freeQoS technical documentation

How freeQoS works, from the router to the screen: architecture, installation and sizing, network flows, measurements, NetFlow, shaping, capacity and insights, API, security, operations, troubleshooting, and a reference of every screen and endpoint.

Contents

1. Overview

2. Architecture

3. Installation and updates

4. Routers and inventory

5. Measurements: throughput, latency, QoE

6. NetFlow traffic

7. Plans and shaping

8. Capacity and business insights

9. Interface

10. API

11. Security

12. Operations

13. Troubleshooting

14. Settings reference

15. Data model

16. Known limitations and FAQ

Appendix A. Interface reference

Appendix B. API endpoint index

1. Overview

freeQoS is an **out-of-band** QoS/QoE controller for internet service providers running MikroTik routers. It reads the routers through their API, measures what each subscriber experiences, and installs CAKE queues to enforce the plans that were sold. It never sits in the traffic path.

It covers three jobs of an ISP:

- **See**: each subscriber's throughput, latency and bufferbloat, the load on every link and node, traffic by destination and by application (NetFlow).
- **Shape**: one CAKE queue per subscriber at the plan rate, under one parent queue per link, created and maintained automatically.
- **Integrate**: an operating API (`/api/v1`), a Preseem-compatible inventory API (`/model/v1`) and a consumption API (`/usage/v1`).

Design principles

Principle	What it means in practice
Out-of-band	No box in the path: freeQoS reads and writes the RouterOS configuration; a freeQoS outage cuts no subscriber.
Simulation by default	On first start nothing is written to the routers; writing is enabled in the interface, after reading the plan.
Strict ownership	freeQoS never modifies or deletes a RouterOS entry that does not carry `freeqos:managed` **and** the identifier of its instance.
Plan first	A subscriber is only throttled once a plan is pushed for it (API or Plans page); a subscriber that is only detected is observed.
Never make things up	A missing measurement is shown as missing ("-", "no ping reply"), never as a zero or a reassuring value.
Everything automatic	Adding a router is enough: CAKE queues, NetFlow export, discovery of subscribers and links happen by themselves.

Compared with Preseem and LibreQoS

	freeQoS	Preseem	LibreQoS
Place in the network	Out-of-band (RouterOS API)	Inline (bridge)	Inline (Linux server)
Where shaping happens	On the MikroTik routers (CAKE simple queues)	On the appliance	On the server (CAKE/HTB)
Latency measurement	Active probe (ping from the NAS, in the subscriber's VRF)	Passive (TCP)	Passive (TCP, eBPF)
Inventory API	Preseem-compatible `/model/v1`	Native	Files / integrations

A direct consequence of being out-of-band: latency is measured by a ping from the router, which requires the subscriber's box to answer pings (see section 5).

Two control loops

	Slow central loop	Fast local loop
Where	freeQoS, on a management server	On the PoP router itself
Period	10 s and more	Below one second
Role	Collects, keeps the reference data, scores QoE, **sets the rate baselines** (plans, parent queues)	Would react to radio fades and latency **within** those baselines
Status	Implemented	**Out of scope**, deliberately not implemented

freeQoS is the slow loop. The closed QoE loop of section 7 does not change that: it works at the scale of minutes, over windows of several minutes.

The three bottlenecks to shape

1. **The last mile of each subscriber**, on the PoP: one CAKE queue per subscriber.
2. **The radio backhaul or sector**: one parent queue per link, at the radio's real capacity.
3. **The internet egress**: optional, on the gateway. freeQoS reads it but installs no queue there.

What an out-of-band controller can and cannot see

Signal	Inline shaper (LibreQoS, Preseem)	freeQoS (out-of-band)
Throughput per subscriber	Shaper counters	`/interface` of the PPPoE session or the subscriber's queue — **equivalent**
Throughput per site or backhaul	Shaper tree	PoP aggregation and radio capacity — **equivalent**
Throughput against plan	Yes	Yes — **equivalent**
Hierarchical shaping	HTB + CAKE on the shaper	RouterOS simple queues + CAKE, tree built from the topology — **equivalent**
RTT per subscriber	**Passive**, from the timestamps of every TCP flow	**Active** ping from the PoP, in batches
TCP retransmissions	Passive (eBPF)	**Impossible**: it requires seeing the packets
Latency under load (bufferbloat)	Measured continuously on real traffic	**Derived** by correlating the probe's RTT with the throughput of the same sample

Everything that can be read from counters is at parity. Everything that requires inspecting packets is not, and never will be from a management server: it is the price of being out-of-band, not a gap in the implementation. The interface marks such columns "n/a" rather than showing an empty value that could be read as zero.

Key notions

The rest of this documentation relies on these words. Each one is covered in detail in its own section.

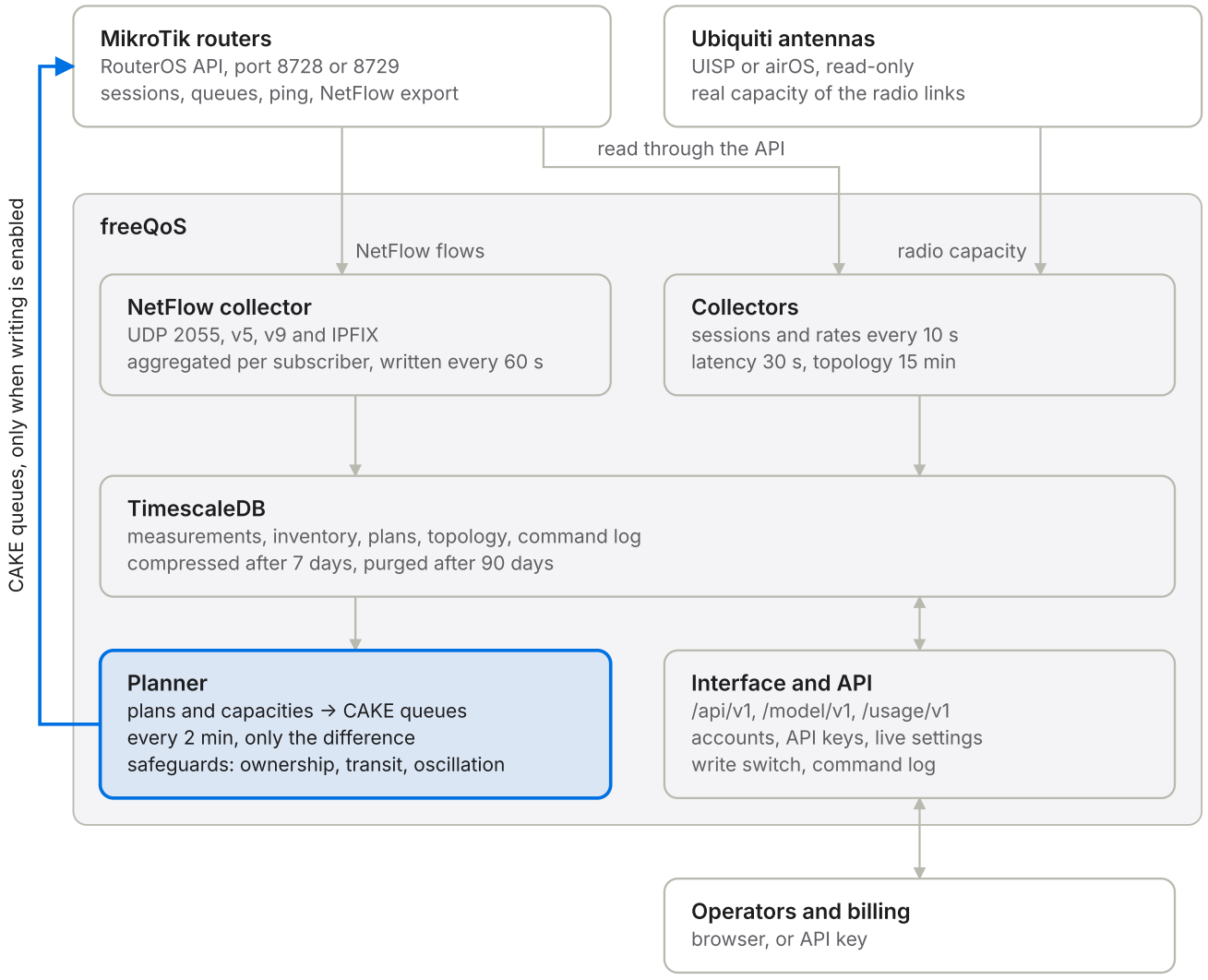
Notion	What it is	Why freeQoS cares
PoP (point of presence)	The router the subscribers of an area connect to	This is where freeQoS measures and limits each subscriber
Core, gateway	The routers between the PoPs and the internet	freeQoS reads them but never puts a limit there
PPPoE	A connection where the subscriber logs in with a username. During the session, RouterOS creates a `<pppoe-login>` interface for that subscriber alone.	That interface carries the subscriber's byte counters
CPE	The equipment at the subscriber's premises (antenna, box)	Its MAC address links the subscriber to its radio sector
Backhaul, sector	A radio link carrying the traffic of a site; a sector is an antenna serving several subscribers	Their capacity varies (weather, interference): they are often the real bottleneck
Rate / volume	A rate is a speed (Mbit/s), a volume a quantity (bytes)	Rates drive shaping, volumes drive billing
Latency, RTT	The round-trip time of a packet, in milliseconds	It is what makes a video call or a game good or bad
Queue	Where packets wait when a link is full	The longer the queue, the higher the latency
Shaping	Deliberately limiting a rate by holding packets in a queue we control	So that the queue forms where it is well managed
CAKE	A modern queueing algorithm built into RouterOS 7	It limits the rate without letting latency explode
Bufferbloat	The latency added by oversized queues when a link saturates	The first cause of a bad experience on a loaded network
QoS / QoE	QoS: the network mechanisms. QoE: what the subscriber feels.	freeQoS tunes QoS to improve QoE, and scores QoE
VRF	A separate routing table on the same router	The ping must leave in the right table, or it takes a detour
DSCP	A priority label in the header of every IP packet (46 = voice)	The measurement ping carries it to jump the queue

Notion	What it is	Why freeQoS cares
NetFlow	A summary of conversations that the router sends to a collector	It tells what, where to, and how much, without copying the traffic
RouterOS API	The programming interface of MikroTik routers	The only way freeQoS reads and writes
TimescaleDB	A PostgreSQL extension for time series	It stores millions of measurements and purges old ones

2. Architecture

freeQoS is **out-of-band**: it is never in the traffic path. It reads the routers and the antennas, stores what it measures, and installs CAKE queues **on the routers**. The routers are the ones limiting traffic. If freeQoS stops, the queues stay in place and traffic flows normally; only measurement, timed boosts and adjustments pause.

Everything is read continuously; a single path writes to the routers



Architecture: sources, cycles, database, and the single write path

Every grey arrow is a read. The coloured arrow is the only write path: it starts at the planner and goes through the safeguards of section 7.

Components

Component	Technology	Role
Application	Python, FastAPI, a single process	API, interface, scheduler, collectors

Component	Technology	Role
Scheduler	Internal asynchronous loop	Runs each cycle at its period; a slow cycle does not block the others
RouterOS client	`librouteros`, API 8728 / 8729	Reads and writes; one connection per router, with a timeout
NetFlow collector	UDP listener in the same process	Decodes v5, v9 and IPFIX, aggregates in memory
Database	PostgreSQL with TimescaleDB	Time series (hypertables) and reference data
Interface	Framework-free HTML and JavaScript, served by the application	Operating pages

A measurement, from the router to the screen

1. The `collect_subscribers` cycle reads `/ppp/active` and the interface counters of each PoP, every 10 s.
2. The difference between two reads gives a rate. Impossible values are discarded (a counter going backwards, a rate above 100 Gbit/s).
3. The rate is written to a hypertable, together with the last RTT measured by `probe_rtt`.
4. Bufferbloat and the QoE score are computed when read, over the requested window. No score is stored in advance: a change of scale applies to the whole history.

A write, from the plan to the queue

1. A plan arrives (API, *Plans* page) or a capacity changes (radio, QoE loop).
2. The planner computes the desired state of each router: queue types, parent queues, subscriber queues.
3. It reads the actual state and produces a **plan**: the list of commands that would move from one to the other.
4. The safeguards filter the plan: ownership, transit, duplicates, oscillation, circuit breaker.
5. If writing is enabled, the plan is sent command by command, parents first, and each command is logged. Otherwise it is only displayed.

An API call that changes a plan runs these five steps **immediately** for that subscriber. The 2-minute reconciliation catches up with everything else (PPPoE reconnection, new address, queue edited by hand).

Instance identity

Each installation has an identifier (`data/instance.id`, or `FREEQOS_INSTANCE_ID`). It is written into the comment of everything it creates: `freeqos:managed instance=<id>`. This is what lets two installations, or freeQoS and an operator, never overwrite each other's work.

3. Installation and updates

One command installs everything: `install.sh` sets up Docker if needed, writes a `.env` with random secrets, builds and starts the stack, then waits for the application to answer. Running the same script again updates while keeping the data.

Required resources

The figures below were **measured** on 8 October 2026 on a virtual machine with 4 vCPU and 16 GB of RAM: the real freeQoS application, the `timescale/timescaledb:latest-pg16` image, synthetic data. They give orders of magnitude; your network will refine them.

What consumes resources

Component	Idle	Under load	Note
Application (RAM)	75 MB	190 MB at 5,000 NetFlow flows/s; 575 MB after 15,000 flows/s	Memory taken at peak is not given back: size for the peak
Database (RAM)	190 MB	Grows with use	The TimescaleDB image sets its own cache to 25% of the machine's RAM (4 GB out of 16 GB)
Database journal (WAL)	512 MB	—	Fixed disk space
Docker images	2.8 GB	—	TimescaleDB 2.5 GB, application 0.3 GB

The NetFlow collector uses a single core. This is the real sizing limit:

NetFlow flows received per second	CPU (one core)	Loss
1,000	6%	0%
5,000	45%	0%
10,000	77%	3%
15,000	98%	17%

Beyond about 5,000 flows per second, datagrams are lost. Adding cores changes nothing; a faster core helps a little. To reduce the volume: export only from the internet edge (one vantage point instead of two), or enable sampling on the routers.

The number of flows per second depends on usage and cannot be guessed. To measure it, read `flows_seen` in `GET /api/v1/netflow/status` one minute apart, at peak time.

Disk

Measured per subscriber connected around the clock:

Table	Per day, raw	After compression (7 days)	Over 90 days
Subscriber measurements (`subscriber_metrics` , every 10 s)	1.2 MB	0.08 MB (6.5% of raw)	≈ 15 MB
NetFlow (`flow_metrics` + `flow_app_metrics` , per minute)	up to 0.83 MB	**no compression**	up to 75 MB

The NetFlow figure is a ceiling: it assumes traffic every minute, day and night, across three usage families. A real subscriber produces less. The two NetFlow tables are not compressed today, only purged after 90 days: they are the largest disk item.

> **How TimescaleDB stores measurements.** A *hypertable* is split into one-day chunks. After 7 days, each chunk is rewritten as compressed columns: neighbouring values of a column look alike and compress very well. After 90 days, the whole chunk is dropped at once, without scanning the table.

Recommended sizing

The same figures apply to a physical machine and to a virtual machine.

Network	CPU (cores or vCPU)	RAM	SSD disk
Absolute minimum (trial, < 50 subscribers)	2	2 GB	30 GB
Small network (≤ 100 subscribers)	2	4 GB	40 GB
≤ 500 subscribers	4	8 GB	80 GB
≤ 1,000 subscribers	4	8 GB	150 GB
≤ 5,000 subscribers	8	16 GB	600 GB

These values keep a 50% disk margin over the measured worst case (≈ 90 MB per subscriber over 90 days), plus the images and the journal. They assume the NetFlow volume stays under 5,000 flows/s; otherwise, see above. An SSD is required: the database writes continuously.

2 GB and 2 CPU run, but with no headroom. The system and Docker take about 400 MB, the application 100 to 200 MB, the database about 700 MB with its cache. That is enough to try freeQoS, not for production: a NetFlow peak takes memory that is not given back. **For production, start at 2 CPU and 4 GB.**

Analysing both directions behind NAT keeps a pairing table (see *Both directions*, section 6). Full (100,000 conversations), it takes about 76 MB, plus up to about 15 MB for downloads waiting for their upload: under 100 MB at worst, already covered by the sizes above.

Physical machine or virtual machine

Both work, with the same sizes. What changes:

	Physical machine	Virtual machine
CPU	A fast core matters more than many cores (the NetFlow collector uses one)	**Reserved** vCPUs, without over-allocation on the host: a slowed-down shared core loses NetFlow datagrams, so bytes go missing from the measurements
RAM	Nothing special	**Reserved** RAM, ballooning off: the database sizes its cache at start-up on the memory it sees
Disk	Local SSD	Virtual disk (virtio) on SSD storage of the host; avoid slow shared storage
Network	Nothing special	virtio NIC; UDP port 2055 reachable from the routers (bridged, or an explicit `2055/udp` forward if the host NATs the VM)

To know where you stand, read `flows_seen` in `GET /api/v1/netflow/status` twice, one minute apart, at peak time: the difference divided by 60 is your flows per second. Under 5,000, the sizes above are enough.

On the routers, freeQoS adds:

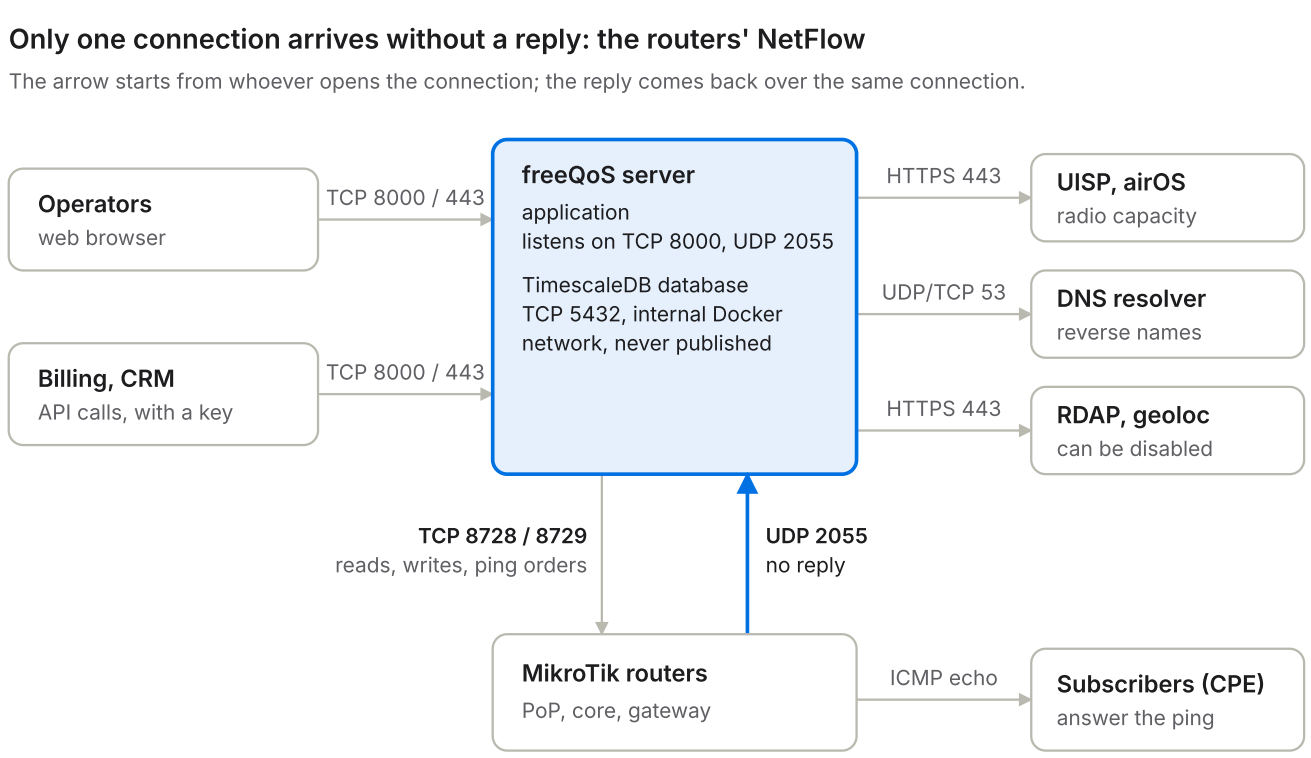
- an API read every 10 s (sessions and counters);
- 20 subscribers × 5 pings every 30 s per PoP;
- the NetFlow export, which uses router CPU.

Watch the routers' CPU load in *Devices* after commissioning.

Prerequisites

Item	Requirement
System	64-bit Linux. The script installs Docker and `git` itself on Debian, Ubuntu, RHEL, Rocky or Alma (`apt` or `dnf`).
Rights	`root` or `sudo`
Address	A fixed IP, reachable from the routers (for NetFlow) and able to reach them (for the API)
Internet access during installation	`github.com` , `get.docker.com` , Docker Hub or `mirror.gcr.io` , `pypi.org`
Routers	RouterOS 7, a dedicated account (step 3 below)
Optional	A DNS name (an `A` record at OVH or any registrar) pointing to the server, for HTTPS. The HTTPS proxy is included

Network flows



Network flows: who opens which connection, on which port

A stateful firewall only needs a rule in the direction the connection is opened: the reply is accepted as part of the connection. The only exception is NetFlow: a one-way UDP send, from the router to freeQoS, with no reply.

#	From → to	Protocol, port	Content	Needed
1	Browser → freeQoS	TCP 80 and 443 (the included proxy), or 8000 directly	Interface	Yes
1b	Let's Encrypt → freeQoS	TCP 80	Proves the domain is yours, then renews the certificate	If a domain is set
2	Billing, CRM → freeQoS	TCP 8000 or 443	API, with a key	If integrated
3	freeQoS → routers	TCP 8728 (clear) or 8729 (TLS)	Reads, queue writes, ping orders	Yes
4	Routers → freeQoS	UDP 2055, no reply	NetFlow	For traffic and `/usage/v1`
5	Router → subscribers	ICMP *echo request*, then the subscriber's *echo reply*	Latency probe; does not go through freeQoS	For latency

#	From → to	Protocol, port	Content	Needed
6	freeQoS → UISP controller	HTTPS 443	Capacity of the radio links	If `BACKHAUL_PROVIDER=uisp`
7	freeQoS → Ubiquiti antennas	HTTPS 443 (`/status.cgi`)	Radio capacity, read on each antenna	If `BACKHAUL_PROVIDER=airos`
8	freeQoS → DNS resolver	UDP and TCP 53	Reverse names of the addresses seen	Recommended
9	freeQoS → `rdap.org`	HTTPS 443	Owner of the addresses	Enabled by default, can be disabled
10	freeQoS → `ipapi.co`, `ipwho.is`, `freeipapi.com` (HTTPS 443), `ip-api.com` (HTTP 80)		Geolocation	Enabled by default, can be disabled
11	Application → database	TCP 5432, internal Docker network	Reads and writes of measurements	Never published outside the server
12	freeQoS → FreeRADIUS database	Port of the SQL database	Plans read from RADIUS	If `PLAN_PROVIDER=freeradius_sql`
13	Server → internet	HTTPS 443	Installation and updates	During installation

On the router, the firewall's `input` chain must accept TCP 8728 or 8729 from the freeQoS address. The NetFlow export leaves through the `output` chain, which is usually open.

Step-by-step installation

1. Prepare the server. A virtual machine sized according to the table above, with a fixed IP and an SSD.

2. Open flows 1, 3, 4 and 13 of the table, plus 2 if a billing system calls the API.

3. Create the freeQoS account on each router. In the RouterOS terminal:

```
/user/group add name=freeqos policy=read,write,api,test
/user add name=freeqos group=freeqos password=<strong-password> address=<freeQoS-IP>/32
/ip/service set api address=<freeQoS-IP>/32
```

`address=` restricts the account and the API service to the freeQoS address. For the encrypted API, enable `api-ssl` (port 8729) with a certificate, and tick *SSL* when adding the router. To start read-only, there is nothing to change on the router: leave *Allow freeQoS to write on this router* unticked when you add it in *Devices* (the default). freeQoS then reads and measures without writing anything, and you allow writing later, router by router, from the same page (see *Rolling out router by router*, section 7).

4. Install freeQoS on the server:

```
curl -fsSL https://raw.githubusercontent.com/tatdorian/freeQos-/HEAD/install.sh | sudo sh
```

If the repository is private, this link does not answer without authentication. Clone the repository with an access token instead, then run `sudo ./install.sh` from its folder.

Without asking anything, the script:

1. installs Docker, the `docker compose` plugin and `git` if they are missing;
2. fetches the code into `/opt/freeqos` (main branch, or `FREEQOS_BRANCH`);
3. writes `.env`: random PostgreSQL password, `APP_PORT=8000`, `NETFLOW_PORT=2055`;
4. switches to `mirror.gcr.io` if Docker Hub is unreachable;
5. moves the web proxy to 8081 / 8443 if another web server already holds 80 / 443 (written to `.env` as `HTTP_PORT` / `HTTPS_PORT`);
6. builds, starts, waits for `/health` (3 minutes at most) and prints the address;
7. prints the **first Administrator account**, which the application creates by itself on its first start (see step 6). Nothing is asked: the installation runs unattended;
8. makes freeQoS **start with the server**: Docker is enabled at boot, every container is `restart: unless-stopped`, and a systemd service `freeqos` runs `docker compose up -d` at boot, even after a `docker compose down`.

`sudo systemctl status freeqos` shows it; `sudo systemctl disable freeqos` turns the automatic start off.

5. Check.

```
cd /opt/freeqos && sudo docker compose ps
curl -s http://127.0.0.1:8000/health/ready
```

The three containers `freeqos-app`, `freeqos-db` and `freeqos-proxy` must be `running`, and `/health/ready` must answer `"status":"ready"`.

6. The first Administrator account is created by the application itself, on its first start, before it answers anyone. There is no "create the first account" screen left open to whoever reaches the server first.

- Email: `FREEQOS_ADMIN_EMAIL` (in `.env`, or in front of the install command), `admin@freeqos.local` by default.
- Password: drawn at random (`Xk7pQ-m2RtW-...`, about 114 bits, without look-alike characters), never written to the log. `install.sh` prints it at the end. It is also kept in `data/first-admin.txt` (readable only by the service), and that file is **deleted as soon as the password is changed**:

```
sudo docker compose exec app cat data/first-admin.txt
```

```
curl -fsSL .../install.sh | sudo FREEQOS_ADMIN_EMAIL=noc@example.com sh
```

This works the same with a plain `docker compose up` or on Kubernetes: nothing to run by hand. Open `http://<freeQoS-IP>/` (or `:8000`), log in, change the password from the account menu. The other accounts (Viewer, Operator, Administrator) are created in *Settings › Accounts*.

Command line, in the container: `python -m app.cli count-accounts` prints how many accounts exist, `python -m app.cli reset-password --email ...` gives an account a new random password (and closes its sessions), and `python -m app.cli create-admin --email ...` creates the first account if there is none. `FIRST_ADMIN_AUTO=false` turns the automatic creation off; the first visit then offers to create the account, as before.

7. Give it a domain name and HTTPS (recommended), from the interface:

1. At OVH (or any registrar), in the DNS zone of your domain, add an `A` record: a sub-domain such as `qos` pointing to the server's public IP. It takes from a few minutes to a few hours to spread.
2. Open TCP **80** and **443** to the server from the internet. Port 80 must stay open: it is how Let's Encrypt checks the domain, first and at each renewal.
3. *Settings › Domain & HTTPS*: type the name (`qos.example.com`), optionally an email for certificate expiry notices, then *Save and apply*.

The included proxy (Caddy, container `freeqos-proxy`) obtains the certificate within a minute and renews it on its own. `http://` on that name then redirects to `https://`. The server's IP keeps answering in plain HTTP: it is the way back in if the DNS is broken. *Check now* tells whether the name points to this server and whether HTTPS answers; the last error from the proxy is shown in plain words. *Remove* goes back to HTTP only.

Once HTTPS works, close port 8000 to the outside (or set `APP_BIND=127.0.0.1` in `.env`). API: `GET /api/v1/public-access`, `PUT /api/v1/public-access` (`{"domain", "email"}`, Administrator), `DELETE /api/v1/public-access`, `POST /api/v1/public-access/check`.

The domain is stored in the database and pushed again to the proxy at each start of freeQoS. The proxy's admin interface (port 2019) is only reachable inside the Docker network, never published. Its certificates live in the `caddy_data` volume.

`.env` variable	Default	Role
`HTTP_PORT` / `HTTPS_PORT`	`80` / `443`	Ports of the proxy on the server. The certificate needs 80 and 443.
`APP_PORT` / `APP_BIND`	`8000` / `0.0.0.0`	Direct access to the application, without the proxy
`FREEQOS_SUBNET`	`172.29.84.0/24`	Internal Docker network. Change it only if it clashes with one of yours
`FORWARDED_ALLOW_IPS`	the subnet above	Who may tell the application the visitor's real address and HTTPS

8. Add the routers. *Devices* → *Connect a router*: name, IP, the account from step 3, role (`pop`, `core` or `gateway`). *Test* checks the connection without saving anything. After adding, the automatic commissioning described in section 4 runs; follow its report.

9. Let it run in simulation, for at least one full cycle. Check the write plan in *Settings*: planned queues, skipped links and their reason. Then enable writing.

10. Schedule the backup. For example, once a night:

```
0 3 * * * cd /opt/freeqos && make backup
```

Copy the `backups/` folder off the server regularly. `make backup` does not include the instance identifier: save it once with `sudo docker compose exec -T app cat /app/data/instance.id > instance.id`, or set it with `FREEQOS_INSTANCE_ID` in `.env`.

What must be kept

File	Content	If lost
`.env`	Database password, ports	The database becomes unreachable.
`data/secret.key`	Encryption key for router passwords	Router passwords stored in the database become unreadable; they must be entered again.
`data/instance.id`	Identifier of this instance	Queues created earlier become those of "another instance" and are no longer touched.
`pgdata` volume	All measurements, the inventory, the plans	History lost.

Backup and restore: `make backup` and `make restore` (with the `timescaledb_pre_restore` / `post_restore` steps).

Kubernetes

freeQoS recognises a pod (`KUBERNETES_SERVICE_HOST`, `/var/run/secrets/kubernetes.io`). Three things to set:

- **`NETFLOW_COLLECTOR_ADDRESS`**: the address the routers must send to (UDP Service, LoadBalancer or node). Without it, freeQoS sets no NetFlow target rather than announcing the pod IP.
- **A persistent volume** for `data/` (key and instance identifier).
- **`FREEQOS_INSTANCE_ID`** for a readable, stable identifier.

RouterOS prerequisites

Item	Value
API service	`api` (8728) or `api-ssl` (8729) reachable from freeQoS

Item	Value
Account	A group with the policies `read,write,api,test` (`test` is used by the ping probe). The `write` policy can stay in place even for a router you want read-only: that is decided in freeQoS (*Devices*, per router)
NetFlow export	Set up by freeQoS itself; UDP port 2055 must be reachable from the routers
Version	RouterOS 7 recommended (VRF, `ping vrf=`, CAKE)

Updating

1. If two instances drive the same routers, stop one **before** updating.
2. Run `install.sh` again (or `git pull` then `docker compose up -d --build`).
3. On the first start of a new version, database migrations apply by themselves; the database keeps its data.

4. Routers and inventory

freeQoS only knows the routers it is given. Everything else (subscribers, links, tree, antennas) is discovered from them.

How freeQoS talks to a router

freeQoS uses a single channel: the **RouterOS API**. No SSH, no SNMP, no script installed on the router.

The protocol. The API is a TCP connection on port 8728 (clear) or 8729 (TLS-encrypted, `api-ssl` service). It carries the same commands as the router's terminal, as words: `/ppp/active/print`, `/queue/simple/add`, `/ping`. The router answers with lines of `key=value` fields, one per object. freeQoS therefore reads exactly what a technician would see in the terminal.

The rights. The RouterOS account belongs to a group, and the group grants *policies*:

Policy	What it allows	Used for
`api`	Connecting through the API	Everything
`read`	Reading configuration and state	Sessions, counters, routes, neighbours
`write`	Changing the configuration	Queues, queue types, NetFlow export, restrictions
`test`	Running `/ping` and `/tool/traceroute`	The latency probe

Without `write`, freeQoS runs read-only: measurement, latency and NetFlow (if the export was set up by hand) work, shaping does not. The same result, without touching the router, is one switch in *Devices*: the per-router *writing allowed* switch (see *The write switch and the account's rights*, section 7).

Failures. Each request has a timeout (`timeout_s`, 5 s by default). Routers are queried in parallel: a router that does not answer misses its read and reports it, the others are read normally.

Declaring a router

Two sources add up:

- **The database**, through the interface or `POST /api/v1/pops/routers`. This is the normal case; the password is encrypted with `data/secret.key` and never comes out of any response.
- **A file** (`ROUTERS_FILE`, YAML or JSON) or the `ROUTERS='[{…}]'` variable. Useful for declarative deployment; a router from the file can be hidden from the interface without touching the file.

Field	Default	Role
`name`	—	Unique identifier of the PoP
`host`	—	Management IP or name
`port`	8728	8729 for `api-ssl`

Field	Default	Role
`username` / `password`	`qos-ro`	RouterOS account
`role`	`pop`	`pop`, `core` or `gateway`; a core or gateway is never treated as a subscriber
`use_ssl`, `tls_verify`	`false`, `strict`	`strict`, `fingerprint` (with a SHA-256 `tls_fingerprint`) or `insecure`
`loopback`	inferred	Identity of the router in the tree (a /32 on a `lo*` interface, or the router-id)
`timeout_s`	5	Between 0.5 and 60 s
`pppoe_interface_pattern`	`<pppoe-{login}>`	Name of a subscriber's dynamic interface
`write_enabled`	`true` in the API and the file, **unticked** in the interface	Allow freeQoS to write on this router (queues, CAKE types, NetFlow export, restrictions). `false` = read-only: everything is read and measured, the plan is computed and shown, nothing is sent

The form of *Devices › Connect a router* sends `write_enabled` explicitly, unticked by default: a router added from the interface stays read-only until you allow it. The API keeps `true` as its default so that an existing integration behaves as before. `PATCH /api/v1/pops/routers/{id}` changes it (`write_enabled`, with `confirm: true` and an optional `reason` to allow writing); a router of the file is switched with `PUT /api/v1/pops/routers/file/{name}/write` (`enabled`, `confirm`, `reason`). The file can be overridden towards **read-only** only: a router whose file entry says `write_enabled: false` cannot be allowed from the interface.

`POST /api/v1/pops/routers/test` tries a connection without saving anything; on failure, the response gives a hint (port closed, wrong credentials, missing `api` policy, certificate).

Automatic commissioning

As soon as a router is added, freeQoS immediately runs, in order, the cycles that would otherwise run at their own pace. Each step has 90 s at most.

1. Check the account's actual write rights.
2. Read the PPPoE sessions.
3. Load the plans.
4. Read the ports.
5. Discover the topology (tree, uplink, loopback).
6. Create the CAKE queue types and the subscriber queues.
7. Set up the NetFlow export.
8. Apply the traffic restrictions.
9. Read the DNS cache.
10. Measure latency.

The report (`GET /api/v1/pops/provisioning/{router}`) then reads the router back and says what is **actually** in place. If writing is disabled — for this router in *Devices*, or globally in *Settings* — or the account lacks `write`, the state becomes `blocked`, with the action that unblocks it; steps 6 to 8 are skipped and nothing is sent. A read-only router still goes through steps 2 to 5, 9 and 10: it appears in the tree, its subscribers are measured, and its plan can be read in *Settings*. Allowing writing on it later runs the commissioning again.

Who is a subscriber on this PoP?

RouterOS has no "subscribers" table. The census therefore starts from the subnets the router serves (`/ip/address`). Each subnet is classified: point-to-point, routing transit, PPPoE server or **subscriber**.

Seven sources are then cross-checked; none can erase what another one saw.

Source	What it reveals
`/ppp/active`	PPPoE subscribers
ARP	Static-IP subscribers (the entry disappears after a few minutes of silence)
DHCP leases	DHCP subscribers, even silent ones
Bridge host table	Layer-2 presence behind a VLAN-filtering bridge
Static routes	Routed blocks (/29…) that never appear in ARP
Existing queues	Subscribers already managed
MNDP / LLDP	Neighbouring equipment

The result also states what it does not know: unreadable source, empty subscriber subnet, address outside any known subnet.

Topology

The tree is built from neighbours, shared subnets, tunnels, MAC addresses and, if configured, UISP. It is refreshed every 15 minutes. Everything can be corrected by hand and the correction survives rediscovery: move a box, change a parent, merge two boxes that are the same device, create or delete a link, correct a role.

How the tree is discovered

No single source is enough. freeQoS cross-checks seven:

Source	What it teaches
`/ip/neighbor` (MNDP, LLDP, CDP)	For each port, the device on the other side: name, model, MAC, IP. This is physical adjacency. MikroTik equipment and most radios announce themselves this way to their direct neighbours.
`/interface/ethernet`	The negotiated speed of each port: the physical ceiling of the link

Source	What it teaches
`/ip/address`	The IP network of each interface, hence who shares a segment with whom
UISP	The radio links and their current real capacity
`/ppp/active`, `caller-id` field	The MAC of each PPPoE subscriber's CPE
`/ip/arp`	Addresses present on routed VLANs without PPPoE (candidates, never subscribers)
The configuration (`/ip/route`, OSPF, BGP, VLAN, bridges)	What the routers **do** with traffic: this is what gives the hierarchy

Three rules make the tree reliable:

1. **A router is identified by its loopback**, not by its name or an interface address. Two sites configured from the same template often have the same /30 link; only the loopback is unique.
2. **The hierarchy comes from the declared role** (gateway, core, PoP), not from a guess based on the hardware model.
3. **A subscriber's sector comes from a join**: the MAC read in `caller-id` is matched against the stations known to UISP. It is the only way to know which antenna a subscriber goes through, hence its real chain of bottlenecks. Without UISP, only the PoP is known.

MAC addresses are normalised before the join: RouterOS writes `AA:BB:CC:DD:EE:FF`, UISP sometimes `aa-bb-cc-dd-ee-ff`; without normalisation, the join would fail silently. When the sector of a subscriber is unknown, its queue is created **without a parent** rather than under a guessed one: the last mile is shaped correctly, the backhaul contention is not. Attaching a subscriber to the wrong backhaul would be worse than doing nothing.

Where the hierarchy comes from

`/ip/neighbor` answers a weak question — *who sees whom* — which is also true of two devices plugged into the same switch. The configuration answers the strong ones, because it is what makes the network:

Read	What it establishes
`/ip/route`	**Who is above**: the default route says where the router sends what it cannot route
`/routing/ospf/neighbor`, `/routing/bgp/session`	A **proven** adjacency: two routers exchanging routes, not two that merely see each other
`/interface/vlan`, `/interface/bridge/port`, `/interface/bonding`	Through which **physical port** a given traffic leaves, hence which link a client hangs from

A box gets its parent from, in order of strength:

1. the parent **set by hand** in the tree editor — the operator always has the last word;
2. the parent **proven by the routing table**;

3. the **shortest path** in the graph, used only where the configuration says nothing (unmanaged devices, discovered neighbours).

Each box is marked "manual", "route" or "inferred", so you know what is established and what is assumed. A **multi-homed** router (two equal default routes) has no single parent: freeQoS refuses to pick one, says so, and falls back on the shortest path. A ring (two PoPs linked to the core *and* to each other) is where guessing goes wrong — both paths have the same length — and where the routing table settles it.

On a shared segment (a switch, a management VLAN where MNDP shows everyone), a box without a proven parent gets its **most likely** link, drawn **dashed** and marked *uncertain*, never a mesh. The panel offers to confirm it or correct it in one click.

One device, one box

The same router can be seen several times: as a managed router *and* as a neighbour of the core, under several management addresses, in IPv4 and IPv6. freeQoS reconciles these views into one box using what cannot change:

- the **serial number** (`/system/routerboard` , or the `system-id` of `/system/license` for a CHR);
- the **RouterOS identity**;
- **every interface MAC** — a neighbour only reveals the MAC of the interface facing it.

Three rules prevent wrong merges. **Two declared routers never merge** with each other. A **MAC claimed by several** routers stops identifying them (virtual machines cloned from the same image share interface MACs). The **name merges nothing**: it is unique only by convention. When identity cannot be proven, the operator decides: *Same device as…* folds one box into another, *Split* undoes it. Probable duplicates (same words in a different order, "CCR DS" / "DS-CCR") are flagged with a *Merge* button, never merged automatically. A box that groups several observations lists them.

How the loopback is found

When the loopback is not declared, it is looked for in the configuration, in this order:

Source	What is read
Loopback interface	A host address on `lo` , `lo0` , `loopback*` , `dummy0` , `bridge-loopback` , `lo-bridge` …
`router-id`	`/routing/id` , OSPF and BGP instances (structured API first, then the `/export` text)
Address comment	A /32 on any bridge, commented "loopback" or "router-id"
Isolated /32	As a last resort, and the tree says so

Uniqueness is checked, not assumed: the database refuses two routers with the same loopback, and if discovery finds two anyway, the address is left out of the index with a warning.

A subscriber's CPE is not one more device

A subscriber's box arrives by two paths: a `/ppp/active` session (the subscriber, with its login and queues) and a `/ip/neighbor` entry at the end of the PoP's port (a discovered device). freeQoS joins

them by **equality**, never by resemblance: for PPPoE, the session's `caller-id` equals the MAC announced by the neighbour (rebuilt from an `fe80::` link-local address when that is all the neighbour announces); for a routed VLAN, the client's declared address equals the neighbour's address. The subscriber is then counted once. A router of the inventory is never reclassified as a CPE, even if it opens a PPPoE session itself.

The tree follows the inventory

Adding, removing, disabling a router or changing its role **starts a new discovery within the minute**, whatever path changed the inventory (interface, API, file). A removed router leaves the tree; if the device still exists and a neighbour still sees it, it comes back as an **unmanaged** device, which is what it has become. An empty inventory erases nothing: that would be a database read failure, not a deletion. The graph never forgets on its own, so that a device briefly invisible (radio fade, reboot) does not disappear; *Forget vanished devices* removes what you choose.

A known router is never silently absent

Situation	What *Devices* shows
Secret that cannot be decrypted (key changed)	**Skipped** badge, with the reason
Invalid record (unknown role, badly typed loopback)	Same, and the other routers carry on
Unreadable database inventory	A global warning
Hidden by hand	Listed under "removed from the inventory", with **Restore**
Present but not collected, whatever the cause	**Not collected** badge

The router also keeps its box in the tree, marked "skipped": a skipped PoP is a situation to fix, and it must not look like a PoP that never existed.

A VLAN that carries clients is a site

For a radio operator, a VLAN usually carries a village, a relay or a zone; the router is only its head. freeQoS therefore turns **every VLAN carrying at least one declared or seen client** into a site of its own. A management, transit or supervision VLAN carries no subscriber and does not become a site. The name comes from the interface, i.e. from what you wrote on the router:

Interface	Site
`vlan-francophonie`	Francophonie
`vlan-zone-altair`	Zone Altair
`vlan101`, `ether1.101`	VLAN 101

Each site records the **router that serves it**: this is how shaping still finds the subscribers of a VLAN site on their router.

Clients in the tree: automatic, and editable

Every client has a place in the *Network tree*, found automatically, and the box shows how it got there (*Placed by*, in the side panel). In order:

Placed by	Meaning
placed by hand	You dragged it (or its whole group) onto a box, or used *Attach a client under this box*.
a device with its name	A discovered device has the client's name (login, name entered on the client, or label; case, accents and punctuation ignored). That box **is** the client: marked "★ client", no duplicate. A name carried by two devices links nothing. A router never becomes a client.
its own discovered box	The client's own box (static IP, or CPE recognised by MAC or address) hangs under a device: the client hangs under the same one.
its sector	Its radio sector (PPPoE caller-id, or the sector of its record).
the router of its VLAN site	Its site is a VLAN: it hangs under the router carrying it, in a "VLAN ..." box. These clients used to vanish from the tree.
its site	The box of its site.
nowhere yet	Nothing above applies: it is kept visible in **Unplaced clients**.

Clients declared by hand and never seen are in the tree too. Clients are grouped per box ("N client(s)"), click to list them.

What an Administrator can change, all undoable:

- **drag a client, or a whole group, onto any box**: they hang there. *Back to automatic place* on the client undoes it;
- on a device: **This box is a client** (choose it) links a device to a client; **Not this client** refuses a link found by name, for good;
- **Attach a client under this box** places a client from its name or login.

API: `GET /api/v1/topology` returns `placements` (login → box and how) and `client_nodes`; `PUT /api/v1/topology/client-places` (`{"logins", "node_key"}`), `DELETE /api/v1/topology/client-places/{login}`, `PUT /api/v1/topology/client-links` (`{"node_key", "login"}`, empty login = not a client), `DELETE /api/v1/topology/client-links/{node_key}`.

Static-IP subscribers

A subscriber without PPPoE (fixed IP, routed block, VLAN) is declared in *Static clients*: a name, one or more addresses or networks, a PoP. It is then measured, limited and scored like a PPPoE subscriber.

A static-IP client announces nothing: no session, no RADIUS plan. **The declaration is the only possible source**, and freeQoS treats it as the truth, like the router inventory. PPPoE and static subscribers live in the same table, distinguished by their `kind`, and follow exactly the same path for plans, overrides, boosts and audit.

Rule	Why
The reference must not encode the address (`town-hall`, not `static:120:10.0.0.5`)	The queue name derives from it; an address change must not destroy and recreate the queue, losing overrides and history
A subnet stays a subnet	A client sold a /29 has its whole block capped. A PPPoE session is always a /32
Declaring writes the queue immediately	The reply says *Queue set*, *Queue to set* (simulation), *No queue* (with the reason, usually no plan), *PoP without a router* (with the routers that exist), or *Conflict*
Only this client's queue is written	The full plan is computed (parents, CAKE types), then restricted to this queue: declaring one client never rewrites the others
A declaration never fails because a router is silent	The record is saved, the report says what could not be written, reconciliation catches up
The PoP name is matched tolerantly	Case, accents, punctuation and the word "PoP" are ignored: `PoP Francophonie`, `pop-francophonie` and `francophonie` are the same site. Two really distinct sites that would look alike are never merged: the match is "ambiguous" and nothing is written
The sector is declared	No `caller-id` exists; the *Sector* field attaches the client to its link so that it counts in that link's sharing
Measured only once a queue exists	Its bytes are read on its queue; until then it shows its plan and state, without a rate

Several address ranges for one client. A VLAN client usually has a single IP pool. When the same client has **several pools on the same VLAN** (an extra public block, a service pool), or a block routed behind its CPE, list them in *Additional ranges* (`extra_prefixes` in the API). Each item is an IP (`10.0.1.5`), a block (`10.0.1.0/28`) or a MikroTik pool range (`10.0.1.10-10.0.1.50`, split into the exact covering blocks; spaces around the dash are accepted). The main *IP address, subnet or range* field takes the same input: a range or several comma-separated IPs typed there keep their first block as the client's address and add the others to its additional ranges (editing the address alone with a range **adds** its other blocks to the ranges already declared). They all share **one** queue — RouterOS accepts a list as target (`target=10.0.0.0/29,10.0.1.0/28`) and applies a single `max-limit` to the whole list — because the client bought **one** rate; two queues would give it two. The target is canonical (sorted, a block contained in another one dropped), so reading it back in RouterOS's own order is not a change. The queue hangs under a parent link only if that link's segment contains **every** range; a range claimed by two clients makes both ambiguous, and neither queue is written. An address of any range confirms the client's presence and is never proposed as a candidate. Several **clients** on the same VLAN are simply declared separately, one record each.

Alone on its VLAN. When a single client is declared on a VLAN (and no PPPoE server listens on it), every network of that VLAN interface belongs to it — **all of them** when the VLAN carries several pools — plus its declared ranges: its queue targets the whole list, and its traffic is attributed to it in NetFlow.

Census and candidates. The census of a PoP (`GET /api/v1/pops/census`, see above) confirms declared clients — an address inside a declared block is "seen active" — and lists the addresses that

match nothing. `GET /api/v1/static-clients/candidates` returns those candidates and `GET /api/v1/static-clients/candidates/diagnostic` the reason each ARP line was set aside. A candidate is never turned into a subscriber automatically: a printer, a camera or another operator's device leaves exactly the same trace, and nobody can guess the sold rate. Declaring always goes through `POST /api/v1/static-clients` (or the *Add a client* form) with a reference and a plan.

Ubiquiti antennas and radio capacity

The capacity of a radio link changes with rain, interference and alignment. freeQoS only knows it by **asking the antenna**, in one of two ways:

- **airOS** (`BACKHAUL_PROVIDER=airos`): each antenna is polled on its local API (`https://<antenna>/status.cgi`), with no UISP needed. Antennas are added in *Devices › Add an antenna*, or **automatically**: when common airOS credentials are given (`AIROS_USERNAME` , `AIROS_PASSWORD` , set by the installer), every Ubiquiti radio discovered as a neighbour, with an address, is added and attached to the PoP of the router that sees it. Nothing is ever removed or changed automatically, and a radio already known (same address, MAC or name) is not duplicated.
- **UISP** (`BACKHAUL_PROVIDER=uisp`): the UISP controller's API, read-only, which also gives the station → access point attachments used for the sector join.

A declared backhaul is matched to the link that carries it by the **physical identity** of the radio: its UISP device id or its MAC (`uisp_device_id` , compared without case or separators). Without it, the match falls back on equal names, and failing that the parent queue keeps the port's negotiated speed — the ceiling of the ethernet cable, not of the radio. `BACKHAUL_PROVIDER=mock` is a deterministic lab simulator.

Radio health (*Devices › Radio health*) flags, per access point and per CPE:

Signal	Threshold
SNR	below 20 dB
Signal	below −75 dBm
CCQ	below 75%
Airtime	above 70% (saturated)
Capacity	below 70% of nominal: degraded (rain, interference, misalignment)
Freshness	no reading for 5 minutes: silent, capacity unknown

Wired or radio: the capacity of each link

The capacity used for a link's parent queue and for load percentages depends on its medium:

- **Wired**: the declared capacity, or the port's negotiated speed.
- **Radio**: the capacity its antenna announces live (airOS / UISP).
- **Auto** (default): the lowest known value.

Saturation risks › wired / radio… (or `PUT /api/v1/capacity/media`) declares each uplink; `DELETE /api/v1/capacity/media/{router}/{interface}` goes back to automatic. A radio link must name its antenna.

Empty sites and services waiting for an address

- **Empty sites** are removed every 10 minutes when they have neither subscriber nor backhaul and their name is declared nowhere. A site declared but still empty (router added, no session yet) is kept.
- **Services pushed without an address** (MAC only, Preseem style) or before their router is known are looked for every 2 minutes in the routers' ARP and DHCP tables; once found, they become full subscribers — placed, shaped, visible.

5. Measurements: throughput, latency, QoE

Throughput

Byte counters are read through the RouterOS API, then turned into rates by the difference between two reads.

Object	Source	Period
PPPoE subscriber	Dynamic interface `<pppoe-login>`	10 s
Static-IP subscriber	Its own queue	10 s
Port, link	`/interface`	10 s
Radio backhaul	UISP or airOS (announced capacity)	30 s

Two safeguards reject absurd values. A counter going backwards (restarted session) does not produce a spike. A rate above `MAX_PLAUSIBLE_BPS` (100 Gbit/s) is discarded.

How a rate is computed

A router never gives a rate. It keeps **counters**: the total number of bytes received (`rx-byte`) and sent (`tx-byte`) by each interface since it was created. freeQoS reads these counters at regular intervals and divides the difference by the elapsed time.

> Example: at 10:00:00 the counter reads 5,000,000,000 bytes; at 10:00:10, 5,012,500,000. Difference: 12,500,000 bytes, i.e. 100,000,000 bits, in 10 s → **10 Mbit/s**.

Three details matter:

- **Where a PPPoE subscriber's counters are.** `/ppp/active` gives the login, the address and the session uptime, but not the bytes. Those are on the dynamic interface `<pppoe-login>`. freeQoS makes two reads and links them by name.
- **Direction.** The router counts from its own point of view: what it **receives** from the subscriber (`rx`) is the subscriber's **upload**, what it **sends** to it (`tx`) is the subscriber's **download**. freeQoS swaps the two.
- **Reconnection.** When a PPPoE session restarts, the interface is recreated and its counters start from zero. If the session uptime goes backwards or a counter decreases, freeQoS writes **no** rate for that cycle. A gap in the curve is visible; a fake spike would distort averages and scores.

A static-IP subscriber has no interface of its own: its traffic mixes with that of a VLAN or a port. Its bytes are read from the counters of **its queue** (`/queue/simple` , `bytes` field). Consequence: a static-IP subscriber is only measured once a queue targets it.

The throughput of a link

A link's throughput comes from the counters of the **port** that carries it, with the same safeguards as subscribers. **RouterOS counts per interface, not per adjacency**: when a switch sits between the router and several devices, `/ip/neighbor` sees several neighbours on the same port, and giving the counter to each would multiply the total. Measurements are therefore stored per *(router, interface)*; a link inherits its port's throughput, and the interface shows a *shared* badge when several adjacencies share the port. A link declared by UISP, without a local port, has no counter and says so.

Direction. `rx` and `tx` stay those of the router: what it sends to the device opposite, and what it receives from it. Depending on whether the neighbour is upstream (gateway) or downstream (sector), the same `tx` is upload or download. The link table shows both directions without guessing; the network tree orients them towards the child (↓ = download).

Two time scales. History comes from collection (every 10 s). For "how much is going through *now*", the live measure calls `/interface/monitor-traffic` on the router, a read-only command; if it fails (version, rights, virtual port), the last collected value is returned with the reason.

Reset queues: every client back to Automatic

Devices › Reset queues deletes every freeQoS queue of the router and lays them down again from the plans. Every client of that router (sessions on it, static-IP clients, and the clients of its VLAN sites) goes back to **Automatic**: its mode, boost and forced limit are removed, it is back on its plan, and its *Activity* says so (`queues reset on <router>`). A bandwidth schedule that targets the client applies again at its next window.

A client whose usage stays well **over** its limit (25 % and more) is shown **Over its limit**, not "at its limit": the router is not holding that limit (queue missing, removed by hand on the router, or bypassed by FastTrack). *Reset queues* lays it down again.

When the queue does not see the traffic

A client's throughput is read, in this order, from **its queue** (what passed, after the cap, both ways), from **its interface** (the PPPoE interface, or the VLAN when it is alone on it), and from **NetFlow**. Both router counters are read at every cycle and compared:

- a queue that stays **flat** (under 2 kb/s, or under 5 % of the interface) while the interface carries traffic (over 64 kb/s) is **blind**, not capping: the interface takes over;
- when no router counter moves but NetFlow sees the client's traffic, NetFlow gives the throughput.

The usual cause of a blind queue is **FastTrack** on the router (`/ip firewall filter` with `action=fasttrack-connection`): fasttracked packets skip simple queues, so the queue neither counts nor caps them. Another cause is traffic that does not cross the queue's router. The client's page shows it in *Client details › Measured by*, with the reason, and the container log says `Mesure de <client> par interface : ...`. To have the cap applied too, exclude the clients' traffic from FastTrack, or disable FastTrack.

Latency: the probe

A subscriber's latency is the round-trip time measured **from its PoP** to its address. It goes neither through the core nor through freeQoS: the router does the pinging.

How a ping measures latency

The router sends a small ICMP *echo request* packet to the subscriber's address. The subscriber's CPE answers with an *echo reply*. The time between sending and the reply is the **RTT** (round-trip time).

A single ping means nothing: one packet may wait behind a download, another may go straight through. freeQoS therefore sends 5, 200 ms apart, and keeps:

- the **median** (the value displayed): it ignores a single very slow ping;
- the minimum and the maximum;
- the **jitter**: the variation from one ping to the next, which hurts voice;
- the **loss**: the share of pings left unanswered.

Why from the PoP. The ping leaves from the router serving the subscriber, not from the freeQoS server. It therefore measures the last segment, the one the operator controls and that decides the experience: radio, CPE, queue. A ping from freeQoS would add the core and the management network, unrelated to what the subscriber experiences.

Why a round-robin. Pinging 2,000 subscribers every 30 s would load the routers. Each cycle takes 20 subscribers per PoP, and the next one resumes where the previous one stopped. PoPs are probed in parallel.

Parameter	Value	Setting
Cycle	30 s, all PoPs in parallel	`RTT_INTERVAL_S`
Batch	20 subscribers per cycle and per PoP, round-robin	`RTT_BATCH_SIZE`
Pings	5, 200 ms apart	`RTT_COUNT`, `RTT_PING_INTERVAL_MS`
Priority	DSCP 46 (EF)	`RTT_PROBE_DSCP` (0 = disabled)
Freshness	A measurement older than 5 min is no longer displayed	`RTT_MAX_AGE_S`

Three choices make the measurement faithful.

> **VRF, in two sentences.** A VRF is a separate routing table on the same router: subscriber routes can live in `CUST-INET` while the `main` table serves management. A packet only sees the routes of the table it leaves in.
>
> **DSCP, in two sentences.** Every IP packet carries in its header a 6-bit label stating its class of service. The value 46 (*Expedited Forwarding*) means "voice, send first", and queues that honour it serve it before the rest.

1. Ping in the subscriber's VRF. If subscriber routes live in a VRF (for example `CUST-INET`), a ping without a table leaves through the `main` table, goes up to the core and comes back: 700 to 900 ms instead of 2 ms. freeQoS therefore picks the table from the most specific route to the subscriber:

1. a connected route first;
2. otherwise a route that does not point to the upstream gateway;
3. a VRF is preferred over `main`.

The parameter sent is `vrf=`, falling back to `routing-table=` for versions that do not know it.

2. Priority ping. The probe's packets carry DSCP 46 (EF). CAKE puts them in the *Voice* tin, with its default setting (`diffserv3`) as with `diffserv4`. They do not queue behind the subscriber's traffic: the latency measured is that of the path, not that of a subscriber saturating its plan. Only `CAKE_DIFFSERV=besteffort` cancels this effect. If the router refuses the DSCP parameter, the ping is sent again without it.

3. Slow fallback. If the median exceeds the interval between pings with at least 50% loss, the subscriber is re-tested with 3 pings 1 s apart for 10 minutes. This avoids counting merely late pings as lost.

When latency stays empty

A subscriber that has never answered is not a degraded subscriber: its CPE often filters ICMP. It is flagged `ever_answered=false`, excluded from the score and from the QoE loop, and the interface explains why. The **Find the cause** button (`GET /api/v1/rtt/diagnose`) replays the probe by hand and returns a verdict:

Code	Meaning	Action
`ok`	The subscriber answers now	The earlier silence was temporary
`no_test_policy`	The account is not allowed to run `/ping`	Add the `test` policy to the group
`rate_limited`	The subscriber answers one ping per second but drops a burst	None: the probe switches to the slow pace on this router by itself
`loopback_return_path`	The subscriber answers, but not to the router's loopback address	None: the probe now pings without a source
`no_route`	The router has no route to this subscriber	Check that the subscriber is really behind this PoP
`firewall`	Router rules drop ICMP before accepting established connections (rules listed)	Move an `accept icmp` rule up
`client_blocks_icmp`	The gateway answers, the subscriber never does	Allow ICMP echo on the CPE's WAN side
`router_cannot_ping`	The router gets no reply from its own gateway either	Check the router's output/input firewall

The response also contains `routing_table`, the `route_candidates` and a control ping to the gateway.

Latency by segment

Executive › Where the delay comes from splits latency into three segments, measured from the **same** router with the **same** method (a series of pings, median kept with min, max, jitter and loss):

Segment	From → to	What it isolates
`access`	PoP → its subscribers	The last mile: radio, CPE, queues
`gateway` (next hop up)	PoP → its default gateway	The link towards the core
`internet`	PoP → public targets (`LATENCY_INTERNET_TARGETS` , by default `1.1.1.1` and `8.8.8.8`)	The transit and the internet beyond

If the internet segment is much higher than the next hop, the delay is outside your network. `GET /api/v1/latency` returns the three segments per router; `GET /api/v1/latency/clients` returns the latency each subscriber lives.

Live check of one subscriber

On a subscriber's panel, *Measure on the router now* reads its PPPoE interface and its queue live for two seconds, adds the NetFlow rate and the last recorded sample, and returns a verdict that says **where** throughput is lost, if it is (`GET /api/v1/subscribers/{id}/live`). It is the quickest way to compare what the interface shows with what the router sees.

Bufferbloat

Bufferbloat is the latency that load **adds**. freeQoS computes it by crossing the RTT and the rate of the same sample: latency when the link is loaded, minus latency when it is not. No dedicated test is run.

Why bufferbloat exists

Every device (CPE, radio, router) has a buffer to absorb bursts. When the link is full, packets pile up in it and wait their turn. Manufacturers size these buffers for throughput, not for latency.

> Example: a 1 MB buffer on a 10 Mbit/s link. When full, it holds 8 Mbit, i.e. **0.8 s** of waiting for the last packet in. A video call whose packets sit behind a download takes those 800 ms.

An idle latency test does not see it: the buffer is only full under load.

How freeQoS computes it

Each collection cycle writes one sample per subscriber: its rate and its last RTT. Over the requested window:

1. **Idle latency** = the 20th percentile of **all** RTTs. This is the observed floor, without being fooled by a single low value.

2. **Loaded slice** = the samples whose rate is at least the 60th percentile of the subscriber's rates. If the subscriber's rate is constant, the cut is at the median.
3. **Loaded latency** = the 90th percentile of the RTTs in the loaded slice.
4. **Bufferbloat** = loaded latency − idle latency, never negative.

At least 4 samples are needed, including 2 in the loaded slice. Otherwise freeQoS gives **no** grade: showing A+ for a subscriber who never downloaded anything would be a false good result.

Grade	Added latency	Colour
A+	≤ 5 ms	green
A	≤ 30 ms	green
B	≤ 60 ms	amber
C	≤ 100 ms	amber
D	≤ 200 ms	red
F	> 200 ms	red

QoE score (0 to 100)

The score is the **minimum** of two components:

- **Idle latency:** `100 − 0.6 × (RTT − 10 ms)`. Below 10 ms, no penalty.
- **Bufferbloat:** interpolation between the points 0 ms → 100, 5 → 90, 30 → 80, 60 → 65, 100 → 50, 200 → 25 and 400 → 0.

The weakest link sets the score. A satellite link at 600 ms stays bad even without bufferbloat; a link at 8 ms that climbs to 400 ms under load is not excellent.

Each score states its basis (`basis`):

- `composite`: both components;
- `load`: bufferbloat only;
- `latency`: RTT only, a mere substitute when no load can be correlated.

Display thresholds: ≥ 80 green, ≥ 50 amber, below that red.

"At plan limit"

A subscriber filling its plan has high latency by construction. When the probe is not prioritised, its samples at the limit are set aside and the *At plan limit* badge explains it. When the probe is prioritised (DSCP 46 and CAKE), the measurement stays accurate even at the limit: nothing is set aside.

6. NetFlow traffic

Counters say **how much** a subscriber consumes. NetFlow says **what** and **where to**.

How NetFlow works

The problem. There are two ways to know what traffic is made of. The first is to copy every packet to a probe (*port mirroring*): on a 10 Gbit/s internet edge, that is 10 Gbit/s more to carry. The second is to ask the router to **summarise** what it sees. That is NetFlow: a few tens of kbit/s to describe thousands of conversations.

A flow. For NetFlow, a conversation is a *flow*: all packets with the same source address, the same destination address, the same ports and the same protocol. A video call gives a few flows; a web page, a few dozen.

The cache. The router keeps one line per ongoing flow in memory. For each packet, it adds its bytes to the flow's line. It sends nothing until the flow has *expired*:

- **inactive timeout**: no packet for 15 s, the flow is considered finished and exported;
- **active timeout**: a flow still in progress is exported anyway every minute, then its count starts again from zero.

RouterOS's default active timeout is **30 minutes**: an hour of streaming would show up as two records, half an hour late. freeQoS therefore sets `active-flow-timeout=1m` and `inactive-flow-timeout=15s` on each router.

The export. Expired flows leave in UDP datagrams towards the collector. UDP resends nothing: a datagram that does not arrive is lost for good. This is why the collector listens permanently and does no slow processing while receiving.

The formats.

Version	Principle	Particularity
v5	Fixed format, IPv4 only	Readable immediately
v9 (default set by freeQoS)	The router first sends **templates** describing the fields, then data referring to them	IPv4 and IPv6; without the template, the data is unreadable
IPFIX (v10)	Standardised version of v9	Works the same way

In practice: after a freeQoS restart, v9 data is ignored until the router resends its templates, which it does regularly. A few minutes of traffic may be missing; this is normal and counted.

Sampling. A very busy router may only examine one packet in N. It announces it, and freeQoS multiplies volumes by N.



NetFlow: from the router's cache to a subscriber's account

Step 4 is the heart of the collector: it turns "45 MB between two addresses" into "45 MB downloaded by this subscriber".

Collector

freeQoS listens on UDP port 2055 (`NETFLOW_PORT`) and decodes NetFlow v5, v9 and IPFIX. Flows are aggregated in memory then written every 60 s (`NETFLOW_FLUSH_INTERVAL_S`).

Four counting rules:

1. **Direction comes from the subscriber, not the interface.** A flow whose destination falls in a subscriber's block is download for it; one whose source falls in it, upload. Rewiring therefore distorts

nothing.

2. **The vantage point is part of the key.** The same byte crosses the PoP then the internet edge, and both export it. Counters are kept per vantage point, never added up.
3. **An unknown address is not a subscriber.** It goes into the list of "hosts seen", used only to help declare subscribers.
4. **The other end is kept.** The remote address of each flow is recorded to answer "who watches which service".

Attribution, in detail

freeQoS builds an **index of subscriber blocks**: each PPPoE subscriber's address (/32) and the networks of static-IP subscribers (/29, /30…). For each flow, it looks up the **most specific** block containing the source, then the destination. Blocks are sorted by prefix length: a lookup costs a few comparisons, whatever the number of subscribers. An internet edge sends tens of thousands of flows per minute; a slow lookup would lose datagrams.

Vantage points

The same byte crosses several routers, and each can export it. freeQoS distinguishes two vantage points:

Vantage point	Where	What it sees
`edge`	The internet edge, upstream of the core	Everything going to or coming from the internet, once
`pop`	The subscriber's PoP	The same traffic, plus local traffic, where VLAN and sector are known

Counters are kept per vantage point and are **never added up**. Consumption (Traffic page, `/usage/v1` API) is read **direction by direction**: `NETFLOW_ACCOUNTING_VANTAGE=auto` (the default) takes, for each subscriber and each window, the larger of the two vantage points for download and, separately, the larger for upload. It is the same byte seen twice, so the larger is the right figure and the sum would be wrong. `edge` or `pop` force a single vantage point for both directions. For the list of conversations, freeQoS keeps a single source per conversation **and per direction**: the one closest to the subscriber.

Both directions

Each direction is measured on its own, from end to end. Three situations used to lose one of them:

Situation	What was lost	What freeQoS does
NAT at the internet edge. The edge masquerades its clients: downloads reach its **public** address	Download at the edge: no subscriber block contains the public address	Reads the **translated** address exported by the router (IPFIX fields 225–228, 281–282), or pairs the download with its upload (below). With `auto`, downloads seen by the PoP routers count meanwhile

Situation	What was lost	What freeQoS does
Asymmetric routing. Upload leaves through one router, download comes back through another, both at the same vantage point	The direction carried by the router that saw less traffic overall	The best exporter is chosen **per direction**: download from the router that sees the most download, upload likewise. Nothing is added up
One source per conversation. The PoP exports only upload, the edge sees the download	The direction the first source did not see	The source of each conversation is chosen per direction

Translated addresses. RouterOS puts the post-NAT addresses and ports in its v9 and IPFIX records when the `nat-src-address`, `nat-dst-address`, `nat-src-port` and `nat-dst-port` fields of `/ip/traffic-flow/ipfix` are on. freeQoS turns them on itself when they are off (one `set`, under the same write switch as the rest of the export). A download addressed to the public address is then tied to the client named by its translated destination.

Pairing without NAT fields. For each upload towards the internet, freeQoS remembers *(remote address, remote port, protocol, client port)* → subscriber, kept 10 minutes after its last upload (the oldest go first when 100,000 are held). A download coming back from that remote address and port, towards that client port, belongs to that subscriber. When the router changed the client port, pairing without the port is tried only towards an address already recognised as the NAT's public address, and only when **a single** subscriber talks to that remote end: two subscribers on the same server, and freeQoS does not guess. A download that arrives before its upload in the same window is replayed when the window closes. `GET /api/v1/netflow/status` counts `nat_translated` (tied by the NAT field), `nat_matched` (tied by pairing) and `nat_unmatched` (looked like a NAT return, no upload to name it), and gives `directions`: the bytes tied to a subscriber per vantage point and per direction since start-up.

The warning on the Traffic page. When the internet exit sees more than 1 MB of upload and less than a fifth of that in download, the page says that it translates your clients' addresses, and what is counted meanwhile.

Rate of a flow

A record carries a volume and a duration (first → last packet). The rate is the volume divided by the **actual** duration. With RouterOS's 30-minute timeout, dividing by an assumed minute showed a rate thirty times too high; this is why the duration is always read from the record.

Usage families

Traffic is classified into broad families by **service port**: the smaller of the two ports, because the client picks its port at random above 32,768 and the server listens below.

Port	Family
80, 443, 8080, 8443	web
53	dns
25, 110, 143, 465, 587, 993, 995	email

Port	Family
3478, 5004, 5060, 5061	voice / video
3074, 27015	gaming
500, 1194, 1723, 4500, 51820, ESP/AH protocols	vpn
6881 to 6999	p2p
ICMP	diagnostics

Encryption sends Netflix, YouTube and the rest of the web through port 443: the port cannot tell them apart, and the family is honestly called "web". What distinguishes Netflix from YouTube is the **name** of the remote address (below).

What is left out of the list of conversations

- **Management traffic** (SNMP, BGP, BFD, RADIUS, syslog, NetFlow, Winbox, RouterOS API): the network administering itself, not a subscriber.
- **DNS queries** (ports 53, 853, 5353): asking 8.8.8.8 for a site's address is not "going to 8.8.8.8". Their volume is still counted.
- **Private addresses**: two subscribers talking to each other are not an internet destination for either of them.

At most 2,000 destinations are kept per 60-second window, so that a subscriber running p2p does not trigger a burst of writes. A destination not seen for 7 days is forgotten; its name is kept.

Export set up automatically

freeQoS configures the export on each router itself, every 10 minutes if needed (`NETFLOW_EXPORT_AUTO` , `NETFLOW_EXPORT_INTERVAL_S`):

```
/ip/traffic-flow set enabled=yes interfaces=all active-flow-timeout=1m inactive-flow-timeout=15s
/ip/traffic-flow/target add dst-address=<collector> port=2055 version=9
/ip/traffic-flow/ipfix set nat-src-address=yes nat-dst-address=yes nat-src-port=yes nat-dst-
port=yes
```

The last line is only sent when one of those fields is read as off: without them, downloads behind NAT cannot be tied to a client (see *Both directions*). A router whose RouterOS has no `/ip/traffic-flow/ipfix` menu is left as it is.

The collector address is not guessed. It is the local address the system would use to reach **this** router (a UDP socket opened towards it, without sending a packet). It is therefore correct even on a server with several interfaces.

Special cases:

- **Container (Docker, Podman, Kubernetes):** the local address would be the container's, unreachable from the routers. Without `NETFLOW_COLLECTOR_ADDRESS`, freeQoS sets no target and says so.
- **Target towards another collector:** never touched. A third-party tool already receiving the flows keeps receiving them.
- **Stale target:** a target carrying the mark of **this** instance and pointing to an old address is removed.

Writing goes through the same path as queues: simulation by default, circuit breaker, audit.

Exporters and vantage points

A router whose export was set up by freeQoS is declared as an **exporter** at the same time, with its vantage point deduced from its role (`gateway` → `edge`, otherwise `pop`). An exporter that sends without being declared still appears, marked `unknown`: a badly configured PoP must **show up**, not disappear. *Traffic › Advanced: NetFlow exporters* declares by hand an exporter that is not one of your routers, with its vantage point and its **sampling** rate. Declaring sampling matters: a router sampling 1 in 1,000 reports a thousandth of the real traffic, and nothing else would show it.

"No datagram received": the causes

The *Traffic* page names the cause and the action:

Message	What is missing	Action
The collector is not listening	The UDP socket could not open (port in use, rights)	Read the error shown; check `NETFLOW_PORT`
No router is declared	Nothing can export	*Devices* › add a router
N router(s) declared, none exports yet	The export is not set up yet	It is set up automatically at the next pass (requires writing to be enabled)
Export configured on N router(s), but no datagram reaches …	The **network path**, not the configuration	Is UDP 2055 published (`2055:2055/udp` in Docker)? A firewall between the PoP and freeQoS? Is the announced address reachable from the PoP?
Flows received, none matched to a subscriber	No subscriber block contains the addresses	Subscribers not collected yet, or clients with public addresses not in `NETFLOW_CUSTOMER_NETWORKS`

The most misleading one is the path: without the `/udp` suffix, Docker publishes TCP, the router exports, the collector listens, and nothing meets, without an error anywhere. The repository's `docker-compose.yml` publishes `2055/udp`, and a test locks it.

Orphan records

`orphan_records` in `GET /api/v1/netflow/status` counts v9/IPFIX data that arrived before its template. A counter that rises then stabilises is normal (after a restart, the templates come back within minutes). A counter that **keeps** rising means the exporter never sends its templates.

Which addresses are subscribers, which are infrastructure

- `NETFLOW_CUSTOMER_NETWORKS` (default `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`, `100.64.0.0/10` (CGNAT), `fd00::/8`): where subscribers live. If your subscribers have public addresses, add their blocks, otherwise their conversations are taken for transit.
- Infrastructure addresses are learnt automatically (the freeQoS server, the declared routers and their addresses, the exporters); `NETFLOW_INFRASTRUCTURE_NETWORKS` adds what the inventory does not know (a transit link, a monitoring network). A client pinging one of your routers is shown as reaching **your network**, not an unidentified internet address.

Naming addresses

NetFlow only carries addresses. When a subscriber reaches an address never seen before, it is recorded "to be named". A separate loop then names it, in batches of 40 every 30 s, most recent first. Naming may need a network request, hence a wait: doing it while receiving would block the collector. An address reached ten seconds ago is named on the next pass.

Source, by order of confidence	What it gives	What it costs
Catalogue	Address blocks published by the large services (Netflix, YouTube, Meta…)	Nothing: works without internet, immediate answer
Router DNS cache	The name the subscriber **asked for** (`netflix.com`), following CNAMEs	Nothing. Only if subscribers resolve through the router; a subscriber querying 8.8.8.8 directly bypasses it.
Reverse name (PTR)	E.g. `ipv4-c001-par001.1.oca.nflxvideo.net`: follows a service that changes blocks	One DNS query per new address
RDAP (registry)	Organisation, AS number, country, announced block	One HTTP call to `rdap.org` per address
Geolocation	Country, region, city	One HTTP call to `ipapi.co` (then fallback services), or nothing with a local MaxMind database

An address without a reverse name is asked again 3 times at most. A name read from the DNS cache is no longer attributed after 24 h: the address may have changed owner.

A CDN is not streaming. Cloudflare, Akamai or Fastly serve a recipe site, a video or a software update alike. They are classified `cdn`, never `streaming`, so that a restriction placed on them is a conscious choice. The interface shows the source of each name (catalogue, reverse name, registry).

Privacy. RDAP and web-service geolocation are **enabled by default**. They send to third parties the addresses your subscribers visit. To avoid it:

- `IPFINDER_GEOIP_DB` points to a local MaxMind database (`.mmdb`): same result, no outgoing call;
- `IPFINDER_GEOIP_ENABLED=false` and `IPFINDER_RDAP_ENABLED=false` turn these two sources off (changeable live);
- the catalogue, the DNS cache and reverse names are enough to recognise the large services.

Each source is isolated: a resolver that breaks, a registry that rate-limits or a geolocation service that fails never costs the verdict the others gave. **The reverse name beats the block**: an Open Connect cache hosted at your premises is in no published block, yet it carries the most traffic. An address with no reverse name is still marked resolved — "this address has no name" is an answer, and most of the internet is in that case.

What the Traffic page shows

Block	Question it answers	Where it comes from
Who consumes	Volume per subscriber over the period	The database
Which services the traffic comes from	"Who is streaming on this sector?"	The database, over the chosen period
Where the traffic goes	A world map (one circle per place, area = volume; scroll to zoom, click for details) and the volume per country, **including the unlocated share**	The database. The map background (Natural Earth 1:110m, public domain) is served by freeQoS: no tile is downloaded
Who talks to whom, client by client	One line per conversation *client* ↔ *destination*: service, port, volume, average rate, and the **live** rate for conversations happening now	The database, plus the collector's in-memory window for the live column
Destinations reached	The record of an address: reverse name, service **and on what grounds**, organisation, AS, country, city, and the named list of subscribers reaching it	The database and the catalogue
Find an IP	"This address or this domain name: who and where?" — any address, seen on the network or not	An on-demand analysis with the authorised sources only; nothing is written

Searching conversations. Four filters combine — free search (client address, login, remote address, reverse name, organisation, service), PoP, category, usage family — and every value in the table is clickable to filter on it. The lists only offer values that exist in the data shown. The **unidentified** share is shown like the others: a page that only showed what it can name would hide the largest unknown.

A machine without a subscriber record counts too. "This address reached that one" is the observation; attaching it to a subscriber is an interpretation, which may be missing (monitoring station, camera, router). Such lines appear under the client address, marked *not declared*.

Volume over time per subscriber: `GET /api/v1/netflow/subscribers/{id}/series`. **Live connections** (`GET /api/v1/netflow/connections`) read the collector's in-memory window, the only truly live view: it empties at each write and fills up again — that is not a failure.

Traffic restrictions

A restriction targets a catalogue service for one or more subscribers. Two actions:

Action	What is installed
`block`	An `address-list` for the service and two `filter action=drop` rules (one per direction)

Action	What is installed
`limit`	An `address-list`, two `mangle` marks and two `/queue/tree` queues attached to `global`, one per direction

Every 5 minutes (`RESTRICTIONS_INTERVAL_S`), the address list of each rule is recomputed and only the difference is pushed. A new Netflix address seen by NetFlow therefore joins the list by itself.

No rule is ever created automatically: throttling a service is a business decision. `GET /api/v1/traffic-rules/{id}/preview` shows what a rule targets today. IPv6 is not installed yet: IPv6 prefixes of a rule are set aside and the plan says so.

What a rule contains. A name; an effect (`block`, or `limit` with download and upload caps in kbps, Mbps or Gbps); its targets — catalogue **services**, **categories** (`streaming`, `social networks`, `gaming`, `voice / video`, `cdn`, `cloud`, `updates`, `dns`, `messaging`) and hand-entered **prefixes**; optionally a protocol (`tcp`, `udp`, `icmp`) and service ports (`443`, `6881-6999`); and **for whom**: every client, some sites (PoP or VLAN site), or some clients.

A rule is a criterion, not a snapshot. Its address set is recomputed at each pass from two sources: the **published blocks** cover servers no client has reached yet, the **discovered addresses** cover what lies outside them (a cache hosted at your premises, a server rented elsewhere). An address already inside a kept block is not added: it would change nothing and lengthen a list the router walks through **for every packet** (`RESTRICTION_ADDRESS_LIMIT`).

Writing. Saving an enabled rule, editing it or re-enabling it installs it on the routers **immediately** (still subject to the write switch, the circuit breaker and the audit). **Suspending a rule lifts it** at once; **deleting** it lifts it too. A rule without any criterion (no service, category or prefix) is refused: it would target the whole internet, and on an internet edge would cut the network. The *Last applied* column is what distinguishes a rule **saved** from a rule **installed**. `POST /api/v1/traffic-rules/apply` reinstalls everything (dry run by default).

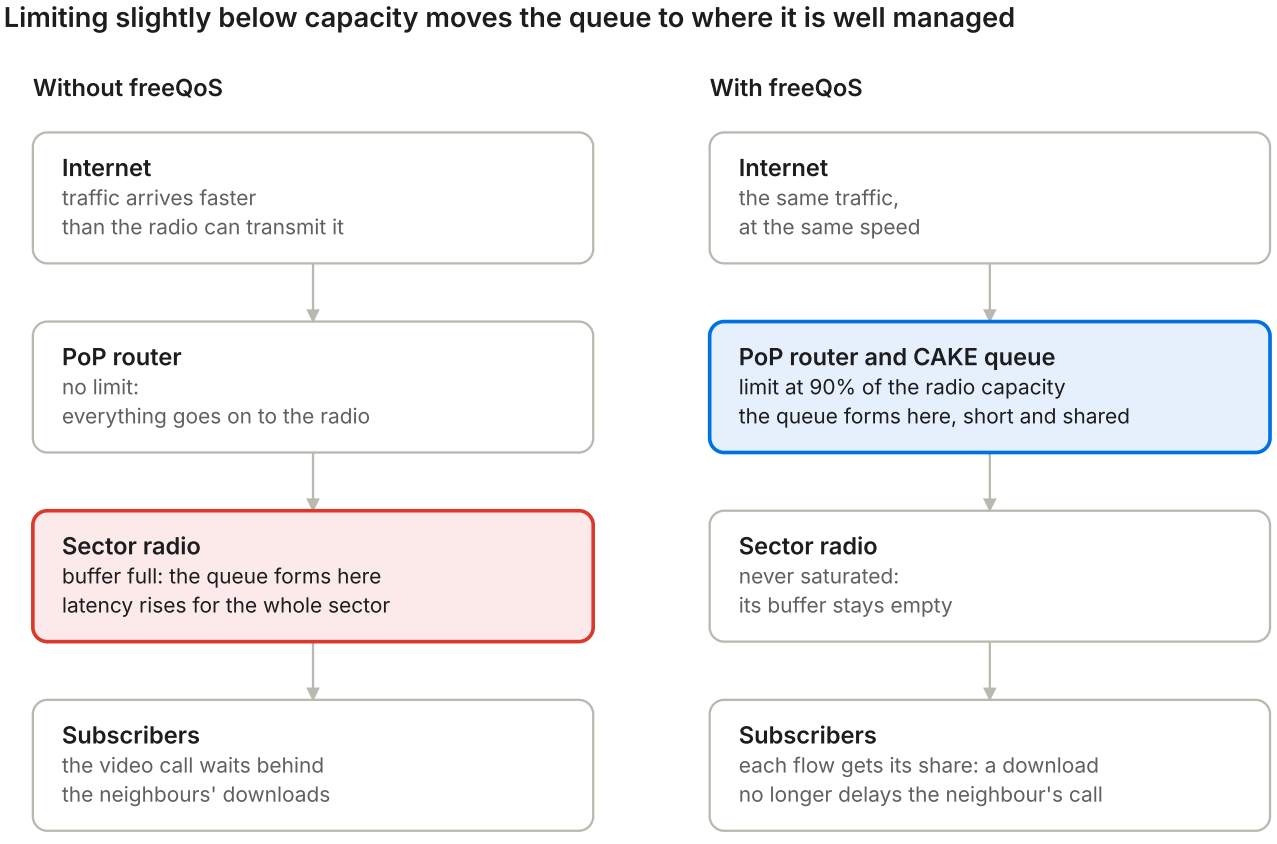
7. Plans and shaping

This is the only part of freeQoS that writes to the routers. It always follows the same path: **compute a plan, show it, apply it**, with several safeguards in between.

How shaping works

The queue always forms at the bottleneck. When a device receives faster than it can transmit, it stores the excess in a queue. That queue appears at the slowest point of the path: often a sector's radio or the subscriber's CPE. Their buffers are large and sort nothing: a video call waits there behind a download. This is the bufferbloat of section 5.

The idea of shaping. Traffic is deliberately limited **slightly below** the bottleneck capacity, at a place we control: the PoP router. The PoP then becomes the slowest point. The queue forms there, in a CAKE queue that keeps it short and shares it fairly. The radio's buffer stays empty. About 10% of throughput is given up (`SHAPING_SAFETY_FACTOR=0.90`), and latency stays low even when the sector is full.



Shaping: where the queue forms, without and with freeQoS

How a limit acts on TCP. Most traffic (web, video, downloads) uses TCP, which speeds up as long as nothing is lost. When the subscriber exceeds its limit, the extra packets wait in the queue. If the wait lasts, CAKE drops one (or marks it, with ECN). The TCP sender understands it is going too fast and slows down. The rate thus settles on the limit, without a growing queue.

What CAKE does

CAKE (*Common Applications Kept Enhanced*) is the queueing algorithm freeQoS uses. It combines several mechanisms:

Mechanism	What it does	Effect for the subscriber
Flow isolation	Each conversation has its own small queue, served in turn	A download does not block a call
Triple isolation (`triple-isolate`)	Sharing happens first between hosts, then between the flows of a host	A host opening 100 connections does not get 100 shares
Active queue management (COBALT)	Measures the time each packet spends in the queue; if it stays too long, drops or marks a packet to slow the sender	The queue stays short: a few milliseconds instead of hundreds
Priority classes (*tins*, `diffserv`)	Reads the DSCP label and serves voice before the rest	Packets marked EF go first, including the latency probe
Overhead compensation (`overhead`)	Counts encapsulation bytes (PPPoE, VLAN) on top of each packet	The limit matches the real rate on the wire
NAT mode (`nat`)	Looks at addresses before translation	Fairness between hosts works behind CGNAT
Reference RTT (`rtt`)	Sets how fast queue management reacts	50 ms suits an access network

How queues nest

RouterOS queues form a tree:

- A sector's **parent queue** caps the **sum** of its subscribers' traffic at 90% of the link's measured capacity.
- **Each subscriber queue**, attached to that parent, caps its subscriber at its plan. It has two limits, `max-limit=upload/download`, and two CAKE queue types, one per direction.

A subscriber alone on a quiet sector gets its full plan. When everyone pulls at once, the parent queue shares the available capacity. If radio capacity drops (rain, interference), the parent queue drops on the next cycle, and the queue stays at the PoP.

Where a subscriber's plan comes from

The plan belongs to the subscriber, not to the PoP: a PoP has a capacity, not a plan. It is resolved in this order:

1. **The plan written for this subscriber**, pushed through the API (billing, Preseem integration) or entered on the *Plans* page. The last write wins.
2. **A subscriber pushed without a rate** gets the default plan (`DEFAULT_PLAN_DOWN_MBPS` / `DEFAULT_PLAN_UP_MBPS`, 100/20).

3. **A subscriber that is only detected** has no plan: it is observed, never throttled to a rate nobody sold, unless `DEFAULT_PLAN_FOR_DETECTED_CLIENTS=true`.

Deleting a subscriber's plan removes its queue on the next cycle.

What is installed on the router

Two queue types: `freeqos-cake-up` and `freeqos-cake-down`. Their CAKE options come from the settings:

Setting	Default	Role
`CAKE_OVERHEAD`	22	Encapsulation bytes counted per packet
`CAKE_RTT_MS`	50	Reference RTT for queue management
`CAKE_DIFFSERV`	RouterOS default	`besteffort`, `diffserv3`, `diffserv4`, `diffserv8`, `precedence`
`CAKE_FLOWMODE`	RouterOS default	Flow isolation (`triple-isolate` recommended)
`CAKE_NAT`	RouterOS default	Essential behind CGNAT
`CAKE_ACK_FILTER`	RouterOS default	Thins ACKs on a very asymmetric link
`CAKE_WASH`, `CAKE_MPU`	RouterOS default	Clearing DSCP on egress, minimum billed packet size

One parent queue per link (`freeqos-parent-<link>`): radio backhaul or sector. Its rate is `max(5 Mbit/s, measured capacity × 0.90)` (`SHAPING_SAFETY_FACTOR`, `SHAPING_FLOOR_MBPS`). The queue thus forms in CAKE, where it is controlled, rather than in the radio's buffer.

One queue per subscriber:

- **Target:** the subscriber's address (`target=10.20.0.10/32`), never the `<pppoe-…>` interface. The interface is recreated at each reconnection, and RouterOS reverses the direction of `max-limit` on it.
- **Parent:** the link whose network contains the subscriber's address; the most specific wins.
- **Write order:** parents first. RouterOS refuses a child whose parent does not exist.

Every object carries the comment `freeqos:managed instance=<id>`.

Reconciliation

Every 2 minutes (`SHAPING_RECONCILE_INTERVAL_S`), freeQoS reads the queues in place, computes the desired state and sends **only the difference**. A queue that is already right gets no command. Only the field that changes is rewritten: a plan change only touches `max-limit`.

Safeguards

Safeguard	Effect
Simulation by default	`ENFORCEMENT_ENABLED=false` at startup. Plans are computed and displayed, nothing is written. Writing is enabled in the interface, after reading the plan. If the variable is `false` in the environment, enabling is locked.
Read-only router	A router whose *writing allowed* switch is off in *Devices* receives **nothing**, whatever the global switch says and even for a gesture made by hand (rate entered, *Reset queues*). Its plan is still computed and shown. A router added from the interface starts read-only.
Ownership	Only objects marked by **this** instance are modified or deleted. A queue from another instance or created by hand is never touched, nor set to 0/0.
Transit links	No parent queue on a routing link, towards a managed router, towards the gateway or the core, or on the uplink. The reason is shown (`skip_reason`).
Duplicate targets	Two queues that would target the same address: neither is written and a planning error is reported.
Anti-oscillation	If a line's target, parent or rate returns to a previous value (A→B→A) within 1 h, the line is frozen and reported (`frozen_lines`). *Reset queues* unfreezes it.
Circuit breaker	A plan of more than 500 actions on one router (`ENFORCEMENT_MAX_ACTIONS`) is refused as a whole: it almost always betrays a badly computed desired state (empty inventory, lost capacity). Only *Reset queues*, explicitly requested, is sent in batches.
Audit	Every command sent is logged with its author (`GET /api/v1/shaping/audit`).

Manual adjustments

- **Rate override** (`PUT /api/v1/shaping/policies`): sets the rate of a link or a subscriber, above the computed one.
- **Boost** (`POST /api/v1/shaping/boosts`): a higher rate for a duration; it is removed when it expires (checked every 30 s).
- **Dry run** (`POST /api/v1/shaping/plan`): the commands that would be sent, without sending anything.

Units. Every rate field has a **kbps / Mbps / Gbps** selector: a subscriber capped at 512 kbps is typed as such, not as `0.512 Mbps` , and displayed as "512 kbps". Internally everything is converted once, at the input, to Mbit/s. The API accepts `max_down_mbps` , `max_down_kbps` or `max_down_gbps` (same for upload, and `down_*` / `up_*` for a boost); two units for the same direction are refused as ambiguous.

Which rate wins. From strongest to weakest: a running **boost**, a permanent **override** (a hand-set rate, or the client's **bandwidth mode**), the **bandwidth schedule** in its current time slot (see below), the subscriber's **plan**. A boost goes over the override then disappears; it does not erase it. A schedule never undoes an override: a decision taken on *this* subscriber (capped for non-payment, for instance) is not lifted because the night begins.

Boosts. A duration (1 minute to 7 days), then either a multiplier of the plan (the interface offers ×2, ×3 and ×5; the API accepts more than 1, up to 50) or explicit rates, and a reason. It is written immediately if writing is allowed and **expires by itself**: a job checks expiries every 30 s and brings the queue back to

its normal rate — RouterOS knows nothing about the duration. A boost without an expiry is refused: it would be an override in disguise that never goes away.

A rate set by hand is written at once. Setting a subscriber's or a link's rate writes the corresponding queue immediately (the full plan of the router, restricted to that queue), and the reply says what was written, on which router, or what prevented it (`apply_now=false` only records the intention).

Bandwidth modes (Backup, Night…)

A **bandwidth mode** is a named rate given to a client in one click, from the **Clients** page: *Backup* when the backhaul is degraded, *Extreme backup* for a crisis or non-payment, *Night*… The operator picks a name in a list instead of typing rates (*Modes*, or `/api/v1/bandwidth-modes`).

- **Rate:** fixed (`rate_kind=fixed`) or a **share of the client's plan** (`percent` : 200 = twice the plan). A share needs the client's plan; a client without a known plan can only get a fixed mode. In the form, the unit is a switch: **kbps**, **Mbps** or **% of plan**. A rate typed in kbps is stored in Mbps (512 kbps = 0.512 Mbps), and shown again in kbps when the mode is edited.
- **Automatic** is not a stored mode: it is the absence of a mode. The client follows its plan and, when it has one, its **bandwidth schedule** — its rate then changes with the hour on its own.
- **Where it sits:** a client's mode is its permanent override (`shaping_policies.max_down_mbps/max_up_mbps` , with `mode_id` and `mode_name`). It is therefore above the schedule (a client in *Backup* stays in *Backup* when the night begins) and below a running boost. Back to Automatic removes the override and keeps a running boost.
- **Modes that stop boosts** (`disables_boost` , API only): entering the mode clears the running boost, and a new boost is refused (`409 Boost is unavailable in <mode>. Change the mode first.`). The form no longer offers this option: a mode saved from the form has it off.
- **Its clients:** the mode form itself has a list of clients to tick (*Clients in this mode*), with search and a site filter: saving the mode also puts the ticked clients in it and puts the unticked ones back to Automatic — for a new mode as for an existing one. On the *Modes* board, *Add or remove clients* on a mode opens the same list.
- **Editing a mode's rate** re-applies it at once to every client in it. A mode given to clients cannot be deleted (`409`): put them back to Automatic or in another mode first.
- **Writing:** like a hand-set rate, a mode change writes the client's queue at once (when writing is allowed) and the reply says what was written.
- **Changing a client's plan keeps its mode and its boost:** only a manual limit (an override without a mode) is lifted, so that the new plan takes effect. A mode expressed as a share of the plan follows the new plan at once.

Client actions (`POST /api/v1/clients/actions` , `{"logins": [...], "action": "mode" | "boost" | "stop_boost", ...}`) act on one or several clients. The reply gives a result **per client** (`succeeded` , `failed` , `results[]`): a partial failure never looks like a complete success. It needs the Operator level, for one client as for several.

Boost or temporary plan, from the Clients page. On one or several selected clients (or from a client's page):

- **Duration:** a quick choice (30 min, 1 h, 4 h, 1 day, 7 days), a **custom** value in minutes, hours or days (up to 365 days), or **until stopped** (`boost_until_stopped: true`): it runs until someone clicks *Stop*

boost (`action: "stop_boost"`). Internally it still has an expiry, five years away, so that the expiry check and the planner treat it like any boost; the interface shows it as "until stopped".

- **Rate:** the client's **current rate multiplied** by any factor above 1 (up to ×50, `boost_multiplier`). The current rate is the one in force without the boost, by the same rule as the planner: its mode or manual limit, else its schedule slot, else its plan. A client held at 550 kbps by a mode gets ×1.5 = 825 kbps, not ×1.5 of its base plan. The dialog shows the current rate and the result before applying. Or a **temporary plan**: explicit download and upload rates (`boost_down_mbps`, `boost_up_mbps`), higher **or lower** than the plan. At the end of the duration the client goes back to its plan, its schedule or its mode on its own.
- **When the boost ends** (stopped, or its time is up), the client goes back to exactly what it had before: its mode, its schedule or its plan. The boost never erases the mode.
- The log records what and for how long ("x1.5 of the current rate (550 kbps → 825 kbps down, …) for 4 h", "temporary rate 5 / 2 Mbps for 3 days", "… until stopped").

Clients page, map, alerts and reports

Clients page. Four cards on top: *Clients* (and the number of sites), *Connected* (with a bar, green above 80 %, orange from 50 %, red below), *In a mode*, *Boosts*. A card is also a filter: tap it to list those clients. For each client, **Usage now** is a bar of its download rate against its limit, with a word: *Quiet* (under 10 %), *Normal*, *Busy* (60 % and more), *At its limit* (90 % and more). The IP address is not shown in the list: it is on the client's page (search by IP still works).

Made for **hundreds of clients** (20 sites of 20 clients): search, filters by site, connection and mode, **sort** (name, site, not connected first, highest usage), and **pages** of 25, 50, 100 or 200. Sorted by site, each site gets a header with its count, its connected clients and a *Select the site* button. The box at the top of the list selects every client matching the filters, on every page.

- **Traffic** (with clients selected): the traffic of the whole selection as a chart, then each client with its own small chart, its current rate and its volume, over 1 hour, 24 hours or 7 days.
- **Export**: the list as filtered, every page, in two formats. **Excel (.xlsx)**: a formatted workbook — coloured header, columns at the right width, filters, frozen header row, real numbers and dates, status coloured (green online, red offline, grey never seen), use of the limit coloured, and a second sheet summing up each site. **CSV**: a plain table whose separator follows the browser's language (`;` and a decimal comma in French, so that Excel opens it in columns). Columns: client, status, last seen, site, kind, IP address, mode, what sets the limit, boost and its end, download and upload now, use of the limit, limit, plan, latency, session.
- **+ Add a client** (Administrator): declares a static-IP or VLAN client (address, subnet or range, additional ranges, plan, CPE MAC) and lists the declared clients and the clients by VLAN. It replaces the former *Subscribers* tab, whose content is now on this page.

Client page. *Status* (online or offline, since when), *Usage now* (gauge and word against its limit), *Limit* (and where it comes from), *Experience* (score out of 100 with a word, the latency and the slowness under load — or the latency alone when no score is measured). Then the usage chart, a **Diagnosis**, the mode and boost, the connection (IP, site, session), **What it uses** and the recent changes.

The **Diagnosis** answers "why is it slow?" in three lines, each with a coloured dot (green, orange, red; grey when not measured, with the reason):

- **Radio signal** of the client's antenna (CPE), found by its MAC (PPPoE caller-id, or the CPE MAC declared for a static client) among the antennas freeQoS reads: *Good* from −65 dBm, *Fair* down to −75 dBm, *Weak* below, with the access point, the SNR and the link quality.
- **Its site**: the current load of the site against the radio capacity of its backhaul: *OK*, *Busy* from 70 %, *Saturated* from 90 % — every client of the site is then slowed, not only this one.
- **Limit on the router**: whether the cap is really held (*Applied*), or *NOT held* and why (FastTrack, a queue hidden or disabled), from the background check of the routers.

What it uses: the volume downloaded and uploaded over 24 hours and 30 days, and the five main uses over 7 days (streaming, social networks, web, calls, gaming…) from the traffic flows. All of it comes from one call, `GET /api/v1/clients/{login}/health`, open to every level.

Map (`#/map`, every level). The sites **and the clients** on an OpenStreetMap map. A site is a pin: its size follows the number of clients, its colour the share connected (green 80 % and more, orange 50–80 %, red below). A client is a dot (green online, red offline, grey never seen): at street zoom (or when its site is selected), each client appears at its own position — a small square, linked to its site — or, without a position, in a ring around its site. Moving is smooth: drag (the map keeps a little momentum when released), pinch with two fingers on a phone, the mouse wheel, the + / − buttons, a double click, or the arrow keys. A search box finds a site or a client (by name or IP). Tap a site: its counts, its clients (offline first), *Show its clients*, *PDF report*. Tap a client: its status, site, mode, position, *Open the client*.

An **Administrator** gives a position (*Position* / *Place it*) in three ways: **type or paste coordinates** (`13.5137, 2.1098`, or a Google Maps / OpenStreetMap link), **use the GPS** of the phone (on site, over HTTPS), or **tap the map**. *Remove from the map* takes it off. API: `GET /api/v1/site-locations`, `PUT /api/v1/site-locations/{site}` and `GET /api/v1/client-locations`, `PUT /api/v1/client-locations/{login}` (`{"latitude", "longitude"}`), and their `DELETE`. The map tiles come from `tile.openstreetmap.org`: without internet, the pins stay usable on a plain background.

Alerts and error codes. Every alert carries a stable **code**, the clients concerned and **what to check**:

Code	Level	Meaning	What to check
`ERR_POP_TIMEOUT`	critical	Several clients of a site (at least 2, and half of those measured) went offline together	The site's shared link: backhaul radio, power, site router
`ERR_ROUTER_WRITE`	critical	The last change of a client failed on the router (last 24 h)	The router's API access on the Devices page
`ERR_CLIENT_DISCONNECT`	warning	One client offline, its site otherwise up	Client side: power, antenna alignment, cable, PPPoE login
`ERR_HIGH_LATENCY`	warning	Latency of 150 ms or more right now	Radio signal and the site's load
`ERR_NO_ROUTER`	warning	The last change is saved but no router carries the client	The site's router, or the client's session
`ERR_QUEUE_CONFLICT`	warning	A queue set by hand on the router targets the client	Remove it, or let freeQoS manage it
`ERR_NEVER_SEEN`	info	Declared but never measured	Not connected yet, or a wrong address / login

Code	Level	Meaning	What to check
`MODE_CRISIS`	info	Clients in a mode that stops boosts	Put them back to Automatic when it is over

PDF report. *Alerts › PDF report* (all sites, or the site chosen in the list) and *Map › a site › PDF report* (audit of one site) download a PDF made in the browser (no server, no internet needed): the counts, every alert with its code and what to check, the sites, the offline clients with their IP and last contact, for a site all its clients, and the list of codes.

Editing a client, and adding one by hand

Discovery and hand entry live side by side; neither erases the other.

- **What discovery gives** (PPPoE sessions, the static inventory): login, IP address, site, session, traffic. It is refreshed at every cycle and is not edited by hand for a PPPoE client: the next session would rewrite it.
- **What only a person knows**, entered with **Edit client** on the client's page (Operator and above): name, phone, email, address, CPE MAC, note. It is kept in its own table (`client_profiles`) and never touched by discovery. The CPE MAC entered here takes precedence for the radio diagnosis, when the PPPoE caller-id does not give one.
- **A static-IP client** also has its network part editable there by an Administrator: IP or subnet, VLAN, site.

The client's page shows its **IP address** in the header (click to copy) and its details in *Client details*. The list shows the name when one is set, with the login and the IP below it; the search finds a client by its name too.

Add a client (Clients, **+** button, Administrator) offers two ways:

- **PPPoE login**: declare a client before it connects: login, site, optionally a name, a plan and the CPE MAC. It shows as *never seen* until its first session; discovery then joins it **by its login** and the details entered are kept.
- **Fixed IP / VLAN**: a client without a session, measured on its address (as before).

Every change is written to the client's *Activity*. API: `GET /api/v1/clients/{login}/profile` (discovered facts and details), `PUT /api/v1/clients/{login}/profile`, `POST /api/v1/clients` (`{"login", "pop_name", "display_name", "cpe_mac", ...}`), `GET /api/v1/client-profiles`.

Ubiquiti UISP (antennas)

Devices › Ubiquiti UISP connects the **UISP controller** (formerly UNMS), which knows every airMAX / airFiber / LTU antenna of the network. An **Administrator** gives its address (`https://uisp.example.net`) and an **API token** with read access (UISP › Settings › Users › API tokens), then *Test the connection* and *Save*. The token is stored **encrypted** (same vault as router passwords) and never comes back through the API. Unticking *Verify the certificate* accepts a self-signed UISP.

The section then lists the antennas UISP knows: model, role, IP, UISP site, status, **radio capacity** down and up, **signal** and **airtime** — the potential of each link. **Attach** a backhaul antenna to its site: its capacity is read every backhaul cycle and becomes the ceiling of the site's queue, like an antenna

added by its own address (*Add an antenna*). API: `GET/PUT/DELETE /api/v1/uisp`, `POST /api/v1/uisp/test`, `GET /api/v1/uisp/devices`, `PUT/DELETE /api/v1/uisp/links/{device_id}` (`{"name", "pop_name"}`). The environment variables `BACKHAUL_PROVIDER=uisp`, `UISP_BASE_URL`, `UISP_TOKEN` keep working for a controller declared in the file.

UISP traffic, in the network

Once UISP is connected, freeQoS reads **every minute**, for each device UISP manages: the throughput of its radio port (or of its busiest port), its radio capacity, its signal, whether it is online, and, for an access point, the **stations** (CPEs) hanging on it with their MAC, IP, signal and traffic. The last reading is shown in the table (*Traffic now*: towards the clients, what an AP sends and what a station receives); each row is kept 7 days (`uisp_device_metrics`, `GET /api/v1/uisp/devices/{id}/history?minutes=`).

It goes into what freeQoS already shows:

- **Network tree.** Every UISP device is a box of the tree, and each station hangs on its access point. Where no router port measures the link to a UISP device, the link shows the throughput UISP reads (`uisp_rates` in `GET /api/v1/topology`).
- **Place it by hand.** *Place* (in the table, Administrator) says under which box a device is connected, and whether it is a client's CPE. Dragging the box in the tree does the same. The automatic placement stays for everything else, and the choice is undone with *Automatic*.
- **Clients.** A station whose MAC or IP is a client's gives that client's *Radio signal* diagnosis. The list of stations of an AP names the client of each CPE.
- **Email alerts.** An access point, or a device attached to a site, that UISP reports offline is a problem that can be emailed (*Settings › Email alerts*).

API: `GET /api/v1/uisp/traffic` (the reading, with each device's box, place set by hand, client and stations).

Email alerts

Settings › Email alerts (Administrator) takes the **SMTP account** freeQoS sends from: server, port and security (STARTTLS 587, SSL/TLS 465, or none for a local relay), user name and password (stored **encrypted**, never returned), sender and recipients. *Send a test email* checks it at once; a failure says why in plain words (login refused, wrong port, host unknown, certificate…). Gmail and Microsoft 365 need an **app password**.

Every minute the server looks for failures, whether or not anyone has the interface open:

Problem	When
Router not answering freeQoS	Its sessions could not be read for more than a minute
Site down	At least 2 clients, and half of those seen today, dropped together (not counted when the router itself does not answer)
Measurement failing	A collection cycle has failed for three of its periods
UISP device offline	An access point, or a device attached to a site, is reported offline by UISP

Problem	When
Limit not held	(off by default)

A problem is emailed **once**, when it has lasted the chosen delay (3 minutes by default), and once more when it is **resolved**; the problems of a minute go in one email. The section lists the recent problems and whether they were emailed. API: `GET/PUT/DELETE /api/v1/notifications/smtp`, `POST /api/v1/notifications/test`, `GET /api/v1/notifications/history`.

Bandwidth schedules (day, evening, night)

Many operators sell rates that change with the time of day: a plan reduced in the evening peak, opened at night, normal during the day — and switched **automatically**, on time. A **bandwidth schedule** (*Schedules*, or `/api/v1/bandwidth-schedules`) is a set of **time slots**, each with the rate that applies during it. *Applies to* is a segmented control (*All*, *Clients*, *Sites*, *Plans*); except for *All*, it opens a **list to tick**, with search and *All* / *None*. Each slot is one card: its name, the hours (*18:00 → 23:00*), the rate with a **% / Mbps / Mode** switch, and the days as round pills (tap to turn a day off). The hours are not typed: **tap** an hour to open a wheel (hours, then minutes by 5), or **drag it up or down** to move it by a quarter of an hour (up = later, down = earlier); the mouse wheel and the arrow keys work too. **Time zone** is a menu that starts on the browser's own time zone and shows the time it is there; the list of schedules shows each schedule's next switch in **your** local time. The rate is either:

- a **percentage of the plan** (`percent`: 200 = twice the plan at night, 70 = 30 % less in the evening), which follows every subscriber's own plan without copying it; or
- a **fixed rate** (`down_mbps` / `up_mbps`; a single direction is allowed, the other one keeps its plan); or
- a **bandwidth mode** (`mode_id`: *Backup*, *Night*… the same modes you give a client by hand). With *Applies to › Clients*, this puts one or several chosen clients in a mode during the slot. The schedule keeps a **reference** to the mode, not a copy of its rate: edit the mode on the *Modes* page and every slot using it follows at once. A mode used by a schedule cannot be deleted until those slots are changed.

Outside every slot, the normal plan applies.

Field	Meaning
`start`, `end`	Local `HH:MM`. An end before the start crosses midnight (`22:00`–`06:00`); `24:00` is midnight at the end of a slot (`00:00`–`24:00` = the whole day)
`days`	The days on which the slot **starts** (`mon` … `sun`; empty = every day). Friday's night runs into Saturday morning
`timezone`	IANA name of the subscribers' time zone (`Europe/Paris`, `Africa/Abidjan`…). The hours are local to it and daylight saving time is handled; the interface pre-fills the browser's zone
`scope`, `targets`	`all` clients; `pops` (site names, matched like everywhere else, case and accents ignored); `plans` (the plan's **download rate** in Mbps: `20` = every 20 Mbps plan); `subscribers` (PPPoE logins or static-client references)

Two slots of one schedule may not overlap — which one would win at 19:00 between "evening 18:00–23:00" and "peak 19:00–21:00"? The overlap is refused when saving, with the day and time where it

happens.

One schedule per subscriber. When several schedules target the same subscriber, the most **precise** one governs it alone: named subscribers, then plans, then sites, then all clients; at equal precision, the most recent. Schedules never combine: the operator can always say, subscriber by subscriber, where its rate comes from. The *Clients* button of a schedule (`GET /api/v1/bandwidth-schedules/{id}/preview`) lists the subscribers it governs, their plan and their rate **right now**, and those a more precise schedule takes away from it.

The switch is automatic. A job re-reads the schedules every 30 s and compares, for each schedule, the slot running now to the one of the previous pass: as soon as a slot starts or ends, the queues are re-planned and written (author `system:schedule` in the audit), without waiting for the 2-minute reconciliation — which, in any case, writes the scheduled rate too. Saving, suspending or deleting a schedule applies it at once. The same safeguards as reconciliation apply: nothing is written while writing is off, and a scheduled switch never deletes a queue. The oscillation guard does not watch `max-limit`, so three switches a day are never mistaken for an oscillation. The last switch (when, why, how many writes, on which routers) is shown under the list.

The subscriber list shows a scheduled rate with a *schedule* badge (the schedule and slot in the tooltip), and the *Plans* page shows, under each plan, the rate applied right now and the governing schedule.

The shaping map

Settings › Shaping and writing to the routers shows the **map**, not the commands: where the network is capped, at how much, and where that cap comes from (measured capacity, override, plan, bandwidth schedule, boost, QoE tightening). `GET /api/v1/shaping/points` returns it, read-only.

State of a point	Meaning
Queue set	The queue is on the router, as planned
To set	It will be written on the next pass (or once writing is enabled)
No queue	The planner skipped it, with its reason: no rate to apply, address claimed twice, link disabled by hand…
Conflict	A third-party queue already targets this address; RouterOS would only apply the first
Manual queue	A queue set by the operator, without `freeqos:managed`: shown because it caps, **never** modified

Points without a queue are listed like the others: a map that only showed what works would leave the rest to be found nowhere. The raw details stay available under the map: analysis of what exists (`GET /api/v1/shaping/state`), the plan computed on demand, and the command log.

Where a subscriber queue points, and when it is skipped

The subscriber's address is **read on the router when the plan is computed**, never taken from the database: a stored address may be one cycle late, and if the subscriber reconnected in between, the pool may have given its IP to a neighbour. It is written in canonical form with its prefix (`10.20.0.12/32`):

RouterOS always rewrites a bare address that way, and sending it bare would produce a difference at every cycle.

Situation	What happens
Subscriber **offline**	No queue. Writing on its last known address would cap whoever got it next
Subscriber **reconnected** on another IP	`set target=…` on the existing queue: the queue name does not depend on the address
Two subscribers on the same address	Neither queue. One of the two is stale, nobody knows which, and RouterOS would only apply the first

Each skipped subscriber is listed with its reason. **Automatic writes never delete**: boost expiry and the immediate write of a cap add queues, they never remove any. Without this rule, a momentary `/ppp/active` failure would make everyone look offline and erase the queues of a whole PoP. Only a plan read in the interface (or *Reset queues*) can delete. `SUBSCRIBER_QUEUE_TARGET=interface` restores the old behaviour (queue on the `<pppoe-…>` interface) for a network that depends on it; it is not recommended.

Are the caps actually held?

Three different questions are often confused: what freeQoS **wants** to install (the plan), what it **wrote** (the log), and what the network **applies**. Only the third is felt by the subscriber, and on RouterOS a queue can exist, carry the right rate, read without error — and cap nothing:

Cause	What you see	What happens
FastTrack	A normal queue whose counter does not move	`action=fasttrack-connection` lets established connections skip the rest of the path, **simple queues included**. Enabled by default in RouterOS's factory firewall
Hidden queue	Two queues, each with its rate	RouterOS only applies the **first** queue matching a target; the next ones are decoration
Disabled queue	A perfectly readable rate	`disabled=yes` caps nothing
Rate mismatch	Interface and router disagree	Cap changed in the database, never pushed

A background job checks these four causes **on the routers**, queue by queue, every 3 minutes (`GET /api/v1/shaping/limits` reads the result without waiting for the routers). It feeds *Settings › Are the caps actually held?* and the *Limit* column of the subscribers, where a cap the network does not hold shows **NOT HELD** with its cause. freeQoS **does not touch the firewall**: FastTrack is a performance decision that is not its own. It names it and gives the line to paste (`/ip firewall filter disable [find action=fasttrack-connection]`). It does fix what belongs to it: a queue disabled by hand is **re-enabled** by reconciliation, and a queue hidden by another is reported as a conflict.

The write switch and the account's rights

freeQoS writes to a router only when **all three** agree:

1. the **global switch** (*Settings › Shaping and writing to the routers*) is on;
2. the router's own **writing allowed** switch (*Devices*, list of polled routers) is on;
3. the RouterOS **account** has the rights (below).

The global switch turns writing on or off **without a restart**; the database keeps its state. Turning it on asks for a confirmation and a reason, recorded in the log; turning it off is immediate. `ENFORCEMENT_LOCKED=true`, or `ENFORCEMENT_ENABLED=false` in the environment, forbids turning it on from the interface.

The **per-router switch** works the same way: turning it on first shows what freeQoS would write on that router, then asks for a confirmation and a reason; turning it off is immediate. Each change is recorded in the command log (`GET /api/v1/shaping/audit`, verb `switch`) with its author and its reason. While it is off, every write path skips the router — reconciliation, CAKE queue types, NetFlow export, restrictions, boosts, bandwidth schedules, the closed QoE loop, a rate written at once, *Reset queues* and commissioning — and nothing is sent, not even for a gesture made by hand. Reading, measurement, the latency probe and NetFlow collection go on as usual. The router's plan is still computed: the map and the plan of *Settings* show it marked **read-only**, so you can read what would happen before allowing it. A router of `ROUTERS_FILE` accepts `write_enabled` too (default `true`); the interface can make it read-only, and allow it again as long as the file does.

Writing needs the `write` and `api` policies. freeQoS reads the **real rights** of the account on the router (`/user` and `/user/group`) and gives one of three verdicts: can write, cannot (with the missing policy, or *Writing is disabled for this router in freeQoS (Devices)* when the per-router switch is off — the account's own rights are then still reported, in `account_can_write`), or undetermined. When `/user` cannot be read (an account authenticated by RADIUS, for example), freeQoS **does not block**: it tries the command and reports what RouterOS answers, translating `not enough permissions` into the fix to make. A separate write account can be declared per router (`rw_username`); `REQUIRE_SEPARATE_WRITE_ACCOUNT=true` makes it mandatory.

Rolling out router by router

The safe way to put freeQoS next to an existing shaper (Preseem, scripts, RADIUS rate limits) is, entirely from the interface:

1. Declare the core and every NAS in *Devices*, leaving *Allow freeQoS to write on this router* unticked. Leave the global switch off.
2. Observe for as long as needed: tree, subscribers, latency, traffic, and in *Settings › Shaping and writing to the routers* the plan each router would receive, marked read-only.
3. Allow writing on **one** pilot NAS (its switch in *Devices*: the plan is shown, a reason is asked), then turn the global switch on. Only that NAS receives commands.
4. Then NAS by NAS. Turning a router's switch off puts it back to read-only at once; the global switch stops everything.

Closed QoE loop

Every 5 minutes, freeQoS looks at the QoE of each sector over the last 15 minutes.

- **Tighten:** if at least 2 subscribers of the sector fall below 55, the link's queue is tightened by 10%. The queue then forms again in CAKE rather than in the radio.
- **Floor:** it never goes below 50% of the computed rate.
- **Relax:** one step is given back after 3 healthy cycles in a row. Tighten fast, relax slowly, to avoid oscillation.
- **What never moves:** a subscriber's purchased plan. Only the sector's shared envelope changes.
- **What does not count:** a subscriber that has never answered the ping.

A single degraded subscriber points at its last mile (CPE, home Wi-Fi), not at the sector. This is why at least two are needed.

On and off. The setting `qoe_loop_enabled` (*Settings › Write safeguards*, live, on by default) turns the loop on or off. Off, the loop reads no score and decides nothing, and every tightening it had applied is released: the planner ignores the tightenings at once, the next reconciliation brings the links back to their normal envelope, and the stored tightenings are reset (`last_action` = `release`) as soon as the setting changes, without waiting for the loop's next pass, so that turning the loop back on starts from scratch rather than re-applying old decisions (turning it back on resets them again, in case some were left over). Like every write, the loop is also subject to the global switch and to each router's switch: on a read-only router, the tightening is decided and the plan computed, nothing is sent.

8. Capacity and business insights

The measurements answer operating questions ("is it working?"). The same data, read over days, answers capacity and commercial ones: how much was sold behind each link, when links saturate, who is about to leave, who is ready for a bigger plan, and how many more subscribers each site can take.

Sold against real capacity

For each PoP, freeQoS divides the **sum of the plans sold** by the **measured capacity** of the site (`GET /api/v1/capacity`):

Oversubscription	Verdict
up to 5:1	comfortable
5:1 to 20:1	to watch
above 20:1	tight
nothing sold / capacity not measured	said as such, never shown as 0

A PoP at 12:1 is not broken: it becomes so the day its subscribers use it at the same time. That is why the **busy-hour peak** actually observed is shown next to the ratio.

When a link saturates

For each link, the **occupancy** is the busy-hour peak divided by its capacity: below 80% *free*, from 80% *loaded*, from 95% *saturated*. It can exceed 100%: the capacity of a port is its negotiated speed, that of a radio a measurement of the moment, and an occupancy above 100% means the capacity retained is underestimated — exactly what needs to be seen.

A link is flagged **to reinforce** when its **average** occupancy over the period is above 80%, with at least 10 samples. A peak at 100% proves nothing — it is what a well-sized link does on a match night; an average above 80% means the next growth will be paid in latency for everyone.

Saturation risks (*Executive*, `GET /api/v1/capacity/hotspots`) apply the gauges' thresholds — from 70% a link is to watch, from 90% it no longer holds one more peak — and split links into two sides: the **internet side** (the gateway's uplink and the PoP-to-core links, where a saturation hits every client) and the **PoP side** (links towards subscribers, VLANs and relays, where it only hits what hangs below). A radio link that only carries 70% of its nominal capacity is flagged as degraded.

Who is about to leave, who is ready for more

Insights (`GET /api/v1/insights/subscribers?days=7`) compares the chosen period (7, 14 or 30 days) with the one before it, so it needs **twice the period** of history; the page says how many days it has.

List	Rule
At risk of leaving	A poor experience (QoE score below 50); **or** usage collapsing (current average below 30% of the previous one, which was above 10 kbps); **or** a line that used to be used and has been silent for 3 days or more
Ready for a bigger plan	At 90% of its plan or more during at least 15% of the period, **with a good experience** (score 50 or more, or not measured)

A subscriber who saturates its plan **and** whose latency rises is not in the second list: that is a network problem first, and selling it a bigger plan would fix nothing. Subscribers spending more than 20% of their samples above 90% of their plan are also listed in the capacity view as living *at their ceiling*.

How many more subscribers a site can take

For each site and access point (`GET /api/v1/insights/capacity`), freeQoS takes the measured busy-hour peak, divides it by the number of subscribers to get what each one adds at peak, and divides the room left **under 80% of the capacity** by that contribution. This is the site's own measurement, not a theoretical oversubscription ratio. A site where 20% or more of the subscribers already have a poor experience has **no room**, whatever the apparent margin. Without a known capacity or a busy-hour measurement, the answer is "unknown", with the reason.

Volumes

The capacity view also ranks subscribers by **volume** over the period. The top of instantaneous rates names whoever is downloading right now; the volume over a week names whoever weighs on the network. They are almost never the same subscribers.

9. Interface

Errors say what happened and what to do: a title (*Not allowed with your account*, *Not possible right now*, *A router or service did not answer*, *freeQoS cannot be reached*…), the server's reason, and the next step. An error from a button shows in a corner without blocking the page; a page that cannot load says so in a banner and retries on its own.

Each page reads in sections, with a table of contents at the top and a tooltip on every figure that deserves an explanation. The two long configuration pages, **Settings** and **Devices**, show one section at a time: their table of contents works as tabs.

Page	Question it answers	Sections
Dashboard (first page, both roles)	Is everything all right today?	A banner: **All good**, or the problems that need you, most urgent first, each one a click away from the clients concerned; clients connected, traffic now, sites, modes and boosts; one card per site. Below, for the Administrator only: *Network detail* (throughput, traffic per router, top consumers, radio backhauls)
Rules (Administrator)	What may clients reach?	Rules in force, and the form to write one: block or cap a service (YouTube, Netflix…), a category or IP prefixes, for every client, chosen sites or chosen clients
Executive	Where is the risk?	Saturation risks, latency per subscriber, load and queues per node, health over time (QoE heatmap)
Traffic	Who consumes what, where to?	Consumers, services, countries, who talks to whom, destinations, IP lookup (restrictions moved to **Rules**)
Network tree	How is the network wired?	Discovered tree, editable with the mouse (move, re-parent, merge, hide)
Clients	Who are my clients, and how are they doing?	Every client under its **PoP**, with its connection (PPPoE, static IP, or **VLAN 2060**: a VLAN is how a client is connected, never a site of its own); its mode, boost and usage; *+ Add a client* (Operator) declares PPPoE, static-IP and VLAN clients. The map button opens the **Map**.
Plans (from *Settings › Plans & map*)	Which rate for whom?	Default plan, each subscriber's plan and its source, packages pushed by the API
Insights	What to do commercially?	Subscribers at risk of leaving, ready for a bigger plan, sites with room to grow
Devices	Which equipment?	Router health, adding a router, Ubiquiti antennas, radio health, inventory
API	How to integrate?	API keys, links to the API guide and to this documentation
Settings	How does the tool behave?	Accounts, settings, writing to the routers, caps held or not, command log

Read the Settings page before enabling writes

The *Shaping and writing to the routers* section shows, before enabling, what freeQoS would do on each router: queues created, modified, deleted, and links skipped with their reason. A router whose writing is disabled in *Devices* is listed too, marked **read-only**: its plan is what it would receive once allowed, and it is left out of the commands counted when the global switch is turned on.

Two banners can appear at the top of the interface:

- **Other instance detected:** another freeQoS installation creates queues on the same routers. Its objects are not touched; one of them must be stopped.
- **Frozen lines:** anti-oscillation has frozen one or more queues. The banner says which and why.

Offline, or not measured?

A client is **Offline** when its router answered but the client was not in its sessions (`/ppp/active`) for 90 seconds: the client really left (CPE off, line down, PPPoE session closed by the router or RADIUS). freeQoS never closes a PPPoE session itself: it only writes queues, firewall rules and the NetFlow export.

A client is **Not measured · router X not answering freeQoS** when freeQoS could not read the sessions of its router. The client may be connected; it is freeQoS that no longer reaches the router. Hover the text for the reason:

Reason shown	What to check on the router or the network
connection refused: the API service is off or restricted to another address	`/ip service print` : `api` (8728) or `api-ssl` (8729) enabled, and its `address=` contains the address freeQoS comes from **as the router sees it** (after NAT or through a tunnel it may not be the server's public IP)
no answer in time	Firewall `input` chain (TCP 8728/8729 from freeQoS), the route back to freeQoS, CPU of the router (*Devices › Health*)
login refused	The API account and its password (`/user print` , `/log print where topics~"account"`)
no route to the router	Routing between the freeQoS server and the router

`GET /api/v1/pops/read-status` gives, for each router, the time of the last successful read, since when it fails and why. The container log says the same: `sudo docker compose logs app | grep "Collecte impossible"` .

RouterOS calls are blocking and run in threads: `IO_THREADS` (default 64) sets how many. A small pool (the Python default is the number of CPUs + 4) let latency probes and configuration exports hold every thread, and session reads ran out of time on routers that were fine.

Empty latency

When a subscriber has no latency, the cell does not stay silent: it says whether the subscriber has never answered, whether the measurement is too old, or whether the probe has not run yet. *Find the cause* runs the diagnostic described in section 5.

Search, units and empty values

- **Search** (top of every page, `GET /api/v1/search`): one field finds a subscriber, an IP, a MAC, a device or a site — type what you have in front of you.
- **Units**: rate fields accept kbps, Mbps or Gbps, and values are displayed in the most readable unit.
- **Empty is not zero**: a value that was not measured shows "-" or a reason, never 0. A subscriber that is declared but never measured is listed with gaps, not zeros.
- **Errors are shown**: a tab that cannot load says which error, rather than staying blank.
- **No external dependency**: no framework, CDN or downloaded map tile; the interface works on a management server cut off from the internet. After an update, the browser cannot serve the old scripts: their address carries a fingerprint of their content.

Every section and every figure of the interface carries an (i) with its explanation. Appendix A reproduces all of them, page by page.

10. API

Everything the interface does goes through the API. The full guide, with `curl` examples and the reference generated by the server, is at `<your-freeqos-url>/api-guide`. This section gives its structure and rules.

Three APIs, one server, the same keys

API	Path	Use	Units
Operating	`/api/v1`	Everything the interface does: routers, plans, limits, measurements, traffic, diagnostics	Mbit/s
Model	`/model/v1`	Commercial inventory: `accounts`, `packages`, `sites`, `access_points`, `services`. Same contract as the Preseem model API: an existing integration works by changing the URL and the key.	kbit/s
Usage	`/usage/v1`	Bytes consumed per sold line, per hour, day or month	bytes

How the commercial model reaches the network

A billing system (Splynx, UISP CRM, Powercode…) knows its customers and what it sold them, but not the routers. The `/model/v1` API makes the link. It takes the exact shape of the Preseem model API: an existing integration only has to change URL and key.

Collection	What it describes	Fields that matter
`accounts`	A customer (person or company)	`id`, `name`
`packages`	A plan from the catalogue	`down_speed`, `up_speed` in **kbit/s**
`sites`	A tower, a location	`name`: it becomes the **PoP name** in freeQoS
`access_points`	A radio sector of a site	`site`, `ip_address`
`services`	A sold line: a customer, a plan, an address	`account`, `package`, `attachments` (networks and CPE MAC), `parent_device_id` (the sector)

What happens when a service is pushed (`PUT /model/v1/services/{id}`):

1. **The rate** is the service's own if it has one, otherwise its package's. A commercial exception on one line does not require creating a package for it.
2. **The PoP** is found by walking service → access point → site → site name. freeQoS finds the router to configure through that name.
3. **With an address** (`network_prefixes`), the service becomes a static-IP subscriber: it is placed on its router, measured and limited.
4. **With only a MAC**, it is set aside. As soon as that MAC appears in a router's ARP or DHCP table, freeQoS deduces the address and places the service.

5. **An identifier already used by a record entered by hand** is refused (`409`): the API never takes over a human action.

`PUT` creates or replaces: billing can replay its whole inventory every night without risk. The identifier in the URL is authoritative; a body carrying another one is refused (`400`).

Model API detail	Behaviour
Lists	`GET /model/v1/<collection>?page=1&limit=500` returns `{"data": [...], "paginator": {...}}`; without `limit`, everything is returned
Codes	`200` for every success (including `DELETE`), `400` for malformed JSON or a contradictory identifier, `401` for a missing or refused key **and for a key lacking the scope** (as Preseem does), `404` for an absent object, `409` for an identifier owned by a hand-entered record
Responses	The record as Preseem returns it: an unset field (a rate, for example) is **omitted**, never `null`; the CPE MAC is lower case
Enforcement report	For a service, what was written on the router is in the `X-FreeQoS-Enforcement` response header, outside the body
Several prefixes	All of them share **one** queue, whose target lists them (`target=10.0.0.0/29,10.0.1.0/28`): one service, one subscribed rate. All of them count in the traffic measurement

Usage API parameters (`GET /usage/v1/services` and `/usage/v1/services/{id}`): `start` and `end` (ISO 8601, UTC), or `days` (default 30, up to 366) when `start` is absent; `bucket` = `total`, `hour`, `day` or `month`; `vantage` = `auto`, `edge` or `pop` to override the accounting vantage point.

For a PPPoE subscriber, the simplest is still `PUT /api/v1/plans/{login}`: the login is enough to find it on its router.

A plan can come before the subscriber. A management interface that feeds the RADIUS creates the subscriber and its plan before its first PPPoE session, and an initial sync pushes offline subscribers too. `PUT /api/v1/plans/{login}` therefore accepts a login never seen connected: the plan is kept (`"pending": true`, state `en-attente-de-connexion`), listed under *Plans › Waiting for first connection* and in the `pending` field of `GET /api/v1/plans`, and applied by itself at the first session — the plan source reads the same table for every new login. `DELETE` forgets a pending plan (a misspelt login would otherwise wait forever). Use the PPPoE login, which is the RADIUS username: unlike an address, it never changes. With `PLAN_PROVIDER=freeradius_sql`, a new subscriber's plan comes from RADIUS at its first connection: a pending plan is then refused (`409`) rather than silently ignored.

Consumption (`/usage/v1`) comes from NetFlow, not from queue counters: those restart from zero at each reconnection and do not exist for a subscriber without a queue. With no router exporting, this API returns zeros and says so in its `source` field.

Authentication

A key is created in the *API* tab. The secret is shown only once. Two scopes:

- **read**: every `GET`;
- **read + write**: also `POST`, `PUT`, `PATCH`, `DELETE`.

A key can never manage keys, accounts or passwords: that stays behind a person's login.

Three equivalent forms are accepted everywhere:

```
Authorization: Bearer <key>
X-API-Key: <key>
Authorization: Basic base64(<key>:)      # Preseem form
```

`GET /model/v1` checks a key in one call: it returns the collections, the key prefix and its scopes.

The secret is drawn once, shown once, and the database only keeps its SHA-256 fingerprint. A lost key is revoked and replaced. A refusal never says *why* (unknown, disabled, expired): distinguishing the cases would give an oracle to whoever tries keys at random; the server log does say it.

Conventions

- JSON in and out. *Down* = towards the subscriber, *up* = from the subscriber.
- ISO 8601 timestamps in UTC.
- A PPPoE subscriber is identified by its login, a static-IP subscriber by its reference, a model object by the `id` chosen in the URL.
- `PUT` creates or replaces: replaying the whole inventory is harmless.
- A plan or limit change is written to the router **in the same call**.

The enforcement report

Every response that touches a queue contains an `enforcement` object telling what actually happened:

`state`	Meaning
`file-posee`	Queue written (or already correct)
`file-retiree`	Queue removed (plan deleted)
`file-a-poser`	Would be written: simulation
`ecarte`	Not written; `reason` says why (subscriber offline, shared address, infrastructure address…)
`conflit`	Another queue already targets this subscriber; to be cleaned by hand
`sans-routeur`	No router carries this subscriber yet
`erreur`	The router refused or is unreachable; `reason` gives the RouterOS message

The calls that matter

Need	Call
Add a router	`POST /api/v1/pops/routers` then `GET /api/v1/pops/provisioning/{name}`
List subscribers and their plan	`GET /api/v1/plans`
Give a plan (even before the first connection)	`PUT /api/v1/plans/{login}` with `down_mbps`, `up_mbps` or `package_id`
Remove a plan	`DELETE /api/v1/plans/{login}`
Force a limit (unpaid bill, fair use)	`PUT /api/v1/shaping/policies`
Temporary boost	`POST /api/v1/shaping/boosts`
Declare a static-IP subscriber	`POST /api/v1/static-clients`
Push the commercial inventory	`PUT /model/v1/{collection}/{id}`
Read consumption	`GET /usage/v1/services`
A subscriber's latency, diagnostic	`GET /api/v1/rtt/diagnose`
Overall state	`GET /api/v1/status`

A few usage rules:

- A plan for an unknown login returns `404`: the subscriber must first be seen on a router or declared.
- Giving a plan lifts a forced limit on that subscriber: the plan becomes the rule again.
- The report of an API call is authoritative. In simulation, automatic writing is blocked even for a `write` key: `state` is then `file-a-poser`.

Reference

`/openapi.json` describes every route, generated by the installed server: it cannot drift from the code. The `/api-guide` page makes it readable, with search.

11. Security

freeQoS holds router credentials that can write across the whole network. Security follows a simple principle: **what writes is rare, explicit and logged**.

Interface accounts

- **First account:** created by the application on its first start, as an Administrator (`edit`), with a random password kept in `data/first-admin.txt` (mode 0600) until it is changed. The first-visit screen only appears if this is turned off (`FIRST_ADMIN_AUTO=false`).
- **Three access levels**, each including the rights of the previous one, enforced by the **server** (a refused write is a `403` naming the level it needs, whatever the interface shows):

Level (`role`)	Sees	Can do	Interface
Viewer (`read`)	Everything	Nothing	Every page, read-only banner
Operator (`operator`)	Clients	**Manages the clients**: adds them (PPPoE login declared ahead, static IP, VLAN), edits their details and connection, changes plan, mode and boost on one or many clients, defines bandwidth modes and schedules; sees at a glance whether there is a problem	**Dashboard** (today's status), **Clients**, **Modes**, **Schedules**, **Alerts**, **Activity**; no network pages, no Settings (password and log out from the account card)
Administrator (`edit`)	Everything	Everything the Operator does, plus **the network**: routers and PoPs (Devices), antennas and their API (Ubiquiti), traffic **Rules**, the network tree, detailed monitoring (Traffic, Insights, Executive, network detail of the Dashboard), API keys, Settings, accounts	Every page; the network pages are grouped under *Network* in the sidebar

The writes open to the Operator form a **closed list** (`POST/DELETE /api/v1/shaping/boosts` , `PUT /api/v1/clients/{login}/mode` , `PUT /api/v1/clients/{login}/profile` , `POST /api/v1/clients` , `POST/PATCH/DELETE /api/v1/static-clients` , `POST /api/v1/clients/actions` , `PUT/DELETE /api/v1/plans/{login}` , `POST/PATCH/DELETE /api/v1/bandwidth-modes` and `/api/v1/bandwidth-schedules`): a route added later is reserved to the Administrator until it is opened on purpose. Every change is recorded with the **account** that made it, in the **activity log** below and in the enforcement audit. - **Activity log:** every change made on a client (bandwidth mode, boost, boost stopped, plan, plan removed) is recorded in `client_activity` : who, when, which client, the detail, and the state of the queue afterwards (written, pending on the router because writing is off, no router, refused). It is read on the **Activity** page (search, CSV export), in each client's sheet, and through `GET /api/v1/activity?login=&q=&limit=` . Kept 400 days. An API key keeps its two scopes: `read` , or `write` (Administrator rights except accounts, which no key can manage). The stored values `read` and `edit` are those of the former two roles: existing accounts keep their level without migration. - **Passwords:** 12 characters minimum, too-common words are refused. They are hashed with scrypt (N=2^14, r=8, p=1, salted, compared in constant time). - **Anti-guessing:** 5 failures in 5 minutes lock the email **and** the client address for 5 minutes. Each repeat doubles the wait, up to 1 h. - **Session:** an `HttpOnly` , `SameSite=Strict` cookie, valid 24 h, 30 days at most (`SESSION_TTL_HOURS` , `SESSION_MAX_HOURS`). Writes also check `Origin` and `Sec-Fetch-Site` . - **Log:** every login (successful, refused, locked), logout and account change is recorded with the address and browser. At login, everyone sees the date and origin

of their previous login. - **Last Administrator:** at least one active Administrator account always remains; the last one can be neither deleted, downgraded nor disabled. - **Sessions:** at most 10 open sessions per account; everyone sees their own sessions (browser, address, last activity) and can close them remotely. Changing a password or a role closes the other sessions. - **Journal retention:** login events are kept 180 days (*Settings › Accounts › Login journal*) and are also written to the container log (`auth login_failed …`), usable by fail2ban.

Behind an HTTPS reverse proxy (the included one is already set up), set `FORWARDED_ALLOW_IPS=<proxy address>` so that the application trusts the proxy's `X-Forwarded-Proto` : the session cookie is then sent `Secure` , HSTS is set, and the address recorded for each login is the client's, not the proxy's.

Locked out? Give the account a new password from the server; measurements and inventory are untouched:

```
sudo docker compose exec app python -m app.cli reset-password --email noc@example.com
```

HTTP headers

Header	Effect
`Content-Security-Policy`	Only scripts served by freeQoS run
`X-Frame-Options: DENY` , `frame-ancestors 'none'`	The page cannot be embedded elsewhere
`Referrer-Policy: same-origin`	Internal addresses do not leak to external links
`Strict-Transport-Security`	HTTPS only: no going back to plain HTTP

API keys

`read` or `read + write` scope. A key never manages keys, accounts or passwords. It can be disabled and revoked. Every write made with a key is logged as `api:<name> (<prefix>)` .

Router credentials

- Passwords are encrypted in the database with `data/secret.key` and never come out of any response, not even encrypted.
- **Least privilege:** a dedicated RouterOS account (`read,write,api,test`), without `ssh` , `ftp` , `winbox` or `policy` .
- **Separate accounts:** `REQUIRE_SEPARATE_WRITE_ACCOUNT=true` requires a distinct account for writing.
- **Encryption:** `api-ssl` (8729) with `tls_verify=strict` or a fingerprint, rather than the clear API, when freeQoS and the routers do not share an isolated management network.

Writing to the network

- Simulation by default; explicit enabling, after reading the plan.
- `ENFORCEMENT_ENABLED=false` in the environment locks writing: the interface cannot turn it back on.

- Writing is allowed **router by router** in *Devices*; a router added from the interface starts read-only, and a read-only router receives nothing at all.
- Only objects of **this** instance are touched; the others are read, never modified.
- The circuit breaker refuses an abnormally large plan.
- The command log (`/api/v1/shaping/audit`) gives the author, target and result of each write.

Network exposure

Port	Use	Who needs access
8000/TCP (or 443 behind a TLS proxy)	Interface and API	Operators, billing system
2055/UDP	NetFlow	The routers only
8728 or 8729/TCP **outgoing**	RouterOS API	freeQoS towards the routers

It is recommended to put the interface behind HTTPS and to restrict 2055/UDP to the routers' addresses.

12. Operations

Monitoring freeQoS

Call	Answers	Use
`GET /health`	Does the process answer? Touches neither the database nor the routers.	Liveness (container restart)
`GET /health/ready`	Does the database answer, is data arriving? `503` if a cycle keeps failing.	Readiness, monitoring
`GET /api/v1/status`	State of each cycle, instance identifier, other instances seen, frozen lines	Diagnostics
`GET /api/v1/status/runs`	History of cycle runs	Diagnostics

Monitor `/health/ready`, not `/health`: a collector can fail for an hour while the process still answers.

The cycles

Each cycle can be run immediately with `POST /api/v1/jobs/{name}/run`.

Cycle	Period	Role
`collect_subscribers`	10 s	Subscriber sessions and rates
`collect_links`	10 s	Port rates
`collect_backhauls`	30 s	Radio capacity (UISP, airOS)
`probe_rtt`	30 s	Latency probe
`expire_boosts`	30 s	End of boosts
`ip_intel`	30 s	Naming addresses
`netflow_flush`	60 s	Writing aggregated flows
`reload_inventory`	60 s	Re-reading the router list
`reconcile_shaping`	2 min	Queues: desired versus actual state
`refresh_plans`	5 min	Re-reading plans
`qoe_closed_loop`	5 min	QoE loop per sector
`traffic_restrictions`	5 min	Address lists of restrictions
`netflow_export`	10 min	NetFlow export on the routers

Cycle	Period	Role
`discover_topology`	15 min	Network tree
`verify_caps`	3 min	Are the caps actually held (section 7)
`dns_names`	2 min	Reading the routers' DNS cache
`place_unplaced_services`	2 min	Placing services pushed without an address
`purge_empty_pops`	10 min	Removing empty sites

The scheduler runs each cycle once immediately at startup, then at its period. A cycle never overlaps itself: it runs, then sleeps for the rest of its period; an overrun is counted, not stacked.

Live settings

Most operating settings can be changed in *Settings* without a restart (`PUT /api/v1/settings/{name}`): shaping and CAKE options apply at the next plan, cadences at the next turn of the scheduler. They are stored in the database with their history (`GET /api/v1/settings/history` : who, what, when, why). **The database wins**: the environment only seeds a value on the very first start, so once a setting has been changed in the interface, editing `.env` no longer affects it. `GET /api/v1/settings` shows each setting's value, default and origin; `DELETE /api/v1/settings/{name}` brings it back to its default.

Deliberately out of the interface's reach: database access, secrets, and `ENFORCEMENT_ENABLED=false` when it is set in the environment.

Data and retention

Measurements live in TimescaleDB hypertables:

- split by day (`CHUNK_INTERVAL_HOURS=24`);
- compressed after 7 days (`COMPRESSION_AFTER_DAYS`);
- deleted after 90 days (`RETENTION_DAYS`).

Without TimescaleDB, freeQoS runs on plain PostgreSQL, without compression or automatic retention.

Backup and restore

```
make backup                                  # backups/YYYYMMDD-HHMMSS/: base.dump + secret.key
make restore DIR=backups/20261008-0300       # replaces the database, asks for confirmation
```

`make restore` stops the application, wraps `pg_restore` in `timescaledb_pre_restore()` / `post_restore()` , puts `secret.key` back, then restarts. Copy backups off the server.

`make backup` does not copy `data/instance.id` . On a new server, copy it back or set `FREEQOS_INSTANCE_ID` ; otherwise queues already in place will be seen as another instance's and left alone.

Common actions

Command	Effect
`make update`	Fetches the code and restarts the application; no data lost
`make backup` / `make restore DIR=…`	Backup and restore (see above)
`make logs`	Follows the application log
`make reset-db`	**Erases** the database (measurements, routers, antennas, topology, settings, static clients) after confirmation, and regenerates the encryption key: routers must be declared again

Situation	Action
Start again from scratch on a router (inconsistent queues, old version)	*Reset queues* in *Settings*: removes all queues of **this** instance, unfreezes lines, then reconciliation reinstalls the desired state
Wrong latencies after a change of method	`POST /api/v1/rtt/reset-history` (clears 48 h of measurements and the scores derived from them)
A router's password changed	`PATCH /api/v1/pops/routers/{id}` then `POST /api/v1/pops/provisioning/{name}`
Vanished boxes in the tree	`POST /api/v1/topology/forget-stale`
Stop writing urgently	Switch in *Settings*, or `ENFORCEMENT_ENABLED=false` then restart
Stop writing on one router only	Its *writing allowed* switch in *Devices* (immediate)
Stop the closed QoE loop	`qoe_loop_enabled` = No in *Settings* (tightenings released at the next reconciliation)
Stop the latency probe	`PUT /api/v1/rtt`

Several instances

Two installations must not drive the same routers. If it happens, each only touches its own objects (`instance=<id>` comment), but they fight over subscribers. The *Other instance* banner reports it. For an update or a migration, stop the old one **before** starting the new one.

13. Troubleshooting

All these cases come from the lab or the first deployment. For each: the symptom, the check that settles it, then the action.

700 to 900 ms latency on nearby subscribers

- **Cause:** the ping leaves through the `main` table while subscriber routes are in a VRF. It goes up to the core and back.
- **Check:** *Find the cause* → `routing_table` must name the VRF (`CUST-INET`), not `main`. On the router, `/ping <subscriber> vrf=CUST-INET count=5` must give a few ms.
- **Action:** up to date, freeQoS picks the VRF by itself. If `routing_table` stays `main`, look at `route_candidates`: the route to the subscriber may be a default route towards upstream. After fixing, `POST /api/v1/rtt/reset-history` clears the wrong measurements.

Empty latency ("no reply", "probe silent")

Run *Find the cause* and read the `code` (table in section 5). The two most frequent cases:

- `client_blocks_icmp`: the subscriber's CPE filters ping. This is not a degradation; the subscriber is excluded from QoE.
- `no_test_policy`: the RouterOS account's group lacks the `test` policy.

High latency on a saturated subscriber

- **Symptom:** a single subscriber at 300 or 500 ms, filling its plan.
- **Cause:** real bufferbloat. Without a CAKE queue, the queue forms in a buffer along the path (CPE, radio, test link).
- **Action:** give it a plan slightly **below** the real capacity of its line, so the queue forms in CAKE. Example: 800 kbit/s on a line that holds 1 Mbit/s. If latency stays high, the real capacity is lower than expected: lower the plan further.

A queue is rewritten every 2 minutes

Two writers are fighting over the queue: two freeQoS instances, a script, or another queue targeting the same address.

- **Check:** `GET /api/v1/status`. Read `other_instances` (another installation is writing) and `frozen_lines` (anti-oscillation froze the line).
- **Action:** stop the extra instance, then *Reset queues*. If a queue created by hand targets the same subscriber, remove it by hand: freeQoS will not touch it.

A parent queue appears on a transit link

Recent versions skip routing links, the uplink, the gateway and links towards other managed routers.

- **Check:** the *Settings* page lists skipped links with their `skip_reason`.

- **Action:** a queue created by an old version stays until *Reset queues*. If the link is not recognised as transit, correct its role in the tree (`PATCH /api/v1/topology/nodes/{key}`).

Nothing is written to the routers

In order:

1. Is writing enabled? `GET /api/v1/shaping/enforcement` .
2. Is it locked by the environment? `ENFORCEMENT_ENABLED=false` in `.env` .
3. Is writing allowed on **this** router? Its switch in *Devices* (`write_enabled` in `GET /api/v1/pops/routers`); a router added from the interface starts read-only.
4. Does the account have `write` ? `GET /api/v1/shaping/capability` .
5. Did the plan trip the circuit breaker? The log says so.
6. Does the subscriber have a plan? A subscriber that is only detected is observed, not limited.

No NetFlow traffic

1. `GET /api/v1/netflow/status` : are datagrams arriving?
2. `GET /api/v1/netflow/export` : is the target set, and to which address?
3. In a container, is `NETFLOW_COLLECTOR_ADDRESS` set?
4. Is UDP port 2055 open between the routers and freeQoS?

Scores stay old

Scores are computed over a sliding 15-minute window. After a measurement fix, old samples still weigh in. `POST /api/v1/rtt/reset-history` starts again from clean measurements.

The router refuses the connection

`POST /api/v1/pops/routers/test` gives a hint:

Hint	Check
Port closed	`api` or `api-ssl` service enabled, and allowed from the freeQoS address
Credentials refused	Account name and password
`api` policy missing	Group of the RouterOS account
Certificate refused	`api-ssl` certificate, or `tls_verify=fingerprint`

14. Settings reference

All settings are read from the environment or from `.env`. The variable name is the one in the table, in capitals. Those marked ✱ can also be changed live in *Settings*.

Application and database

Variable	Default	Role
`APP_ENV`	`lab`	Environment label
`LOG_LEVEL`	`INFO`	Log verbosity
`AUTH_ENABLED`	`true`	Login required; only disable on an isolated lab
`SESSION_TTL_HOURS` / `SESSION_MAX_HOURS`	24 / 720	Idle session / maximum duration
`DATABASE_URL`	`postgresql://qos:…@localhost:5432/qos`	PostgreSQL / TimescaleDB database
`DB_POOL_MIN` / `DB_POOL_MAX`	1 / 8	Connections
`DB_AUTO_MIGRATE`	`true`	Migrations at startup
`CHUNK_INTERVAL_HOURS`	24	Hypertable chunking
`COMPRESSION_AFTER_DAYS`	7	Compression
`RETENTION_DAYS`	90	Purge
`APP_SECRET_KEY_FILE`	`data/secret.key`	Encryption key (generated if missing)
`FREEQOS_INSTANCE_ID`	`data/instance.id`	Instance identifier in comments

Inventory and sources

Variable	Default	Role
`ROUTERS_FILE` / `ROUTERS`	empty	Routers declared by file or as JSON
`BACKHAUL_PROVIDER`	`mock`	`uisp`, `airos` or `mock` (lab simulator). In production, choose `uisp` or `airos`.
`UISP_BASE_URL`, `UISP_TOKEN`	—	UISP controller (read-only)
`AIROS_USERNAME`, `AIROS_PASSWORD`	—	Direct access to the antennas
`PLAN_PROVIDER`	`clients`	`clients` (API and *Plans* page) or `freeradius_sql`

Variable	Default	Role
`RADIUS_DSN`, `RADIUS_RATE_ATTRIBUTE`	—, `Mikrotik-Rate-Limit`	Plans read from FreeRADIUS

Cycles

Variable	Default
`SUBSCRIBER_INTERVAL_S` / `LINK_INTERVAL_S`	10
`BACKHAUL_INTERVAL_S`	30
`INVENTORY_REFRESH_INTERVAL_S`	60
`SHAPING_RECONCILE_INTERVAL_S`	120
`PLAN_REFRESH_INTERVAL_S`	300
`TOPOLOGY_REFRESH_INTERVAL_S`	900
`SCHEDULER_ENABLED`	`true`

Latency

Variable	Default	Role
`RTT_ENABLED`	`true`	Probe on (switchable live with `PUT /api/v1/rtt`)
`RTT_INTERVAL_S`	30	Cycle
`RTT_BATCH_SIZE`	20	Subscribers per cycle and per PoP
`RTT_COUNT` / `RTT_PING_INTERVAL_MS`	5 / 200	Pings per subscriber
`RTT_PROBE_DSCP`	46	Probe priority (0 = none)
`RTT_MAX_AGE_S`	300	Maximum age of a displayed measurement
`LATENCY_INTERNET_TARGETS`	`1.1.1.1`, `8.8.8.8`	Targets of the internet segment

Plans and shaping

Variable	Default	Role
`ENFORCEMENT_ENABLED`	`false`	Writing to the routers; an explicit `false` = lock
`ENFORCEMENT_MAX_ACTIONS` ✱	500	Circuit breaker per plan
`REQUIRE_SEPARATE_WRITE_ACCOUNT` ✱	`false`	Distinct write account required

Variable	Default	Role
`DEFAULT_PLAN_DOWN_MBPS` / `UP` ✱	100 / 20	Default plan
`DEFAULT_PLAN_FOR_DETECTED_CLIENTS` ✱	`false`	Also limit subscribers that are only detected
`SHAPING_SAFETY_FACTOR` ✱	0.90	Share of measured capacity given to the parent queue
`SHAPING_FLOOR_MBPS` ✱	5	Minimum rate of a parent queue
`SHAPING_PRUNE` ✱	`true`	Remove queues that became useless
`SHAPING_ADOPT_FOREIGN_QUEUES` ✱	`false`	Never take over other people's queues
`SHAPING_QUEUE_FOR_DETECTED_LINKS` ✱	`true`	Parent queues on discovered links
`CAKE_OVERHEAD` / `CAKE_RTT_MS` ✱	22 / 50	CAKE options
`CAKE_DIFFSERV`, `CAKE_FLOWMODE`, `CAKE_NAT`, `CAKE_ACK_FILTER`, `CAKE_WASH`, `CAKE_MPU` ✱	RouterOS	Advanced CAKE options
`BOOST_CHECK_INTERVAL_S`	30	Boost expiry

QoE loop

Variable	Default	Role
`QOE_LOOP_ENABLED` ✱	`true`	Closed QoE loop on or off; off releases its tightenings
`QOE_LOOP_INTERVAL_S`	300	Cycle
`QOE_WINDOW_MINUTES`	15	Analysis window
`QOE_SCORE_THRESHOLD`	55	Score below which a subscriber is degraded
`QOE_MIN_DEGRADED_SUBSCRIBERS`	2	Degraded subscribers needed to act on a sector
`QOE_TRIM_STEP` / `QOE_TRIM_FLOOR`	0.10 / 0.50	Tightening step / floor
`QOE_RECOVERY_CYCLES`	3	Healthy cycles before relaxing

NetFlow and traffic

Variable	Default	Role
`NETFLOW_ENABLED` / `NETFLOW_PORT`	`true` / 2055	Collector

Variable	Default	Role
`NETFLOW_FLUSH_INTERVAL_S`	60	Writing aggregates
`NETFLOW_ACCOUNTING_VANTAGE` ✱	`auto`	Where usage is read: `auto` (each direction where it is seen best), `edge` or `pop`
`NETFLOW_EXPORT_AUTO` ✱	`true`	Export set up by freeQoS
`NETFLOW_EXPORT_INTERVAL_S`	600	How often the export is checked
`NETFLOW_COLLECTOR_ADDRESS`	auto	Address announced to the routers (required in a container)
`NETFLOW_TRACK_DESTINATIONS` / `NETFLOW_DESTINATION_LIMIT` ✱	`true` / 2,000	Destinations kept (7 days)
`NETFLOW_TRACK_HOSTS` / `NETFLOW_HOST_LIMIT`	`false` / 500	Unknown hosts seen
`IPFINDER_ENABLED` ✱	`true`	Naming addresses (rDNS, RDAP, GeoIP, each can be disabled live)
`IPFINDER_INTERVAL_S` / `IPFINDER_BATCH_SIZE` ✱	30 / 40	Naming pace (only the batch size is live)
`RESTRICTIONS_INTERVAL_S`	300	Restriction updates
`RESTRICTION_ADDRESS_LIMIT` ✱	5,000	Maximum addresses per list

15. Data model

Reference data

Table	Content
`pops`	Sites: PoPs and VLAN sites, with the router serving each
`subscribers`	Every subscriber, PPPoE or static (`kind`), unique `login`, plan, PoP, last address, last seen
`static_clients`	Declared static-IP clients and services pushed through `/model/v1` (`source`); `extra_prefixes` holds their additional ranges
`client_plans`	The plan written for each client, and its source (`api`, `ui`)
`routers`, `hidden_file_routers`, `read_only_file_routers`	Routers added from the interface (encrypted password, unique loopback, writing allowed or not, last connection diagnostic); routers of the file hidden by hand, or made read-only from the interface
`backhauls`, `airos_antennas`	Radio links and the antennas polled for their capacity
`link_media`	Links declared wired (with a capacity) or radio (with their antenna)
`topology_nodes`, `topology_links`, `topology_aliases`, `topology_layout`	Discovered graph, manual merges, box positions and hand-made choices
`subscriber_attachments`	Subscriber → radio sector
`vlan_sightings`	Presence observed by the PoP census
`shaping_policies`, `qoe_link_states`	Rates set by hand and boosts; tightening decided by the QoE loop (kept apart so that one never overwrites the other)
`enforcement_audit`	Every command sent, with its author and result
`runtime_flags`, `runtime_settings`	Live switches (write, probe) and settings changed from the interface, with their history
`traffic_rules`	Restrictions as entered: a criterion, never a frozen address list
`bandwidth_schedules`	Time-of-day bandwidth schedules: time slots (validated, without overlap), scope and targets, time zone
`uisp_controllers`, `uisp_links`	The UISP controller connected from the interface (address, encrypted token, last reading) and the antennas attached to a site
`site_locations`, `client_locations`	Position of each site (by name) and client (by login) on the map: latitude, longitude, who placed it
`app_users`, `app_sessions`, `auth_events`, `api_keys`	Accounts, sessions, login journal, API keys (fingerprints only)

Table	Content
`model_accounts`, `model_packages`, `model_sites`, `model_access_points`, `model_services_unplaced`	The Preseem-compatible commercial model
`netflow_exporters`	Exporters and their vantage point and sampling
`collector_runs`	History of cycle runs

Time series (hypertables)

Table	Content
`subscriber_metrics`	Per subscriber, every 10 s: rates, byte counters, RTT, session uptime
`interface_metrics`	Per router port: rates, counters, state, negotiated speed — the source of link throughput
`backhaul_metrics`	Per radio: capacity (total, down, up), signal, airtime, MCS, online
`qoe_scores`	Reserved for stored scores (scores are currently computed when read)
`flow_metrics`, `flow_app_metrics`	NetFlow volume per subscriber and per usage family, **with the vantage point in the key**

What subscribers reach

Table	Content	Lifetime
`flow_destinations`, `flow_destination_buckets`	The pair (subscriber address, remote address): volumes, last port and protocol, first and last seen. The subscriber may be unknown	Measurement: purged after 7 days without traffic
`flow_hosts`	Addresses seen in flows and attached to no record (only if `NETFLOW_TRACK_HOSTS=true`)	24 h
`ip_intel`	What is known about an address: reverse name, service, family, organisation, AS, country, and on what grounds	Knowledge: kept

A row is created in `ip_intel` with no resolution date **as soon as a subscriber reaches the address**: that is the naming queue, and what makes discovery dynamic. Three views give the last point of each series: `subscriber_latest`, `interface_latest`, `backhaul_latest`. Aggregations use `date_bin` rather than `time_bucket`: the same queries work on plain PostgreSQL.

16. Known limitations and FAQ

Limitations, by design

- **No code in the packet path.** Consequence: no TCP retransmissions, and latency comes from an active probe that requires the subscriber's box to answer ping.
- **Radios are never driven.** Their capacity is read; writing only concerns RouterOS queues, address lists, firewall restrictions and the NetFlow export.
- **No RADIUS writes** (no CoA): plans can be read from FreeRADIUS, never written back.
- **No fast local loop** on the PoP: freeQoS sets baselines, it does not react below the second.
- **Restrictions are IPv4 only**: IPv6 prefixes are set aside and the plan says so.
- **A static-IP client is only measured once a queue targets it.**
- **The NetFlow collector uses a single core**: beyond about 5,000 flows per second, datagrams are lost (section 3).
- **The two NetFlow tables are not compressed**: they are the largest disk item (section 3).
- **RDAP and web geolocation are on by default** and send the addresses your subscribers visit to third parties (section 6).

FAQ

Does freeQoS cut subscribers if it stops? No. The queues stay on the routers and traffic flows; only measurement, timed boosts and adjustments pause. A boost that should have expired keeps running until freeQoS comes back.

Can I try it without touching my routers? Yes: it starts in simulation, and a router added from the interface starts read-only. Keep its *writing allowed* switch off in *Devices* (or the global switch off) and read the plan; giving the account only `read,api,test` also makes writing impossible, from the router side.

Will it touch the queues I created myself? Never. Only objects carrying `freeqos:managed instance=<this instance>` are modified or deleted; your queues are shown on the map as manual queues.

Why does a subscriber show no latency? Most often its CPE does not answer ping (`client_blocks_icmp`). *Find the cause* says which of the eight causes applies (section 5).

Why is a subscriber not capped? Is writing on, globally and on its router (*Devices*)? Does it have a plan? Is it online? Then look at *Are the caps actually held?*: FastTrack is the first cause of a cap that does not cap (section 7).

Why is the Traffic page empty? The page names the cause: no router, export not set up yet, writing off, or a network path problem — most often UDP 2055 not published or filtered (section 6).

Can two freeQoS run on the same routers? They will not overwrite each other's objects, but they will fight over subscribers. Run one per network; stop the old one before starting a new one.

How do I move freeQoS to another server? `make backup`, copy the backup **and the instance identifier** to the new server, `make restore`, set the same `FREEQOS_INSTANCE_ID` or copy `data/instance.id` into the volume, then update the routers' NetFlow target if the address changes (freeQoS does it by itself at the next export pass).

Where are the logs? `make logs` (or `sudo docker compose logs -f app`). Every command written to a router is also in *Settings › Log of commands sent*.

Appendix A. Interface reference

The explanations below are exactly those the interface shows behind each (i), extracted from the application at build time: they cannot differ from what operators read on screen.

Clients

The daily page of Operators (who also have Modes, Alerts, Activity, Schedules and Map, and no Settings): every client, its connection, its usage against its limit, its bandwidth mode, its boost and its plan, for one client or for a selection. The four cards on top filter the list; search, filters, sort and pages handle hundreds of clients; the list exports to CSV; Traffic charts the selected clients. An Administrator adds a client (static IP, VLAN, ranges) with the + button. Clicking a client opens its page (Overview: status, usage now against its limit, limit, latency, usage over 1 hour, 24 hours or 7 days, connection; Plan & mode: change its mode, plan and boost; Activity: its history).

Client operations

What it shows. The daily page: every client, its connection, its usage, its bandwidth mode and its boost.

How it is measured. Operators only see this page and their account. They act on one client or on a selection.

How to read it. Tick clients to change their mode or boost them together; a group change reports a result per client. The Plan button sets the rate the client subscribed to.

Bandwidth mode

What it shows. The named rate the client is in: Automatic, or a mode such as Backup or Night.

How it is measured. Automatic follows the plan and, when there is one, the bandwidth schedule: the rate changes with the hour on its own. A mode stays until the client is put back to Automatic.

How to read it. “Manual limit” is a rate set by hand by an Administrator, outside the modes.

Connection

Connected = a measurement less than 90 s old. Otherwise, when the client was last seen.

Current usage

Download (↓) and upload (↑) of the client right now, from the last measurement.

Not in automatic

Clients in a bandwidth mode or under a manual limit: their rate does not follow their plan and schedule.

Boosts active

Clients with a running boost. A boost ends on its own at its expiry.

Alerts

What needs someone, grouped by cause, each alert with a stable error code (ERR_POP_TIMEOUT, ERR_CLIENT_DISCONNECT…) and what to check. PDF report for every site or one site.

Connection health

What it shows. What needs someone, grouped: a site where several clients are down together is one site problem, not many client problems.

How it is measured. Also listed: clients offline on a site that is up, high latency (150 ms or more right now), clients in a mode that stops boosts, clients never measured.

How to read it. “View clients” opens the Clients page with these clients selected, ready for a mode or a boost.

Activity

The log of every change made on clients.

Team activity

What it shows. Every change made on a client: bandwidth mode, boost, plan. Who asked for it, when, and whether it reached the router.

How it is measured. “Applied”: the queue was written. “Pending on router”: writing is off or the router is unreachable, it is written later. “Refused”: nothing changed (for instance a boost in a mode that stops boosts).

How to read it. Kept 400 days. Export CSV gives the list shown.

Map

The sites and the clients on a map, coloured by connection. A site opens its clients and its PDF report; an Administrator places sites and clients by coordinates, GPS or a tap on the map.

Modes

Every bandwidth mode, the clients in each one, and creating or editing modes. Open to Operators.

Bandwidth modes

What it shows. Named rates operators give a client in one click: a fixed rate, or a share of the client's plan.

How it is measured. Defining modes needs the Administrator level; giving one to clients, the Operator level. Changing a mode's rate applies it at once to every client in it.

How to read it. A mode that “stops boosts” clears the running boost and refuses new ones: use it for a crisis or non-payment.

Schedules

The bandwidth schedules: rates that change with the time of day, on their own. Open to Operators.

Client management

What it shows. Bandwidth schedules: rates that change with the time of day, on their own (Schedules tab).

How it is measured. Open to Operators: these are commercial decisions about clients. Adding clients, sites, routers, antennas and APIs stays with the Administrator.

Bandwidth schedules

What it shows. Rates that change with the time of day (day, evening, night, weekend), on their own, for all clients, some sites, some plans or named clients.

How it is measured. Each time slot sets a share of the plan (200 % = twice the plan) or a fixed rate. A client in a bandwidth mode or with a boost is not affected until it is back in Automatic.

Dashboard

What is happening on the network right now.

Network throughput

What it shows. Total traffic of all your subscribers (PPPoE and static clients) over time. Download is drawn above the axis, upload below.

How it is measured. Read every 10 s from each subscriber’s queue on its router, then summed. Solid line = average of each time step; dotted line = the highest 10-second moment inside the step, so a short burst is not flattened away.

How to read it. Traffic that belongs to no subscriber — a bandwidth test between routers, device management, an undeclared client — is not in this curve: it shows in Routers — live traffic as “not from clients”. Filter by router or client with the menus.

What to do. A flat line at 0 while clients are online usually means the collection stopped: check the health dot at the top left and Devices › Polled routers.

Routers — live traffic

What it shows. Each router, live: what it exchanges with the internet on its uplink, what its own clients consume, and the gap between the two.

How it is measured. Uplink port counters and subscriber queues, read every 10 s. Router CPU and port load come from the same poll.

Scale.

- Green: under 70%: comfortable
- Amber: 70 to 90%: watch it — peaks start to queue

- Red: over 90%: saturated — latency rises for everyone behind it

How to read it. The coloured state line at the top says whether each collection (subscribers, ports, latency) is running. A large “not from clients” share is traffic of no known client.

What to do. Red collection state: open Devices › Polled routers and test the connection of that router.

Top consumers

What it shows. The subscribers using the most bandwidth right now.

How it is measured. Last 10-second measurement of each subscriber’s queue, sorted by download.

How to read it. A client at the top of this list is not a problem in itself — it is using what it pays for. It matters when its row is red (at its limit) and the node it belongs to is saturated too.

Radio backhauls

What it shows. The radio links declared in the inventory (UISP / airOS), with what they can carry now and how loaded they are.

How it is measured. Current capacity is read from the radio (it drops when the signal fades); nominal capacity is what the link is rated for. Load = traffic ÷ current capacity.

Scale.

- Green: under 70%: comfortable
- Amber: 70 to 90%: watch it — peaks start to queue
- Red: over 90%: saturated — latency rises for everyone behind it

What to do. Current capacity far below nominal: alignment, interference or rain fade — check signal and SNR on the antenna page.

Executive

Where the risk is: saturation, experience, load and queues per node, health over time. Clicking a node or a client opens the Selection panel.

Saturation risks

What it shows. Every link compared with what it can carry, worst first.

How it is measured. Traffic of the link ÷ its capacity (the lowest of: rate set by hand, measured radio capacity, port speed). Bar = now; marker = the peak of the selected period.

Scale.

- Green: under 70%: comfortable
- Amber: 70 to 90%: watch it — peaks start to queue
- Red: over 90%: saturated — latency rises for everyone behind it

How to read it. Internet side (gateway uplink, PoP-to-core): a saturation hits every client. PoP side (towards subscribers, VLANs, relays): only the clients behind that link. A link with “capacity unknown”

cannot be judged — set its rate in the Network tree.

What to do. Peak regularly over 90%: upgrade the link or lower the plans sold behind it. Only the peak is high: usually fine, links are meant to be full at the busiest minute.

Latency by client

What it shows. The experience each subscriber really gets, with a verdict (good / fair / poor) and the reason in plain words.

How it is measured. The PoP router pings each client 5 times every 30 s (200 ms apart, from its loopback). Median, p95, jitter and loss come from those pings; “under load” compares latency when the client’s line is busy with latency at rest.

Scale.

- Green: good: median under 30 ms, no loss, little bufferbloat
- Amber: fair: 30–100 ms, spikes, some loss, or +30 to +150 ms under load
- Red: poor: over 100 ms, 2%+ loss, +150 ms under load, or score under 50
- Grey: at plan limit: judged apart (see below)

How to read it. Samples taken while the client uses 85%+ of its plan are left out: a full line makes its own queue — that is not a network fault. “no reply” on every client of a router means the probe itself is failing, not the clients.

What to do. Many poor clients behind the same node: look at that node’s load and backhaul. A single poor client: its CPE, signal or home Wi-Fi.

Load by node

What it shows. Traffic of each node (site) compared with the sum of its clients’ limits.

How it is measured. Sum of the clients’ queues every 10 s, divided by the sum of their limits.

How to read it. This is a share of what was sold, not of the link: 100% means every client is at its cap at once, which practically never happens.

Queues by node

What it shows. Each node (PoP or router) with its clients: traffic now vs limit, limits applied, plans sold, worst latency and experience.

How it is measured. Queue counters every 10 s; latency from the probe every 30 s. A VLAN client is listed under the router that carries it.

How to read it. Click a row to open its detail below. The “Rate” field lets you force a client’s limit by hand — it then overrides its plan until you clear it.

Selection

Details of the node or client selected in the table above: link, plans, limits and the clients behind it.

Node

What it shows. Detail of the selected node: its link (capacity and load), the limits and plans of its clients, and how each client is doing.

How to read it. Oversubscription = plans sold ÷ link capacity. Some is normal (clients are not all active at once); far above 3× the link will be full at busy hours.

Client

What it shows. Detail of the selected client: its plan and where it comes from, the limit really applied, its usage and its latency at rest and under load.

Health over time

What it shows. One colour per time step for three indicators: experience score, latency and load vs limit. Read it left to right to see when things went wrong.

How it is measured. Steps of the selected period (e.g. 5 min over 24 h). Steps where a client was at its plan limit are left out of latency and score.

Scale.

- Green: green: good
- Amber: orange: fair / watch
- Red: red: poor / saturated
- Grey: grey: no measurement in that step

How to read it. For a node: the worst client of the step. For the whole network: all clients together. A red band at the same hour every evening is a capacity problem; a red band at random times is more often radio.

Traffic

Who consumes what, where the traffic goes, and the traffic restrictions.

Who consumes

What it shows. Volume per subscriber over the period.

How it is measured. From NetFlow, each client counted once per direction: downloads where they are seen best, uploads likewise (the best exporter for each direction, never two added up).

How to read it. This is volume (GB), not speed: a client streaming all evening outweighs one who ran a short speed test.

Which services the traffic comes from

What it shows. Traffic grouped by recognised service (YouTube, Netflix, Steam, Microsoft updates…).

How it is measured. The remote address of each flow is matched with a catalogue of published address blocks, then with its reverse name and owner. Content is never inspected — it stays encrypted.

How to read it. “unknown” is traffic to addresses that match no catalogue entry yet; it shrinks as names are resolved.

Where the traffic goes

What it shows. Map of the destinations your clients reach: each country and city sized by the volume exchanged with it.

How it is measured. From NetFlow: each remote address is located from its address block (geolocation database). Hover a point for its volume and clients; click it to open the detail of that place.

How to read it. A location is that of the server’s block, not of the company: a CDN often answers from a nearby city even for a foreign service.

Who talks to whom, client by client

What it shows. Each client and the internet addresses it exchanges with: volume, rate while active and live rate.

How it is measured. NetFlow records of the period, grouped by client and remote address.

How to read it. “Rate when active” divides by the time the conversation was really active, so a 10-second download at 100 Mbps shows 100 Mbps — not the 1 Mbps a whole-hour average would give.

Destinations reached

What it shows. Internet addresses reached by your clients, with their owner, location and volume.

How it is measured. From NetFlow; names come from the catalogue, reverse DNS and (if enabled) the registry.

Find an IP

What it shows. Look up any IP address or domain: owner, location, recognised service, and which of your clients reach it.

How to read it. Useful when a client complains about one site: see whether others reach it too and how much traffic it carries.

Traffic restrictions

What it shows. Rules that block or cap traffic towards a service, a category or an address range — for everyone, some sites, or some clients.

How it is measured. Written on the routers as address lists plus firewall (block) or mangle + queue tree (cap). The address list is refreshed automatically from the catalogue and from what NetFlow discovers.

What to do. A rule that seems to have no effect: check “Last applied” and that the router is reachable; new addresses join the list at the next refresh.

Network tree

The discovered tree, editable with the mouse: drag a box onto another to re-parent it, correct a role, create or remove a link, merge two boxes that are the same device, hide a box. Each link carries its measured throughput. *Forget vanished devices* removes boxes not seen for a chosen time; declared routers and hand-made links are never removed.

Plans

Which rate for whom: the default plan, every client's plan and its source, and the packages pushed through the API.

Default plan

What it shows. The limit given to a client pushed by the API (or entered) without a rate. A client only DETECTED on a router, with no plan pushed, is observed and never throttled.

How to read it. 0 = no limit. To also cap detected clients, turn on default_plan_for_detected_clients in Settings. Changing it re-writes the queues of the clients that use it.

Clients

What it shows. Each client and the plan applied to it, with where that plan comes from.

How to read it. Source: pushed by the API (billing / CRM), set by hand here, or the default plan. A limit forced in Subscribers (Rate) overrides all three — it is shown here so you know why the plan is not what is applied.

Packages pushed by the API

Offers received from your billing system or CRM through the public API (/model/v1/packages). Clients pushed with a package get its rates automatically.

Insights

The commercial reading of the measurements, comparing the chosen period with the one before.

At risk of leaving

What it shows. Clients showing the signs that come before a cancellation.

How it is measured. Any of: experience score under 50 over the period; usage down more than 70% versus the previous period; no traffic at all for 3 days or more (for a client that used to consume).

What to do. Call before they do: a CPE swap, a realignment or a plan change is cheaper than a lost client.

Ready for a bigger plan

What it shows. Clients that live at their plan ceiling while their experience stays good — the plan limits them, not the network.

How it is measured. At 90%+ of their plan in 15% or more of the samples of the period, with a score of 50 or more.

How to read it. A client at its ceiling with poor latency is not here on purpose: selling it more would not fix what is a network problem.

Sites and access points room for more subscribers

What it shows. How many more subscribers each site or access point can take at today’s usage.

How it is measured. Busy-hour peak per subscriber, against a target of 80% of the capacity (above that, queues fill and latency rises before the link is full).

How to read it. A site where 20% or more of the clients already have a poor experience shows no room, whatever the margin on paper.

Every subscriber plan, usage, experience

What it shows. All subscribers with their limit, current usage, latency and whether the router really holds the cap.

How to read it. Click a client to see its plan, its queue on the router and its history. The search box accepts a login, an IP, a MAC or a site.

Devices

Routers and antennas: their health, adding them, the Ubiquiti UISP controller (connect it, see every antenna's radio capacity, attach backhauls to their site), and the inventory. One section at a time: the table of contents works as tabs.

Router health

What it shows. CPU, memory and uptime of each polled router.

Scale.

- Green: CPU under 70%: fine
- Amber: 70–90%: busy — queues and NetFlow may lag
- Red: over 90%: overloaded — packets and measurements suffer

What to do. A sustained high CPU on a PoP: too many simple queues for the hardware, or a firewall rule doing heavy work.

Polled routers

What it shows. The routers freeQoS reads (and writes queues to).

How to read it. Reset queues deletes every freeQoS queue of that router and rewrites them cleanly — use it when caps behave strangely. Queues not created by freeQoS are never touched.

Connect a router

What it shows. Add a MikroTik router: its address, API port and an account with read rights (and write rights if freeQoS should shape).

How to read it. The API-SSL port (8729) is preferred. A dedicated account with the “read, write, api, test” policies is enough — “test” is needed for the latency probe.

Ubiquiti antennas

airOS access points and CPEs polled for their radio state: signal, noise, CCQ, capacity. Their capacity feeds the saturation and room-for-more calculations.

Radio health access points and their CPEs

What it shows. Each access point and the CPEs connected to it, with the radio quality of each link.

Scale.

- Green: signal above −68 dBm, SNR above 25 dB, CCQ above 90%
- Amber: signal −68 to −75 dBm, SNR 20–25 dB
- Red: signal below −75 dBm, SNR under 20 dB or CCQ under 75%

What to do. Weak signal on one CPE: alignment or obstacle. Weak on all CPEs of an AP: interference — check the noise floor and change channel.

Add an antenna

Declare an airOS device by its address and credentials. Credentials are encrypted at rest with the controller key.

Inventory

Every device freeQoS knows, with where it was discovered.

Known sites

The PoPs and VLAN sites known to freeQoS. A site pushed by the API takes the place of an automatically discovered one.

API

API keys, and the way to the API guide.

Create a key

What it shows. Create a key for an external system (billing, CRM) to call the public API.

How to read it. The key is shown once — copy it then. “read” gives GET; “write” adds PUT and DELETE. Give each system its own key so you can revoke one without breaking the others.

Endpoints

What it shows. The routes an external application (billing, CRM, provisioning) calls with a key.

How to read it. /model/v1 and /usage/v1: clients, packages, sites, sectors and usage. /api/v1: everything the interface does — routers, plans, forced limits, boosts.

Example

A ready-to-run call with curl. The key goes in Basic authentication as the username, with an empty password.

Settings

Accounts, operating settings, and everything about writing to the routers.

Accounts

Who can log in to this interface, your own password and sessions, and (for edit accounts) the journal of logins.

Who can log in

What it shows. The accounts of this interface.

Scale.

- Grey: Read only: sees everything; every change is refused by the server
- Green: Edit: can change everything, including accounts

How to read it. There is always at least one active edit account: the last one cannot be deleted, demoted or disabled. Changing someone’s role or password logs them out everywhere.

My password

What it shows. Change your own password. Your other browsers are logged out; this one stays connected.

How to read it. At least 12 characters. Passwords from attacker lists (Password2024!, azerty123…), repetitive ones or ones containing your email are refused. A short sentence is both stronger and easier to remember than a short complicated word.

My sessions

What it shows. Every browser currently logged in to your account: device, address, when it logged in and when it was last active.

How it is measured. A session ends after 24 h without activity, and after 30 days in any case. At most 10 per account — the oldest is closed beyond that.

What to do. A device or address you do not recognise: log it out, then change your password.

Login journal

What it shows. Every login, failed attempt, block, logout and account change, with address and device. Kept 6 months.

Scale.

- Green: Login
- Red: Failed login / Blocked: someone typed a wrong password
- Amber: Account changed or deleted

How to read it. After 5 failures the email and the address are blocked 5 min, then 10, 20… up to 1 h if it continues. Many failures from one unknown address = someone guessing.

What to do. Repeated failures on a real account from an unknown address: change that password and, if exposed to the internet, restrict access to the interface.

Operational settings

What it shows. Settings stored in the database and applied without restart.

How to read it. “default” = the value from the environment / built-in default. Apply writes it; the source column then says “database”.

Services and IP location

What it shows. What NetFlow keeps about the destinations your clients reach, and how addresses are named (catalogue, reverse DNS, registry, geolocation).

How to read it. Volumes per client are measured whatever these say; they only decide what is kept per destination and how much naming work (and outbound queries) the controller does.

Shaping

What it shows. How queues are computed and written: safety factor on measured capacity, floor rate, default plan, adoption of queues freeQoS did not create, and how a queue targets the client.

How to read it. A change here is applied at the next reconciliation (or right away with Reset queues in Devices).

CAKE

What it shows. Parameters of the CAKE queues — the queue discipline that keeps latency low while a line is full.

How to read it. Keep the defaults unless you know why: overhead and MPU must match the encapsulation (PPPoE, VLAN) for the rate to be exact; rtt sets how fast CAKE reacts.

Write safeguards

Limits that stop an automatic loop from changing too much at once (circuit breaker), and whether a separate write account is required on the routers.

Traffic

Where the volume of each client is counted when both the internet edge and the PoPs export NetFlow — so a byte is never counted twice.

Collection cadences

What it shows. How often each collection runs.

How to read it. Shorter = fresher figures, but more load on the routers and the database. A 10 s subscriber poll and a 30 s latency probe suit most networks.

What stays out of reach of the interface

Settings that can only change in the environment (.env) — the database address, the encryption key… — because the interface itself depends on them.

Shaping and writing to the routers

What it shows. Whether freeQoS writes queues by itself.

How to read it. On: a reconciliation every 2 min writes any missing or wrong queue. Off: it writes only when you act (Rate, plan change, Reset queues). Either way, only queues marked freeqos:managed are ever modified.

Are the caps actually held

What it shows. Compares the limit freeQoS intends for each client with what is really written on its router.

How to read it. A mismatch means the router was changed by hand, a queue failed to write, or another queue matches first (RouterOS applies the first matching queue).

What to do. Use Devices › Reset queues on that router to rebuild its freeQoS queues from scratch.

Log of commands sent

Every command freeQoS sent to a router: when, by whom (account or automatic loop), what, and the result. The first place to look when a cap changed unexpectedly.

Detail panels

What opens when you click a link, a node, a client or an address.

Right now

Last 10-second measurement.

Clients on this link

What it shows. The clients of this node with how much of their limit they use, their latency and score.

How to read it. Several clients red on latency at the same time as the link is loaded = the link is the bottleneck. Only one client red = look at that client.

Link

What it shows. What freeQoS knows about this node's link: capacity, load, limits applied and plans sold behind it.

How to read it. Capacity unknown: set the rate of the link in the Network tree so load and oversubscription can be judged.

Plan

What it shows. The plan of this client, where it comes from, and the limit really applied.

How to read it. When "limit applied" differs from the plan, the limit was forced by hand (Rate) or a temporary boost is running.

Plan usage

What it shows. How much of its limit the client uses right now, and its latency at rest and under load.

How to read it. At 85%+ of its plan the client is "at plan limit": its latency then reflects its own full queue and is not held against the network.

Limit usage

How much of its forced limit the client uses right now, and its latency at rest and under load.

Who reaches this address

Your clients that exchanged traffic with this address over the period, with the volume of each.

Columns and figures

Every other label that carries an (i) in the interface, in alphabetical order. *Where* names the place when the same word means something else elsewhere.

Label	Where	Meaning
Access		PoP to its subscribers: median of their medians (p90 when hovering). The part of the latency your access network adds.
Access level		What the account may do. Each level includes the previous ones; the server enforces them. Viewer: sees everything, changes nothing. Operator: bandwidth mode, boost and plan of clients, one or several at a time. Administrator: everything else, such as adding clients, PoPs, routers, antennas and their API (Ubiquiti), modes, settings, API keys and accounts.
Account	Accounts	The account concerned (for a failed login, the email that was typed — it may not exist).

Label	Where	Meaning
Address	Accounts	IP address the session was opened from (behind a reverse proxy, set FORWARDED_ALLOW_IPS so this is the real client).
Address		IP address (or block) of the client: its queue targets this address.
Addresses		Number of distinct internet addresses reached.
At ceiling		Share of the time the client was at 90%+ of its limit over the period. 15% or more with a good experience: candidate for a bigger plan.
At plan limit		Clients that used their whole plan the whole period. Their latency was that of their own full queue — a client downloading at its cap delays its own pings. That is the plan doing its job, not a network fault, so they are not counted as poor.
Avg download		Average download over the period.
Avg rate		Average rate over the period (volume ÷ period).
Backhaul capacity		Sum of the measured capacity of the radio backhauls declared in the inventory. "-" = no radio backhaul declared.
Boost		Temporary extra rate given to the client; it ends on its own at the time shown.
Bufferbloat		Grade of the latency added when the line is loaded. Scale: Green, A+ / A: up to +30 ms — imperceptible; Amber, B / C: +30 to +100 ms — calls degrade while someone downloads; Red, D / F: over +100 ms — real-time use breaks under load.
Busiest node		The node using the largest share of its limit right now. Scale: Green, under 70%: comfortable; Amber, 70 to 90%: watch it — peaks start to queue; Red, over 90%: saturated — latency rises for everyone behind it.
Busy-hour peak		Highest load reached during the busiest hour of the period — what the capacity must hold.
By	Accounts	Who made the change, when it is not the account itself (e.g. an administrator resetting a password).
Capacity		What the link can carry. The lowest of: the rate set by hand on the link, the capacity measured on the radio, and the port speed. "unknown" = none of the three is known: load and saturation cannot be judged for this link.
Capacity / limit	Table column	On a node row: what its uplink (towards the core or the transit) can carry. On a client row: the cap applied to the client. Wired link: the capacity declared, or the port speed. Radio link: the capacity its antenna announces live (airOS / UISP). Auto: the lowest known. A node has no plan — its clients do. Its download / upload % is taken against this capacity. Grey "limits" = uplink capacity unknown: the sum of the clients' limits is shown instead.
Capacity read		Capacity read from the radio.
Category		Family of the service: streaming, gaming, voice/video, CDN, cloud, updates...

Label	Where	Meaning
CCQ		Client Connection Quality: share of radio frames that went through without retransmission. Scale: Green, 90 to 100%; Amber, 75 to 90%; Red, under 75%: flagged as an issue — retransmissions eat throughput and add latency.
Cities		Number of distinct cities among the addresses reached.
Client		The subscriber: PPPoE login, or the reference of a static-IP client.
Client experience		Share of measured clients with a good experience. Clients at their plan limit and clients not measured are counted apart, so neither makes the network look worse than it is. Scale: Green, 80 to 100: good; Amber, 50 to 79: fair — noticeable at busy times; Red, under 50: poor — the client feels it; Grey, grey / "-": not measured yet.
Clients	Table column	Number of clients of this node.
Clients	Routers — live traffic	Sum of the traffic of this router's subscribers (their queues).
Clients measured		Clients that answered the latency probe over the period.
Clients seen		Clients with traffic seen over the period. Each client is counted once, even if its traffic crosses several routers.
Country		Country of the address, from the geolocation of its block.
CPU		Processor load of the router. Scale: Green, under 70%; Amber, 70 to 90%; Red, over 90%.
Created by	Accounts	The account that created this one ("setup" = the very first account).
Datagrams		NetFlow packets received from this exporter. Not increasing = the router stopped exporting, or a firewall drops UDP 2055.
Destination		Internet address the client exchanged traffic with.
Detail	Accounts	Extra information: new role, why it failed, how long the block lasts...
Device	Accounts	Browser and system, read from the browser's User-Agent.
Distance		Radio distance between the access point and the CPE.
Down		Volume received by the clients.
Download		What your clients receive, from the internet to them. Sum of the subscriber queues (tx side of the router), read every 10 s.
Download now		Current download, with its share of the limit.
Downstream		Total received by your subscribers over the period, at the counting point (each byte counted once).

Label	Where	Meaning
Effect		Block the traffic, or cap it at a rate.
Event	Accounts	What happened: login, failed login, block, logout, account change...
Experience		Score from 0 to 100 summing up what the client feels. The weaker of two parts: latency at rest (no penalty up to 10 ms, then −0.6 point per ms) and the latency added under load (bufferbloat: +5 ms ≈ 90, +30 ms ≈ 80, +100 ms = 50, +200 ms = 25). The weakest part decides: low latency does not make up for heavy bufferbloat, and the reverse. Scale: Green, 80 to 100: good; Amber, 50 to 79: fair — noticeable at busy times; Red, under 50: poor — the client feels it; Grey, grey / "-": not measured yet.
Experience score	Health over time	Experience score of the step (the worst client of the node). Scale: Green, 80 to 100: good; Amber, 50 to 79: fair — noticeable at busy times; Red, under 50: poor — the client feels it; Grey, grey / "-": not measured yet.
Fair		Noticeable but usable: latency 30–100 ms, spikes, a little loss, or +30 to +150 ms under load.
Flows		Number of NetFlow records counted.
For whom		Every client, some sites, or some clients.
Forced rate		Limit set by hand in Subscribers (Rate): it overrides the plan until you clear it.
From		Start of the link (the upstream equipment).
Good		Latency under 30 ms, stable, no loss, little or no bufferbloat.
Headroom		Capacity left at the peak of the period. Under 10% at the peak: the next growth in usage will show as latency.
Interface		Router interface concerned.
Internet	Routers — live traffic	What this router exchanges with the outside on its uplink port, with the port load.
Internet edge	Traffic	NetFlow measured at the internet exit (above the core). In Automatic mode, counted for each direction it sees best: each client once, internet traffic only. Behind NAT, downloads are tied back to the client by the translated address or by matching them with their upload.
Internet side	Saturation risks	Gateway uplink and PoP-to-core links: a saturation here hits every client.
Jitter		How much latency varies from one ping to the next in the last series. Scale: Green, under 10 ms: smooth; Amber, 10 to 30 ms: calls may crackle; Red, over 30 ms: choppy voice and video.
Kind		PPPoE subscriber (discovered from the router sessions) or static-IP client (declared by hand).
Last activity	Accounts	Last request made with this session.

Label	Where	Meaning
Last applied		When the rule was last written on the routers.
Last change		When the plan was last changed, and by what (API, an account, the default).
Last login	Accounts	Last successful login of the account.
Latency	Health over time	90th percentile of the latency of the step: 9 pings out of 10 were faster. Scale: Green, under 30 ms: good — calls, games and browsing feel instant; Amber, 30 to 100 ms: noticeable — fine for streaming, sluggish for games; Red, over 100 ms: poor — video calls stutter, pages hesitate.
Latency		Round-trip time between the PoP router and the client — how long a packet takes to go and come back. The router pings the client 5 times every 30 s (200 ms apart, from its loopback). The median of the series is shown, so one slow ping does not colour the client. “no reply” = the whole last series was lost. If every client of a router shows it at once, the probe is failing (firewall, source address, account without the “test” right) — not the clients. Scale: Green, under 30 ms: good — calls, games and browsing feel instant; Amber, 30 to 100 ms: noticeable — fine for streaming, sluggish for games; Red, over 100 ms: poor — video calls stutter, pages hesitate.
Limit		The cap really applied to the client. In order of priority: the limit forced by hand in Subscribers (Rate), otherwise its plan, otherwise the default plan.
Limit applied		The cap written on the router, when it differs from the plan (forced by hand, or a boost).
Limits applied		Sum of the limits really applied to the clients (forced, plan or default).
Live		Current rate of the conversation, from the real duration of its last NetFlow records. Empty = not active right now.
Load		Busiest direction compared with the port speed. Scale: Green, under 70%: comfortable; Amber, 70 to 90%: watch it — peaks start to queue; Red, over 90%: saturated — latency rises for everyone behind it.
Load now		Current traffic of the node.
Load vs limit	Health over time	For a node: its most loaded client against its own limit. For the whole network: total traffic against the sum of limits. Scale: Green, under 70%: comfortable; Amber, 70 to 90%: watch it — peaks start to queue; Red, over 90%: saturated — latency rises for everyone behind it.
Location		Approximate location of the address (city, country) from its block — an indication, not a proof.
Loss		Share of pings lost in the last series of 5. A loss while the client is at its plan limit is not counted against the network: its own traffic filled the queue. Scale: Green, 0%; Amber, under 2%: occasional; Red, 2% and more: calls cut, downloads slow down.
Measured rate		Rate measured on the router for this client.

Label	Where	Meaning
Memory		Memory used on the router.
Method		HTTP method: GET reads, PUT creates or replaces, DELETE removes.
Next hop up		Router to its default gateway — the next hop up: the core for a PoP, the gateway for the core, the transit provider for the gateway.
Node		A site (PoP or router) and its clients. A VLAN client is counted with the router that carries it.
Noise		Radio noise floor, in dBm (lower is better, e.g. −95 is quieter than −85). A high noise floor means interference on the channel.
Not from clients	Routers — live traffic	Internet minus clients: traffic of no known client. Bandwidth tests, device management, routers talking to each other, or clients not declared yet. A large, steady share is worth investigating.
Object		The collection concerned: accounts, packages, sites, access points, services.
On the router		What is really written on the router for this client.
Opened	Accounts	When this session logged in.
Origin		Where the item comes from.
Oversubscription		Plans sold ÷ capacity of the link. Some oversubscription is normal — clients are not all active at the same time. Watch the evening peak rather than the ratio alone. Scale: Green, up to 1×: everything sold fits at once; Amber, 1 to 3×: usual for residential access; Red, over 3×: the link will be full at busy hours.
P95		The bad moments: 95% of the pings were faster than this. A p95 far above the median (3× or more, and over 100 ms) means spikes: the line is fine most of the time but stalls now and then — typical of radio interference or a queue filling up.
Package		Offer pushed by the API.
Path		Route of the call, after the base URL.
Plan	Table column	The plan sold to the client (download / upload).
Plan used		Current traffic as a share of the client's limit. At 85% and above the client is "at plan limit": its latency is then left out of its verdict.
Plans		Sum of the plans sold. "no plan" = the clients are limited by hand or by the default plan.
Poor experience		Clients the network is failing right now. Median over 100 ms, 2%+ loss, +150 ms or more under load, or an experience score under 50.
PoP		Point of presence the client is attached to.

Label	Where	Meaning
PoP side	Saturation risks	Links towards subscribers, VLANs and relays: a saturation only hits what hangs below.
PoPs	Traffic	NetFlow measured on each PoP router, next to the clients, before any NAT. In Automatic mode, counted for each direction it sees better than the internet exit.
Port		Service port on the remote side (443 = HTTPS, 53 = DNS…). “-” for ICMP (ping).
Port speed		Negotiated speed of the port (for a VLAN or bridge: the speed of the physical port carrying it).
Prefix		Address block.
Proto		Transport protocol: tcp, udp, icmp…
Published prefixes		Address blocks announced by this owner on the internet.
Queue		freeQoS queue on the router for this client.
Radio		Radio link: its capacity is read live on its antenna and follows rain fade, interference and alignment. Scale: Green, live: at or near its nominal capacity; Amber, under 70% of nominal: degraded; Red, silent: no recent reading — capacity unknown.
Rate TX/RX		Radio link rates (transmit / receive).
Rate when active		Volume divided by the time the conversation was really active. Not divided by the whole period: a 10-second download at 100 Mbps reads 100 Mbps, which is what the client experienced.
Received		Volume received by the client from this address over the period.
Reverse name		Name returned by reverse DNS for this address.
Role	Accounts	Read only: sees everything, changes nothing (refused by the server, not just hidden). Edit: can change everything, including accounts.
Role		Role of the router: PoP (serves clients), core, or gateway (internet exit).
Room for		How many more subscribers this site can take at the current usage. Busy-hour peak per subscriber against 80% of the capacity.
Router		The router concerned.
Routers		Each router, live: what it exchanges with the internet on its uplink, what its own clients consume, and the gap between the two. Uplink port counters and subscriber queues, read every 10 s. A large gap (“not from clients”) is traffic of no known client: tests, management, or clients not yet declared.
Sample		Age of the last measurement. “stale” = the figures are not current: the collection of this router is late or stopped.

Label	Where	Meaning
Sampling		“all” = every flow is exported; 1:N = one in N, volumes are multiplied back by N.
Score		Experience score from 0 to 100: the weaker of latency at rest and latency added under load (bufferbloat). Scale: Green, 80 to 100: good; Amber, 50 to 79: fair — noticeable at busy times; Red, under 50: poor — the client feels it; Grey, grey / “-”: not measured yet.
Seen		Last time it was seen.
Sent		Volume sent by the client to this address over the period.
Service		Recognised service behind the address: from the catalogue of published blocks, the domain name or the owner of the address.
Session		How long the PPPoE session has been up (static clients have none). A session that keeps restarting points to a CPE or radio problem.
Share		Share of the total traffic.
Signal		Strength of the radio signal received, in dBm (closer to 0 is stronger). Scale: Green, above −68 dBm; Amber, −68 to −75 dBm; Red, below −75 dBm: weak — alignment or obstacle.
Site		PoP or VLAN site the client belongs to.
SNR		Signal-to-noise ratio, in dB: signal minus noise. Scale: Green, above 25 dB: clean; Amber, 20 to 25 dB: usable, lower modulation; Red, under 20 dB: flagged as an issue — unstable link.
Sold		Sum of the plans sold to these clients. “no plan” = limited by hand or by default.
Sold (down)		Sum of the download plans sold on this site.
Sold (up)		Sum of the upload plans sold on this site.
Sold / plan	Table column	On a node row: the plans sold behind it divided by its uplink capacity. On a client row: its plan. Scale: Green, up to 1×: everything sold fits at once; Amber, 1 to 3×: usual for residential access; Red, over 3×: the uplink will be full at busy hours.
Sold throughput		Sum of the download plans of all subscribers, and how much of it is used right now. A few percent is normal: clients are not all active at the same time. That is what makes oversubscription possible.
Source		Where the plan comes from: pushed by the API, set by hand, or the default plan — or a limit forced in Subscribers, which overrides all three.
State	Accounts	A disabled account can no longer log in, and its open sessions are closed at once.
State		Current state.
Subscriber		The subscriber: PPPoE login, or the reference of a static-IP client.
Subscribers		Number of subscribers.

Label	Where	Meaning
Subscribers online		Subscribers with a measurement in the last 2 minutes (PPPoE session up, or static client with traffic).
Target		Address or block the queue applies to.
Throughput		Current download and upload.
To		End of the link (the downstream equipment).
To clients		Traffic going towards the clients on this port.
To internet		Traffic going towards the internet on this port.
Towards		Equipment on the other side of the port, when known (from discovery or the network tree).
Traffic identified		Share of the traffic flows tied to a known client (the figure is since the controller started). The rest is network management, routers talking to each other, or addresses of clients not declared yet — those appear in the entry-aid list.
Traffic now		Download of all subscribers at the last measurement; upload below. Sum of the subscriber queues, read every 10 s. Compared with the backhaul capacity, or with the sum of limits when no capacity is declared.
Type		How the link was found: discovered on the router, observed in traffic, or set by hand.
Under load		Latency while the client's line is busy, and how much the load adds over rest (bufferbloat). Pings taken while the client was transferring are compared with pings taken at rest. Scale: Green, A+ / A: up to +30 ms — imperceptible; Amber, B / C: +30 to +100 ms — calls degrade while someone downloads; Red, D / F: over +100 ms — real-time use breaks under load.
Up		Volume sent by the clients.
Upload		What your clients send, from them to the internet. Sum of the subscriber queues (rx side of the router), read every 10 s.
Upload now		Current upload, with its share of the limit.
Upstream		Total sent by your subscribers over the period, at the counting point (each byte counted once).
Uptime		Time since the last restart. A recent restart nobody planned is worth a look (power, crash).
Usage		Detected application of the conversation.
Vantage		Where this exporter measures: internet edge (above the core) or at a PoP. Set automatically for your own routers. Each direction is read from one vantage only (the one that sees it best), so a byte seen at both is counted once.
Version		NetFlow format received: v5, v9 or IPFIX.

Label	Where	Meaning
Vs limit		Current download as a share of the limit. Scale: Green, under 70%: comfortable; Amber, 70 to 90%: watch it — peaks start to queue; Red, over 90%: saturated — latency rises for everyone behind it.
Wanted		What freeQoS intends to write.
When	Accounts	Date and time of the event (hover for the full date).
Where		Which segment adds the delay: access side, towards the core / gateway, or above it (transit, internet).
Where the cap comes from		Origin of the limit: plan, forced by hand, boost or default plan.
Why		What lowered the verdict, in plain words: high latency, spikes, loss, bufferbloat — or “at plan limit”.
Why it does not throttle		Reason why the cap is not held on the router (no queue, another queue matching first, a parent queue blocking...).
Worst latency		Latency of the worst client of the node; “no reply” if one of them lost all its pings.
Worst score		Lowest experience score among the clients of the node. Scale: Green, 80 to 100: good; Amber, 50 to 79: fair — noticeable at busy times; Red, under 50: poor — the client feels it; Grey, grey / “-”: not measured yet.

Appendix B. API endpoint index

The server exposes 207 operations. This index is generated from its own OpenAPI description, so it cannot drift from the code; the *API guide* (`/api-guide`) gives the parameters and bodies of each one.

Public API (model)

Method	Path	What it does
`GET`	`/model/v1`	Available collections
`GET`	`/model/v1/{collection}`	List a collection
`GET`	`/model/v1/{collection}/{object_id}`	Read a record
`PUT`	`/model/v1/{collection}/{object_id}`	Create or replace a record
`DELETE`	`/model/v1/{collection}/{object_id}`	Remove a record

Public API (usage)

Method	Path	What it does
`GET`	`/usage/v1`	What this API returns
`GET`	`/usage/v1/services`	Usage of every service
`GET`	`/usage/v1/services/{service_id}`	Usage of one service

PoPs

Method	Path	What it does
`GET`	`/api/v1/pops/health`	Health of the collected routers (CPU, memory, uptime)
`GET`	`/api/v1/pops/provisioning/{router_name}`	Progress of the automatic setup
`POST`	`/api/v1/pops/provisioning/{router_name}`	Run the automatic setup again
`GET`	`/api/v1/pops/read-status`	Last successful read of each router's sessions
`GET`	`/api/v1/pops/routers`	Router inventory (file + database)
`POST`	`/api/v1/pops/routers`	Save a router
`DELETE`	`/api/v1/pops/routers/file/{name}`	Remove a router from the file inventory
`POST`	`/api/v1/pops/routers/file/{name}/restore`	Show a previously removed file router again
`PUT`	`/api/v1/pops/routers/file/{name}/write`	Allow or stop writing on a router of the file inventory

Method	Path	What it does
`POST`	`/api/v1/pops/routers/test`	Test a connection without saving it
`PATCH`	`/api/v1/pops/routers/{router_id}`	Edit a router
`DELETE`	`/api/v1/pops/routers/{router_id}`	Remove a router from the inventory
`POST`	`/api/v1/pops/routers/{router_id}/probe`	Test a saved router

Plans

Method	Path	What it does
`GET`	`/api/v1/plans`	The plan of every client, and where it comes from
`POST`	`/api/v1/plans/refresh`	Apply the plans now (after a default-plan change)
`PUT`	`/api/v1/plans/{login}`	Set the plan of a client (even before its first connection)
`DELETE`	`/api/v1/plans/{login}`	Remove a client's plan (observed, not throttled)

Shaping

Method	Path	What it does
`POST`	`/api/v1/shaping/apply`	Apply a plan (writes to the router)
`GET`	`/api/v1/shaping/audit`	Log of the commands sent
`GET`	`/api/v1/shaping/boosts`	Running boosts
`POST`	`/api/v1/shaping/boosts`	Give a temporary rate boost
`DELETE`	`/api/v1/shaping/boosts/{login}`	Remove a boost before it expires
`GET`	`/api/v1/shaping/capability`	Real rights of the account on a router
`GET`	`/api/v1/shaping/enforcement`	State of the write flag
`PUT`	`/api/v1/shaping/enforcement`	Allow or stop writing to the routers
`GET`	`/api/v1/shaping/limits`	Are the applied caps actually held?
`POST`	`/api/v1/shaping/plan`	Compute the commands, without sending anything
`GET`	`/api/v1/shaping/points`	Where shaping applies on the network
`GET`	`/api/v1/shaping/policies`	Rate overrides set by hand
`PUT`	`/api/v1/shaping/policies`	Set the rate of a link or a subscriber

Method	Path	What it does
`DELETE`	`/api/v1/shaping/policies/{scope}/{target_key}`	Remove an override
`GET`	`/api/v1/shaping/qoe`	What the closed QoE loop decided, per sector
`POST`	`/api/v1/shaping/qoe/run`	Trigger one pass of the closed QoE loop
`POST`	`/api/v1/shaping/routers/{router_name}/clean-queues`	Delete every freeQoS queue of a router, then lay them down again
`GET`	`/api/v1/shaping/state`	What is already configured on the routers
`GET`	`/api/v1/topology`	Network graph as the controller understands it
`POST`	`/api/v1/topology/discover`	Run topology discovery again
`POST`	`/api/v1/topology/forget-stale`	Forget the vanished devices of the tree
`POST`	`/api/v1/topology/links`	Create a link by hand between two nodes
`DELETE`	`/api/v1/topology/links/{key}`	Remove a link from the tree
`GET`	`/api/v1/topology/links/{key}/live`	Measure this link now
`GET`	`/api/v1/topology/links/{key}/throughput`	Measured throughput of a link
`POST`	`/api/v1/topology/merge`	Declare that two boxes are the same device
`DELETE`	`/api/v1/topology/merge/{alias_key}`	Undo a manual merge
`PATCH`	`/api/v1/topology/nodes/{key}`	Correct the role of a device
`PATCH`	`/api/v1/topology/nodes/{key}/layout`	Move a box of the tree
`PATCH`	`/api/v1/topology/nodes/{key}/parent`	Re-parent a box (drag a link)
`PATCH`	`/api/v1/topology/nodes/{key}/visibility`	Hide or show a box again
`GET`	`/api/v1/topology/routers/{router_name}/export`	Full config of a PoP

Static clients

Method	Path	What it does
`GET`	`/api/v1/pops/census`	Census of the clients of a PoP, every kind
`GET`	`/api/v1/static-clients`	Inventory of static-IP clients
`POST`	`/api/v1/static-clients`	Declare a static-IP client
`GET`	`/api/v1/static-clients/candidates`	Addresses detected on a routed VLAN, undeclared

Method	Path	What it does
GET	/api/v1/static-clients/candidates/diagnostic	Why a client on a VLAN is seen, or is not
GET	/api/v1/static-clients/enforcement	Queue state of every declared client, and what is missing
GET	/api/v1/static-clients/vlans	Declared clients, grouped by VLAN
PATCH	/api/v1/static-clients/{client_id}	Edit a static-IP client
DELETE	/api/v1/static-clients/{client_id}	Remove a client from the inventory

Metrics

Method	Path	What it does
GET	/api/v1/backhauls	List of radio backhauls
GET	/api/v1/backhauls/latest	Last known capacity per backhaul
GET	/api/v1/backhauls/{backhaul_id}/metrics	Capacity series of a backhaul
GET	/api/v1/bufferbloat	Bufferbloat grade (latency under load) per subscriber
GET	/api/v1/collection/cycles	State of the measurement cycles
GET	/api/v1/heatmap	Executive heatmap: QoE / RTT / utilisation over time
GET	/api/v1/insights/capacity	How many more subscribers each site can take
GET	/api/v1/insights/subscribers	Churn risk and upgrade candidates
GET	/api/v1/network/tree	PoP tree → backhauls, capacity and load
GET	/api/v1/overview	Headline figures of the dashboard
GET	/api/v1/pops	List of PoPs
POST	/api/v1/pops/orphans/delete	Delete every site no device declares
DELETE	/api/v1/pops/{pop_id}	Remove a PoP and its data
GET	/api/v1/ports/live	Every router port with its current throughput
GET	/api/v1/search	Instant lookup: subscriber, IP, MAC, device, site
GET	/api/v1/subscribers	List of subscribers
GET	/api/v1/subscribers/latest	The subscribers of a PoP and their last sample
GET	/api/v1/subscribers/{subscriber_id}	Record of one subscriber

Method	Path	What it does
`DELETE`	`/api/v1/subscribers/{subscriber_id}`	Delete a subscriber and its history
`GET`	`/api/v1/subscribers/{subscriber_id}/live`	Live check of one subscriber
`GET`	`/api/v1/subscribers/{subscriber_id}/metrics`	Throughput series of a subscriber
`GET`	`/api/v1/throughput`	Aggregate network throughput over time

Capacity

Method	Path	What it does
`GET`	`/api/v1/capacity`	Sold capacity, link occupancy, real usage
`GET`	`/api/v1/capacity/hotspots`	Saturation points: headroom left on every link
`GET`	`/api/v1/capacity/media`	Wired or radio, link by link, and the antennas to pick
`PUT`	`/api/v1/capacity/media`	Declare a link wired (fixed capacity) or radio (live)
`DELETE`	`/api/v1/capacity/media/{router_name}/{interface}`	Back to automatic (smallest known capacity)

Traffic (netflow)

Method	Path	What it does
`GET`	`/api/v1/netflow/applications`	Traffic breakdown by usage
`GET`	`/api/v1/netflow/catalogue`	Services the controller can recognise
`GET`	`/api/v1/netflow/connections`	Live client connections
`GET`	`/api/v1/netflow/destinations`	Destinations reached by the clients
`GET`	`/api/v1/netflow/destinations/{address}`	Detailed record of a destination reached
`POST`	`/api/v1/netflow/destinations/{address}/resolve`	Analyse an address again
`GET`	`/api/v1/netflow/export`	NetFlow export configured on the routers
`POST`	`/api/v1/netflow/export/apply`	Configure the NetFlow export on the routers
`GET`	`/api/v1/netflow/exporters`	Devices that export flows
`POST`	`/api/v1/netflow/exporters`	Declare an exporter (or fix its declaration)
`PATCH`	`/api/v1/netflow/exporters/{exporter_id}`	Edit an exporter

Method	Path	What it does
`DELETE`	`/api/v1/netflow/exporters/{exporter_id}`	Remove an exporter from the list
`POST`	`/api/v1/netflow/flush`	Write the current window right away
`GET`	`/api/v1/netflow/hosts`	Addresses seen in the flows and matched to no record
`GET`	`/api/v1/netflow/intel`	State of address enrichment
`POST`	`/api/v1/netflow/intel/run`	Name the pending addresses right away
`GET`	`/api/v1/netflow/locations`	Where the traffic goes: volume by place and country
`GET`	`/api/v1/netflow/lookup/{query}`	Find an IP: who holds it and where it is
`GET`	`/api/v1/netflow/pairs`	Who talks to whom: client and destination reached
`GET`	`/api/v1/netflow/status`	State of the NetFlow collector
`GET`	`/api/v1/netflow/subscribers/{subscriber_id}/series`	Volume over time for a subscriber
`GET`	`/api/v1/netflow/top`	Who consumes, and how much
`GET`	`/api/v1/netflow/vantages`	Traffic seen at each vantage point

Traffic restrictions

Method	Path	What it does
`GET`	`/api/v1/traffic-rules`	The saved restrictions
`POST`	`/api/v1/traffic-rules`	Save a restriction (installed on the routers at once when enabled)
`POST`	`/api/v1/traffic-rules/apply`	Apply the restrictions on the routers
`PATCH`	`/api/v1/traffic-rules/{rule_id}`	Edit a restriction
`DELETE`	`/api/v1/traffic-rules/{rule_id}`	Delete a restriction and lift it
`GET`	`/api/v1/traffic-rules/{rule_id}/preview`	What this rule targets today

Antennas

Method	Path	What it does
`GET`	`/api/v1/pops/antennas`	Ubiquiti antennas polled live

Method	Path	What it does
POST	/api/v1/pops/antennas	Save an antenna
POST	/api/v1/pops/antennas/test	Test an antenna without saving it
PATCH	/api/v1/pops/antennas/{antenna_id}	Edit an antenna
DELETE	/api/v1/pops/antennas/{antenna_id}	Remove an antenna
POST	/api/v1/pops/antennas/{antenna_id}/probe	Test a saved antenna
GET	/api/v1/radios	Radio health of every AP and its CPEs

Operations

Method	Path	What it does
POST	/api/v1/jobs/{job_name}/run	Replay a collection cycle immediately
GET	/api/v1/latency	Latency by segment: access, upstream, internet
GET	/api/v1/latency/clients	Latency by client: the experience each one lives
GET	/api/v1/rtt	State of the latency probe (RTT)
PUT	/api/v1/rtt	Turn the latency probe (RTT) on or off
GET	/api/v1/rtt/diagnose	Run the latency probe by hand, with and without source
POST	/api/v1/rtt/reset-history	Clear recent latency measurements (and scores derived)
GET	/api/v1/status	State of the controller and its cycles
GET	/api/v1/status/counters	Global counters of the reference data
GET	/api/v1/status/runs	History of the collection cycles

Settings

Method	Path	What it does
GET	/api/v1/settings	Operational settings in force
GET	/api/v1/settings/history	Who changed which setting, and when
PUT	/api/v1/settings/{name}	Set a setting (stored in the database)
DELETE	/api/v1/settings/{name}	Reset a setting to its default

Health

Method	Path	What it does
`GET`	`/health`	Liveness (the process answers)
`GET`	`/health/ready`	Readiness (database + data freshness)

Accounts

Method	Path	What it does
`GET`	`/api/v1/auth/events`	Login and account journal (edit rights)
`POST`	`/api/v1/auth/login`	Log in with email and password
`POST`	`/api/v1/auth/logout`	Close the current session
`POST`	`/api/v1/auth/password`	Change my own password
`GET`	`/api/v1/auth/sessions`	My open sessions (every browser logged in)
`POST`	`/api/v1/auth/sessions/close-others`	Log out every other browser
`DELETE`	`/api/v1/auth/sessions/{session_id}`	Close one of my sessions
`POST`	`/api/v1/auth/setup`	Create the FIRST account (edit rights)
`GET`	`/api/v1/auth/status`	Who am I, and is a first account needed?
`GET`	`/api/v1/users`	List the accounts (edit rights)
`POST`	`/api/v1/users`	Create an account (edit rights)
`PATCH`	`/api/v1/users/{user_id}`	Change role, password or state (edit rights)
`DELETE`	`/api/v1/users/{user_id}`	Delete an account (edit rights)

API keys

Method	Path	What it does
`GET`	`/api/v1/api-keys`	Registered API keys (never the secrets)
`POST`	`/api/v1/api-keys`	Create a key (the secret is returned here only)
`PATCH`	`/api/v1/api-keys/{key_id}`	Enable or disable a key
`DELETE`	`/api/v1/api-keys/{key_id}`	Revoke a key for good

Bandwidth schedules

Method	Path	What it does
`GET`	`/api/v1/bandwidth-schedules`	Time-of-day bandwidth schedules
`POST`	`/api/v1/bandwidth-schedules`	Create a bandwidth schedule (applied right away)
`PATCH`	`/api/v1/bandwidth-schedules/{schedule_id}`	Edit a bandwidth schedule
`DELETE`	`/api/v1/bandwidth-schedules/{schedule_id}`	Delete a bandwidth schedule
`GET`	`/api/v1/bandwidth-schedules/{schedule_id}/preview`	Which subscribers this schedule governs, and their rate right now

Clients

Method	Path	What it does
`GET`	`/api/v1/client-profiles`	Every client's hand-entered details
`POST`	`/api/v1/clients`	Declare a PPPoE client by hand, before its first session
`GET`	`/api/v1/clients/{login}/health`	A client's health: experience, radio, site, limit, usage
`GET`	`/api/v1/clients/{login}/profile`	A client: discovered facts and its details
`PUT`	`/api/v1/clients/{login}/profile`	Edit a client's details (name, CPE MAC, contact)

Map

Method	Path	What it does
`GET`	`/api/v1/client-locations`	Where the clients are, for the map
`PUT`	`/api/v1/client-locations/{login}`	Place a client on the map
`DELETE`	`/api/v1/client-locations/{login}`	Remove a client from the map
`GET`	`/api/v1/site-locations`	Where the sites (PoPs) are, for the map
`PUT`	`/api/v1/site-locations/{name}`	Place a site on the map
`DELETE`	`/api/v1/site-locations/{name}`	Remove a site from the map

Modes

Method	Path	What it does
`GET`	`/api/v1/activity`	Changes made on clients: who, when, what

Method	Path	What it does
`GET`	`/api/v1/bandwidth-modes`	Bandwidth modes a client can be put in
`POST`	`/api/v1/bandwidth-modes`	Create a bandwidth mode
`PATCH`	`/api/v1/bandwidth-modes/{mode_id}`	Edit a bandwidth mode
`DELETE`	`/api/v1/bandwidth-modes/{mode_id}`	Delete a bandwidth mode
`POST`	`/api/v1/clients/actions`	Change mode or boost of one or several clients
`PUT`	`/api/v1/clients/{login}/mode`	Put one client in a bandwidth mode

Notifications

Method	Path	What it does
`GET`	`/api/v1/notifications/history`	Problems detected, and whether they were emailed
`GET`	`/api/v1/notifications/smtp`	The SMTP account (never its password)
`PUT`	`/api/v1/notifications/smtp`	Save the SMTP account for alert emails
`DELETE`	`/api/v1/notifications/smtp`	Remove the SMTP account (no more emails)
`POST`	`/api/v1/notifications/test`	Send a test email now

Public access

Method	Path	What it does
`GET`	`/api/v1/public-access`	Domain name and HTTPS of the interface
`PUT`	`/api/v1/public-access`	Serve the interface under a domain name, in HTTPS
`DELETE`	`/api/v1/public-access`	Stop serving the interface under a domain name
`POST`	`/api/v1/public-access/check`	Check the DNS and the HTTPS of the domain

Topology

Method	Path	What it does
`PUT`	`/api/v1/topology/client-links`	Say which client a box is (or that it is none)
`DELETE`	`/api/v1/topology/client-links/{node_key}`	Back to the automatic link (by name)
`PUT`	`/api/v1/topology/client-places`	Hang clients under a box of the tree

Method	Path	What it does
`DELETE`	`/api/v1/topology/client-places/{login}`	A client goes back to its automatic place

Ubiquiti

Method	Path	What it does
`GET`	`/api/v1/uisp`	The Ubiquiti UISP controller connected, if any
`PUT`	`/api/v1/uisp`	Connect the Ubiquiti UISP controller
`DELETE`	`/api/v1/uisp`	Disconnect the Ubiquiti UISP controller
`GET`	`/api/v1/uisp/devices`	Antennas known to UISP, with their radio capacity
`GET`	`/api/v1/uisp/devices/{device_id}/history`	Throughput of one UISP device over time
`PUT`	`/api/v1/uisp/links/{device_id}`	Attach a UISP antenna to a site
`DELETE`	`/api/v1/uisp/links/{device_id}`	Detach a UISP antenna from its site
`POST`	`/api/v1/uisp/test`	Test the connection to UISP (stored or given)
`GET`	`/api/v1/uisp/traffic`	Every UISP device: throughput, capacity, signal, its place