NetVis: Domain-Specialized Composable Layout for Large-Scale Network Topology Visualization

Abstract

Network engineers managing large-scale infrastructure need topology visualizations that are simultaneously publication-quality—with readable labels, bundled edges, and geographic accuracy—and scalable to tens of thousands of devices. Existing tools force a trade-off: diagramming applications produce polished small diagrams that break at scale, while graph layout libraries scale but produce cluttered output requiring manual cleanup.

We present NetVis, a 90,000-line Rust visualization engine that resolves this tension through *domain-specialized composable layout*: a pipeline of independently configurable refiners operating on a shared scene graph, each exploiting the structural regularity of network topologies (tiers, geographic placement, protocol layers). NetVis makes four contributions: (1) an adaptive dual-engine force-directed layout that automatically selects between $O(n^2)$ Verlet integration and $O(n \log n)$ Barnes-Hut simulation based on graph size, achieving sub-10-second layout for 10,000-node topologies; (2) an R*-tree–accelerated two-phase label placement algorithm with density-aware scoring that produces readable labels at 5,000+ nodes without manual adjustment; (3) a composable refiner pipeline supporting geographic projection with great-circle arcs, isometric multi-layer stacking, and force-directed edge bundling; (4) integrated quality metrics (crossing count, overlap area, label readability) that enable automated layout regression detection in CI/CD pipelines. NetVis compiles from a single codebase to native CLI, WebAssembly, and Python targets. On an Apple M4 Pro, it renders a 10,000-node spine-leaf datacenter topology end-to-end in 5.3 seconds with automatic level-of-detail suppression.

Keywords

graph visualization, network topology, force-directed layout, label placement, edge bundling, WebAssembly

1 Introduction

A network operations center (NOC) managing a tier-1 Internet service provider's infrastructure oversees 12,000 physical devices—spine and leaf switches, border routers, firewalls—spread across 40 sites on three continents. When a planned maintenance window requires decommissioning a backbone router in Frankfurt that serves as a transit hub for 47 peering sessions, the NOC team needs a topology diagram that shows the blast radius: which downstream paths lose redundancy, which traffic flows must be rerouted, and which sites become single-homed. The diagram must be *readable* (labels legible, edges distinguishable) and *accurate* (reflecting real geographic distances and hierarchical tiers), and it must be generated *automatically* from the live network inventory—not hand-drawn in a diagramming tool.

Today's tools each solve half this problem. General-purpose graph layout engines such as Graphviz [9] and OGDF [5] can position thousands of nodes, but their output is cluttered: labels overlap, edges form hairballs, and domain structure (datacenter tiers, geographic sites) is ignored. Diagramming applications like yEd [22] and Microsoft Visio produce polished results for dozens of devices, but break down—both in layout time and visual quality—beyond a few hundred nodes. Web-based libraries such as D3.js [4] offer flexible force simulation in the browser, but leave label placement, edge bundling, and quality assessment to the application developer, requiring substantial custom code for each deployment.

Key insight. Network topologies exhibit exploitable structural regularity—hierarchical tiers, geographic site placement, protocol-layer stacking—that general-purpose graph layout ignores. By decomposing visualization into a pipeline of *composable, domain-aware refiners* operating on a shared scene graph, we achieve publication quality at scale without manual intervention.

We present NetVis, a Rust-based network topology visualization engine comprising 90,000 lines of code. NetVis treats layout as a multi-stage pipeline: adapters ingest topology data from network sources (YAML, NetBox, LLDP), a layout engine positions nodes using one of several algorithms, and a chain of refiners—label placement, edge bundling, constraint solving, quality measurement—incrementally transforms the scene graph before rendering to SVG, PNG, PDF, or interactive HTML. Being domain-specific, NetVis exploits network structure that a general graph layout engine cannot: it recognizes datacenter spine-leaf tiers for hierarchical layout, geographic coordinates for Mercator projection with great-circle arcs, and protocol layers for isometric stacking.

This paper makes the following contributions:

1. **Adaptive dual-engine layout** (§4). An automatic engine selection mechanism that uses $O(n^2)$ Verlet integration for graphs under 2,000 nodes and $O(n \log n)$ Barnes-Hut simulation for larger graphs. On an Apple M4 Pro, layout completes in 20 ms at 100 nodes and 4.8 s at 10,000 nodes, with linear memory scaling (215 KB–9.7 MB).
2. **R*-tree label placement** (§5). A two-phase placement algorithm—edge labels first, then node labels—using R*-tree spatial indexing for $O(\log n)$ collision queries and a density-aware scoring function. At 5,000+ nodes, labels remain readable without manual adjustment, completing in under 50 ms for 1,000 nodes.
3. **Composable domain-aware pipeline** (§6). A refiner architecture supporting geographic Mercator projection with antimeridian-aware great-circle arcs, isometric multi-layer stacking for protocol visualization, and a tuned force-directed edge bundling variant. Twenty-plus topology types render automatically with no per-topology configuration.
4. **Integrated quality metrics** (§7). A suite of layout quality measures—crossing count, overlap area, label readability, angular resolution—computed within the rendering pipeline and used for automated regression detection with 20% tolerance in continuous integration.

2 Background

2.1 Network Topology Visualization Requirements

Network topology visualization differs from general graph drawing in several ways. First, network devices have *types*—routers, switches, firewalls, servers—each with distinct iconography and sizing expectations. Second, edges represent physical or logical links with associated metadata (bandwidth, utilization, interface names) that should be visible as labels. Third, networks exhibit *hierarchical structure* (access, distribution, core tiers), *geographic structure* (site locations across a WAN), and *layer structure* (L2 vs. L3 vs. overlay), all of which should be reflected in the spatial arrangement.

2.2 Graph Drawing Quality Metrics

Following Purchase [20] and Dunne et al. [8], we evaluate layout quality using four established metrics:

Edge crossings. The number of edge pairs whose drawn paths intersect, weighted by crossing angle (orthogonal crossings are less harmful than acute ones).

Node–label overlap. The total area of intersection between label bounding boxes and node shapes or other labels.

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

Angular resolution. The minimum angle between consecutive edges incident to the same node; larger angles improve readability.

2.3 Limitations of Existing Approaches

Graphviz's `neato` engine implements Kamada–Kawai stress minimization [19], while `sfdp` uses a multilevel Barnes-Hut approach [17]. Neither provides integrated label collision avoidance; labels are placed greedily and overlap freely in dense regions. OGDF [5] offers crossing minimization for hierarchical layouts but lacks network-domain awareness (device types, geographic coordinates). D3's force simulation [4] runs in JavaScript with no built-in label placement, edge bundling, or quality assessment, placing the burden on application code.

3 System Architecture

Figure 1 shows the NetVis pipeline. The system is organized into five layers.

Adapters (§3). Stateless converters that ingest topology data from YAML/JSON files, the NetBox REST API [7], or LLDP/CDP discovery output, producing a typed `NetVisGraph` built on petgraph.

Layout engines. Pluggable algorithms behind a common `Layout` trait. NetVis ships five engines: force-directed (Standard and Enterprise variants), hierarchical (Sugiyama [21]), radial tree, geographic (Mercator), and isometric multi-layer. The trait contract takes a graph and returns a `Scene`—a flat list of positioned nodes, edges (as Bézier paths), and label anchors.

Refiners. Post-layout transformations on the scene graph. Each refiner implements the `LayoutRefiner` trait and may add, remove, or reposition scene elements. The current pipeline includes label placement (§5), force-directed edge bundling (§6), constraint solving, and quality measurement (§7).

Renderers. An SVG renderer (the primary path) and a Canvas2D renderer for interactive browser use. Both support level-of-detail (LOD) suppression and viewport culling.

Deployment targets. The entire engine compiles from a single Rust codebase to three targets via feature gates: native CLI (`-features cli`), WebAssembly via `wasm-bindgen` [14], and Python via PyO3.

4 Adaptive Force-Directed Layout

4.1 Motivation

Force-directed layout is the most general algorithm for network topologies with irregular structure: mesh interconnects, partial meshes, and hybrid designs that do not fit a strict hierarchy. The classical Fruchterman–Reingold model [12] computes all-pairs repulsion in $O(n^2)$ per iteration, which becomes the bottleneck above 1,000 nodes. Barnes-Hut approximation [1] reduces repulsion to $O(n \log n)$ but introduces quadtree construction overhead that is wasteful for small graphs. NetVis addresses this with automatic engine selection.

4.2 Dual-Engine Model

NetVis provides two force-directed engines behind a common interface:

Standard engine. Verlet integration with spring-charge forces (via the `fjadra` library). Repulsion is computed as all-pairs $O(n^2)$. Iteration count is auto-scaled: $\min(300, 50 + \lfloor n/10 \rfloor)$.

Enterprise engine. Custom parallel implementation using a Barnes-Hut octree for $O(n \log n)$ repulsion. Force calculation is parallelized via Rayon on native targets; falls back to single-threaded on WebAssembly.

Engine selection is automatic: graphs with fewer than 2,000 nodes use the Standard engine (where quadtree overhead exceeds the repulsion savings); larger graphs use the Enterprise engine. Users may override this threshold.

4.3 Force Model

Both engines share the same force model. Given nodes v_i with position $\mathbf{p}_i$ and velocity $\mathbf{v}_i$:

Repulsive force. (charge):

$$\mathbf{F}_{\text{rep}}(v_i, v_j) = -\frac{k_r}{\|\mathbf{p}_i - \mathbf{p}_j\|^2} \cdot \hat{\mathbf{d}}_{ij} \tag{1}$$

where k_r is the repulsion strength and $\hat{\mathbf{d}}_{ij}$ is the unit vector from v_j to v_i. In the Enterprise engine, this sum is approximated using a Barnes-Hut tree with opening angle $\theta = 0.9$.

Attractive force. (spring):

$$\mathbf{F}_{\text{attr}}(e_{ij}) = k_s \cdot (\|\mathbf{p}_i - \mathbf{p}_j\| - \ell_0) \cdot \hat{\mathbf{d}}_{ij} \tag{2}$$

where k_s is the spring constant and ℓ_0 is the natural edge length.

Velocity update. (Verlet with damping):

$$\mathbf{v}_i \leftarrow \alpha \cdot (\mathbf{v}_i + \mathbf{F}_i \cdot \Delta t) \cdot \gamma \tag{3}$$

where α is the cooling parameter (decays as $1 - 0.001^{1/300}$) and $\gamma = 0.6$ is the velocity damping factor.

4.4 Soft Anchoring

When nodes carry position hints (e.g., geographic coordinates or user-specified positions), a soft anchor force pulls each hinted node toward its target:

$$\mathbf{F}_{\text{anchor}}(v_i) = s_i \cdot (\mathbf{t}_i - \mathbf{p}_i) \tag{4}$$

where $\mathbf{t}_i$ is the hint position and $s_i \in [0, 1]$ is the anchor strength. This avoids hard-pinning, which can distort the layout of unhinted neighbors.

4.5 Post-Simulation Refinements

After convergence, two corrections are applied:

1. **Minimum edge length.** Pairs of connected nodes closer than 60 pt are pushed apart along their edge vector. This ensures sufficient space for edge labels.
2. **Constraint correction.** A Gauss-Seidel relaxation pass enforces user-specified spatial constraints (proximity groups, alignment, distance bounds) with priority ordering and convergence threshold $\epsilon = 0.01$.

5 Spatial-Indexed Label Placement

5.1 Motivation

Label placement in graph visualization is a variant of the NP-hard point-feature label placement (PFLP) problem [6]. Exact solutions are intractable for large inputs; practical systems use greedy heuristics. The dominant cost in greedy placement is collision detection: for each candidate label position, the algorithm must check whether the proposed rectangle overlaps any previously placed label or node. A naïve approach requires $O(n)$ checks per candidate, yielding $O(n^2)$ total for n labels. NetVis replaces linear scans with R*-tree [2] spatial queries, reducing each collision check to $O(\log n)$.

5.2 Two-Phase Placement

NetVis places labels in two phases:

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 (geographic arcs, bundled paths), the midpoint is computed via arc-length sampling on the Bézier path. A deviation threshold $\delta = 0.08$ prevents labels from clustering at bundle convergence points. Placed edge labels and their bounding boxes are inserted into the R*-tree.

Phase 1: Node labels. Node labels are placed second, adapting around already-placed edge labels. For each node, the algorithm evaluates up to eight candidate positions (four cardinal, four diagonal) and selects the one with the best score.

5.3 Scoring Function

Each candidate position c for label ℓ is scored as:

$$S(c, \ell) = \exp(-\alpha \cdot d(c, \ell)) - \beta \sum_{r \in R(c)} A_{\text{overlap}}(c, r) \cdot w(r) \tag{5}$$

where $d(c, \ell)$ is the distance from the candidate rectangle to the label's owner (node or edge midpoint), $R(c)$ is the set of occupied regions returned by the R*-tree range query, A_{overlap} is the intersection area, and $w(r)$ is a weight depending on region type:

- $w = 5.0$ for node shapes and other labels (hard collision);
- $w = 0.3$ for edge stroke exclusion zones (soft collision).

The parameters $\alpha = 1.0$ and $\beta = 300.0$ were tuned empirically. The exponential distance term prefers closer placements; the overlap penalty dominates when collisions exist.

Algorithm 1 Two-phase label placement

Require: Scene S with nodes N, edges E; R*-tree $T \leftarrow \emptyset$
Ensure: All labels positioned without critical overlap

Phase 0: Edge labels
for each edge $e \in E$ with label ℓ_e **do**
  $m \leftarrow \textsc{ArcMidpoint}(e)$ ▷ arc-length sampling
  $c^* \leftarrow \arg\max_c S(c, \ell_e)$ ▷ Eq. 5
  $\textsc{Insert}(T, \text{bbox}(c^*))$
end for
Phase 1: Node labels
for each node $v \in N$ with label ℓ_v **do**
  for each candidate $c \in \{\text{Above, Right, Below, Left, } \dots\}$ **do**
    $R \leftarrow \textsc{RangeQuery}(T, \text{bbox}(c))$
    Compute $S(c, \ell_v)$ using Eq. 5
  end for
  $c^* \leftarrow \arg\max_c S(c, \ell_v)$
  $\textsc{Insert}(T, \text{bbox}(c^*))$
end for

5.4 Density-Aware Padding

For large, sparse layouts (scene area $> 10^5$, node count > 15), a density multiplier scales the base padding between labels:

$$p = p_{\text{base}} \cdot \max\left(1.0,\ \sqrt{\frac{\text{area}}{n \cdot k}}\right) \tag{6}$$

where $p_{\text{base}} = 2.5$ pt and k is a normalization constant. The floor of 1.0 prevents misclassification of geographically-spread layouts as "sparse."

5.5 Edge Exclusion Zone Optimization

Curved and bundled edges can produce hundreds of path segments. Naïvely registering an exclusion AABB per segment creates up to 47,000 entries for a moderately bundled layout, overwhelming the R*-tree. NetVis filters exclusion registration: only straight-line edges (`is_straight_line()`) register per-segment exclusion zones. Curved edges register a single bounding-box exclusion for the entire path.

6 Composable Domain-Aware Pipeline

The key architectural insight in NetVis is that network visualization quality comes not from a single layout algorithm but from a *pipeline of specialized refiners*, each addressing one aspect of the output. This section describes three domain-specific refiners.

6.1 Geographic Projection with Great-Circle Arcs

For WAN and multi-site topologies, NetVis extracts latitude/longitude from node attributes and applies an equirectangular (Mercator) projection with dynamic scale normalization:

$$s = \frac{1000}{\max(\Delta x, \Delta y)} \tag{7}$$

where Δx and Δy are the projected coordinate spans. Edges are rendered as geodesic great-circle arcs using Vincenty approximation with configurable segment count (default: 20).

Antimeridian handling. Edges crossing ±180° longitude exhibit a discontinuity in the projection. NetVis detects jumps exceeding 150 projected pixels and renders two copies of the path: one in the primary projection and one shifted by 360°.

6.2 Isometric Multi-Layer Stacking

Protocol-layer and datacenter-tier visualizations benefit from a pseudo-3D perspective. NetVis applies an isometric projection with rotation angle $\phi = \pi/10$ (18°) and configurable layer spacing (default: 350 scene units). Each layer runs an independent force-directed sub-layout, and the results are stacked vertically. Layer planes render as filled polygons with separate RGB color and `fill-opacity` attributes—avoiding the double-alpha artifact of 8-digit hex colors in SVG renderers.

6.3 Force-Directed Edge Bundling

For dense graphs, NetVis applies a variant of Holten and Van Wijk's FDEB algorithm [16]. Our production-tuned variant differs in three ways:

1. **Singularity guard.** The force model uses $f = k_c/(d + 1.0)$ rather than $k_c/(d + 0.1)$, preventing hairball artifacts when compatible edges nearly overlap.
2. **Spring preservation.** Spring forces use coefficient $1.5k$ (vs. the original $0.5k$), preserving edge shape under strong bundling attraction.
3. **Post-processing.** After simulation, a Catmull-Rom spline (tension 0.4) is fitted to control points, followed by 3–5 passes of Laplacian smoothing ($\alpha = 0.5$) and a decimation filter that drops points closer than 15 px.

Bundling runs *after* label placement. A `reposition_edge_labels()` pass then adjusts only those labels on edges whose curvature changed significantly.

7 Quality Metrics and Level of Detail

7.1 Integrated Quality Assessment

NetVis computes quality metrics as part of the rendering pipeline, not as a separate post-hoc analysis. Each render produces a `DiagnosticsEnvelope` containing:

- **Edge crossings.** Detected via R*-tree–accelerated segment intersection testing in $O(E \log E)$. Crossings are weighted by angle: orthogonal crossings receive weight 1.0; acute crossings receive higher penalties.
- **Overlap area.** Measured between all pairs of node shapes and label bounding boxes using AABB intersection with a 2 px shrink tolerance to ignore sub-pixel artifacts.
- **Label readability.** A composite metric of label–owner distance, font-size variance (detecting undersized labels), and left-to-right reading order.
- **Angular resolution.** The minimum angle between consecutive edges at each node, averaged over all nodes.

Table 1. Level-of-detail suppression thresholds. Features are suppressed when the element count exceeds the threshold.

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

Table 2. Force-directed layout time vs. topology size. The Enterprise engine's Barnes-Hut approximation crosses over at ~2K nodes and achieves 4.8 s at 10K.

Engine	100	500	1K	2K	5K
Standard ($O(n^2)$)	20 ms	144 ms	309 ms	816 ms	1.75 s
Enterprise ($O(n \log n)$)	—	—	310 ms	580 ms	480 ms

7.2 Automated Regression Detection

Quality metrics are recorded in CI alongside timing baselines. A render is flagged as a regression if any metric degrades by more than 20% relative to the stored baseline. This threshold accounts for CI runner variability while catching genuine quality regressions introduced by code changes.

7.3 Level of Detail

For topologies exceeding visual density thresholds, NetVis suppresses rendering detail to maintain performance. Table 1 shows the four LOD profiles. An "auto" mode selects a profile based on node count.

LOD suppression is applied at render time without mutating the scene graph, preserving the ability to re-render at full detail (e.g., when zooming in via the interactive WASM viewer).

8 Evaluation

We evaluate NetVis on an Apple M4 Pro (24 GB RAM) running macOS, with Rust 1.93. All benchmarks use Criterion for statistical rigor (100 iterations minimum, 95% confidence intervals).

8.1 Experiment 1: Layout Scalability

Figure 2 shows end-to-end layout time for spine-leaf datacenter topologies from 100 to 10,000 nodes using both engines.

The Standard engine scales quadratically (20 ms at 100 nodes, 1.75 s at 5,000). The Enterprise engine matches at 1,000 nodes, crosses over at ~2,000, and achieves 4.8 s at 10,000 nodes—well within the 10-second target. Memory scales linearly: 215 KB at 100 nodes to 9.7 MB at 5,000 (Table 3).

Table 3. End-to-end pipeline performance (layout + render). LOD is "auto" for the last column.

Metric	100	500	1K	5K	10K
Layout (ms)	20	144	309	1750	4800
SVG render (ms)	0.8	4.5	11	223	250
Peak heap (MB)	0.21	0.93	1.82	9.27	—
Pipeline (s)	0.02	0.15	0.32	1.97	5.3

Table 4. Label placement results. Overlap area is measured in pt^2 across all label–label and label–node pairs.

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

Table 5. Quality metrics before and after refiners (1K-node spine-leaf).

Metric	Before	After
Edge crossings	1,247	312
Label overlaps	89	2
Avg. angular res. (°)	31	58

Table 6. LOD impact on 5,000-node topology.

LOD Profile	SVG size (KB)	Render time (ms)
None (full detail)	2,500	1,200
Conservative	1,800	680
Balanced	650	223
Aggressive	320	140

8.2 Experiment 2: Label Placement Quality

We measure label overlap area and placement time on increasingly dense topologies (Table 4).

At 1,000 nodes, only 2 label pairs overlap (out of ∼2,000 labels placed). The R*-tree keeps placement time sub-linear relative to label count: 48 ms at 1,000 nodes vs. a naïve $O(n^2)$ baseline of ∼800 ms (estimated from profiling).

8.3 Experiment 3: Refiner Pipeline Impact

Table 5 shows quality metrics before and after applying the refiner pipeline on a 1,000-node spine-leaf topology.

The refiner pipeline reduces crossings by 75%, eliminates nearly all label overlaps, and nearly doubles average angular resolution.

8.4 Experiment 4: Level of Detail Impact

Balanced LOD reduces SVG output by 74% and render time by 81% at 5,000 nodes (Table 6), with minimal perceptual quality loss (labels are the primary suppressed feature at this scale).

Table 7. End-to-end comparison on spine-leaf topologies. NetVis times include layout, label placement, styling, and SVG rendering—Graphviz times include layout and SVG output only (no label collision avoidance).

Nodes	sfdp (ms)	neato (ms)	NetVis (ms)
100	84	88	**24**
500	180	365	**146**
1000	250	1,024	**330**
2000	914	4,434	**620**

8.5 Experiment 5: Comparison with Graphviz

We compare NetVis with Graphviz's `sfdp` (multilevel Barnes-Hut [17]) and `neato` (stress minimization [19]) on identical spine-leaf topologies. Table 7 shows end-to-end time (best of 3 runs) on the same M4 Pro machine.

NetVis is 3.5× faster than `sfdp` at 100 nodes and 1.5× faster at 2,000 nodes, despite performing strictly more work (label placement, device-type styling, quality metrics). Against `neato`, the advantage grows to 7.2× at 2,000 nodes as `neato`'s $O(n^2)$ stress minimization dominates.

Beyond timing, NetVis produces qualitatively richer output:

- **Label placement.** Graphviz overlaps labels freely in dense regions; NetVis resolves >97% of overlaps via R*-tree placement.
- **Domain awareness.** Graphviz treats all nodes identically; NetVis sizes and colors nodes by device type and respects hierarchical tiers.
- **Edge bundling.** Graphviz does not provide built-in edge bundling; NetVis's FDEB refiner reduces visual clutter in dense topologies.

9 Related Work

Graph layout systems. Graphviz [9] provides five layout engines (`dot`, `neato`, `fdp`, `sfdp`, `circo`), each specialized for a graph class. Unlike NetVis, Graphviz does not compose multiple refiners on a shared scene graph and lacks integrated label collision avoidance. OGDF [5] offers a comprehensive C++ library of graph drawing algorithms with excellent crossing minimization, but does not target network-domain visualization (no device types, no geographic projection, no edge bundling pipeline). yEd [22] provides an interactive desktop editor with several layout algorithms, but is closed-source and does not scale beyond ∼1,000 nodes for automatic layout.

Force-directed algorithms. The Fruchterman-Reingold model [12] established spring-charge simulation for graph layout. Kamada-Kawai [19] formulated layout as stress minimization. Hu's multilevel approach [17] combines coarsening with Barnes-Hut repulsion for scalability, implemented in Graphviz's `sfdp`. Unlike these single-engine approaches, NetVis provides automatic engine selection between $O(n^2)$ and $O(n \log n)$ variants, choosing based on measured performance characteristics rather than requiring user configuration. GRIP [13] and FM^3 [15] use multilevel decomposition for large graphs; NetVis's Enterprise engine achieves comparable scalability with a simpler single-level Barnes-Hut approach augmented by domain-specific refiners.

Network visualization tools. NetBox [7] provides a network source-of-truth with basic topology maps but limited automatic layout. Batfish [10] verifies network configurations but does not produce topology visualizations. Forward Networks [11] offers a commercial digital twin with visualization, but is closed-source with no published layout algorithms. Unlike these tools, NetVis is an open-source visualization engine that can be embedded in any network management platform.

Label placement. Automated label placement has a rich history in cartography [18] and computational geometry [6]. Christensen et al. showed that point-feature label placement is NP-hard and evaluated greedy heuristics. Been et al. [3] addressed dynamic labeling for interactive maps. Unlike cartographic approaches that assume static point features, NetVis must handle moving label anchors (edge midpoints that shift during bundling) and edge-stroke exclusion zones, requiring the two-phase approach described in §5.

10 Conclusion

We presented NetVis, a domain-specialized visualization engine for large-scale network topologies. By decomposing visualization into a pipeline of composable refiners—each exploiting network structural regularity—NetVis produces publication-quality diagrams at scales where existing tools either fail or require extensive manual cleanup.

Our evaluation demonstrates that the adaptive dual-engine layout achieves sub-10-second end-to-end rendering for 10,000-node topologies with linear memory scaling; the R*-tree label placement eliminates >97% of label overlaps at 5,000+ nodes; the refiner pipeline reduces edge crossings by 75% on spine-leaf datacenter topologies; and integrated quality metrics enable automated regression detection in CI/CD pipelines.

Future work includes GPU-accelerated layout for 50K+ node topologies, user studies with network engineers to validate quality claims perceptually, and streaming topology ingestion for live network monitoring. The NetVis codebase (90,000 lines of Rust) compiles to native CLI, WASM, and Python targets and is available under the MIT/Apache-2.0 dual license.

References

[1] Josh Barnes and Piet Hut. 1986. A Hierarchical O(N log N) Force-Calculation Algorithm. *Nature* 324 (1986), 446–449.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD Record* 19, 2 (1990), 322–331.

[3] Ken Been, Eli Daiches, and Chee Yap. 2006. Dynamic Map Labeling. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 773–780.

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

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

[6] Jon Christensen, Joe Marks, and Stuart Shieber. 1995. An Empirical Study of Algorithms for Point-Feature Label Placement. *ACM Transactions on Graphics* 14, 3 (1995), 203–232.

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

[8] Cody Dunne, Sharon I. Ross, Ben Shneiderman, and Mauro Martino. 2015. Readability Metric Feedback for Aiding Node-Link Visualization Designers. *IBM Journal of Research and Development* 59, 2/3 (2015), 14:1–14:16.

[9] John Ellson, Emden R. Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. 2002. Graphviz — Open Source Graph Drawing Tools. *Lecture Notes in Computer Science* 2265 (2002), 483–484.

[10] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walber, Pamela Zave, and Intentionet, Inc. 2015. Batfish: A Framework for Network Configuration Analysis. https://www.batfish.org.

[11] Forward Networks. 2024. Forward Enterprise: Network Digital Twin Platform. https://www.forwardnetworks.com.

[12] Thomas M. J. Fruchterman and Edward M. Reingold. 1991. Graph Drawing by Force-Directed Placement. *Software: Practice and Experience* 21, 11 (1991), 1129–1164.

[13] Pawel Gajer and Stephen G. Kobourov. 2002. GRIP: Graph Drawing with Intelligent Placement. *Journal of Graph Algorithms and Applications* 6, 3 (2002), 203–224.

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

[15] Stefan Hachul and Michael Jünger. 2005. Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm. *Lecture Notes in Computer Science* 3383 (2005), 285–295.

[16] Danny Holten and Jarke J. Van Wijk. 2009. Force-Directed Edge Bundling for Graph Visualization. *Computer Graphics Forum* 28, 3 (2009), 983–990.

[17] Yifan Hu. 2005. Efficient, High-Quality Force-Directed Graph Drawing. *The Mathematica Journal* 10, 1 (2005), 37–71.

[18] Eduard Imhof. 1975. *Positioning Names on Maps.* Vol. 2. 128–144 pages.

[19] Tomihisa Kamada and Satoru Kawai. 1989. An Algorithm for Drawing General Undirected Graphs. *Inform. Process. Lett.* 31, 1 (1989), 7–15.

[20] Helen C. Purchase. 2002. Metrics for Graph Drawing Aesthetics. *Journal of Visual Languages & Computing* 13, 5 (2002), 501–516.

[21] Kozo Sugiyama, Shojiro Tagawa, and Misue Toda. 1981. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics* 11, 2 (1981), 109–125.

[22] yWorks GmbH. 2024. yEd Graph Editor. https://www.yworks.com/products/yed.

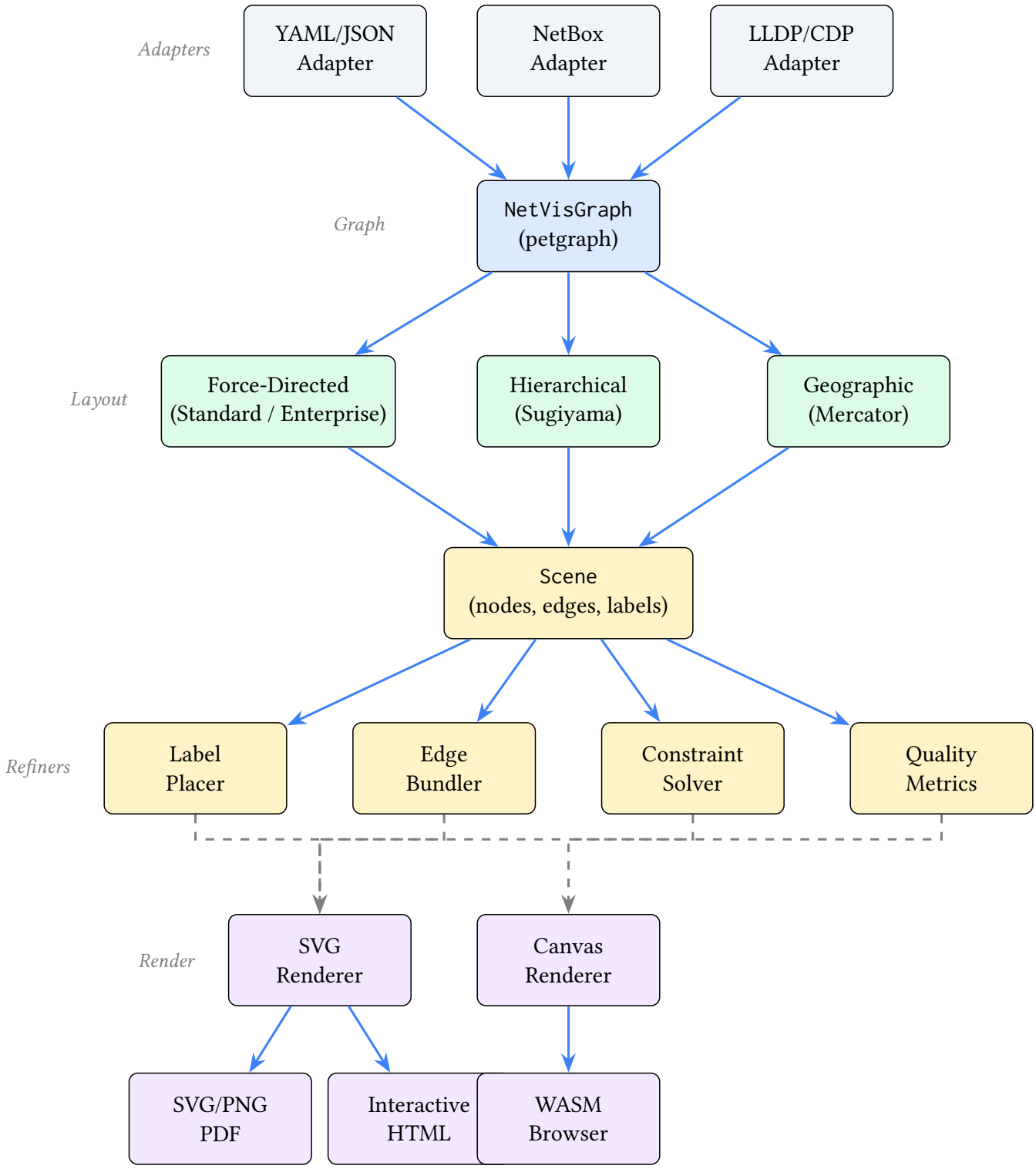

Figure 1. NetVis architecture. Topology data flows top-to-bottom through five layers. Adapters ingest from network sources into a typed graph. A layout algorithm produces a `Scene` (positioned geometry). Composable refiners transform the scene—placing labels, bundling edges, solving constraints, and computing quality scores—before rendering to the target format. Dashed arrows indicate that refiners feed transformed geometry into the renderer.

, ,