BLUEDOT

ENGINEERING REFERENCE

# Touch Operational API

## Theory of Operations — Operational Control Interface

*QuantumTouch presents the operational plane of the BlueDot BNG through a single authenticated REST interface. Where the BNG Telemetry API publishes raw subscriber and traffic state, the Touch Operational API publishes the operation of the node itself — composed health, retained time series, event lifecycle, a merged audit timeline, configuration snapshots, and identity administration. This document describes that interface for engineers automating the BNG or integrating it with an existing NMS or OSS/BSS.*

| | |
|---|---|
| **Document Type** | BlueDot Theory of Operations |
| **Component** | QuantumTouch — BNG Operational REST API |
| **Audience** | Integrators — Network Automation / NMS / OSS-BSS |

| | |
|---|---|
| **Source Tree** | `ipoe/quantumtouch-bng` @ v0.9.11 ( `quantumtouch-core` 0.9.0) |
| **Interface** | Northbound REST over HTTPS, base path `/api/bng` |
| **Companion** | *BNG Telemetry API Theory of Operations* |
| **Classification** | **BlueDot Proprietary** |
| **Date** | July 2026 |

# Contents

# Purpose and Framing

The BlueDot BNG exposes two northbound interfaces, and the distinction between them is the organising idea of this document.

The first is the **BNG Telemetry API** — the controller's own REST surface, described in the companion *BNG Telemetry API Theory of Operations*. It answers questions about **subscribers and traffic**: who is connected, what plan they hold, how many octets they passed in each QoS colour, how full the address pools are. It is deliberately raw. It stores no history, computes no trends, and returns cumulative counters and current state for a consumer to interpret.

The second is the **Touch Operational API**, the subject of this document. It is served by QuantumTouch, BlueDot's operational layer for the BNG, and it answers a different class of question: **is this node healthy, what changed, who changed it, what was it like before, and what will it be like if I change it again.** It is the interface an automation engineer uses to *operate* a BNG — not to read subscriber counters, but to poll composed health, retain and query time series, manage an event lifecycle with acknowledgement, read a merged audit timeline across four independent sources, capture and restore configuration snapshots, and administer the credentials that gate all of it.

Three properties define the interface and are worth stating before any endpoint is described.

**It is the same interface the QuantumTouch web UI uses.** The QuantumTouch SPA is an ordinary client of the endpoints in this document. It holds no privileged channel, no private RPC, and no internal-only feed. Every dashboard, chart, event console, audit view, and snapshot dialog in the product is assembled from the calls documented here. The practical consequence is parity: anything the operator can see or do through the UI, a script can see or do through the API, with identical fidelity and identical authorisation.

**It composes rather than merely proxies.** The controller reports whether VPP is connected; QuantumTouch reports that *together with* the reply-path stall detector, the RADIUS reachability verdict, the BMC sensor state, the uplink gateway probe, and the pool-pressure signal, reduced to a single verdict a monitoring system can alarm on. The controller emits an audit line; QuantumTouch merges it with the systemd journal, its own event log, and the captive-portal access log into one normalised, source-tagged timeline. This composition is the value the operational plane adds over the raw plane, and it is deliberately exposed through the API rather than trapped in the UI.

**It retains what the controller does not.** The BNG controller is a window onto the present. QuantumTouch runs a sampler that writes to a local ring buffer and a Prometheus time-series database, retains an append-only event log with rotation and archival, keeps a sticky event history that survives condition recovery, and stores versioned configuration snapshots with content hashes. History, in this system, lives in the operational plane.

**A note on the two planes together.** An integration does not have to choose. QuantumTouch fronts the controller on the same TLS listener, under the same credential, through an explicit passthrough described in *The Controller Passthrough* below. A single API key against a single host and port reaches both the operational endpoints in this document and every raw telemetry endpoint in the companion document. That is the intended integration posture and it is the shortest path from a bare box to a populated NMS.

# Architecture and the Origins of the Data

Understanding where each value comes from tells an integrator what it can and cannot mean, how fresh it is, and what happens to it when a dependency fails.

## Core and plugin

QuantumTouch is composed of two Python packages. `quantumtouch-core` is a Flask application framework, not a helper library. It owns the application factory, the entire authentication and authorisation stack (API keys, sessions, CSRF, rate limiting, account lockout, TLS and certificate lifecycle), the settings store, the Prometheus exposition endpoint, the interactive API browser, and the single-page-application static serve. `quantumtouch-bng` is a plugin that supplies BNG-specific routes, settings validation, and metric content. The plugin contains no authorisation logic of its own; it declares the role each route requires and core enforces it.

The plugin registers one blueprint containing every route module. Core mounts that blueprint at the base path `/api/bng`. Every path in this document, except the controller passthrough, is that prefix plus the route's own rule.

Registering the blueprint is not a passive act — it starts three background subsystems that produce much of what the API later returns: the metrics sampler thread, the authentication and platform gauge collectors, and, when destinations are configured, the audit shipper. All three are idempotent and all three are suppressed under test.

## Five origins

Values returned by this API come from five places, and the origin determines the failure mode.

**The BNG controller**, over plaintext loopback at `http://localhost:3000`. Health, counters, session state, and the entire configuration surface originate here. Timeouts are short and deliberate — two seconds for the hot read paths, four to fifteen seconds for provisioning and reset operations. When the controller is unreachable, QuantumTouch does not fail silently or hang; it returns `502` with a machine-readable error code and, on `/health`, still returns the portion of the picture it can compute itself.

**The VPP data plane**, through `vppctl`. Version strings, interface counters, port addressing, and the uplink gateway probe come from here. The uplink probe is worth calling out: because VPP owns the data plane, a kernel `ping` from the web tier would exit through the out-of-band management NIC and prove nothing about the subscriber path. QuantumTouch therefore asks VPP itself to ping the gateway and parses the result.

**The host and the platform** — `/proc/net/dev` for veth throughput, `/sys/bus/pci/devices` and `ethtool` / `devlink` / `modinfo` for NIC and driver inventory, `os.statvfs` for disk occupancy, `journalctl` for service logs, and `ipmitool` for BMC sensors, power supplies, and fans.

**QuantumTouch's own persistent state**, under the data directory (by default `/opt/bng-deploy/data`): the append-only event log and its archives, the sticky event history, acknowledgement records, configuration snapshots, the settings document, session records, and the audit shipper's spools and offsets.

**The time-series layer** — a Prometheus instance scraping QuantumTouch's own `/metrics` endpoint, backed by an in-process ring buffer that continues to fill regardless of Prometheus's state. This dual structure, and the rules governing which one answers a given query, is described in *Time Series and Analytics*.

## The sampler

A single daemon thread ticks every **five seconds** and is the heartbeat of the operational plane. On each tick it assembles one unified sample containing aggregate QoS rates, session-state counts, the reply-path state, controller health, controller counters, and per-port counters. That sample is then handed to every registered metric emitter in turn.

Two design rules govern the sampler and both are visible in the API's behaviour.

**An emitter failure never kills the tick.** Each emitter is invoked inside its own exception guard, so a fault in one metric family cannot stop the others from publishing.

**A failed tick holds gauges rather than zeroing them.** If the controller is unreachable for a tick, gauges retain their last known value and counters simply do not increment. Rendered over time this produces a *flat line*, which reads correctly as "sampling paused". Zeroing would produce a cliff, which reads incorrectly as "traffic stopped". Integrators building alarms on these series should treat a flat line as a possible collection outage and cross-check `up{job="bng"}` or `GET /api/bng/health`.

Counters present a second subtlety. The controller exposes absolute cumulative values; the Prometheus counter type only accepts increments. Each emitter therefore computes a delta against the previous observation. A **negative delta is never emitted as a negative value** — it is recognised as a controller restart, the baseline is silently reset, and a dedicated counter, `bng_metric_baseline_reset_total{family=…}`, is incremented so the reset is visible in monitoring. The first observation of any series seeds the baseline and emits nothing.

## The reply-path stall detector

This is the most BNG-specific health concept in the system and it exists nowhere in the raw telemetry plane. The sampler compares the access port's receive rate (CPE toward BNG) against its transmit rate and tracks **consecutive** ticks in which the emit ratio falls below threshold while the receive rate is above a load floor. Any healthy tick resets the counter. The requirement for consecutive samples is what prevents a single slow tick from flapping an alarm.

The resulting state — `rx_pps`, `tx_pps`, `ratio`, `stall_samples` — is published on `GET /api/bng/health` and as the `bng_reply_path_*` metric family. It detects a class of fault that neither the controller nor a simple interface counter can see: subscriber traffic arriving and being accepted, but replies not making it back out.

# Access Model and Conventions

A small number of conventions apply across the entire interface. They are stated once here and not repeated per endpoint.

## Transport

QuantumTouch is served over **HTTPS only**, bound by default to `0.0.0.0:8443`. There is no plain-HTTP listener and therefore no HTTP-to-HTTPS redirect — a request to port 80 is refused at the socket, not redirected.

Three certificate sources are supported. The default is a **local certificate authority**, self-signed at install with an 825-day validity and subject alternative names covering the LAN address, loopback, `localhost`, and the host's name. A **customer-supplied certificate and key** may be imported, in which case automatic renewal is disabled. **ACME** issuance is supported natively for deployments with a publicly resolvable name.

For an integrating script the practical guidance is: copy the CA certificate off the box once and pin it. Disabling verification is not the recommended posture for a management-plane credential.

Certificate renewal is a hot reload — the worker reloads on certificate change without a restart, so a long-lived integration does not observe a connection outage at renewal.

HTTP Strict Transport Security is **off by default**, because a local-CA trust relationship is fragile and an HSTS pin against a certificate the operator may later replace is a support hazard.

## Credentials

Two credential types converge on the same request-local identity before any route sees them.

**API keys** are the automation path. A key is presented in one place and one place only:

```
Authorization: Bearer quantum_<32 hex characters>
```

There is no `X-API-Key` header and no query-parameter form. Keys are stored as unsalted SHA-256 digests; the raw key material is returned exactly once, at creation, and is never recoverable afterwards. Each key carries an identifier, a role, an optional label, and three optional restrictions described below.

**Session cookies** are the browser path, used by the QuantumTouch UI. A successful login sets two cookies: `qt_sess`, which is `HttpOnly`, `Secure`, and `SameSite=Lax`, carrying a signed session identifier; and `qt_csrf`, deliberately readable by JavaScript so the SPA can echo it back.

The two paths differ in exactly four ways:

| | Session cookie | Bearer API key |
|---|---|---|
| CSRF token required | **Yes**, on all mutating verbs | **No** — a header cannot be planted by a cookie attack |
| Identity kind | `session` | `automation` |
| Controller passthrough | Injects the session's controller bearer | Preserves the caller's own header |
| IP allowlist / path scope | Not applicable | Enforced |

Beyond those, a route handler cannot tell the two apart and does not need to. **An integrating script uses a bearer key and never needs a CSRF token.**

## Roles

Authorisation is a three-tier monotonic hierarchy:

| Role | Rank | Grants |
|---|---|---|
| `read_only` | 0 | All read and telemetry endpoints |
| `read_write` | 1 | Everything `read_only` grants, plus event acknowledgement and lifecycle actions |
| `admin_only` | 2 | Everything above, plus key administration, snapshots, audit shipping, and provisioning |

A higher rank satisfies a lower requirement. An unrecognised role held fails closed; an unrecognised role required also fails closed. There is a fourth, out-of-hierarchy service role, `metrics_scraper`, used exclusively by the Prometheus scrape credential and always paired with a path scope restricting it to `/metrics`.

The default is deny. A route with no explicit declaration requires `read_only`; opening a route to anonymous access requires an explicit and greppable decorator, and almost nothing carries one. In particular, **`GET /api/bng/health` is authenticated** — it is a composed operational verdict, not a public liveness probe. So is `/metrics`.

Roughly, the surface divides as: the entire read and telemetry surface at `read_only`; event acknowledgement and clearing at `read_write`; and API key administration, configuration snapshots, audit-shipping destinations, and the setup wizard at `admin_only`.

## Key restrictions

Three optional restrictions can be attached to any key at creation, and using them is the difference between a credential and a scoped credential. All three are absent by default, and absent means unrestricted.

`ip_allowlist` — a list of CIDRs checked against the caller's source address. A request from outside the list is rejected with `403`.

`path_scope` — a list of glob patterns checked against the request path. A request outside the scope is rejected with `403`. This is how the Prometheus scrape credential is confined to `/metrics` and nothing else.

`expires_at` — an epoch timestamp after which the key is rejected with `401`.

The recommended posture for an NMS integration is a `read_only` key, scoped by IP to the polling host, with no expiry (so a silent expiry cannot break collection unnoticed), and a separate `admin_only` key held out of automation for snapshot and key operations.

## Error semantics

The governing rule is: **401 means "I do not know who you are"; 403 means "I know who you are, and no."**

| Condition | Status | Body |
|---|---|---|
| No credential, malformed, or revoked | `401` | — |
| Key expired | `401` | `{"error": "key expired"}` |
| Session expired | `401` | `{"error": "session expired"}` |
| Bad password or locked account | `401` | `{"error": "invalid credentials"}` |
| Insufficient role | `403` | `{"error": "insufficient role", "required": "admin_only", "have": "read_write"}` |
| Source IP not allowlisted | `403` | `{"error": "key not authorized from this IP"}` |
| Path outside key scope | `403` | `{"error": "key not authorized for this endpoint"}` |
| CSRF missing or mismatched (cookie path only) | `403` | — |
| Rate limit exceeded | `429` | `{"error": "rate_limit_exceeded", "group": …, "retry_after": …}` plus a `Retry-After` header |
| Controller unreachable | `502` | `{"error": {"code": "CONTROLLER_UNREACHABLE", "message": …}}` |
| Controller returned a non-JSON body | varies | `{"error": {"code": "BAD_UPSTREAM", "message": …}}` |

The direct client guidance follows from the table. **Retry a `401` only after re-minting a credential — never in a loop. Never retry a `403`;** it is a policy decision and will not change on repetition. **Honour `Retry-After` on `429`.** Treat `502` as a transient dependency failure and back off.

Note that a locked account and a wrong password return an identical `401` with an identical body. The distinction is recorded in the audit log but is deliberately not disclosed on the wire, because disclosing it enumerates valid usernames.

## Rate limiting and lockout

Two independent protections operate at different granularities.

**Per-source-IP rate limiting** is keyed on the route group and the caller's address: five login attempts per five minutes, thirty system calls per minute, sixty self-service calls per minute, and one hundred per minute by default. A single operator setting halves every limit. Keys may be marked rate-limit exempt — the Prometheus scrape credential is.

**Per-username account lockout** is separate: ten failures within a fifteen-minute window triggers a thirty-minute lock; a successful authentication clears the record entirely.

The asymmetry is intentional. The per-IP limit returns an informative `429` because it is keyed on the caller's own address and disclosing it is safe. The per-username lockout returns an opaque `401` because it is keyed on someone else's identity and disclosing it is not.

## Value encoding

Several conventions govern how values appear on the wire and misreading them is the most common integration error.

**Time-series sample values are strings, not numbers.** Both time-series endpoints return points as a two-element array `[timestamp, value]` where the timestamp is a float epoch second rounded to one decimal place and the value is a **decimal string** formatted to four places. This holds even for integer-valued series — a session count arrives as `"42.0000"`. This is the Prometheus wire convention and it is preserved deliberately so query code remains portable. A client must parse the second element.

**Timestamps are epoch seconds as floats** throughout the operational plane — in events, audit entries, snapshots, and time-series ranges. The one exception is the systemd journal reader, which passes through the journal's native **microsecond** timestamps as strings on `GET /api/bng/audit/journal`. The normalised audit timeline converts these to float seconds; the raw journal endpoint does not.

**Counters that originate as unsigned 64-bit integers may be serialised as decimal strings** where they cross from the controller, so that byte counts remain exact beyond the $2^{53}$ safe-integer limit of JavaScript numbers.

**Absent data is an empty string or `null`, consistently within an endpoint but not across them.** The driver inventory uses `""` for every unreadable field so a client never has to null-check; the health and event surfaces use `null`. Read the per-endpoint notes.

## Polling cadence

The interface is **pull only**. QuantumTouch does not push, stream, or open a webhook toward a monitoring system. (Outbound audit shipping, described later, is a separate and explicitly configured mechanism, and it ships audit records — not telemetry.)

Endpoints carry short server-side caches so that several UI tabs and a polling NMS share one upstream request rather than multiplying load on the controller and the data plane. The cache is process-local and reset on restart. Only successful responses are cached, so a transient upstream error is never replayed for a full cache lifetime.

| Endpoint | Server cache | Suggested client interval |
|---|---|---|
| `/health` | 1 s | 5–15 s |
| `/counters` | 1 s | 2–10 s |
| `/counters/{ip}` | none | on demand |
| `/access` | 30 s | 60 s |
| `/about` | 30 s | 300 s or on change |
| `/drivers` | 60 s | 300 s or on demand |
| `/hardware` | 10 s (shared IPMI cache) | 30–60 s |
| `/dataplane-stats` | stateful — see below | 5 s, single collector |
| `/throughput/recent`, `/metrics/query_range` | none needed | 5–30 s |
| `/events` | 30 s (uplink probe), 10 s (IPMI) | 5–15 s |
| `/audit/timeline` | none | 10–60 s, with cursor |

Two endpoints deserve specific care.

`GET /api/bng/dataplane-stats` **is stateful, not a gauge read.** It reports the delta since *the previous request to that endpoint*, divided by elapsed wall time. Three consequences follow. The first call after a process restart returns zero rates, because there is no prior sample. Calls less than half a second apart return zero rates, by a divide-by-tiny guard. And **two independent clients polling concurrently steal each other's window** — each sees only the traffic since the other's call. Designate a single collector for this endpoint, or use the Prometheus interface counters instead.

`GET /api/bng/events` **has write side effects.** Synthesising the live event set reconciles it against the persisted history, which updates the history file and appends lifecycle transitions to the audit log. It is not a safe or idempotent `GET` in the HTTP sense. This is by design — the event lifecycle advances on observation — but it means the endpoint should be polled by one collector at a sane interval, not fanned out.

## Subprocess-backed endpoints

Several endpoints shell out to platform tooling and are correspondingly slow. Client timeouts should account for this: `/drivers` may take one to two seconds on a dense chassis and issues up to seven distinct tool invocations; `/hardware` and `/hardware/test` invoke `ipmitool` with timeouts up to ten seconds; `/about` invokes `vppctl` and `uname`; `/audit/journal` invokes `journalctl` with a fifteen-second ceiling and returns `504` if it is exceeded.

# The Controller Passthrough

The single most important endpoint for an integrator building a complete NMS feed is not in the `/api/bng` namespace at all.

```
GET|POST|PUT|PATCH|DELETE /v1/<path>
```

This is a transparent reverse proxy onto the BNG controller's own REST surface — the interface described in the companion *BNG Telemetry API Theory of Operations*. It is mounted at the application root rather than under `/api/bng`, because it must mirror the controller's native `/v1` namespace exactly.

**What this means in practice.** A single API key against `https://<box>:8443` reaches both planes:

```
https://<box>:8443/api/bng/health              → operational health (composed)
https://<box>:8443/v1/bng/counters             → raw QoS band counters
https://<box>:8443/v1/bng/sessions             → raw session inventory
https://<box>:8443/v1/bng/sessions/{ip}        → full subscriber record
https://<box>:8443/v1/bng/pools                → address pool occupancy
https://<box>:8443/v1/bng/plans                → service rate tiers
```

The integrator does not need a second credential, a second port, a second TLS trust relationship, or network reachability to the controller's loopback listener. The passthrough is the bridge between the two documents.

**Forwarding semantics.** The query string and the raw request body are forwarded unchanged. Hop-by-hop headers are stripped in both directions, so an upstream `Connection: close` cannot prematurely terminate the downstream connection. Redirects are **not** followed — a `3xx` from the controller is returned to the caller as a `3xx`. The response is fully buffered rather than streamed. The timeout is twenty seconds.

**Credential handling.** If the caller presents an `Authorization` header, it is preserved verbatim and passed to the controller, which applies its own role check. If the caller is a browser session with no `Authorization` header, QuantumTouch injects the session's per-session controller bearer, which causes the controller to attribute the request to that operator rather than to an anonymous caller. **For a script the first path applies: your key travels through and the controller sees it.**

**Failure.** If the controller is unreachable the proxy returns `502` with `{"ok": false, "error": "controller unreachable: …"}`.

A defence-in-depth note worth understanding: authorisation is applied twice on this path. QuantumTouch gates the proxy route, and the controller independently evaluates the bearer against its own role model. A permission that one layer would grant and the other would not is denied.

# System State and Health

## Composed health

```
GET /api/bng/health                                    read_only
```

This is the endpoint an NMS polls for up/down and capacity alarming. It merges the controller's own health document with QuantumTouch's reply-path state, which the controller cannot compute because it does not observe the wire.


```
{
  "status": "ok",
  "vpp":    { "connected": true },
  "radius": { "auth": { "reachable": true }, "acct": { "reachable": true } },
  "sessions": {
    "connected": 8102, "captive": 274, "offering": 45, "total": 8421,
    "onboard_refused": 0, "max": 100000, "at_cap": false
  },
  "pools": {
    "subs-pool": { "utilization": 0.5140, "alloc_failures": "0", "pressure": "ok" }
  },
  "persistence": {
    "save_ms":            { "last": 6, "max": 41 },
    "write_lock_wait_ms": { "last": 0, "max": 12 }
  },
  "reply_path": { "rx_pps": 41822.55, "tx_pps": 41790.10, "ratio": 0.9992, "stall_samples": 0 }
}
```


The `status` field rolls up to `ok`, `degraded`, or `fault`. The rule is deliberate: `fault` when VPP is disconnected or **both** RADIUS endpoints are unreachable; `degraded` when exactly one RADIUS endpoint is unreachable; and **pool pressure is reported but does not gate the verdict** — a deliberately full pool is an operational condition to act on, not a service outage.

The `sessions` block carries the capacity-edge signals. `onboard_refused` is a monotonic count of onboards the data plane rejected, which climbs once a capped box is full. `max` and `at_cap` report the compiled session cap and whether the node is currently refusing new sessions; when VPP is down the cap is unknowable and **both fields are omitted rather than guessed**. The `persistence` gauges expose the controller's own write-path timing and are the early-warning signal for the database-bound write ceiling that ultimately bounds sessions per node.

`reply_path` is QuantumTouch's contribution and is described in *Architecture* above. It is present on every response.

**Failure behaviour is the point of this endpoint.** If the controller is unreachable, the call returns `502` — but the body is still a well-formed health document, with `status: "fault"`, zeroed session counts, an `error` object carrying `CONTROLLER_UNREACHABLE`, **and the live `reply_path` block**. A monitoring system therefore always receives a parseable verdict rather than a connection error, and can distinguish "the BNG is down" from "the management plane is down".

## Component versions

```
GET /api/bng/about                                    read_only
```

Assembles a component version matrix from five heterogeneous sources — the installed Python packages, `vppctl show version`, `uname -r`, the controller's own reported software version, and the host operating system.

```
{
  "bng":        { "version": "0.9.11" },
  "core":       { "version": "0.9.0" },
  "vpp":        { "version": "vpp v24.06-release built by …" },
  "controller": { "version": "1.2.0" },
  "kernel":     { "version": "6.8.0-45-generic" },
  "host":       { "hostname": "bng-edge-01", "distro": "Ubuntu 24.04.1 LTS", "arch": "x86_64" }
}
```

Every field degrades to the literal string `"unknown"` on any failure and the call **always returns 200**. A box with no VPP running, or an unreachable controller, still renders a complete version matrix with the fields it could determine. Note that `vpp.version` is the full multi-line output of `show version`, not a bare semantic version.

This is the correct endpoint for inventory reconciliation and for gating an automation script on a minimum platform version.

## Host throughput and disk

```
GET /api/bng/dataplane-stats                          read_only
```

Reports per-interface throughput on the kernel-side veth peers of the BNG data path, plus root filesystem occupancy. Only interfaces matching the BNG naming convention are included; management traffic is deliberately excluded.

```
{
  "ok": true,
  "interfaces": [
    { "name": "bng-sub-host", "rx_bps": 118203441.2, "tx_bps": 9922011.7,
      "rx_bytes_total": 44012993881, "tx_bytes_total": 3910228841,
      "link_speed_gbps": 10.0 }
  ],
  "rx_bps": 118203441.2,
  "tx_bps": 9922011.7,
  "link_speed_gbps": 10.0,
  "disk_total_gb": 456.7,
  "disk_used_gb": 123.4,
  "disk_pct": 27.0
}
```


Two units notes. Despite their names, `rx_bps` and `tx_bps` are **bytes per second**, not bits — they derive directly from the kernel byte columns. Multiply by eight for a bit rate. And `link_speed_gbps` **defaults to 10.0** when the kernel reports no negotiated speed, which is always the case for a veth; it is a denominator for a utilisation gauge, not a measured property.

`rx_bytes_total` and `tx_bytes_total` are monotonic kernel-lifetime counters and are the values to prefer if you are computing rates yourself. Deltas are floored at zero, so a counter reset reads as zero rather than as a negative rate.

Recall the statefulness warning in *Access Model*: the `_bps` fields are computed against this endpoint's own previous invocation, and concurrent clients interfere. Use `rx_bytes_total` / `tx_bytes_total` with your own two-sample arithmetic if more than one consumer needs this data.

## Chassis, power, and thermal

```
GET  /api/bng/hardware                              read_only
POST /api/bng/hardware/test                         read_only
```

`GET /hardware` returns BMC-derived platform state via IPMI. If IPMI has not been configured, the response is simply `{"configured": false}` and nothing further is attempted. When configured:

```
{
  "configured": true,
  "psus": [
    { "name": "PSU1", "status": "ok", "watts": 412, "vin_v": 230.0,
      "iout_a": 1.8, "fan_rpm": 8400, "temp_c": 38,
      "metrics": [ { "key": "Vin", "value": 230.0, "unit": "Volts" } ] }
  ],
  "fans":  [ { "name": "FAN1", "rpm": 8600, "status": "ok", "unit": "RPM" } ],
  "temps": { "CPU0": 47, "CPU1": 46, "Inlet": 22 },
  "alerts": [
    { "id": "sensor:FAN2", "kind": "sensor_status", "severity": "critical",
      "sensor": "FAN2", "value": 0.0, "unit": "RPM",
      "summary": "FAN2 reported lcr",
      "detail": "ipmitool flagged FAN2 as lcr (value 0.0 RPM)." }
  ]
}
```

Three things make this more useful than a raw `ipmitool` scrape. **Power supplies are regrouped.** IPMI presents each PSU as several independent sensor rows ( `PSU1_Vin` , `PSU1_Iout` , `PSU1_Pout` , `PSU1_FAN` ); QuantumTouch regroups them into one record per supply with correctly-united readings and convenience scalars. **Firmware heterogeneity is absorbed.** Modern BMCs emit a ten-column sensor table; older ones emit a three-column variant with the unit embedded in the sensor name. Both are parsed. **Alerts track the BMC's own thresholds** rather than hardcoded limits — each reading is cross-checked against the threshold columns the BMC itself reports, so alerting follows the platform's configuration.

`POST /hardware/test` is a one-shot connectivity probe using operator-supplied credentials, used by the settings UI's "Test connection" control. It accepts `bmc_host` , `username` , `password` , and `interface` (one of `lanplus` , `lan` , or `open` ), and it **always returns HTTP 200**, discriminating on an `ok` boolean and a `reason` string ( `invalid_request` , `ipmitool_missing` , `timeout` , `ipmitool_failed` ). This is deliberate so a UI can render a specific message rather than a generic network error. Passwords are redacted from any error text before it is returned.

The `open` interface addresses the local BMC in-band via `/dev/ipmi0` and needs neither host nor credentials; `lan` and `lanplus` are remote and require both.

## NIC and driver inventory

```
GET /api/bng/drivers                                    read_only
```

Enumerates every Ethernet-class PCI device on the chassis, groups them by bound kernel driver, and reports firmware, DDP, and addressing state. This is the endpoint that answers "what NICs are in this box, what firmware are they running, and which ones has VPP taken over".

```
{
  "drivers": [
    {
      "name": "ice",
      "module_version": "1.13.7",
      "description": "Intel(R) Ethernet Connection E800 Series Linux Driver",
      "supported_pci_ids": ["8086:1593"],
      "nics": [
        {
          "pci": "0000:13:00.0", "driver": "ice", "vendor_device": "8086:1593",
          "vendor_name": "Intel Corporation Ethernet Controller E810-C",
          "ifname": "eno1np0", "mac": "3c:ec:ef:aa:bb:cc", "link_state": "up",
          "firmware_version": "3.81 0x8000685b 1.3643.0",
          "ddp_track_id": "0xc0000001", "ddp_version": "1.3.30.0",
          "nvm_version": "3.81", "serial": "3CECEFAABBCC",
          "binding_note": "", "vpp_name": "",
          "ip_addrs": ["100.64.0.1/20"]
        }
      ]
    }
  ]
}
```


Notable semantics. **A NIC with no bound driver is grouped under the literal key `"unbound"`.** **`supported_pci_ids` lists the device IDs actually present on this chassis**, not the driver's full support table. **DDP and serial fields are only populated for the Intel `ice`, `i40e`, and `iavf` drivers**, which are the ones exposing them through `devlink`; every other driver reports empty strings.

The most operationally significant field is `binding_note`. When a NIC is bound to a userspace driver — `vfio-pci`, `uio_pci_generic`, or `igb_uio` — the kernel tools can no longer read its firmware or DDP state, and the note explains exactly that. For these NICs the MAC address and link state are backfilled from VPP's own port inventory instead, so the record is still complete.

`ip_addrs` merges kernel-side and VPP-side addressing, deduplicated. Sub-interface addresses roll up to their physical parent and are annotated with the owning sub-interface, so an entry may read `"100.64.255.2/24 (FortyGigabitEthernet47/0/0.200)"` rather than a bare CIDR. Link-local and loopback addresses are excluded.

Every field degrades to `""` rather than `null` or absence, so a client never has to null-check. The endpoint always returns 200.

## Raw counter and access passthroughs

```
GET /api/bng/counters                                read_only
GET /api/bng/counters/{ip}                           read_only
GET /api/bng/access                                  read_only
```

These are thin proxies onto the corresponding controller endpoints, provided so that a UI or script already talking to QuantumTouch does not need to change base paths for the most frequently polled reads. The response bodies are the controller's own and are documented in the companion *BNG Telemetry API* — the node-wide QoS band aggregate, the per-subscriber band breakdown, and the enabled access interfaces respectively.

The node-wide `/counters` read is cached for one second so that multiple dashboard tabs share a single upstream request. The **per-IP read is deliberately uncached**, because per-subscriber polling happens in a single drilldown view where a stale rate would be misleading. `/access` is cached for thirty seconds.

For any controller endpoint not in this short list, use the `/v1/*` passthrough.

# Time Series and Analytics

Retention is the operational plane's distinctive contribution. The controller returns current state and cumulative totals; QuantumTouch retains history and serves it back as queryable series.

## The dual-store architecture

Two independent stores hold time series, and the relationship between them is a deliberate design rather than a migration artefact.

**The in-process ring buffer** is filled by the sampler on every five-second tick. It holds two resolutions: a fast bucket at five-second spacing covering one hour, and a slow bucket at five-minute spacing covering thirty days. Per-subscriber series live in the fast bucket only, with four hours of retention. The buffer is periodically persisted so a process restart does not lose the recent window.

**Prometheus** runs as a container bound to loopback, scraping QuantumTouch's own `/metrics` endpoint every fifteen seconds, with thirty days or five gigabytes of retention, whichever binds first. Every series carries two external labels identifying the plane and the box.

The rule governing the two is precise and worth internalising:

> **Prometheus is authoritative for history. The controller live path is authoritative for "right now". The ring buffer is the standby for both, and the permanent owner of per-subscriber series.**

Queries route through a facade that reads Prometheus first and falls back to the ring buffer. The fallback tree distinguishes *environmental* failure from *our* failure:

| Condition | Behaviour |
|---|---|
| Prometheus not scraping us | Serve from sampler, `reason: "prom_down"` |
| Prometheus returned empty, sampler has data | Serve from sampler, `reason: "prom_empty"` |
| Prometheus unreachable or timed out | Serve from sampler, `reason` names the fault |
| Prometheus returned a query error | **Raise** `502` — do not fall back |

The last row is the important one. A malformed query or a Prometheus 5xx is a defect on our side, and silently serving older numbers from the fallback store would mask it. Degradation falls back; bugs surface.

Which store answered is **plumbed to the client, not hidden** — responses carry a `source` field. The UI renders a badge distinguishing historical from recent data and a banner when it is running degraded. An integrating script should record `source` alongside the data if provenance matters to it.

**Cardinality is bounded by construction.** Per-subscriber series are architecturally excluded from Prometheus; they live only in the ring buffer and on the controller's on-demand endpoints. The total series count on a BNG is around sixty-five, roughly five megabytes at thirty days. This bounds storage, bounds query cost, and bounds the blast radius of an over-broad query or a leaked scrape credential — there is no per-subscriber data in the TSDB to expose.

## Aggregate throughput

```
GET /api/bng/throughput/recent?range=<r>&step=<s>      read_only
```

Returns all eight aggregate series in one payload, so a multi-series chart gets consistent timestamps across every line without correlating separate responses.

| Parameter | Type | Default | Rules |
|---|---|---|---|
| `range` | enum | `1h` | One of `5m`, `15m`, `1h`, `6h`, `24h`, `1w`, `1m`. Anything else returns `400`. |
| `step` | integer seconds | computed | Floored at 5 s. A non-integer value silently falls back to the default. |

`1m` **means one month (thirty days), not one minute.** There is no one-minute range. This is the single most common misreading of the interface.

The default step targets a sensible point count: roughly 720 points for ranges up to an hour, and roughly 240 beyond, so a thirty-day query does not return a half-million points.

```
{
  "ok": true,
  "range": { "start": 1753000000.0, "end": 1753003600.0, "step": 5 },
  "series": {
    "agg_up_green_bps":  [[1753000000.0, "8122340.0000"], [1753000005.0, "8140021.0000"]],
    "agg_up_yellow_bps": [],
    "agg_up_red_bps":    [],
    "agg_dn_green_bps":  [],
    "agg_dn_yellow_bps": [],
    "agg_dn_red_bps":    [],
    "agg_up_pps":        [],
    "agg_dn_pps":        []
  }
}
```

**All eight keys are always present**; a series with no data is an empty array, never absent. Values are bits per second for the `_bps` series and packets per second for the `_pps` series. Remember that sample values are strings.

A sample is emitted for a target timestamp only if the stored sample is within two steps of it. A gap wider than that produces **missing points rather than a flat line**, so a collection outage is visibly a gap in the data.

The single `400` is `{"ok": false, "error": "unknown range"}`.

## Prometheus-shaped query

```
GET /api/bng/metrics/query_range?metric=&range=&step=&ip=   read_only
```

A Prometheus-compatible range query over a curated metric set. The response envelope deliberately mimics Prometheus's `/api/v1/query_range` so that query and charting code remains portable.

| Parameter | Rules |
|---|---|
| `metric` | One of `sessions_connected`, `sessions_captive`, `sessions_offering`, `subs_bps_up`, `subs_bps_dn`. Otherwise `400`. |
| `range` | Same enum as `/throughput/recent`. Otherwise `400`. |
| `step` | Integer seconds, floored per range (5 s up to `1h`, rising to 600 s for `1m`). |
| `ip` | **Required** when `metric` begins with `subs_bps_`; ignored otherwise. Missing gives `400`. |

```
{
  "status": "success",
  "data": {
    "resultType": "matrix",
    "result": [
      { "metric": { "__name__": "subs_bps_dn", "ip": "100.64.12.30" },
        "values": [[1753000000.0, "2411882.0000"]] }
    ],
    "range": { "start": 1753000000.0, "end": 1753003600.0, "step": 5 }
  }
}
```

`result` is an empty array when the series does not exist — an unknown subscriber IP, or a series the sampler has not populated yet. `data.range` is a QuantumTouch extension; real Prometheus does not return it.

Per-range step floors match the sampler's own cadence so that chart lag is bounded by the five-second tick rather than by query-side aggregation.

Errors are `400` with `{"status": "error", "error": …}` and one of `"unknown metric"`, `"unknown range"`, or `"ip required for per-subscriber metric"`.

Note the retention asymmetry for per-subscriber queries: those series live only in the fast ring buffer, so a long range on `subs_bps_*` is bounded by that buffer's window regardless of the range requested.

## Prometheus exposition

```
GET /metrics                                    metrics_scraper scope
```

Standard Prometheus text exposition, served on the **same HTTPS listener** as the rest of the API — not a sidecar and not a separate port. The handler performs no polling; it renders values the sampler has already set on its own tick, so a scrape costs nothing beyond serialisation.

**It is authenticated.** The scrape credential is an ordinary API key from the same store, subject to the same middleware, the same audit trail, and the same revocation control. That was chosen specifically so that revoking a scrape credential works exactly like revoking any other, rather than requiring a second authentication surface. The credential is constrained to the `metrics_scraper` role with a path scope of `/metrics`, carries no expiry, and is exempt from rate limiting.

Successful scrapes deliberately **do not** emit a key-usage audit event — at a fifteen-second interval that would be nearly six thousand events per box per day and would drown real signal. *Failed* scrapes do emit, and increment a failure counter, so a revoked or rotated credential surfaces on the security health indicator within one scrape interval.

The published families:

| Metric | Type | Labels |
|---|---|---|
| `bng_qos_bytes_total` | Counter | `direction={up,down}`, `band={green,yellow,red}` |
| `bng_qos_packets_total` | Counter | `direction`, `band` |
| `bng_throughput_bps` | Gauge | `direction`, `band` |
| `bng_throughput_pps` | Gauge | `direction` |
| `bng_sessions_active` | Gauge | `state={connected,captive,offering}` |
| `bng_reply_path_stalled` | Gauge | — |
| `bng_reply_path_rx_pps`, `bng_reply_path_tx_pps` | Gauge | — |
| `bng_radius_reachable` | Gauge | `server={auth,acct}` |
| `bng_uplink_bytes_total`, `bng_uplink_packets_total` | Counter | `port`, `direction={rx,tx}` |
| `bng_uplink_drops_total` | Counter | `port`, `direction`, `reason={miss,error}` |
| `bng_access_bytes_total`, `bng_access_drops_total` | Counter | `port`, `direction` [, `reason` ] |
| `bng_metric_baseline_reset_total` | Counter | `family` |

Alongside these, the platform publishes an authentication and security family: active sessions, users by role, disabled and MFA-enrolled user counts, API keys by kind, expired and near-expiry key counts, locked accounts, certificate days remaining, webhook queue depth, and audit-shipping spool and liveness gauges — plus counters for login success and failure by reason, session expiry by cause, ACME renewal outcomes, webhook deliveries, and audit-shipping throughput.

Two conventions in that family are worth knowing. `auth_cert_days_remaining` returns a sentinel of `-9999` when no certificate is installed, so a dashboard can distinguish "missing" from "expires today". And `auth_api_keys_total` emits a synthetic zero-valued series on a fresh box so the metric family always exists and recording rules do not break against an absent series.

These collectors re-read on-disk state fresh on every scrape — there is no staleness window and no cache.

## The versioned analytics surface

A separate, explicitly versioned path exists for customer analytics integrations:

```
/api/bng/analytics/v1/*                                admin_only
```

This is the **only versioned path prefix in the system**, and the version marks exactly where the customer-facing compatibility boundary sits. It is backed by the same query facade as the internal charts, rate-limited per key, and returns a uniform envelope carrying the series, the resolved range, the answering `source`, and an `as_of` anchor. Step floors are enforced per range and a request for a finer step than the floor is rejected with `400` — that is the cost guard.

A **PromQL passthrough** exists alongside it, forwarding verbatim to the Prometheus HTTP API. It is feature-flagged and **off by default**, requires an API key rather than a browser session (so a third-party page cannot issue queries through a live operator session), and requires `admin_only` plus a dedicated key scope. When the flag is off it returns `404`**, not** `403`, so an unauthenticated probe cannot confirm the feature exists.

The two surfaces have deliberately different failure contracts. If Prometheus is down, the curated endpoint falls back to the sampler and reports `source: "sampler"`. The PromQL passthrough returns `502` and does **not** fall back — the caller asked for the raw thing, and a substitute would be a lie.

# Events

The event subsystem exists because a raw sensor state is not an operational signal. A fan that dropped below threshold at 03:00 and recovered at 03:04 is invisible to any endpoint that reports current state, and an operator who stepped away misses the entire incident. QuantumTouch therefore models events in three cleanly separated layers.

## Synthesis

The first layer is **pure and stateless**: it takes one snapshot of system state and returns a list of event dictionaries. Nothing is stored, nothing is remembered.

Three sources feed it. **IPMI** contributes power-supply faults and sensor threshold violations, tagged `source: "ipmi"`. **Service reachability** contributes the controller, VPP, RADIUS, and uplink-gateway conditions, tagged `source: "service"`. **Captive portal health** contributes portal outage and recovery, tagged `source: "captive_portal"`.

Event identifiers are stable, namespaced strings — this is what makes acknowledgement and correlation possible:

```
service:controller_unreachable
service:vpp_disconnected
service:radius_auth_unreachable
service:radius_acct_unreachable
service:uplink_gateway_unreachable
psu:PSU1
sensor:FAN2
captive_portal:<portal-ip>:<epoch-of-outage-start>
```

Two synthesis policies matter to an integrator.

**Severity is graded by impact, not by source.** `radius_auth_unreachable` is `critical`, because new subscribers cannot authenticate. `radius_acct_unreachable` is `warning` — accounting is lost, which is a billing problem, but authentication and forwarding continue.

**Standalone deployments do not alarm.** RADIUS events fire only when RADIUS servers are actually configured. A deliberately RADIUS-less deployment would otherwise alarm continuously, because a reachability check against nothing returns unreachable.

If the controller cannot be reached at all, synthesis short-circuits: it emits `service:controller_unreachable` and stops, because nothing further can be truthfully said about the node.

## Sticky lifecycle

The second layer is **stateful and persistent**, and it is what turns a sensor reading into an alarm console.

A persistent map records, per event identifier, when the condition was first seen, when it was last active, whether and when it resolved, its current lifecycle state, and a snapshot of the most recent live event payload. Reconciliation against a fresh synthesis is a pure function — it computes the new map and writes nothing itself.

The transitions:

| Situation | Result |
|---|---|
| Identifier not seen before | Create as `active`, stamp `first_seen_ts` |
| Still present | Refresh `last_active_ts` and the payload snapshot |
| No longer present | Mark `recovered`, stamp `resolved_at` |
| `recovered` and fires again | **Revive** — preserve the original `first_seen_ts`, clear `resolved_at` |
| Operator acknowledges | Remove from the map entirely |

The revive rule is the important one. **A flapping sensor produces one entry per incident, not one entry per flap.** Only an explicit acknowledgement removes an entry; a recurrence after acknowledgement correctly begins a new incident with a fresh `first_seen_ts`.

Refreshing the payload snapshot on every reconciliation means severity and summary updates flow through mid-incident — a warning that escalates to critical is visible on the existing entry rather than creating a second one.

## Reading events

```
GET /api/bng/events                                    read_only
```

Synthesises the live set, reconciles it against history, and returns the current console.

```
{
  "events": [
    {
      "id": "sensor:FAN2",
      "kind": "sensor_status",
      "severity": "critical",
      "sensor": "FAN2",
      "value": 0.0,
      "unit": "RPM",
      "summary": "FAN2 reported lcr",
      "detail": "ipmitool flagged FAN2 as lcr (value 0.0 RPM).",
      "ts": 1753000000.0,
      "source": "ipmi",
      "link": "/bng/health",
      "link_label": "Health",
      "condition_state": "active",
      "first_seen_ts": 1752999000.0,
      "resolved_at": null
    }
  ],
  "count": 3,
  "active_count": 2,
  "recovered_count": 1
}
```


`condition_state` is `active` or `recovered`. `severity` is `critical`, `warning`, or `info`. Acknowledged identifiers are filtered out entirely.

**Sort order is deliberate**: recovered entries first, then active, each group ordered by `first_seen_ts` descending. Recovered entries lead because they are the ones requiring an operator decision — an active fault is still visible on the health endpoint, whereas a recovered one will vanish the moment it is acknowledged and is the item at risk of being missed.

For a recovered entry the payload is rewritten: the summary is prefixed `"RESOLVED — "` and the detail is replaced with the incident window and duration, followed by a note that acknowledgement is awaited.

Recall that this call has write side effects, and that a single collector should own it.

## Acknowledgement

```
DELETE /api/bng/events/{event_id}                    read_write
DELETE /api/bng/events                               read_write
```

Acknowledging a single event adds its identifier to the acknowledgement set, removes it from the history map, and releases any associated outage tracker. It is **idempotent** — re-acknowledging succeeds and does not duplicate the audit line. Acknowledging an unknown identifier also succeeds; there is no existence check, so an integration replaying acknowledgements is safe. The response is `{"ok": true, "acked": "<id>"}`.

The bulk form clears the console: it wipes the history map wholesale and adds the currently-live identifiers to the acknowledgement set. The distinction matters — **recovered entries are dropped, while still-active faults remain suppressed until they resolve and fire again.** The response is `{"ok": true, "cleared": <n>}`.

## The lifecycle log and history

```
GET /api/bng/events/log?limit=<n>                    read_only
GET /api/bng/events/history                          read_only
```

`/events/log` reads the tail of the append-only lifecycle log — the record of every transition, not the current state. Entries are returned newest first; `limit` defaults to 200 and is clamped between 1 and 2000. A non-integer `limit` falls back to the default rather than erroring.

Recorded actions include `fired`, `severity_changed` (carrying the old and new severity), `re_fired`, `condition_resolved` (carrying the incident window), `operator_acknowledged`, `operator_cleared`, and `auto_cleared`. The distinction between the last two is the operationally interesting one: `auto_cleared` means the condition resolved *without* an acknowledgement — the power supply was re-seated before anyone clicked.

The same log is the system's general audit sink; configuration writes and snapshot operations land here too, which is how they reach the merged audit timeline.

`/events/history` returns every event since the history was last cleared, including resolved ones, sorted by first-seen descending, with a computed `duration_s`. For an active event that duration is measured to *now* and therefore grows between polls; for a resolved one it is the incident length.

## Log rotation

The lifecycle log rotates on size, with a policy under the audit retention settings — by default 100 MB, ninety days, twelve archives, gzip-compressed. Rotation is amortised: the append path only considers rotating every hundredth write, so the common case costs nothing.

The rotation sequence is ordered to close a data-loss window. The live file is **renamed first** — an atomic operation, so a concurrent writer either landed in the old file or will land in the fresh one — then a new empty log is created, and only then is the renamed file filtered by age and written out as a compressed archive. Undecodable lines are **kept**, not dropped, on the principle that a corrupt audit line is still evidence.

Archives follow one naming convention with three consumers: the rotator writes them, the audit timeline's event reader globs them so a query spanning a rotation boundary still returns the archived entries, and the audit shipper ships them.

# Captive portal health

```
GET /api/bng/captive-portal/health                    read_only
GET /api/bng/captive-portal/preview                   read_only
```

The health endpoint performs a live probe of the configured portal on every call — it is not cached — and maintains a short in-process ring buffer of recent results.


```
{
  "configured": true,
  "portal_ip": "10.0.0.5",
  "url": "http://10.0.0.5:8088/health",
  "ok": true,
  "status_code": 200,
  "latency_ms": 12,
  "error": null,
  "consecutive_failures": 0,
  "history": [ { "ts": 1753000000.0, "ok": true, "status_code": 200, "latency_ms": 12, "error": null } ]
}
```


If no portal is configured the response is `{"configured": false, "ok": null, "consecutive_failures": 0, "history": []}` — note that `ok` is `null` here and boolean elsewhere.

The success rule is broader than it might appear: **any status from 200 through 499 counts as up.** A 404 means something is listening and speaking HTTP, which is the question being asked. `latency_ms` is populated even on failure, measuring time until the exception. An outage opens only after **two consecutive failures**, which is what keeps a single dropped probe from raising an alarm.

An outage identifier embeds the outage start time, so each distinct outage cycle gets a fresh identifier and correctly bypasses an acknowledgement of the previous one. On recovery the outage is deliberately kept in the event feed with `severity: "info"` and a "recovered (was intermittent)" summary, so an operator must still acknowledge that it happened.

`/captive-portal/preview` is a same-origin HTML proxy of the portal's landing page, provided so the UI can display the live portal in an iframe regardless of whether the operator's browser can route to the portal address. It returns HTML, not JSON: `400` if no portal is configured, `502` if the portal is unreachable, and otherwise the upstream body and content type.

# The Audit Timeline

An operational question — "what happened at 04:12 last Tuesday" — cannot be answered from any single log on a BNG. The relevant evidence is distributed across the controller's audit log, QuantumTouch's own event log, the systemd journal for four services, and the captive portal's access log. The audit timeline exists to merge them.

## Four sources, one shape

| Source | Origin | Default |
|---|---|---|
| `qt-events` | QuantumTouch's own event log and its archives | On |
| `controller-api` | The BNG controller's audit log | On |
| `systemd-journal` | `journalctl` across the BNG service units | On |
| `portal-access` | Captive portal HTTP access lines | **Off — opt in** |

Every entry from every source is normalised to the same eight fields:

```
{
  "ts": 1753000000.5,
  "source": "qt-events",
  "actor": "jdoe (session:9f8b2c14…)",
  "action": "event:qt_config_change",
  "summary": "Plan 'Silver-20M' updated",
  "detail": "rev_id=4412, kind=config_write, severity=info",
  "related_event_id": null,
  "raw": { "…the untouched original source record…" },
  "entry_id": "a1b2c3d4e5f6a7b8"
}
```

`raw` **is what makes lossy normalisation safe.** The original record is preserved verbatim on every entry, so no information is destroyed by the projection and a client can always fall back to source-native fields.

Actions are namespaced by source — `event:<action>`, `controller:<METHOD> <path>`, `service:{lifecycle|health|access|log}`, `portal:access` — so a consumer can filter by origin without inspecting the `source` field.

`entry_id` is a content-derived hash of the timestamp, source, and summary. It is **not persisted**; it is recomputable from any feed of the same data, which is what makes it usable as a cursor without server-side state.

## Actor resolution

A raw controller audit line attributes a change to an opaque session identifier. The timeline resolves these: session identifiers are mapped through the live session store to `"<username> (session:<abbreviated-id>…)"`. API key identifiers and anonymous callers pass through unchanged. An expired session, whose mapping is gone, degrades gracefully to the abbreviated identifier rather than disappearing.

**This is the mechanism that answers "who did this".** Without it, the controller's audit log attributes every UI-driven change to an anonymous or opaque principal.

## Journal detail levels

The systemd journal is voluminous and most of it is not operationally interesting. Journal lines are classified into `lifecycle`, `health`, `access`, and `other`, and the `detail` parameter selects a cumulative ladder:

| `detail` | Includes |
|---|---|
| `lifecycle` *(default)* | Service start, stop, failure, reload, main-process exit |
| `+health` | Plus VPP state changes, heartbeats, IPMI activity |
| `+access` | Plus HTTP access lines |
| `all` | Everything |

Unit attribution prefers the unit *acted upon* over the unit *emitting*, which is usually the init scope, with a regular-expression fallback that recovers a unit name from the message text.

## Querying

```
GET /api/bng/audit/timeline                          read_only
```

| Parameter | Default | Notes |
|---|---|---|
| `since` | now − 3600 | Epoch seconds. Unparseable falls back to the default. |
| `until` | now | Epoch seconds. |
| `limit` | 500 | Clamped to 1–5000. |
| `sources` | the three defaults | Comma-separated; intersected with the four known names. |
| `actor` | — | **Exact match.** |
| `q` | — | Case-insensitive substring over summary and detail. |
| `related_event_id` | — | Exact match. Used for deep-linking from an event. |
| `detail` | `lifecycle` | Journal ladder, above. |
| `since_cursor` | — | Incremental polling; see below. |
| `format` | — | `csv` or `json` for a file download. |

Sources are fetched **in parallel**, each with its own timeout guard, so wall clock is the slowest source rather than the sum. **A failing source never fails the request** — it is reported as degraded and the rest of the timeline is returned.

```
{
  "entries": [ … ],
  "count": 42,
  "window": { "since": 1752996400.0, "until": 1753000000.0 },
  "cursor": "a1b2c3d4e5f6a7b8",
  "sources": {
    "qt-events":       { "count": 10, "ok": true,  "error": null, "latency_ms": 3 },
    "controller-api":  { "count": 0,  "ok": false, "error": "connection refused", "latency_ms": 12 },
    "systemd-journal": { "count": 32, "ok": true,  "error": null, "latency_ms": 140 },
    "portal-access":   { "count": 0,  "ok": true,  "error": null, "latency_ms": 0 }
  },
  "truncated": false
}
```

**The `sources` object always contains all four keys**, including sources not requested, which appear zero-filled. A client therefore never has to handle a missing key. Note that the per-source `count` is that source's contribution *before* filtering and limiting, not the number of its entries that survived into `entries`.

Ordering is newest first, with ties broken by a fixed source priority ( `qt-events` , then `controller-api` , then `systemd-journal` ). Since the sort is newest-first, truncation drops the **oldest** entries; `truncated` reports when that happened.

**There is deliberately no deduplication and no clock-skew correction.** Every source is a clock on the same box. A single real-world moment appearing in two sources produces two rows, which is correct — they are two independent observations. The source-priority tiebreak exists to make those colocated rows read in a meaningful order, not to collapse them.

`since_cursor` supports incremental polling: pass back the `cursor` from the previous response and receive only entries newer than it. Be aware of what this is and is not — it reduces payload and client re-render, but the backend still fetches and merges the full window. It is not backend pagination. A cursor that has aged out of the window silently returns the full window rather than erroring.

Two supporting endpoints complete the surface. `GET /api/bng/audit/sources` probes the three default sources over a short window and reports liveness and latency without returning entries — useful as a collection health check. `GET /api/bng/audit/timeline/{entry_id}` resolves a single entry by its hash; note that it re-derives a **fixed one-hour window**, so an entry older than an hour returns `404` even though its identifier is valid.

## Raw source access

Two endpoints bypass normalisation and return a source in its native shape, for cases where the projection loses something needed.

```
GET /api/bng/audit/log?limit=<n>                      read_only
GET /api/bng/audit/journal?lines=<n>&units=<csv>      read_only
```

The first tail-reads the controller's audit log directly and returns its records verbatim, newest first, alongside a count of lines that failed to parse. It returns `404` with an explanatory body if the log file does not exist.

The second shells `journalctl` across a default set of BNG service units — the deploy service, the controller, VPP, the VPP bridge, the portal, and the platform apply units — and projects each record to timestamp, unit, priority, message, syslog identifier, and PID. **The journal's native microsecond timestamps are passed through as strings**, unlike everywhere else in the operational plane. Units may be overridden by parameter. Failure modes are explicit: `500` if `journalctl` is unavailable or exits non-zero, `504` if it exceeds its fifteen-second budget.

## Export

```
GET /api/bng/audit/export?from=&to=&format=&sources=&q=&only_logins=   read_only
```

A bulk download for compliance archival and offline analysis, distinct from the timeline's own `format` parameter.

`from` and `to` are **required** and accept either epoch seconds or ISO-8601; naive timestamps are interpreted as UTC. The window is capped at **one year**. `format` is `csv`, `json`, or `ndjson` — note that both `json` and `ndjson` produce newline-delimited JSON. `only_logins` restricts the export to authentication events, which is the common compliance query.

The CSV carries a fixed column set — timestamp, source, actor, action, summary, related event, detail, and the full raw record as embedded JSON — and applies **formula-injection defence**: any cell beginning with a character a spreadsheet would interpret as a formula is prefixed with an apostrophe.

Two behaviours to note. The export **degrades rather than fails** — a source that errors is skipped and you receive a partial export with a `200`, so verify the row count against expectation rather than assuming completeness. And the export itself is audited: an export event is emitted recording the actor, window, format, filters, and row count.

Validation errors are `400` with an explanatory `error` string; the messages cover missing bounds, unparseable timestamps, an inverted range, an over-long range, and an unknown format.

## Retention and projection

```
GET /api/bng/audit/storage                              read_only
```

Reports the audit log's disk footprint and projects when it will reach its cap.

```
{
  "current_size_mb": 12.345,
  "archive_count": 3,
  "oldest_archive_at": "2026-04-12T00:00:00Z",
  "oldest_live_entry_at": 1752700000.0,
  "projected_full_in_days": 23
}
```

The projection is a linear extrapolation from the observed growth rate against the configured cap. It returns `null` when there is insufficient data — no entries, zero size, or less than a minute of history — and `0` when already at or over the cap. Note that `oldest_archive_at` is read from the archive **filename** rather than the file's modification time, so it survives file copying and restoration. The endpoint always returns 200 and degrades every field independently.

# Outbound shipping

```
GET    /api/bng/audit/shipping/destinations            admin_only
POST   /api/bng/audit/shipping/destinations            admin_only
PATCH  /api/bng/audit/shipping/destinations/{name}     admin_only
DELETE /api/bng/audit/shipping/destinations/{name}     admin_only
POST   /api/bng/audit/shipping/destinations/{name}/test admin_only
GET    /api/bng/audit/shipping/status                  admin_only
```

The one place where the operational plane pushes rather than waits. A background daemon tails the audit log from a persisted byte offset and fans each new line into **one on-disk spool per destination, each with its own delivery thread and retry budget** — so a slow SIEM endpoint cannot stall an S3 archive.

```
event_log.jsonl ──tail──▸ shipper daemon
                            └▸ spool/<destination>/queue.jsonl ──drain──▸ send(batch)
```

Three destination types are supported.

`syslog` speaks RFC 5424 over TCP, UDP, or TLS, using octet-counting framing for the stream transports. Facility, severity, application name, TLS verification, and a CA file are configurable.

`http` posts batched JSON to any endpoint — Splunk HEC, Datadog, Loki, a Vector pipeline. A `2xx` is success. A `5xx`, a timeout, a `408`, or a `429` is retried. Other `4xx` responses are treated as success and **not** retried: a malformed batch that a server will reject forever is a poison pill, and retrying it indefinitely would block every subsequent entry behind it.

`s3` batches on an interval, defaulting to five minutes. When the batch window has not elapsed the destination reports a distinct "not yet" outcome that the daemon treats as *skip and keep spooled* rather than as a failure — a healthy S3 destination therefore does not accumulate a phantom failure count between batches.

Retry is three attempts with exponential backoff. On persistent failure entries remain spooled up to a hundred-megabyte cap, beyond which the **oldest entries drop first**. Log rotation is detected by offset comparison, and the newer archives are shipped before resuming the live tail, so the pre-rotation tail is never lost.

Destination secrets — bearer tokens and S3 secret keys — are encrypted at rest under a box-local key and are **never returned by the API**. The list endpoint redacts them and reports only a `has_secret` boolean. A `PATCH` that omits a secret field preserves the stored one, so a partial update cannot accidentally blank a credential.

The type of an existing destination is **immutable**; change it by deleting and recreating.

The `test` endpoint sends one synthetic entry. Importantly, **it builds the destination from the stored configuration only** and ignores the request body — this prevents the endpoint from being used to make the BNG issue arbitrary outbound requests.

`GET /shipping/status` reports per-destination spool size, last success and last attempt timestamps, consecutive failure count, batches sent, entries shipped, and an `up` boolean. On a box with no daemon it returns a zero skeleton rather than an error. These same values are published as Prometheus gauges, so shipping health can be alarmed on without polling this endpoint.

# Configuration Snapshots

The snapshot subsystem gives a BNG the configuration-archive semantics an operator expects from carrier equipment: capture the whole box, verify it, compare it against the running state, and put it back.

## What is captured

A snapshot captures **five domains** that together constitute the node's configuration:

| Domain | Source | If missing |
|---|---|---|
| `qt_settings` | The QuantumTouch settings document | `{}` |
| `controller_config` | The controller's full configuration | **Capture aborts** |
| `vpp_startup_conf` | The VPP startup configuration | `""` |
| `grub_cmdline` | The kernel command line, including drop-ins | `""` |
| `sysctl_hugepages` | The hugepages sysctl drop-in | `""` |

The asymmetry is load-bearing. A missing sysctl drop-in is normal on a freshly-installed box and captures as empty. An unreachable controller **aborts the capture with** `503`, because a snapshot missing the controller's configuration would, on a later merge-mode restore, silently appear to contain no plans and no pools.

**Secrets are captured in full, by design** — RADIUS shared secrets and captive portal credentials included. This matches the behaviour of configuration archives on established carrier platforms, where a restored archive must actually restore service. Every snapshot therefore carries a warning to that effect in its `warnings` array, and snapshot endpoints are `admin_only` throughout.

## The snapshot object


```
{
  "id": "snap-20260623T160955Z-pre-upgrade",
  "name": "pre-upgrade",
  "description": "Before the 1.2.0 controller rollout",
  "tags": ["golden"],
  "kind": "manual",
  "created_at": 1782700000.0,
  "created_by": "jdoe",
  "qt_version": "0.9.11",
  "core_version": "0.9.0",
  "schema_version": 1,
  "content_hash": "sha256:4f1a…",
  "captured": { "qt_settings": {}, "controller_config": {}, "vpp_startup_conf": "", "grub_cmdline": "", "sysctl_h
  "warnings": ["Contains RADIUS shared secrets and captive-portal credentials. Treat as sensitive."]
}
```


The identifier encodes the capture time and a slug of the name, which makes a directory listing chronologically sortable and human-readable.

`qt_version` and `core_version` record the software that produced the snapshot, so a restored configuration carries the provenance of its origin.

## Content hashing

The hash covers **only the captured configuration** — not the name, not the tags, not the timestamp, not the description:

```
content_hash = "sha256:" + sha256(canonical_json(captured))
```

Two consequences follow, both intended. Renaming or re-tagging a snapshot does not change its hash. And **two snapshots of identical configuration are visibly duplicates by matching hash**, regardless of when they were taken or what they were called. The UI renders the trailing hex as a fingerprint chip for exactly this comparison.

The hash is verified at three points, with escalating strictness: **on read**, as a non-fatal warning appended to the snapshot's `warnings`; **on restore**, as an additional confirmation gate; and **on import**, as a hard rejection.

Layered above the hash is an HMAC provenance signature, applied only on export and computed under a box-local secret. On import the signature yields one of three outcomes: a match means the snapshot originated on this box; a mismatch means it came from another box and is accepted **with a warning**; and an absent signature means a legacy export and is accepted silently. **This is a provenance hint, not an authorisation gate** — cross-box import is a supported workflow, not an attack to be blocked.

## Managing snapshots

```
GET    /api/bng/snapshots?tag=<t>                    admin_only
POST   /api/bng/snapshots                            admin_only
GET    /api/bng/snapshots/{id}                       admin_only
DELETE /api/bng/snapshots/{id}                       admin_only
POST   /api/bng/snapshots/{id}/tags                  admin_only
POST   /api/bng/snapshots/_prune                     admin_only
GET    /api/bng/snapshots/{id}/export                admin_only
POST   /api/bng/snapshots/import                     admin_only
```

Creation takes a required `name`, an optional `description`, and optional `tags` matching `[A-Za-z0-9_-]{1,32}`. It returns `201` with the full snapshot object, `400` on validation failure, `503` if the controller could not be read, and `500` if the settings document is not valid JSON.

The listing returns metadata only — the `captured` block is stripped — sorted newest first, and may be filtered by tag. Hash verification runs during listing, so a corrupted or tampered snapshot is flagged in its `warnings` without needing a separate check. A file that fails to parse entirely is skipped rather than blackholing the listing.

Tag assignment replaces the tag list wholesale. Export returns the snapshot with its provenance signature attached and an attachment content disposition. Import accepts that document, strips the signature before storage, and returns the import verdict:

```
{
  "id": "snap-20260623T160955Z-pre-upgrade",
  "signature_verified": true,
  "origin": "same-box",
  "warnings": []
}
```

`origin` is `same-box`, `cross-box`, or `legacy`.

**Pruning** is policy-driven and disabled by default. When enabled it applies a maximum count and optionally a maximum age, with **union semantics** — a snapshot is pruned if it trips either cap. Snapshots tagged `golden` are **protected**: they are excluded before the caps are evaluated, so they neither get pruned nor consume a slot in the count budget. Pruning runs inline after each creation and on a scheduled trigger, and the `_prune` endpoint invokes it on demand. Every pruned snapshot is recorded in the audit log. Manual deletion is unconditional and has no undo.

## Diff

```
GET /api/bng/snapshots/{id}/diff                     admin_only
GET /api/bng/snapshots/diff?from=<a>&to=<b>          admin_only
```

The first captures live state without storing it and compares it against a stored snapshot — "what has changed since". The second compares two stored snapshots — "what changed between these two points".


```
{
  "diff": {
    "qt_settings": [
      { "path": "bng.audit.retention.max_size_mb", "current": 100, "snapshot": 250 }
    ],
    "controller_config": [
      { "path": "plans.Silver-20M.dn_cir_bps", "current": 20000000, "snapshot": 25000000 }
    ],
    "vpp_startup_conf": {
      "changed": true,
      "unified_diff": "--- current\n+++ snapshot\n@@ …",
      "current": "…", "snapshot": "…"
    },
    "grub_cmdline":     { "changed": false },
    "sysctl_hugepages": { "changed": false }
  },
  "requires_reboot": true,
  "reboot_reasons": ["grub_cmdline_changed"]
}
```


Structured domains are flattened to dot-paths and compared leaf by leaf. **Lists compare whole rather than element-wise** — a DNS server list with one entry changed reports as one changed path, not as an insertion and a deletion. A value present on one side and absent on the other appears as `null`, so additions, removals, and modifications are structurally identical and a client renders them uniformly. Text domains report either `{"changed": false}` or a unified diff.

`requires_reboot` is computed, not guessed, and `reboot_reasons` can only ever contain `grub_cmdline_changed` or `hugepages_count_changed`. Note that a changed VPP startup configuration does **not** set it — that is an in-band service restart, not a reboot.

## Restore

```
POST /api/bng/snapshots/{id}/restore                    admin_only
```

| Field | Default | Meaning |
|---|---|---|
| `confirm` | — | Must be exactly `true` unless `dry_run` is set |
| `dry_run` | `false` | Plan only: no lock taken, no writes performed |
| `mode` | `"merge"` | `merge` applies the snapshot's items; `replace` first removes current items absent from the snapshot |

**Concurrency is enforced with an exclusive on-disk lock.** A second restore attempt while one is in progress returns `409` with the holder's process ID and start time, both in the body and in a response header. This is the guard that makes restore safe to expose to automation.

**Always dry-run first.** A dry run short-circuits before the lock is taken and before anything is written, returning the same step structure with `would-` prefixed statuses. Its one honest limitation is that the controller configuration step cannot be verified without performing it, and reports as `would-ok` with a note to that effect.

A real restore proceeds in ordered steps, and the order is chosen so that the riskiest write happens first, while nothing else has been touched:

1. `vpp_startup_conf` — the only hard abort. A failure here stops the restore with nothing else modified.
2. `vpp_restart` — performed **only if** the startup configuration actually changed.
3. `controller_replace_deletes` — replace mode only, removing items in reverse dependency order.
4. `controller_config` — applied in the controller's natural dependency order: DNS, then pools, then plans, then RADIUS, then captive portal, then interface enablement, then uplinks. Uplinks are last because sub-interface creation requires the parent access port to be up.
5. `qt_settings` — written atomically.
6. `grub_and_sysctl` — each gated on an actual text change, with the GRUB command line surgically substituted rather than the file rewritten.

```
{
  "ok": false,
  "result": "partial",
  "steps": [
    { "name": "vpp_startup_conf",  "status": "ok",      "detail": "" },
    { "name": "vpp_restart",       "status": "skipped", "detail": "startup.conf unchanged" },
    { "name": "controller_config", "status": "ok",      "detail": "9 blocks applied" },
    { "name": "qt_settings",       "status": "ok",      "detail": "" },
    { "name": "grub_and_sysctl",   "status": "failed",  "detail": "update-grub rc=1: …" }
  ],
  "requires_reboot": true,
  "reboot_reasons": ["grub_cmdline_changed"],
  "extras_deleted": [],
  "mode": "merge"
}
```

`result` is `ok`, `partial`, `failed`, or `needs_deep_reset`. Status codes map accordingly: `200` for success, `207 Multi-Status` **for partial**, `409` for a held lock or a wedged pipeline, `400` for a missing confirmation, `404` for an unknown snapshot.

**The restore is not transactional and there is no automatic rollback.** This is stated plainly because it drives integration design: a partial result is a real outcome an automation must handle, and the per-step report exists precisely so it can. The recommended pattern is dry-run, then restore, then diff to confirm convergence.

In replace mode, the subscriber and captive pools are hardcoded as non-deletable, and a delete returning "not found" is treated as success so the operation is idempotent.

If the controller's state machine is found wedged, the result is `needs_deep_reset` with a hint naming the recovery action. **A deep reset is never triggered automatically.**

# Identity and Key Administration

```
GET    /api/bng/api-keys?include_sessions=<bool>      admin_only
POST   /api/bng/api-keys                              admin_only
PATCH  /api/bng/api-keys/{id}                         admin_only
DELETE /api/bng/api-keys/{id}                         admin_only
POST   /api/bng/api-keys/{id}/rotate                  admin_only
```

Key administration is itself an API, which is what makes credential lifecycle automatable — provisioning a new monitoring system, rotating a credential on a schedule, or revoking one during an incident are all scripted operations rather than console work.

The listing returns metadata only:

```
{
  "keys": [
    { "id": "nms-poller", "label": "Solarwinds collector", "roles": ["read_only"],
      "revoked": false, "created_at": 1782700000.0,
      "fingerprint": "sha256:9f8b2c146e3a4a7d", "imported": false }
  ]
}
```

**Raw key material is never returned here, or anywhere except at creation.** The `fingerprint` is a truncated digest suitable for identifying a key in the UI or in a log without disclosing it. Browser session records are stored alongside API keys and are filtered out unless `include_sessions` is set.

Creation takes an `id`, a `role` (or a `roles` list), an optional `label`, and the three optional restrictions described in *Access Model*. The response is the one and only disclosure of the key:

```
{ "id": "nms-poller", "raw": "quantum_9f8b2c146e3a4a7d1c0e4d5a6b7c8d9e", "fingerprint": "sha256:9f8b2c146e3a4a7d"
```

Store it at that moment; it cannot be recovered.

Revocation is a `PATCH` setting `revoked`, returning `204`. Deletion removes the record entirely, also `204`. Rotation revokes the existing key and mints a replacement under the same identifier, preserving the role and label and returning the new raw material.

**Every mutation triggers a controller reload** so the controller's own view of valid credentials converges immediately. A key revoked through this API stops working on both planes within one reload cycle.

**One operational caution for rotation.** Rotation replaces the credential in-place with no overlap window — the moment it returns, the old key is dead. For a scraper or a poller that cannot tolerate a gap, the safer pattern is a **two-identity rotation**: create a new key under a new identifier, deploy it to the consumer, confirm traffic on the new credential, then revoke the old one. This is the pattern the platform's own scrape-credential rotation uses.

# Provisioning and Lab Surfaces

## Setup wizard

```
POST /api/bng/setup-wizard/apply                    admin_only
```

Applies a complete day-one configuration in a single call: service plans, address pools, captive portal, RADIUS servers, data-plane interface selection, platform identity, and the authentication mode. Each section is dispatched to the controller independently, in a fixed order, inside its own error guard.

**The call always returns `200`.** There is no aggregate failure status. Outcomes are reported per section:

```
{
  "results": [
    { "section": "Plans",          "ok": true,  "detail": "2/2 applied" },
    { "section": "Pools",          "ok": false, "detail": "1/2 applied (poolB: 422)" },
    { "section": "Captive Portal", "ok": true,  "detail": "portal_ip=10.0.0.5 on 100.64.64.0/18" },
    { "section": "RADIUS",         "ok": true,  "detail": "auth=2, acct=2" },
    { "section": "Data Plane",     "ok": true,  "detail": "uplink=… access=…" },
    { "section": "Platform",       "ok": true,  "detail": "recorded: hostname=…, tz=…" },
    { "section": "Authentication", "ok": true,  "detail": "mode=radius" }
  ],
  "saved_at": 1782700000.0
}
```

The `results` array always contains exactly these seven entries in this order, so a script can index them positionally. **An automation must inspect every `ok` field; a `200` alone does not mean the configuration applied.**

Three sections are recorded rather than applied. Data-plane interface selection is persisted for later finalisation on the dataplane workflow; platform identity requires installer-level privilege; and the authentication mode is a settings value rather than a controller call. The response details say so explicitly.

The submitted configuration is persisted as a wizard snapshot with **RADIUS secrets and the local password redacted** in the stored copy. The plaintext values are of course forwarded to the controller, which needs them — the redaction applies to the on-disk record.

## Load generation

```
POST /api/bng/demo/start                    read_only
POST /api/bng/demo/stop                     read_only
GET  /api/bng/demo/status                   read_only
POST /api/bng/demo/reset                    read_only
POST /api/bng/demo/deep-reset               read_only
POST /api/bng/demo/trex/start               read_write
POST /api/bng/demo/trex/stop                read_write
GET  /api/bng/demo/trex/status              read_write
```

These are lab and acceptance-test surfaces, and they exist for a specific reason that matters during evaluation: **a BNG can be driven to a realistic subscriber and traffic state through its own API, with no external test equipment.**

The **session generator** creates synthetic subscribers through the real DORA path — they are genuine sessions in the controller's session table, with real plans applied and real policers programmed — and optionally emits accounting. Rate and packet size per session are configurable and clamped to sane bounds. It reports live progress: sessions created, connected and captive counts, packets and bytes sent, elapsed runtime, and any error.

`reset` tears down the synthetic sessions by their reserved MAC prefix, leaving real subscribers untouched. `deep-reset` is destructive: it snapshots the controller's BNG configuration, issues a full persistent reset, and replays the configuration. It exists for a wedged state machine and returns the pre-reset snapshot in its response so nothing is lost.

The **traffic generator** drives external load through a remote host, supporting a steady rate or an oscillating pattern stepping through a list of rates. It is `read_write` throughout, including its status endpoint.

For a competitive evaluation the practical value is that a bake-off script can, through one API and one credential, bring a bare box to configured state, drive subscribers onto it, generate representative traffic, and then read the QoS counters, the retained time series, and the health verdict that result — with no manual step in the loop.

# Integration Patterns

## Establishing a session

The complete bootstrap for a monitoring integration:

```
# 1. An administrator mints a scoped read-only key (one time, from an admin key
#    or the UI). The raw value is returned exactly once.
curl --cacert bng-ca.pem \
     -H "Authorization: Bearer $ADMIN_KEY" \
     -H "Content-Type: application/json" \
     -d '{"id":"nms-poller","role":"read_only","label":"NMS collector",
          "ip_allowlist":["10.20.0.0/24"]}' \
     https://bng-edge-01:8443/api/bng/api-keys

# 2. The integration polls with that key. No CSRF token, no cookie.
curl --cacert bng-ca.pem \
     -H "Authorization: Bearer quantum_…" \
     https://bng-edge-01:8443/api/bng/health

# 3. The same key reaches the raw telemetry plane through the passthrough.
curl --cacert bng-ca.pem \
     -H "Authorization: Bearer quantum_…" \
     https://bng-edge-01:8443/v1/bng/counters
```

## Choosing a channel

Four distinct channels are available and each suits a different purpose. Using the wrong one is the most common source of integration friction.

| Need | Channel |
|---|---|
| Alarm on node up/down and capacity | `GET /api/bng/health`, polled every 5–15 s |
| Standard metric collection into an existing TSDB | Scrape `/metrics` with a scoped key |
| Charts and historical queries | `/throughput/recent`, `/metrics/query_range`, or `/analytics/v1/*` |
| Per-subscriber traffic and QoS detail | The `/v1/*` passthrough — raw telemetry |
| Fault console with acknowledgement | `/api/bng/events` plus the `DELETE` acknowledgement |
| "Who changed what, when" | `/api/bng/audit/timeline` |
| Continuous compliance feed into a SIEM | Configure an audit shipping destination |
| Change control and rollback | `/api/bng/snapshots` with dry-run restore |

The general principle: **for anything about subscribers, use the raw telemetry plane; for anything about the node's operation, use this one.**

## Two-sample rate derivation

The raw telemetry plane returns cumulative counters, and the companion document describes the two-sample pattern for converting them to rates. Where this API is concerned the same rules apply to any counter it passes through, with two additions specific to the operational plane.

**Counters may reset without warning.** A controller restart re-baselines every cumulative counter. In the raw plane this appears as a decrease between samples and must be treated as a reset rather than as negative traffic. In the Prometheus plane it is handled for you — the emitter detects the reset, re-baselines silently, and increments `bng_metric_baseline_reset_total`. **Alarming on that counter is the cheapest way to see controller restarts in monitoring.**

**Prefer the retained series over hand-rolled sampling.** If a rate is available from `/throughput/recent`, `/metrics/query_range`, or a Prometheus scrape, use it. Those paths already implement the reset handling, the gap semantics, and the step alignment described earlier. Hand-rolled two-sample arithmetic is appropriate for values this API exposes only as cumulative totals — the per-interface byte counters on `/dataplane-stats`, for instance.

## Incremental audit collection

The efficient pattern for a continuous audit feed is cursor-based:

```
GET /api/bng/audit/timeline?since=<t0>&limit=1000
  → note the returned `cursor`

GET /api/bng/audit/timeline?since=<t0>&since_cursor=<cursor>&limit=1000
  → only entries newer than the cursor; note the new `cursor`
```

Three cautions. Advance `since` as well as the cursor, or the window will grow without bound and the backend will re-merge an ever-larger span. Check `truncated` — if it is true, entries were dropped from the *oldest* end and the window should be narrowed or the limit raised. And check the per-source `ok` flags: a degraded source produces a silently incomplete timeline, which for a compliance feed is worse than an error.

For genuine compliance archival, prefer configured **audit shipping** over polling. It is push-based, spooled, retried, and survives restarts and log rotation.

## Change control

The intended change-control loop, fully scriptable:

```
POST /api/bng/snapshots              {"name": "pre-change-4412"}
    → capture and verify

  … apply the change, through /v1/* or the wizard …

GET  /api/bng/snapshots/{id}/diff
    → confirm exactly what changed, and whether a reboot is implied

  … if the change must be reverted …

POST /api/bng/snapshots/{id}/restore  {"dry_run": true}
    → review the step plan
POST /api/bng/snapshots/{id}/restore  {"confirm": true}
    → apply; expect 200, or 207 with a per-step report
GET  /api/bng/snapshots/{id}/diff
    → confirm convergence
```

Tag snapshots that must survive automatic pruning with `golden`.

## Robustness checklist

The behaviours most likely to surprise a first integration, collected in one place:

- `1m` **means one month.** There is no one-minute range on either time-series endpoint.
- **Time-series values are strings.** Parse the second element of every sample pair, including for integer-valued series.
- `/dataplane-stats` **is stateful.** One collector only, or use the cumulative byte counters instead.
- `GET /api/bng/events` **mutates.** One collector, sane interval.

- `/setup-wizard/apply` **always returns 200.** Inspect every per-section `ok`.
- `/audit/export` **degrades to partial.** Verify the row count.
- **Restore is not transactional.** Handle `207`.
- **An invalid** `step` **or** `limit` **does not error** — it silently falls back to the default. Only an invalid `range`, an unknown `metric`, and a missing required `ip` return `400`.
- **Never retry a** `403`**.** Honour `Retry-After` on `429`. Re-mint before retrying a `401`.
- **A flat time series may mean sampling stopped**, not that the value fell to zero. Cross-check health or the scrape liveness.

# Boundaries and Stability

## What this interface is not

Three boundaries are worth stating plainly so an integration is scoped correctly.

**It is not the subscriber telemetry interface.** Per-subscriber counters, session records, address pools, plans, and the DHCP protocol counters live on the controller's own API and are documented in the companion *BNG Telemetry API Theory of Operations*. Reach them through the `/v1/*` passthrough. Deliberately, **no per-subscriber series exist in the Prometheus store** — that data is served on demand from the controller and retained briefly in the ring buffer, which is what bounds cardinality, storage, and exposure.

**It is not a push interface for telemetry.** Consumers poll, or they scrape Prometheus. QuantumTouch does not open webhooks or streams toward a monitoring system. Audit shipping is the one outbound mechanism and it ships audit records to explicitly configured destinations — not telemetry, and not to arbitrary callers.

**It is not the log stream.** Structured service logs are written to standard output and consumed through the operator's log pipeline. The journal endpoints in this document provide bounded, filtered access for incident investigation; they are not a substitute for a log shipper on a busy box.

An integration therefore has four channels available — structured operational REST, the raw telemetry passthrough, the Prometheus scrape, and the log stream — and should use each for what it is suited to.

## Versioning and compatibility

Three version tracks operate independently and conflating them is a common mistake.

| Track | Mechanism | Contract |
|---|---|---|
| **Product** | Semantic versioning on the `quantumtouch-bng` and `quantumtouch-core` packages | Ordinary release versioning |
| **API path** | `/api/bng/analytics/v1/*` | The **only** versioned path prefix — the customer-facing compatibility boundary |
| **Metric registry** | A registry-wide version plus a per-metric `since_version` | Semantic versioning inside the registry, so a consumer can pin against it |

The unversioned `/api/bng/*` operator routes are an internal contract between the QuantumTouch UI and its backend, shipped together and evolving together. They are documented here because integrators legitimately use them and because parity with the UI is a design commitment — but the formal compatibility promise attaches to the versioned analytics prefix and to the metric registry.

**Response evolution is additive.** New fields are added to existing payloads on the standing assumption that consumers ignore keys they do not recognise. An integration should parse defensively and not fail on unknown fields.

The PromQL passthrough is deliberately **not** versioned by QuantumTouch — it inherits Prometheus's own `v1`, because verbatim mirroring of the upstream contract is the entire point of the surface.

Data formats carry their own schema versions independent of the software version: the key store and the snapshot format each version separately, with migration applied transparently on read.

## Authoritative reference

The payloads in this document are representative and correct in structure. For request and response shapes at field-level precision, the **interactive API browser served at `/api-docs`** on every box is generated from the running code and is the authoritative source. The machine-readable metric registry, exposing each family's type, labels, unit, source, and introducing version, is served from the versioned analytics surface and is the authoritative reference for anything scraped from `/metrics`.

# Summary

QuantumTouch presents the BlueDot BNG's operational plane through a single authenticated REST interface, served over HTTPS on one port, under one credential model, alongside a transparent passthrough to the raw subscriber telemetry documented in the companion reference. One API key against one host reaches both planes.

Where the telemetry API answers *what are the subscribers doing*, this interface answers *how is the node operating and what has been done to it*. It composes health from the controller, the data plane, the platform, and its own reply-path detector into a single alarming verdict that remains meaningful even when the controller is unreachable. It retains time series in a dual store — Prometheus for history, an in-process ring buffer as standby and as the permanent owner of per-subscriber series — and reports which store answered rather than hiding the distinction. It models events as incidents with a sticky lifecycle and explicit acknowledgement, so a condition that recovers unobserved is not lost. It merges four independent evidence sources into one normalised audit timeline, resolves opaque session identifiers to real operator names, and ships that record onward to a SIEM with per-destination spooling and retry. It captures, hashes, diffs, and restores complete node configuration with dry-run planning, per-step reporting, and concurrency locking. And it makes credential lifecycle itself an API, so provisioning, scoping, rotation, and revocation are scripted operations.

The property that ties it together is parity. The QuantumTouch web interface is an ordinary client of these endpoints and holds no privileged path. Everything an operator can see or do through the product, an integrator can see or do through the API — which means an NMS integration, an automation pipeline, or a custom operations portal is not working around the management plane but building on the same foundation the management plane is built on.