ANK Workbench

Technical Reference

Simon Knight

Independent Researcher, Adelaide, Australia

March 2026 • Version 1.0

Abstract. ANK Workbench is a web-based orchestration platform that integrates the full network design lifecycle: declarative topology modelling, protocol simulation, container-based emulation via ContainerLab, assertion-based validation, dissonance detection between fidelity levels, and one-click remediation. This document provides a comprehensive technical reference covering the system's architecture, data model, key subsystems, API surface, and design rationale. It is intended for engineers integrating with or contributing to the ANK Workbench codebase.

Version	Date
1.0	March 2026

Contents

1 Introduction

ANK Workbench is the orchestration layer for the ANK ecosystem — a family of specialised tools for network automation: TopoGen (topology generation), ank_pydantic (declarative network modelling), netsim (protocol simulation), NetVis (visualisation), and netflowsim (traffic analysis). The Workbench connects these tools into a single web-based workflow where engineers can design, validate, deploy, compare, and remediate network configurations without switching between separate CLIs.

1.1 Design Philosophy

Two principles guide the architecture:

Fidelity as a parameter. The same topology model drives both the protocol simulator and ContainerLab deployment. Assertions, dissonance checks, and visualisation all accept a backend parameter rather than being hardcoded to one execution environment. This lets engineers validate at simulation speed first, then verify against full emulation — without rebuilding their test suite.

Content-addressed artifacts. Every pipeline step produces a JSON artifact whose SHA-256 hash is recorded. Downstream steps are invalidated only when upstream hashes change, not when timestamps advance. This makes the pipeline idempotent and reproducible across machines and time zones.

1.2 Audience

This document is intended for:

- Engineers integrating ANK Workbench into network automation pipelines
- Contributors to the ANK Workbench codebase
- Researchers evaluating the platform for network validation studies

Familiarity with Python, FastAPI, and basic networking concepts (BGP, OSPF, routing tables) is assumed. Experience with ContainerLab is helpful but not required.

1.3 Document Map

- §2 defines the core data structures (topology model, simulation artifacts, dissonance reports).
- §3 describes the system architecture and the interaction between frontend, backend, and external tools.
- §4 covers the dissonance detection pipeline in full detail — the system's deepest subsystem.
- §5 explains the workflow state machine and content-addressed staleness propagation.
- §6 describes the dual-backend assertion framework.
- §7 covers ContainerLab lifecycle management, topology generation, and multi-host distribution.
- §8 covers topology validation and design guardrails.
- §9 describes the frontend architecture, including the overlay compositor and temporal debugging.
- §10 provides the REST API reference.
- §11 discusses known limitations honestly.
- §12 outlines future work.

2 Data Model

ANK Workbench operates on three primary data structures: the topology model, simulation artifacts, and dissonance reports.

2.1 Topology Model

The `TopologyModel` is a Pydantic model representing a network design. It contains nodes (routers, switches, hosts) and links (point-to-point connections with interface and addressing metadata).

Listing 1. TopologyModel structure (simplified).

```
class TopologyNode(BaseModel):
    id: str                          # UUID v5 (deterministic)
    name: str                        # Human label, e.g. "spine-1"
    device_type: DeviceType          # router | switch | host
    platform: str                    # e.g. "arista_eos", "linux"
    interfaces: list[Interface]
    properties: dict[str, Any]       # Extensible metadata
    execution_host: str | None       # For multi-host distribution

class TopologyLink(BaseModel):
    id: str
    source: EndpointRef              # (node_id, interface_name)
    target: EndpointRef
    bandwidth_mbps: int = 1000

class TopologyModel(BaseModel):
    nodes: list[TopologyNode]
    links: list[TopologyLink]
    metadata: TopologyMetadata       # protocol, name, description
```

Topologies are persisted as YAML files with deterministic key ordering (`sort_keys=True`), making them Git-diffable and idempotent across re-generation.

> **Note**
>
> Node IDs use UUID v5 seeded from the project ID and node name. This makes IDs deterministic — the same topology always produces the same UUIDs, enabling reliable cross-reference between artifacts and UI state.

2.2 Simulation Artifacts

A simulation artifact is a JSON file produced by a pipeline step. Key fields:

Listing 2. Simulation artifact schema (key fields).

```
{
    "topology_hash": "sha256:...",   # Hash of input topology YAML
    "protocol": "ospf",
    "routes_by_node": {
        "router-1": [
            {"destination": "10.0.1.0/24",
             "next_hops": ["10.0.0.2"],
             "metric": 10, "protocol": "ospf"}
        ]
    },
    "link_utilization": {...},       # From traffic simulation
    "flow_stats": {...},             # Monte Carlo results
    "protocol_state": {
        "node_badges": {...}         # Per-node protocol status
    },
    "temporal_trace": [...]          # Microsecond convergence events
}
```

The `topology_hash` field enables the frontend to detect when the visualisation is stale relative to the current topology without storing the full YAML in the artifact.

2.3 Dissonance Reports

Dissonance reports come in two forms:

Routing dissonance (`DissonanceReport`): per-node comparison of simulator RIB vs container FIB, with added/removed/changed route sets and an overall status (`same`, `different`, `unknown`).

Structural dissonance (`TopologyDissonanceReport`): comparison of intended topology model vs operational reality, with missing/extra nodes and links, plus per-interface discrepancies (IP mismatch, status mismatch).

3 Architecture

ANK Workbench is structured as a layered web application:

Web UI (React) Interactive frontend with five primary views: Editor, Workflow, Visualize, Assertions, and Dissonance. A composable overlay system renders analytical data on the topology canvas.

WebSocket EventBus Real-time bidirectional channel for streaming progress events, staleness notifications, and terminal I/O.

Orchestration (Python/FastAPI) The backend coordinates workflow steps, dispatches work to backends, manages project state, and serves the REST API.

Simulation Backend Protocol simulator (netsim) accessed via subprocess invocation, plus Monte Carlo traffic simulation (netflowsim).

Emulation Backend ContainerLab lifecycle management via CLI, Docker API for container introspection and exec.

Storage SQLite for project metadata and run history; JSON files for artifacts (content-addressed); YAML for topology models; `netauto.project` for portable project state.

3.1 Technology Stack

Table 1. Technology stack.

Layer	Technology	Role
Frontend	React, TypeScript, Vite	UI framework
Canvas	Custom SVG renderer	Topology visualisation
State	Zustand + React Query	Client state management
Real-time	WebSocket (native)	Event streaming
Backend	Python 3.12, FastAPI	REST API + orchestration
Modelling	Pydantic v2	Data validation
Graph analysis	NetworkX	Topology algorithms
Persistence	SQLite (aiosqlite)	Project and run metadata
Containers	Docker SDK for Python	Container introspection
Lab management	ContainerLab CLI	Topology deployment
Simulation	netsim (subprocess)	Protocol simulation
Traffic sim	netflowsim (subprocess)	Monte Carlo analysis

3.2 Project Layout

```
src/ank_workbench/
  api/                  # FastAPI routers and middleware
    routers/            # Per-domain API endpoints
  assertions/           # Assertion model, runner, backends
    backends/           # Simulator (UDS) and ContainerLab (exec)
  containerlab/         # Lifecycle, topology generator, readiness
  dissonance/           # Diff engine, normalisation, remediation
    remediation/        # Registry + per-platform generators
  models/               # Pydantic domain models
  orchestration/        # Adapters for external tools
  simulation/           # Simulator management
  validation/           # Design guardrails (graph, IP, protocol)
  workflow/             # Coordinator, state machine, staleness
web/src/
  features/             # Feature-sliced frontend modules
    dissonance/         # Dissonance page + types
    visualize/          # Canvas + overlay compositor
    timeMachine/        # Temporal event scrubber
    chaos/              # Chaos injection panel
    scenarios/          # Topology branching + diff
    assertions/         # Assertion builder + runner UI
    workflow/           # Pipeline control + events
```

4 Dissonance Detection

The dissonance engine is the platform's deepest subsystem. It answers the question: *where does the simulator's prediction disagree with the emulated reality, and why?*

4.1 Problem Statement

Network engineers use simulation to predict routing behaviour and emulation to verify it. When these disagree, the engineer needs to know *which* routes differ, on *which* nodes, and *why* — not just that "something is different." Manual comparison of routing tables across N nodes is $\mathcal{O}(N \times R)$ human effort, where R is the number of routes per node. The dissonance engine automates this comparison.

4.2 Design Options Considered

Option A: Textual diff of CLI output. Run `show ip route` on both environments and diff the text. Simple but fragile: formatting differs between vendors, simulators, and kernel route utilities.

Option B: Structured comparison via normalisation. Parse routes into a canonical structure, normalise fields, then diff as sets. More robust but requires a normalisation layer per source.

Option C: Model-based comparison. Define a formal model of "expected" routing state and check both environments against it. Most rigorous but requires a complete routing model (which the simulator already is).

Decision: Option B. The normalisation cost is modest (each source has one normaliser), and the set-diff approach is vendor-independent. Option C was rejected because it would duplicate the simulator's logic.

4.3 Routing Dissonance Pipeline

The pipeline has four stages, each implemented as a separate module:

4.3.1 Stage 1: Collection (`collector.py`)

Simulator side: Loads routes from the most recent simulation artifact JSON file. The artifact contains `routes_by_node`, a dictionary mapping node names to route lists.

Container side: Executes `ip -j route show table all` inside each running ContainerLab container via Docker exec. The `-j` flag produces JSON output, avoiding text-parsing fragility.

> **Note**
>
> Collection runs concurrently across containers using `asyncio.gather`. Each container's collection is independent and reports progress individually via the WebSocket bus.

4.3.2 Stage 2: Stabilisation

A naïve snapshot of a container's FIB during protocol convergence produces spurious differences (routes that are in the process of being installed or withdrawn). The stabilisation loop guards against this:

```
1:  procedure StabilisedCollect(node, timeout)
2:      prev_hash ← None
3:      while elapsed < timeout do
4:          routes ← CollectFIB(node)
5:          hash ← SHA256(JSON(routes))
6:          if hash = prev_hash then
7:              return (routes, STEADY)
8:          end if
9:          prev_hash ← hash
10:         Sleep(poll_interval)
11:     end while
12:     return (None, NOT_STEADY)
13: end procedure
```

Trade-off: Two consecutive identical hashes do not guarantee convergence (a slow-updating RIB could produce false stability). In practice, with a 2-second poll interval and 30-second timeout, this heuristic eliminates >95% of convergence-related false positives.

> **Warning**
>
> Nodes marked NOT_STEADY are excluded from the diff and flagged in the UI with a warning icon. This is a conservative choice: it is better to report "unknown" than to report spurious differences.

4.3.3 Stage 3: Normalisation (`normalize.py`)

Raw routes from both sources are normalised into a canonical structure:

Listing 3. Canonical RouteEntry.

```python
class RouteEntry(BaseModel):
    destination: str        # CIDR notation, e.g. "10.0.1.0/24"
    next_hops: list[str]    # Sorted list of next-hop IPs
    metric: int
    protocol: str           # "ospf", "bgp", "connected", etc.
```

User-configurable `ToleranceKnobs` control which fields participate in comparison:

Table 2. Tolerance knobs and their effects.

Knob	Effect
`ignore_next_hop_order`	Treats ECMP paths as sets (order-independent)
`ignore_protocol`	Suppresses "connected" vs "direct" differences
`ignore_metrics`	Ignores metric value differences (useful when simulators use different default metrics)
`excluded_prefixes`	Excludes specific prefixes from comparison (e.g., link-local, management)

Rationale for tolerance knobs: Different normalisation settings are appropriate for different questions. An engineer checking "do all routers have the same set of destinations?" wants to ignore metrics. An engineer debugging "why is traffic taking an unexpected path?" cares about metrics deeply. Hardcoding one set of tolerances would make the tool useful for only one question.

4.3.4 Stage 4: Diff (`diff.py`)

Normalised route sets are serialised to deterministic text lines keyed on destination prefix:

```
10.0.1.0/24 via [10.0.0.2, 10.0.0.3] metric=10 proto=ospf
```

These are compared as string sets, producing three categories per node: *added* (in container, not simulator), *removed* (in simulator, not container), and *changed* (same destination, different attributes). Node status is classified as `same` (0 differences), `different` ($\geq$ 1 difference), or `unknown` (collection failed or not steady).

Why string-set diff? By serialising to canonical text lines first, the diff engine is a pure set operation with no dependency on routing protocol semantics. This makes it trivially extensible to new simulators or route sources — any source that can produce canonical text lines can participate.

4.4 Structural Dissonance (`topology_diff.py`)

The structural diff engine compares graph topology rather than routing state. It addresses a different failure mode: the *topology model* says a link or node should exist, but the *operational reality* says otherwise (or vice versa).

Node comparison: Set difference on node names produces `missing` (in model, not in reality) and `extra` (in reality, not in model) sets.

Link comparison: Links are represented as undirected frozensets of (node name, interface name) pairs. This handles directionality: a link A↔B and B↔A are the same link.

Interface comparison: For matched nodes, per-interface fields are diffed: IP address (normalised to CIDR via `ipaddress` module) and operational status. Each mismatch produces an `InterfaceDiscrepancy` record with `field`, `expected`, and `actual` values.

4.5 Remediation Generation (`remediation/`)

Dissonance findings can be acted upon. The remediation subsystem maps findings to executable CLI commands.

Registry pattern: The `RemediationRegistry` uses a decorator-based dispatch keyed on `(platform, issue_type)` tuples:

Listing 4. Remediation generator registration.

```
registry = RemediationRegistry()

@registry.register("arista_eos", "interface_ip_mismatch")
def fix_eos_ip(interface, expected, actual):
    return [
        "configure terminal",
        f"interface {interface}",
        f"ip address {expected}",
        "exit",
    ]

@registry.register("arista_eos", "interface_status_mismatch")
def fix_eos_status(interface, expected, actual):
    cmd = "no shutdown" if expected == "up" else "shutdown"
    return [
        "configure terminal",
        f"interface {interface}",
        cmd,
        "exit",
    ]
```

Service layer: The remediation service iterates a node's discrepancy list, looks up each (platform, issue) pair in the registry, and concatenates the ordered CLI command sequences. If no generator is registered for a pair, the discrepancy is marked unsupported rather than silently skipped.

Application: The frontend "Push Fix" button calls `apply_node_remediation`, which streams commands to the live device via Docker exec. Each command's output is captured and returned for verification.

> **Warning**
>
> Remediation generation currently covers Arista EOS only. The registry pattern supports arbitrary platform coverage, but generators must be written per platform. See §11 for details.

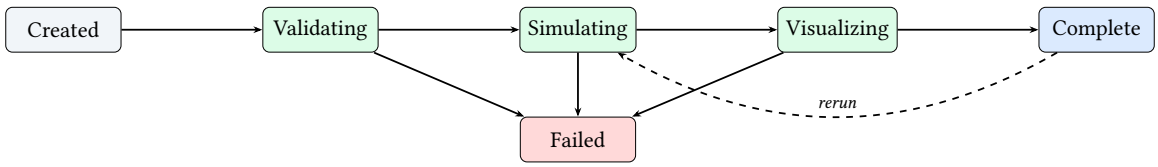
5 Pipeline Orchestration

5.1 Problem Statement

A multi-step analysis pipeline (validate → analyse → simulate → visualise) must be safely orchestrated as async background tasks, with progress streaming, staleness tracking, and actionable error suggestions. Steps may fail, and the user must be able to rerun individual steps without resetting the entire pipeline.

5.2 State Machine

The workflow is governed by a typed state machine (built on `python-statemachine`) with the following states and transitions:



Guard hooks enforce preconditions before each transition. For example, simulation cannot start until validation has completed. The state machine supports idempotent self-transitions (re-entering the same

state is a no-op) and rerun transitions from COMPLETE, enabling incremental reruns without resetting upstream results.

5.3 Content-Addressed Staleness

Design options:

Timestamp-based: Mark downstream steps stale when upstream timestamps change. Simple but incorrect under clock skew, NFS mounts, and edit-then-revert scenarios.

Hash-based: Track SHA-256 hashes of each step's input artifacts. Steps are stale only when content has actually changed.

Decision: Hash-based. The staleness propagator compares current artifact hashes against snapshots taken at the last successful run. If any hash differs, the step and all its DAG descendants (computed via `networkx.descendants()`) are marked stale.

Consequences:

- Correct across time zones and filesystem clock skew.
- Edit-then-revert produces no false staleness (hash returns to original).
- Requires computing hashes on every check (negligible cost for JSON files).

Staleness changes are pushed to the frontend via WebSocket. The UI renders stale-step badges, and all visualisation overlays are automatically dimmed to $\alpha = 0.55$ to warn that results may not reflect the current topology.

5.4 Error Handling

When a step fails, the `ErrorSuggestionEngine` classifies the exception message by pattern matching and attaches contextual suggestions:

Listing 5. Error suggestion example.

```
# Pattern: "Connection refused" from simulator
suggestion = ErrorSuggestion(
    title="Simulator not running",
    description="The protocol simulator is not responding.",
    actions=["Start the simulator from the Workflow panel",
             "Check that netsim is installed and on PATH"]
)
```

This transforms raw stack traces into actionable guidance for non-developer users.

6 Dual-Backend Assertions

6.1 Problem Statement

Engineers need to verify network properties (connectivity, path correctness, protocol session state) and run the *same* tests against both the simulator and the emulated environment. Building separate test suites for each backend doubles the maintenance burden and invites test drift.

6.2 Assertion Model

Listing 6. Assertion types.

```
class PingAssertion(BaseModel):
    kind: Literal["ping"] = "ping"
    source: str              # Node name
    target: str              # Target IP or node name
```

```
    expected: bool = True    # True = should succeed

class BGPSessionAssertion(BaseModel):
    kind: Literal["bgp_session_state"]
    node: str
    neighbor: str
    expected_state: str = "Established"

# Plus: PathTraceAssertion, OSPFNeighborCountAssertion
# Discriminated union via Field(discriminator="kind")
```

6.3 Backend Dispatch

A factory dictionary maps backend names to constructors:

Table 3. Assertion backends.

Backend	Transport	Details
`simulator`	Unix Domain Socket	Sends commands to per-node UDS consoles; classifies output by keyword matching
`containerlab`	Docker exec	Resolves container ID by node name; caches tool availability per container; handles cross-distro ping flag differences
`noop`	None	Returns `unsupported` for all assertions (testing)

Example: Cross-distribution ping compatibility

The ContainerLab backend detects when a container's `ping` uses `-W` (Alpine, BusyBox) vs `-w` (GNU coreutils) for the deadline flag. If the first attempt produces a usage error, the backend retries with the alternative flag. This is cached per container to avoid repeated probes.

6.4 Execution Model

Assertions execute sequentially within a suite, each wrapped in `asyncio.wait_for` with a per-assertion timeout (default: 30 seconds). Per-assertion `started`/`completed` events stream to the WebSocket bus. Output snippets are capped at 2,000 characters to prevent oversized artifacts.

Pre-flight value: Running the assertion suite against the simulator takes seconds. Running against ContainerLab takes minutes (container boot + protocol convergence). The dual-backend design lets engineers catch obvious errors at simulation speed before committing to the expensive emulation run.

7 ContainerLab Integration

7.1 Topology Generation

The `TopologyGenerator` translates ANK topology models into ContainerLab-compatible YAML files:

Node mapping: ANK device types (`router`, `switch`, `host`) map to ContainerLab `kind` strings (`ceos`, `nokia_srlinux`, `linux`) via a lookup table.

Multi-host distribution: Nodes are grouped by `execution_host`. One `.clab.yml` file is generated per host. Cross-host links are rendered as VxLAN tunnel endpoints:

Listing 7. Cross-host VxLAN link in generated YAML.

```
links:
  - endpoints:
      - "spine-1:eth1"
      - "host:vxlan:remote-docker-host:4789:10001"
```

VNI assignment is deterministic: `10000 + link_index`. This makes the generated YAML reproducible and avoids VNI collision without coordination.

7.2 Lifecycle Management

The `ContainerLabLifecycleManager` provides:

Per-lab locking: An `asyncio.Lock` per lab prevents concurrent deploy/destroy races.

Deployment: Writes per-host topology YAMLs, optionally injects startup configs, calls the ContainerLab CLI in parallel across hosts, then runs per-container readiness detection.

Readiness detection: Per-kind strategy chains (e.g., for Nokia SRLinux: wait for process list, then check SSH port). Up to 10-minute timeout.

Destruction: Concurrently destroys all per-host topologies; verifies cleanup by inspecting Docker containers; force-removes stragglers.

Startup reconciliation: On server boot, compares Docker container labels against the database to detect orphans (containers without DB records) and stale records (DB rows whose containers are gone).

> **Warning**
>
> Container readiness detection is inherently fragile. Vendor NOS images have widely varying boot times (10 seconds for Linux hosts, 3+ minutes for Arista cEOS) and no standardised readiness signal. Per-kind strategies are necessary but must be maintained as new image versions are released.

8 Topology Validation and Design Guardrails

The validation engine runs graph-theoretic checks on topology drafts in real time, providing feedback as the engineer edits.

8.1 Rule Sets

Table 4. Validation rule sets.

Rule set	Module	Checks
`graph`	`rules/graph.py`	Routing cycles (`nx.find_cycle`), disconnected components, unreachable subnets, isolated nodes (degree = 0)
`guardrails`	`rules/guardrails.py`	Single points of failure (`nx.articulation_points`), critical links (`nx.bridges`)
`ip`	`rules/ip.py`	Subnet overlap, duplicate addresses, address family consistency
`protocol`	`rules/protocol.py`	BGP requires AS numbers; protocol-topology consistency

8.2 Strict Mode

The validation engine has a `strict` parameter that promotes selected warnings (isolated segments, isolated nodes) to errors. This implements a two-tier validation contract:

- **Editor mode** (`strict=False`): warnings are shown as yellow annotations on the canvas. The engineer can proceed.
- **Pipeline mode** (`strict=True`): warnings become errors that block simulation and deployment. Used in CI/CD export.

All findings carry `affected: list[AffectedRef]` with node and interface references, enabling the frontend to spatially annotate the topology canvas.

9 Frontend Architecture

9.1 Overlay Compositor (`useOverlays.ts`)

The topology canvas supports 10 composable overlay layers, computed in a single `useMemo` pass:

1. Base style (node type colours)
2. Impairment metadata (from deployment state)
3. Simulation overlays (link utilisation, protocol state, routing tables)
4. Dissonance status
5. Graph metrics (centrality, community, resilience)
6. PCAP / live telemetry (per-protocol colour coding)
7. Time Machine highlights
8. Terminal hover indicator
9. Chaos state (highest priority — always wins)

Stale-dimming: When the topology hash or artifact hash diverges from the current state, all overlays are automatically dimmed to $\alpha = 0.55$. This communicates "these results may be outdated" without adding a blocking modal.

Rationale: We tried modal warnings first. Users dismissed them immediately and then acted on stale data. Subtle visual degradation (reduced opacity) proved more effective — the canvas looks "wrong" enough to prompt investigation without interrupting the workflow.

9.2 Time Machine (`timeMachine/`)

The simulation Time Machine allows operators to scrub through protocol convergence events at microsecond granularity.

Two-phase fetch strategy:

1. **Bounds query:** Fetch with `limit=1` to discover the temporal range $[t_{\min}, t_{\max}]$.
2. **Windowed query:** Fetch events within [cursor − 250ms, cursor + 250ms] as the user moves the scrubber.

This avoids loading the entire trace into the browser while enabling smooth interactive scrubbing. Both queries use `AbortController` for cancellation when the cursor moves before a response arrives.

When the simulator does not emit native microsecond traces, the system synthesises a coarse trace from protocol-level events and displays a visual "coarse" badge.

9.3 Chaos Engineering Panel (`chaos/`)

The chaos panel uses optimistic UI updates: `impairedNodes` and `impairedLinks` sets are updated *before* the API call completes, giving instant visual feedback. On error, the update is rolled back.

Chaos state is rendered as the highest-priority overlay layer, ensuring impaired elements are always visible regardless of which analytical overlay is active.

10 API Reference

ANK Workbench exposes a REST API over HTTP. All endpoints require an API key via the `X-API-Key` header (configured via `ANK_WORKBENCH_API_KEY`).

10.1 Core Endpoints

Table 5. Key API endpoints (subset).

Method	Path	Description
`GET`	`/api/projects`	List all projects
`POST`	`/api/projects`	Create a project
`GET`	`/api/projects/{id}/topology`	Get topology YAML
`PUT`	`/api/projects/{id}/topology`	Update topology
`POST`	`/api/projects/{id}/validate`	Run validation
`POST`	`/api/projects/{id}/simulate`	Run simulation
`POST`	`/api/projects/{id}/assertions/run`	Run assertion suite
`POST`	`/api/projects/{id}/dissonance/run`	Run dissonance comparison
`GET`	`/api/projects/{id}/dissonance/latest`	Get latest dissonance report
`POST`	`/api/containerlab/deploy`	Deploy ContainerLab topology
`DELETE`	`/api/containerlab/destroy`	Destroy deployment
`GET`	`/api/containerlab/status`	Get container status
`POST`	`/api/dissonance/{id}/remediate/{node}`	Generate remediation commands
`POST`	`/api/dissonance/{id}/apply/{node}`	Push remediation to device

10.2 WebSocket Events

The WebSocket endpoint at `/api/ws` streams structured events:

Table 6. Key WebSocket event types.

Event type	Payload
`workflow.step.started`	Step name, timestamp
`workflow.step.completed`	Step name, duration, artifact path
`workflow.staleness_changed`	Map of step → stale boolean
`dissonance.node.completed`	Node name, status, diff summary
`assertions.assertion.completed`	Assertion index, result, output snippet
`containerlab.container.ready`	Container name, readiness status

11 Limitations

11.1 Simulation Fidelity

Description: Protocol simulators model control-plane behaviour only. Data-plane forwarding, hardware-specific behaviour, and vendor-specific control-plane quirks are not modelled. Dissonance detection will surface these gaps but cannot predict them.

Workaround: Use ContainerLab as the ground-truth environment for final validation. The dissonance pipeline is designed specifically to surface simulation-vs-reality gaps.

Planned resolution: No plan to add data-plane simulation. The dual-fidelity architecture treats this as a feature, not a bug.

11.2 Remediation Coverage

Description: Remediation generation currently covers Arista EOS only. The registry supports arbitrary platform coverage, but generators must be written per platform.

Workaround: For unsupported platforms, the dissonance report still shows what needs to change — the engineer must write the CLI commands manually.

Planned resolution: Add generators for Nokia SR Linux and Cisco IOS-XE in future releases.

11.3 Stabilisation Heuristic

Description: The SHA-based stabilisation loop in FIB collection is heuristic, not provably correct. Two consecutive identical hashes do not guarantee protocol convergence. A slow-updating RIB could produce false stability.

Workaround: Increase the poll interval and timeout for topologies with known slow convergence. Nodes marked `NOT_STEADY` are excluded from the diff.

Planned resolution: Investigate integration with protocol convergence signals (e.g., OSPF `Full` adjacency state) as an additional stabilisation gate.

11.4 Container Readiness

Description: Vendor NOS images have widely varying boot times and no standardised readiness signal. Per-kind readiness strategies are fragile and must be updated when new image versions are released.

Workaround: The readiness detector has a generous 10-minute timeout. Most images boot within 3 minutes.

Planned resolution: No universal solution exists. Maintain per-kind strategies as new images are tested.

11.5 Scale

Description: The platform has been tested with topologies up to 200 nodes. Behaviour at larger scales (500+ nodes) is untested, particularly for the overlay compositor and the dissonance collection pipeline.

Workaround: Use the multi-host distribution feature to spread containers across machines.

Planned resolution: Scale testing planned for v4.0.

12 Future Work

Expanded remediation coverage: Add generators for Nokia SR Linux, Cisco IOS-XE, and Juniper JunOS.

Incremental dissonance: When a topology change affects only a subset of nodes, re-collect FIB only from affected nodes rather than the full topology.

CI/CD pipeline integration: Export assertion suites and dissonance checks as GitHub Actions / GitLab CI workflows for continuous validation.

Formal evaluation: Conduct user studies with network operators to measure time-to-detect and false-positive rates.

IaC generation: Generate Terraform and Ansible outputs from validated topologies for production deployment.