vq-db REST API

Sevana Ou PTE

68 Circular Road #02-01 Singapore 049422

© 2026 Sevana Ou PTE. All rights reserved.

vq-db REST API

The `vq-db` process exposes a small JSON HTTP API used by the bundled web dashboard and by any third-party tool that needs to query the call-quality data collected from one or more `vq-core` probes.

Overview

- **Transport:** plain HTTP/1.1 (no TLS, no authentication — terminate with nginx/HAProxy in front for production).
- **Listening port:** configured by `dashboard.port` in the YAML config (default **9126**).
- **Methods:** every endpoint listed below accepts `GET` only; any other method returns `501 Not Implemented`.
- **Encoding:** responses are pretty-printed JSON (`Content-Type: application/json`) except for the explicit CSV download mode on `/stats`, which returns `text/csv` plus a `Content-Disposition: attachment; filename=…` header.
- **Errors:** on application errors the server returns HTTP `503` with a JSON body of the form `{ "error": "<message>" }`. Required-parameter violations are reported the same way. Unknown paths fall through to the static-file handler (if `dashboard.root` is configured) and otherwise return `404`.
- **Agent metadata:** most endpoints include an `instance` object identifying the agent this `vq-db` is reporting for (`server.instance.id` / `.name` in the YAML config).

```
{
  "instance": { "id": "agent_1", "name": "First instance" }
}
```

Common types and parameters

Stream identifier

`stream_id` (alias `ref_id` in some responses) is a stringified `LinkId` that uniquely names a stream regardless of whether it currently lives in memory (active) or in the database (finished). It is opaque — clients should treat it as a token returned by the server and pass it back verbatim.

Search filter

Several endpoints accept a filter expression for either the in-memory list (`mem_filter`) or the database list (`db_filter`). The on-the-wire form is the URL-encoded representation of a `SearchFilter`:

```
<sort_field>/<sort_dir>/<page_offset>/<max_count>[/<hex_expression>]
```

Field	Meaning
`sort_field`	Column to sort by. Allowed: `start_time` (default), `end_time`, `duration`, `sevana_mos`, `sevana_rfactor`, `network_mos`, `jitter`, `src_ip`, `dst_ip`, `sip_src`, `sip_dst`.
`sort_dir`	`asc` or `desc`. Default `desc`.
`page_offset`	Zero-based row offset for paging.
`max_count`	Page size. Defaults to `dashboard.max-streams` from the YAML config (40).
`hex_expression`	Optional: a hex-encoded text filter expression (see *Filter expression grammar*).

Slashes inside the filter are escaped as `%2F`. The whole string is then URL-encoded as the `mem_filter` / `db_filter` query value.

Filter expression grammar

The optional `hex_expression` payload is a simple boolean/arithmetic expression evaluated per stream. Available variables (for in-memory rows; the database list maps the same names to SQL columns):

Variable	Meaning
`start_time`	Stream start (Unix seconds).
`sevana_mos`	Sevana MOS for the latest reported chunk.
`network_mos`	Network MOS (G.107 E-model).
`sevana_rfactor`	Sevana R-factor (0–100).
`jitter`	Last interval jitter (seconds).
`rtt_delay`	Round-trip delay (seconds).
`duration`	Stream duration so far (ms).
`src_ip` , `dst_ip`	Network endpoints.
`sip_src` , `sip_dst`	SIP user@host strings (empty if SIP correlation was not available).

Supported operators: arithmetic (`+ - * /`), comparison (`= != < <= > >=`), boolean (`and or not`), parentheses. Example:

```
sevana_mos < 3.5 and jitter > 0.020
```

Date interval

Both the in-memory and database lists accept an optional `[start, end]` time window. Provide one or both of `mem_start_timestamp` / `mem_end_timestamp` (and the `db_*` counterparts), each as a Unix-seconds integer. Streams whose start time falls outside the window are dropped from the result.

Paging actions

To page through the lists without recomputing offsets client-side, pass an `action` query parameter:

`action`	Effect on the in-memory list (`mem_*`)	Effect on the database list (`db_*`)
`mem_first`	jump to offset 0	—
`mem_last`	jump to the last page	—
`mem_next`	advance by one page	—
`mem_prev`	go back one page	—
`mem_refresh`	re-fetch the current page	—
`db_first`	—	jump to offset 0
`db_last`	—	jump to the last page
`db_next`	—	advance by one page
`db_prev`	—	go back one page
`db_refresh`	—	re-fetch the current page

If `action` is omitted the offsets from the filter strings are honored as-is.

Endpoints

`GET /stats` — list active and finished streams

The main listing endpoint. The same path serves three behaviors based on the query parameters: a JSON listing (default), a CSV download of the database, or a CSV download of the active stream list.

Query parameters

Parameter	Type	Default	Description
`mem_filter`	string	`""`	URL-encoded `SearchFilter` for the in-memory list.
`db_filter`	string	`""`	URL-encoded `SearchFilter` for the database list.
`mem_start_timestamp`	int	—	Unix seconds; lower bound of the in-memory date window.
`mem_end_timestamp`	int	now	Unix seconds; upper bound of the in-memory date window.
`db_start_timestamp`	int	—	Unix seconds; lower bound of the database date window.
`db_end_timestamp`	int	now	Unix seconds; upper bound of the database date window.
`sip_callid`	string	—	Restrict **both** lists to the streams correlated with this SIP Call-ID (exact match). See *Navigating between calls and streams*.
`action`	string	—	See *Paging actions*.
`show_active`	bool	`false`	Include the in-memory list (`active_streams`) in the JSON response.
`show_finished`	bool	`false`	Include the database list (`finished_streams`) in the JSON response.
`mem_show_count`	bool	`false`	Include the in-memory total count without the list body.
`db_show_count`	bool	`false`	Include the database total count without the list body.
`download_db`	bool	`false`	Stream the database list as CSV (overrides JSON mode).
`download_active`	bool	`false`	Stream the active list as CSV.
`show_detectors`	bool	`false`	Include PVQA detector columns in CSV exports.

If `download_db` is requested but the database is disabled (`database.engine` is empty), the server responds with HTTP `503` and `Database is not active; cannot export finished streams as CSV.`

JSON response


```
{
  "instance": { "id": "agent_1", "name": "First instance" },

  "total_active_streams_count":     12,
  "available_active_streams_count": 12,
  "active_streams": {
    "streams": [
      {
        "stream_id":      "…opaque…",
        "ref_id":         "…opaque…",
        "instance":       { "id": "agent_1", "name": "First instance" },
        "start_time":     "2026-05-25 09:13:42.001",
        "duration":       42.5,
        "duration_audio": 41.8,
        "sevana_mos":     4.1,
        "network_mos":    4.3,
        "sevana_rfactor": 92,
        "jitter":         0.0023,
        "source":         "192.168.1.10:30002",
        "destination":    "192.168.1.20:30000",
        "sip_src":        "alice@pbx.example",
        "sip_dst":        "bob@pbx.example",
        "sip_callid":     "9f6e@pbx.example"
      }
    ]
  },

  "total_finished_streams_count":     1734,
  "available_finished_streams_count": 1734,
  "finished_streams": {
    "streams": [ … same row shape … ]
  }
}
```


Keys that depend on the request:

- `active_streams` / `total_active_streams_count` / `available_active_streams_count` appear only when `show_active=true`.
- `finished_streams` / `total_finished_streams_count` / `available_finished_streams_count` appear only when `show_finished=true` (and the database is enabled).
- `available_active_streams_count` (alone) appears when `mem_show_count=true`.
- `available_finished_streams_count` (alone) appears when `db_show_count=true`.
- Setting both `show_active=true` and `show_finished=true` returns both lists in a single call (this is what the bundled dashboard does on load).

CSV download

When `download_db=true` or `download_active=true`, the response is `text/csv` with a `Content-Disposition: attachment; filename=vq-db-finished.csv` (or `vq-db-active.csv`) header. The header row of the CSV depends on `show_detectors`:

- without it: a basic row per stream (timestamps, endpoints, MOS, jitter, codec, …);
- with it: the basic row plus one column per PVQA detector counter.

For `download_db` the `db_filter` is applied; for `download_active` the entire in-memory list is exported with the detector columns.

`GET /streamhistory` — single-stream history

Returns the per-chunk timeline plus an aggregated summary for one stream.

Query parameters

Parameter	Required	Description
`stream_id`	yes	Stream identifier obtained from `/stats`.

If `stream_id` is missing the server returns HTTP `503` with `{"error":"Param stream_id is required."}`. If the ID does not match anything in memory or in the database, the response contains `{"error":"bad ref id"}` (also HTTP 503).

Response


```
{
  "instance": { "id": "agent_1", "name": "First instance" },

  "chunks": [
    {
      "link_id":          "…opaque…",
      "stream_end_time": 1716625984123,
      "start_time":       0.000,
      "end_time":         10.200,
      "sevana_mos":       4.2,
      "network_mos":      4.3,
      "jitter":           0.0019,
      "sevana_rfactor":   95
    }
  ],

  "sevana_mos":            4.1,
  "network_mos":           4.3,
  "jitter":                0.0021,
  "rtt_delay":             0.018,
  "decoded_filename":      "agent_1/2026-05-25/…wav",
  "codec":                 "PCMA",
  "rtp_packet_counter":    2150,
  "lost_packet_counter":   3,
  "amr_nb_switch_counter": 0,
  "amr_wb_switch_counter": 0,

  "source":                "192.168.1.10:30002",
  "destination":           "192.168.1.20:30000",
  "ssrc":                  "5a3f7e10",
  "sip_src":               "alice@pbx.example",
  "sip_dst":               "bob@pbx.example",
  "sip_callid":            "9f6e@pbx.example",

  "sevana_rfactor":        93,

  "Dead_Air":              4,
  "Click":                 1,
  "SNR":                   0,
  "_… one key per PVQA detector …_": 0,

  "json_report":           "{ … verbose internal JSON … }"
}
```


Top-level fields describe the whole stream; `chunks[]` lists the per-interval reports in chronological order. Detector counters (`Dead_Air`, `Click`, `SNR`, …) are flat keys at the top level — the field name is the detector name with spaces replaced by underscores, the value is the number of intervals that triggered the detector. `json_report` carries the full internal report and is intended for debugging — most clients can ignore it.

`GET /report` — single interval report

Returns the per-interval report for one specific chunk of a stream.

Query parameters

Parameter	Required	Description
`stream_id`	yes	Stream identifier.
`timestamp`	yes	`stream_end_time` of the desired chunk (as returned by `/streamhistory`).

Response


```
{
  "source":                 "192.168.1.10:30002",
  "destination":            "192.168.1.20:30000",
  "ssrc":                   "5a3f7e10",
  "stream_id":              "…opaque…",
  "codec":                  "PCMA",
  "sevana_mos":             4.1,
  "network_mos":            4.3,
  "jitter":                 0.0019,
  "rtt_delay":              0.018,
  "duration":               10.200,
  "rtp_packet_counter":     510,
  "lost_packet_counter":    0,
  "illegal_packet_counter": 0,
  "detectors_report":       "<text/CSV-style detector dump>",
  "amr_nb_switch_counter":  0,
  "amr_wb_switch_counter":  0,
  "sevana_rfactor":         95,
  "json_report":            "{ … verbose internal JSON … }"
}
```


If the timestamp does not match any stored interval, the response is HTTP 503 with `{"error":"Report not found"}`.

SIP call lifecycle endpoints

In addition to the per-stream RTP quality data, `vq-db` records the SIP signalling lifecycle of each call (decoded from the protobuf bus by `vq-core`) into the `rtpmon_sip_events` table. Three endpoints expose it: an aggregated per-call list (`/sip_calls`), the full event timeline for one call (`/sip_call`), and a flat cross-call event log (`/sip_events`).

> **Timestamps are UNIX milliseconds.** Unlike the `/stats` family (which uses Unix *seconds*), every timestamp accepted or returned by the SIP endpoints — `start_timestamp`, `end_timestamp`, `timestamp`, `invite_timestamp`, `duration` — is in **milliseconds** since epoch, matching the protobuf wire format.

All three endpoints require the database to be enabled. If `database.engine` is empty the server returns HTTP `503` with the body `Database is not active.`.

Event types and enum labels

`event_type` identifies the kind of SIP event; the endpoints also emit a human-readable `*_label` alongside each numeric enum:

event_type	event_type_label	Meaning
1	start	Call established (a 2xx answered the INVITE).
2	end	Call terminated normally (BYE).
3	reinvite	Mid-call re-INVITE / UPDATE (media renegotiation).
4	failed	Call never established (rejected, canceled, or timed out).

direction	direction_label (also bye_direction_label)
1	caller_to_callee
2	callee_to_caller
other	unknown

reason	reason_label
1	rejected
2	canceled
3	timeout
other	unknown

Event object shape

Both `/sip_call` and `/sip_events` return SIP events using the same JSON shape. Common keys are always present; the remaining keys depend on `event_type`:

```
{
  "id":               1234,
  "event_type":       1,
  "event_type_label": "start",
  "call_id":          "9f6e@pbx.example",
  "timestamp":        1716625984123
}
```

`event_type`	Additional keys
`1` start	`invite_timestamp`, `setup_code`, `caller`, `callee`, `caller_ua`, `callee_ua`, `caller_codecs`, `callee_codecs`
`2` end	`duration` (ms), `bye_direction`, `bye_direction_label`, `response_codes` (CSV string)
`3` reinvite	`direction`, `direction_label`, `is_request` (bool), `peer`, `peer_ua`, `peer_codecs` (the renegotiated side)
`4` failed	`invite_timestamp`, `reason`, `reason_label`, `response_code`, `reason_phrase`, `caller`, `callee`

`GET /sip_calls` — aggregated call list

Folds the raw events into one summary row per `call_id`, most-recently-active first (ordered by the call's latest event timestamp).

Query parameters

Parameter	Type	Default	Description
`start_timestamp`	int	—	Lower bound (Unix **ms**) on event timestamps.
`end_timestamp`	int	—	Upper bound (Unix **ms**) on event timestamps.
`call_id`	string	—	Restrict the result to a single SIP Call-ID (exact match). URL-encode it; the server URI-decodes the value. Handy to fetch the summary of one specific call.
`limit`	int	`dashboard.max-streams` (40)	Page size; values `<= 0` fall back to `50`.
`offset`	int	`0`	Zero-based row offset; negatives are clamped to `0`.

A call is included if any of its events falls within the `[start_timestamp, end_timestamp]` window. When `call_id` is given, `total_calls_count` is `1` if the call exists and `0` otherwise.

Response


```
{
  "instance": { "id": "agent_1", "name": "First instance" },
  "total_calls_count": 152,
  "calls": [
    {
      "call_id":          "9f6e@pbx.example",
      "caller":           "alice@pbx.example",
      "callee":           "bob@pbx.example",
      "start_timestamp": 1716625984123,
      "end_timestamp":   1716626042777,
      "duration":         58654,
      "reinvite_count":  1,
      "outcome":          "established",
      "setup_code":       200
    }
  ]
}
```


- `total_calls_count` is the number of distinct `call_id`s matching the window (independent of `limit` / `offset`), for paging.
- `outcome` is `established`, `failed`, or `unknown`.
- `setup_code` is present only when the call was established; `response_code`, `reason`, and `reason_label` are present only when the call `failed`.
- A timestamp of `0` means the corresponding event was not observed (e.g. `start_timestamp == 0` for a call seen only via a re-INVITE or failure).

`GET /sip_call` — full event timeline for one call

Returns every recorded event for a single `call_id`, in chronological order.

Query parameters

Parameter	Required	Description
`call_id`	yes	The SIP Call-ID. URL-encode it; the server URI-decodes the value.

If `call_id` is missing the server returns HTTP `503` with `{"error":"Param call_id is required."}`. An unknown `call_id` is not an error — it yields an empty `events` array with `event_count: 0`.

Response


```
{
  "instance":    { "id": "agent_1", "name": "First instance" },
  "call_id":     "9f6e@pbx.example",
  "event_count": 3,
  "events": [
    { "id": 1234, "event_type": 1, "event_type_label": "start", "call_id": "9f6e@pbx.example",
      "timestamp": 1716625984123, "invite_timestamp": 1716625983900, "setup_code": 200,
      "caller": "alice@pbx.example", "callee": "bob@pbx.example",
      "caller_ua": "…", "callee_ua": "…", "caller_codecs": "PCMA", "callee_codecs": "PCMA" },
    { "id": 1240, "event_type": 3, "event_type_label": "reinvite", "call_id": "9f6e@pbx.example",
      "timestamp": 1716626001500, "direction": 1, "direction_label": "caller_to_callee",
      "is_request": true, "peer": "alice@pbx.example", "peer_ua": "…", "peer_codecs": "PCMU" },
    { "id": 1255, "event_type": 2, "event_type_label": "end", "call_id": "9f6e@pbx.example",
      "timestamp": 1716626042777, "duration": 58654, "bye_direction": 2,
      "bye_direction_label": "callee_to_caller", "response_codes": "200" }
  ]
}
```


`GET /sip_events` — flat event log

Returns individual SIP events across all calls, most-recent first (ordered by `event_timestamp desc`), optionally filtered by type and time window.

Query parameters

Parameter	Type	Default	Description
`type`	int	— (all types)	Restrict to one `event_type` (`1`..`4`).
`start_timestamp`	int	—	Lower bound (Unix **ms**).
`end_timestamp`	int	—	Upper bound (Unix **ms**).
`limit`	int	`dashboard.max-streams` (40)	Page size; values `<= 0` fall back to `50`.
`offset`	int	`0`	Zero-based row offset; negatives are clamped to `0`.

Response

```
{
  "instance": { "id": "agent_1", "name": "First instance" },
  "total_events_count": 4123,
  "events": [ … event objects, see *Event object shape* … ]
}
```

`total_events_count` reflects the same `type`/time filters but ignores `limit`/`offset`, for paging.

Navigating between calls and streams

The SIP Call-ID is the join key between the signalling view (`/sip_calls`, `/sip_call`) and the media view (`/stats`). Each RTP stream row carries the `sip_callid` it was correlated with, and both listing endpoints accept an exact-match Call-ID filter, so a dashboard can move in either direction:

- **From a call to its streams:** take the `call_id` from a `/sip_calls` row and request `GET /stats?show_active=true&show_finished=true&sip_callid=<call_id>` to list just the RTP streams of that call.
- **From a stream to its call:** take the `sip_callid` shown on a `/stats` row and request `GET /sip_call?call_id=<call_id>` for the full signalling timeline, or `GET /sip_calls?call_id=<call_id>` for the folded summary.

The `sip_callid` filter on `/stats` applies to both the active and finished lists and combines with the other filters (date window, `db_filter`/`mem_filter` expression). Correlation is best-effort: a stream that was never matched to SIP signalling has an empty `sip_callid` and will not appear under any Call-ID filter.

`GET /instance_list` — agent enumeration

Lists every probe agent that has ever published into this `vq-db`. Useful for multi-probe dashboards.

Response

```
[
  { "id": "agent_1", "name": "First instance" },
  { "id": "agent_2", "name": "Second instance" }
]
```

The list is read directly from the database (`getAgentList`). If the database is disabled the array is empty.

`GET /server_stats` — server runtime statistics

Returns a snapshot of process-level state for the `vq-db` instance itself.

Response


```
{
  "instance":   { "id": "agent_1", "name": "First instance" },

  "capturers":  [ ],
  "hep":        { "device": "", "src_port_range": "", "dst_port_range": "",
                  "packets_total": 0, "packets_rtp": 0, "packets_sip": 0,
                  "packets_truncated": 0 },
  "decoder_pool": {
      "workers": [ ],
      "packets_received": 0,
      "packets_distributed": 0
  },

  "server_time": "2026-05-25 09:13:42.001",

  "pvqa_instance_counter":  0,
  "pvqa_processed_seconds": 0,

  "active_decoder_counter": 0,
  "total_decoder_counter":  0,
  "uptime":  86342,
  "version": "Server v1.8.4 , build number 13191"
}
```


In `vq-db` the capturer / decoder / PVQA fields are zero — those metrics are populated only by `vq-core`. The interesting fields here are `server_time`, `uptime` (seconds since process start), `version`, and `instance`.

`GET /dashboard/summary` — combined live + historical summary

Returns a single JSON envelope that merges (a) the most recent `instance_statistics` snapshot received from `vq-core` over the ZeroMQ bus and (b) two historical aggregates (`1h` / `24h`) computed from the local database. Designed for a single dashboard “overview” widget.

Request

No parameters. Only `GET` is accepted.

Response


```
        {
          "instance": { "id": "agent_1", "name": "First instance" },

          "core_available": true,
          "core": {
              "instance": { "id": "agent_1", "name": "First instance" },
              "capturers": [
                  { "device": "eth0", "src_port_range": "", "dst_port_range": "",
                    "packets_total": 1284321, "packets_rtp": 1240005,
                    "packets_sip": 21345, "packets_truncated": 0,
                    "packets_dropped_hw": 0, "packets_erroneous": 0, "mbuf_alloc_failed": 0 }
              ],
              "hep": { "device": "", "packets_total": 0, "packets_rtp": 0,
                       "packets_sip": 0, "packets_truncated": 0 },
              "decoder_pool": { "workers": [], "packets_received": 0, "packets_distributed": 0 },
              "server_time": "2026-05-26 10:15:00.123",
              "pvqa_instance_counter": 4,
              "pvqa_processed_seconds": 18432,
              "active_decoder_counter": 12,
              "active_audio_decoder_counter": 8,
              "total_decoder_counter":  9871,
              "sip_call_counter": 37,
              "resip_message_counter": 1840,
              "resip_sip_message_counter": 920,
              "mem_allocated_bytes": 268435456,
              "mem_heap_size": 314572800,
              "mem_pageheap_free_bytes": 8388608,
              "mem_alloc_count": 4821334,
              "mem_free_count": 4802901,
              "mem_allocs_per_sec": 1250.0,
              "mem_frees_per_sec": 1243.5,
              "cpu_user_seconds": 1832.4,
              "cpu_system_seconds": 412.7,
              "cpu_usage_percent": 23.5,
              "uptime":  86342,
              "version": "Server v1.8.16 , build number 14302"
          },

          "live": {
              "active_streams":   42,
              "session_started":  1287,
              "session_finished": 1245
          },

          "historical": {
              "good_mos_threshold": 3.6,
              "1h":  { "stream_count": 120,  "good_sevana_count": 110,  "good_network_count": 115,
"with_sevana_mos":  95,
                       "avg": { "sevana_mos": 4.10, "network_mos": 4.30, "rfactor": 0.08, "duration": 38.2 } },
              "24h": { "stream_count": 2880, "good_sevana_count": 2710, "good_network_count": 2790, "with_sevana_mos":
2310,
                       "avg": { "sevana_mos": 4.00, "network_mos": 4.20, "rfactor": 0.10, "duration": 41.7 } }
          }
        }
```

Field reference

Path	Meaning
`core_available`	`false` until the first `instance_statistics` message has been received from `vq-core` (e.g. the bus is down or the probe has just started). When `false` the `core` object is empty.
`core.capturers[]`	One entry per capture device on `vq-core`. The packet counters (`packets_total`, `packets_rtp`, `packets_sip`, `packets_truncated`) are cumulative since the `vq-core` process started. The NIC-level capture-loss counters (`packets_dropped_hw` — dropped by NIC HW when RX queues are full, `packets_erroneous` — erroneous packets/ierrors, `mbuf_alloc_failed` — RX mbuf allocation failures) are populated only on the DPDK capture path and stay `0` on libpcap/HEP.
`core.hep`	Same shape as a `capturers[]` entry, populated when a HEP listener is configured on `vq-core`.
`core.active_decoder_counter`	Decoder slots currently busy in `vq-core`.
`core.active_audio_decoder_counter`	Audio decoder slots currently busy in `vq-core` (subset of `active_decoder_counter`).
`core.total_decoder_counter`	Decoder slots used in total since the `vq-core` process started. Resets when `vq-core` restarts.
`core.pvqa_instance_counter`	Number of PVQA analyser instances currently running in `vq-core`. Present only when `vq-core` is built with PVQA support.
`core.pvqa_processed_seconds`	Total seconds of audio decoded and fed to PVQA since `vq-core` started.
`core.sip_call_counter`	Live `SipManager` Call objects in `vq-core` right now. A value that keeps climbing while decoders stay flat flags Call objects that are never released (e.g. calls with no BYE).
`core.resip_message_counter` / `core.resip_sip_message_counter`	Live reSIProcate object counts (leak indicators); the second is the SIP-specific subset.
`core.mem_allocated_bytes` / `core.mem_heap_size` / `core.mem_pageheap_free_bytes`	tcmalloc memory figures in bytes (live usage / total mapped / free-but-retained). All `0` when `vq-core` is not built against tcmalloc.
`core.mem_alloc_count` / `core.mem_free_count`	Cumulative tcmalloc allocation / free call counts since `vq-core` started; a widening gap indicates live objects (and possible leaks). Both `0` unless `vq-core` is built against tcmalloc **and** running with allocation tracking (`track-allocations`) enabled.

Path	Meaning
`core.mem_allocs_per_sec` / `core.mem_frees_per_sec`	Allocator load: tcmalloc allocation / free rate over the last sampling interval. Same gating as `mem_alloc_count` — `0` unless `track-allocations` is enabled.
`core.cpu_user_seconds` / `core.cpu_system_seconds`	Cumulative CPU time (user / kernel) consumed by the `vq-core` process, in seconds.
`core.cpu_usage_percent`	`vq-core` CPU utilisation over the last sampling interval (100% = one fully-busy core).
`core.uptime`	Seconds since `vq-core` started.
`live.active_streams`	Streams currently held in `vq-db`'s in-memory map (i.e. seen `stream_start` but no `stream_finish` yet).
`live.session_started` / `session_finished`	Streams observed by **`vq-db`** since `vq-db` itself started. Resets when `vq-db` restarts; not necessarily equal to `core.total_decoder_counter` (different reset point and different observer).
`historical.good_mos_threshold`	The configured threshold applied to **both** `good_sevana_count` and `good_network_count` (see `dashboard.good-mos-threshold` in the YAML — default `3.6`). Echoed so clients can label charts without re-reading config.
`historical.1h` / `24h`	Aggregates over the matching trailing window from `rtpmon_statistics`. `stream_count` is the row count; `good_sevana_count` is rows with `sevana_mos >= good_mos_threshold`; `good_network_count` is rows with `network_mos >= good_mos_threshold`; `with_sevana_mos` is rows with a non-null `sevana_mos` (i.e. PVQA actually ran). `avg.duration` is in seconds. The two `good_*` counts can differ — Sevana MOS reflects measured audio quality from PVQA, while Network MOS is the G.107 E-model estimate from transport metrics.

Configuration

```
dashboard:
  port: 9126
  good-mos-threshold: 3.6   # streams at or above this MOS count as "good"
```

Data retention

The data returned by the listing and detail endpoints (`/stats` finished list, `/streamhistory`, `/report`, and the SIP endpoints) is not kept forever. A background sweep (hourly) expires old data according to two independent lifetimes configured in the YAML; both default to `0`, which means **keep forever** (no sweeping). Durations accept a unit suffix — `7d`, `12h`, `30m`, `10200ms`, or a bare number (seconds).

```yaml
database:
  records-lifetime: 30d   # stream records (and SIP events); 0 = keep forever
  audio-lifetime: 7d      # decoded-audio rows; 0 = keep forever
```

- **`database.records-lifetime`** — bounds how long a stream stays queryable. A stream is expired as a **whole unit**: the parent stream row together with all of its interval / statistics / audio children are removed once the stream has no row newer than the cutoff *and* the stream itself is old. “Old” means it has only already-expired report rows, **or** it was opened before the cutoff. The second condition is what ages out a stream that never produced a report (opened on `stream_start` but seen with no interval/final) — it expires by its open time rather than lingering indefinitely. An in-flight stream (any recent row, or opened within the lifetime) is never deleted out from under the ingest pipeline.

  SIP events (`/sip_calls`, `/sip_call`, `/sip_events`) are not tied to a stream, so they are pruned independently by their own event timestamp under the same `records-lifetime`.

- **`database.audio-lifetime`** — bounds how long decoded-audio rows are retained, independently of the record lifetime. Once audio is swept, `decoded_filename` on a stream may reference data that is no longer present.

Clients that need long-term history should export it (the CSV download on `/stats`) before it ages out, or run with the lifetimes set to `0`.

Static dashboard

If `dashboard.root` is configured in the YAML, any request that does not match one of the endpoints above and that resolves to an existing file under that directory is served as a static asset. A request for the root path (`/`) returns `index.html`. This is the mechanism the bundled web dashboard uses.

Examples

```
        # Ask for a single combined snapshot of active + finished streams (what the dashboard does).
        curl 'http://localhost:9126/stats?show_active=true&show_finished=true'

        # Page 2 of the database list, 25 rows per page, sorted by Sevana MOS ascending.
        curl 'http://localhost:9126/stats?show_finished=true&db_filter=sevana_mos/asc/25/25'

        # Last 24 hours of finished streams as CSV (with detector counters).
        NOW=$(date +%s); YESTERDAY=$((NOW - 86400))
        curl -OJ "http://localhost:9126/stats?download_db=true&show_detectors=true&db_start_timestamp=${YESTERDAY}
&db_end_timestamp=${NOW}"

        # Drill into one stream and one chunk.
        SID="…stream_id from /stats…"
        curl "http://localhost:9126/streamhistory?stream_id=${SID}"
        curl "http://localhost:9126/report?stream_id=${SID}&timestamp=1716625984123"

        # SIP call lifecycle (timestamps are Unix milliseconds here).
        curl 'http://localhost:9126/sip_calls?limit=25'
        curl 'http://localhost:9126/sip_call?call_id=9f6e%40pbx.example'
        curl 'http://localhost:9126/sip_events?type=4'   # only failed-call events

        # Navigate between a SIP call and its RTP streams via the shared Call-ID.
        CID='9f6e@pbx.example'
        curl "http://localhost:9126/sip_calls?call_id=${CID}"                                  # the call summary
        curl "http://localhost:9126/stats?show_active=true&show_finished=true&sip_callid=${CID}" # its RTP streams

        # Process metadata.
        curl http://localhost:9126/server_stats
        curl http://localhost:9126/instance_list
```

Error responses

HTTP code	When
`200`	Successful response, body is JSON (or CSV in download mode).
`404`	Path does not match any handler and no static file exists.
`501`	The request method is not `GET`.
`503`	An exception was thrown while processing — body is usually `{"error": "<message>"}`. Common cases: missing required parameter, unknown `stream_id`, database disabled when CSV export is requested. A few cases (e.g. the SIP endpoints when the database is disabled) return a plain-text body such as `Database is not active.`.