VQ monitor

Sevana Ou PTE

68 Circular Road #02-01 Singapore 049422

VQ monitor

Document objectives

This document describes the application purpose, requirements, software architecture and user operations for the VQ monitoring server software from Sevana Ou.

Application purpose

The application provides access to call traffic quality measurement for cloud service deployments. Measurement is based on massive E-model MOS calculation + Sevana Ou's PVQA (Perceptual Voice Quality Assessment) technology for selected calls.

Hardware requirements

Resource	Requirement
CPU	Any modern x64 Intel/AMD CPU. On high-load systems the design favours multi-core CPUs because decoding and PVQA analysis are CPU-bound and run in dedicated worker threads.
Memory	Depends on load. 1 GB is enough for testing and low volume. For live capture on a busy aggregation interface plan few KB per active stream (for Network MOS calculation) for core
I/O subsystem	Must be reasonably balanced. The most of processing are in-memory; however if we configure the system to make the massive audio dumps - good throughput is needed.
Network	Live capture needs an interface able to receive a mirrored/SPAN copy of the production VoIP traffic (DPDK/libpcap). For HEP-only deployments a normal interface (no libpcap, no DPDK) is sufficient.

Software requirements

- Any modern 64-bit Linux distribution.

- Depending on availability of DPDK drivers - they can be used or not (good for high volume traffic).
- The server does **not** require an external database by itself. Monitoring agent doesn't use database software at all; dashboard component depends on embedded SQLite by default; PostgreSQL/MySQL are supported through SOCI for centralized multi-agent installations.
- Root privileges may be required to capture on raw sockets (typical for `--cap_net_raw` or sudo).

Operation model

The application follows the standard Linux service model.

The deployment is now split into **two cooperating processes**:

- `vq-core` — traffic capture / decoding / PVQA analysis / `.pcap` REST endpoint. Publishes per-stream and per-interval reports on a ZeroMQ PUB socket (Unix or BSD).
- `vq-db` — persistent storage (SQLite/PostgreSQL/MySQL via SOCI), CSV export, alarm processing, and the web dashboard. Subscribes to one or more `vq-core` ZeroMQ feeds.

Both processes can run on the same host or on separate hosts. Multiple `vq-core` instances on different probe machines may feed a single `vq-db` instance — this is the recommended layout for multi-agent / multi-site deployments. Only `vq-core` is mandatory — `vq-db` can be replaced easily on your side using open specifications which we provide.

Both processes:

- run in background (or attached to the console depending on configuration);
- write logs to configurable file paths (and optionally to syslog);
- cache operational data using plain text files and audio files (no proprietary storage format);
- accept `SIGTERM` for shutdown and `SIGHUP` to reopen log files for log-rotation.

`vq-core` captures traffic on one or more network interfaces (and/or via the HEP listener for HEP-encapsulated traffic from SBC/proxies), and produces reports to the system bus.

Real-time analyzer

- The RTP capturer (internally libpcap / DPDK on a NIC, and/or the HEP UDP listener) feeds RT(C)P into the decoders.
- The SIP capturer listens for SIP traffic (TCP/UDP, or HEP for TLS signalling) and reconstructs call setup. It correlates the RTP addresses and SSRC values with SIP transactions so streams can be tagged with caller/callee URIs.
- The highly optimized receiver monitor calculates Network MOS — it takes almost no computational resources.
- For selected calls/streams only: the audio decoder turns RTP payload into PCM in a sliding 680 ms in-memory window and calculates the PVQA MOS; the result is sent to the system bus.
- The `vq-db` companion subscribes, stores the results in the database, exports to CSV if configured, evaluates alarm thresholds, and serves the web dashboard.

Only analysis results are stored in the database — never raw payload — and even that is optional (`database.engine` may be left unset; `vq-db` component can be replaced by custom one).

> server architecture
> server architecture

Traffic capturer

Captures traffic on one or multiple network interfaces (via DPDK / libpcap) and/or on a HEP listener (for HEP-encapsulated mirrors from SBC/proxies), and feeds the packets into the processing pool.

The capturer supports:

- filtering by source and destination UDP port ranges (`server.src-port-range` , `server.dst-port-range`) — useful to scope captures to known RTP ranges and reduce CPU. Individual ports or sub-ranges inside that range can be carved out with `server.src-exclude-ports` / `server.dst-exclude-ports` .
- DPDK / libpcap
- HEP v2/v3 over UDP (`server.hep-port`).

Audio decoder

Decodes RTP payload into waveform (PCM) audio.

Supported codecs:

- iLBC
- AMR-WB*
- AMR-NB*
- EVS*
- G.711 (PCMA/PCMU)
- GSM (full rate)
- G.722
- OPUS
- GSM HR
- GSM EFR
- G.729

AMR-NB / AMR-WB / EVS data may be encapsulated into IuUP frames inside RTP; the decoder parses both layouts.

* Usage fees may apply from the patent holders. These codecs can be prohibited at runtime by setting `evs-enabled` / `amr-wb-enabled` / `amr-nb-enabled` to `false` in the `codecs` config section.

SRTP support

The server can analyze SRTP-protected media in two ways:

1. `media.use-srtp: true` — **encrypted-only mode.** The server treats all media as encrypted and does **not** attempt to decode the payload. PVQA analysis is disabled (no audio is available), but the server still produces:
    - Packet counts (sent / received / lost / out-of-order / duplicate).
    - Jitter (RFC 3550 §A.8 with adjacent-packet guards).
    - Network MOS estimated via the G.107 E-model from packet loss, jitter, and RTT.
    - Optional RTCP statistics when RTCP is observed.

    This mode is the right choice when SIP is not visible (so no SRTP keys are available) but transport-layer quality is still of interest. CPU usage in this mode is a fraction of the full-decode mode, which makes it suitable for very high stream counts.

2. **SDP-keyed SRTP decryption (per-call).** When the SIP capturer sees an `m=audio` offer/answer with `a=crypto:` attributes carrying SRTP keys, the stream decoder opens an `SrtpSession` for the matched RTP flow, unprotects RTP (and SRTCP) frames, and runs the full decoding + PVQA analysis pipeline. This works on a per-call basis: SDES key exchanges are extracted automatically from SIP; DTLS-SRTP exchanges are not supported for now.

The `media.rtcp-mux` option controls whether RTCP packets are expected on the same UDP port as the RTP stream (the WebRTC-style default of `true`) or on the adjacent port (`false`, classic RTP profile).

Call-quality analyzer

This component calculates a MOS estimate in two complementary ways.

Network MOS (ITU-T G.107 E-model)

The network MOS implementation is the **simplified E-model** from G.107 with codec-specific impairment parameters from G.113:

```
d_oneway = rtt / 2 + jitter + jb_delay              (ms; jb_delay = 0 here)
Id       = 0.024·d + 0.11·max(0, d - 177.3)
Ie_eff   = Ie + (95 - Ie) · Ppl / (Ppl + Bpl)        (BurstR = 1)
R        = 93.2 - Id - Ie_eff                        (clamped to [0, 100])
MOS      = 1 + 0.035·R + 7·10^-6·R·(R - 60)·(100 - R) (clamped ≥ 1)
```

Per-codec Ie / Bpl values are looked up from a built-in table (PCMU/PCMA, G.722, G.729, G.723, iLBC, GSM, AMR-NB, AMR-WB, EVS, Opus, Speex). A safe default (Ie = 0, Bpl = 25) is used for unknown codecs.

This algorithm gathers network statistics directly from the decoder component and does not require successful audio decoding — it works for both plain RTP and `use-srtp: true` (encrypted) modes. It is also very inexpensive in CPU. Streams with fewer than 10 received RTP packets are reported as MOS = 0 (insufficient data).

Sevana MOS (PVQA)

Sevana Ou's PVQA algorithm runs on the decoded PCM and detects intra-audio impairments. After splitting the audio it calculates the ratio between affected (impaired) and clean audio time slots, and predicts a MOS score that maps to the ITU-T P.800 scale. PVQA is required for the Sevana MOS / Sevana R-factor / detector report. Details are at Sevana website.

The two metrics are complementary: Network MOS reflects the transport (loss/jitter/delay/codec), Sevana MOS reflects what the listener would hear. Both are reported in every interval / per-stream report.

Network API

`vq-db` exposes the dashboard / database API on `dashboard.port` (default 9126):

Path	Description
`/stats`	Stream list (active and finished) with filtering / sorting
`/streamhistory`	Single-stream detail page
`/report`	Single interval/report row (JSON)
`/instance_list`	List of probe agents publishing to this `vq-db`
`/server_stats`	Process metrics: uptime, version, instance counters

The full network API is documented in the companion *VQ monitor network API* article.

Configuration

Both `vq-core` and `vq-db` are configured via a single YAML file (default `vq-monitor.cfg`). The command-line surface is intentionally minimal:

Flag	Purpose
`--config <path>`	Path to the YAML configuration file.
`--pcap-replay <file>`	Replay a `.pcap` as a live input source (real-time, timestamps preserved). Coexists with live capture and HEP.
`--wav-output-dir <path>`	Write decoded per-stream `.wav` files here.
`--rtp-output-dir <path>`	Write dumped per-stream `.rtp` files here.

All other tuning lives in the YAML file. A reference configuration looks like this:

```yaml
logging:
    level:                  debug               # debug / info / error / none
    core:                   vq-core.log         # vq-core log file
    db:                     vq-db.log           # vq-db log file
    console:                true                # mirror logs to stdout

server:
    daemon:                 false                # it is normal, vq-core runs via systemD usually
    capture:                                     # interfaces to capture RTP/SIP on
        - lo
        - eth0

    instance:                                    # monitoring agent identity (forwarded with reports)
        id:                 agent_1
        name:               First instance

    src-port-range:         ''                   # 'a-b' filters source UDP ports
    dst-port-range:         ''                   # 'a-b' filters destination UDP ports
    src-exclude-ports:      ''                   # ports/ranges to drop inside the range, e.g. '5060, 3478-3479'
    dst-exclude-ports:      ''                   # destination ports/ranges to drop inside the range
    hep-port:               0                    # HEP listener port; 0 (default) disables HEP
    rtp-detect-limit:       10                   # min RTP packets to count as a real stream; fewer -> ghost
    capture-sip:            true                 # also capture SIP (UDP and TCP) for correlation
    require-sip:            false                # only report RTP streams correlated to a SIP call
    capture-only-incoming:  false                # local-testing helper; not for prod
    pidfile-core:           vq-core.pid
    pidfile-db:             vq-db.pid

    thread-pool:
        live:           8                        # decoder/analyzer threads for live capture
        max-streams:
50                      # process-wide cap (all workers) on concurrent live RTP streams admitted at all; limits
CPU/memory overload
        max-streams-audio:
0                   # 0 = PVQA only on SIP track-list match; N>0 = also decode+PVQA the first N concurrent
streams when no SIP patterns are set

    save-audio-directory:                        # optional dump of decoded .wav files

media:
    use-srtp:               false                # encrypted-only mode (network MOS only if enabled)
    rtcp-mux:               true                 # RTCP muxed with RTP on same port
    skip-illegal-streams:   true                 # drop streams with unknown payload types
    check-call-end:         0s                   # analyze only the tail when calculating Sevana MOS
    report-interval:        10200ms              # multiple of PVQA chunk size (680 ms)

# It is about Sevana MOS calculation
pvqa:
    config:                 pvqa.cfg             # PVQA analyzer settings file
    license:                pvqa_server.lic      # PVQA license file

codecs:
    # Payload types are needed when SIP traffic is not present
    amr-nb-ptype:           []                   # one entry per payload type, e.g. - 112
    amr-nb-octet-ptype:     []
    amr-wb-ptype:           []
    amr-wb-octet-ptype:     []
    isac-16k-ptype:         -1                   # single payload type (scalar int); -1 disables
    isac-32k-ptype:         -1                   # single payload type (scalar int); -1 disables
    opus-ptype:             []                   # e.g. - 120-16000-1   (ptype/samplerate/channels)
    evs-ptype:              []                   # e.g. - 109/mime/wb   (ptype/mime|g192/bandwidth)
    gsm-efr:                -1

    # These are about AMR codecs licensing:
    # evs-enabled:          false
    # amr-wb-enabled:       false
    # amr-nb-enabled:       false

dashboard:                                       # served by vq-db, optional
    port:               9126
    max-streams:        40                       # rows per dashboard section
    root:               dashboard
    good-mos-threshold: 3.6                      # MOS at/above this counts as "good" (default 3.6)

database:                                        # used by vq-db only, optional
    engine:             sqlite3                  # sqlite3 / postgresql / mysql (SOCI)
```

```
    connection:         "db=vq-monitor.sqlite"
    skip-zero-mos:      true                    # skip non-PVQA streams in storage
    records-lifetime:   7d                      # purge rows older than 7 days
    csv:                vq-db.csv               # optional CSV mirror

alarm:                                          # zero or more entries
#    - name:     'low-r-factor'
#      counter:  r_factor                        # r_factor / sevana_mos / network_mos /
#                                                # duration / packet_loss / jitter /
#                                                # free_disk_space
#      limit:    70                              # threshold (semantics depend on counter)
#      interval: 1h
#      command:  ./send_alarm.sh $r_factor $sevana_mos $network_mos \
#                                $packet_loss $jitter $duration $limit
```

Configuration directives — reference

Section.directive	Meaning
`logging.level`	`debug` / `info` / `error` / `none`.
`logging.core` / `logging.db`	Log file paths for `vq-core` / `vq-db`.
`logging.console`	Also write logs to stdout.
`server.zeromq-port`	Port for the ZeroMQ event publisher (default 9125).
`server.zeromq-control-port`	Port for the ZeroMQ control (`ZMQ_REP`) socket that accepts `Command` s (default 9127).
`server.instance-report-interval`	Cadence of the `instance_stats` event (default `1000ms`). Distinct from `media.report-interval`.
`server.zeromq-bind`	Bind address for the ZeroMQ publisher. Default `127.0.0.1` (loopback only). Set to `0.0.0.0` or a specific NIC IP to allow off-host subscribers. The bus has no authentication — restrict access at the firewall.
`server.daemon`	Detach from the console. Usually `false` because `vq-core` runs under systemd.
`server.capture`	List of interfaces to capture on (`eth0` , `lo` , `any` , …).
`server.instance.id` / `.name`	Monitoring agent identity, forwarded with reports. Multi-agent dashboards key on `id`.
`server.src-port-range` / `.dst-port-range`	`start-finish` UDP port ranges (empty disables filter).
`server.src-exclude-ports` / `.dst-exclude-ports`	Comma-separated UDP ports / sub-ranges to drop even when inside the capture range (e.g. `5060, 3478-3479`). Enforced in the BPF filter (libpcap) and in software (DPDK).
`server.hep-port`	HEP v2/v3 UDP listener port. `0` disables.
`server.rtp-detect-limit`	Minimum RTP packet count for a flow to be announced as a real stream (`stream_start`). Flows ending below this are reported as ghosts. Default `10`.
`server.capture-sip`	Also capture SIP (UDP and TCP) for RTP↔SIP correlation.
`server.require-sip`	Only report RTP streams that correlated to a SIP call (non-empty call-id); uncorrelated streams are dropped entirely (no start/report/ghost). Requires `capture-sip`. Evaluated continuously, so mid-call SIP correlation still counts. Default `false`.
`server.capture-only-incoming`	Local-testing helper: drop packets whose destination IP is not bound to the capture interface. Not for production.
`server.pidfile-core` / `.pidfile-db`	PID file paths for `vq-core` / `vq-db`.
`server.thread-pool.live`	Decoder / analyzer threads for live capture. `0` = `nproc - 1`.

Section.directive	Meaning
`server.thread-pool.max-streams`	Process-wide cap on the number of concurrent live RTP streams admitted at all (audio + network-only), across every decoder worker. Back-pressure against CPU/memory overload.
`server.thread-pool.max-streams-audio`	Count-based PVQA trigger. `0` (default) = decode audio + run PVQA only for streams matching the SIP track list. `N>0` = when **no** SIP track patterns are configured, also decode + PVQA the first `N` concurrent streams (process-wide), the rest staying network-MOS-only. A non-empty track list overrides this (it is ignored).
`server.save-audio-directory`	Path to dump decoded `.wav` files. Empty = no audio dump.
`media.use-srtp`	Treat media as encrypted; produce network-only metrics (no PVQA).
`media.rtcp-mux`	RTCP is muxed with RTP on the same port.
`media.skip-illegal-streams`	Drop streams with unknown payload types; do not compute Network MOS for them either.
`media.check-call-end`	Only analyze the last *N* seconds of each call when computing Sevana MOS (`15s` , `1m` , …). `0s` disables.
`media.report-interval`	Interval between report rows. Should be a multiple of the 680 ms PVQA chunk size.
`pvqa.config`	Path to the PVQA analyzer config file (default `/etc/vq-monitor/pvqa.cfg`).
`pvqa.license`	Path to the PVQA license file (default `/etc/vq-monitor/pvqa.lic`).
`codecs.amr-nb-ptype` / `.amr-nb-octet-ptype`	AMR-NB payload types (bandwidth-efficient / octet-aligned). Used when SIP traffic is not present to negotiate them.
`codecs.amr-wb-ptype` / `.amr-wb-octet-ptype`	AMR-WB payload types (bandwidth-efficient / octet-aligned).
`codecs.isac-16k-ptype` / `.isac-32k-ptype`	iSAC payload type for 16 kHz / 32 kHz sample rates. Single scalar int each (`-1` disables), **not** a list.
`codecs.opus-ptype`	Opus spec list. Each entry is `ptype-samplerate-channels` , e.g. `120-16000-1` .
`codecs.evs-ptype`	EVS spec list. Each entry is `ptype/mime|g192/nb|wb|swb|fb` , e.g. `109/mime/wb` .
`codecs.gsm-efr`	GSM-EFR payload type. `-1` disables.
`codecs.evs-enabled` / `.amr-wb-`	Per-codec licensing gate. Set `false` to refuse decoding that codec at runtime.

Section.directive	Meaning
`enabled` / `.amr-nb-enabled`	
`dashboard.port` / `.max-streams` / `.root` / `.good-mos-threshold`	dashboard settings (port, rows per section, asset root, and the MOS at/above which a stream counts as “good”; default 3.6).
`database.engine` / `.connection`	SQC1 engine (`sqlite3` / `postgresql` / `mysql`) and connection string. Empty = no DB.
`database.skip-zero-mos`	Don’t store streams without a Sevana MOS.
`database.records-lifetime`	Auto-purge records older than this duration (`7d` , `12h` , …).
`database.csv`	Path to a flat-file CSV mirror of all rows.
`alarm[].name` / `.counter` / `.threshold` / `.command`	Threshold alarms. Counters: `r_factor` , `sevana_mos` , `network_mos` , `duration` , `packet_loss` , `jitter` , `free_disk_space` . The command runs in a shell; `$r_factor` , `$sevana_mos` , `$network_mos` , `$packet_loss` , `$jitter` , `$duration` , `$limit` , `$free_disk_space` are expanded before invocation.

Example launch:

```
./vq-core --config /etc/vq-monitor/vq-monitor.cfg
./vq-db   --config /etc/vq-monitor/vq-monitor.cfg
```

Use `export LC_ALL=C` if you see locale-related parsing errors.

To stop a process use `kill <PID>` (SIGTERM); send `SIGHUP` to reopen rotated log files.

Performance and large deployments

The current release is explicitly tuned for high-volume probes. The points below are the relevant tuning knobs in roughly the order they matter:

1. **Cap the PVQA workload first.** Audio decoding + PVQA is the CPU-intensive part of the pipeline, so bound how many streams do it. By default a stream is decoded only when its caller/callee SIP address matches the **track list**; everything else is Network-MOS-only. When you have no SIP visibility (or just want a sample), set **`server.thread-pool.max-streams-audio: N`** to decode + PVQA the first `N` concurrent streams regardless of

SIP — `0` keeps the SIP-only default, and any SIP track patterns take priority over this cap. Separately, **`server.thread-pool.max-streams`** caps the *total* concurrent streams admitted at all (audio + network-only), process-wide across all decoder workers; streams beyond it are dropped, not downgraded.

2. **Pick the right SRTP mode.** If decryption keys are not available (no SIP visibility) set `media.use-srtp: true`. The server then skips RTP payload decoding and PVQA entirely, producing network-only metrics at a fraction of the CPU cost — typical workloads see >5× more concurrent streams per core.
3. **`media.skip-illegal-streams: true`** drops streams whose payload types are not registered (the default). On mirrored aggregation links this filters out spurious traffic without consuming a worker slot.
4. **`media.check-call-end <N>`** restricts PVQA to the last *N* seconds of each call. This is helpful when only call-tail quality is interesting (e.g. for IVR or call-center hang-up analytics) and proportionally lowers CPU.
5. **`server.src-port-range`** / **`dst-port-range`** filter at the capture stage. On a busy 10G mirror this is the single biggest reduction in CPU usage.
6. **`save-audio-directory`** disables WAV dumping when empty (recommended for production). Audio dumps were the original “cache” mechanism and remain the biggest source of disk I/O when enabled.
7. **Split storage from capture.** Run `vq-db` on a separate host for sites with sustained high call rates; the ZeroMQ PUB/SUB feed between the two scales linearly.
8. **Multi-agent fan-in.** Several `vq-core` probes can publish to one `vq-db` (set `server.instance.id` differently on each). The dashboard then exposes them under `/instance_list` and the aggregator queries treat agents as a filtering dimension. This is the recommended way to scale beyond a single capture host without sharding the database.

Extension modules

Extension modules can be loaded by the server to:

- send reports to custom destinations,
- decide which calls should be analyzed (so PVQA decoding is skipped for streams known to be uninteresting).

Modules are C++ `.so` shared libraries. The path is supplied via the `extension-module` directive (see the platform-specific configuration template if your build enables it). The plugin interface is the same one used by the bundled CSV / database / ZeroMQ reporters and is documented in `source/vq_core/plugin-model.h`; see the demo plugin under `demo/` for a working example.