NetVis: Network Topology Visualization Engine

Technical Reference

Simon Knight

Adelaide, Australia

March 2026 • Version 1.1

Abstract. NetVis is a Rust-based network topology layout and visualization engine comprising 90,000 lines of code. It transforms complex multi-layer network topologies into publication-quality visualizations using composable layout algorithms, R*-tree–accelerated label placement, force-directed edge bundling, and adaptive level-of-detail rendering. This document is the definitive technical reference for NetVis. It describes the data model, architecture, ten layout algorithms, seven edge routing refiners, rendering pipeline, topology diffing, temporal analysis, traffic animation, annotation markup, interactive editing, visual effects, 16 quality metric families, and public API. Appendices provide the full YAML topology schema, performance baselines with comparative benchmarks against Graphviz, and a complete configuration reference.

Version	Date	Changes
1.0	March 2026	Initial release.
1.1	March 2026	Add routing refiners, topology diffing, timeline analysis, traffic animation, annotation markup, interactive editor, visual effects, expanded quality metrics (16 families), five new layout algorithms, and configuration reference appendix.

Contents

1 Introduction

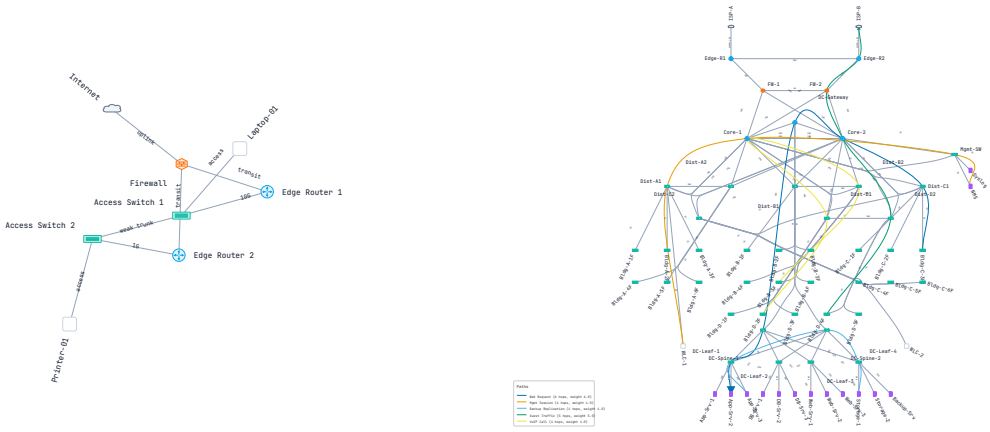
NetVis is a network topology layout and visualization engine. Given a graph of network devices and connections (from a YAML file, NetBox [8] instance, or LLDP discovery), it produces publication-quality SVG diagrams, interactive HTML visualizations, or PNG/PDF exports. It addresses the gap between general-purpose graph layout tools (Graphviz [10], OGDF [6], D3.js [5]) and manual diagramming applications (yEd [20], Visio).

1.1 Design Philosophy

Two principles guide every design decision in NetVis:

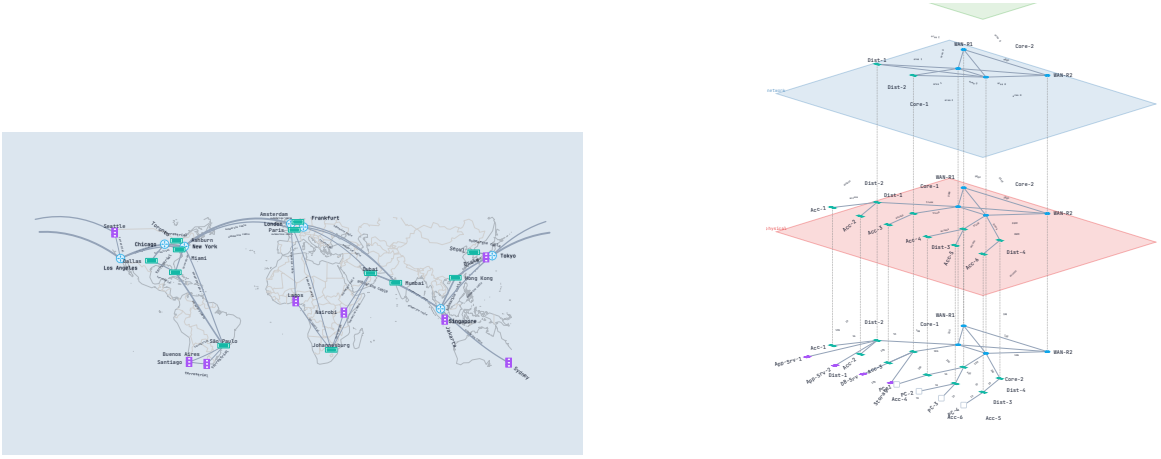
Domain-specialized composition. Network topologies have exploitable structural regularity—hierarchical tiers, geographic site placement, protocol-layer stacking. Rather than using a single general-purpose layout algorithm, NetVis decomposes visualization into a pipeline of composable refiners, each specialized for one aspect of the output.

Quality by default. The system should produce readable, labeled, styled diagrams without manual post-processing. Quality metrics (crossings, overlaps, label readability) are measured as part of the rendering pipeline, not as an afterthought.



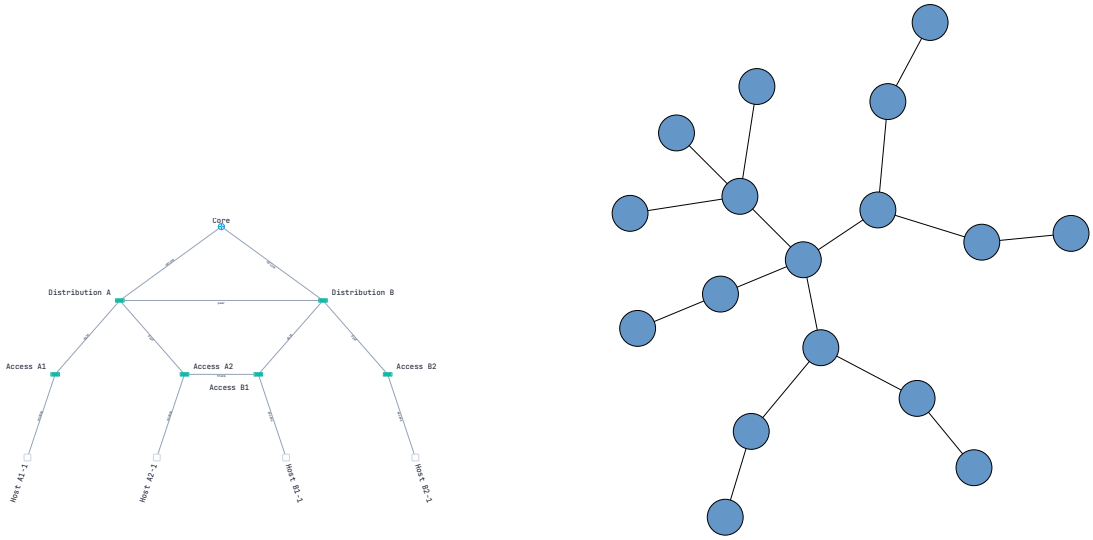
(a) Campus edge (force-directed, 8 nodes) **(b)** Enterprise campus (72 nodes, edge bundling)

Figure 1. Representative NetVis outputs (1/3). All diagrams generated automatically from YAML with no manual adjustment.



(a) Global CDN (geographic, great-circle arcs, map layer) **(b)** Protocol stack (isometric multi-layer, 3 planes)

Figure 2. Representative NetVis outputs (2/3).



(a) Hierarchical datacenter (Sugiyama-style layered layout) **(b)** Hub-and-spoke WAN (radial tree layout)

Figure 3. Representative NetVis outputs (3/3).

1.2 Audience

This document is intended for:

- Engineers embedding NetVis as a library in a network management platform (via Rust, Python, or WASM).
- Contributors to the NetVis codebase.
- Researchers who want to understand the algorithms in detail.

Familiarity with Rust and basic graph theory is assumed. Familiarity with network engineering concepts (spine-leaf, OSPF, BGP) is helpful but not required.

1.3 How to Read This Document

- **Quick start:** Read §2 (By Example), then §17 (API and Usage).
- **Data model and architecture:** §3–§4.

- **Deep dive into algorithms:** §5–§15.
- **Analysis features:** §9–§13 (diff, timeline, traffic, annotations, editor).
- **Deployment and configuration:** §18 and Appendix B.
- **Known issues:** §20.

2 NetVis by Example

This section presents four topologies of increasing complexity, each illustrating different NetVis capabilities. All examples can be rendered with a single CLI command.

2.1 Example 1: Basic Campus Edge

A minimal 8-node campus network demonstrating the core YAML schema (Appendix A): node types, edge attributes, directed links, and force-directed layout (§5).

Basic Campus Edge

```yaml
name: Basic campus edge
render:
  layout: force_directed
  theme: network
  seed: 123
nodes:
  - id: inet
    label: Internet
    node_type: cloud
  - id: fw1
    label: Firewall
    node_type: firewall
  - id: r1
    label: Edge Router 1
    node_type: router
    attributes: { role: edge, site: hq }
  - id: s1
    label: Access Switch 1
    node_type: switch
  - id: h1
    label: Laptop-01
    node_type: host
edges:
  - from: inet
    to: fw1
    label: uplink
    directed: true
    attributes: { bandwidth_gbps: 10 }
  - from: fw1
    to: r1
    label: transit
  - from: r1
    to: s1
    label: 10G
  - from: s1
    to: h1
    label: access
```

```
netvis –input basic.yaml -o basic.svg
```

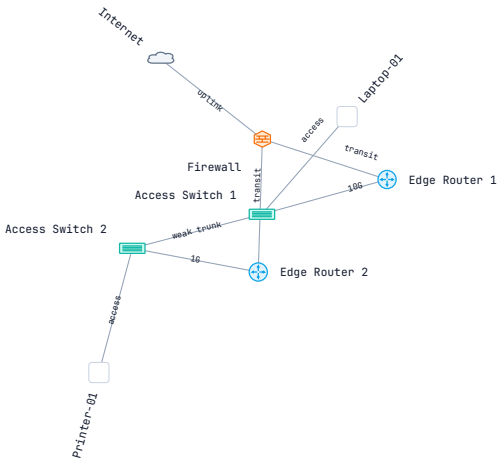


Figure 4. Rendered output of Example 1: an 8-node campus edge topology with force-directed layout, automatic label placement, and device-type icons.

2.2 Example 2: Traffic Animation Overlay

A 9-node NOC topology with live traffic flow animation (§11). The `traffic:` block references edges by `id` and assigns utilization levels, directions, and animation styles.

Traffic NOC Demo (excerpt)

```
name: NOC Traffic Dashboard
render:
  theme: dark
  width: 1400
  height: 900
nodes:
  - id: core1
    label: Core-1
    node_type: router
  - id: agg1
    label: Agg-1
    node_type: l3_switch
  # ... (7 more nodes)
edges:
  - from: core1
    to: agg1
    id: core1-agg1
    label: 40G
traffic:
  animation_style: dot
  edges:
    - id: core1-agg1
      utilization: 0.85
      direction: bidirectional
    - id: agg1-access1
      utilization: 0.45
      direction: forward
```

```
netvis —input noc.yaml —traffic noc.yaml -o traffic.svg
```

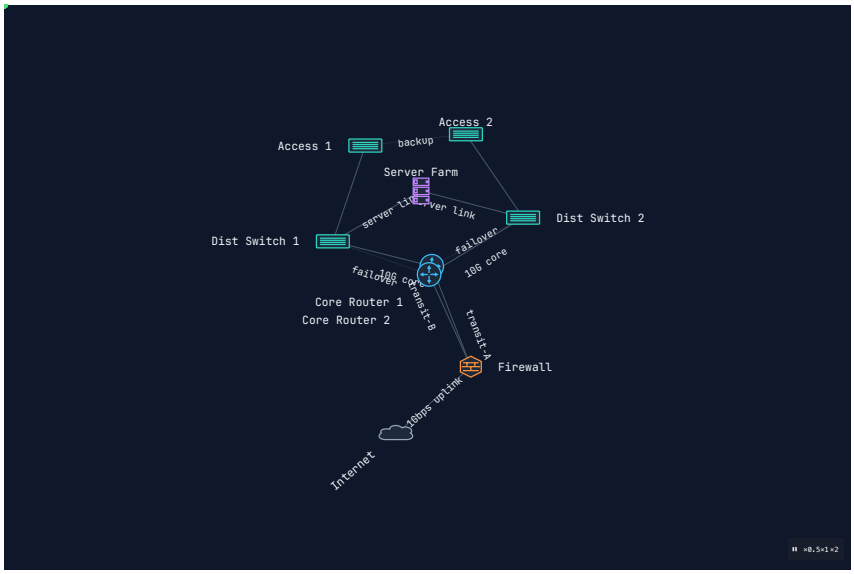


Figure 5. Rendered output of Example 2: a NOC dashboard with animated traffic flow overlays. Edge color encodes utilization (green → amber → red); animated dots show traffic direction and rate.

2.3 Example 3: Geographic CDN

A 30-node global CDN using geographic layout (§5) with real-world coordinates, background map layer, and great-circle arc edges.

Global CDN Network (excerpt)

```yaml
name: Global CDN Network
render:
  preset: geographic
  theme: dark
  width: 1800
  height: 1000
geographic:
  show_map: true
  great_circle_arcs: true
nodes:
  - id: nyc
    label: New York
    node_type: router
    lat: 40.7128
    lon: -74.0060
    highlight: true
    attributes: { region: north-america, tier: primary }
  - id: london
    label: London
    node_type: router
    lat: 51.5074
    lon: -0.1278
    highlight: true
  - id: tokyo
    label: Tokyo
    node_type: router
    lat: 35.6762
```

```
    lon: 139.6503
edges:
  - from: nyc
    to: london
    label: submarine cable
    edge_type: submarine
    attributes: { bandwidth_gbps: 400, cable: "Marea / AEC-2" }
```

```
netvis —input cdn.yaml -o cdn.svg
```

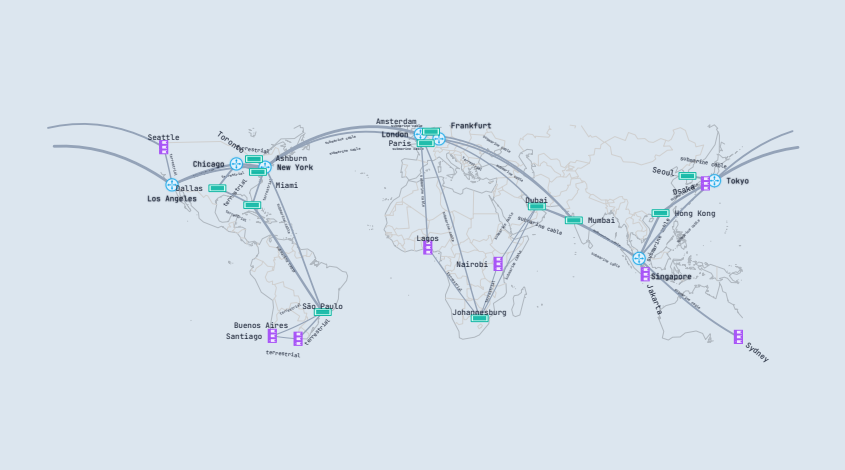


Figure 6. Rendered output of Example 3: a global CDN with geographic layout, great-circle arc edges, and a background map layer.

2.4 Example 4: Isometric Multi-Layer

A 44-node enterprise WAN visualized as three isometric protocol layers (§5): Physical (all devices), Network (L2/L3 only), and Routing (OSPF routers only).

Isometric Multi-Layer (excerpt)

```
name: Isometric multi-layer enterprise WAN
render:
  layout: isometric
  width: 1800
  height: 1400
  theme: network
layers:
  - id: physical
    label: Physical
    order: 0
  - id: network
    label: Network
    order: 1
  - id: routing
    label: Routing
    order: 2
nodes:
  - id: p-core1
    label: Core-1
    node_type: router
    attributes: { layer: physical }
  - id: n-core1
```

```
    label: Core-1
    node_type: router
    attributes: { layer: network }
  - id: r-core1
    label: Core-1
    node_type: router
    attributes: { layer: routing }
edges:
  - { from: p-core1, to: p-dist1, label: "10G" }
  - { from: n-core1, to: n-dist1, label: "trunk" }
  - { from: r-core1, to: r-dist1, label: "area 0" }
```

```
netvis –input multilayer.yaml -o multilayer.svg
```

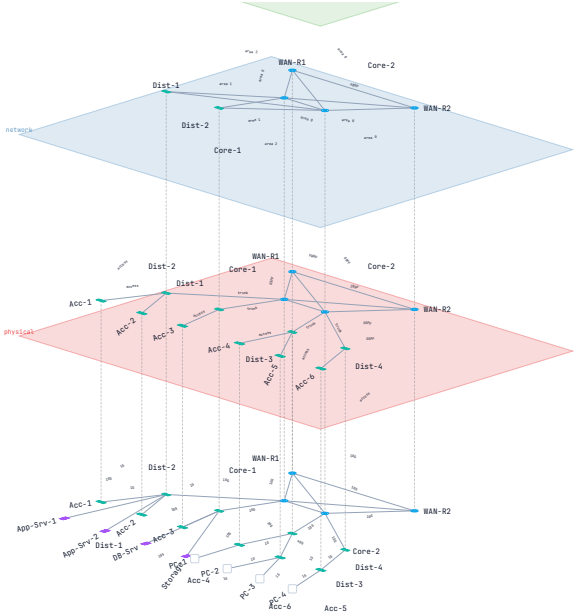


Figure 7. Rendered output of Example 4: isometric multi-layer visualization with three protocol planes (Physical, Network, Routing) separated along the Z-axis.

3 Data Model

NetVis uses a directed graph as its core data structure, built on the `petgraph` library [4].

3.1 Graph Types

Listing 1. Core graph types

```
pub type NetVisGraph = StableGraph<NodeData, EdgeData>;
pub type NodeId = NodeIndex<u32>;
pub type EdgeId = EdgeIndex<u32>;
```

`StableGraph` is used (rather than `Graph`) because node and edge indices remain valid after removal—important for the editor's undo/redo system.

3.2 Node Data

Each node carries:

Listing 2. NodeData (simplified)

```rust
pub struct NodeData {
    pub id: String,           // Unique identifier (e.g., "router-fra-01")
    pub label: Option<String>, // Display label
    pub node_type: Option<String>, // Device type: router, switch, server, ...
    pub attributes: HashMap<String, Value>, // Arbitrary metadata
    pub position: Option<Point2D>, // Hint for layout
    pub group: Option<GroupId>,    // Group membership
    pub layer: Option<LayerId>,    // Layer membership
}
```

Note

The `node_type` field drives device-specific rendering: icon shape, default color, and default size. The built-in types are: `router`, `switch`, `server`, `firewall`, `cloud`, `l3_switch`. Unknown types render as circles.

3.3 Edge Data

Listing 3. EdgeData (simplified)

```rust
pub struct EdgeData {
    pub label: Option<String>,
    pub weight: f64,              // Default: 1.0
    pub edge_type: Option<String>,
    pub directed: bool,           // Default: false
    pub attributes: HashMap<String, Value>,
}
```

3.4 Topology Input Format

The primary input format is YAML:

Listing 4. Example topology (campus network)

```yaml
name: Campus Edge Network

render:
  preset: default
  theme: network
  width: 1200
  height: 800

nodes:
  - id: internet
    label: Internet
    node_type: cloud

  - id: fw
    label: Perimeter FW
    node_type: firewall

  - id: core-sw
    label: Core Switch
    node_type: l3_switch

  - id: srv1
    label: Web Server
    node_type: server

edges:
  - from: internet
    to: fw
```

```
    label: 1Gbps

  - from: fw
    to: core-sw
    label: 10Gbps

  - from: core-sw
    to: srv1
    label: 1Gbps
```

4 Architecture

NetVis is organized as a six-layer pipeline:

1. **Adapters.** Stateless converters that ingest topology data from external sources (YAML, NetBox [8] REST API, LLDP/CDP JSON) and produce a `NetVisGraph`. Each adapter implements the `Adapter` trait with three methods: `fetch`, `validate`, `to_graph`.
2. **Layout engines.** Pluggable algorithms behind a common `Layout` trait. A layout takes a `NetVisGraph` and returns a `Scene`—a flat list of positioned nodes, edges (as Bézier paths), and label anchors. Ten engines ship: force-directed (Standard and Enterprise), hierarchical (Sugiyama [18]), radial tree, geographic (Mercator), isometric multi-layer, stress majorization [12], multilevel coarsening [14], symmetry-aware, auto-tuner, and composite (multi-region).
3. **Refiners.** Post-layout transformations on the scene graph. Each refiner implements `LayoutRefiner` and may add, remove, or reposition scene elements. Current refiners: label placement, edge bundling (FDEB), crossing reduction, constraint solving, overlap removal, grid snapping, label relaxation, and seven edge routing refiners (smart/orthogonal/bus/arc/port-based; see §8).
4. **Renderers.** An SVG renderer (primary) and a Canvas2D renderer (for interactive browser use). Both support level-of-detail suppression and viewport culling.
5. **Export.** SVG output is optionally converted to PNG (via `resvg`), PDF (via `svg2pdf`), or self-contained interactive HTML (with embedded WASM binary and pan/zoom controls).
6. **Analysis.** Post-render modules: topology diffing (§9), temporal snapshots (§10), traffic flow animation (§11), and SVG annotation markup (§12).

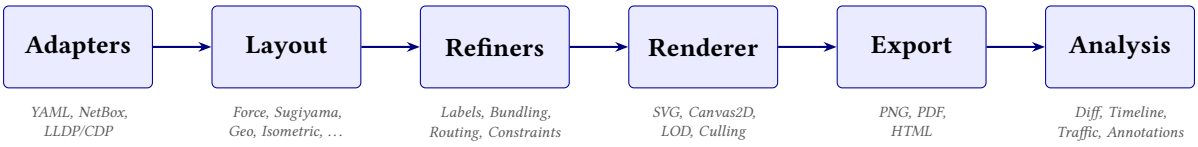


Figure 8. The NetVis six-layer pipeline.

4.1 Design Decision: Scene Graph as Intermediate Representation

Problem: Layout algorithms produce coordinates, but renderers need styled, labeled geometry. Where does styling happen?

Options considered:

(a) Style during layout (couples styling to layout code).

(b) Style during rendering (couples styling to renderer).

(c) Use an intermediate representation (`Scene`) that carries positioned, styled geometry.

Decision: Option (c). The `Scene` struct carries fully positioned nodes (with shapes, colors, labels) and edges (as Bézier paths). Refiners transform the scene between layout and rendering.

Consequences: Positive: renderers are simple (they just draw what the scene contains). Negative: the scene must carry all rendering data, increasing memory usage. At 5,000 nodes, the scene consumes approximately 9.7 MB of heap.

4.2 Design Decision: Trait-Based Composition

Problem: How should layout algorithms and refiners be composed?

Decision: All layout engines implement `trait Layout`, all refiners implement `trait LayoutRefiner`. A `LayoutPipeline` chains a layout engine with zero or more refiners:

Listing 5. Pipeline composition

```
let pipeline = LayoutPipeline::new(ForceDirectedLayout::new())
    .add_refiner(LabelPlacer::new())
    .add_refiner(BundlingRefiner::new(BundlingConfig::default()))
    .add_refiner(ConstraintSolver::new());

let scene = pipeline.layout(&graph)?;
```

Consequences: Positive: new algorithms and refiners can be added without modifying existing code. Negative: the trait boundary prevents cross-component optimization (e.g., bundling cannot influence label placement during the same pass). We mitigate this with `reposition_edge_labels()`, which re-places labels after bundling.

5 Layout Algorithms

5.1 Force-Directed Layout

5.1.1 Problem

Position nodes in 2D such that connected nodes are close and unconnected nodes are separated. The general case has no closed-form solution; iterative simulation is standard.

5.1.2 Design Options

(a) **Standard ($O(n^2)$):** Compute all-pairs repulsion directly [11]. Simple, deterministic, good for small graphs.

(b) **Barnes-Hut ($O(n \log n)$):** Approximate far-field repulsion using a spatial tree [2]. Faster for large graphs but adds tree-building overhead.

(c) **Multilevel:** Coarsen the graph, layout the coarsened version, then refine [14]. Best theoretical scaling but complex to implement correctly.

5.1.3 Decision

NetVis ships both (a) and (b) and automatically selects between them. Graphs under 2,000 nodes use the Standard engine (where quadtree overhead exceeds repulsion savings); larger graphs use the Enterprise engine.

5.1.4 Implementation

The Standard engine wraps the `fjadra` Rust library (Verlet integration [19] with spring-charge forces). The Enterprise engine is a custom implementation using a Barnes-Hut quadtree [2] with opening angle $\theta = 0.9$. On non-WASM targets, force calculation is parallelized via Rayon.

Key parameters:

- Velocity damping: $\gamma = 0.6$
- Alpha decay: $1 - 0.001^{1/300}$
- Iteration count: $\min(300, 50 + \lfloor n/10 \rfloor)$
- Minimum edge length: 60 pt (enforced post-simulation)

5.1.5 Usage

Listing 6. Force-directed layout

```rust
use netvis::{ForceDirectedLayout, ForceConfig, Layout};

// Standard engine (auto-selected for <2K nodes)
let layout = ForceDirectedLayout::new();
let scene = layout.layout(&graph)?;

// Explicit Enterprise engine
let config = ForceConfig::enterprise();
let layout = ForceDirectedLayout::with_config(config);
let scene = layout.layout(&graph)?;
```

5.2 Hierarchical Layout (Sugiyama)

5.2.1 Problem

Visualize DAG-structured or tiered networks (e.g., spine-leaf datacenters) with nodes arranged in horizontal layers and minimized edge crossings.

5.2.2 Implementation

The layout follows the four-phase Sugiyama framework [18]:

1. **Cycle removal.** Reverse edges to make the graph acyclic.
2. **Layer assignment.** Longest-path-first heuristic.
3. **Crossing minimization.** Barycenter or median heuristic (configurable).
4. **Coordinate assignment.** Assign x-coordinates to minimize edge bends.

Performance: 19 ms at 1,000 nodes.

5.3 Geographic Layout

5.3.1 Problem

Visualize WAN or multi-site topologies with nodes placed at their real-world geographic coordinates.

5.3.2 Implementation

Nodes with `latitude`/`longitude` attributes are projected using an equirectangular projection with dynamic scale normalization: $s = 1000 / \max(\Delta x, \Delta y)$. Edges are rendered as geodesic great-circle arcs (Vincenty approximation, 20 segments).

> **Antimeridian Wrapping**
>
> Edges crossing $\pm 180^\circ$ longitude exhibit a projection discontinuity. NetVis detects jumps exceeding 150 projected pixels and renders two copies of the path: one in the primary projection and one shifted by 360°. This is a pragmatic heuristic, not a formal solution.

5.4 Isometric Multi-Layer Layout

5.4.1 Problem

Visualize protocol stacks or layered architectures with a pseudo-3D perspective.

5.4.2 Implementation

Each layer runs an independent force-directed sub-layout. Results are projected isometrically with rotation angle $\phi = \pi/10$ (18°) and configurable layer spacing (default: 350 scene units). Layer planes render as filled SVG polygons.

> **Note**
>
> SVG layer planes must use RGB hex color + separate `fill-opacity` attribute (not 8-digit hex), because some SVG renderers double-apply alpha from 8-digit hex.

5.5 Radial Tree Layout

Best for hub-and-spoke topologies. Places a root node at the center with children radiating outward in concentric rings. Ring radii scale with subtree size to prevent overlap.

5.6 Stress Majorization Layout

5.6.1 Problem

Force-directed simulation can produce inconsistent results for graphs with strong symmetry or complex shortest-path structure. Stress majorization [12] directly minimizes the *stress* function—the sum of squared differences between geometric distances and ideal graph-theoretic distances.

5.6.2 Implementation

The layout computes all-pairs shortest paths (BFS for unweighted, Dijkstra for weighted graphs), then iteratively solves a system of weighted Laplacian equations. Convergence is typically faster than force simulation for moderate graphs (500–5,000 nodes).

Key parameters: `link_distance` (default: 100.0), `max_iterations` (default: 300), `epsilon` convergence threshold (default: 0.1).

5.6.3 Usage

Listing 7. Stress majorization layout

```
use netvis::{StressMajorizationLayout, StressConfig, Layout};

let config = StressConfig { link_distance: 120.0, ..Default::default() };
let layout = StressMajorizationLayout::new(config);
let scene = layout.layout(&graph)?;
```

5.7 Multilevel Layout

5.7.1 Problem

Force-directed layouts plateau at ~10,000 nodes due to $O(n^2)$ or $O(n \log n)$ per-iteration cost. Multilevel coarsening [14] reduces the problem size before layout, then refines the result.

5.7.2 Implementation

Three phases:

1. **Coarsen.** Heavy-edge matching contracts pairs of adjacent nodes into super-nodes, repeating until the graph has fewer than 100 nodes.
2. **Base layout.** The coarsened graph is laid out using force-directed simulation.
3. **Refine.** Super-nodes are expanded back to their constituents; positions are interpolated and locally relaxed.

5.7.3 Usage

Listing 8. Multilevel layout

```
use netvis::{MultilevelLayout, Layout};

let layout = MultilevelLayout::new();
let scene = layout.layout(&graph)?;  // auto-coarsens, layouts, refines
```

5.8 Symmetry-Aware Layout

5.8.1 Problem

Network topologies frequently contain topologically regular substructures—rings (metro SONET), stars (hub-and-spoke), and cliques—that force simulation distorts into irregular shapes.

5.8.2 Implementation

The layout pre-processes the graph to detect star and ring substructures and locks them into geometrically perfect shapes (regular polygons for rings, equidistant radial placement for stars) before delegating remaining nodes to a base layout engine.

5.9 Auto-Tuner Layout

5.9.1 Problem

Force-directed layout quality depends heavily on `charge_strength` and `link_distance` parameters, which vary by topology structure. Manual tuning is tedious.

5.9.2 Implementation

Wraps any layout engine and automatically tunes parameters over multiple passes using iterative hill-climbing to minimize a composite objective of node overlap count and edge-length variance.

5.9.3 Usage

Listing 9. Auto-tuner layout

```
use netvis::{AutoTunerLayout, ForceDirectedLayout, Layout};

let base = ForceDirectedLayout::new();
let layout = AutoTunerLayout::new(base);
let scene = layout.layout(&graph)?;  // runs multiple passes
```

5.10 Composite Layout

5.10.1 Problem

Real-world networks combine different structural paradigms: a WAN backbone has geographic placement, while each site's internal fabric is hierarchical. A single layout algorithm cannot serve both.

5.10.2 Implementation

Assigns different layout algorithms to different sub-graphs within a single canvas. A `CompositeConfig` maps group IDs to layout engines; a parent-level layout positions groups relative to each other.

5.10.3 Usage

Listing 10. Composite multi-region layout

```
use netvis::{CompositeLayout, CompositeConfig, Layout};

let config = CompositeConfig::new()
    .group("wan", "geographic")
    .group("dc-east", "hierarchical")
    .group("dc-west", "hierarchical")
    .parent_layout("force_directed");
```

```
let layout = CompositeLayout::new(config);
let scene = layout.layout(&graph)?;
```

When used with the `--cluster` CLI flag, NetVis applies community detection (Louvain, label propagation, or edge betweenness) to automatically partition the graph into groups, which are then visualized with group boundary overlays.

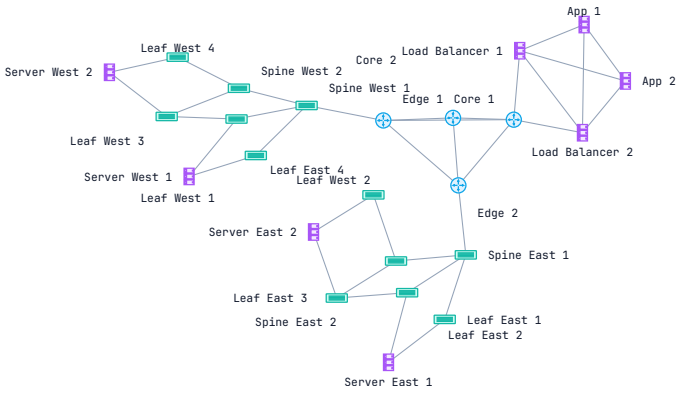


Figure 9. Automatic clustering with community-detection group boundaries. The Louvain algorithm partitions nodes into communities; each community receives a distinct color overlay and boundary polygon.

6 Label Placement

6.1 Problem

Place text labels near their owner (node or edge) without overlapping other labels, nodes, or edges. This is a variant of the NP-hard point-feature label placement (PFLP) problem [7].

6.2 Design Options

(a) **No collision avoidance** (Graphviz [10] approach): place labels at fixed positions; accept overlaps.

(b) $O(n^2)$ **greedy:** For each label, check all previously placed labels for collision.

(c) **R*-tree–accelerated greedy:** Use a spatial index for $O(\log n)$ collision queries.

6.3 Decision

Option (c). The R*-tree [3] build cost (~5 ms) is recouped after approximately 20 collision queries, which occurs by ~50 labels. The spatial index is implemented via the `rstar` crate [17].

6.4 Algorithm

NetVis uses a two-phase greedy placement:

Phase 0 (edge labels): Edge labels are placed first because they are more constrained—each must sit near its edge's midpoint. For curved edges, the midpoint is computed via arc-length sampling. A deviation threshold $\delta = 0.08$ prevents labels from clustering at bundle convergence points.

Phase 1 (node labels): Node labels are placed around each node, avoiding already-placed edge labels. For each node, the algorithm evaluates 8 candidate positions (4 cardinal + 4 diagonal) and selects the highest-scoring one.

Scoring function:

$$S(c, \ell) = \exp(-\alpha \cdot d(c, \ell)) - \beta \sum_{r \in R(c)} A_{\text{overlap}}(c, r) \cdot w(r)$$

Where:

- $d(c, \ell)$: distance from candidate position to label owner
- $\alpha = 1.0$: distance decay rate
- $\beta = 300.0$: overlap penalty weight
- $w(r) = 5.0$ for labels/nodes (hard collision), 0.3 for edge strokes (soft collision)

Edge exclusion zone optimization: Curved/bundled edges can produce 47,000+ segment AABBs. NetVis skips per-segment exclusion zone registration for non-straight edges, registering only a single bounding-box exclusion per curved path.

6.5 Performance

Nodes	Time	Overlaps	Avg. Distance
100	<1 ms	0	8.2 pt
500	12 ms	0	9.1 pt
1000	48 ms	2	10.3 pt
5000	210 ms	18	12.7 pt

Table 1. Label placement performance on spine-leaf topologies.

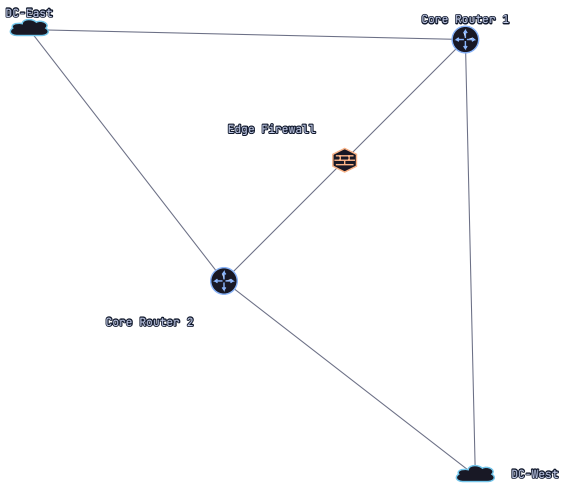


Figure 10. Arc-length label placement on curved edges. Labels are positioned at the midpoint of each edge's arc length rather than its geometric centroid, producing readable text even on great-circle arcs and bundled paths.

7 Edge Bundling

7.1 Problem

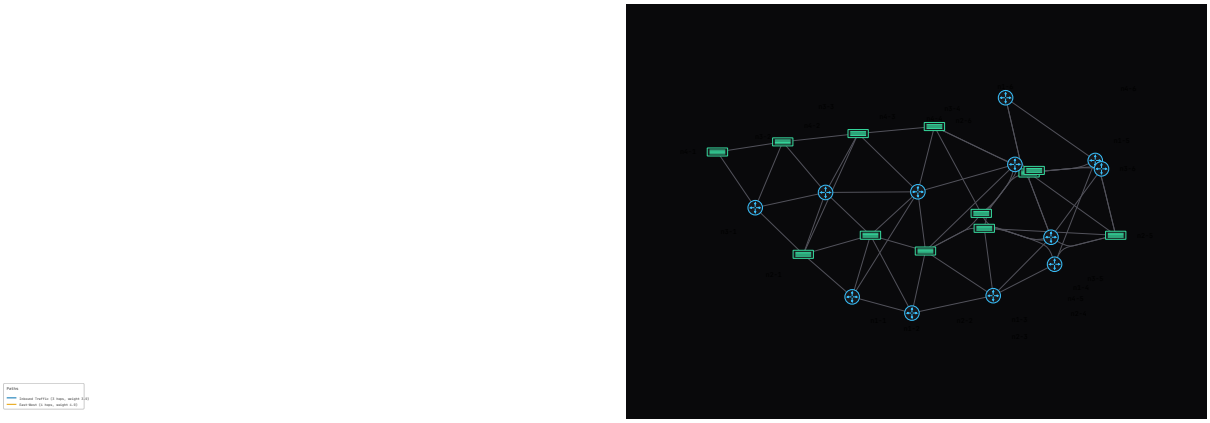
Dense graphs produce visual clutter from crossing edges. Edge bundling groups spatially compatible edges into bundles, reducing clutter.

7.2 Implementation

NetVis implements a variant of Holten and Van Wijk's Force-Directed Edge Bundling (FDEB) [15]. Our variant differs from the original in three ways:

1. **Singularity guard:** Force model uses $f = k_c/(d + 1.0)$ (not $d + 0.1$), preventing hairball artifacts when compatible edges nearly overlap.
2. **Spring preservation:** Spring coefficient $1.5k$ (not $0.5k$) maintains edge shape under bundling.
3. **Post-processing:** Catmull-Rom spline (tension 0.4) + 3–5 passes of Laplacian smoothing ($\alpha = 0.5$) + decimation filter (drop points <15 px apart).

> **Note**
>
> Bundling runs *after* label placement. A `reposition_edge_labels()` pass re-places labels on edges whose curvature changed significantly. This ordering is intentional: labels should avoid bundled paths, not the other way around.



(a) FDEB bundling on a spine-leaf topology

(b) Edge bundling showcase with Laplacian smoothing

Figure 11. Force-directed edge bundling (FDEB) reduces visual clutter by grouping spatially compatible edges. Post-processing with Catmull-Rom splines and Laplacian smoothing produces clean, readable bundles.

8 Edge Routing Refiners

In addition to FDEB bundling, NetVis provides seven edge routing refiners, each implementing the `LayoutRefiner` trait:

SmartRoutingRefiner. Bends edges around node and label obstacles using an R*-tree–based visibility graph with A* pathfinding. Resulting waypoints are smoothed into Bézier splines.

OrthogonalRoutingRefiner. Converts edges to axis-aligned Manhattan paths with configurable corner radius.

BusRoutingRefiner. Detects star-hub nodes and converts spoke edges into a shared backbone bus line with perpendicular stubs.

ArcRoutingRefiner. Curves straight edges into circular arcs. Configurable arc height and direction.

PortDockingRefiner. Adjusts edge endpoints to dock at specific interface/port positions on node boundaries, distributing connections evenly around the perimeter.

CrossingReductionRefiner. Post-layout heuristic that swaps node positions within the same layer to reduce edge crossings.

GridSnapRefiner. Snaps node positions to a configurable grid for a clean, aligned appearance.

(a) Bus routing: shared backbone with perpendicular stubs **(b)** Bubble-set group boundaries with smart routing

Figure 12. Edge routing refiners transform straight edges into structured paths. The bus refiner (a) consolidates star-hub spokes into a shared bus line; bubble sets (b) define smooth, convex group regions.

9 Topology Diffing

NetVis can structurally compare two `NetVisGraph` instances and produce a visual diff. The `diff` module computes:

- Added, removed, and modified nodes (by ID and attribute comparison).
- Added, removed, and modified edges.

The `apply_diff_styles()` function color-codes the rendered scene: green for added elements, red/dimmed for removed, amber for modified. A diff legend is automatically inserted.

Listing 11. CLI topology diff

```
netvis --input new.yaml --diff old.yaml -o diff.svg
```

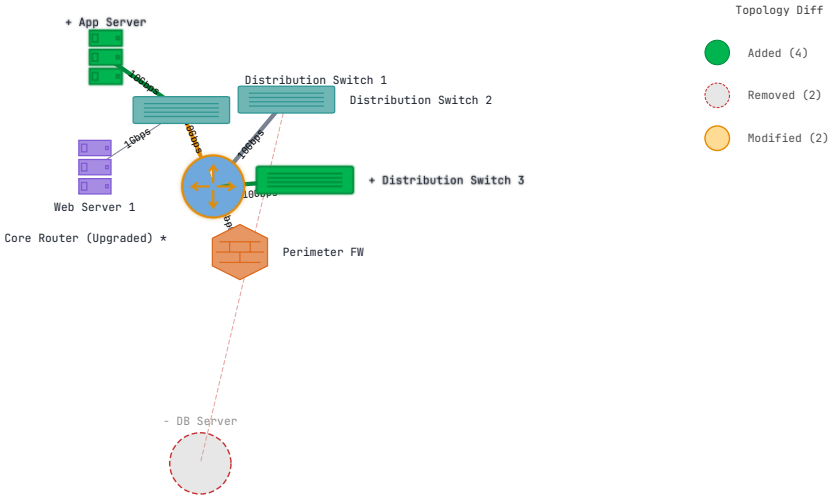


Figure 13. Topology diff output. Added elements are shown in green, removed elements in red/dimmed, and modified elements in amber. A legend is automatically inserted into the rendered SVG.

10 Temporal Analysis

The `timeline` module tracks topology evolution over time.

10.1 Snapshot Discovery

`discover_snapshots()` scans a directory of YAML topology files, extracts timestamps from filenames or metadata, and returns them in chronological order as `SnapshotEntry` values.

10.2 Entity History

`build_node_history()` and `build_edge_history()` trace an entity's lifecycle across snapshots: when it first appeared, all attribute changes, and whether it has been removed. Results are returned as `EntityHistory` with a vector of `ChangeRecord` entries.

10.3 CLI Subcommands

Listing 12. Timeline CLI

```
# List all snapshots in a directory
netvis timeline --dir snapshots/

# Query history of a specific node
netvis query --dir snapshots/ --node router-fra-01
```

11 Traffic Flow Animation

The `traffic` module injects animated flow overlays into rendered SVGs.

11.1 Data Model

Each edge can carry traffic metadata:

Listing 13. Traffic configuration

```
traffic:
  edges:
    - id: edge-core-to-agg
```

```
      utilization: 0.75
      direction: bidirectional
      style: dot
```

11.2 Rendering

`animate_svg()` post-processes the SVG string to inject:

- CSS `@keyframes` for dot/dash flow along edge paths.
- Utilization-to-color mapping: green (low) → amber → red (congested).
- Play/pause and speed controls as an SVG overlay.
- Zero-traffic edges receive CSS dimming (`opacity: 0.3`).

Figure 5 shows a rendered traffic overlay from Example 2 (§2).

12 Annotation Markup

The `annotation` module injects text annotations and callouts into rendered SVGs.

Listing 14. Annotation configuration

```
annotations:
  - anchor: router-fra-01
    type: callout
    text: "Primary BGP peer"
    shape: box
    layer: notes
```

Annotations support layer filtering (`show_only_layers`, `hidden_layers`), pointer lines from callout to anchor node, and shapes: circle, box, and arrow.

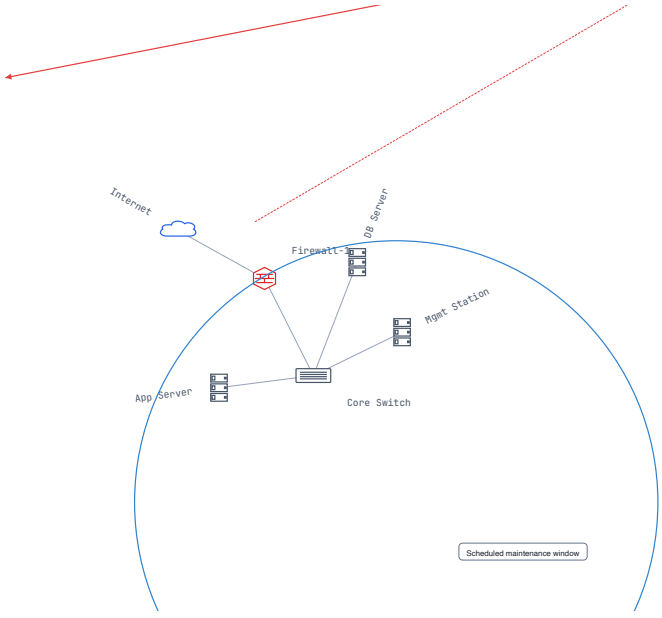


Figure 14. Annotation markup overlay on a NOC topology. Callout annotations are anchored to specific nodes with pointer lines, supporting box, circle, and arrow shapes.

13 Interactive Editor

The `editor` module provides interactive topology editing for browser deployments via the WASM target.

13.1 Command Pattern

All mutations are represented as `Command` objects: `MoveNodeCommand`, `TransactionCommand`, etc. Each command stores only a delta (e.g., 24 bytes for a node move—source and destination coordinates), not a full graph snapshot. An `UndoStack` with 50-action capacity provides undo/redo.

> **Note**
>
> Delta-based undo uses ~41,000x less memory than snapshot-based undo for a 5,000-node topology.

13.2 Web Component

A Lit-based `<netvis-editor>` Web Component wraps the WASM editor:

Listing 15. Web Component usage

```
<netvis-editor
  topology="campus.yaml"
  layout-algorithm="force-directed"
  read-only="false">
</netvis-editor>
```

Provides declarative attributes, an imperative API (`setTopology()`, `resetView()`, `exportSVG()`), and JSX type augmentations for React/Vue integration.

14 Visual Effects

NetVis provides a pluggable visual effects system that maps semantic effect names to SVG filter definitions.

14.1 Effect Registry

The `EffectRegistry` deduplicates identical filter definitions via content-addressed hashing. Effects are assigned to nodes or edges via the `highlight` field in YAML:

Listing 16. Assigning effects in YAML

```
nodes:
  - id: core-router
    highlight: true          # resolves to "glow-selection"
  - id: alert-node
    highlight: "emphasis"    # resolves to "glow-emphasis"
  - id: combo-node
    highlight: "stack:drop-shadow-subtle+glow-selection"
```

14.2 Built-In Effects

Name	Type	Description
`drop-shadow-subtle`	Drop shadow	Light offset shadow
`drop-shadow-emphasis`	Drop shadow	Strong offset shadow
`inner-shadow-boundary`	Inner shadow	Inset shadow for group boundaries
`glow-selection`	Glow	Selection highlight
`glow-status`	Glow	Status indicator
`glow-emphasis`	Glow	Emphasis marker
`stack:<a>+<b>`	Composed	Combine any two base effects

Table 2. Built-in visual effects.

14.3 Theme Adaptation

All effects adapt to the active theme. Glow intensity, rim sharpness, and shadow direction vary by theme family:

- **Light/Print:** Crisp rim, slightly stronger glow.
- **Dark/Network:** Softer, reduced intensity (no rim).
- **Blueprint:** Technical ink-like rim with soft halo.
- **High-contrast:** Tighter blur, strongest opacity.

Effect Budget

SVG filter performance degrades at ~50+ filtered elements. The `effects.max_filtered_elements` config key (default: 50) sets a budget; exceeding it emits a `BudgetExceeded` warning and auto-disables further filters.

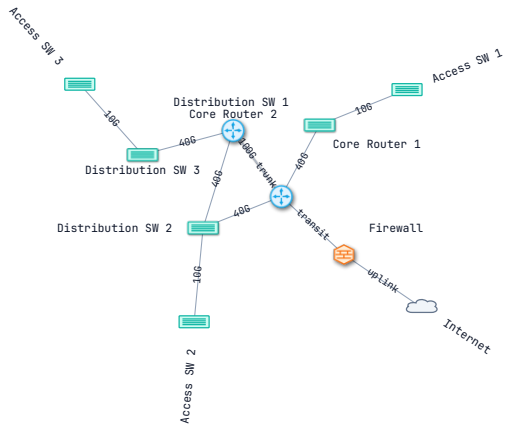


Figure 15. Visual effects showcase (light theme). Nodes demonstrate drop shadows, glow selection, emphasis glow, and composed `stack:drop-shadow+glow` effects. All effects adapt their intensity and color to the active theme.

15 Quality Metrics

NetVis computes quality metrics as part of every render, following the aesthetic metrics framework of Purchase [16] and Dunne et al. [9], producing a `DiagnosticsEnvelope` alongside the output.

15.1 Metrics

NetVis computes 16 metric families, aggregated into a unified `AestheticScore` (0–100) via `src/quality/scoring.rs`:

Edge crossings. R*-tree–accelerated segment intersection testing ($O(E \log E)$). Crossings are weighted by angle: orthogonal crossings receive weight 1.0; acute crossings receive higher penalties.

Overlap area. AABB intersection between all label/node pairs. Sub-pixel overlaps (<2 px) are ignored.

Label readability. Composite of label–owner distance, font-size variance, and left-to-right reading order.

Angular resolution. Minimum angle between consecutive incident edges at each node.

Node alignment. Degree to which nodes align on axes or grid lines.

Edge clearance. Minimum distance from non-incident edges to nodes.

Coincident edges. Detection of duplicate or overlapping edge paths.

Contrast. Label color contrast against background (WCAG-derived).

Edge-length variance. Coefficient of variation across all edge lengths; lower is better.

Spatial distribution. Evenness of node distribution across the canvas (entropy-based).

Path straightness. Curvature penalty for bent edges.

Viewport coverage. Whether elements extend beyond viewport bounds.

Node–edge overlap. Non-incident edge segments crossing node bounding boxes.

Density heatmap. Spatial density heatmap for identifying visual hotspots.

15.2 Topology Linting

Separate from aesthetic quality, the `lint` module performs structural topology checks: isolated nodes, disconnected subgraphs, duplicate edge IDs, and missing labels. Lint results appear as `Warning`-severity entries in the `DiagnosticsEnvelope`.

15.3 CI Regression Detection

Quality baselines are stored in `benches/regression_baselines.json`. CI fails if any metric degrades by more than 20%. This threshold accounts for runner variability while catching genuine regressions.

16 Rendering Pipeline

16.1 SVG Renderer

The primary renderer produces SVG using the `svg` crate. Elements are ordered by the painter's model: background (map layers) → group boundaries → edges → nodes → labels → effects.

16.2 Level of Detail

LOD suppresses visual details at configurable thresholds, following the approach of Abello et al. [1]:

Profile	Labels	Edge Lbl.	Interfaces	Effects
Conservative	2000	1000	1000	500
Balanced	500	300	300	200
Aggressive	200	100	100	50

Table 3. LOD suppression thresholds (elements suppressed when count exceeds value).

LOD is applied at render time without mutating the scene. “Auto” mode selects a profile based on node count.

16.3 Viewport Culling

For pan/zoom scenarios, elements outside the viewport (plus a configurable margin) are skipped during rendering. Applied at render time.

16.4 Interactive HTML Export

The `export::html` module generates self-contained HTML files with the WASM binary embedded as base64. The resulting file can be opened directly from `file://` without a web server, providing pan, zoom, and node-drag interactivity.

Listing 17. Interactive HTML export

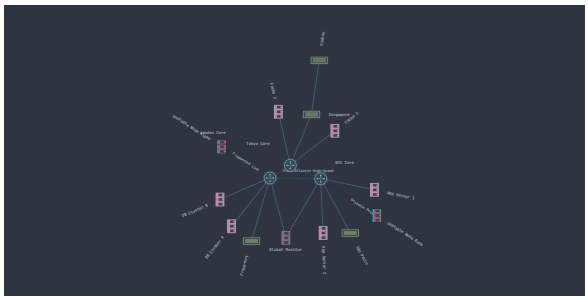
```
netvis --input topology.yaml --format html -o topology.html
```

16.5 Progressive Rendering

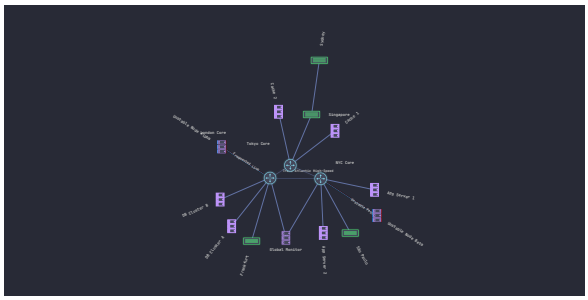
For very large topologies, the renderer can operate in progressive mode: elements are rendered in chunks with progress callbacks and cooperative cancellation. This prevents blocking the UI thread in WASM deployments.

16.6 Theme Gallery

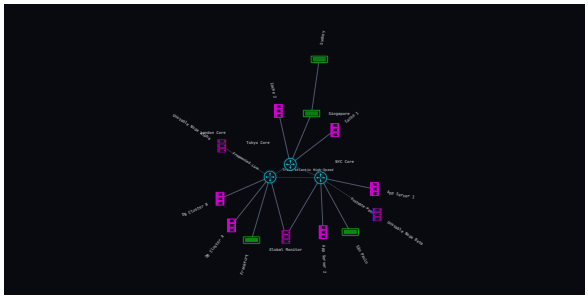
NetVis ships with 16 built-in themes. Figure 16 shows the same enterprise campus topology rendered with four different themes, demonstrating how the rendering pipeline adapts colors, glows, shadows, and label styling to each theme family.



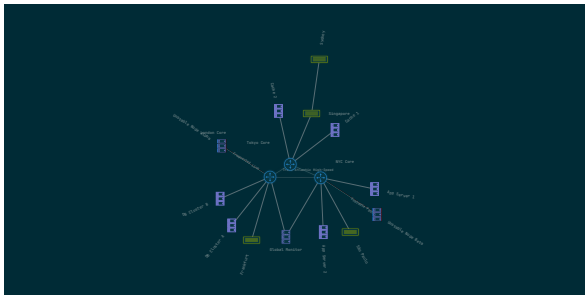
(a) Nord



(b) Dracula



(c) Cyberpunk



(d) Solarized Dark

Figure 16. Theme comparison: the same topology rendered with four of NetVis's 16 built-in themes. Each theme adapts node icons, edge colors, label styling, background, glow intensity, and shadow direction.

17 API and Usage

17.1 End-to-End Example

Listing 18. Complete example: graph construction through rendering

```rust
use netvis::contracts::v1 as nv;
use netvis::{EdgeData, NetVisGraph, NodeData, SCHEMA_VERSION};

// Enable deterministic mode for reproducible output
let _det = nv::DeterministicMode::enable();

// Build graph
let mut graph = NetVisGraph::new();
let r1 = graph.add_node(NodeData::new("router1").node_type("router"));
let s1 = graph.add_node(NodeData::new("switch1").node_type("switch"));
let s2 = graph.add_node(NodeData::new("switch2").node_type("switch"));
let srv = graph.add_node(NodeData::new("server1").node_type("server"));
graph.add_edge(r1, s1, EdgeData::default().label("10G"));
graph.add_edge(r1, s2, EdgeData::default().label("10G"));
graph.add_edge(s1, srv, EdgeData::default().label("1G"));
graph.add_edge(s2, srv, EdgeData::default().label("1G"));

// Render
let options = nv::RenderOptions::default()
    .format(nv::OutputFormat::Svg)
    .width(1200.0)
    .height(800.0);

let envelope = nv::render(&graph, &options)?;

// Access output
let svg_string = envelope.output.as_svg_string();
let warnings = envelope.diagnostics.count_by_severity(nv::Severity::Warning);
println!("Rendered {} nodes, {} warnings", graph.node_count(), warnings);
```

17.2 CLI Usage

Listing 19. Common CLI commands

```
# Render a topology
netvis --input topology.yaml -o output.svg

# Use hierarchical layout
netvis --input topology.yaml --preset sugiyama

# Dark theme, large canvas
netvis --input topology.yaml --theme dark --width 1920 --height 1080

# Export to PNG or interactive HTML
netvis --input topology.yaml --format png --dpi 150
netvis --input topology.yaml --format html -o topology.html

# Topology diff
netvis --input new.yaml --diff old.yaml -o diff.svg

# Filter nodes before rendering
netvis --input topology.yaml --include-node "router-*" -o filtered.svg
netvis --input topology.yaml --path-from core --path-to edge -o path.svg

# Traffic animation overlay
netvis --input topology.yaml --traffic traffic.yaml -o animated.svg

# Annotation markup
netvis --input topology.yaml --annotations notes.yaml -o annotated.svg

# Timeline and query subcommands
netvis timeline --dir snapshots/
netvis query --dir snapshots/ --node router-fra-01
```

17.3 WASM / JavaScript

Listing 20. JavaScript integration

```
import { init, renderToSVG, renderWithDiagnostics,
         createInteractive } from '@netvis/core';

await init();

// Static SVG
const svg = await renderToSVG(topologyJson);
document.getElementById('viz').innerHTML = svg;

// SVG + diagnostics (recommended for embedders)
const { output, diagnostics } = await renderWithDiagnostics(topologyJson);

// Interactive (pan, zoom, drag)
const viz = await createInteractive(topologyJson, '#viz-container');
viz.resetView();
```

A background Web Worker API (`layoutWorkerRun`) is also available for offloading layout computation with progress callbacks that stream intermediate node positions.

17.4 Web Component

A Lit-based `<netvis-editor>` Web Component provides a declarative integration path:

Listing 21. Declarative Web Component

```
<netvis-editor topology="campus.yaml"
               layout-algorithm="force-directed">
</netvis-editor>
```

See §13 for the imperative API.

17.5 API Quick Reference

Function	Input	Returns
`nv::render`	Graph, Options	RenderEnvelope
`Layout::layout`	Graph	Scene
`SvgRenderer::render`	Scene	SVG string
`render_to_png`	SVG, PngOptions	PNG bytes
`render_to_pdf`	SVG, PdfOptions	PDF bytes
`ExportBuilder::build`	Config	Multi-format output

Table 4. Core API entry points.

18 Deployment

NetVis compiles from a single Rust codebase to three targets:

Native CLI. `cargo build –release –features cli`. Full feature set including HTTP-based adapters (NetBox), parallel layout (Rayon), and file I/O.

WebAssembly [13]. `wasm-pack build –target web`. Single-threaded, no file I/O. Canvas + SVG rendering. Automatic engine selection and LOD apply as in native mode.

Python. `maturin build –features python`. PyO3 with ABI3 stability (Python 3.8+).

Feature gates (`cli`, `wasm`, `python`) control which dependencies are included. WASM-incompatible crates (`reqwest`, `tokio`, `rayon`) are gated to native-only.

18.1 Deterministic Mode

For CI/CD pipelines, deterministic mode ensures reproducible output:

```
let _det = DeterministicMode::enable();
// All HashMap iterations sorted, RNG seeded, layout deterministic
```

19 Performance

Measured on Apple M4 Pro, 24 GB RAM, macOS, Rust 1.93.

Operation	100	500	1K	2K	5K
Force-directed layout	20 ms	144 ms	309 ms	816 ms	1.75 s
Hierarchical layout	1.5 ms	8 ms	19 ms	—	—
SVG render	0.8 ms	4.5 ms	11 ms	76 ms	223 ms
Peak heap allocation	215 KB	952 KB	1.9 MB	3.8 MB	9.7 MB

Table 5. Performance baselines by topology size (spine-leaf topologies).

10,000-node spine-leaf (Enterprise engine, auto LOD): 5.3 s end-to-end.

Tool	100 nodes	1K nodes	2K nodes
Graphviz `sfdp`	84 ms	250 ms	914 ms
Graphviz `neato`	88 ms	1,024 ms	4,434 ms
NetVis (full pipeline)	**24 ms**	**330 ms**	**620 ms**

Table 6. Comparative benchmark on identical spine-leaf topologies. NetVis times include layout, label placement, styling, and SVG rendering.

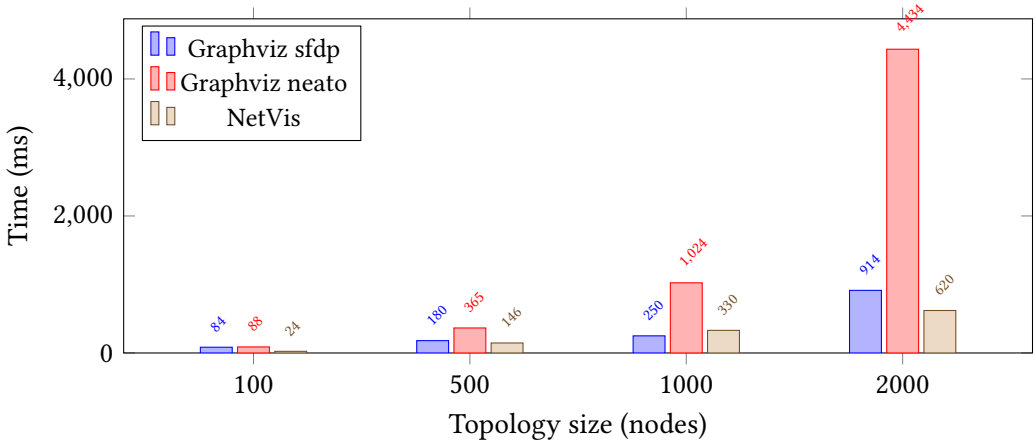


Figure 17. Comparative performance on identical spine-leaf topologies. NetVis includes the full pipeline (layout, labels, styling, SVG render); Graphviz includes layout and SVG output only. Apple M4 Pro, 24 GB RAM, best of 3 runs.

20 Limitations

20.1 Scale Ceiling at ~10,000 Nodes

Description: The Enterprise engine handles 10,000 nodes in ~5 s. The multilevel layout (§5) improves scaling through graph coarsening, but 50,000+ nodes would benefit from GPU-accelerated force calculation.

Workaround: Use multilevel, hierarchical, or geographic layout (all scale better than standard force-directed), or apply LOD aggressively.

Planned resolution: GPU layout via WebGPU compute shaders is under investigation.

20.2 Greedy Label Placement

Description: Labels are placed sequentially; earlier placements constrain later ones. There is no global optimization. In pathological cases (very dense clusters), some labels may overlap.

Workaround: Suppress labels via LOD at high density, or increase canvas dimensions.

Planned resolution: A simulated-annealing post-pass is planned.

20.3 Single-Threaded WASM

Description: The WASM target cannot use Rayon for parallel force calculation. Large layouts are slower in the browser than natively.

Workaround: Use Web Workers for background layout computation (the WASM API supports this pattern).

Planned resolution: WebAssembly threads (SharedArrayBuffer) once browser support stabilizes.

20.4 Non-Determinism Without Explicit Mode

Description: By default, HashMap iteration order is non-deterministic. This affects layout reproducibility.

Workaround: Enable `DeterministicMode` for CI and testing.

20.5 Adapter Coverage

Description: Only YAML, NetBox, and LLDP/CDP adapters are built in. No SNMP, streaming telemetry, or intent-based sources.

Workaround: Export topology from your tool to YAML and feed it to NetVis.

Planned resolution: The `Adapter` trait is public; community adapters are encouraged.

20.6 No User Study Validation

Description: Quality claims are based on automated metrics (crossings, overlaps, readability scores), not validated by perceptual user studies with network engineers.

Planned resolution: A controlled user study comparing NetVis output with Graphviz and hand-drawn diagrams is planned.

21 Future Work

1. **GPU-accelerated layout.** WebGPU compute shaders for 50,000+ node topologies.
2. **Streaming topology ingestion.** The timeline module (§10) supports file-based snapshot analysis; live gNMI/gRPC streaming would extend this to real-time updates.
3. **Simulated-annealing label post-pass.** Global optimization to resolve remaining overlaps in dense clusters.
4. **User study.** Controlled perceptual evaluation with network engineers.
5. **Collaborative editing.** The editor module (§13) provides single-user undo/redo; CRDT-based multi-user conflict resolution is the next step.
6. **SNMP and streaming telemetry adapters.** The `Adapter` trait is public; community adapters for SNMP, Napalm, and gNMI are encouraged.

References

[1] James Abello, Frank Van Ham, and Neeraj Krishnan. ASK-GraphView: A large scale graph visualization system. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):669–676, 2006.

[2] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, 19(2):322–331, 1990.

[4] bluss. petgraph: Graph data structure library for Rust. https://github.com/petgraph/petgraph, 2024.

[5] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3: Data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[6] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2013.

[7] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.

[8] DigitalOcean. NetBox: The leading solution for modeling and documenting modern networks. https://netbox.dev, 2024.

[9] Cody Dunne, Sharon I. Ross, Ben Shneiderman, and Mauro Martino. Readability metric feedback for aiding node-link visualization designers. *IBM Journal of Research and Development*, 59(2/3):14:1–14:16, 2015.

[10] John Ellson, Emden R. Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz — open source graph drawing tools. *Lecture Notes in Computer Science*, 2265:483–484, 2002.

[11] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[12] Emden R. Gansner, Yifan Hu, and Stephen C. North. A sparse stress model. *Journal of Graph Algorithms and Applications*, 17(1):21–46, 2013.

[13] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices*, 52(6):185–200, 2017.

[14] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. *Lecture Notes in Computer Science*, 3383:285–295, 2005.

[15] Danny Holten and Jarke J. Van Wijk. Force-directed edge bundling for graph visualization. *Computer Graphics Forum*, 28(3):983–990, 2009.

[16] Helen C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, 2002.

[17] Adam Stocker. rstar: R*-tree spatial index for Rust. https://github.com/georust/rstar, 2024.

[18] Kozo Sugiyama, Shojiro Tagawa, and Misue Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[19] Loup Verlet. Computer "experiments" on classical fluids. *Physical Review*, 159(1):98–103, 1967.

[20] yWorks GmbH. yEd graph editor. https://www.yworks.com/products/yed, 2024.

A Topology YAML Schema

Listing 22. Full YAML topology schema (abbreviated)

```
name: string              # Required: topology name
render:                   # Optional: rendering hints
```

```
  preset: string          #   Layout preset (default|sugiyama|radial|geo)
  theme: string           #   Theme (network|dark|light)
  width: number           #   Canvas width in pixels
  height: number          #   Canvas height in pixels
  lod: string             #   LOD profile (none|conservative|balanced|aggressive|auto)
nodes:                    # Required: list of nodes
  - id: string            #   Unique node identifier
    label: string         #   Display label (optional, defaults to id)
    node_type: string     #   Device type (router|switch|server|firewall|cloud)
    attributes: map       #   Arbitrary key-value metadata
    position:             #   Position hint (optional)
      x: number
      y: number
    group: string         #   Group membership (optional)
    layer: string         #   Layer membership (optional)
edges:                    # Required: list of edges
  - from: string          #   Source node ID
    to: string            #   Target node ID
    id: string            #   Explicit edge ID (optional, for traffic/filter targeting)
    label: string         #   Edge label (optional)
    weight: number        #   Edge weight (default: 1.0)
    directed: boolean     #   Directed edge (default: false)
    attributes: map       #   Arbitrary key-value metadata
groups:                   # Optional: group definitions
  - id: string
    label: string
    style: map            #   Boundary style overrides
layers:                   # Optional: layer definitions for isometric layout
  - id: string
    label: string
    order: number         #   Stacking order (lower = bottom)
traffic:                  # Optional: traffic flow animation overlay
  edges:
    - id: string          #   Edge ID (matches edges[].id)
      utilization: number #   0.0--1.0 utilization level
      direction: string   #   forward|reverse|bidirectional
      style: string       #   dot|dash animation style
annotations:              # Optional: SVG annotation markup
  - anchor: string        #   Node ID to anchor near
    type: string          #   text|callout
    text: string          #   Annotation text
    shape: string         #   circle|box|arrow (callout only)
    layer: string         #   Annotation layer name
```

B Configuration Reference

B.1 CLI Flags

B.1.1 Input and Output

Flag	Type	Default	Description
`--input, -i`	path	(required)	Topology file; globs and - for stdin
`--output, -o`	path	auto	Output file path
`--config`	path	—	JSON config file
`--format`	string	`svg`	Output format: svg, png, pdf, html
`--dpi`	integer	96	DPI for PNG export

B.1.2 Layout

Flag	Default	Description
`--layout`	auto	Algorithm: force-directed, hierarchical, radial, isometric, geographic
`--preset`	—	Layout preset: default, wan, datacenter, sugiyama, geographic, …
`--template`	—	Topology template: spine-leaf, hub-spoke, ring, multi-tier
`--seed`	42	RNG seed for deterministic layout

B.1.3 Appearance

Flag	Default	Description
`--theme`	`network`	Theme (16 built-in): light, dark, network, blueprint, datacenter, print, high-contrast, solarized-{dark,light}, nord, catppuccin, brewer, dracula, tokyo-night, monokai, cyberpunk
`--labels`	`label`	Label mode: off, label, id
`--node-size`	auto	Node radius in scene units
`--lod`	auto	LOD profile: none, conservative, balanced, aggressive, auto

B.1.4 Canvas and Viewport

Flag	Default	Description
`--width`	1200	SVG viewport width (px)
`--height`	800	SVG viewport height (px)
`--padding`	40	ViewBox padding (px)
`--padding-ratio`	0.06	Padding as fraction of scene

B.1.5 Force-Directed Tuning

Flag	Default	Description
`--link-distance`	30.0	Target link length (scene units)
`--link-strength`	auto	Edge spring strength (0.0–1.0)
`--charge-strength`	−30.0	Many-body charge (negative = repel)
`--max-iterations`	300	Maximum simulation iterations
`--soft-anchor-strength`	0.0	Anchor strength for hinted nodes
`--refine`	false	Apply post-layout edge refinement

B.1.6 Geographic Layout

Flag	Default	Description
`--show-map`	false	Show background coastline map
`--great-circle-arcs`	false	Render edges as curved arcs
`--show-distances`	false	Show geodesic distance labels
`--distance-unit`	km	Distance unit: km or mi

B.1.7 Filtering

Flag	Description
`--show-type`	Only render nodes of these types (comma-separated, repeatable)
`--hide-type`	Hide nodes of these types
`--show-tag`	Only render nodes with these tags
`--hide-tag`	Hide nodes with any of these tags
`--show-group`	Only render nodes in these groups
`--hide-group`	Hide nodes in these groups
`--filter-file`	Load filter rules from YAML file
`--path`	BFS path query between two node IDs
`--highlight-path`	Highlight a named path (repeatable)

B.1.8 Analysis Features

Flag	Description
`--diff <path>`	Compare against a base topology and visualize changes
`--traffic <path>`	Traffic overlay configuration YAML
`--annotations <path>`	Annotation markup YAML
`--hide-layers`	Suppress named annotation layers
`--show-layers`	Force-show named annotation layers
`--cluster`	Enable automatic community detection
`--algorithm`	Clustering algorithm: louvain, label-propagation, edge-betweenness
`--metric <field>`	Map a node attribute to color

B.1.9 Quality and Diagnostics

Flag	Default	Description
`--quality-threshold`	85.0	Quality score warning threshold
`--fail-on-low-quality`	false	Exit with error if below threshold
`--deterministic`	true	Deterministic output mode
`--diagnostics`	—	Structured diagnostics: json, yaml, ndjson
`--verbose`, `-v`	false	Show debug statistics

B.2 CLI Subcommands

Command	Description
`list-examples`	List built-in topology examples
`print-example <name>`	Print a built-in topology to stdout
`list-templates`	List layout templates
`list-adapters`	List data source adapters
`import`	Import topology from adapter into YAML
`schema`	Print JSON Schema (config, topology, or diagnostics)
`print-config <name>`	Print an example config to stdout
`validate`	Validate topology/config against schema
`migrate`	Normalize files to current schema version
`check-quality`	Analyze SVG files for quality issues
`diff`	Compare two topologies, visualize changes
`timeline`	Time-series topology evolution analysis
`query`	Topology query operations

B.3 JSON Configuration File

The `--config` flag accepts a JSON file with the following structure:

Listing 23. NetVisConfig JSON

```
{
  "schema_version": "1.0",
  "layout": "force_directed",
  "quality": "balanced",
  "width": 1200.0,
  "height": 800.0,
  "effects": {
    "max_filtered_elements": 50
  }
}
```

Key	Type	Default	Description
`layout`	string	`force_directed`	Layout algorithm
`quality`	string	`balanced`	Quality preset: quick, balanced, quality
`width`	float	800.0	Viewport width
`height`	float	600.0	Viewport height
`effects.max_filtered_elements`	int	50	SVG filter budget

Table 7. JSON configuration keys.

Quality presets control iteration multipliers: `quick` (0.33x), `balanced` (1.0x), `quality` (3.0x).

Index