implications for network testing.

Familiarity with IP routing protocols (OSPF, BGP) and basic Rust syntax is assumed. Experience with network emulation tools (Containerlab, ns-3) is helpful but not required.

Section 2 defines the core data model (devices, wires, interfaces). Section 3 presents the system architecture and the tick pipeline. Section 4 explains the convergence predicate in detail. Section 5 covers the forwarding architecture (RIB/FIB, ECMP). Section 6 surveys all 15 protocol families (including RIP, STP, VRRP, and DHCP Relay). Section 7 describes the YAML topology format and chaos engineering. Section 8 covers EVPN/VXLAN at simulation abstraction. Section 9 documents the CLI tool, interactive daemon, diagnostic commands (traceroute, dig, iperf), and path tracing engine. Section 11 provides end-to-end usage examples. Section 12 catalogues the 40+ example topologies shipped with netsim and their protocol coverage. Section 13 documents performance characteristics, benchmarking infrastructure, and scalability. Section 14 lists known limitations honestly. Section 15 outlines future work.

Readers wanting to get started quickly should read sections 2 and 11, then refer to individual sections as needed.

2 Data Model

This section defines the core abstractions that all other components build upon.

2.1 Devices

netsim models five device types, all implementing the `Device` trait:

Table 1. Device types and their roles.

Type	Rust type	Description
Router	`Router`	Full L3 device with RIB/FIB, protocol daemons (OSPF v2/v3, BGP, IS-IS, RIP, MPLS, RSVP-TE, BFD, LLDP, LACP, VRRP, DHCP Relay), bridge domains (STP), VRFs
Host	`Host`	L3 endpoint with a default gateway; generates and sinks traffic
Switch	`Switch`	L2 device with MAC learning and VLAN support
Hub	`Hub`	L1 device that floods all frames to all ports
Wire	`Wire`	Point-to-point link with latency, loss, jitter, and capacity modelling

Note

A sixth device type, `DnsServer`, implements the `Device` trait for internal DNS resolution (used by the `dig` command). It is not stored in the engine's type-segregated vectors and cannot be instantiated from YAML—it exists solely to provide DNS query/response handling within the simulation.

Design Decision

Why five types instead of a generic "Node"?
Type-segregated storage (`Vec<Option<Box<Router>>>`, `Vec<Option<Box<Host>>>`, etc.) enables Rayon's `par_iter_mut` to process each device type independently without `unsafe` code [13]. A single `Vec<Box<dyn Device>>` would require `unsafe` downcasting or runtime locks for parallel mutation. The five-type design trades some code duplication for zero-cost safe parallelism.

Each device is identified by a `DeviceId(u32)` and contains a vector of interfaces. Interfaces are identified by `InterfaceId(u32)` and carry IPv4/IPv6 addresses, MAC addresses, and link-state information.

2.2 Wires

A `Wire` connects exactly two interfaces on different devices. It models:

- **Latency**: Frames spend a configurable number of ticks in transit (implemented as a `VecDeque` of in-flight frames).
- **Packet loss**: Probabilistic drop using a seeded PRNG.
- **Jitter**: Random variation added to the base latency.
- **Capacity**: Interface bandwidth for utilisation tracking (added in Phase 118).

2.3 Identifiers

Table 2. Identifier types used throughout netsim.

Type	Inner	Usage
`DeviceId`	`u32`	Index into device storage vectors
`InterfaceId`	`u32`	Index into a device's interface vector
`WireId`	`u32`	Index into the wire storage vector
`RouterId`	`Ipv4Addr`	OSPF/BGP router identifier

Note

`InterfaceId(u32::MAX)` is a sentinel value used for null routes (traffic explicitly dropped). This appears in the FIB when a route has no valid next-hop.

3 System Architecture

netsim processes a simulation in three stages: *build*, *run*, and *output*.

3.1 Build Stage

The Builder reads a YAML topology file and constructs the in-memory simulation state:

1. **Schema validation**: The YAML is validated against the topology schema. Missing required fields, unknown device types, and invalid IP addresses are caught here.
2. **Device construction**: Each device entry creates a `Router`, `Host`, `Switch`, or `Hub` with configured interfaces.
3. **Wire creation**: Each link entry creates a `Wire` connecting two interfaces. Latency, loss, and jitter are applied from named impairment profiles or inline values.
4. **Pattern resolution**: Pattern-based selectors (`by_tier: spine`, `name_pattern: "r*"`) are expanded to concrete device/link lists.
5. **Event scheduling**: Failure patterns (`FailurePattern`) are expanded into a sequence of `ScheduledEvents`. Convergence-relative events (`at: converged + 500ms`) are deferred until convergence is detected at runtime.

Design Decision

Why separate build from run?
Separating build from run allows the Builder to perform expensive validation (schema checks, pattern expansion, event scheduling) once, producing an immutable simulation configuration. The engine then runs without re-validating on every tick. This also enables future features like topology caching and incremental rebuilds.

3.2 Run Stage: The Tick Loop

The engine advances simulation time one tick at a time (default: 1 ms/tick). It uses a **Zero-Copy Structure-of-Arrays (SoA)** architecture to maximise parallel execution performance and CPU cache hit rates. Device storage uses stable memory locations (SlotMaps), allowing pointers and indices to persist across ticks without the $O(N)$ reconstruction overhead typical of object-oriented designs.

Each tick executes eight phases in strict order:

Table 3. The 8-phase tick pipeline.

Phase	Name	Description
P0	Drain commands	Dequeue pending gRPC commands for deterministic delivery
P1	Device→Wire	Move outbound frames from device output queues to connected wires
P2	Tick devices	Run all protocol state machines (OSPF SPF, BGP decision, BFD timers, etc.). Parallel via Rayon if ≥500 devices
P2.5	EVPN sync	Snapshot-then-apply EVPN Loc-RIB synchronisation
P3	Tick wires	Advance in-transit frames by one tick; apply impairments (loss, jitter)
P4	Wire→Device	Deliver arrived frames to destination device input queues
P5	Convergence check	Evaluate the composite convergence predicate (section 4)
P6	Hooks & quiescence	Check quiescence; fire convergence hooks; schedule deferred events
P7	Export	Write PCAP, BMP, NetFlow snapshots

Note

The phase ordering is load-bearing. P1 must precede P2 so that devices process frames that were sent in the *previous* tick, not the current one. P2.5 must follow P2 so that EVPN updates reflect the current tick's BGP processing. Reordering phases will break protocol correctness.

3.3 Determinism Guarantees

Three properties ensure that two runs of the same topology produce identical results:

1. **Integer ticks**: All protocol timers are expressed in integer ticks. No floating-point arithmetic appears in timing-sensitive code paths.
2. **No shared mutable state during P2**: During device ticking, each device reads only from its own input queue and writes only to its own output queue. Rayon's `par_iter_mut` is safe because no two devices access the same memory.
3. **Seeded PRNGs**: All random number generators (ECMP hashing, probabilistic failure injection, jitter) use deterministic seeds derived from the topology configuration.

3.4 Adaptive Parallelism

Design Decision

Problem: Rayon's thread-pool scheduling has non-trivial overhead for small workloads.
Options considered: (1) Always parallel. (2) Always serial. (3) Adaptive threshold.
Decision: Adaptive threshold at 500 devices. Below 500, serial iteration. At or above 500, Rayon `par_iter_mut`.

Trade-off: The threshold is a compile-time constant, not auto-tuned. A topology with 499 devices runs serially even if the machine has 64 cores. Benchmarking on an 8-core CPU shows that serial execution is 2–3% faster for 486 devices, validating the 500-device threshold as a reasonable heuristic for modern hardware.

3.5 Output Stage

After the simulation terminates (convergence detected, max ticks reached, or user abort), netsim exports:

- **Routing matrix** (JSON): Per-device FIB entries with next-hops, metrics, and protocol origins.
- **PCAP captures**: Per-interface packet captures for offline analysis in Wireshark.
- **BMP feeds**: BGP Monitoring Protocol messages for integration with route collectors.
- **NetFlow/IPFIX**: Flow-level traffic summaries.
- **Convergence report**: Tick-by-tick protocol state transitions showing exactly when each protocol stabilised.
- **Assertion results**: Pass/fail for all topology-defined assertions.
- **Topology diff**: Before/after comparison of routing state across topology changes.
- **Capacity report**: Per-interface utilisation against configured bandwidth.
- **Path trace**: Hop-by-hop forwarding path analysis with anomaly detection (loops, blackholes, asymmetry).
- **FIB export**: Routing state in netauto/fib/v1.0 schema for integration with external network analysis tools.

4 Convergence Detection

Convergence detection is netsim's most distinctive feature. This section explains the design in full.

4.1 Problem

Network simulators typically determine "convergence" by one of two methods:

1. **Timeout**: Run for a fixed number of seconds/ticks and assume convergence. Fragile—too short misses slow convergence; too long wastes time.
2. **Quiescence**: Stop when no more packets are in flight. Incorrect—LLDP, BFD, and other periodic protocols never stop sending packets.

Neither method answers the operator's actual question: "Have all routing protocols finished computing and installed stable forwarding state?" The problem is well-documented: delayed Internet routing convergence [9] can cause transient loops and black holes that persist for seconds after a topology change.

4.2 Design

netsim defines convergence as the simultaneous stability of 11 independent protocol states, sustained for N consecutive ticks (default $N = 10$, configurable via `EngineConfig`):

$$converged(t) = \bigwedge_{i=1}^{11} (\forall\, \tau \in [t{-}N{+}1,\ t] \ :\ stable_i(\tau)) \tag{1}$$

The 11 flags fall into two categories: *table stability flags* (1–7) that require sustained quiescence for N ticks, and *protocol session flags* (8–11) that are instantaneous checks.

#	Flag	Stable when…
Table stability (threshold-based):		
1	`fib_stable`	No global IPv4 FIB entry changed for N consecutive ticks
2	`vrf_fibs_stable`	No per-VRF FIB entry changed for N ticks
3	`mpls_lfib_stable`	No MPLS LFIB entry changed for N ticks
4	`ldp_bindings_stable`	No LDP label binding changed for N ticks
5	`vpnv4_loc_rib_stable`	No VPNv4 route changed for N ticks
6	`bgp_loc_rib_stable`	No BGP IPv4 unicast Loc-RIB entry changed for N ticks
7	`evpn_loc_rib_stable`	No EVPN route changed for N ticks
Protocol session state (instantaneous):		
8	`ospf_converged`	All OSPF neighbours in Full or Down state
9	`bgp_converged`	All reachable BGP sessions in Established state
10	`rsvp_converged`	All configured RSVP-TE tunnels in Up state
11	`isis_converged`	No IS-IS adjacency in Initializing state

Final convergence requires *both* gates to pass simultaneously: all seven table flags must be stable for N consecutive ticks, *and* all four protocol session checks must be true.

4.2.1 Dirty-Flag Mechanism

Each forwarding table uses a `changed: bool` dirty flag. When a route is installed, removed, or modified, the engine compares the new entry against the existing one—if the value actually differs, the flag is set to `true`. This equality check prevents false positives from idempotent reinstalls (e.g., OSPF recomputing the same SPF result).

```
// In check_convergence() (Phase 5 of the tick loop):
any_changed = any_fib_changed
            | any_vrf_fib_changed
            | any_lfib_changed
            | any_ldp_changed
            | any_vpnv4_changed
            | any_evpn_changed
            | any_bgp_loc_rib_changed;

if any_changed {
    stable_ticks = 0;    // Reset the threshold counter
} else {
    stable_ticks += 1;   // Count consecutive stable ticks
}
converged = stable_ticks >= threshold
         && ospf_converged
         && bgp_converged
         && rsvp_converged
         && isis_converged;
```

After each convergence check, all dirty flags are cleared so they are fresh for the next tick. This clear-after-read pattern ensures that a flag set during tick t is visible exactly once, during the Phase 5 check at the end of tick t.

4.2.2 Per-Protocol Session Checks

The four protocol session flags use different stability criteria tailored to each protocol's FSM:

- **OSPF**: Converged when every neighbour is in `Full` or `Down` state. Adjacencies in Init, 2-Way, ExStart, Exchange, or Loading block convergence because they indicate ongoing database synchronisation.
- **BGP**: Converged when every *reachable* peer is in `Established`. A peer is considered unreachable if its IP has no FIB entry or no up interface matching the route—unreachable peers are skipped to prevent an interface-down event from permanently blocking convergence.
- **RSVP-TE**: Converged when every configured LSP is in `Up` state. Non-Up tunnels affect label programming, so they must complete before the LFIB can be considered stable.
- **IS-IS**: Converged when no adjacency is in `Initializing` state. Both `Up` and `Down` adjacencies are considered stable terminal states.

4.2.3 EVPN Loc-RIB Integration

EVPN convergence tracking is separated from the regular BGP IPv4 Loc-RIB. The `BgpInstance` maintains a dedicated change flag:

```
evpn_loc_rib: BTreeMap<EvpnNlri, BgpEvpnRoute>,
evpn_loc_rib_changed_this_tick: bool,
```

Route insertions compare the new route against the existing entry before setting the flag, preventing churn from identical route re-advertisements. The EVPN synchronisation pass (Phase 2.5 of the tick loop) propagates Loc-RIB entries across established iBGP sessions, preserving the original `peer_router_id` to prevent route-reflector-induced flapping.

Bridge domain FDB changes (`fdb_changed_this_tick`) are tracked separately from the EVPN Loc-RIB. FDB changes from data-plane MAC learning do not directly affect convergence—only EVPN control-plane route changes do. Additionally, MAC ageing events explicitly do *not* set the FDB change flag, since ageing is routine housekeeping, not a convergence-relevant state change.

> **Design Decision**
>
> **Why exclude observability protocols?**
> PCAP, BMP, and NetFlow generate continuous background traffic that never stops. Including them in the convergence predicate would prevent convergence from ever being detected. Similarly, LLDP sends periodic hellos every 30 seconds regardless of routing state—it is explicitly excluded from quiescence checks via `EtherType::is_control_plane() == false`.

4.2.4 Quiescence: The Orthogonal Signal

Quiescence is tracked separately from convergence and serves a different purpose. A network is *quiescent* when no control-plane packets are in transit for W consecutive ticks (default $W = 5$):

$$quiescent(t) = \forall\, \tau \in [t-W+1,\ t] \ :\ \neg\, has\_control\_plane(\tau) \tag{2}$$

The control-plane check scans every device's interface queues (inbound and outbound) and every wire's in-transit buffer for frames matching either of two tests:

1. **EtherType-level**: `EtherType::is_control_plane()` returns `true` only for ARP frames. LLDP (`0x88CC`) and Slow Protocols (LACP, `0x8809`) are deliberately excluded.
2. **IP protocol-level**: A two-gate test—the protocol must be classified as control-plane *and* must not be a periodic keepalive. This excludes BFD, IS-IS Hello, LDP keepalive, RSVP refresh, VRRP advertisements, and RIP request/response from blocking quiescence. OSPF Hello packets *do* block quiescence because they signal active neighbour maintenance.

> **Note**
>
> Quiescence and convergence can diverge in both directions. A network can be quiescent (no control-plane packets in flight) but not converged (FIB still rippling from a recent change). Conversely, a network can be converged (stable forwarding state) but not quiescent (OSPF Hellos still flowing). The `run_until_converged()` method uses convergence, not quiescence, as its stopping criterion.

> **Note**
>
> Protocols added after the initial 11 flags—RIP, STP, VRRP, and DHCP Relay—do not require additional convergence flags. RIP installs routes into the RIB, which updates the FIB (tracked by flag #1). STP modifies port forwarding state on Switches and Router bridge domains (which affects frame delivery but not the routing FIB). VRRP changes the active gateway, which manifests as a FIB change on the relevant router. DHCP Relay forwards packets without modifying routing state. The existing 11-flag predicate therefore covers these protocols transitively.

4.3 Convergence as a Time Coordinate

Once convergence is detected at tick t_c, netsim makes t_c available as a named time anchor. Subsequent events can reference it:

Listing 1. Convergence-relative event scheduling.

```
events:
  - at: converged + 100    # 100 ticks after convergence
    action: link_down
    link: r1-r2

  - at: converged + 500ms  # 500ms after convergence
    device: PE1
    command: show bgp summary

assertions:
  - type: convergence_time
    max_ticks: 200         # fail if convergence takes > 200 ticks
```

This enables three workflows that are impossible with timeout-based convergence:

1. **CI-gatable assertions**: "Fail this pipeline if convergence takes longer than 200 ticks after a link failure."
2. **Convergence-relative chaos**: "Inject a second failure 100 ticks after convergence from the first failure."
3. **Post-convergence inspection**: "Run `show bgp summary` on PE1 exactly 500 ms after convergence to verify the BGP table."

5 Forwarding Architecture

5.1 RIB/FIB Separation

Each router maintains two distinct data structures:

- The **RIB** (Routing Information Base) stores all routes learned from all protocols, tagged with their source protocol, administrative distance, and metric.
- The **FIB** (Forwarding Information Base) stores only the best routes selected from the RIB. The FIB is what the forwarding engine consults when processing a packet.

> **Design Decision**
>
> **Why separate RIB and FIB?**
> Real routers maintain this separation because the control plane (protocol daemons) and data plane (forwarding ASIC) operate independently. netsim mirrors this to ensure that protocol interactions (e.g., OSPF and BGP both advertising the same prefix) are resolved correctly via administrative distance, and to enable accurate modelling of RIB→FIB installation delays.

5.2 Dual-Index FIB

The FIB uses two co-maintained data structures:

1. A **binary trie** for longest-prefix-match (LPM) lookups. Implemented as `Vec<TrieNode>` where each `TrieNode` has `[Option<usize>; 2]` children and an optional `FibEntry`. LPM walks the trie bit by bit from MSB to LSB, tracking the last seen entry. Worst case: 32 steps for IPv4.
2. A **HashMap** (`HashMap<Ipv4Net, FibEntry>`) for $O(1)$ exact-match lookups during FIB update change detection.

To efficiently handle high-churn routing environments (e.g., Internet-scale BGP), netsim implements **Incremental FIB Updates**. Adding or removing a prefix from the RIB results in an $O(\log N)$ update to the Trie rather than an $O(N)$ full FIB rebuild. Atomic multi-prefix updates allow hundreds of route changes to be committed to the FIB in a single batch, drastically improving simulator convergence times on large topologies.

> **Design Decision**
>
> **Problem**: FIB updates must not trigger spurious convergence resets. If a protocol daemon re-announces an unchanged route, and the FIB's change flag is set, convergence detection restarts unnecessarily.
> **Options considered**: (1) Trie-only FIB with entry comparison on every update ($O(W)$ per lookup + $O(W)$ per update comparison). (2) HashMap-only FIB ($O(1)$ updates but $O(n)$ LPM via linear scan). (3) Dual-index with trie for LPM and HashMap for change detection.
> **Decision**: Dual-index. The HashMap enables $O(1)$ equality checking during updates (`old_entry == new_entry?`), and the trie provides $O(W)$ LPM for forwarding. Both are updated atomically via `replace_entries`.
> **Trade-off**: Double memory for the FIB. In practice, FIB memory is negligible compared to protocol state (BGP RIB, OSPF LSDB).

5.3 ECMP

When the RIB installs multiple equal-cost routes to the same prefix, the FIB stores all next-hops. At forwarding time, a flow-consistent hash selects among them:

Listing 2. ECMP hash computation.

```
fn ecmp_hash(src_ip: u32, dst_ip: u32, nhops: usize) -> usize {
```

```
    (src_ip.wrapping_add(dst_ip.rotate_left(16))) as usize % nhops
}
```

This ensures all packets of a given (src, dst) flow traverse the same path, while distributing distinct flows across available next-hops.

> **Warning**
>
> The ECMP hash does not include L4 ports. This means all traffic between a given (src_ip, dst_ip) pair follows the same path regardless of application. This is simpler than real router implementations (which typically hash on the 5-tuple) but sufficient for routing-level validation.

6 Protocol Implementations

netsim implements the following routing and infrastructure protocols. Each implementation targets RFC-correct behaviour at tick granularity, not vendor-specific quirks.

Table 5. Protocol coverage.

Protocol	RFCs	Scope
OSPF v2	2328 [14]	Areas, stub/NSSA, virtual links, SPF with incremental recalculation
IS-IS	ISO 10589	L1/L2, wide metrics, TLV extensibility
BGP-4	4271 [16]	iBGP, eBGP, route reflection, confederations, communities, AS-path filtering
BGP EVPN	7432 [17]	Type-2/3/5, RT import/export, ARP suppression
MPLS/LDP	5036 [1]	Label distribution, PHP, ECMP label stacking
RSVP-TE	3209	Explicit paths, bandwidth reservation, FRR
BFD	5880 [8]	Async mode, microsecond-granularity timers (mapped to ticks)
LACP	802.3ad	Fast/Slow timers (100/3000ms), LAG bundling, actor/partner state machines
LLDP	802.1AB	Periodic hellos, neighbour table, TLV parsing
VXLAN	7348 [10]	Encap/decap, head-end replication
RIPv2	2453 [11]	Periodic/triggered updates, split horizon, 15-hop metric limit, route poisoning
RIPng	2080 [12]	IPv6 distance-vector routing
STP	802.1D	Root bridge election, port roles (Root/Designated/Blocking), topology change detection
VRRP v2/v3	3768 [7], 5798 [15]	Master/Backup election, virtual IP, tracked interfaces, priority decrements, virtual MAC (00:00:5E:00:01:VRID)
DHCP Relay	2131 [5]	Broadcast-to-unicast conversion, multiple helper addresses, GIADDR population, Option 82 (Circuit ID, Remote ID)
OSPFv3	5340 [3]	IPv6 OSPF with link-local addressing
SRv6	8754 [6], 8986	SRH codec, SID table, End/End.X behaviours, headend encap (v2.2, in progress)

Note

All protocol timers (OSPF dead interval, BGP hold time, BFD detection multiplier, LACP fast/slow, RIP update interval, VRRP advertisement interval) are converted to integer tick counts at build time. A 10-second OSPF dead interval at 1 ms/tick resolution becomes 10,000 ticks.

6.1 OSPF

The OSPF implementation follows RFC 2328. Each router runs an independent OSPF instance with:

- Neighbour state machine (Down → Init → 2-Way → ExStart → Exchange → Loading → Full)
- Link-State Database (LSDB) synchronised via Database Description and LSA exchange
- SPF computation triggered on LSDB change (not on every tick)
- Area support: backbone (Area 0), stub, NSSA, virtual links

6.2 BGP

The BGP implementation supports both iBGP and eBGP with:

- Full FSM (Idle → Connect → OpenSent → OpenConfirm → Established)
- Route reflection (cluster-id, originator-id, no client-to-client)
- Best-path selection (13-step decision process matching Cisco IOS)
- Community, extended community, and large community support
- AS-path prepending, local-pref, MED
- Address families: IPv4 unicast, VPNv4, EVPN

6.3 MPLS and RSVP-TE

Label switching is implemented with:

- MPLS label stack with push/swap/pop operations
- LDP for automatic label distribution following the IGP topology
- RSVP-TE for explicit label-switched paths with bandwidth constraints
- Penultimate hop popping (PHP)
- L3VPN with VRF import/export using VPNv4 route targets

6.4 BFD, LACP, LLDP

Infrastructure protocols:

- **BFD**: Bidirectional Forwarding Detection for fast failure detection. BFD session state changes trigger IGP/BGP convergence immediately rather than waiting for hold-timer expiry.
- **LACP**: Link Aggregation Control Protocol with tick-aligned timers (Fast: 100/300 ticks, Slow: 3000/9000 ticks). LAG bundles present as a single logical interface to the routing protocols.
- **LLDP**: Periodic neighbour discovery. Processed by `tick_lldp()` in `tick_control()`. Excluded from quiescence checks because it never stops sending.

6.5 RIP

netsim implements both RIPv2 (RFC 2453 [11], IPv4) and RIPng (RFC 2080 [12], IPv6). Key features:

- **Hop-count metric**: 15-hop limit with 16 as infinity (unreachable). Split horizon with poisoned reverse prevents count-to-infinity.
- **Periodic updates**: Default 30-second interval with triggered updates on topology changes for faster convergence.
- **Interface activation**: Enabled per-interface via `rip: true` in the YAML topology.

Design Decision

Why include RIP in a modern simulator?
RIP is rarely deployed in new networks, but it remains valuable for three reasons: (1) it is the simplest routing protocol to understand, making netsim useful for educational settings; (2) legacy networks still run RIP in stub areas; (3) RIP's distance-vector behaviour (count-to-infinity, split horizon) exercises different convergence dynamics than OSPF's link-state SPF, broadening the convergence predicate's test coverage.

6.6 STP

The Spanning Tree Protocol implementation provides L2 loop prevention for Switch devices:

- **Root bridge election**: Lowest bridge ID wins. BPDUs propagate root bridge identity across the switched domain.
- **Port roles**: Root, Designated, and Blocking roles are computed from received BPDUs.
- **State transitions**: Disabled → Listening → Learning → Forwarding, with configurable forward delay.
- **Topology change detection**: TCN BPDUs trigger MAC table flushes to prevent stale forwarding.

6.7 VRRP

Virtual Router Redundancy Protocol (RFC 3768 [7] for v2, RFC 5798 [15] for v3) provides first-hop gateway redundancy:

- **Master/Backup election**: Priority-based (default 100, range 1–254); highest priority becomes Master.
- **Virtual IP**: A shared IP address floats between Master and Backup routers. Hosts configure this as their default gateway.
- **Tracked interfaces**: Interface state changes adjust VRRP priority dynamically (configurable decrements), enabling upstream-aware failover.
- **Preemption**: Configurable—a higher-priority router can reclaim Master status when it recovers.
- **Virtual MAC**: Standard 00:00:5E:00:01:*VRID* address, avoiding gratuitous ARP storms on failover.
- **IPv4 and IPv6**: Both address families supported via VRRPv3.

6.8 DHCP Relay

The DHCP Relay Agent (RFC 2131 [5]) enables hosts in remote subnets to obtain addresses from a centralised DHCP server:

- **Broadcast-to-unicast**: Converts DHCP Discover/Request broadcasts into unicast packets to configured helper addresses.
- **Multiple servers**: Each interface can list multiple `helper_addresses`, forwarding to all.
- **GIADDR**: The relay populates the Gateway IP Address field so the server selects the correct subnet pool.
- **Option 82**: Appends Relay Agent Information (Circuit ID and Remote ID sub-options) for downstream accounting. Pre-existing Option 82 from untrusted sources is dropped for security.
- **Statistics**: Per-interface counters for requests forwarded, replies forwarded, and drops.

Design Decision

Why DHCP Relay without a full DHCP server?
netsim models the *relay* path, not the server's address pool logic. This is sufficient to validate enterprise campus designs where DHCP Relay traversal through firewalls and VRFs is the common failure mode. A full DHCP server would add complexity without improving the relay-path validation that operators care about.

6.9 SRv6 (Segment Routing over IPv6)

netsim implements the SRv6 data plane per RFC 8754 [6] (Segment Routing Header) and RFC 8986 (SRv6 Network Programming). This is an active development area (milestone v2.2).

6.9.1 Segment Routing Header (SRH)

The SRH is an IPv6 Routing Header with `routing_type = 4`. The implementation provides a full codec (`src/srv6/srh.rs`):

```
pub struct ParsedSrh {
    pub next_header: u8,       // inner payload (IPv4=4, IPv6=41)
    pub hdr_ext_len: u8,       // in 8-octet units (excl. first 8)
    pub routing_type: u8,      // must be 4
    pub segments_left: u8,
    pub last_entry: u8,
    pub flags: u8,
    pub tag: u16,
    pub segments: Vec<Ipv6Address>,  // on-wire order (reversed)
}
```

`serialize_srh()` accepts segments in forward order (first hop → last hop) and stores them *reversed* on the wire per RFC 8754. `deserialize_srh()` validates routing type, `segments_left` ≤ `last_entry`, and buffer bounds before parsing.

6.9.2 SID Table and Endpoint Behaviours

Each router maintains a SID table keyed by IPv6 address:

```
srv6_sid_table: BTreeMap<Ipv6Address, Srv6SidEntry>
```

Two endpoint behaviours are implemented:

- **End** (RFC 8986 §4.1): Transit endpoint. Decrements `segments_left`, updates the outer IPv6 DA to the next segment, and re-submits to IPv6 forwarding. The End SID is auto-derived from the locator prefix and the last octet of the router ID.
- **End.X** (RFC 8986 §4.2): Layer-3 cross-connect. Identical to End but overrides the egress interface and next-hop link-local address, bypassing the FIB lookup. Useful for traffic engineering.

6.9.3 Headend Encapsulation

Static headend encapsulation is configured via encap rules:

```
pub struct Srv6EncapRule {
    pub dst: IpNet,               // match IPv4 or IPv6 prefixes
    pub segments: Vec<Ipv6Address>, // segment list (forward order)
}
```

Rules are sorted at configuration time by longest prefix first (then by prefix string, then segment list) for deterministic matching. When a rule matches, the headend wraps the inner packet in an outer IPv6 frame with an SRH, setting the outer destination address to the first active segment.

6.9.4 YAML Configuration

SRv6 is configured per-device in the topology file:

```yaml
devices:
  - name: R1
    router_id: 1.1.1.1
    srv6:
      locator: "2001:db8:1::/48"
      sids:
        - sid: "2001:db8:1:100::"
          behavior: end.x
          interface: eth0
      encap:
        - dst: "10.0.3.0/24"
          segments:
            - "2001:db8:2::1"
            - "2001:db8:3::1"
```

Note

Transit forwarding is SRH-blind: routers that are not the target of a SID forward on the outer IPv6 DA only. SRH processing occurs only when the DA matches a local SID table entry. This matches real router behaviour and means that SRv6 does not require any changes to existing IPv6 FIB lookup logic.

Warning

SRv6 is under active development (milestone v2.2, Phase 125–126). The SRH codec and SID table are complete. Endpoint behaviour wiring into the forwarding pipeline, `show srv6 sid` commands, and service SID types (End.DT4, End.DT6) are planned for upcoming phases. Flavours (PSP, USP, USD) and SRv6 Policy with colour/endpoint steering are scoped for Phase 126.

7 Declarative Topology & Chaos Engineering

7.1 YAML Topology Format

A netsim topology is a single YAML file with the following top-level sections:

Listing 3. Top-level YAML structure.

```yaml
# Metadata
name: "dc-fabric-486"
tick_ms: 1  # optional, default 1ms

# Network elements
devices: [...]
links: [...]

# Protocol configuration
ospf: {...}    # optional
bgp: {...}     # optional
# ... other protocols

# Operational
traffic: [...]             # optional
impairment_profiles: [...]  # optional
failure_patterns: [...]     # optional
events: [...]              # optional
assertions: [...]          # optional
```

7.2 Pattern-Based Selectors

Rather than configuring each device individually, netsim supports selectors that match groups of devices or links:

Listing 4. Pattern-based selectors.

```
impairment_profiles:
  - name: wan-latency
    selector:
      by_tier: [core, edge]   # match by device tier
    latency_ms: 10
    loss_percent: 0.01

  - name: access-links
    selector:
      name_pattern: "access-*"  # match by name glob
    latency_ms: 2

  - name: geographic
    selector:
      by_distance: true    # compute from coordinates
    latency_model: speed_of_light
```

Design Decision

Why selectors instead of per-device configuration?
A 486-device topology would require 630+ link configurations without selectors. Pattern-based selectors reduce this to a handful of profiles, making topologies readable and maintainable. The trade-off is that selectors can match unintended devices if patterns are too broad—netsim logs which devices each selector matched during the build stage to aid debugging.

7.3 Declarative Assertions

To enable automated CI/CD validation, netsim supports declarative assertions that are evaluated after convergence. This allows operators to encode their network intent directly in the topology file:

Listing 5. Declarative assertions.

```
assertions:
  # End-to-end reachability intent
  - type: reachability
    source: host-db
    destination: 10.0.1.1
    expected: true

  # Protocol state invariant
  - type: ospf_neighbor_state
    router: core-spine1
    interface: eth0
    expected_state: Full
```

If any assertion fails, netsim exits with a non-zero status code and provides a detailed diagnostic report, making it ideal for integration into automated infrastructure pipelines.

7.4 Failure Patterns

netsim supports four failure pattern types for chaos engineering [2]:

Table 6. Failure pattern types.

Type	Behaviour
`probabilistic`	At each `check_interval`, fail with probability p. Recovery after n ticks.
`periodic`	Deterministic on/off cycling with configurable up/down durations.
`flapping`	Rapid state oscillation (up→down→up) to stress convergence.
`cascade`	Dependency-triggered chain failures: when a primary link fails, dependent links/devices also fail.

Each `FailurePattern` is expanded into a sequence of `ScheduledEvents` at build time. This cleanly separates *what should happen* (the pattern) from *when it happens* (the schedule), and ensures deterministic replay.

8 EVPN/VXLAN at Simulation Abstraction

Data centre fabric validation is a primary use case for netsim. This section documents the EVPN/VXLAN implementation in detail.

8.1 Problem

Validating a leaf-spine EVPN/VXLAN fabric typically requires Containerlab with FRR or EOS containers. This consumes gigabytes of RAM (each FRR container runs a full Linux userspace), takes minutes to boot, and produces non-deterministic convergence behaviour. An operator wanting to validate "does my EVPN fabric converge correctly after a spine failure?" should not need to wait 3 minutes and hope for consistent results.

8.2 Unified Router as L2+L3 Device

Unlike many simulators that model switches and routers as separate device types, netsim's `Router` serves as a unified L2+L3 device. Each Router can hold zero or more *bridge domains*, making it capable of acting as a pure L3 router, a VTEP with bridging, or an IRB (Integrated Routing and Bridging) gateway—all within the same device abstraction.

```
// Router L2 state (router.rs)
bridge_domains: BTreeMap<u32, BridgeDomain>,  // keyed by VNI
interface_to_bd: BTreeMap<InterfaceId, u32>,  // interface -> VNI
```

The L2/L3 forwarding decision is interface-driven: if an ingress interface is mapped to a bridge domain via `interface_to_bd`, the frame enters `process_frame_l2`. Otherwise, it proceeds to IPv4/IPv6 L3 forwarding. Within the L2 path, an IRB check fires first: if the frame's destination MAC matches the bridge domain's gateway MAC, the frame is routed into the associated VRF, bypassing L2 bridging entirely.

8.3 Bridge Domain and FDB

Each bridge domain maintains an independent forwarding database:

```
pub struct BridgeDomain {
    vni: u32,
    fdb: BTreeMap<MacAddress, FdbEntry>,
    local_interfaces: BTreeSet<InterfaceId>,
```

```
    remote_vteps: BTreeSet<Ipv4Address>,  // BUM flood list
    mac_timeout: u64,                     // default 300s
    esi_local_interfaces: BTreeMap<EthernetSegmentId,
                                   BTreeSet<InterfaceId>>,
}
```

FDB entries have three types with a strict override hierarchy:

1. **Static**: Manually configured, never ages, highest priority. Cannot be overridden by dynamic learning.
2. **EVPN**: Control-plane learned via BGP Type-2 routes. Never ages (removed only on BGP withdrawal). Carries MAC mobility sequence numbers for move detection.
3. **Dynamic**: Data-plane learned on local ingress frames. Ages out after `mac_timeout` ticks. Cannot overwrite static or EVPN entries.

`BTreeMap` is used instead of `HashMap` throughout for deterministic iteration order—critical for reproducible BUM flooding and consistent `show` command output.

> **Note**
>
> MAC ageing events deliberately do *not* set `fdb_changed_this_tick`. Ageing is routine housekeeping, not a convergence-relevant state change. Only new learning events and control-plane updates affect the convergence predicate.

8.4 BGP EVPN Control Plane

netsim implements all five EVPN route types defined in RFC 7432 [17] and RFC 9136:

Table 7. EVPN NLRI types.

Type	Name	Purpose
1	EAD (Ethernet Auto-Discovery)	Mass withdrawal on ES failure; aliasing for multi-homed load balancing. Two sub-types: per-ES (`ethernet_tag=0`) and per-EVI (`ethernet_tag=VNI`).
2	MAC/IP Advertisement	Advertises host MAC and optional IP binding. Used for remote MAC learning and ARP suppression.
3	Inclusive Multicast Ethernet Tag (IMET)	Establishes the VTEP flood list for BUM traffic replication per VNI.
4	Ethernet Segment Route	PE discovery for multi-homing. Used to build the candidate list for DF election.
5	IP Prefix Route	Advertises IP prefixes for inter-subnet routing via the VXLAN fabric (symmetric IRB).

8.5 VXLAN Encapsulation and Decapsulation

The VXLAN data plane handles encapsulation and decapsulation in the Router's forwarding pipeline:

Encapsulation path. When a bridge domain needs to forward a frame to a remote VTEP:

1. **MTU check**: Verify that the inner frame plus 42 bytes of VXLAN/UDP/IP overhead fits within the egress interface MTU (default 1500). Drop with counter if exceeded.
2. **UDP source port**: Computed from a hash of the inner frame's source and destination MACs (mapped to range 49152–65535). This provides ECMP entropy for the underlay, per RFC 7348.
3. **Inner frame encoding**: A compact binary format with magic bytes (`NSFR`), version, MAC addresses, EtherType, TTL, and optional L3/L4 metadata.
4. **Multi-homing metadata**: When the frame originates from a multi-homed Ethernet Segment, a 15-byte metadata block (`NSMH` magic + 10-byte ESI) is appended for split-horizon filtering at the remote VTEP.
5. **Underlay forwarding**: The outer IP packet is forwarded via the underlay FIB using the destination VTEP IP.

Decapsulation path. On receiving a VXLAN packet (UDP dst port 4789):

1. Parse VXLAN header, extract VNI.
2. Look up the bridge domain by VNI.
3. Detect and strip optional multi-homing metadata.
4. Decode the inner frame.
5. Call `process_frame_l2` with `from_vxlan=true` and the ingress ESI (if present).

The `from_vxlan=true` flag is the critical hairpin prevention mechanism: it prevents re-replication of decapsulated frames back to remote VTEPs, which would cause infinite flooding loops.

8.6 MAC Mobility

MAC mobility detection follows RFC 7432 Section 15.2. Each EVPN Type-2 route carries a MAC Mobility extended community with a `sequence` counter and a `sticky` flag:

- When a MAC is learned on a new interface, the local VTEP reads the current sequence number via `get_mac_sequence(vni, mac)`, increments it, and re-originates the Type-2 route.
- On receiving a remote Type-2 route, the selection rule is: (1) higher sequence wins, (2) on tie, lower originating IP wins for stability.
- MAC sequences are tracked in a `BTreeMap<(u32, MacAddress), u32>` inside the BGP instance, keyed by (VNI, MAC).

8.7 ARP Suppression

When a host sends an ARP request, the local VTEP checks its EVPN Type-2 routes for a matching IP→MAC binding. If found, the VTEP replies directly without flooding the ARP request across the VXLAN fabric. This reduces BUM traffic significantly in large fabrics. ARP suppression is enabled by default in `BridgeDomainConfig` and can be disabled per bridge domain.

8.8 Snapshot-Then-Apply Synchronisation

Design Decision

Problem: How should EVPN Loc-RIBs be synchronised across routers in a tick-based simulation?
Options considered:

1. **Per-message queuing**: Model individual BGP UPDATE messages with per-session send/receive queues. High fidelity but complex (O(sessions × routes) queue operations per tick) and introduces ordering-dependent non-determinism.
2. **Snapshot-then-apply**: At each tick, snapshot all EVPN Loc-RIBs, then have each router apply remote updates from the snapshot.

Decision: Snapshot-then-apply (Phase P2.5 in the tick pipeline). This preserves `peer_router_id` metadata to prevent route-reflector-induced flapping while avoiding per-message queue complexity.
Trade-off: Cannot model BGP UPDATE message ordering bugs or per-session backpressure. This is an acceptable trade-off for a tool that targets configuration validation, not BGP implementation testing.

The synchronisation pass performs split-horizon filtering: routes are not advertised back to the peer from which they were learned, checked by comparing `route.peer_router_id` against the destination peer's router ID. Importantly, `peer_router_id` is preserved from the original route—not replaced with the reflecting router's ID—to prevent EVPN flap in route-reflector topologies.

8.9 EVPN Multi-Homing

netsim supports EVPN all-active multi-homing for redundant host connectivity, implementing the full Type-1/Type-4 route exchange:

- **Ethernet Segment Identifier (ESI)**: A 10-byte identifier that groups interfaces on different VTEPs connecting to the same multi-homed host or switch. ESI labels are derived deterministically via FNV-1a hash of (ESI bytes ‖ router_id bytes), clamped to the 20-bit MPLS label range.
- **Designated Forwarder (DF) election**: Uses RFC 7432 service-carving with modulo-based selection: `df = candidates[vni % len(candidates)]`. Candidates are discovered via Type-4 Ethernet Segment routes and sorted by VTEP IP for determinism.
- **Split-horizon filtering**: Two levels of loop prevention:
  1. *BUM flood-time gating*: When flooding a VXLAN-decapped BUM frame to a local interface with an ESI, check `is_local_df`. If not DF for this VNI, skip that interface.
  2. *Unicast split-horizon*: When forwarding a VXLAN-decapped unicast to a local interface, if the ingress ESI (carried in the multi-homing metadata block) matches that interface's ESI, drop the frame.
- **Aliasing**: When a received Type-2 route carries a non-zero ESI that matches a local Ethernet Segment, the FDB entry is installed as `Local(interface)` instead of `RemoteVtep`. When the ESI is known but the local interface is not present, `select_aliasing_vtep()` performs deterministic load balancing across all known PEs for that ESI using a hash of (VNI, MAC).

9 CLI Tool & Interactive Daemon

The `netsim` CLI provides both batch execution (run a topology, collect results) and interactive operation (attach to a live simulation, inspect state, inject failures). This section documents both modes.

9.1 CLI Subcommands

The CLI is the primary interface to netsim. It provides five categories of subcommands:

Table 8. CLI subcommand categories.

Category	Commands
Simulation	`run <topology>` — execute a simulation to completion. Options: `--output`, `--format [ascii|json]`, `--max-ticks`, `--convergence-threshold`, `--pcap`, `--scenario`, `--verbose`. `validate <topology>` — check YAML syntax without executing. `example <name>` — print a built-in example topology (`simple`, `ospf-triangle`, `data-center`, `wan-mesh`, `enterprise-campus`, `service-provider`, etc.).
Daemon	`daemon start <topology>` — start a background daemon. Options: `--name`, `--tick-interval [50ms|100ms|1s]`, `--log-level`. `daemon stop <name>` — stop a daemon (`--rm-log` to clean up). `daemon status <name>` — query daemon metrics. `daemon list [--clean]` — list running daemons. `daemon log <name> <preset>` — change log verbosity at runtime.
Interactive	`netsim` (no args) — launch TUI selector (daemon → device → REPL). `attach <daemon> <node>` — open interactive REPL on a device. `exec <daemon> <node> "<cmd>"` — execute a single command.
Analysis	`routing-matrix <daemon>` — extract full routing matrix as JSON. `diff <before.json> <after.json>` — compare two routing matrices. `capacity <topology> <matrix>` — analyse link utilisation. `trace <daemon> <src> <dst>` — trace forwarding path hop-by-hop. `fib <daemon>` — export FIB in netauto/fib/v1.0 schema.

Example: Batch simulation in a CI pipeline

```
# Run topology, output JSON, fail on assertion failures
netsim run topology.yaml --format json --output results.json

# Extract routing matrix from a running daemon
```

```
netsim routing-matrix my-fabric --output before.json

# Compare routing state before and after a change
netsim diff before.json after.json --format ascii
```

9.2 Daemon Architecture

The daemon runs a simulation as a long-lived background process, enabling multiple users to attach, inspect, and inject commands concurrently. This transforms netsim from a batch-mode tool into an interactive, always-on simulation environment — akin to having a virtual lab where engineers can "walk up" to any device at any time.

9.2.1 Threading Model

Design Decision

Problem: The simulation engine is single-threaded and `!Send` (it holds non-thread-safe state such as protocol FSMs, FIBs with interior references, and mutable device vectors). But gRPC requires an async runtime. How to bridge the two?
Options considered:

1. **Async engine**: Rewrite the engine to be async-compatible. Invasive; every protocol state machine would need refactoring, and `.await` points would complicate the deterministic tick loop.
2. **Separate OS thread**: Run the engine on a dedicated OS thread; communicate via channels.
3. **Mutex wrapping**: Wrap the engine in `Arc<Mutex<Engine>>` and acquire the lock from async handlers. Would require the engine to be `Send`, and creates lock contention between gRPC handlers and the tick loop.

Decision: Separate OS thread. The engine runs its tick loop on a dedicated thread. A Tokio multi-threaded runtime on the main thread hosts the gRPC server. Communication uses `std::sync::mpsc` channels: commands flow from gRPC handlers to the engine thread, responses flow back via per-request `tokio::sync::mpsc` channels.
Trade-off: Requires channel-based indirection for every command. In practice, the overhead is negligible compared to tick processing time. The benefit is that the engine thread never blocks on I/O, and the gRPC server can handle arbitrary concurrent connections without starving the simulation.

The resulting architecture has three layers of concurrency:

```
+-------------------------------------------+
|  CLI Client (netsim exec / attach)        |
|  Connects via UDS to gRPC server          |
+-------------------------------------------+
        |           |           |
        v           v           v
+-------------------------------------------+
|  gRPC Server (Tokio async runtime)        |
|  NetsimDaemon service implementation      |
|  Handles Exec, Attach, Status, Stop,      |
|  SetLogPreset RPCs concurrently           |
+-------------------------------------------+
        |  std::sync::mpsc (commands)
        v
+-------------------------------------------+
|  Engine Thread (dedicated OS thread)      |
|  Tick loop: process commands -> tick       |
|  Shared atomics: tick_count, converged,   |
|  average_tick_ms (lock-free reads)        |
```

```
+-----------------------------------------+
```

Shared state between the gRPC server and the engine thread is communicated via two mechanisms:

1. **Command channel** (`std::sync::mpsc`): gRPC handlers send `EngineCommand` variants to the engine thread. Each command carries a per-request response channel for sending results back asynchronously.
2. **Atomic scalars** (`Arc<AtomicU64>`, `Arc<AtomicBool>`): The engine writes tick count, convergence status, and average tick duration every tick. The `Status` RPC reads these with `Ordering::Relaxed`, providing a lock-free path for metrics without blocking the tick loop.

9.2.2 Command Pipeline

The engine processes commands at tick boundaries, ensuring consistent state. The `EngineCommand` enum defines three variants:

```rust
pub enum EngineCommand {
    Exec {
        node_name: String,
        command: String,
        response_tx: Option<mpsc::Sender<(String, i32)>>,
    },
    GetRoutingMatrix {
        response_tx: mpsc::Sender<String>,
    },
    Attach {
        node_name: String,
        input_rx: Receiver<String>,
        output_tx: Sender<(String, i32)>,
    },
}
```

At the start of each tick (phase P0 of the tick pipeline), the engine calls `try_recv()` in a non-blocking loop to drain all pending commands. This guarantees that:

- Commands never observe mid-tick intermediate state — they always see the state at a tick boundary.
- The tick loop is never blocked waiting for commands — if no commands are pending, `try_recv()` returns immediately.
- Two users issuing commands simultaneously see consistent results because command delivery is serialised within the single-threaded tick loop.

9.2.3 Interactive Attach Sessions

The `Attach` RPC creates a persistent bidirectional session between a CLI user and a simulated device. Unlike `Exec` (fire-and-forget), attach sessions persist across ticks:

1. The gRPC handler sends an `EngineCommand::Attach` with Tokio channel endpoints to the engine thread.
2. The engine stores the session in an internal `attach_sessions: Vec<AttachSession>`.
3. Every tick, `process_attach_sessions()` drains each session's `input_rx` for pending commands, executes them with full `&mut self` access, and sends results to `output_tx`.
4. Sessions are cleaned up when the user sends `exit`/`quit`, presses Ctrl-D, or the channel disconnects.

On the client side, the CLI runs a `rustyline`-based REPL on a blocking thread with full tab completion, command hints, and persistent history. This creates a workflow where a user can type IOS-style

commands interactively while the simulation continues ticking in the background — commands are queued and executed at the next tick boundary.

9.2.4 Process Lifecycle

The `daemon start` command forks a background process using the `daemonize` crate. The startup sequence is carefully ordered:

1. **Parse CLI arguments**: Name, socket path, PID file, topology file, tick interval, log level.
2. **Daemonize**: Fork via `daemonize`, redirect stdout/stderr to log file, write PID file. This happens *before* building the Tokio runtime to isolate file descriptors.
3. **Load topology**: Parse YAML, build the Engine with command channel (`engine.create_command_channel()`).
4. **Spawn engine thread**: Start the tick loop on a dedicated OS thread (the engine is `!Send`, so it must be created and used on the same thread).
5. **Start gRPC server**: Build Tokio multi-threaded runtime, bind to `.netsim/<name>.sock`, register SIGTERM handler for graceful shutdown.

The daemon writes three files to a `.netsim/` directory:

```
.netsim/
  <name>.sock    # Unix Domain Socket for gRPC
  <name>.pid     # Process ID file
  <name>.log     # Structured log with file appender
```

Using Unix Domain Sockets (UDS) instead of TCP ports avoids port conflicts when running multiple daemons on the same machine, provides implicit access control via filesystem permissions, and eliminates the TCP overhead for local communication.

9.2.5 Daemon Management

The CLI provides a full lifecycle management interface:

Table 9. Daemon management commands.

Command	Description
`daemon start <name> <topo>`	Fork daemon, load topology, begin ticking
`daemon stop <name>`	Graceful shutdown via `Stop` RPC or SIGTERM
`daemon status <name>`	Query live metrics (ticks, convergence, devices)
`daemon list [--clean]`	Discover all running daemons from `.netsim/*.sock`; `--clean` removes stale entries
`daemon restart <name>`	Stop then start with same configuration
`daemon log <name> <preset>`	Change log verbosity at runtime

9.2.6 Dynamic Log Control

Log verbosity can be changed at runtime via `daemon log <name> <preset>` without restarting the simulation. Three presets are available:

Table 10. Log presets.

Preset	Filter
`normal`	`info` — standard operational output
`protocol_debug`	OSPF, BGP, IS-IS, BFD at `debug` level; other modules at `info`
`full_trace`	`trace` — all modules, maximum verbosity

This is implemented via the `SetLogPreset` gRPC RPC, which reloads the `tracing` subscriber filter on-the-fly using `tracing-subscriber`'s reload layer. Because the filter swap is atomic, there is no window where log events are dropped or duplicated.

9.3 gRPC Service Interface

The daemon exposes five RPCs via Protocol Buffers:

Table 11. gRPC service definition (`proto/netsim.proto`).

RPC	Type	Description
`Exec`	Server stream	Execute a command, stream output lines back
`Attach`	Bidi stream	Interactive REPL session with command/response streaming
`Status`	Unary	Return PID, uptime, tick count, convergence status, average tick duration, device list
`Stop`	Unary	Graceful shutdown
`SetLogPreset`	Unary	Change log verbosity at runtime

The `Status` RPC returns live metrics from shared atomics (tick counter, convergence flag, average tick duration), avoiding lock contention with the engine thread. The response includes a `DeviceInfo` array listing every device by name, type, and interfaces—enabling the TUI to build its device selection menu without parsing the topology file.

9.4 IOS-Style Command Tree

The interactive REPL implements a `BTreeMap`-based command vocabulary tree that provides tab completion and prefix matching.

- **Prefix matching**: Partial commands are expanded to their unique completion (`sh ip ro` → `show ip route`).
- **Ambiguity detection**: When a prefix matches multiple commands (`sh i` → `show interface` or `show ip` or `show isis`), netsim reports an error with all matching completions.
- **Dynamic completion**: Interface names (`eth0`, `eth1`, …) are completed dynamically based on the attached device's actual interfaces.
- **Hints**: The completer shows the first available completion as a grey hint while typing, implemented via the `rustyline` `Hinter` trait.

Design Decision

Why IOS-style CLI in a simulator?
Network engineers have decades of muscle memory for IOS commands. Making the simulator's interactive interface match that mental model reduces the learning curve to zero for the inspection workflow. An engineer who can type `show ip route` on a production router can type the same command in netsim and get comparable output.

9.5 Available Commands

The REPL supports over 50 commands across four categories:

Table 12. Command categories in the interactive REPL.

Category	Commands
Show (30+ variants)	`show ip route [vrf NAME]`, `show ip dhcp relay`, `show ipv6 [route|neighbors]`, `show interfaces`, `show arp`, `show ospf [neighbors|database]`, `show ospfv3 [neighbors|database|interface|route]`, `show bgp [summary|neighbors|vpn]`, `show bgp ipv6 [unicast|summary]`, `show bgp evpn`, `show evpn ethernet-segment`, `show isis [neighbors|database|spf|route|interface]`, `show rip [database|neighbors]`, `show stp [bridge|port]`, `show vrrp [brief|detail]`, `show vxlan [vtep|vni]`, `show mpls [forwarding|lfib]`, `show ldp [bindings|neighbors]`, `show rsvp tunnels`, `show bfd [sessions]`, `show lacp [details]`, `show lldp neighbors`, `show vrf [all|detail|NAME]`, `show sr`, `show gre [tunnels]`, `show bmp`, `show traffic`, `show routing-matrix`, `show trace <destination>`
Diagnostic	`ping <target> [-c COUNT]`, `traceroute <target> [-m MAX_TTL]`, `dig <hostname> [@server]`, `iperf <target> [-d DURATION] [-i INTERVAL]`
Configuration	`route add <prefix> <nexthop>`, `interface [shutdown|no shutdown]`, `mpls [lfib add|ftn add]`, `bgp vpn-originate`, `gre tunnel add`
Session	`help`, `exit`, `quit`

9.6 Ping and Traceroute

The `ping` and `traceroute` commands are notable because they drive *re-entrant simulation stepping*. When a user types `ping 10.0.1.1`, netsim does not simply check reachability in the current routing

table—it injects ICMP packets into the simulation and advances the tick loop until replies arrive or a timeout is reached.

- **ping**: Sends ICMP Echo Requests from the attached host, advances the simulation tick by tick, collects Echo Replies, and reports sent/received/lost counts with min/max/avg RTT. Supports `-c|--count` for repeat count (default 4). Only available on Host devices.
- **traceroute**: Sends probes with incrementing TTL, advances the simulation, collects ICMP Time Exceeded and Echo Reply messages, and displays hop-by-hop paths with latency. Supports `-m|--max-ttl` (default 30). Ticks up to 50,000 ticks per probe.

> **Note**
>
> Both commands support device hostname resolution in addition to IP addresses. `ping r1` resolves the hostname `r1` to its first interface IP via an internal device registry, providing a convenient shorthand.

9.7 Diagnostic Tools: dig and iperf

Two additional diagnostic commands extend the operator toolkit:

- **dig**: DNS query tool that resolves hostnames against a configured DNS server, reports query time in milliseconds, and displays response status codes (NOERROR, NXDOMAIN, timeout). Uses a state-machine implementation that drives simulation stepping.
- **iperf**: Throughput measurement tool that generates constant-bit-rate traffic between two endpoints. Reports per-interval statistics (bytes sent, bits/second) and total flow summary. Configurable duration, reporting interval, packets/second, and packet size.

9.8 Static Analysis: Path Tracing, Diffing, and Capacity

netsim provides several static analysis tools for "Day 2" operations, enabling operators to analyse routing state without injecting packets:

- **Path Tracing**: Computes hop-by-hop forwarding paths by walking the FIB/LFIB. Identifies routing loops, blackholes, and asymmetric routing. Traces handle complex overlays (MPLS, VXLAN) and recursive next-hops natively.
- **Topology Diffing**: Compares two routing matrices (e.g., before and after a planned configuration change) to quantify the blast radius. It highlights exactly which subnets shifted paths, providing empirical proof of change impact.
- **Capacity Planning**: Predicts network congestion by overlaying a traffic matrix onto the current routing paths. Flags links that exceed their configured bandwidth, helping to catch bottlenecks before they cause packet loss.

> **Design Decision**
>
> **Why static path tracing instead of packet-based traceroute?**
> Traceroute (above) injects real ICMP packets and advances the simulation — it shows the *actual* forwarding path including ECMP hash selection. Path tracing walks the FIB *without* advancing the simulation — it shows *all possible* paths and detects anomalies like loops and blackholes that traceroute might miss (a packet would simply be dropped). The two tools are complementary: traceroute for specific-flow validation, path tracing for exhaustive analysis.

9.9 JSON Structured Output

netsim provides structured JSON output at two levels:

Simulation-level JSON. `netsim run --format json` produces a `SimulationResult` object containing:

```
{
  "simulation": {
    "name": "dc-fabric",
    "converged": true,
    "converged_at_tick": 847,
    "final_tick": 1347
  },
  "commands": [
    {"tick": 947, "device": "PE1",
     "command": "show bgp summary", "output": "...",
     "exit_code": 0}
  ],
  "assertions": [
    {"message": "h1 -> h2 reachable", "success": true}
  ],
  "errors": []
}
```

Command-level JSON. Individual `show` commands support `--json` or a trailing `json` keyword for structured output. This enables programmatic integration—a CI script can parse `show ipv6 route --json` output directly rather than screen-scraping ASCII tables.

> **Warning**
>
> JSON output is not yet uniformly implemented across all `show` commands. `show evpn ethernet-segment json` and `show ipv6 [route|neighbors] --json` have full JSON support. Other commands currently render ASCII tables via `.to_table()`; JSON variants are being added incrementally.

9.10 Command Determinism

Commands received via gRPC are buffered and drained at phase P0 of the tick pipeline. This ensures that the simulation state observed by a command is always the state at a tick boundary, never a mid-tick intermediate state. Two users attaching simultaneously see consistent results because command delivery is serialised within the tick loop.

9.11 TUI Device Selector

Running `netsim` with no arguments launches a terminal user interface that presents a nested menu:

1. **Daemon selection**: Lists all running daemons (discovered from `.netsim/*.sock` files) with their status.
2. **Device selection**: Queries the selected daemon's `Status` RPC for its device list, grouped by type (routers, hosts, switches).
3. **REPL**: Opens an interactive session on the selected device with full tab completion.

This provides a zero-configuration discovery workflow: start a daemon, then run `netsim` and navigate to any device without remembering daemon names or device identifiers.

10 Python Bindings

netsim provides native Python bindings via PyO3, exposing the simulation engine directly to Python without serialisation overhead or network round-trips. Unlike the daemon mode (Section 9), which

communicates via gRPC over Unix Domain Sockets, the Python bindings embed the Rust engine in-process — Python calls invoke Rust functions directly through the CPython C API.

Design Decision

Two integration paths: netsim offers two distinct ways to interact with the simulation programmatically:

1. **Daemon + gRPC**: A long-running background process with CLI or custom gRPC clients. Best for interactive exploration, multi-user access, and operational workflows.
2. **Python bindings (PyO3)**: Direct in-process embedding. Best for scripted test automation, CI/CD pipelines, custom analysis, and integration with Python data science tooling (pandas, matplotlib, pytest).

Trade-off: The daemon supports concurrent multi-user access and persistent sessions. The Python bindings are single-threaded (matching the engine's `!Send` constraint) but offer zero-copy access to engine state and the full expressiveness of Python for orchestration.

10.1 Architecture

The binding is implemented as a Rust crate (`crates/netsim-py/`) that compiles to a CPython extension module via `maturin`. The module exports six PyO3 classes:

Table 13. Python binding classes.

Class	Rust Wrapper	Purpose
`Simulation`	`Engine`	Topology construction, simulation control, event scheduling
`SimulationIterator`	Engine tick iterator	Pythonic `for tick in sim.run_iter()`
`PyDevice`	Device trait object	Per-device state access (counters, interfaces, routing)
`RoutingTable`	FIB accessor	Route lookup and enumeration
`Counters`	Interface counters	Packet/byte statistics (attribute and dict access)
`InterfaceInfo`	Interface state	Name, MAC, IP, up/down, per-interface counters

All classes that hold engine references are marked `#[pyclass(unsendable)]`, which tells PyO3 to enforce single-threaded access. This mirrors the engine's `!Send` constraint at the Python level — attempting to share a `Simulation` across Python threads raises a `RuntimeError`.

10.2 Topology Construction API

The `Simulation` class provides a fluent builder API with method chaining for ergonomic topology construction:

Listing 6. Building a triangle topology with OSPF.

```
from netsim_py import Simulation

with Simulation() as sim:
    # Build topology with method chaining
```

```
    sim.add_router("r1").add_router("r2").add_router("r3")

    # Add interfaces with IP addressing
    sim.add_interface("r1", "eth0", ip="10.0.12.1/24")
    sim.add_interface("r1", "eth1", ip="10.0.13.1/24")
    sim.add_interface("r2", "eth0", ip="10.0.12.2/24")
    sim.add_interface("r2", "eth1", ip="10.0.23.2/24")
    sim.add_interface("r3", "eth0", ip="10.0.13.3/24")
    sim.add_interface("r3", "eth1", ip="10.0.23.3/24")

    # Wire devices together
    sim.add_link("r1", "r2", iface_a="eth0", iface_b="eth0")
    sim.add_link("r1", "r3", iface_a="eth1", iface_b="eth0")
    sim.add_link("r2", "r3", iface_a="eth1", iface_b="eth1")

    # Enable OSPF (auto-assigns router ID, enables on all
    # interfaces with IPs, default cost 10)
    sim.enable_ospf("r1")
    sim.enable_ospf("r2")
    sim.enable_ospf("r3")

    # Run until protocols converge
    converged = sim.run_until_converged(max_ticks=5000)
    print(f"Converged: {converged} at tick {sim.current_tick()}")
```

Hosts are created with optional IP and gateway configuration:

```
sim.add_host("h1", ip="10.0.1.100/24", gateway="10.0.1.1")
sim.add_host("h2", ip="10.0.2.100/24", gateway="10.0.2.1")
sim.add_link("h1", "r1")  # auto-creates interfaces
```

10.3 Simulation Control

Four methods provide different levels of control over the tick loop:

Table 14. Simulation control methods.

Method	Behaviour
`sim.run(max_ticks)`	Run for exactly `max_ticks` ticks. Supports `on_progress` callback every 100 ticks.
`sim.run_until_converged()`	Run until `is_converged()` returns `True` or `max_ticks` reached. Returns `bool`.
`sim.step()`	Execute a single tick. Returns current tick number.
`sim.run_iter(max_ticks)`	Return a Python iterator that yields tick numbers. Supports `stop_on_converged=True`.

All methods honour Ctrl-C via `py.check_signals()`, raising `KeyboardInterrupt` cleanly without leaving the engine in an inconsistent state.

The iterator interface is particularly useful for custom convergence logic:

```
for tick in sim.run_iter(max_ticks=10000):
    if tick % 500 == 0:
        status = sim.convergence_status()
        print(f"Tick {tick}: OSPF={status['ospf_converged']}, "
              f"BGP={status['bgp_converged']}")
    if sim.is_converged():
        break
```

10.4 Device State Inspection

The `device()` method returns a `PyDevice` wrapper that provides live access to device state. Each property access reads fresh data from the engine — there is no stale caching:

Listing 7. Inspecting device state after convergence.

```python
# Get device wrapper
r1 = sim.device("r1")

# Aggregate traffic counters
print(r1.counters)
# => Counters(packets_sent=1847, packets_received=1923, ...)

# Per-interface details
for iface in r1.interfaces:
    print(f"{iface.name}: {iface.ip}/{iface.prefix_len} "
          f"up={iface.is_up} tx={iface.counters.packets_sent}")

# Routing table access
rt = r1.routing_table
print(f"Routes: {len(rt)}")
for route in rt.routes:
    print(f"  {route['prefix']} via {route['next_hop']} "
          f"dev {route['interface']}")

# Specific route lookup
route = rt.lookup("10.0.23.0/24")
if route:
    print(f"Next hop: {route['next_hop']}")
```

10.5 Event Scheduling

The Python API exposes the engine's event scheduler for chaos engineering and failure injection:

Listing 8. Scheduling link failure and recovery.

```python
# Schedule link failure at tick 1000
fail_id = sim.schedule_event(1000, "interface_down",
    device="r1", interface="eth0")

# Schedule recovery 500 ticks later (relative)
sim.schedule_event_after(500, "interface_up",
    device="r1", interface="eth0")

# Dynamic topology changes
sim.schedule_event(2000, "remove_device",
    device="r3", mode="graceful")
sim.schedule_event(3000, "add_device",
    name="r4", type="router")
sim.schedule_event(3001, "add_wire",
    device_a="r1", device_b="r4")

# Cancel an event before it fires
sim.cancel_event(fail_id)
print(f"Pending events: {sim.pending_event_count()}")
```

Supported event actions: `interface_down`, `interface_up`, `remove_device` (with `graceful` or `immediate` mode), `remove_wire`, `add_device`, and `add_wire`.

10.6 Exception Hierarchy

The bindings define a structured exception hierarchy rooted at `NetsimError`:

- `NetsimError` — base class for all netsim exceptions.
- `DeviceNotFoundError` — raised when referencing a non-existent device by name or ID.

- `TopologyError` — raised for invalid topology operations (duplicate device names, adding interfaces to hosts, etc.).
- `SimulationError` — raised for runtime simulation errors (invalid event actions, missing arguments).

This enables idiomatic Python error handling:

```python
from netsim_py import Simulation, DeviceNotFoundError

try:
    dev = sim.device("nonexistent")
except DeviceNotFoundError as e:
    print(f"Device not found: {e}")
```

10.7 Integration with pytest

The Python bindings integrate naturally with pytest for network validation test suites:

Listing 9. pytest integration for network validation.

```python
import pytest
from netsim_py import Simulation

@pytest.fixture
def triangle_network():
    """Build and converge a triangle OSPF topology."""
    sim = Simulation()
    sim.add_router("r1").add_router("r2").add_router("r3")
    sim.add_interface("r1", "eth0", ip="10.0.12.1/24")
    sim.add_interface("r2", "eth0", ip="10.0.12.2/24")
    sim.add_link("r1", "r2", iface_a="eth0", iface_b="eth0")
    sim.enable_ospf("r1").enable_ospf("r2").enable_ospf("r3")
    sim.run_until_converged()
    return sim

def test_ospf_full_mesh_reachability(triangle_network):
    """All routers should have routes to all subnets."""
    sim = triangle_network
    r1_routes = sim.device("r1").routing_table.to_dict()
    assert "10.0.12.0/24" in r1_routes

def test_link_failure_reconvergence(triangle_network):
    """Traffic should reroute after a link failure."""
    sim = triangle_network
    sim.schedule_event(sim.current_tick() + 1,
        "interface_down", device="r1", interface="eth0")
    sim.run_until_converged(max_ticks=5000)
    assert sim.is_converged()
```

11 Usage

This section provides end-to-end examples of increasing complexity.

11.1 Example 1: Simple Reachability Test

Listing 10. Minimal reachability test.

```yaml
devices:
  - name: r1
    type: router
    interfaces:
      - name: eth0
        ipv4: 10.0.1.1/24
  - name: h1
```

```
    type: host
    interfaces:
      - name: eth0
        ipv4: 10.0.1.10/24
        gateway: 10.0.1.1
links:
  - endpoints: [r1:eth0, h1:eth0]
assertions:
  - type: reachability
    source: h1
    destination: 10.0.1.1
    expected: true
```

Question answered: "Can host h1 reach router r1?"

Run with: `netsim run topology.yaml`. The output reports assertion results (pass/fail) and convergence timing.

11.2 Example 2: OSPF Reconvergence After Link Failure

Listing 11. OSPF reconvergence test (abbreviated).

```
# 5-router OSPF ring: r1--r2--r3--r4--r5--r1
devices:
  - name: r1
    type: router
    # ... interfaces with OSPF area 0
  # ... r2 through r5

events:
  - at: converged + 100
    action: link_down
    link: r1-r2

assertions:
  - type: reachability
    source: h1
    destination: h2
    expected: true  # via alternate path
  - type: convergence_time
    max_ticks: 200  # must reconverge within 200 ticks
```

Question answered: "Does traffic reroute after a link failure, and does it reconverge within 200 ticks?"

11.3 Example 3: BGP L3VPN with Post-Convergence Inspection

Listing 12. BGP VPN with convergence-relative scripting.

```
# PE/CE topology with VRFs and iBGP route reflection
script:
  - at: converged + 500
    device: CE1
    command: ping 10.200.1.10
  - at: converged + 1s
    device: PE1
    command: show bgp vpnv4 vrf CUSTOMER-A
```

Question answered: "Are VPN prefixes reachable after convergence? What does the VPNv4 table look like?"

11.4 Example 4: Chaos Engineering

Listing 13. Probabilistic failure with cascade rules.

```
failure_patterns:
  - name: chaos-monkey
    trigger: !probabilistic
      check_interval: 100
      probability: 0.1
    selector:
      by_tier: spine
    action: fail_random_link
    recovery: !after_ticks
      ticks: 50

cascade_rules:
  - name: dual-failure-cascade
    trigger: !dependency_based
      primary_selector:
        link_name_pattern: "spine-*"
    action: fail_devices

assertions:
  - type: reachability
    source: host1
    destination: host2
    expected: true  # survives chaos
```

Question answered: “Does the fabric survive random spine failures with cascading effects?”

11.5 Example 5: VRRP Gateway Redundancy

Listing 14. VRRP first-hop redundancy with tracked uplinks.

```
devices:
  - name: r1
    type: router
    interfaces:
      - name: eth0
        ipv4: 10.0.1.2/24
      - name: eth1
        ipv4: 10.0.100.1/24  # uplink
  - name: r2
    type: router
    interfaces:
      - name: eth0
        ipv4: 10.0.1.3/24
      - name: eth1
        ipv4: 10.0.200.1/24  # uplink
  - name: h1
    type: host
    interfaces:
      - name: eth0
        ipv4: 10.0.1.10/24
        gateway: 10.0.1.1     # virtual IP

vrrp:
  groups:
    - id: 1
      interface: eth0
      virtual_ip: 10.0.1.1
      routers:
        - device: r1
          priority: 150       # master
```

```
          track_interface: eth1
          track_decrement: 60
        - device: r2
          priority: 100       # backup

events:
  - at: converged + 100
    action: interface_down
    device: r1
    interface: eth1           # tracked uplink fails

assertions:
  - type: reachability
    source: h1
    destination: 10.0.100.1
    expected: false           # r1 uplink is down
  - type: reachability
    source: h1
    destination: 10.0.200.1
    expected: true            # r2 takes over as master
```

Question answered: "When the primary gateway's uplink fails, does VRRP failover to the backup and restore host reachability?"

11.6 Example 6: DHCP Relay Across Subnets

Listing 15. DHCP relay with Option 82.

```
devices:
  - name: r1
    type: router
    interfaces:
      - name: eth0
        ipv4: 10.0.1.1/24
        helper_addresses: [10.0.100.10]
      - name: eth1
        ipv4: 10.0.100.1/24
  - name: dhcp-server
    type: host
    interfaces:
      - name: eth0
        ipv4: 10.0.100.10/24
        gateway: 10.0.100.1
  - name: client
    type: host
    interfaces:
      - name: eth0
        ipv4: 10.0.1.0/24     # unconfigured
        gateway: 10.0.1.1
```

Question answered: "Does the relay agent correctly forward DHCP Discover packets from the client subnet to the server, populating GIADDR and Option 82?"

12 Example Topologies

netsim ships with over 40 example topologies, embedded in the CLI binary via `include_str!()`. They serve three purposes: onboarding (learning the YAML format), regression testing (every example is gated by a convergence test), and benchmarking (the larger examples are used for performance measurement). This section surveys the examples by category.

12.1 Built-in Examples (`netsim example`)

The `netsim example <name>` subcommand prints a complete, runnable topology to stdout. Eight examples are available:

Table 15. Built-in example topologies.

Name	Protocols	Devices	Demonstrates
`simple`	—	2	Minimal connectivity
`ospf-triangle`	OSPF	5	Basic IGP routing
`data-center`	OSPF	20	Spine-leaf L3 fabric
`wan-mesh`	OSPF	18	Full-mesh WAN with costs
`enterprise-campus`	OSPF	18	3-tier campus hierarchy
`mixed-network`	OSPF	15	Router/Switch/Hub mix
`large-enterprise`	OSPF, Traffic	440	Multi-area enterprise
`service-provider`	OSPF, Traffic	108	P/PE/CE hierarchy

Example: Generate and run a topology

```
# Print the service-provider example
netsim example service-provider > sp.yaml

# Run it
netsim run sp.yaml --format json --output sp-results.json
```

12.2 Extended Examples

The `examples/` directory contains 40+ topologies covering specialised protocols and advanced scenarios. Table 16 highlights the most interesting ones, grouped by the protocol stack they exercise.

Table 16. Selected extended example topologies.

Example	Protocols	Dev.	Demonstrates
IGP & Link-State Routing			
`isis-hierarchical`	IS-IS	8	L1/L2/L1L2 areas
`isis-lan`	IS-IS	4	Broadcast LAN DIS election
`scale-test-ring`	OSPF	100	100-device ring convergence
BGP & Policy			
`bgp-ipv6-multi-as`	BGP	3	Multi-AS IPv6 peering
`bgp-community-policy`	OSPF, BGP	3	NO_EXPORT, NO_ADVERTISE
`docs-bgp-rr-loop`	BGP	3	iBGP route reflection loops
MPLS, SR & Tunneling			
`docs-mpls-ldp-oam`	OSPF, MPLS/LDP	5	LSP ping/traceroute

Example	Protocols	Dev.	Demonstrates
`rsvp-te-tunnel`	OSPF, RSVP-TE	4	Explicit path tunnels
`sr-mpls-transport`	OSPF, SR-MPLS	6	Segment Routing with SRGB
`gre-overlay`	OSPF, GRE	5	GRE over OSPF underlay
L3VPN & Data Centre			
`l3vpn-service-provider`	OSPF, BGP, MPLS	7	VRF, VPNv4, PE/CE
`dc-fabric-irb`	OSPF, BGP, VXLAN	8	Integrated routing/bridging
`dc-fabric-multi-homing`	BGP	10	EVPN multi-homing
`dual-stack-data-center`	OSPF	20	IPv4 + IPv6 fabric
Enterprise & Campus			
`vrrp-redundancy`	VRRP	3	First-hop gateway failover
`dhcp-relay-simple`	DHCP Relay	3	Broadcast-to-unicast relay
`rip-simple-network`	RIP	4	Distance-vector convergence
Fast Failover & Infrastructure			
`bfd-fast-failover`	OSPF, BGP, BFD	3	Sub-second BFD failover
`lag-lacp`	LACP	4	Dynamic LAG negotiation
Traffic & Chaos			
`dc-fabric-traffic`	OSPF, Traffic	10	Traffic generation in DC
`qos-congestion`	OSPF, Traffic	4	Congestion & queue drops
`chaos-simple`	—	6	Failure injection patterns
`datacenter-disaster`	—	10	Multi-failure disaster
`jitter-validation`	—	6	Impairment validation
Dual-Stack & Scale			
`dual-stack-service-provider`	OSPF, Traffic	108	IPv4+IPv6 SP
`large-enterprise`	OSPF, Traffic	440	Multi-area enterprise

Note

Every example in this table is covered by an automated regression test in `tests/example_topology_regression.rs`. Each test verifies that the topology parses, builds, and converges within 50,000 ticks with a convergence threshold of 500 stable ticks. This ensures that protocol changes never silently break existing topologies.

12.3 Protocol Coverage Across Examples

Table 17 shows which protocols are exercised by the example suite. The combination ensures that every protocol implementation has at least one end-to-end topology test.

Table 17. Protocol coverage across the example suite.

Protocol	Examples	Key topologies
OSPF v2	20+	`ospf-triangle`, `large-enterprise`
OSPFv3	2	`dual-stack-*`
IS-IS	3	`isis-hierarchical`, `isis-lan`
BGP (iBGP)	6	`docs-bgp-rr-loop`, `l3vpn-*`
BGP (eBGP)	3	`bgp-ipv6-multi-as`, `dc-fabric`
BGP EVPN	3	`dc-fabric-irb`, `dc-fabric-multi-homing`
MPLS/LDP	3	`docs-mpls-ldp-oam`, `l3vpn-*`
RSVP-TE	1	`rsvp-te-tunnel`
SR-MPLS	1	`sr-mpls-transport`
BFD	1	`bfd-fast-failover`
LACP	2	`lag-lacp`, `lag-basic`
GRE	1	`gre-overlay`
VXLAN	2	`dc-fabric-irb`, `dc-fabric-multi-homing`
RIP	1	`rip-simple-network`
STP	1	`stp-loop-prevention`
VRRP	1	`vrrp-redundancy`
DHCP Relay	1	`dhcp-relay-simple`
Traffic gen	8	`*-traffic`, `large-enterprise`

13 Performance

This section documents netsim's performance characteristics, benchmarking infrastructure, and scalability.

13.1 Benchmark Infrastructure

netsim includes a Criterion-based benchmark suite in `benches/` that measures simulation performance across topology sizes and workload types. Three benchmark suites target different aspects:

Table 18. Benchmark suites.

Suite	File	What it measures
Scale	`benches/scale.rs`	Throughput (ticks/sec) across topology sizes from 2 to 486 devices. Tests five workload types: ControlPlane, Mixed, DataPlane, DataPlaneMultiFlow, MixedWithFailure.
Realistic	`benches/realistic.rs`	Parallel vs. serial execution comparison. Tracks ticks/sec, packet loss %, throughput (pps), and latency percentiles (P50/P95/P99).
Dual-stack	`benches/dual_stack_scale.rs`	IPv4 vs. IPv6 convergence overhead. Expected: dual-stack adds $< 1.5\times$ overhead vs. single-stack.

Scenario registry. Benchmarks draw from a scenario registry that defines four reference topologies at different scales:

Table 19. Benchmark reference scenarios.

Scenario	Devices	Topology	Purpose
`two-host-wire`	2	Direct link	Baseline overhead
`three-host-chain`	3	Linear chain	Minimal forwarding path
`wan-mesh-150`	150	25 routers, 5 hosts each	Medium-scale WAN
`dc-fabric-486`	486	6 spines, 30 leaves, 15 hosts/leaf	Large-scale DC fabric

13.2 Metrics

Each benchmark run collects six metrics:

1. **Ticks/sec**: Simulation throughput—how many ticks the engine can execute per wall-clock second. Higher is better.
2. **Convergence tick**: The tick at which the convergence predicate (section 4) first becomes true. Lower means faster protocol convergence.
3. **Packet loss %**: Fraction of generated traffic that is dropped. Used to validate data-plane correctness under failure scenarios.
4. **Throughput (pps)**: Packets delivered per second of simulation time. Measures data-plane capacity.
5. **Latency percentiles**: P50, P95, P99 packet delivery times. Measures forwarding-path quality.
6. **Determinism variance**: Standard deviation of convergence tick across repeated runs. Should be zero for a deterministic engine.

> **Note**
>
> Benchmarks default to 12 repetitions (`DEFAULT_REPEATS: 12`) and a 500-tick budget (`DEFAULT_TICK_BUDGET: 500`). The 12-repetition count is chosen to detect non-determinism: if any run produces a different convergence tick, the benchmark flags a determinism warning.

13.3 Scalability

netsim's performance scales with topology size across four tiers:

Table 20. Scalability tiers.

Tier	Devices	Example	Characteristics
Minimal	2–5	`simple`, `ospf-triangle`	Sub-millisecond per tick; useful for unit tests
Small	6–20	`data-center`, `enterprise-campus`	Milliseconds per tick; protocol interaction testing
Medium	100–150	`scale-test-ring`, `wan-mesh-150`	Tens of milliseconds per tick; regional network scale
Large	440–486	`large-enterprise`, `dc-fabric-486`	<0.5 ms per tick; adaptive parallelism engages

13.4 Zero-Copy SoA Device Storage

The engine uses a type-segregated Structure of Arrays (SoA) layout instead of a single polymorphic `BTreeMap<DeviceId, Box<dyn Device>>`:

```rust
pub struct DeviceStorage {
    pub routers: Vec<Option<Box<Router>>>,
    pub hosts:   Vec<Option<Box<Host>>>,
    pub switches: Vec<Option<Box<Switch>>>,
    pub other:   Vec<Option<Box<dyn Device>>>,
}
```

Devices are indexed by `DeviceId` as a direct array index ($O(1)$ lookup). The `Option` wrapper handles sparse IDs after device removal. This layout provides two benefits:

1. **Lock-free parallelism**: Rayon can call `par_iter_mut()` independently on each `Vec` with no interference—each vector is a separate heap allocation with no shared mutable state.
2. **Type-specific fast paths**: Code that only needs routers (e.g., OSPF SPF, BGP decision process) can iterate `devices.routers` directly without dynamic dispatch or downcasting.

13.5 BatchQueue and Priority Queuing

Each interface holds a bounded FIFO queue that groups consecutive frames by destination MAC into batches:

```rust
pub struct BatchQueue {
    batches: VecDeque<PacketBatch>,
    depth: usize,    // max packets (default 1000)
    drops: u64,      // overflow counter
}
```

`enqueue_frame()` appends to the last batch if the destination MAC matches; otherwise it creates a new batch. This reduces per-frame overhead for burst traffic to the same destination. `drain_all_batches()` uses `std::mem::take()` for $O(1)$ drain without per-element moves.

Interfaces use a three-tier priority queue that wraps three `BatchQueue` instances:

Table 21. Priority queue tiers.

Tier	Queue	Traffic types
High	`control`	ARP, OSPF, BGP, IS-IS, ICMP, VRRP, BFD, LLDP
Normal	`data`	UDP, TCP, generic IPv4/IPv6
Low	`background`	Background traffic

Dequeue is strict-priority: all control-plane packets are drained before any data-plane packets. Rate-limited draining (`drain_rate_limited()`) supports byte-budget-based traffic shaping.

13.6 Phase Timing Instrumentation

The engine instruments every phase of the tick loop using `Instant::now()` / `.elapsed().as_micros()` to build a cumulative microsecond-precision timing profile:

```
pub struct PhaseTimingTotals {
    pub transfer_to_wires_us: u64,
    pub forwarding_us: u64,
    pub control_us: u64,
    pub finalize_us: u64,
    pub wires_us: u64,
    pub transfer_from_wires_us: u64,
    pub convergence_us: u64,
    pub ticks: u64,
}
```

The benchmark suite prints per-phase breakdowns after the first run of each scenario, enabling identification of bottleneck phases. For example, on a 486-device DC fabric, forwarding typically consumes 60–70% of tick time, while convergence checking takes $< 5\%$.

13.7 Incremental FIB with Dirty Flags

Rather than recomputing the full FIB every tick, each forwarding table uses a `changed: bool` dirty flag that is set only when an entry actually differs from the existing one. This equality-checked install means that an OSPF SPF recomputation producing the same routes does not trigger a false convergence reset. The same incremental pattern is applied across all protocol tables:

- `fib_changed` — global IPv4 FIB and per-VRF FIBs
- `lfib_changed_this_tick` — MPLS LFIB
- `ldp_bindings_changed` — LDP label bindings
- `vpnv4_loc_rib_changed` — VPNv4 Loc-RIB
- `evpn_loc_rib_changed_this_tick` — EVPN Loc-RIB
- `bgp_loc_rib_changed` — BGP IPv4 unicast Loc-RIB
- `routes_changed` — per-protocol (OSPF, IS-IS, RIP)

All flags are cleared after the convergence check at the end of each tick, implementing a clear-after-read pattern that ensures each change is visible exactly once.

13.8 Adaptive Parallelism

The engine uses two thresholds to decide when parallelism is worthwhile:

- **Device threshold** (default: 500): Below this, device ticking (phase P2) runs serially. At or above, Rayon `par_iter_mut` is used.
- **Packet threshold** (default: 1,000): Below this, packet delivery (phases P1, P3, P4) runs serially. At or above, packet processing is parallelised.

> **Design Decision**
>
> **Why two thresholds?**
> Device ticking and packet processing have different computational profiles. Device ticking involves running protocol state machines (OSPF SPF, BGP decision process)—CPU-intensive with variable cost per device. Packet processing involves queue operations and frame copying—memory-intensive with uniform cost per packet. Separate thresholds allow each to switch independently based on workload.
> **Trade-off**: Both thresholds are compile-time defaults, not auto-tuned at runtime. A topology just below the threshold misses parallelism even on machines with many cores.

13.9 Execution Modes

netsim supports two execution modes, selectable at build time or via configuration:

- **Parallel** (default): Uses Rayon with auto-detected worker threads. The number of threads can be overridden via `worker_threads` in `EngineConfig`.
- **Serial**: Disables all parallelism. Useful for debugging (deterministic single-threaded execution makes backtraces readable) and for measuring parallelism speedup in benchmarks.

The Realistic benchmark suite explicitly compares parallel and serial execution on the same topology to measure Rayon overhead and parallelism benefit. A run is flagged as a regression if parallel throughput drops below 90% of serial throughput for topologies with ≥ 10 nodes. This comparison is tracked across releases to detect performance regressions.

13.10 Regression Testing

All 20+ example topologies are covered by convergence regression tests.[1] Each test verifies:

1. The YAML topology parses without errors.
2. The Builder constructs the simulation without errors.
3. The simulation converges within 50,000 ticks.
4. Convergence is sustained for 500 consecutive stable ticks.

These tests run in CI on every commit, ensuring that protocol changes or engine modifications never silently break convergence behaviour for any reference topology.

13.11 Control Plane Policing (CoPP)

netsim implements per-destination token-bucket rate limiting for ICMP error generation, preventing routers from being overwhelmed by error storms (e.g., traceroute to an unreachable destination generating thousands of TTL-exceeded messages):

[1] `tests/example_topology_regression.rs`

```
pub struct IcmpRateLimiter {
    buckets: BTreeMap<Ipv4Address, TokenBucket>,
    config: RateLimitConfig,
}

RateLimitConfig {
    bucket_size: 10,       // max burst of 10 ICMP errors
    refill_tokens: 1,      // 1 token per refill
    refill_interval: 100,  // refill every 100 ticks
}   // => 10 ICMP errors/sec at 10ms/tick
```

Each destination IP address gets its own `TokenBucket`. Before generating a TTL-exceeded or destination-unreachable ICMP error, the router calls `allow_error(dst_ip, current_tick)`. If the bucket is empty, the error is silently dropped. `BTreeMap` is used for the bucket map to ensure deterministic iteration order. Rate limit parameters are configurable per router via `set_icmp_rate_limit()`.

13.12 End-to-End Hardening

The test suite includes 20+ integration tests covering four hardening categories:

1. **Resiliency & Failure Scenarios**: RSVP-TE LSP convergence under node crash, MPLS label stack determinism under cascading failures, link flap and brown-out scenarios.
2. **Robustness & Attack Defence**: Graceful handling of malformed packets (TTL=0, cyclic labels, excessive MPLS stack depth), control-plane fuzzing, route spoofing defence.
3. **Performance & Telemetry**: High-throughput data-plane validation ensuring control-plane protocols are not starved, bounded memory growth under routing loops (TTL expiration).
4. **Control Plane Policing**: Token-bucket rate limiting validation, per-protocol burst/refill configuration, DoS resilience.

14 Limitations

14.1 No TCP/UDP Transport Modelling

Description: netsim abstracts the transport layer. There is no TCP congestion window, RTT estimation, flow control, or UDP socket modelling. Traffic generators produce frames at a configured rate without transport-layer feedback.

Workaround: For transport-layer research, use ns-3 or a container-based emulator. netsim is designed for routing-level validation, not transport-level simulation.

Planned resolution: Basic TCP throughput estimation is under consideration for a future release, but full cwnd/ssthresh modelling is out of scope.

14.2 Idealised Device Behaviour

Description: netsim models RFC behaviour, not vendor-specific implementation quirks. Real routers have bugs, non-standard defaults, and proprietary extensions that netsim does not reproduce.

Workaround: Use netsim for "does the design work in theory?" and Containerlab for "does it work with this specific NOS version?"

Planned resolution: No current plan. Modelling vendor bugs would undermine the determinism guarantee.

14.3 No Detailed Queuing Theory

Description: The `BatchQueue` is FIFO with drop-tail. There is no RED, WRED, WFQ, or priority queuing. Interface capacity tracking (Phase 118) is recent and tracks utilisation but does not model queuing delays.

Workaround: For QoS validation, use a container-based emulator with a real queuing discipline.

Planned resolution: Priority queuing and basic RED are candidates for a future milestone.

14.4 Scale Ceiling

Description: netsim has been tested to 486 devices. It is not validated at 10,000+ device scale. Memory is bounded by single-machine RAM; there is no distributed execution mode.

Workaround: For topologies larger than ~500 devices, validate representative subsets rather than the full topology.

Planned resolution: Distributed execution via partitioned tick processing is listed as future work.

14.5 EVPN Synchronisation Fidelity

Description: The snapshot-then-apply EVPN synchronisation pattern does not model individual BGP UPDATE message ordering or per-session backpressure. It cannot reproduce bugs that depend on the order in which a route reflector processes updates from different clients.

Workaround: For BGP implementation testing (as opposed to configuration validation), use real BGP daemons in containers.

Planned resolution: No current plan. The snapshot-then-apply design is a deliberate trade-off favouring determinism over message-level fidelity.

14.6 BFD Auto-Session Wiring

Description: BFD sessions configured via `bfd: true` in YAML are not automatically created for all protocol neighbours. Tests verify the BFD feature works, but full auto-wiring from the YAML flag requires additional Builder logic.

Workaround: Configure BFD sessions explicitly in the YAML topology.

Planned resolution: Deferred to a future milestone.

15 Future Work

- **Distributed execution**: Partition the tick pipeline across multiple machines for 10,000+ device topologies.
- **TCP transport modelling**: Add basic TCP throughput estimation (bandwidth-delay product) without full congestion control simulation.
- **Network digital twin integration**: Connect netsim to production network controllers to serve as a pre-deployment validation backend.
- **SRv6 completion**: The SRv6 foundation (SRH codec, SID table, End/End.X) is implemented. Remaining work includes service SIDs (End.DT4/DT6), SRv6 Policy with colour/endpoint steering, flavours (PSP/USP/USD), IGP locator advertisement, and MPLS-TP OAM/protection (milestone v2.2, Phases 125–129).
- **Multicast**: IGMP/MLD snooping and PIM-SM (Proposed milestone v2.3).

- **Topogen integration**: Native export format for the Topogen topology generator to enable automated topology generation and simulation.
- **RAPID STP / MSTP**: Extend STP to Rapid STP (802.1w) and Multiple STP (802.1s) for faster L2 convergence.
- **Full DHCP server**: Add pool-based address allocation to complement the existing relay implementation.
- **Extended Python bindings**: Expand the PyO3 bindings with BGP/OSPF configuration APIs, YAML topology loading, packet capture access, and integration with Jupyter notebooks for interactive visualisation of convergence timelines.

References

[1] L. Andersson, I. Minei, and B. Thomas. LDP Specification. RFC 5036, 2007.

[2] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *IEEE Software*, 33(3):35–41, 2016.

[3] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. OSPF for IPv6. RFC 5340, 2008.

[4] Containerlab. Containerlab: Container-based networking labs, 2024. Accessed: 2024-01-15.

[5] R. Droms. Dynamic Host Configuration Protocol. RFC 2131, 1997.

[6] C. Filsfils, D. Duber, H. Elmalky, R. Bonica, N. Spring, C. Pignataro, J. Leddy, D. Mozes, and S. Matsushima. IPv6 Segment Routing Header (SRH). RFC 8754, 2020.

[7] R. Hinden. Virtual Router Redundancy Protocol (VRRP). RFC 3768, 2004.

[8] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, 2010.

[9] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. In *Proc. SIGCOMM*, pages 175–187. ACM, 2000.

[10] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN). RFC 7348, 2014.

[11] G. Malkin. RIP Version 2. RFC 2453, 1998.

[12] G. Malkin and R. Minnear. RIPng for IPv6. RFC 2080, 1997.

[13] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proc. HILT*, pages 103–104. ACM, 2014.

[14] J. Moy. OSPF Version 2. RFC 2328, 1998.

[15] S. Nadas. Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6. RFC 5798, 2010.

[16] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, 2006.

[17] A. Sajassi, R. Aggarwal, N. Bitar, A. Isaac, J. Uttaro, J. Drake, and W. Henderickx. BGP MPLS-Based Ethernet VPN. RFC 7432, 2015.

A YAML Schema Quick Reference

Top-level key	Type	Description
`name`	String	Topology name
`tick_ms`	Integer	Tick duration in ms (default: 1)
`devices`	List	Device definitions
`links`	List	Link definitions
`ospf`	Object	OSPF configuration
`bgp`	Object	BGP configuration
`isis`	Object	IS-IS configuration
`mpls`	Object	MPLS/LDP configuration
`rsvp`	Object	RSVP-TE configuration
`rip`	Object	RIP/RIPng configuration
`stp`	Object	STP configuration
`vrrp`	Object	VRRP group configuration
`traffic`	List	Traffic generator definitions
`impairment_profiles`	List	Named impairment profiles
`failure_patterns`	List	Chaos engineering patterns
`events`	List	Scheduled events
`assertions`	List	Post-simulation assertions
`script`	List	Convergence-relative commands

B CLI Command Quick Reference

Command	Description
`show ip route`	Display IPv4 FIB
`show ip route vrf NAME`	Display VRF-specific FIB
`show bgp summary`	Display BGP neighbour summary
`show bgp vpnv4 vrf NAME`	Display VPNv4 routes for a VRF
`show bgp evpn`	Display EVPN routes
`show ospf neighbor`	Display OSPF adjacencies
`show ospf database`	Display OSPF LSDB
`show isis adjacency`	Display IS-IS adjacencies
`show mpls forwarding`	Display MPLS LFIB
`show mpls ldp bindings`	Display LDP label bindings
`show interface`	Display interface status
`show lldp neighbors`	Display LLDP neighbour table
`show lacp`	Display LACP state

Command	Description
`show bfd sessions`	Display BFD session state
`show rip`	Display RIP routing table and neighbours
`show stp`	Display STP bridge and port state
`show vrrp`	Display VRRP group state and priority
`show vxlan vtep`	Display VXLAN VTEP endpoints
`show ip dhcp relay`	Display DHCP relay statistics
`show ipv6 route`	Display IPv6 FIB
`show ospfv3 neighbor`	Display OSPFv3 adjacencies
`show bgp ipv6 unicast`	Display BGP IPv6 routes
`show routing-matrix`	Display full routing matrix
`show trace DESTINATION`	Display static forwarding path
`ping DESTINATION`	ICMP ping (drives simulation stepping)
`traceroute DESTINATION`	Traceroute (drives simulation stepping)
`dig HOSTNAME`	DNS query (drives simulation stepping)
`iperf TARGET`	Throughput measurement

All `show` commands support `--json` for structured output.