NetCfg: A Declarative Network Configuration Compiler with Graph-Backed Primitive Composition

Simon Knight

Abstract

Network operators managing thousands of devices face a fundamental tension: topology intent—"mesh these routers, assign IPs from this pool, build a BGP overlay"—must be manually translated into vendor-specific device configurations, a process that is error-prone, non-deterministic, and fragile at scale. Existing tools address fragments of this problem: IPAM systems (NetBox) manage address allocation but do not generate configs; automation frameworks (Ansible, Nornir) push configs but lack topology awareness; and network modelling tools (Batfish) verify configs but cannot produce them.

We present **NetCfg**, a declarative network configuration compiler that bridges this intent–configuration gap. NetCfg takes two inputs: a *blueprint* declaring topology transformations as composable primitives, and a *mapping* declaring how to extract vendor-specific configuration stanzas from the resulting topology. The system compiles these through five stages: (1) parse and validate both documents, (2) clone the topology into a transactional shadow, (3) execute primitives against the shadow, (4) evaluate mapping rules against the transformed topology to produce a *DeviceIR* intermediate representation, (5) render DeviceIR through vendor-specific Jinja2 templates, and (6) write per-device configuration files via atomic transactional output.

NetCfg compiles a 10,000-node topology with full-mesh generation, IP allocation, and protocol-layer construction in 12.5 s. A 1,000-node end-to-end pipeline completes in 372 ms.

1 Introduction

A tier-1 ISP operating 10,000 routers across 200 sites must periodically re-address its backbone: perhaps a merger introduces overlapping address space, or a security policy mandates /31 point-to-point links replacing legacy /30 allocations. The operator's intent is straightforward—"for every backbone link, allocate a /31 from pool 10.0.0.0/16; build a BGP overlay with ASN 65000 as the base"—but translating this intent into 10,000 vendor-specific configuration files is an exercise in combinatorial fragility. A single misallocated subnet or a duplicated BGP session can trigger a routing loop affecting millions of users.

Today's tooling forces operators to choose between *topology awareness* and *configuration generation*. Source-of-truth systems such as NetBox [1] and Nautobot [2] maintain device inventories and IP allocations, but produce no runnable configuration. Automation frameworks such as Ansible [3] and Nornir [4] render and push templates, but have no model of the network graph—they treat each device as an independent unit, blind to adjacency relationships. Network modelling tools such as Batfish [6] can verify configurations, but operate read-only: they consume configs, they do not produce them. The operator is left to bridge these tools manually, with spreadsheets and ad-hoc scripts serving as the de facto integration layer between intent and config.

Key insight. *Network configuration is a compilation problem, not a templating problem.* The operator's intent describes transformations over a typed, layered graph—meshing nodes, allocating addresses to edges, promoting physical nodes to protocol overlays. If we model the network as a graph with columnar property storage, execute intent declarations as composable primitives against a transactional shadow of that graph, and then extract device configurations via declarative mapping rules, we can produce deterministic, diffable, vendor-neutral intermediate representations that compile down to device-specific configurations.

We present **NetCfg**, a Rust-based network configuration compiler that embodies this insight. The operator provides two documents:

- A **blueprint** declaring what topology transformations to apply ("mesh these nodes, provision IPs to those edges, build a BGP overlay")—the *what* of the network design.
- A **mapping** declaring how to extract configuration stanzas from the resulting topology ("for every BGP node, emit a stanza with `local_asn` and `peer_ip`")—the *how* of config generation.

The compiler processes these through a five-stage pipeline (§3): parse → shadow → execute primitives → evaluate mappings → render templates → write files. Each stage produces a well-defined intermediate representation: the blueprint produces a typed AST; transformation produces a desired topology with accumulated `DeviceContext` state per node; mapping evaluation produces `DeviceIR` stanzas; and rendering produces vendor-specific configuration text.

Contributions.

1. **Two-document compilation model (§3).** Separation of topology intent (blueprint) from config extraction (mapping), connected by a graph-backed intermediate topology where each node carries a typed `DeviceContext` accumulating state across primitive execution.
2. **Shadow topology execution (§4).** Transactional isolation via serialisation-based cloning, with sequential primitive composition, design-rule assertions, and deterministic output guarantees.
3. **Composable primitives with identity-based IPAM (§5).** Twelve domain-specific primitives including a five-pass IP allocator with prefix-trie-backed VRF isolation, hostname-keyed allocation stickiness, routing and access policy generation, and WebAssembly plugin extensibility.
4. **DeviceIR and provenance-tracked rendering (§6).** A stable intermediate representation where every stanza carries lineage back to its mapping rule, enabling per-line config auditability.
5. **Eleven operational modes (§8).** `generate`, `validate`, `plan`, `inspect`, `ci-run`, `import`, `lint`, `fmt`, `impact`, `report`, and

`export-clab` supporting the full lifecycle from development through CI/CD, linting, and emulation.

6. **Evaluation at scale (§10).** Benchmarks demonstrating end-to-end compilation of 10,000-node topologies in 12.5 s, with per-component breakdowns.

2 Background: Network as a Typed, Layered Graph

We model the network as a directed graph $G = (V, E, \tau, \lambda, D)$ where V is a set of nodes (devices, protocol instances), $E \subseteq V \times V$ is a set of edges (links, peering sessions), $\tau : V \to T$ assigns each node a *type* (Router, Switch), $\lambda : V \to L$ assigns each node a *layer*, and $D : T \to \textit{DataFrame}$ maps each type to a columnar property store.

Layers. are the key structural concept. A layer represents a plane of the network: `physical` for the hardware topology, `bgp` for BGP peering sessions, `ospf` for OSPF adjacencies. A single physical router (node 1, layer `physical`) may have corresponding protocol nodes (node 101, layer `bgp`; node 201, layer `ospf`), each linked to node 1 via a *parent mapping*. Layers allow the compiler to model the physical and logical topologies simultaneously, with cross-layer references preserving traceability.

Per-node state. Each node's properties are stored as a JSON blob (`data_json`) in a Polars DataFrame keyed by node type. NetCfg deserialises this blob into a typed `DeviceContext` struct:

Listing 1. DeviceContext: the per-node state model that accumulates through primitive execution.

```
pub struct DeviceContext {
    pub hostname: String,
    pub device_os: Option<String>,     // template selector
    pub src_ip: Option<String>,        // written by provision_ips
    pub dst_ip: Option<String>,        // written by provision_ips
    pub subnet: Option<String>,        // written by provision_ips
    pub src_ipv6: Option<String>,      // dual-stack support
    pub dst_ipv6: Option<String>,
    pub peer_interfaces: Option<HashMap<String, String>>,
    pub stanzas: Vec<Stanza>,          // accumulated config units
    pub metadata: HashMap<String, JsonValue>, // extension point
}
```

Primitives progressively enrich this context: `mesh_nodes` establishes edges, `provision_ips` writes `src_ip`/`dst_ip`/`subnet`, `build_protocol_layer` deep-merges config overlays into the cloned node's metadata. By the time rendering occurs, each node's `DeviceContext` contains all the information needed to produce its configuration file.

This model is implemented by the NTE topology engine [9], which provides a directed graph (built on the `petgraph` Rust crate) with Polars DataFrames.

3 Compilation Pipeline

NetCfg's compilation pipeline has five stages, each producing a distinct intermediate representation. Figure 1 shows the data flow.

3.1 Stage 1: Parse and Validate

The blueprint parser deserialises YAML into a typed `Blueprint` AST containing layer specifications, each with an ordered list of primitive specifications. Three validation passes run at parse time:

- **Schema validation** via `serde_valid`: field types, non-empty names, integer ranges. Unknown YAML keys are rejected via `deny_unknown_fields`, catching typos at parse time.
- **Selector compilation**: each selector string is compiled into an evaluable Common Expression Language (CEL) program, surfacing syntax errors before execution.
- **Layer ordering validation**: a forward scan ensures every `requires` entry names a previously-declared layer, preventing silent execution-order bugs.

Parse errors use `miette` diagnostics with source spans pointing at the offending line in the YAML.

3.2 Stage 2: Shadow Topology

The executor clones the base topology via a serialisation round-trip (byte buffer → independent copy), creating an isolated *shadow topology*. This guarantees complete isolation: mutations to the shadow cannot affect the source.

3.3 Stage 3: Execute Primitives

The executor iterates layers and primitives sequentially, applying each primitive to the shadow. Mutations from primitive n are visible to primitive $n{+}1$ within the same layer—this sequential visibility is the basis of primitive composition (§5).

After all primitives execute: (1) structural invariants are validated, (2) user-defined design-rule assertions are checked, and (3) the shadow is committed as the *desired topology*. This desired topology contains fully populated `DeviceContext` for every node—hostnames, IP addresses, protocol metadata, interface assignments—accumulated across all primitive executions. See §4 for details.

3.4 Stage 4: Evaluate Mappings

The *mapping document* declares rules for extracting configuration stanzas from the transformed topology. Each rule has a selector (matching nodes or edges) and a stanza template specifying the `kind`, `key`, and `fields` of the emitted DeviceIR stanza. Listing 2 shows a mapping that extracts BGP configuration from every node.

Listing 2. Mapping document: extracts BGP stanzas from the transformed topology.

```
rules:
  - selector: "all_nodes"
    stanza:
      kind: "bgp"
      fields:
        node: "site-a-r1"
        local_asn: "65001"
        description: "full-mesh BGP session"
```

The mapping evaluator (provided by the NTE engine [9]) iterates rules, evaluates selectors against the desired topology, and emits one `DeviceIR` stanza per matched entity per rule. Stanzas carry *provenance*: each stanza records the `rule_id` of the mapping rule that produced it, enabling lineage tracing.

The key separation: the blueprint says *what* the network should look like (topology transformations); the mapping says *how* to extract device configuration from the result. This separation means the same blueprint can produce Cisco IOS, Arista EOS, or Juniper JunOS output by swapping only the mapping and templates.

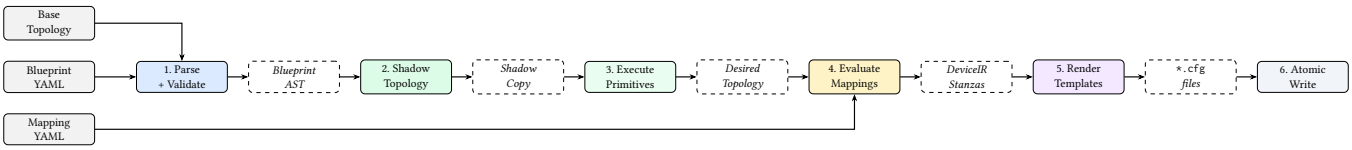

Figure 1. NetCfg compilation pipeline. Three inputs (base topology, blueprint, mapping) flow through six stages, each producing a distinct intermediate representation. The mapping document enters at stage 4, *after* primitive execution—it operates on the transformed topology, not the input.

3.5 Stage 5: Render Templates

DeviceIR stanzas are grouped by the `fields.node` value (hostname), then rendered through Jinja2 templates via the MiniJinja engine. Template selection is keyed on `device_os`: a node with `device_os: "cisco"` renders through `cisco.j2`. The template receives the full `DeviceContext` as its context, plus the stanza list:

Listing 3. Cisco template (cisco.j2): renders hostname and IP from DeviceContext fields.

```
{{ hostname }}
{% if subnet is defined %}
  ip address {{ subnet }}
{% endif %}
```

This produces output like:

Listing 4. Generated configuration for core-0.

```
core-0
  ip address 10.0.0.72/30
```

Rendering is parallelised across nodes via Rayon, exploiting the independence of per-device config generation.

3.6 Output: Atomic Write

The `TransactionalOutput` writer collects all files in temporary locations (using `NamedTempFile` in the same directory for same-filesystem guarantee), then atomically renames all via POSIX `rename(2)` on commit. If any stage fails, all temp files are dropped without committing—no partial output directory is ever visible. Three artefacts are emitted per node:

- `{hostname}.conf`: the rendered device configuration,
- `{hostname}.deviceir.json`: the raw DeviceIR stanzas (audit),
- `{hostname}.lineage.json`: per-line mapping from config lines to the `rule_id` that produced them (provenance).

4 Shadow Topology Execution

4.1 Transactional Isolation

The shadow topology provides isolation via serialisation-based cloning:

Listing 5. Shadow topology: deep clone via serialisation round-trip.

```
pub fn new(current: &Topology)
    -> Result<ShadowTopology, ShadowError>
{
    let bytes = current.save_to_bytes()?;
    let shadow = Topology::load_from_bytes(&bytes)?;
    Ok(ShadowTopology { shadow })
}
```

This guarantees complete isolation: mutations to the shadow cannot affect the source. The design choice of serialisation over `Clone` ensures a deep copy of all internal state (graph structure, DataFrames, metadata) without requiring the graph engine to expose a Clone implementation.

4.2 Primitive Composition via Sequential Visibility

Primitives execute in declaration order. A fresh `PrimitiveContext` is created for each, but all share the same underlying shadow. This enables a critical composition pattern:

1. `mesh_nodes` creates edges between selected routers.
2. `provision_ips` sees those new edges (they exist in the shadow) and allocates IP addresses to them, writing `src_ip`/`dst_ip` into each endpoint node's `DeviceContext`.
3. `build_protocol_layer` clones nodes into a BGP overlay. Because IPs were already written in step 2, the cloned BGP nodes *inherit* their parent's IP addresses via deep merge—no explicit propagation needed.

This sequential accumulation of `DeviceContext` state is the mechanism by which the pipeline "compiles" high-level intent into fully populated per-device configuration context.

4.3 Design-Rule Assertions

Blueprints may declare assertions: post-execution invariants checked before commit. Seven check types: `FieldExists`, `MinCount`, `MaxCount`, `MinEdges`, `UniquePerGroup`, `FieldInCidr`, and `CustomCel` (arbitrary CEL expressions). Each has severity `error` (pipeline failure) or `warning` (diagnostic only). This enables "policy as code" directly in the blueprint:

Listing 6. Design-rule assertions declared in a blueprint.

```
assertions:
  - name: "all routers have hostname"
    severity: error
    select: all()
    check:
      field_exists: hostname
  - name: "IPs within allocation"
    severity: error
    select: "nodes[src_ip != null]"
    check:
      field_in_cidr:
        field: src_ip
        cidr: "10.0.0.0/8"
```

4.4 Determinism Guarantee

Theorem 4.1 (Determinism). *Given a fixed blueprint B, mapping M, and input topology G, the output configuration set $C = compile(G, B, M)$ is deterministic.*

Proof sketch. Serialisation round-trip produces a bit-identical shadow. Primitives execute in declaration order. Selectors evaluate deterministically. Node ID allocation is monotonic. IP allocation iterates edges sorted by (src, dst). DeviceIR stanzas use `BTreeMap` (sorted keys) and are post-sorted by `(kind, key)`. Template rendering is pure. Therefore every intermediate state, and hence C, is uniquely determined by (G, B, M). □

5 Transformation Primitives

All primitives access the topology exclusively through `PrimitiveContext`, which provides CEL selector compilation, immutable/mutable graph access, batch property writes, and monotonic ID allocation. Every primitive follows the *Selector → Execute → Mutate* pattern and the *two-pass borrow* discipline: collect all reads into local vectors (immutable borrows), then apply all mutations (mutable borrows), satisfying Rust's ownership rules without `unsafe` code.

5.1 `mesh_nodes`: Topology Construction

Creates edges between selected nodes. For a full mesh of n nodes, produces $\binom{n}{2}$ undirected edges ($src < dst$). Also supports hub-and-spoke and route-reflector topologies. Idempotent: existing edges are detected via a `HashSet<(i32, i32)>` and skipped.

5.2 `provision_ips`: Five-Pass IPAM

Allocates subnets from a CIDR pool to edges. The five-pass algorithm:

1. **State discovery:** Scan nodes for existing IP assignments.
2. **Edge selection:** Evaluate CEL selector; sort by (src, dst).
3. **Pre-reservation:** Insert existing allocations into the prefix trie, preventing re-allocation.
4. **Fulfilment:** Allocate subnets using the configured strategy.
5. **Write-back:** Write `src_ip`, `dst_ip`, `subnet` into each endpoint's `DeviceContext`.

Identity-based stickiness. The allocator uses a canonical identity —`sorted(hostname_a, hostname_b)`—as the primary key. If a prior allocation exists for the same identity, the same prefix is returned without pool search. This anchors allocations to stable semantic identity, not integer IDs.

VRF isolation. Per-VRF state uses a `PrefixSet<Ipv4Net>` (prefix trie) for $O(\log n)$ containment. Hierarchical pool carving allows a global `/8` to be carved into named `/24` blocks. Three strategies: *dense* (sequential), *sparse* (growth headroom), *hash* (topology-independent).

5.3 `build_protocol_layer`: Overlay Construction

Clones physical nodes into a named protocol layer (`bgp`, `ospf`). For each node: (1) allocate new ID, (2) add to target layer, (3) establish parent mapping to physical node, (4) deep-merge config overlay. Optionally clones edges between source-layer nodes into the overlay, preserving adjacency for protocol-aware operations.

Cross-layer traceability: the parent mapping links each protocol node to its physical origin. Since `provision_ips` already enriched the physical node's `DeviceContext` with IP addresses, the cloned protocol node inherits them via the deep merge—this is how IP addresses flow from IPAM allocation to BGP configuration without explicit wiring.

5.4 Additional Primitives

`allocate_resources`: generic pool allocation for ASNs, VLANs, etc. `global_pool` / `global_resource_pool`: register named pools at blueprint scope for hierarchical carving across primitives. `conditional`: CEL-based branching within blueprints. `custom_primitive`: named, reusable primitive groups with parameters. `inject_secrets`: reads environment variables into node metadata. `build_routing_policy` / `build_access_policy`: attach routing policy (route-map) or access-control (ACL) stanzas to matched nodes, stored in `DeviceContext.stanzas` for downstream template rendering. `wasm_plugin`: executes a WebAssembly binary against matched nodes via the `wasmtime` runtime (optional feature flag), enabling third-party primitive logic without recompiling the core.

6 DeviceIR: The Intermediate Representation

DeviceIR is the phase boundary between topology transformation and vendor-specific rendering. It is a list of *stanzas*:

Listing 7. DeviceIR stanza: the unit of configuration extracted by mapping evaluation.

```rust
pub struct Stanza {
    pub kind: String,                    // "bgp", "system"
    pub key: String,                     // stable ordering key
    pub fields: BTreeMap<String, Value>, // sorted for determinism
    pub provenance: Provenance,          // { rule_id }
}
```

How stanzas are produced. The mapping evaluator (from NTE [9]) iterates mapping rules, evaluates each rule's selector against the desired topology, and for each match emits a stanza with the rule's `kind` and `fields`. Stanza fields can reference node properties: `fields.local_asn` might resolve to the `asn_base` value that `build_protocol_layer` deep-merged into the node's metadata. The evaluator serialises the `nte_topology::Topology` to bytes and loads it into the NTE wrapper type, bridging the two crate boundaries without leaking internal representations.

Deterministic ordering. `BTreeMap` (not `HashMap`) guarantees sorted field keys. Stanzas are post-sorted by `(kind, key)`. The output is byte-identical across runs and git-diffable.

Provenance. Each stanza records its `rule_id`. At render time, each line of generated config is attributed to the rules that produced it, emitted as a `{hostname}.lineage.json` artefact. This closes the auditability loop: given a line of config, an operator can trace it back to the mapping rule, and from there to the blueprint primitive that created the underlying topology state.

Grouping. Stanzas are grouped by `fields.node` (hostname). Each group is rendered through the template selected by the node's `device_os` field, producing one `.conf` file per device.

7 Configuration Diffing

The diff engine computes a `MutationPlan` between two topologies by exporting each to JSON and comparing:

1. **Node diff:** per layer, compare node sets by ID. Additions, deletions, and modifications (field-by-field property comparison via `extract_properties` unpacking the `data_json` blob).
2. **Edge diff:** compare by (src, dst) pair with structural property comparison.

The `MutationPlan` enables incremental updates: only devices whose properties changed receive new configurations. The `plan` command displays the plan; the `ci-run` command uses it to compute per-device config diffs.

8 Operational Modes

NetCfg provides eleven CLI commands spanning the development-to-deployment lifecycle:

`generate`. runs the full five-stage pipeline, writing `.conf`, `.deviceir.json`, and `.lineage.json` per node. `–emit-ir` additionally writes the raw `DeviceContext` JSON for each node, enabling external tooling to consume the intermediate state.

`validate`. executes the pipeline in dry-run mode: all primitives run with `ctx.dry_run() = true`, skipping DataFrame writes but reporting what *would* change. Optionally validates templates by performing a trial render to memory without writing files. This catches blueprint errors, template syntax issues, and assertion failures without producing output.

`plan`. computes the topology `MutationPlan` (current vs. desired) and displays additions, removals, and property changes. With `–diff`, it also renders before/after configs and shows a unified text diff per device using the `similar` crate.

`inspect`. provides three views into intermediate state: `–ast` prints the blueprint AST tree; `–topology` exports the graph as Graphviz DOT (with per-layer subgraph clustering, hostname labels, and subnet annotations on edges); `–trace <node_id>` replays primitive execution and prints before/after `DeviceContext` snapshots for a specific node at each primitive step.

`ci-run`. is the CI/CD integration point. It runs: (1) `validate` (fail fast on errors), (2) `plan` (compute topology delta), (3) render before/after configs to memory and compute per-device unified diffs. With `–json`, it emits a structured payload:

Listing 8. ci-run JSON output: topology delta and per-device config diffs.

```
{
  "status": "success",
  "topology_additions": 42,
  "topology_removals": 0,
  "configuration_diffs": {
    "core-0": "-  ip address 10.0.0.4/30\n+  ..."
  }
}
```

This payload integrates with GitHub Actions, GitLab CI, or any CI system that can parse JSON—enabling automated config review as part of merge-request workflows.

`import`. converts a YAML topology definition into an NTE archive consumed by all other commands.

`lint`. runs design-rule assertions without the full pipeline. Exit code 1 only on `error`-severity failures; warnings are reported but do not fail.

`fmt`. standardises blueprint YAML formatting via parse-and-reserialise. With `–check`, exits non-zero if changes are needed (CI gate / pre-commit hook).

`impact`. calculates the blast radius of a blueprint change, classifying affected nodes as high (node removal, critical property change, physical link change), medium (node addition, logical link change), or low (metadata-only) risk.

`report`. generates a Markdown architecture report with node inventory, physical link tables, and a `netviz-json` topology visualisation block.

`export-clab`. generates a Containerlab `clab.yaml` simulation file, mapping `device_os` to container kinds and optionally bind-mounting generated configs into OS-specific remote paths.

9 End-to-End Example

We trace a concrete compilation through all stages. Given a base topology with 12 routers across 3 sites (4 per site), and the blueprint from Listing 9:

Listing 9. Three-site mesh blueprint.

```
version: 1
layers:
  - name: input
    primitives:
      - type: mesh_nodes
        selector: "nodes[site='a']"
        mesh_type: full
      - type: mesh_nodes
        selector: "nodes[site='b']"
        mesh_type: full
  - name: input
    primitives:
      - type: provision_ips
        selector: "edges[src>=0]"
        pool: "10.1.0.0/24"
        subnet_size: 30
        strategy: dense
  - name: input
    primitives:
      - type: build_protocol_layer
        selector: "nodes[true]"
        layer: bgp
        config:
          protocol_type: bgp
          asn_base: "65000"
```

Stages 2–3 (Shadow + Execute) produce a desired topology where:

- `mesh_nodes` created $2 \times \binom{4}{2} = 12$ edges (6 per site, sites A and B).
- `provision_ips` allocated /30 subnets from 10.1.0.0/24 to each edge, writing `src_ip`, `dst_ip`, and `subnet` into each endpoint node's `DeviceContext`. Node `site-a-r1` now has `src_ip: "10.1.0.1"`, `subnet: "10.1.0.0/30"`.
- `build_protocol_layer` cloned all 12 physical nodes into the `bgp` layer (IDs 13–24), deep-merging `{protocol_type: "bgp", asn_base: "65000"}` into each clone's metadata. Each BGP node inherits its parent's IP addresses.

Table 1. Per-component and end-to-end benchmark results.

Benchmark	Scale	Time (ms)
Topology generation	1,000 nodes	0.37
CEL selector evaluation	1,000 nodes	1.83
CEL selector evaluation	10,000 nodes	18.56
`provision_ips`	1,000 edges	374
`provision_ips`	10,000 edges	12,657
`mesh_nodes`	10,000 nodes	4,814
`build_protocol_layer`	10,000 nodes	6,395
End-to-end pipeline	1,000 nodes	**372**
End-to-end pipeline	10,000 nodes	**12,499**

Stage 4 (Mapping) evaluates rules against this topology. A mapping rule with `selector: "all_nodes"` and `kind: "bgp"` emits one stanza per node, each with `fields.node` set to the hostname.

Stage 5 (Render) groups stanzas by hostname and renders through `cisco.j2`, producing:

Listing 10. Generated output for site-a-r1.

```
site-a-r1
  ip address 10.1.0.0/30
```

10 Evaluation

All benchmarks use Criterion.rs on a single core of an Apple M-series processor.

Selector evaluation scales linearly. 1.83 ms at 1,000 nodes; 18.56 ms at 10,000—10.2× increase for 10× scale.

Primitives are I/O-bound on DataFrame writes. `provision_ips` at 10,000 edges (12.7 s) is dominated by per-edge writes. Batching into bulk Polars operations is a known optimisation path.

End-to-end is practical. 1,000 nodes in 372 ms (interactive); 10,000 nodes in 12.5 s (CI/CD batch). These scales cover regional ISPs and large enterprises.

11 Related Work

Source-of-truth systems. NetBox [1] and Nautobot [2] provide device inventory and IPAM via REST APIs. Unlike NetCfg, they are database-backed CRUD applications with no graph model, no compilation pipeline, and no config generation. NetCfg's IPAM primitive provides equivalent allocation semantics within the compiler, eliminating the need for a separate IPAM system for topology-scoped allocations.

Automation frameworks. Ansible [3], Nornir [4], and Salt [5] push configs but have no topology model: cross-device relationships must be encoded manually. Batfish [6] verifies but does not produce configs. NetCfg unifies generation and verification in a single pipeline via design-rule assertions.

Network lab tools. netlab [7] generates config for virtual labs from YAML, supporting auto IP allocation and protocol modules. Unlike NetCfg, it targets lab scale (tens of devices), has no structural diff, no transactional execution, no intermediate representation, and no CI/CD integration.

Table 2. Feature comparison with existing tools.

Capability	*NetCfg*	*NetBox*	*Ansible*	*netlab*	*Batfish*
Topology graph model	✓	–	–	✓	✓
IP allocation	✓	✓	–	✓	–
Config generation	✓	–	✓	✓	–
Structural diff	✓	–	–	–	–
Idempotent primitives	✓	✓	–	–	–
Protocol layer model	✓	–	–	✓	✓
Transactional execution	✓	–	–	–	–
CI/CD integration	✓	–	✓	–	–
Config provenance	✓	–	–	–	–
Policy primitives	✓	–	–	–	–
Plugin extensibility	✓	–	✓	–	–
LSP / IDE support	✓	–	–	–	–
Blast radius analysis	✓	–	–	–	–
Lab export	✓	–	–	✓	–

Graph-based network systems. Apstra (Juniper) uses a graph model for intent-based networking but is proprietary and vendor-tied. Forward Networks [10] builds read-only digital twins. NTE [9] provides NetCfg's graph backend; NetCfg adds the domain-specific compilation pipeline, the two-document model, and the DeviceIR phase boundary.

12 Conclusion

We presented NetCfg, a declarative network configuration compiler that bridges the intent–configuration gap through a five-stage compilation pipeline. The two-document model—blueprint for topology transformations, mapping for config extraction—cleanly separates *what* the network should look like from *how* to generate vendor-specific configurations. The shadow topology execution model guarantees deterministic, transactional primitive composition; the DeviceIR intermediate representation enables auditable, reproducible config generation with per-line provenance tracking.

NetCfg compiles 1,000-node topologies in 372 ms and 10,000-node topologies in 12.5 s. Eleven CLI commands—including `generate`, `plan`, `ci-run`, `lint`, `impact`, and `export-clab`—support the full lifecycle from interactive development to automated CI/CD integration. A language server provides real-time diagnostics and auto-completion for blueprint authoring, and a stateless REST API exposes the compiler as a microservice.

Future work includes GPU-accelerated primitive execution via NTE's WebGPU backend, full WASM linear memory I/O for plugin data exchange, and incremental compilation to avoid recompiling unchanged topology regions.

References

[1] DigitalOcean. NetBox: The premier network source of truth. https://netbox.dev/, 2024.
[2] Network to Code. Nautobot: The Network Source of Truth and Automation Platform. https://www.networktocode.com/nautobot/, 2024.
[3] Red Hat. Ansible: Simple IT Automation. https://www.ansible.com/, 2024.
[4] Nornir contributors. Nornir: A pluggable multi-threaded framework. https://nornir.tech/, 2024.

[5] VMware. Salt: Event-driven IT automation. https://saltproject.io/, 2024.
[6] Intentionet. Batfish: Network configuration analysis. https://www.batfish.org/, 2024.
[7] Ivan Pepelnjak. netlab: Virtual networking lab framework. https://netlab.tools/, 2024.
[8] Google. Common Expression Language. https://cel.dev/, 2024.
[9] Simon Knight. NTE: A Hybrid Graph-Relational Engine for Network Topology Management. Technical Report, 2026.
[10] Forward Networks. Forward Enterprise: Network assurance platform. https://www.forwardnetworks.com/, 2024.