NetCfg: A Declarative Network Configuration Compiler

Technical Reference

Simon Knight

March 2026 • Version 1.1

Abstract. NetCfg is a Rust-based network configuration compiler that transforms high-level topology intent into verified, target-specific device configurations. The compiler processes intent through a five-stage pipeline: (1) Intent Transformation via composable primitives, (2) Topological Validation via a DSL-driven assertion engine, (3) Instantiation of a vendor-neutral Device Intermediate Representation (DeviceIR), (4) Target-Specific AST Transformation (TSDM) for hardware-aware transformation, and (5) Syntax Serialisation via simple templates. This report documents the data model, the transformation DSLs, and the AST-based instantiation mechanism that decouples architectural logic from physical hardware constraints.

Version	Date	Changes
1.0	March 2026	Initial release (v1.3 feature set)
1.1	March 2026	Updated to v2.0: policy primitives, WASM plugins, LSP, REST API, 4 new CLI commands

Contents

1 Introduction

What NetCfg is

NetCfg is a command-line tool and Rust library that compiles declarative network topology descriptions into per-device configuration files. It takes a *blueprint* (describing what the network should look like) and a *mapping* (describing how to extract vendor-specific configuration from the topology) and produces deterministic, diffable, auditable configuration artefacts for every device in the network.

NetCfg replaces the manual integration layer between source-of-truth systems (NetBox, Nautobot), automation frameworks (Ansible, Nornir), and network modelling tools (Batfish). It treats network configuration as a *compilation problem*: topology intent is compiled through a typed pipeline, not interpolated through ad-hoc templates.

Design philosophy

Two principles explain every design decision in this document:

1. **Instantiation over templating.** The operator's intent is instantiated through successive stages of abstraction. It moves from a high-level graph to a vendor-neutral DeviceIR, then to a platform-specific DeviceIR (via TSDM), and only finally to CLI syntax. This ensures that hardware quirks and vendor-specific logic are handled via structured AST transformations, not ad-hoc string concatenation.
2. **Separation of what from how.** The blueprint declares *what* the network should look like (topology transformations). The mapping declares *how* to extract device configuration from the result. The same blueprint can produce Cisco IOS, Arista EOS, or Juniper JunOS output by swapping only the mapping and templates.

Audience

This document is intended for:

- Engineers integrating NetCfg into a network automation pipeline.
- Contributors to the NetCfg and NTE codebases.
- Researchers evaluating graph-based approaches to network configuration.

Familiarity with YAML, basic graph theory, and IP networking is assumed. Experience with Rust is helpful but not required for the API sections.

How to read this document

Readers wanting to get started quickly should read §2 (data model), §4 (blueprint DSL), and §7 (CLI reference), then refer to individual sections as needed. §5–§6 cover the compilation pipeline and core mechanisms in depth and can be deferred until required. §8 provides a complete end-to-end worked example that traces a blueprint through every pipeline stage. §10 documents the language server and REST API for editor and CI/CD integration.

2 Background: Network Configuration as Code

NetCfg models 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 $D : T \to DataFrame$ maps each type to a columnar property store (Polars DataFrame).

2.1 Layers

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* (`_source_id` in the node's `data_json`).

Layers allow the compiler to model the physical and logical topologies simultaneously, with cross-layer references preserving traceability from a protocol node back to the physical device that hosts it.

2.2 Per-node state: DeviceContext

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.

```rust
pub struct DeviceContext {
    pub hostname: String,
    pub device_os: Option<String>,        // template selector
    pub management_ip: Option<String>,
    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 subnet_v6: Option<String>,
    pub vrf: Option<String>,              // VRF domain
    pub peer_interfaces: Option<HashMap<String, String>>,
    pub next_interface_index: Option<u32>,
    pub stanzas: Vec<Stanza>,             // accumulated config units
    pub metadata: HashMap<String, JsonValue>, // extension point
}
```

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

> **Note**
>
> The `metadata` field uses `serde(flatten)`, so any JSON key not matching a named field is captured here. This makes `DeviceContext` extensible without schema changes.

2.3 Graph engine dependency

The underlying graph is implemented by the NTE topology engine, which provides a directed graph (built on the `petgraph` Rust crate) with Polars DataFrames for columnar property storage. NetCfg depends on `nte_topology` for the graph and `ank_nte` for the mapping evaluator and DeviceIR types.

3 Intent and Policy Expression

A blueprint is more than a list of topology mutations. It is a *plan document*: a declaration of desired state expressed at a level above the topology graph itself. This section frames the conceptual model that operators use when writing blueprints, before §4 catalogues the concrete syntax.

3.1 The five concerns of a network blueprint

Every blueprint addresses five orthogonal concerns. Table 1 shows how each maps to a DSL construct and the design question it answers.

Table 1. The five concerns of a network blueprint.

Concern	DSL construct	Question answered
Population naming	`groups:`	Who are my spines, leaves, tenants?
Topology intent	`layers:` / primitives	How should they be connected and addressed?
Policy constraints	routing/access/prefix-list primitives	What traffic rules should be enforced?
Design invariants	`assertions:` + `rules:`	What properties must always hold?
Change safety	`blast_radius:`	How much change is acceptable per commit?

The key insight is that the operator never manipulates the topology graph directly. They declare *populations*, *intent*, *policy*, and *invariants*—and the compiler maps these through standard design rules into concrete graph mutations.

3.2 From plan to topology: the instantiation model

Blueprint constructs form a chain of successive refinement:

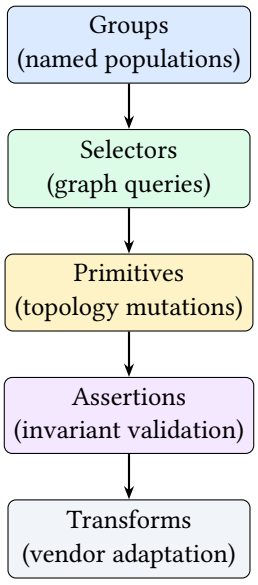

Figure 1. Instantiation chain from plan-stage constructs to vendor-specific output.

Groups and selectors mediate between stages. Groups let the operator name populations once (`$spines`, `$leaves`, `$equities_links`) and reference them everywhere. Selectors compile these references into CEL graph queries. Primitives consume selector results and mutate the topology. Design-rule assertions then validate the *result* after primitives execute, catching violations before commit. Finally, transforms adapt the vendor-neutral DeviceIR to platform-specific syntax.

3.3 Expressing policy at the plan layer

To make this concrete, consider the multi-tenant data-centre case study (`examples/case_studies/04_multi_tenant_d`). The high-level intent is "tenants must be isolated". The operator decomposes this into plan-layer constructs:

1. **Name the populations** — `groups:` defines `$spines`, `$leaves`, `$equities_links`, `$fixed_income_links`, `$risk_links`, and per-tenant union groups like `$equities_or_spines`.
2. **Build the fabric** — `mesh_nodes` with `hub_and_spoke` creates the spine–leaf physical topology.
3. **Address per-tenant** — `provision_ips` with `vrf: equities`, `vrf: fixed-income`, and `vrf: risk` isolates address space into per-tenant VRFs.
4. **Enforce isolation** — `build_access_policy` with an `INTER-VRF-DENY` policy blocks cross-VRF traffic at the spine.
5. **Validate invariants** — assertions check: `tenant-assigned` (every leaf has a tenant), `vrf-configured` (every leaf has a VRF), `vlans-unique-per-tenant` (no duplicate VLANs within a tenant), and per-tenant `is_connected` checks verify that each tenant's subgraph is reachable.
6. **Constrain change** — `blast_radius` limits deletions to three items and caps overall change at 25%.

At no point does the operator add individual edges or set IP addresses on specific interfaces. The plan layer abstracts these details away: populations are named once, intent is expressed through composable primitives, and invariants guard the result.

3.4 Reusable design-rule libraries

Complex assertions can be factored into named `rules:` (CEL macros) and shared across blueprints via `imports:`. Two standard libraries ship with NetCfg:

- **`datacenter-rules.yaml`** — a portable linting ruleset for data-centre fabrics. It validates OSPF point-to-point subnet lengths, MTU consistency across edges, BGP private-ASN ranges, BGP neighbour IP existence, IS-IS level consistency, and IS-IS NET address formatting. Blueprints import it with a single line: `imports: ['datacenter-rules.yaml']`.
- **`hardware-transforms.yaml`** — vendor-specific AST transforms for Cisco NX-OS, Juniper Junos, Cisco IOS-XR, and VyOS. It remaps abstract `Ethernet` interface names into platform-specific formats (e.g. `Ethernet0` → `ge-0/0/0` on Junos) and standardises overlay MTU to 9216.

Assertions reference named rules via the `use_rule` check type, keeping the assertion block concise:

Listing 2. Referencing named rules.

```yaml
rules:
  tenant-assigned: "has(node.tenant) && node.tenant != ''"
  vrf-configured:  "has(node.vrf_name) && node.vrf_name != ''"

assertions:
  - name: every-leaf-has-tenant
    severity: error
    select: $leaves
    check:
      type: use_rule
      name: tenant-assigned
```

4 Blueprint DSL: Declarative Topology Transformations

A blueprint is a YAML document declaring what topology transformations to apply. It is the "what" of the network design.

4.1 Schema and layer ordering

Listing 3. Blueprint top-level schema.

```rust
pub struct Blueprint {
    pub version: u32,                   // must be >= 1
    pub imports: Vec<String>,           // relative paths to merge
    pub layers: Vec<LayerSpec>,         // ordered layer blocks
    pub assertions: Vec<AssertionSpec>, // post-execution checks
    pub rules: HashMap<String, String>, // named CEL macros
    pub groups: HashMap<String, GroupSpec>, // named node sets
    pub blast_radius: Vec<BlastRadiusSpec>, // change policies
    pub transforms: Vec<TransformationSpec>, // vendor adaptation
    pub hardware_library: HardwareLibrary,  // chassis/linecard defs
}

pub struct LayerSpec {
    pub name: String,             // non-empty
    pub requires: Vec<String>,    // must name earlier layers
    pub primitives: Vec<Primitive>,
}
```

Three validation passes run at parse time:

1. **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.
2. **Selector compilation**: each selector string is compiled into an evaluable Common Expression Language (CEL) program, surfacing syntax errors before execution.
3. **Layer ordering validation**: a forward scan ensures every `requires` entry names a previously-declared layer. This catches ordering bugs at parse time.

The optional `imports` field lists relative file paths to other blueprint files. At parse time, layers and assertions from imported files are merged into the importing blueprint, enabling reusable assertion libraries (e.g. a standard linting ruleset shared across projects).

> **Warning**
>
> Duplicate layer names are permitted. Multiple `LayerSpec` blocks with `name: input` are common in practice—each block adds primitives to the same layer. The `requires` mechanism enforces inter-layer ordering, not intra-layer uniqueness.

4.2 Selector DSL (CEL surface syntax)

Selectors identify which nodes or edges a primitive operates on. The surface syntax is a lightweight wrapper around Google's Common Expression Language (CEL):

Listing 4. Selector syntax examples.

```yaml
# Select all nodes
selector: "all()"

# Select nodes by property
selector: "nodes[site='a']"
selector: "nodes[role=='spine' && layer=='input']"
selector: "nodes[type=='Router' and site='dc1']"
```

```
# Select edges
selector: "edges[src>=0]"
selector: "edges[true]"
```

The selector parser:

1. Extracts the target (`nodes` or `edges`) and the filter expression from the bracket syntax.
2. Normalises operators: single `=` becomes `==`; `and`/`or`/`not` become `&&`/`||`/`!`; single `&`/`|` become doubled.
3. Compiles the normalised expression as a CEL program.

At evaluation time, the selector iterates over all nodes (or edges) in the topology, binds each entity's properties as CEL variables (`id`, `type`, `layer`, plus all `data_json` fields), and evaluates the CEL program. Entities for which the program returns `true` are included in the match set.

> **Note**
>
> Undeclared variable references (e.g., querying a field that some nodes lack) silently evaluate to `false` rather than producing an error. This allows heterogeneous topologies where not all nodes have the same properties.

For IP-valued fields, the selector also binds a `{field}_len` variable containing the prefix length, enabling selectors like `nodes[subnet_len == 30]`.

4.3 Named groups

The `groups:` section defines reusable named node or edge sets. Group names are prefixed with `$` when referenced in selectors. Two forms are supported:

Listing 5. Group definitions: shorthand and full form.

```
groups:
  # Shorthand: just a selector string
  spines: "nodes[role=='spine']"
  leaves: "nodes[role=='leaf']"

  # Full form with kind and kind-specific fields
  dmz:
    selector: "nodes[zone=='dmz']"
    kind: security_zone
    trust_level: 10
```

Groups are resolved before primitives execute. The selector parser expands `$group_name` references inline, and cycle detection rejects circular group dependencies at parse time. When blueprints import other files, groups from imported files are merged into the importing blueprint's namespace (name collisions are last-writer-wins).

The union operator (`|`) combines groups: `spines_and_leaves: $spines | $leaves` creates a set containing all nodes matching either group.

4.4 Primitive catalogue

NetCfg provides twenty primitive types organised into five tiers: *Topology*, *Allocation*, *Policy*, *Overlay*, and *Meta*. All primitives follow 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.

Table 2 summarises the catalogue. Full field tables follow.

Table 2. Primitive catalogue overview.

Primitive	Tier	Purpose
`mesh_nodes`	Topology	Create edges between selected nodes (full, hub-and-spoke)
`build_protocol_layer`	Topology	Clone nodes into a named protocol overlay layer
`provision_ips`	Allocation	Allocate IP subnets to edges from a CIDR pool
`allocate_resources`	Allocation	Generic pool allocation for ASNs, VLANs, etc.
`global_pool`	Allocation	Register a named IP pool for hierarchical carving
`global_resource_pool`	Allocation	Register a named resource pool (non-IP)
`build_routing_policy`	Policy	Attach routing policy stanzas (route-maps)
`build_access_policy`	Policy	Attach access-control policy stanzas (ACLs)
`build_prefix_list`	Policy	Create named prefix-list entries on matched nodes
`build_community_list`	Policy	Create named community-list entries on matched nodes
`generate_safe_bgp_filters`	Policy	Auto-generate RFC 1918 bogon filters
`build_vxlan`	Overlay	Assign VXLAN VNI and multicast group bindings
`build_evpn`	Overlay	Assign EVPN route-distinguisher and route-target
`tag_nodes`	Meta	Stamp key–value metadata onto matched nodes
`map_hardware_inventory`	Meta	Assign chassis model and line-card slot mappings
`assert_state`	Meta	Mid-pipeline assertion checkpoint
`conditional`	Meta	CEL-based branching within blueprints
`custom_primitive`	Meta	Named reusable primitive group with parameters
`inject_secrets`	Meta	Read environment variables into node metadata
`wasm_plugin`	Meta	Execute a WebAssembly plugin against matched nodes

4.4.1 `mesh_nodes`

Creates edges between selected nodes. Supports three mesh topologies: `full` ($\binom{n}{2}$ edges), `hub_and_spoke` (hubs full-meshed, spokes connected to all hubs), and `route_reflector` (same topology as hub-and-spoke). Idempotent: existing edges are detected via a `HashSet<(i32, i32)>` and skipped.

Table 3. `mesh_nodes` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for nodes to mesh
`mesh_type`	String	Yes	`full`, `hub_and_spoke`, or `route_reflector`
`hub_selector`	String	H&S	Selector for hub nodes
`spoke_selector`	String	H&S	Selector for spoke nodes
`edge_properties`	JSON	No	Properties for created edges (deferred; see §11.1)
`naming_strategy`	String	No	Interface naming: `eth`, `cisco_ge`, `junos_ge`

Example: full mesh within a site

```
- type: mesh_nodes
  selector: "nodes[site='a']"
  mesh_type: full
  naming_strategy: eth
```

With 4 nodes at site A, this creates $\binom{4}{2} = 6$ edges and assigns interface names `eth0`, `eth1`, etc. to each node's `peer_interfaces` map.

4.4.2 `provision_ips`

Allocates subnets from a CIDR pool to edges. See §6.1 for the full five-pass algorithm.

Table 4. `provision_ips` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for edges
`pool`	String	Yes	CIDR (e.g. `10.0.0.0/24`) or global pool name
`subnet_size`	u8	No	Prefix length per edge (default: 30 for IPv4, 126 for IPv6)
`ipv6_pool`	String	No	IPv6 CIDR or global pool name for dual-stack
`ipv6_subnet_size`	u8	No	IPv6 prefix length per edge
`strategy`	String	No	`dense` (default), `sparse`, or `hash`
`pool_size`	u32	No	When referencing a global pool, size of block to carve
`vrf`	String	No	VRF domain (default: `default`)

4.4.3 `build_protocol_layer`

Clones physical nodes into a named protocol layer. See §6.2 for the overlay construction mechanism.

Table 5. `build_protocol_layer` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for source-layer nodes
`layer`	String	Yes	Target layer name (must differ from source)
`config`	JSON	Yes	Config overlay deep-merged into cloned nodes
`clone_underlying`	bool	No	Also clone edges between selected nodes (default: false)

4.4.4 `allocate_resources`

Generic pool allocator for non-IP resources (ASNs, VLANs, loopback IDs).

Table 6. `allocate_resources` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for nodes
`resource_type`	String	Yes	Key name in `DeviceContext.metadata`
`pool`	String	Yes	Range (e.g. `65000-65999`) or global pool name
`strategy`	String	No	`dense` (default)
`pool_size`	u32	No	When referencing a global pool, number of values to carve

4.4.5 `global_pool`

Registers a named IP address pool at blueprint scope, enabling hierarchical pool carving across multiple `provision_ips` primitives.

Table 7. `global_pool` fields.

Field	Type	Required	Description
`name`	String	Yes	Pool identifier referenced by `provision_ips`
`pool`	String	Yes	CIDR range (e.g. `10.0.0.0/8`)
`vrf`	String	No	VRF domain (default: `default`)

4.4.6 `global_resource_pool`

Registers a named non-IP resource pool at blueprint scope.

Table 8. `global_resource_pool` fields.

Field	Type	Required	Description
`name`	String	Yes	Pool identifier
`resource_type`	String	Yes	Resource category (e.g. `asn`)
`pool`	String	Yes	Value range (e.g. `65000-65999`)

4.4.7 `conditional`

CEL-based branching. The `condition` is evaluated as a CEL expression; if true, `then_primitives` execute, otherwise `else_primitives` (if present) execute.

Table 9. `conditional` fields.

Field	Type	Required	Description
`condition`	String	Yes	CEL expression
`then_primitives`	Vec<Primitive>	Yes	Primitives if condition is true
`else_primitives`	Vec<Primitive>	No	Primitives if condition is false

4.4.8 `custom_primitive`

A named, reusable group of primitives with parameters. The `parameters` map is pushed onto a stack and accessible within the nested primitives.

Table 10. `custom_primitive` fields.

Field	Type	Required	Description
`name`	String	Yes	Primitive name (for reporting)
`parameters`	HashMap<String, JsonValue>	Yes	Named parameters
`primitives`	Vec<Primitive>	Yes	Nested primitive list

4.4.9 `inject_secrets`

Reads environment variables into node metadata. Keys in the `secrets` map are metadata field names; values are environment variable names.

Table 11. `inject_secrets` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`secrets`	HashMap<String, String>	Yes	{metadata_key: env_var_name}

4.4.10 `build_routing_policy`

Attaches routing policy stanzas to matched nodes. Each statement defines a route-map entry with match conditions and set actions, stored as a `Stanza` in `DeviceContext.stanzas` for downstream template rendering.

Table 12. `build_routing_policy` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`name`	String	Yes	Policy name (e.g. `BGP-EXPORT`)
`statements`	Vec<RoutingPolicyStatement>	Yes	Ordered route-map entries

Each `RoutingPolicyStatement` has fields: `name` (sequence number), `action` (`permit` or `deny`), optional `match_condition`, and optional `set_action`. Templates access stanzas via `device.stanzas | selectattr('kind', 'equalto', 'routing_policy')`.

Example: BGP export policy

```yaml
- type: build_routing_policy
  selector: "nodes[role='border']"
  name: "BGP-EXPORT"
  statements:
    - name: "10"
      action: permit
      match_condition: "prefix-list CUSTOMER"
      set_action: "local-preference 100"
    - name: "20"
      action: deny
```

4.4.11 `build_access_policy`

Attaches access-control policy stanzas to matched nodes. Each rule defines a firewall or ACL entry with source, destination, protocol, and port fields.

Table 13. `build_access_policy` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`name`	String	Yes	Policy name (e.g. `EDGE-ACL`)
`rules`	Vec<AccessPolicyRule>	Yes	Ordered ACL entries

Each `AccessPolicyRule` has fields: `name`, `action` (`permit`/`deny`), optional `source`, `destination`, `protocol`, and `port`. Both policy primitives enforce `SelectorTarget::Nodes`—policies cannot target edges.

Example: DMZ access control list

```yaml
- type: build_access_policy
  selector: "nodes[zone='dmz']"
  name: "UNTRUSTED-TO-INTERNAL"
  rules:
    - name: "100"
      action: deny
      protocol: tcp
      destination: "10.0.0.0/8"
    - name: "999"
      action: permit
```

4.4.12 `wasm_plugin`

Executes a WebAssembly plugin binary against matched nodes. Requires the `wasm` Cargo feature flag (`--features wasm`); without it, the pipeline returns a descriptive error. Uses the `wasmtime` v18 runtime.

Table 14. `wasm_plugin` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`plugin_path`	String	Yes	Filesystem path to compiled `.wasm` binary

The execution pipeline: (1) load and compile the `.wasm` module, (2) instantiate per matched node without host imports, (3) resolve the `apply_node` exported symbol, (4) pass the node's `DeviceContext` as serialised JSON. The current implementation validates compilation and symbol resolution; full linear memory I/O for JSON exchange is planned for a future release (see §12).

> **Note**
>
> The `wasm` feature is optional to avoid the `wasmtime` dependency (which adds ~40 MB to the binary) for users who do not need plugin extensibility.

4.4.13 `build_prefix_list`

Creates named prefix-list entries on matched nodes, stored as `Stanza` objects for downstream template rendering.

Table 15. `build_prefix_list` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`prefix_list_name`	String	Yes	Name of the prefix list
`entries`	Vec<PrefixListEntry>	Yes	Ordered prefix-list entries

Each `PrefixListEntry` has: `prefix` (CIDR string), `action` (`permit`/`deny`), optional `le` (less-than-or-equal prefix length), and optional `ge` (greater-than-or-equal prefix length).

4.4.14 `build_community_list`

Creates named community-list entries on matched nodes.

Table 16. `build_community_list` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`community_list_name`	String	Yes	Name of the community list
`entries`	Vec<CommunityListEntry>	Yes	Ordered community-list entries

Each `CommunityListEntry` has: `community` (e.g. `65001:1000`) and `action` (`permit`/`deny`).

4.4.15 `generate_safe_bgp_filters`

Auto-generates RFC 1918 private-address (bogon) filters as prefix-list and routing-policy stanzas on matched nodes.

Table 17. `generate_safe_bgp_filters` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`prefix_list_name`	String	No	Name for the generated prefix list
`policy_name`	String	No	Name for the generated routing policy
`block_rfc1918`	bool	No	Block RFC 1918 prefixes (default: true)
`permit_default`	bool	No	Add a trailing permit-any (default: false)

4.4.16 `build_vxlan`

Assigns VXLAN tunnel bindings to matched nodes. VNIs are allocated sequentially from `vni_base`.

Table 18. `build_vxlan` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for VTEP nodes
`vni_base`	u32	Yes	Starting VNI (e.g. `10000`)
`mcast_group_base`	String	No	Multicast group for BUM (Broadcast, Unknown-unicast, Multicast) replication

4.4.17 `build_evpn`

Assigns EVPN route-distinguisher (RD) and route-target (RT) values to matched nodes.

Table 19. `build_evpn` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for EVPN nodes
`route_distinguisher_base`	String	Yes	RD base (e.g. `auto`)
`route_target_base`	String	Yes	RT base (e.g. `65001:10000`)

4.4.18 `tag_nodes`

Stamps flat key–value metadata onto matched nodes. Tags are merged into each node's `data_json` and are available to subsequent selectors and assertions.

Table 20. `tag_nodes` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`tags`	HashMap<String, JsonValue>	Yes	Key–value pairs to stamp

4.4.19 `map_hardware_inventory`

Assigns chassis model and line-card slot mappings to matched nodes. Used by the TSDM layer (§6.4) to resolve abstract interface names into physical slot/port identifiers.

Table 21. `map_hardware_inventory` fields.

Field	Type	Required	Description
`selector`	String	Yes	CEL selector for target nodes
`chassis_model`	String	Yes	Chassis identifier (e.g. `dcs-7508`)
`slots`	HashMap<u32, String>	Yes	Slot → line-card model mapping

4.4.20 `assert_state`

A mid-pipeline assertion checkpoint. Syntactically identical to a top-level `AssertionSpec` but placed inline within a layer's primitive list. This allows the operator to validate intermediate state between primitives rather than waiting until the post-execution assertion pass.

Table 22. `assert_state` fields.

Field	Type	Required	Description
`name`	String	Yes	Assertion name (for diagnostics)
`severity`	String	No	`error` (default) or `warning`
`select`	String	Yes	CEL selector for items to check
`check`	AssertionCheck	Yes	Predicate (same types as top-level assertions)
`help`	String	No	Remediation hint shown on failure

4.5 Design-rule assertions

Blueprints may declare assertions: post-execution invariants checked after all primitives complete but before commit. Each assertion has a severity: `error` (pipeline failure) or `warning` (diagnostic only).

Listing 6. Design-rule assertions 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"
  - name: "unique hostnames per site"
    severity: warning
    select: all()
    check:
      unique_per_group:
```

```
        field: hostname
        group_by: site
  - name: "private ASN range"
    severity: warning
    select: "nodes[asn != null]"
    check:
      custom_cel:
        expression: "asn >= 64512 && asn <= 65534"
```

Thirteen check types are available:

Table 23. Assertion check types.

Check	Semantics
`field_exists`	Every matched node has the named field in its `data_json`
`min_count`	The matched set contains at least N items
`max_count`	The matched set contains at most N items
`min_edges`	Every matched node has at least N outgoing edges
`unique_per_group`	Within groups defined by `group_by`, the `field` value is unique
`field_in_cidr`	Every matched node's IP field is contained within the specified CIDR
`custom_cel`	Arbitrary CEL expression evaluated per matched entity; fails if false
`use_rule`	Reference a named CEL macro from the `rules:` section
`is_connected`	The matched subgraph (nodes + interconnecting edges) is connected
`is_acyclic`	The matched subgraph contains no cycles
`is_bipartite`	The matched subgraph is bipartite (two-colourable)
`reachability`	Every matched node can reach at least one node in a target selector
`match_schema`	Each matched node's `data_json` validates against a JSON Schema

This enables "policy as code" directly in the blueprint. Assertions are evaluated cross-layer (no layer filter), so a single assertion can validate properties across both physical and protocol layers.

4.6 Named rules (CEL macros)

The `rules:` section defines named CEL macros that can be referenced by assertions via the `use_rule` check type. This avoids duplicating complex CEL expressions across multiple assertions.

Listing 7. Named rules and their use in assertions.

```
rules:
  tenant-assigned: "has(node.tenant) && node.tenant != ''"
  vrf-configured:  "has(node.vrf_name) && node.vrf_name != ''"
  leaf-has-vlan:   "has(node.vlan_id) && node.vlan_id > 0 && node.vlan_id < 4095"
```

Rules are simple string-valued CEL expressions stored in a `HashMap<String, String>`. At evaluation time, a `use_rule` check looks up the named rule and evaluates it as though it were an inline `custom_cel` expression. Rules from imported files are merged into the importing blueprint's rule namespace.

4.7 Blast radius policies

The `blast_radius:` section declares change-safety policies that constrain how much the topology may change in a single commit. Each policy specifies a check against the computed `MutationPlan`:

Table 24. Blast radius check types.

Check	Semantics
`max_deletions`	At most N nodes or edges may be deleted
`max_modifications`	At most N nodes or edges may be modified
`max_percentage_changed`	At most P% of the topology may be altered (0.0–100.0)

An optional `select` field restricts the policy scope to a subset of the topology; when omitted, the policy applies to the entire mutation plan.

Listing 8. Blast radius policies.

```
blast_radius:
  - name: tenant-vrf-protection
    check:
      type: max_deletions
      count: 3
  - name: fabric-change-cap
    check:
      type: max_percentage_changed
      value: 25.0
```

4.8 Target-specific transforms

The `transforms:` section declares AST-to-AST rewrite rules that adapt vendor-neutral stanzas into platform-specific syntax. Each `TransformationSpec` has a `name`, an optional `when` predicate (CEL expression evaluated per device), and a sequence of `RewriteRule` entries:

Table 25. Transform rewrite rule fields.

Field	Type	Required	Description
`name`	String	Yes	Transform name
`when`	String	No	CEL predicate (e.g. `device_os == 'nxos'`)
`rules`	Vec<RewriteRule>	Yes	Sequence of match/apply rewrites

Each `RewriteRule` has a `match_expr` (CEL expression selecting stanzas) and an `apply` map (field name → CEL expression computing the new value):

Listing 9. Vendor-specific transform.

```
transforms:
  - name: nxos-interface-rewrite
    when: "device_os == 'nxos'"
    rules:
      - match_expr: "stanza.kind == 'interface'"
        apply:
          name: "'Ethernet1/' + string(stanza.fields.abstract_index + 1)"
```

5 Compilation Pipeline

NetCfg's compilation pipeline has five distinct stages, operating as a true network compiler. Figure 2 shows the data flow from abstract intent to concrete syntax.

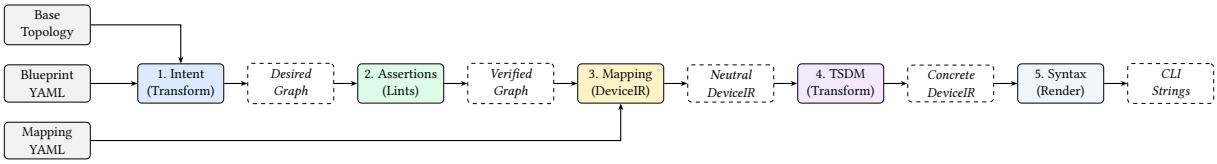

Figure 2. NetCfg compilation pipeline. High-level intent is progressively instantiated through three distinct representations (Verified Graph, Neutral DeviceIR, Concrete DeviceIR) before serialisation.

5.1 Stage 1: Intent Transformation

The blueprint is parsed into an AST and applied to a transactional shadow copy of the topology. Composable primitives (IPAM, Meshing, Overlays) mutate the graph to reach the desired state.

5.2 Stage 2: Topological Assertions

A DSL-driven assertion engine validates the desired graph against architectural invariants (e.g. area IDs must match, MTU consistency). Compilation aborts if invariants are violated.

5.3 Stage 3: DeviceIR Mapping

A Mapping DSL instantiates the rich topology graph into a vendor-neutral **Device Intermediate Representation (DeviceIR)**. This representation captures logical intent (e.g. "Neighbor X is remote-as Y") without hardware context.

5.4 Stage 4: Target-Specific Transform (TSDM)

The **TSDM stage** performs DeviceIR-to-DeviceIR transformations. It transforms the neutral DeviceIR into a platform-specific DeviceIR, remapping logical interfaces to physical ports (e.g. `Eth1` → `Eth1/1/1`) based on the target's chassis model and hardware layout.

5.5 Stage 5: Syntax Serialisation

Jinja2 templates act as a simple emission stage, serialising the final Target-Specific AST into vendor-specific CLI syntax. No transformation logic is applied at this stage; templates perform serialisation only.

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 tracing).

5.5.1 Determinism guarantee

Given a fixed blueprint B, mapping M, and input topology G, the output configuration set $C = compile(G, B, M)$ is deterministic. 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` and are post-sorted. Template rendering is pure.

6 Core Mechanisms

6.1 Five-pass IPAM (`provision_ips`)

The IP allocator (`primitives/ipam.rs`) implements a five-pass algorithm:

1. **State discovery**: Scan all nodes for existing IP assignments (`src_ip`, `dst_ip`, `subnet` fields).
2. **Edge selection**: Evaluate the CEL selector; sort matched edges by (src, dst) for determinism.
3. **Pre-reservation**: Insert existing allocations into the allocator, preventing re-allocation and enforcing stickiness.
4. **Fulfilment**: For each edge, allocate a subnet using the configured strategy (`dense`, `sparse`, or `hash`). Extract host addresses from the subnet.
5. **Write-back**: Write `src_ip`, `dst_ip`, `subnet` (and IPv6 equivalents for dual-stack) 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. If topology IDs change (e.g., after a re-import) but hostnames remain the same, allocations are preserved.

VRF isolation. The allocator maintains per-VRF state. Two primitives using the same CIDR pool in different VRFs can allocate overlapping subnets without conflict. Pool overlap *within* the same VRF is detected and rejected.

Allocation strategies:

- `dense`: Sequential allocation from the pool start. First edge gets the first available subnet.
- `sparse`: Skips every other subnet, leaving growth headroom between allocations.
- `hash`: Subnet position is derived from a hash of the edge identity, producing topology-independent allocations that do not depend on evaluation order.

Hierarchical pools. A `global_pool` primitive registers a large CIDR (e.g., `10.0.0.0/8`). Subsequent `provision_ips` primitives reference the pool by name and specify a `pool_size` (e.g., 24), carving /24 blocks from the parent pool on demand.

Dual-stack. When `ipv6_pool` is specified, both IPv4 and IPv6 addresses are allocated in a single primitive invocation. IPv6 fields (`src_ipv6`, `dst_ipv6`, `subnet_v6`) are written alongside IPv4 fields.

6.2 Protocol overlay construction (`build_protocol_layer`)

`build_protocol_layer` clones physical nodes into a named protocol layer. For each selected source-layer node:

1. Allocate a new monotonic node ID.
2. Add the node to the target layer.
3. Deep-merge the config overlay into the cloned node's metadata.
4. Record `_source_id` as a parent mapping to the physical node.
5. Optionally (`clone_underlying: true`) clone edges between source-layer nodes into the overlay.

Cross-layer traceability: the `_source_id` field 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 deep merge—this is how IP addresses flow from IPAM allocation to BGP configuration without explicit wiring.

Deep merge semantics: When both the base node and the config overlay contain nested JSON objects under the same key, the merge recurses key-by-key (overlay wins on conflict). This preserves sibling keys that the overlay does not mention. For example, a base node with `{bgp: {as: 65000, holdtime: 180}}` and an overlay with `{bgp: {timers: {keepalive: 60}}}` produces `{bgp: {as: 65000, holdtime: 180, timers: {keepalive: 60}}}`.

Idempotency: The primitive scans the target layer for existing nodes with `_source_id` matching source nodes. Already-cloned nodes are skipped, so re-running the primitive after a partial failure produces the correct result.

6.3 Configuration diffing (DiffEngine)

The diff engine (`diff/engine.rs`) computes a `MutationPlan` between two topologies by exporting each layer to JSON and comparing:

1. **Node diff**: Per layer, compare node sets by ID. Detect additions, deletions, and modifications (field-by-field property comparison via `extract_properties`, which unpacks the `data_json` blob).
2. **Edge diff**: Compare by (src, dst) pair with structural property comparison.

The `MutationPlan` contains six change types: `AddNode`, `RemoveNode`, `UpdateNode`, `AddEdge`, `RemoveEdge`, `UpdateEdge`. Changes are sorted deterministically by (change_type, layer, id/src/dst) for reproducible output.

The `plan` command displays the plan as a human-readable diff; the `ci-run` command uses it to compute per-device configuration diffs.

6.4 AST Transformations (TSDM Stage)

The **Target-Specific Device Model (TSDM)** stage implements the compiler's transformation phase via DSL-driven **AST-to-AST rewrite rules**. This stage adapts the vendor-neutral representation to physical hardware constraints.

6.4.1 The Transformation DSL

Transformation rules are defined in the blueprint using a declarative rewrite-rule grammar. Each rule consists of a `match_expr` (CEL predicate) and an `apply` map (field mutations).

Example: NX-OS interface transformation

```
transforms:
  - name: "nxos_transform"
    when: "device_os == 'nxos'"
    rules:
      - match_expr: "kind == 'interface' && name.startsWith('Ethernet')"
        apply:
          name: "name + '/1'"  # Ethernet1 -> Ethernet1/1
          mtu: "9216"          # Force jumbo frames
```

6.4.2 Execution Logic

The transformation engine (`dsl/transform.rs`) performs a recursive traversal of the DeviceIR stanzas. For each stanza, it:

1. Binds the stanza's fields as CEL variables.
2. Evaluates the `match_expr` predicate.
3. If matched, executes the `apply` expressions to mutate the stanza's state.

This structured approach allows complex hardware logic—such as mapping logical ports to specific line-card slots in a modular chassis—to be expressed as declarative rewrite rules rather than fixed code or ad-hoc templates.

The key architectural insight is that `DeviceContext` accumulates state across the entire primitive pipeline:

1. `mesh_nodes`: writes `mesh_type`, `peer_count`, and optionally `peer_interfaces` into metadata.
2. `provision_ips`: writes `src_ip`, `dst_ip`, `subnet`, `vrf`.
3. `build_protocol_layer`: deep-merges all of the above into the cloned protocol node, plus the config overlay.
4. `allocate_resources`: writes the allocated resource value (e.g., `asn`) into metadata.

By the time Stage 4 (rendering) occurs, each node's `DeviceContext` carries hostname, IPs, protocol metadata, interface assignments, and any custom fields—the complete information needed to produce a device configuration file.

7 CLI Reference

NetCfg provides eleven CLI commands spanning the development-to-deployment lifecycle. All commands use `clap` for argument parsing.

7.1 `netcfg generate`

Runs the full five-stage pipeline, writing artefacts to disk.

```
netcfg generate <topology> <blueprint> \
    --templates-dir <DIR> \
    [--output-dir <DIR>] [--emit-ir] [--verbose]
```

Argument	Default	Description
`topology`	—	Path to base topology archive (.nte)
`blueprint`	—	Path to blueprint YAML
`–templates-dir`	—	Directory containing `.j2` templates
`–output-dir`	`./output`	Output directory for generated files
`–emit-ir`	false	Also write `{hostname}.ir.json`
`–verbose`	false	Print per-node rendering progress

Output: one `.cfg` file per device (plus `.ir.json` if `–emit-ir`). Absolute paths are printed to stdout, one per line, suitable for piping.

7.2 `netcfg validate`

Executes the pipeline in dry-run mode: all primitives run with `dry_run=true`, skipping DataFrame writes but verifying what *would* change. Optionally validates templates by performing a trial render to memory.

```
netcfg validate <topology> <blueprint> \
    [--templates-dir <DIR>] [--verbose]
```

Exit codes: 0 on success, non-zero on validation failure (parse errors, assertion failures, template syntax errors, IP pool conflicts).

7.3 `netcfg plan`

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

```
netcfg plan <topology> <blueprint> \
    [--diff] [--templates-dir <DIR>] [--verbose]
```

Output without `--diff`: colourised plan (green for additions, red for removals, yellow for modifications). Summary line: `+N additions, -N removals across M layers`.

Output with `--diff`: unified diff per device, preceded by `diff --cfg {hostname}`.

7.4 `netcfg inspect`

Provides three views into intermediate state:

```
netcfg inspect <path> --ast               # Print blueprint AST
netcfg inspect <path> --topology          # Export as Graphviz DOT
    [--blueprint <FILE>]
netcfg inspect <path> --trace <NODE_ID>   # Trace node state changes
    --blueprint <FILE>
```

- `--ast`: Parses the file as a blueprint and prints the AST tree.
- `--topology`: Exports the graph as Graphviz DOT. With `--blueprint`, applies the blueprint first. Output includes per-layer subgraph clustering, hostname labels, and subnet annotations.
- `--trace <node_id>`: Replays primitive execution and prints before/after `DeviceContext` snapshots for the specified node at each primitive step.

7.5 `netcfg ci-run`

The CI/CD integration point. Runs validate, plan, and config diff in a single invocation.

```
netcfg ci-run <topology> <blueprint> \
    --templates-dir <DIR> [--json]
```

With `--json`, emits a structured payload:

```
{
  "status": "success",
  "topology_additions": 42,
  "topology_removals": 0,
  "configuration_diffs": {
    "core-0": "-  ip address 10.0.0.4/30\n+  ip address 10.0.0.8/30"
  },
  "warnings": []
}
```

This integrates with GitHub Actions, GitLab CI, or any CI system that can parse JSON.

7.6 `netcfg import`

Imports a YAML topology definition into an NTE archive.

```
netcfg import <yaml_path> <output.nte>
```

The importer reads a YAML file in the topogen schema (defining nodes with types, layers, and properties) and serialises it to an `.nte` archive that other commands consume.

7.7 `netcfg lint`

Runs design-rule assertions without the full validation pipeline. Applies the blueprint in dry-run mode, then evaluates assertions and reports results.

```
netcfg lint <topology> <blueprint> [--verbose]
```

Returns assertion results with pass/fail status, severity, and diagnostic messages. Exit code 1 only when `error`-severity assertions fail; warnings are reported but do not cause failure.

7.8 `netcfg fmt`

Standardises blueprint YAML formatting. Parses the blueprint via `Executor::parse()`, re-serialises with `serde_yml`, and compares byte-for-byte. Idempotent.

```
netcfg fmt <blueprint> [--check] [--verbose]
```

With `–check`, exits with code 1 if formatting changes are needed (useful as a pre-commit hook or CI gate). Without `–check`, overwrites the file in place.

7.9 `netcfg impact`

Calculates the operational blast radius of a blueprint change against a base topology.

```
netcfg impact <topology> <blueprint> [--verbose]
```

For each mutation in the computed plan, assigns a risk level:

Risk	Colour	Trigger
High	Red	Node removal; critical property change (`asn`, `router_id`, `src_ip`, `subnet`); physical link add/remove
Medium	Yellow	Node addition; logical link add/remove
Low	Green	Metadata-only update

Critical properties are: `asn`, `router_id`, `src_ip`, `dst_ip`, `subnet`, `loopback`, `mgmt_ip`. Output includes per-node risk attribution with reasons.

7.10 `netcfg report`

Generates an automated Markdown architecture report from a compiled topology.

```
netcfg report <topology> <blueprint> -o <output.md> [--verbose]
```

The report contains three sections:

1. **Node inventory**: hostname, type, OS, ASN, loopback, management IP.
2. **Physical links & IP addressing**: source/destination hosts, interfaces, IPs, subnets.

3. **Topology visualisation**: A `netviz-json` code block with a custom JSON schema (`NetVizNode`, `NetVizEdge`, `NetVizTopology`) suitable for rendering by external visualisation tools.

7.11 `netcfg export-clab`

Generates a Containerlab `clab.yaml` simulation file from a compiled topology, enabling rapid lab deployment for testing (e.g. the three-site mesh from §8).

```
netcfg export-clab <topology> <blueprint> \
    -o <clab.yaml> [--configs_dir <DIR>] [--verbose]
```

Heuristically maps `device_os` to Containerlab container kinds: `cisco_iosxr` → `vr-xrv9k`, `nokia_srl` → `srl`, `arista_eos` → `ceos`, default → `linux`. When `–configs_dir` is provided, bind-mounts generated configuration files into OS-specific remote paths (e.g. `/etc/opt/srlinux/config.json` for SRL, `/mnt/flash/startup-config` for cEOS).

8 End-to-End Worked Example

We trace the `three-site-mesh.yaml` blueprint through every pipeline stage. The base topology has 12 routers across 3 sites (4 per site: site-a-r1 through site-c-r4).

8.1 Input: Blueprint

Listing 10. three-site-mesh.yaml (abridged).

```yaml
version: 1
layers:
  # Stage 1: Full mesh within each site
  - name: input
    primitives:
      - type: mesh_nodes
        selector: "nodes[site='a']"
        mesh_type: full
      - type: mesh_nodes
        selector: "nodes[site='b']"
        mesh_type: full
      - type: mesh_nodes
        selector: "nodes[site='c']"
        mesh_type: full

  # Stage 2: IP allocation per site
  - name: input
    primitives:
      - type: provision_ips
        selector: "edges[src>=0]"
        pool: "10.1.0.0/24"
        subnet_size: 30
        strategy: dense

  # Stage 3: BGP overlay
  - name: input
    primitives:
      - type: build_protocol_layer
        selector: "nodes[true]"
        layer: bgp
        config:
          protocol_type: bgp
          asn_base: "65000"
```

8.2 Stage 1: Parse and validate

The parser validates the YAML, confirms `version: 1`, and compiles each selector string (`nodes[site='a']`, `edges[src>=0]`, etc.) to a CEL program. The normaliser converts `site='a'` to `site=="a"`.

8.3 Stage 2: Shadow topology

The executor clones the base topology via a serialisation round-trip, creating an isolated shadow copy. All subsequent mutations operate on the shadow; the base topology is never modified.

8.4 Stage 3: Primitive execution

mesh_nodes (three invocations): Creates $3 \times \binom{4}{2} = 18$ edges (6 per site). The selector `nodes[site='a']` matches the 4 nodes with `{site: "a"}` in their `data_json`. Each node gets `mesh_type: "full"` and `peer_count: 3` written to its metadata.

provision_ips: The selector `edges[src>=0]` matches all 18 edges. Edges are sorted by (src, dst). The five-pass algorithm allocates /30 subnets from `10.1.0.0/24`:

- Edge (1,2): subnet `10.1.0.0/30`, IPs `10.1.0.1` and `10.1.0.2`
- Edge (1,3): subnet `10.1.0.4/30`, IPs `10.1.0.5` and `10.1.0.6`
- …continuing through all 18 edges.

After this stage, node `site-a-r1` has: `src_ip: "10.1.0.1", subnet: "10.1.0.0/30", vrf: "default"`.

build_protocol_layer: Clones all 12 physical nodes into the `bgp` layer (new IDs 13–24). Each clone receives:

- `protocol_type: "bgp"` and `asn_base: "65000"` (from the config overlay)
- `_source_id: 1` (parent mapping)
- All properties from the physical node, including the IPs written by `provision_ips` (via deep merge)

8.5 Stage 4: Mapping evaluation

The companion mapping (`three-site-mesh-mapping.yaml`):

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

The evaluator matches all nodes and emits one stanza per match. Each stanza carries `provenance: {rule_id: "rule_0"}`.

8.6 Stage 5: Render

Stanzas are grouped by `fields.node` (all mapped to `site-a-r1` in this example). The fallback template produces:

```
kind=bgp key=
```

8.7 Output: Atomic write

Once all five stages complete, three files are written atomically:

- `site-a-r1.conf`: the rendered configuration
- `site-a-r1.deviceir.json`: the raw stanza list
- `site-a-r1.lineage.json`: per-line provenance

9 Error Model

NetCfg uses typed error enums with `thiserror` derivation and `miette` diagnostic annotations. Table 26 summarises the error types.

Table 26. Error enums and their variants.

Enum	Variants	Context
`ParseError`	`YamlSyntax`, `Validation`, `Deserialize`, `LayerOrdering`	Blueprint parsing
`SelectorError`	`InvalidSyntax`, `Compile`, `Evaluate`, `NonBooleanResult`, `Polars`, `Json`, `TopologyAccess`	Selector compilation and evaluation
`PrimitiveError`	`Validation`, `Execute`, `Selector`, `Topology`, `Graph`, `Polars`, `Json`	Primitive execution
`ExecuteError`	`Parse`, `Primitive`, `Shadow`, `Plan`, `Assertion`	Top-level executor
`ShadowError`	`Clone`, `Commit`, `Validation`	Shadow topology lifecycle
`AssertionError`	`Failed`, `Selector`	Assertion evaluation
`DiffError`	`TopologyAccess`, `ParseError`	Configuration diffing
`RenderError`	`Io`, `Jinja`, `Json`, `MissingTemplate`	Template rendering

`ExecuteError` variants carry `miette::Diagnostic` annotations with diagnostic codes (e.g., `netcfg::parse`, `netcfg::primitive::execution_failed`, `netcfg::assertion`) enabling structured error handling in CI pipelines.

`ParseError::YamlSyntax` includes a source span (`miette::SourceSpan`) pointing at the offending line, enabling IDE-quality error rendering.

10 Editor and Service Integration

10.1 Language Server (LSP)

The `netcfg-lsp` crate provides a Language Server Protocol implementation built on `tower-lsp` and Tokio. It communicates via stdin/stdout and offers three capabilities:

1. **Real-time diagnostics.** On file open, change, and save, the server calls `parse_blueprint()` and maps errors to LSP diagnostics with line-column precision. For `serde_yml` errors (which embed location as free text), a regex extracts "at line N column M". Diagnostics are published with source label `ank_netcfg` and severity `ERROR`.
2. **Primitive auto-completion.** When the cursor follows a `type:` token (detected by a suffix heuristic), the server offers completion items for the six core primitives, each with a one-line docstring. Trigger characters are space and colon.

3. **Document synchronisation.** Open documents are tracked in a `DashMap<String, String>` (lock-free concurrent map). Full-text sync (`TextDocumentSyncKind::FULL`) is used for correctness. On close, the document is removed and diagnostics are cleared.

10.2 REST API Microservice

The `netcfg-api` crate exposes the compiler as a stateless HTTP service via `axum`, enabling CI/CD pipelines and enterprise orchestrators to compile blueprints without a local `netcfg` binary.

Endpoint: `POST /compile` (port 3000).

Table 27. Compile endpoint request and response schema.

Direction	Field	Type	Description
Request	`topology_zip_base64`	String	Base64-encoded `.nte` archive
Request	`blueprint_yaml`	String	Raw YAML blueprint
Response	`status`	String	`success` or `error`
Response	`configs`	Map<String, String>	Hostname → rendered config
Response	`device_ir`	Map<String, JSON>	Hostname → DeviceContext IR
Response	`warnings`	Vec<String>	Compilation warnings

The compilation pipeline decodes the base64 payload into a temporary file, loads the topology, parses the blueprint, executes via `Executor::apply()`, and extracts per-device IR. All processing is in-memory with no persistent state; the service scales horizontally behind a load balancer.

11 Limitations

11.1 `edge_properties` not applied (`mesh_nodes`)

Description. The `MeshNodesSpec.edge_properties` field is accepted by the blueprint parser but intentionally not applied. Applying it requires an edge DataFrame API on `nte_topology::Topology` equivalent to `set_dataframe`/`update_dataframe` but keyed on $(source\_id, target\_id)$ pairs.

Workaround. Edge properties can be encoded as node metadata on both endpoints (e.g., `link_bandwidth` as a per-node field keyed by peer ID).

Planned resolution. Awaiting `nte-topology` ≥0.2 edge DataFrame support. A tracking test (`edge_properties_deferred_until_nte_edge_dataframe`) is present in the test suite.

11.2 Mapping AST inspection not implemented

Description. The `inspect --ast` command can parse and display blueprint ASTs but does not yet support mapping documents. Attempting to inspect a mapping file prints "Mapping AST visualisation coming soon."

Workaround. Inspect the mapping YAML directly. The format is simple (a `rules` list with `selector` and `stanza` fields).

Planned resolution. When the mapping DSL grows beyond simple selector/stanza pairs, AST visualisation will be added.

11.3 Shadow validation is a placeholder

Description. The `ShadowTopology::validate()` method currently only checks that the shadow did not become empty. It does not perform structural integrity checks (dangling edges, orphaned protocol nodes, etc.).

Workaround. Design-rule assertions can catch many structural problems. Use `min_edges` and `field_exists` assertions as guardrails.

Planned resolution. Integrate with the NTE policy validation framework when available.

11.4 No intra-primitive rollback

Description. If a primitive fails partway through execution (e.g., `provision_ips` exhausts its pool after allocating half the edges), the shadow topology retains the partial mutations. The pipeline aborts (returning an error), and the shadow is discarded, but there is no mechanism to roll back individual primitives within a shadow.

Workaround. This is usually not a problem: the entire shadow is discarded on error, so no partial state reaches production. The limitation matters only for debugging: the error message may not reflect the full extent of partial mutations.

Planned resolution. No current plan. Intra-primitive rollback would require snapshotting the shadow before each primitive, which doubles memory usage.

11.5 `generate` CLI bypasses mapping document

Description. The `generate` CLI command renders configs directly from `DeviceContext` via templates, bypassing the mapping document. The full five-stage pipeline (with mapping evaluation and DeviceIR) is implemented in `config_gen::generate_configs()` but the CLI does not expose it yet.

Workaround. Use the `generate_configs()` function directly from Rust code for the full pipeline with mapping support.

Planned resolution. Add `–mapping` flag to the `generate` CLI command.

11.6 No incremental compilation

Description. Every invocation recompiles the entire topology from scratch. The diff engine can identify which devices changed, but the pipeline cannot skip unchanged devices during compilation.

Workaround. Use `ci-run –json` to compute the diff, then selectively push only the changed configuration files.

Planned resolution. Under investigation. Incremental compilation requires caching intermediate state (the desired topology) between runs and detecting which blueprint changes invalidate which nodes.

12 Future Work

- **GPU-accelerated primitive execution**: Offloading selector evaluation and IPAM allocation to the GPU via NTE's WebGPU compute backend, targeting 100K+ node topologies.
- **WASM linear memory I/O**: The current `wasm_plugin` primitive validates module compilation and symbol resolution but simulates the JSON exchange via linear memory. Full C-compatible memory allocation between the Rust host and WASM guest is required for production use.
- **API template rendering**: The `POST /compile` endpoint currently returns placeholder configuration text; integrating the MiniJinja rendering engine into the API service is planned.

- **Mapping CLI integration**: Exposing the full five-stage pipeline (including mapping evaluation) through the `generate` CLI command.
- **Incremental compilation**: Caching intermediate state to avoid recompiling unchanged portions of the topology.
- **LSP assertion integration**: Extending the language server to evaluate design-rule assertions in real time, providing inline warnings for selector violations and IP pool capacity.

13 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. Intentionet. Batfish: Network configuration analysis. https://www.batfish.org/, 2024.
6. Ivan Pepelnjak. netlab: Virtual networking lab framework. https://netlab.tools/, 2024.
7. Google. Common Expression Language. https://cel.dev/, 2024.
8. Simon Knight. NTE: A Hybrid Graph-Relational Engine for Network Topology Management. Technical Report, 2026.

A Blueprint YAML Schema Reference

Listing 11. Complete blueprint schema (annotated).

```yaml
version: 1                      # integer, >= 1
imports: [<path>, ...]          # optional, relative file paths to merge

# -- Named groups --
groups:
  <name>: <cel_selector>        # shorthand: just a selector string
  # OR full form:
  <name>:
    selector: <cel_selector>
    kind: security_zone         # optional
    trust_level: <i32>          # required when kind is security_zone

# -- Named rules (CEL macros) --
rules:
  <name>: <cel_expression>      # referenced via use_rule check

layers:
  - name: <string>              # non-empty, layer identifier
    requires: [<string>, ...]   # optional, previously-declared layers
    primitives:

      # --- Topology tier ---

      # --- mesh_nodes ---
      - type: mesh_nodes
        selector: <cel_selector>
        mesh_type: full | hub_and_spoke | route_reflector
        hub_selector: <cel_selector>      # required for h&s
        spoke_selector: <cel_selector>    # required for h&s
        edge_properties: <json_object>    # deferred
        naming_strategy: eth | cisco_ge | junos_ge

      # --- build_protocol_layer ---
      - type: build_protocol_layer
        selector: <cel_selector>
        layer: <string>                   # target layer
        config: <json_object>             # deep-merged into clone
        clone_underlying: <bool>          # default: false

      # --- Allocation tier ---

      # --- provision_ips ---
      - type: provision_ips
        selector: <cel_selector>
        pool: <cidr_or_pool_name>
        subnet_size: <u8>                 # default: 30 (v4), 126 (v6)
        ipv6_pool: <cidr_or_pool_name>    # optional dual-stack
        ipv6_subnet_size: <u8>
        strategy: dense | sparse | hash   # default: dense
        pool_size: <u32>                  # for global pool carving
        vrf: <string>                     # default: "default"

      # --- allocate_resources ---
      - type: allocate_resources
        selector: <cel_selector>
        resource_type: <string>
        pool: <range_or_pool_name>
        strategy: dense                   # default: dense
        pool_size: <u32>
```

```
# --- global_pool ---
- type: global_pool
  name: <string>
  pool: <cidr>
  vrf: <string>

# --- global_resource_pool ---
- type: global_resource_pool
  name: <string>
  resource_type: <string>
  pool: <range>

# --- Policy tier ---

# --- build_routing_policy ---
- type: build_routing_policy
  selector: <cel_selector>
  policy_name: <string>
  statements:
    - name: <string>               # sequence number
      action: permit | deny
      match_condition: <string>    # optional
      set_action: <string>         # optional

# --- build_access_policy ---
- type: build_access_policy
  selector: <cel_selector>
  policy_name: <string>
  rules:
    - name: <string>
      action: permit | deny
      source_prefix: <string>      # optional
      destination_prefix: <string> # optional
      protocol: <string>           # optional
      source_port: <string>        # optional
      destination_port: <string>   # optional

# --- build_prefix_list ---
- type: build_prefix_list
  selector: <cel_selector>
  prefix_list_name: <string>
  entries:
    - prefix: <cidr>
      action: permit | deny
      le: <integer>                # optional
      ge: <integer>                # optional

# --- build_community_list ---
- type: build_community_list
  selector: <cel_selector>
  community_list_name: <string>
  entries:
    - community: <string>          # e.g. "65001:1000"
      action: permit | deny

# --- generate_safe_bgp_filters ---
- type: generate_safe_bgp_filters
  selector: <cel_selector>
  prefix_list_name: <string>       # optional
  policy_name: <string>            # optional
  block_rfc1918: <bool>            # default: true
  permit_default: <bool>           # default: false
```

```
    # --- Overlay tier ---

    # --- build_vxlan ---
    - type: build_vxlan
      selector: <cel_selector>
      vni_base: <u32>
      mcast_group_base: <string>       # optional

    # --- build_evpn ---
    - type: build_evpn
      selector: <cel_selector>
      route_distinguisher_base: <string>
      route_target_base: <string>

    # --- Meta tier ---

    # --- tag_nodes ---
    - type: tag_nodes
      selector: <cel_selector>
      tags: {<key>: <json_value>}

    # --- map_hardware_inventory ---
    - type: map_hardware_inventory
      selector: <cel_selector>
      chassis_model: <string>
      slots: {<u32>: <linecard_model>}

    # --- assert_state (mid-pipeline checkpoint) ---
    - type: assert_state
      name: <string>
      severity: error | warning
      select: <cel_selector>
      check: <assertion_check>         # same types as top-level
      help: <string>                   # optional

    # --- conditional ---
    - type: conditional
      condition: <cel_expression>
      then_primitives: [<primitive>, ...]
      else_primitives: [<primitive>, ...]   # optional

    # --- custom_primitive ---
    - type: custom_primitive
      name: <string>
      parameters: {<string>: <json_value>}
      primitives: [<primitive>, ...]

    # --- inject_secrets ---
    - type: inject_secrets
      selector: <cel_selector>
      secrets: {<metadata_key>: <env_var_name>}

    # --- wasm_plugin (requires --features wasm) ---
    - type: wasm_plugin
      selector: <cel_selector>
      plugin_path: <filesystem_path>    # path to .wasm binary

assertions:
  - name: <string>
    severity: error | warning
    select: <cel_selector>
    help: <string>                      # optional remediation hint
```

```
    check:
      type: field_exists
      field: <field_name>
      # OR
      type: min_count
      count: <integer>
      # OR
      type: max_count
      count: <integer>
      # OR
      type: min_edges
      count: <integer>
      # OR
      type: unique_per_group
      field: <field_name>
      group_by: <field_name>
      # OR
      type: field_in_cidr
      field: <field_name>
      cidr: <cidr>
      # OR
      type: custom_cel
      expression: <cel_expression>
      # OR
      type: use_rule
      name: <rule_name>                  # references rules: section
      # OR
      type: is_connected                 # no additional fields
      # OR
      type: is_acyclic                   # no additional fields
      # OR
      type: is_bipartite                 # no additional fields
      # OR
      type: reachability
      target_selector: <cel_selector>
      # OR
      type: match_schema
      schema: <json_schema_object>

# -- Blast radius policies --
blast_radius:
  - name: <string>
    select: <cel_selector>               # optional, defaults to all
    check:
      type: max_deletions
      count: <integer>
      # OR
      type: max_modifications
      count: <integer>
      # OR
      type: max_percentage_changed
      value: <float>                     # 0.0 to 100.0

# -- Target-specific transforms --
transforms:
  - name: <string>
    when: <cel_expression>               # optional device predicate
    rules:
      - match_expr: <cel_expression>     # selects stanzas
        apply:
          <field>: <cel_expression>      # computes new field value

# -- Hardware library --
```

```
hardware_library:
  chassis:
    - model: <string>
      slots: <integer>
      description: <string>
  linecards:
    - model: <string>
      ports: <integer>
      speed: <string>
```

B Selector DSL Grammar

Listing 12. Selector DSL grammar (informal EBNF).

```
selector     ::= "all()" | target "[" filter "]"
target       ::= "nodes" | "edges"
filter       ::= cel_expression

(* Surface syntax normalisations applied before CEL compilation *)
"="          -> "=="
"and"        -> "&&"
"or"         -> "||"
"not"        -> "!"
"&"          -> "&&"
"|"          -> "||"

(* CEL variables bound per entity *)
nodes:  id (int), type (string), layer (string), + all data_json fields
edges:  id (string), src (int), dst (int), layer (string),
        + all data_json fields

(* IP prefix length extraction *)
For any field with an IP/CIDR value "10.0.0.0/30":
  {field}_len = 30  (integer, bound automatically)

(* Undeclared references *)
Accessing a field that does not exist on a node evaluates to false
(not an error), enabling heterogeneous topologies.
```

C CLI Quick Reference

Table 28. CLI command summary.

Command	Synopsis
`generate`	Full pipeline: parse → transform → render → write
`validate`	Dry-run pipeline with optional template pre-flight
`plan`	Compute and display topology mutation plan (or config diff)
`inspect`	AST visualisation, Graphviz DOT export, or per-node trace
`ci-run`	Combined validate + plan + config diff for CI/CD
`import`	Import YAML topology to NTE archive
`lint`	Run design-rule assertions only
`fmt`	Standardise blueprint YAML formatting
`impact`	Calculate blast radius of a blueprint change
`report`	Generate Markdown architecture report
`export-clab`	Generate Containerlab simulation file