

UZ801 LTE MiFi dongle — ADB auto-enable (host side)

Cheap ~\$10 Qualcomm LTE MiFi dongles (05c6:90b6, "Android"/UZ801, msm8916) boot in **RNDIS mode** with a web UI at `http://192.168.100.1`. GETting `http://192.168.100.1/usbdebug.html` flips them into **ADB mode** (and they come up already-root). Ref:

<https://nickvsnetworking.com/adventures-with-a-10-lte-mifi-dongle/>

The enable step has to run on the **USB host** (the Proxmox hypervisor), because only the host has the RNDIS link to `192.168.100.1` — the OmniWeb phone-agent runs in the unprivileged LXC (CT 102) and reaches the dongle only as a USB device once it is in ADB mode. So this automation is host-side udev + systemd, not agent code.

Files

- `mifi-adb-enable.sh` → `/usr/local/sbin/mifi-adb-enable.sh` (0755)
- `mifi-adb-enable.service` → `/etc/systemd/system/mifi-adb-enable.service`
- `99-mifi-adb-autoenable.rules` → `/etc/udev/rules.d/99-mifi-adb-autoenable.rules`

```
sudo install -m0755 mifi-adb-enable.sh /usr/local/sbin/mifi-adb-enable.sh
sudo install -m0644 mifi-adb-enable.service /etc/systemd/system/mifi-adb-enable.service
sudo install -m0644 99-mifi-adb-autoenable.rules /etc/udev/rules.d/99-mifi-adb-autoenable.rules
sudo systemctl daemon-reload && sudo udevadm control --reload
```

Flow: dongle plugged (RNDIS) → udev fires → `mifi-adb-enable.service` → script brings up the RNDIS iface, curls `usbdebug.html` → dongle reboots into ADB mode → re-enumerates with the ADB interface → the CT-102 udev rule (below) hands the node to the container → the agent's `adb` sees it. On the adb-mode re-enumeration the script runs again, detects the ADB interface (`ff/42/01`) and exits — no loop.

Companion CT-102 USB-passthrough udev change (REQUIRED)

The dongle's Qualcomm VID `05c6` must be handed to the container, and the group **must be plugdev (host gid 100046)**, not container-root, or the agent's adb (user `omniweb-agent`, a plugdev member) cannot open the node. The same applies to the Android phone rules: they were `GROUP="100000"` (container root) which the agent cannot access — devices only worked while adb held the fd from enumeration time and dropped on every re-enumeration. Fixed to `GROUP="100046"`.

In `/etc/udev/rules.d/99-ct102-android.rules` all phone rules use `OWNER="100000", GROUP="100046", MODE="0660"`, plus the dongle VID:

```
SUBSYSTEM=="tty", SUBSYSTEMS=="usb", ATTRS{idVendor}=="05c6",  
OWNER="100000", GROUP="100046", MODE="0660"  
SUBSYSTEM=="usb", ATTR{idVendor}=="05c6", OWNER="100000",  
GROUP="100046", MODE="0660"
```

These host changes are applied live in the lab but should be codified in the OmniCore ansible (monitoring/host role) so they survive a host rebuild.

AMF: Access and Mobility Management Function

The **AMF** is the control-plane anchor of the 5G core. It terminates the **N1** signalling to the UE and the **N2** (NGAP) association to each gNodeB. It handles registration and connection management: it attaches UEs, keeps them reachable, and pages them when they are idle. It owns the **UE contexts** (one per registered subscriber, keyed by SUPI) and the **gNB associations** (one per connected radio, keyed by SCTP association). It selects the SMF for each PDU session and relays session requests between the UE and the SMF. When a subscriber is on the network, the AMF is the NF that knows where they are and how to reach them.

← 5G Core

In the dashboard

The screenshot displays the AMF Dashboard in the OmniWeb interface. The dashboard is titled "AMF Dashboard" with a sub-header "Access and Mobility Management Function - OAM API at https://10.139.35.248:8443". The dashboard is divided into several sections:

- Overview** (selected): This section contains four health cards: API Health (OK), NRF Registration (registered), NF Status (OK), and License (OK).
- Statistics**: This section displays five key metrics: Registered Ues (3,847), Connected Ues (2,913), Idle Ues (934), Connected GNBs (3), and Registration Success Rate (99.4%).
- NF-Specific Status**: This section shows the status of the AMF as "ok".
- NRF Registration**: This section shows the registration status as "registered".

The left sidebar contains a navigation menu with the following items: Host List, Config Archive, Network Topology, Elements (expanded), Signalling (HSS, DRA, OCS, TWAG), EPC (MME, SGW-C, PGW-C, UPF, SCEF), CS Core (MSC, STP, HLR, CAMEL-GW), and Settings.

The AMF dashboard opens with the four standard health cards: **API Health**, **NRF Registration** (whether the AMF is currently registered with the NRF), **NF Status**, and **License**. Each polls on an interval. Below them, it shows the data the AMF owns:

- **UE List**: every UE context the AMF holds, keyed by SUPI, with its registration and connection state. This is the "who is attached right now" view. Select a UE to open its detailed context. The AMF can also list a UE's active PDU sessions by SUPI.
- **gNodeBs**: the connected radios, one row per SCTP association. The table decodes the raw NGAP values into readable **gNB-ID**, **TAC** (tracking area), **PLMN**, and **SST** (slice/service type). An operator can therefore confirm which cells are up and which tracking areas and slices they serve without decoding identifiers by hand. Select a row to open the detailed gNB by its SCTP association ID.
- **Statistics**: the AMF's live counters (registered UEs, connected gNBs, and signalling volumes), for a quick read on load and activity.

- **Maintenance Mode:** whether the AMF is in maintenance. Use it to drain or hold a node during planned work.
- **Configuration:** the runtime configuration the service is currently using.

Actions

The dashboard exposes actions to recover from stuck state. The dashboard confirms each one before it runs, since they clear live state:

- **Clear All UE Contexts** (bulk): removes every UE context from the AMF. Use it as a last resort to recover a node that holds stale or corrupt contexts. Every affected UE must re-register.
- **Clear Cache** (bulk): clears the AMF's NF-discovery cache and UE caches. Use it when the AMF acts on stale peer or lookup data, for example after an NF has moved or re-registered.
- **Deregister** (per-UE): deregisters a single UE and cleans up its state. Use it to drop one subscriber's context that will not clear on its own, rather than reset the whole node.

Beyond the dashboard buttons, the AMF's OAM API exposes finer-grained recovery controls an operator can reach through the backend proxy:

- **Force-disconnect a gNB:** tears down a single gNB by its SCTP association ID. Use it to clear a wedged or misbehaving radio association without touching the others.
- **Force-release a PDU session:** releases one PDU session by its `supi:psi` identifier. Use it to drop a session that is stuck on the AMF's side while the UE stays registered.
- **Deregister or page a UE:** forces a UE to deregister, or pages an idle UE to bring it back to connected. Use paging to test reachability. Use forced deregistration to evict a UE cleanly.

Operations

- **Runtime diagnostics** (`/api/oam/diagnostics`): shows low-level diagnostics for the running service.

- **Resource usage** (`/api/oam/resources`): reports the runtime's memory, process, and scheduler usage for a health check on the node.
- **Peers** (`/api/oam/peers`): lists the configured and discovered NF peer URIs and the current SCTP associations, so an operator can see who the AMF talks to.
- **NRF discovery cache** (`/api/oam/nrf_discovery`): shows the contents of the discovery cache: which NF instances the AMF has learned from the NRF.
- **Set log level** (`/api/oam/log_level`): changes the log level at runtime (one of `emergency`, `alert`, `critical`, `error`, `warning`, `warn`, `notice`, `info`, `debug`) to turn up detail while you troubleshoot.
- **Force NRF re-registration** (`/api/oam/nrf/reregister`): re-registers the AMF with the NRF. Use it to recover a dropped or stale registration.

Related

- **SMF: Session Management Function**: the AMF selects the SMF and relays PDU session requests to it.
- **UDM: Unified Data Management**: the subscriber data the AMF uses during registration.
- **AUSF: Authentication Server Function**: authenticates the UE during registration on the AMF's behalf.
- **NSSF: Network Slice Selection Function**: resolves slice selection for the AMF.
- **← 5G Core**: the overview of all NF dashboards.

AUSF: Authentication Server Function

The **AUSF** authenticates subscribers on behalf of the AMF. When a UE attaches, the AMF asks the AUSF to run 5G-AKA (or EAP-AKA') for that subscriber. The AUSF works with the **UDM** to obtain the key material, derives the anchor key it returns to the AMF, and confirms the authentication result. It holds an active authentication session for each in-flight or established authentication so an operator can track its progress and outcome and, if needed, clear it.

← 5G Core

In the dashboard

The AUSF dashboard opens with the standard health cards across the top (**API Health**, **NRF Registration** (is the AUSF registered with the NRF?), **NF Status**,

and **License**), each polling on an interval. Beneath them are the AUSF's data sections, read live from its OAM API through the OmniWeb proxy:

- **Statistics**: the AUSF's own counters ([AUSF Statistics](#)), which give a running view of authentication activity.
- **Active AUSF authentication sessions**: the list of authentication sessions the AUSF currently holds ([List Active AUSF Authentication Sessions](#)). Each row is keyed by its session identifier, so an operator can see which authentications are in flight or established at a glance.

Actions

Both actions clear authentication state and ask for confirmation first, since they are destructive:

- **Clear a specific authentication session** (per-row): removes a single authentication session by its identifier. Use this to drop one stuck or stale session without disturbing every other subscriber's authentication.
- **Clear All Auth Sessions**: removes every authentication session the AUSF holds. Use this to recover the AUSF from a widespread stuck state. Each affected subscriber re-authenticates on its next attempt.

Operations

Standard OAM controls available on the AUSF:

- **Diagnostics** ([/api/oam/diagnostics](#)): runtime diagnostics for the service.
- **Resources** ([/api/oam/resources](#)): the service's resource usage.
- **Force NRF re-registration** ([/api/oam/nrf/reregister](#)): makes the AUSF re-register with the NRF, to recover if its registration has gone stale.

Related

- **AMF: Access & Mobility Management**: requests authentication from the AUSF for each attaching UE.

- **UDM: Unified Data Management**: supplies the key material the AUSF uses to authenticate a subscriber.
- **← 5G Core**: the 5GC dashboard overview and the other NFs.

BSF: Binding Support Function

The **BSF** registers and resolves the binding between a UE's PDU session and the PCF that serves it. When a session is established, the BSF records which PCF is handling policy for that session, keyed by the session's identifiers: the UE IP address, the DNN, and the S-NSSAI. Other NFs and application functions then query the BSF to discover the correct PCF for a given UE session, so that policy requests reach the PCF already holding that session's context rather than an arbitrary one. The BSF holds one binding per active PDU session and exposes them, along with summary statistics, for monitoring.

[← 5G Core](#)

In the dashboard

The BSF dashboard opens with the standard health cards across the top (**API Health**, **NRF Registration** (is the BSF registered with the NRF?), **NF Status**, and **License**), each polling on an interval. Beneath them are the BSF's data sections, read live from its OAM API through the OmniWeb proxy:

- **PCF Bindings**: the list of active bindings the BSF currently holds ([List all PCF bindings with binding ID, UE IP, DNN, S-NSSAI, PCF FQDN/IP](#)). Each row is keyed by its binding ID and shows the UE IP address, DNN, and S-NSSAI that identify the session, together with the FQDN or IP of the PCF serving it. This answers "which PCF is handling policy for this UE session?" at a glance. Select a binding to open its full detail ([Get detailed PCF binding by binding ID](#)).
- **Statistics**: the BSF's own counters ([BSF statistics: total bindings, bindings by DNN](#)), which give a running view of how many bindings are active in total and how they break down per DNN.

Actions

The BSF dashboard is read and monitoring oriented. It surfaces bindings and statistics for inspection and does not expose destructive per-row or NF-level clear actions.

Operations

Standard OAM controls available on the BSF:

- **Force NRF re-registration** (</api/oam/nrf/reregister>): makes the BSF re-register with the NRF, to recover if its registration has gone stale.

Related

- **PCF: Policy Control**: the NF each binding points to. The BSF is how consumers find the right PCF for a UE session.
- **SMF: Session Management** and application functions: Nbsf consumers that register and look up bindings as sessions come and go.

- ← 5G Core: the 5GC dashboard overview and the other NFs.

CHF: Charging Function

The **CHF** (Charging Function) provides converged online and offline charging for the 5G Core. The SMF consumes it over the N40 interface: as PDU sessions carry traffic, the CHF manages usage quotas, rates that usage through the rating/charging engine, and generates the CDRs (Charge Data Records) that feed billing and reconciliation. Its dashboard surfaces the live charging sessions, the health of the link to the rating engine, and the completed CDR history. An operator can then answer "is a subscriber being charged correctly, and is the rating engine reachable?" without leaving OmniWeb.

← 5G Core

In the dashboard

Across the top sit the standard **health cards** (**API Health**, **NRF Registration** (is the CHF registered with the NRF?), **NF Status**, and **License**), each polling on an interval. Below them the CHF-specific data sections auto-refresh:

- **Active charging sessions:** the live sessions the CHF is metering, each showing its charging reference (`chargingDataRef`), the subscriber (SUPI), the DNN, accumulated usage, and start/update timestamps. This is the working set of "who is being charged right now".
- **Statistics:** a summary view of total sessions, connectivity to the rating/charging engine, and total charged volume.
- **Rating-engine connectivity health:** a dedicated health check that confirms the CHF can reach the rating/charging engine that performs the actual rating. If this is unhealthy, the CHF cannot rate sessions and charging is at risk.
- **Completed CDR search:** a searchable view over the finalised CDRs, filterable by SUPI, DNN, and date range, to confirm what the CHF recorded after a session closes.
- **Configuration:** the CHF's runtime configuration as the service currently sees it.

Select a session to open its full detail by charging data ref, which shows the complete usage and metering state for that one session.

Actions

- **Abort a charging session:** terminates an in-progress charging session by its `chargingDataRef`. Use this to clear a stuck or orphaned session that the SMF no longer maintains, so it stops consuming quota and metering state.
- **Search CDRs:** queries the completed CDR history by SUPI, DNN, and date range. Use this after a session ends to confirm what usage the CHF rated and recorded. It supports billing enquiries, dispute investigation, or a spot-check that the CHF writes charging through correctly.

Operations

- **Set log level:** changes the CHF's log verbosity at runtime (emergency, alert, critical, error, warning, warn, notice, info, debug) to gather more detail while you investigate a charging issue, then return to normal.

- **Force NRF re-registration:** makes the CHF re-register itself with the NRF. Use it to recover if the CHF has fallen out of the registry and the SMF can no longer discover it.

Related

- **SMF:** the N40 consumer that drives charging as PDU sessions carry traffic.
- **PCF:** the policy side of policy-and-charging control that governs what the CHF meters against.
- The rating/charging engine (OCS): the downstream system the CHF rates through. The rating-engine connectivity health card reflects the link to it.
- [← 5G Core](#): the overview of all NF dashboards.

NRF: Network Repository Function

The whole 5G core is built around the **NRF** registry. On start-up, every other NF registers its **NF profile** with the NRF: its type, addresses, PLMNs, the **services** it offers, and its capacity and priority. It then discovers its peers by asking the NRF who is available. The NRF tracks each registered instance's **status**. It holds **status-change subscriptions**, so it can tell an NF when a peer it depends on comes or goes. It watches **heartbeats** to expire instances that stop checking in. When any NF needs to find another, the NRF is the source of truth for what is up and how to reach it.

[← 5G Core](#)

In the dashboard

The NRF has its own dedicated dashboard, richer than the standard NF page because it owns the registry the rest of the core discovers through. It opens with the standard health cards: **API Health**, **NRF Registration** (whether the NRF itself is registered), **NF Status**, and **License**. Each polls on an interval.

Above the data it shows four **metric cards** that summarise the registry: **NF types**, **total instances, registered** (with a percent-of-total bar), and **services** (with the subscription count). Below them are the NRF's core views:

- **Registered NF instances:** every registered instance, grouped by **NF type** (AMF, SMF, UDM, PCF, and so on). The dashboard colour-codes each instance card by type. The card shows its registration status, heartbeat timer, service count, and PLMN. Select an instance to open its full **NF profile**: the detailed record the NRF holds, with an **Open Dashboard** link to that NF's own page and a **Deregister** button. This is the "which NFs are up and are they all registered?" view.
- **Registered services:** a flat table of every service each instance advertises, with its versions and status. An operator can therefore see what capabilities the core currently offers across all NFs.
- **Subscriptions:** the NF status-change subscriptions currently held. Each is an NF's standing request to be notified when a peer's status changes.
- **Statistics:** the NRF's live counters: total NFs by type, subscriptions, and discovery requests, for a quick read on registry size and lookup activity.

Actions

The dashboard exposes actions to manage the registry. The dashboard confirms each one before it runs, since it changes live registration state:

- **Deregister an NF instance** (per-row): removes one instance's registration from the NRF. Use it to clear a stale or wedged registration for a single NF. That NF must then re-register before other NFs can discover it again.
- **Deregister All NFs** (bulk): removes every registration from the NRF at once. Use it as a last resort to reset a registry that holds stale entries. Every NF must then re-register.
- **Remove a subscription:** deletes one status-change subscription. Use it to clear a stale or orphaned subscription left behind by an NF that no longer

needs it.

- **Update an NF profile:** patches a registered instance's profile (for example its capacity, priority, or status). Operators can therefore adjust how the NRF steers discovery towards or away from an instance without a deregistration.

Operations

The NRF's OAM API exposes further controls an operator can reach through the backend proxy:

- **Runtime diagnostics** (`/api/oam/diagnostics`): shows low-level diagnostics for the running service.
- **Resource usage** (`/api/oam/resources`): reports the runtime's memory, process, and scheduler usage for a health check on the node.
- **Manual heartbeat check** (`/api/oam/heartbeat`): triggers an on-demand heartbeat check for an NF instance. It confirms whether the NF is still alive rather than wait for its timer.
- **NF uptime** (`/api/oam/nf_uptime/{id}`): shows an instance's registration time and how long it has been registered, to confirm how long an NF has been up.
- **Set log level** (`/api/oam/log_level`): changes the log level at runtime (one of `emergency`, `alert`, `critical`, `error`, `warning`, `warn`, `notice`, `info`, `debug`) to turn up detail while you troubleshoot.
- **Force NRF re-registration** (`/api/oam/nrf/reregister`): re-registers the NRF. Use it to recover a dropped or stale self-registration.

Related

- **AMF: Access and Mobility Management Function:** registers with the NRF and discovers its peers through it, like every NF below.
- **SMF: Session Management Function:** discovered through the NRF for session management.
- **UDM: Unified Data Management:** one of the subscriber-data NFs the NRF holds registrations for.

- **PCF: Policy Control Function**: registered with the NRF for policy discovery.
- **SCP: Service Communication Proxy**: routes signalling between NFs using the registrations and discovery data the NRF holds.
- **← 5G Core**: the overview of all NF dashboards.

NSSF: Network Slice Selection Function

The **NSSF** selects the network slice instance(s) and the allowed NSSAI for a UE. When the AMF handles a registration or a slice request, it asks the NSSF to resolve the UE's requested S-NSSAIs against the slices configured for that PLMN. The NSSF maps each to its network slice instance (NSI) and to the AMF set that serves it, and returns the allowed NSSAI the UE may use. It holds the configured slice definitions and tracks per-AMF NSSAI availability so an operator can reason about and monitor slice selection.

[← 5G Core](#)

In the dashboard

The NSSF dashboard opens with the standard health cards across the top (**API Health**, **NRF Registration** (is the NSSF registered with the NRF?), **NF Status**,

and **License**), each polling on an interval. Beneath them are the NSSF's data sections, read live from its OAM API through the OmniWeb proxy:

- **Slices**: every configured S-NSSAI, listed with its NSI mappings, the AMF sets that serve it, and the allowed NSSAI per PLMN (`List all configured S-NSSAIs with NSI mappings, AMF sets, and allowed NSSAI per PLMN`). This is the authoritative view of how requested slices resolve to slice instances and AMF sets for each PLMN.
- **NSSAI Availability**: the per-AMF NSSAI availability state (`Per-AMF NSSAI availability state`), which shows which slices each AMF has reported it can currently serve.
- **Statistics**: the NSSF's own counters (`NSSF statistics: total slices, NS selection request counts`), which give a running view of the number of configured slices and of slice-selection request volume.

A separate **slice configuration** view (`Query Slice Configuration`) surfaces the NSSF's runtime slice configuration for reference.

Actions

The NSSF dashboard is read and monitoring oriented. It surfaces slices, NSSAI availability, and statistics without per-item control actions.

Operations

The OAM controls the NSSF exposes are:

- **View runtime configuration** (`/api/oam/config`): the NSSF's live runtime configuration.
- **Force NRF re-registration** (`/api/oam/nrf/reregister`): makes the NSSF re-register with the NRF, to recover if its registration has gone stale.

Related

- **AMF: Access & Mobility Management**: queries the NSSF to resolve requested S-NSSAIs into the allowed NSSAI and serving AMF set for a UE.
- **NRF: Network Repository Function**: where the NSSF registers so other NFs can discover it.
- **← 5G Core**: the 5GC dashboard overview and the other NFs.

PCF: Policy Control Function

The **PCF** makes the policy decisions the rest of the 5G Core enforces. It answers the AMF over N15 for **access & mobility (AM) policy**: the mobility restrictions and RFSP a UE gets on registration. It answers the SMF over N7 for **session-management (SM) policy**: the PCC rules that set each session's QoS, gating, charging control and usage monitoring. The PCF binds a UE's SM policy associations back to itself through the BSF so any NF can find the right PCF for a session. The PCF dashboard is the place to answer "what policies are live, for which UEs and sessions, and are they what I expect?" without leaving OmniWeb. Every OAM response is wrapped `{status, response}`, and the dashboard unwraps it for you.

[← 5G Core](#)

In the dashboard

Across the top sit the four standard health cards (**API Health**, **NRF Registration** (is the PCF registered with the NRF?), **NF Status**, and **License**), each polling on an interval. Use the polling indicator in the header to pause refresh or read the last-update time. Below the cards is the **Actions** bar (see below), then the PCF's data sections, each auto-refreshing.

Statistics: the PCF's live counters (`GET /api/statistics`), the policy-association totals and current counts. This is the quickest read on whether the PCF actively serves AM and SM policy.

AM Policies: every active AM policy association (`GET /api/am_policy`), one row per UE the AMF has an AM policy for. This is the PCF's view of the mobility policy currently applied to each registered UE.

SM Policies: every active SM policy association (`GET /api/sm_policy`), one row per PDU session the SMF holds a policy for. Select a row to open its detail (`GET /api/sm_policy/{id}`), which returns the full PCC rules on that session: the QoS, gating, charging and usage-monitoring decisions the SMF enforces.

Policy Context: the active policy contexts view (`GET /api/policy_context`), the contexts the PCF holds across its AM and SM associations. It is useful for reconciling what the PCF believes is live against the AMF and SMF.

Configuration: the PCF's runtime configuration (`GET /api/oam/config`), the settings the service currently runs with.

Actions

Each per-row action asks for confirmation first, since removing a live association is disruptive to the UE or session it affects.

Remove an AM policy: on any AM Policies row, **Remove** deletes that AM policy association (`DELETE /api/am_policy/{id}`). Use it to clear a stuck or stale mobility policy for one UE.

Remove an SM policy: on any SM Policies row, **Remove** deletes that SM policy association (`DELETE /api/sm_policy/{id}`). Use it to drop a session's policy that will not clear on its own.

Update an SM policy: you can also edit an SM policy association. Send `PATCH` or `PUT /api/oam/sm_policy/{id}` to update the PCC rules on a live SM policy in place, so you can adjust a session's QoS or charging decisions without tearing the association down. Use it to re-drive policy onto an existing session rather than to remove and re-create it.

Operations

The Operations section carries the controls common to every NF:

- **Diagnostics** (`GET /api/oam/diagnostics`): runtime diagnostics for the service.
- **Resource usage** (`GET /api/oam/resources`): memory, processes and schedulers of the running service.
- **Peers** (`GET /api/oam/peers`): the configured NF peer URIs the PCF talks to and their reachability status.
- **NRF discovery cache** (`GET /api/oam/nrf_discovery`): the NRF discovery cache, or the configured NF URIs the PCF resolves peers through.
- **Log level** (`POST /api/oam/log_level`): change the log level at runtime. Choose one of emergency, alert, critical, error, warning, warn, notice, info, or debug.
- **NRF re-registration** (`POST /api/oam/nrf/reregister`): force the PCF to re-register with the NRF, for example after an NRF restart left it stale.

Related

- **SMF**: requests SM policy from the PCF over N7 and enforces the PCC rules the PCF returns.
- **AMF**: requests AM policy from the PCF over N15.
- **BSF**: the binding function the PCF registers SM policy associations with so any NF can find the serving PCF.
- **CHF**: the charging function. The PCF's charging-control decisions drive what the SMF reports over N40.
- [← 5G Core overview](#)

SCP: Service Communication Proxy

The **SCP** sits between the network functions of the 5G Core and carries their service-based traffic for them. Rather than every consumer NF connecting directly to every producer, requests can be routed through the SCP, which forwards each message to a healthy instance of the target NF type, load-balancing across instances and re-trying around failures. It can perform discovery on a consumer's behalf (delegated discovery) and caches NF reachability so that, once it knows a target type, it steers later requests without a fresh lookup. The result is indirect communication: producers and consumers do not need to know or hold connections to one another, and the SCP handles routing, balancing, and cache freshness in between.

← 5G Core

In the dashboard

The SCP dashboard follows the same layout as every 5GC NF page. Four **health cards** run across the top (**API Health**, **NRF Registration** (is the SCP itself registered with the NRF?), **NF Status**, and **License**), each polling on an interval.

Below the cards, the SCP surfaces its own operational data:

- **NF Health**: the load-balancer's view of the NFs it proxies to, with per-instance health and failure counts, so an operator can see which producer instances the SCP considers reachable and which are failing.
- **Discovery Cache**: the contents of the SCP's discovery cache, which NF types are cached, their TTLs, and the cache hit/miss rates. This is what lets the SCP steer repeat requests without a re-discovery.
- **Statistics**: SCP throughput, total proxied requests, a breakdown of requests by target NF type, and the overall cache hit rate.
- **Configuration**: the SCP's current runtime configuration.

Each section auto-refreshes, and NF-type values cross-link to that NF's own dashboard.

Actions

The SCP dashboard does not expose destructive NF-level or per-row action buttons. It proxies traffic rather than owns session or subscriber state that an operator would clear. Its cache, however, is manageable: the OAM API lets an operator view the discovery cache and evict a single stale entry so the SCP re-discovers that target on the next request (see **Operations**). To recover from a producer that has moved or been replaced but is still cached as reachable, clear its entry.

Operations

The SCP's OAM API exposes the following controls:

- **View / manage discovery cache**: read the current discovery cache, and delete an individual cache entry by id to force re-discovery of that target.

- **View runtime configuration:** read the SCP's live configuration.
- **Set log level:** change the log verbosity at runtime. Choose one of `emergency`, `alert`, `critical`, `error`, `warning`, `warn`, `notice`, `info`, or `debug`.
- **Force NRF re-registration:** make the SCP re-register itself with the NRF. This is useful after the NRF or the SCP has restarted and the registration has gone stale.

Related

- **NRF:** the registry the SCP discovers through and re-registers with.
- **AMF, SMF, UDM, AUSF, PCF, CHF, NSSF, BSF:** the NFs whose service-based traffic the SCP can proxy and whose reachability it caches.
- **← 5G Core:** the 5GC section overview and the shared dashboard layout.

SMF: Session Management Function

The **SMF** owns the PDU session. It establishes, modifies and releases each UE's data session, allocates the UE its IP address from a local pool, and drives the user plane over N4/PFCP. It programs the UPF with the packet-detection, forwarding and QoS rules for every session. It also talks to the PCF (N7) for session policy and the CHF (N40) for charging. Use the SMF dashboard to answer "are sessions coming up, is the SMF still associated with its UPF, and is it running out of addresses?" without leaving OmniWeb. The SMF wraps every OAM response `{status, response}`. The dashboard unwraps it for you.

[← 5G Core](#)

In the dashboard

Across the top sit the four standard health cards: **API Health**, **NRF Registration** (is the SMF registered with the NRF?), **NF Status**, and **License**. Each polls on an interval. Use the polling indicator in the header to pause refresh or read the last-update time. Below the cards sits the **Actions** bar (see below). Then come the SMF's data sections, each auto-refreshing.

Statistics: the SMF's live counters (`GET /api/statistics`): session establishment / modification / release totals and current session counts. This is the quickest read on whether the SMF actually sets up sessions.

PDU Sessions: every active PDU session (`GET /api/pdu_session`), returned as `{ pdu_sessions: [...], total }`. Each row carries both a `sm_context_ref` (a UUID that uniquely identifies the SM context) and a `pdu_session_id` (the 3GPP PSI, a small integer that is **not** unique across UEs, since each UE numbers its own sessions from a low base). Every per-session operation (detail lookup and release) therefore keys on `sm_context_ref`. The dashboard's per-row **Release** action passes that UUID rather than the PSI. Alongside the identifiers, a row shows the session's DNN, allocated UE IP and state.

IP Pool: the address pool the SMF allocates UE IPs from (`GET /api/ip_pool/detail`): the pool summary (subnet, total, allocated and available counts) plus the list of currently allocated IPs. Here you see whether the SMF is close to exhausting its address space.

UPF Association: the N4/PFCP link state to the user plane (`GET /api/upf`): whether the SMF is currently associated with its UPF. If sessions fail to establish, check this first.

PFCP Peers: the PFCP peer (UPF) association status (`GET /api/oam/pfcp_peers`): the SMF's view of each UPF it associates with over N4, including association state.

Actions

Each action asks for confirmation first. Several are destructive to live sessions.

Release All PDU Sessions: tears down every PDU session and its PFCP state on the UPF. Use it to clear the SMF back to a clean slate, for example after a

UPF swap or when you drain a node for maintenance.

Clear Cache: clears the SMF's session caches without a session release. Use it when the dashboard or the SMF's internal view looks stale or inconsistent with reality.

PFCP Reassociate: re-establishes the PFCP association with the UPF (SMF-initiated). Use it when the UPF association has dropped and new sessions cannot establish.

Re-associate UPF: forces a PFCP re-association with the **configured** UPF. The UPF address is config-driven. There is no add-UPF or inject-UPF action here. This re-drives the association to the UPF already in the SMF's configuration. In practice it serves the same recovery purpose as PFCP Reassociate.

Release by DNN: type a DNN into the editable **DNN** field (for example `internet`). This releases **all** sessions on that DNN. Use it to drain or reset a single data network, for example to force UEs off one APN/DNN, without touching sessions on other DNNs.

Per-row Release: on any PDU Sessions row, **Release** tears down that one session and its PFCP context on the UPF. It releases by `sm_context_ref`, so it always targets exactly the session in that row even though several UEs may share the same `pdu_session_id`. Use it to drop a single stuck session.

Operations

The Operations section carries the controls common to every NF:

- **Diagnostics** (`GET /api/oam/diagnostics`): runtime diagnostics for the service.
- **Resource usage** (`GET /api/oam/resources`): memory, processes and schedulers of the running service.
- **Peers** (`GET /api/oam/peers`): the configured NF peer URIs the SMF talks to (AMF, PCF, CHF, NRF, and its PFCP UPF peers via PFCP Peers above).
- **Log level** (`POST /api/oam/log_level`): change the log level at runtime. Choose one of emergency, alert, critical, error, warning, warn, notice, info,

or debug.

- **NRF re-registration** (`POST /api/oam/nrf/reregister`): force the SMF to re-register with the NRF, for example after an NRF restart left it stale.

Related

- **AMF**: requests session setup from the SMF over Nsmf.
- **PCF**: supplies session policy to the SMF over N7.
- **CHF**: the charging function the SMF reports usage to over N40.
- **UPF**: the SMF's N4/PFCP peer. It is the user plane the SMF programs and associates with.
- [← 5G Core overview](#)

UDM: Unified Data Management

The **UDM** is the 5G core's subscription data authority. It manages the subscription profiles for every subscriber. It tracks which serving NFs handle each UE and registers the AMF and SMF against a subscriber. It provides the authentication data used to challenge the UE. It reads the underlying subscriber records from the **UDR**, its backing store, and caches them for fast lookup. The **AMF** consults it during registration. The **SMF** consults it during session setup. The **AUSF** draws its authentication material from it. When another NF needs to know a subscriber's profile or where the network serves them, the UDM is the NF that answers.

← 5G Core

In the dashboard

The UDM dashboard opens with the four standard health cards: **API Health**, **NRF Registration** (whether the UDM is currently registered with the NRF), **NF Status**, and **License**. Each polls on an interval. Below them, it shows the data the UDM owns:

- **UE List**: every UE context the UDM holds, keyed by SUPI. This is the "which subscribers the UDM is currently serving" view. The UDM can also return a detailed context for a single subscriber by SUPI. It shows that subscriber's serving-NF registrations and cached subscription data.
- **Statistics**: the UDM's live counters, for a quick read on load and activity.
- **Configuration**: the runtime configuration the service is currently using.

Actions

The dashboard exposes actions to clear cached subscriber data. The dashboard confirms each one before it runs, since it drops live state:

- **Clear Subscriber Cache** (bulk): clears the entire subscriber data cache. Use it when the UDM serves stale profiles, for example after subscription records have changed in the UDR and the UDM should re-read them rather than serve its cached copies.

Beyond the dashboard button, the UDM's OAM API exposes a finer-grained control an operator can reach through the backend proxy:

- **Purge one UE's cache** (`/api/oam/ue_cache/{id}`): drops the cached subscriber data for a single subscriber by SUPI. Use it to force a re-read for one subscriber whose profile has changed, rather than clear the cache for everyone.

Operations

- **Runtime diagnostics** (`/api/oam/diagnostics`): shows low-level diagnostics for the running service.
- **Resource usage** (`/api/oam/resources`): reports the runtime's memory, process, and scheduler usage for a health check on the node.

- **Set log level** (`/api/oam/log_level`): changes the log level at runtime (one of `emergency`, `alert`, `critical`, `error`, `warning`, `warn`, `notice`, `info`, `debug`) to turn up detail while you troubleshoot.
- **Force NRF re-registration** (`/api/oam/nrf/reregister`): re-registers the UDM with the NRF. Use it to recover a dropped or stale registration.

Related

- **UDR: Unified Data Repository**: the backing store the UDM reads subscriber records from.
- **AUSF: Authentication Server Function**: draws its authentication material from the UDM.
- **AMF: Access and Mobility Management Function**: consults the UDM during registration and registers as a serving NF.
- **SMF: Session Management Function**: consults the UDM during session setup and registers as a serving NF.
- **← 5G Core**: the overview of all NF dashboards.

UDR: Unified Data Repository

The **UDR** is the persistent store of subscription, policy and application data that sits behind the UDM and PCF. Where the UDM and PCF are the front-end network functions that other NFs talk to, the UDR is the database they read and write. It is the single source of truth for who a subscriber is and what they are entitled to. In the Omnitouch core the UDR shares its subscriber store with the **HSS**, so one unified repository holds 4G and 5G subscribers rather than two parallel databases.

← 5G Core

In the dashboard

The UDR dashboard opens with the standard NF health cards across the top (**API Health**, **NRF Registration** (is the UDR registered with the NRF?), **NF**

Status, and **License**), each polling on an interval.

Below the cards are the two UDR-specific data sections:

- **HSS Health**: a connectivity health check that confirms the UDR can reach the shared HSS subscriber store it fronts. This answers "is the subscriber database reachable and healthy?" at a glance.
- **Subscribers**: the subscriber list drawn from the HSS, paginated and searchable by **IMSI**. Select a subscriber to return the full subscriber detail for that IMSI, so an operator can confirm a subscriber's provisioning without leaving OmniWeb.

Actions

The UDR dashboard exposes no destructive NF-level or per-row actions. It is a read and lookup view over the shared subscriber store. Its value is in interrogation: check that the store is reachable via the HSS health section, list subscribers, and drill into any one subscriber's full detail by IMSI.

Operations

Alongside the data sections, the UDR carries the operations common to every 5GC network function:

- **Set log level**: raise or lower the runtime log verbosity (`emergency`, `alert`, `critical`, `error`, `warning`, `warn`, `notice`, `info`, `debug`) via `/api/oam/log_level`, to turn up detail while you diagnose an issue.
- **Force NRF re-registration**: push the UDR to re-register with the NRF (`/api/oam/nrf/reregister`) if its registration has gone stale.
- **View runtime configuration**: inspect the UDR's live configuration through the OAM config endpoint.

Related

- **UDM: Unified Data Management**: the front-end network function that serves subscription data out of this store.
- **PCF: Policy Control Function**: reads the policy data held in the UDR.
- **HSS**: the shared subscriber database the UDR unifies with, so 4G and 5G subscribers live in one repository.
- **← 5G Core**: the full 5GC dashboard overview.

API Reference (OpenAPI / Swagger)

OmniWeb's controller (central backend) exposes a single REST API behind the same JWT login that governs the rest of the portal. That API is self-documenting:

- **Swagger UI:** `/api/docs`. Browse every endpoint. Expand a route to see its method, path/query parameters, whether it needs a token, and its summary. Click **Authorize** and paste a bearer JWT to try calls live against the running instance.
- **OpenAPI 3 spec:** `/api/openapi.json`. This is the machine readable document. Use it to import into Postman/Insomnia or to generate clients.

[← Back to Operations Guide](#)

What is documented

The spec is **generated at runtime** by introspecting the live route map, so it always matches the endpoints the running instance actually serves. No hand-

maintained file can drift. For every route it captures:

- the HTTP method and path (with path/query parameters),
- a summary and description taken from the code,
- whether a **bearer JWT** is required, and
- where applicable, the per-element **view/control** permission the route enforces.

Endpoints are grouped by area (Auth, Admin, SIM Bank, Remote SIM, APDU, Test Devices, IR.38, RAEX, TAP, and so on) so you can find a feature's routes quickly.

Getting a token

Most endpoints require a bearer token. Obtain one with `POST /api/auth/login` (the same credentials you use for the portal), then send it as `Authorization: Bearer <token>`. In Swagger UI, use the **Authorize** button so every "Try it out" call carries the token automatically.

Scope

This documents the **controller** API only. The device-agent that runs at remote sites is not part of this API: it dials *out* to the controller over a WebSocket and exposes no inbound REST endpoints, so there is nothing to call on it directly.

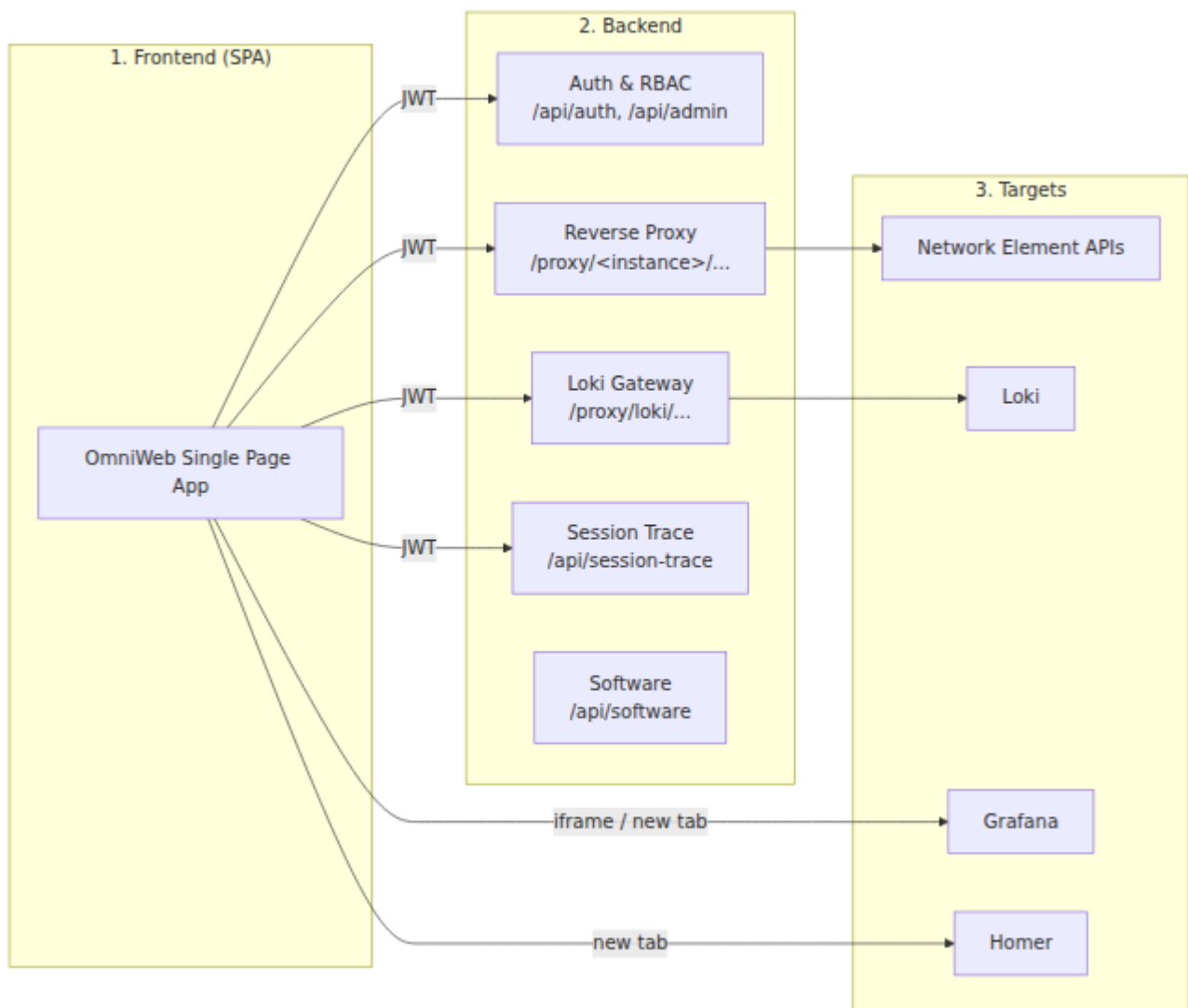
OmniWeb Architecture

This page explains how OmniWeb is put together and how a single browser session reaches every network function, Grafana, and Loki. If you only read one page to understand the system, read this one.

[← Back to Operations Guide](#)

Components

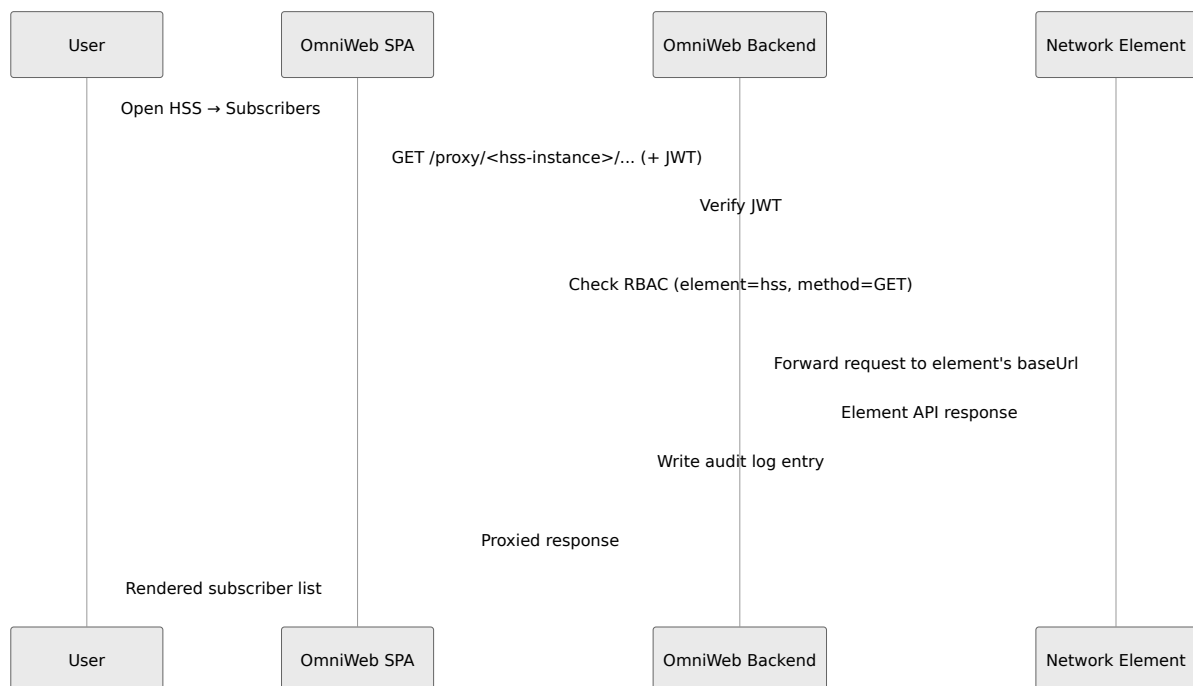
OmniWeb has three moving parts:



Component	Role
Frontend SPA	The user interface. A single-page web application that renders element pages, topology, logs, traces, and embedded dashboards.
Backend	The security and integration layer. It authenticates users, enforces RBAC, and reverse-proxies every request to the correct element API, Loki, or Grafana.
Targets	The things being managed and observed: the network elements' own management APIs, plus Grafana, Loki, and Homer.

Request Flow

Every operational request follows the same path. Nothing in the browser talks to an element directly. It always goes through the backend so that authentication, authorisation, and auditing are enforced centrally.



The proxy path encodes the **instance ID**, so OmniWeb can manage many instances of the same element type (e.g. several HSS nodes) and route each

request to the right backend.

The Configuration Model

OmniWeb is **configuration-driven**. The set of elements, their instances, and where their APIs live is declared in a single file: `public/config/omniweb.json`. The backend loads it at startup and the frontend uses it to build the sidebar, topology, and proxy targets.

```
{
  "elements": [
    {
      "type": "hss",
      "displayName": "HSS",
      "description": "Home Subscriber Server",
      "instances": [
        {
          "id": "my-hss-01",
          "name": "my-hss-01",
          "baseUrl": "https://10.0.0.1:8443",
          "tags": ["primary"]
        }
      ],
      "defaults": {
        "pollIntervalMs": 5000,
        "timeoutMs": 10000
      }
    }
  ]
}
```

Parameter	Type	Required	Default	Description
<code>type</code>	String	Yes	-	Element type. Determines which UI module renders the element. See supported types .
<code>displayName</code>	String	Yes	-	Human-readable name shown in the sidebar and topology.
<code>description</code>	String	No	-	Longer description shown in tooltips and overview pages.
<code>instances</code>	List	Yes	-	One entry per deployed node of this type. See instance parameters below.

Parameter	Type	Required	Default	Description
<code>defaults.pollIntervalMs</code>	Integer	No	5000	How often operational pages auto-refresh, in milliseconds.
<code>defaults.timeoutMs</code>	Integer	No	10000	Proxy timeout for this element type. Slow APIs (e.g. OCS) use a larger value.

Instance parameters:

Parameter	Type	Required	Default	Description
<code>id</code>	String	Yes	-	Unique instance identifier. Used in URLs (<code>/hss/<id></code>) and in the proxy path (<code>/proxy/<id>/...</code>).
<code>name</code>	String	Yes	-	Display name for the instance.
<code>baseUrl</code>	String	Yes	-	The element's management API base URL. The proxy forwards requests here. Self-signed TLS is accepted.
<code>tags</code>	List	No	<code>[]</code>	Free-form labels (e.g. <code>primary</code> , <code>geo-redundant</code>) shown as chips.

Supported Element Types

hss, mme, sgw-c, pgw-c, upf, pcscf, icscf, scscf, tas, epdg, entitlement, smsc, smsc-smpp, ipsmgw, msc, stp, hlr, camel-gw, dra, ocs, twag, ran-monitor, license.

Each type maps to a UI module with its own set of tabs, but all are operated through the same generic patterns. See [Common Operations](#).

How OmniWeb Talks to Each NF

The backend proxy is element-agnostic. It forwards whatever the frontend module asks for to the element's `baseUrl`. The *shape* of those calls depends on the element's native API:

API style	Elements	Notes
Element API	All network functions	The element's own management API, proxied verbatim under <code>/proxy/<instance-id>/...</code> . OmniWeb forwards <code>GET/POST/PUT/PATCH/DELETE</code> as the element expects; the request/response shape is whatever that element defines.
Log query (LogQL)	All (Logs tab)	Routed through <code>/proxy/loki/...</code> to Loki. See Logs & Tracing .

Because everything is proxied, adding an instance is purely a configuration change in `omniweb.json`. No code change is required to manage another node.

API reference (OpenAPI)

A consolidated OpenAPI/Swagger document (`swagger-doc.json`) ships with OmniWeb and describes the backend and element schemas (for example SMSC

frontend registrations and IMS subscriber locations). It is a reference artifact rather than a live API server.

Proxy Timeouts

Each element type has a configurable proxy timeout (`defaults.timeoutMs`). Most elements respond well within the 10-second default. Elements with heavier queries are given more headroom:

Element	Default Timeout	Why
Most elements	10 seconds	Standard API response time
OCS	180 seconds	Rating/CDR queries over large datasets

Monitoring Integration

OmniWeb does not reimplement Grafana, Loki, or Homer. It integrates them so a single login spans the whole stack:

- **Grafana** is served behind the authenticated gateway and embedded as an iframe. Each request is authorised against your OmniWeb login, so there is no second password. See [Grafana](#).
- **Loki** is reached through the backend's `/proxy/loki/*` gateway, which runs LogQL queries on the operator's behalf. See [Logs & Tracing](#).
- **Homer** (SIP capture) is auto-detected at startup (`/api/homer/status`) and, when present, opens in a new tab. It is also authorised against your OmniWeb login.

Environment Reference

Backend integration endpoints are configured via environment variables:

Variable	Default	Description
GRAFANA_URL	http://localhost:3000	Base URL of the Grafana instance OmniWeb proxies to.
LOKI_URL	http://localhost:3100	Base URL of the Loki instance used for logs and tracing.
LDAP_URI	ldap://localhost:389	OpenLDAP server for user/SSH-key sync (optional).
LDAP_BASE_DN	dc=omniweb,dc=local	LDAP base DN.
LDAP_ADMIN_DN	cn=admin,dc=omniweb,dc=local	Bind DN for LDAP writes.
LDAP_ADMIN_PASSWORD	omniweb-admin	LDAP bind password (change this).
AUDIT_API_KEY	omniweb-audit-key	Shared secret for SSH-login audit callbacks (change this).

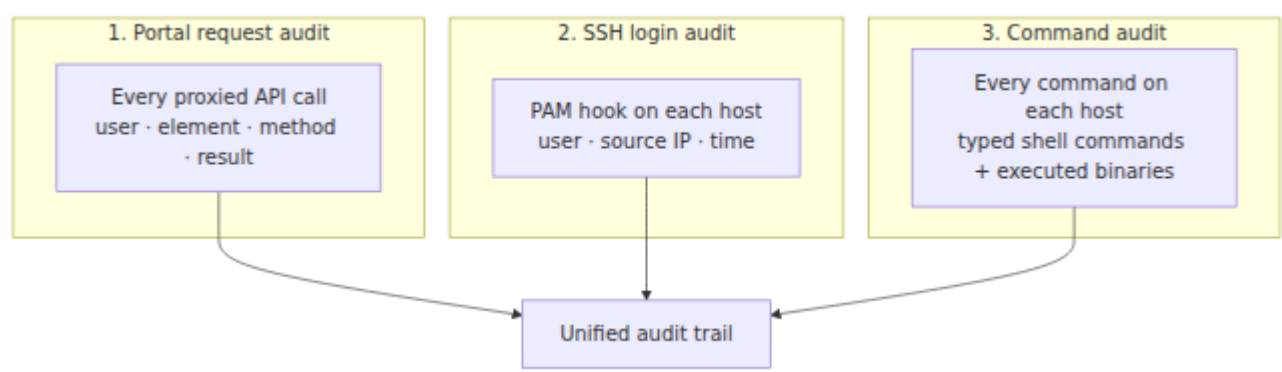
See [Authentication & Access Control](#) for the LDAP/SSH and audit details.

Audit Logging

Because OmniWeb is the single gateway to the network, it is also the single place where activity can be recorded. Auditing in OmniWeb has **three layers** that together answer "who did what, where" across both the portal and the hosts it manages.

[← Authentication & Access Control](#) · [← Operations Guide](#)

The Three Layers



Layer	Captures	Where it appears
Portal request audit	Every action taken <i>through</i> OmniWeb	Settings → Audit
SSH login audit	Every SSH login <i>to a managed host</i>	Settings → Audit (via host callback)
Command audit	Every command <i>run on a host</i> (typed + executed)	Settings → Commands

Portal Request Audit

Every request that passes through the OmniWeb proxy is recorded, regardless of whether it succeeded, failed, or was rejected by **RBAC**. Each entry captures:

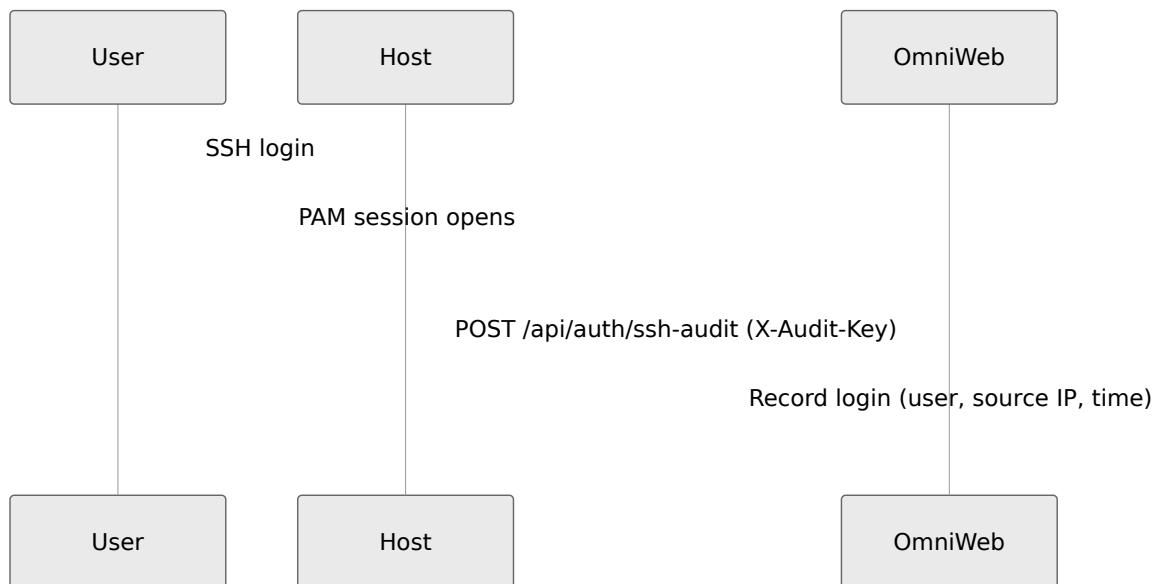
- The **user** who made the request.
- The **element type and instance** targeted.
- The **HTTP method** (which tells you whether it was a view, a change, or a delete).
- The **result** (status code).
- The **timestamp**.

This gives a complete record of who changed what, where, and when, inside the portal. It lives under **Settings → Audit**. Because reads (`GET`), writes (`POST/PUT/PATCH`), and deletes are all distinguishable by method, the audit log doubles as a change history for the network.

The Audit Trail (Settings → Audit): every action with the user, event type, element/instance, HTTP method, result, and source IP. Note the denied `DELETE` (403), the `permission_changed`, and the `ssh_login` events alongside ordinary API requests. Filter by event type, user, element, or date range.

SSH Login Audit

Hosts can report SSH logins back to OmniWeb so that host access shows up in the same audit view as portal activity. A small PAM hook on each host calls OmniWeb when a session opens, reporting the username, source IP, and service detail.



The callback is authenticated with a shared secret so only trusted hosts can write audit entries:

Variable	Default	Description
<code>AUDIT_API_KEY</code>	<code>omniweb-audit-key</code>	Shared secret the host PAM hook presents (as <code>X-Audit-Key</code>) when reporting a login. Change this , and set the matching value on each host.

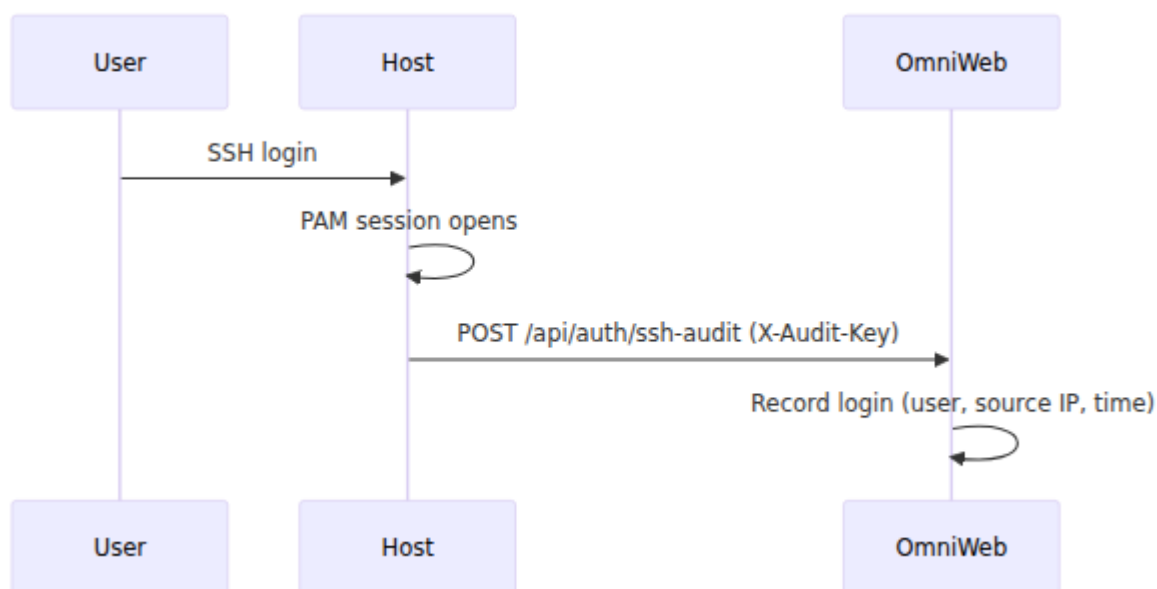
With this in place, every SSH login to a managed host appears under **Settings** → **Audit** alongside portal actions, tying together a person's portal and host activity.

Command Audit

For command-level visibility on the hosts, OmniWeb collects **two complementary streams**, both shipped to Loki and browsable under **Settings → Commands** with a source toggle:

Source	Captures	Loki stream
Typed commands	The exact command lines a user types in an interactive shell, including shell built-ins like <code>echo</code> , <code>cd</code> , <code>export</code>	<code>job="cmdaudit"</code>
Binary executions	Every binary that actually ran on the host (scripts, tools, automation)	<code>job="auditd"</code>

Neither replaces the other: built-ins never run a separate binary, so only the typed-commands stream sees them; and automation/scripts that never sit at an interactive prompt only appear in the binary-execution stream. Together they cover *what a person typed* and *what actually executed*.



Typed commands are captured by a lightweight interactive-shell hook that records each command a user runs (with user, working directory, and TTY). This is the layer that catches built-ins a binary tracer structurally cannot see.

Binary executions come from a lightweight host agent that records every binary run on the host. Because it works in userspace it functions inside LXC containers without kernel audit support. Each entry captures:

Field	Meaning
<code>login</code>	The user who originally logged in; survives <code>sudo/su</code> , so privilege changes do not hide who is acting
<code>uid</code>	Effective UID the command actually ran as
<code>filename</code>	The binary that was executed
<code>cwd</code>	Working directory at the time of execution
(after <code>]:</code>)	The full command line

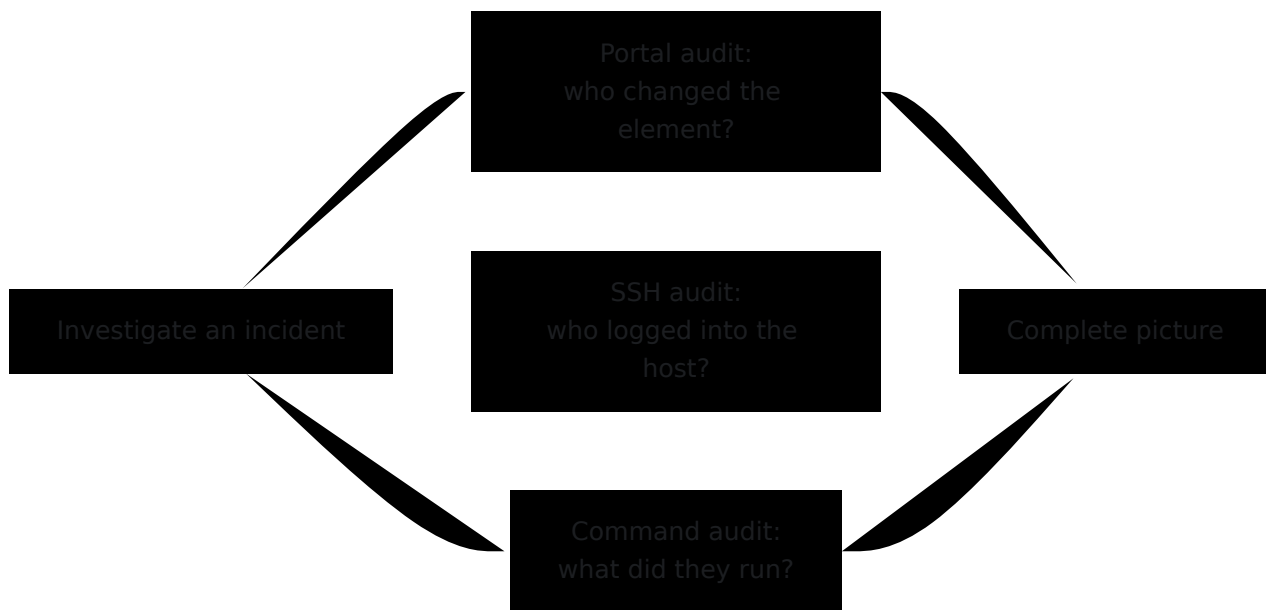
A representative execution entry:

```
2026-04-16T13:18:35 omni-stp01 [login:nickj uid:0 tty:(none)
cwd:/home/nickj filename:/usr/bin/sudo]: sudo tail -10
/var/log/auth
```

The **Settings → Commands** view defaults to the typed-command stream and lets you filter by host, search by user or command, and live-tail activity across the fleet, with the source toggle to switch to the raw binary-execution view when you want to see exactly what ran.

Note: the command streams are an operator-visibility aid captured in userspace. A determined root user can bypass them; kernel-level auditing is the hardening step if you need non-bypassable capture. For the security-relevant, structured record (logins, permission changes, denials), rely on the **Portal** and **SSH login** audit above, which are written to OmniWeb's own database.

How the Layers Fit Together



- A configuration change made **through OmniWeb** → portal request audit.
- A change made **by hand on a host** → SSH login audit (who got in) + command audit (what they did).

Together they leave no blind spot between "changed it in the UI" and "changed it on the box".

Related

- **Logs & Subscriber Tracing**: the general log explorer behind the command streams.
- **SSH Key Login**: the access side of host activity.
- **RBAC**: the permissions whose use the portal audit records.

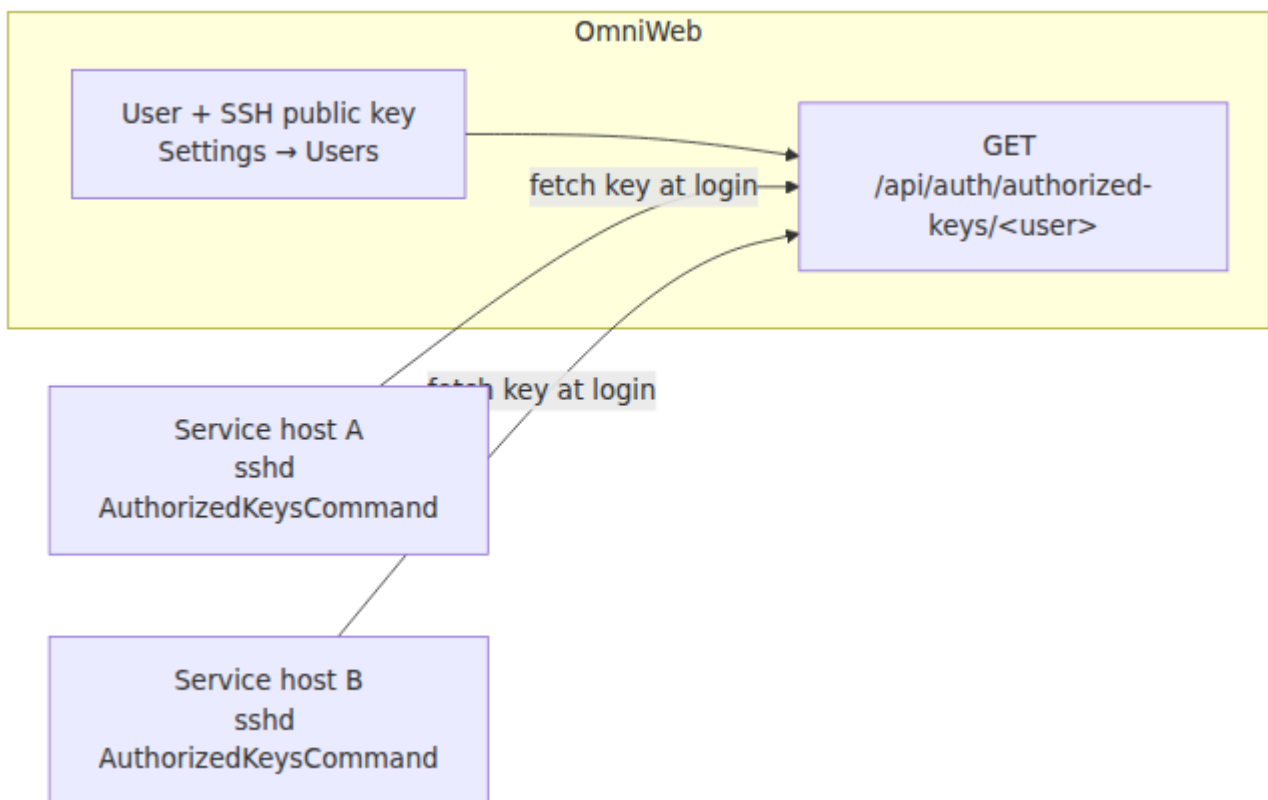
SSH Key Login (Host Access)

OmniWeb's access control extends beyond the portal to the **hosts** that run the network. Each user can carry an **SSH public key**, and the managed hosts fetch those keys **live from OmniWeb** at login time. So OmniWeb is the single place to manage *who can SSH where*, and, combined with **audit logging**, who actually did.

There is no directory server to run and no per-host `authorized_keys` files to maintain. (This replaces an earlier OpenLDAP-sync design; the directory is gone, and keys are served on demand over HTTPS.)

[← Authentication & Access Control](#) · [← Operations Guide](#)

What It Does



You give a user an SSH public key in OmniWeb once. Every managed host's `sshd` is configured to ask OmniWeb for that user's key **at the moment of login** (via OpenSSH's `AuthorizedKeysCommand`). Add a key in OmniWeb and the user can SSH to every enrolled host; disable the user and that access is gone everywhere on their **next** login attempt, with no sync, no drift, and no key files to clean up.

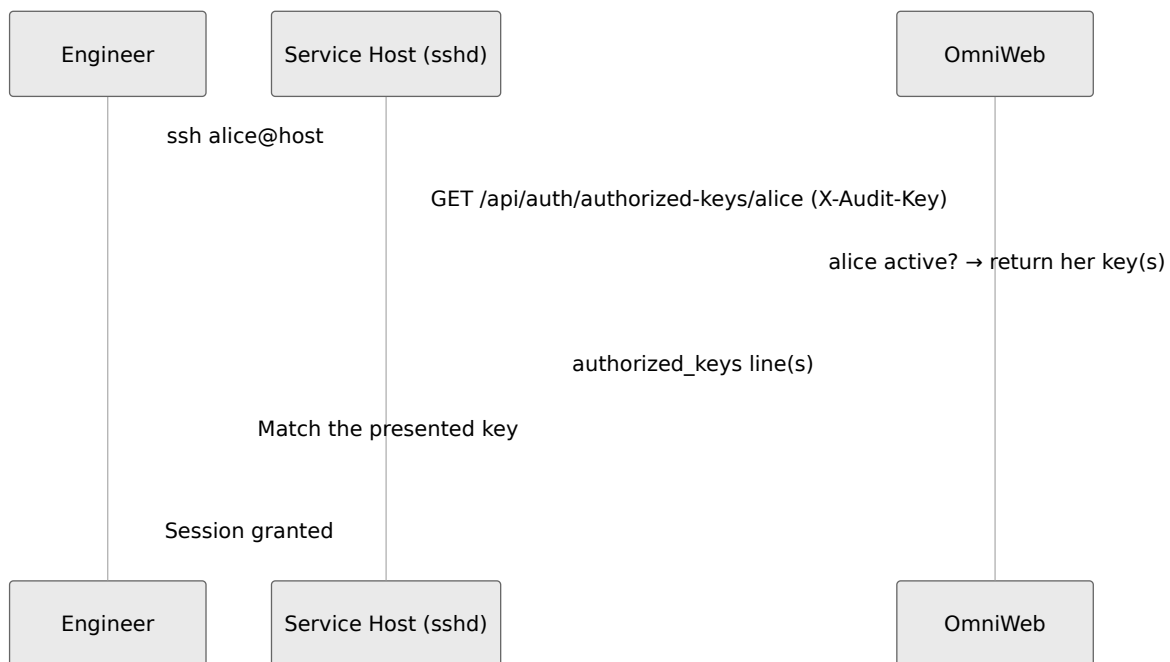
Setting a User's Key

Keys are managed on **Settings → Users**, the same screen that manages portal accounts, RBAC, and active/disabled state. Each user row shows whether an SSH key is configured; create or edit a user to paste one in.

Settings → Users: each user with whether an SSH public key is configured. The key you paste here is the key that grants host access; there is nowhere else to maintain it.

How Hosts Consume It

Each managed host runs a tiny helper wired into `sshd` as its `AuthorizedKeysCommand`. When someone connects, `sshd` runs the helper, which fetches that user's authorised keys from OmniWeb over HTTPS and hands them back for the match:



Key properties of this flow:

- **Live:** the key is fetched per login, so changes take effect immediately.
- **Active-users-only:** OmniWeb only returns keys for **active** users. When you disable or delete a user in OmniWeb, that denies their next login on every host at once. This is the revoke button.
- **No enumeration:** an unknown or disabled user, or one with no key, returns an empty result. The login simply finds no key and is denied, so the endpoint does not leak which usernames exist.
- **Survives an OmniWeb outage:** the helper caches the last good result per user on the host, so a brief OmniWeb outage does not lock everyone out.

The account itself (the Unix user, home directory, shell) is provisioned by configuration management alongside the `sshd` helper. OmniWeb owns the **identity and keys**, and the fleet automation owns **account existence** on each host.

Configuration

The key-serving endpoint is guarded by the same shared secret as the [audit callback](#):

Variable	Default	Description
<code>AUDIT_API_KEY</code>	<code>omniweb-audit-key</code>	Shared secret each host's <code>AuthorizedKeysCommand</code> helper presents (as <code>X-Audit-Key</code>) when fetching keys. Change this , and set the matching value on each host.

Because OmniWeb is the only thing that should be queried and the helper runs on trusted hosts, the endpoint is served over HTTPS and the secret is a second gate on top of the network path.

Why This Matters

- **One place to grant/revoke SSH access** across the whole fleet: the user's active state *is* their host access.
- **No scattered key files** drifting out of sync on individual hosts, and no directory server to operate and back up.
- **A single identity** tying a person's portal access, their SSH access, and, through [audit logging](#), their recorded activity.

Related

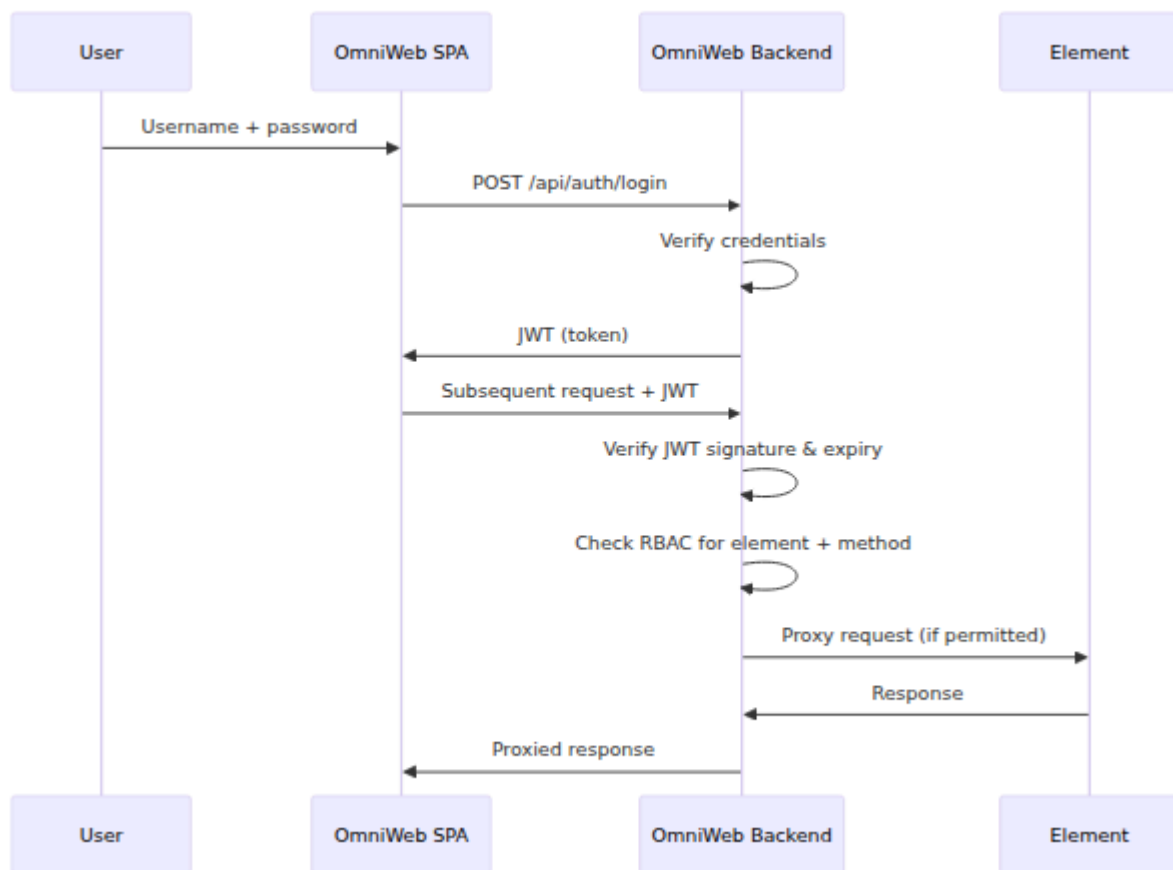
- [Audit Logging](#): record SSH logins and every command run on the hosts.
- [RBAC](#): control what the same users can do *inside* OmniWeb.

Login & JWT Sessions

OmniWeb authenticates operators with a username and password and issues a **JSON Web Token (JWT)**. That token proves who you are on every subsequent request. Crucially, it is the *same* credential that authorises the embedded Grafana, Loki, and Homer. One login spans the whole stack.

[← Authentication & Access Control](#) · [← Operations Guide](#)

Login Flow



The token is issued by `POST /api/auth/login`. Once obtained, the SPA presents it on every API call, and the backend verifies the signature and expiry before doing anything else.

The login screen. Authenticating here yields the JWT that governs the entire stack: element APIs, Loki, Grafana, and Homer.

How the Token Is Carried

The backend accepts the JWT in three locations, which lets the same token work for XHR API calls, browser navigations, and embedded iframes alike:

Location	Used for
Authorization header	Standard API (XHR/fetch) requests from the SPA
Cookie (<code>access_token</code> , <code>HttpOnly</code>)	Browser navigations and embedded content (Grafana iframe, Homer tab)
Query string	Cases where neither a header nor a cookie can be set

The cookie is **HttpOnly** (not readable by JavaScript) and uses `SameSite=Lax`, which protects the token from theft via cross-site script access while still allowing top-level navigations to the embedded tools.

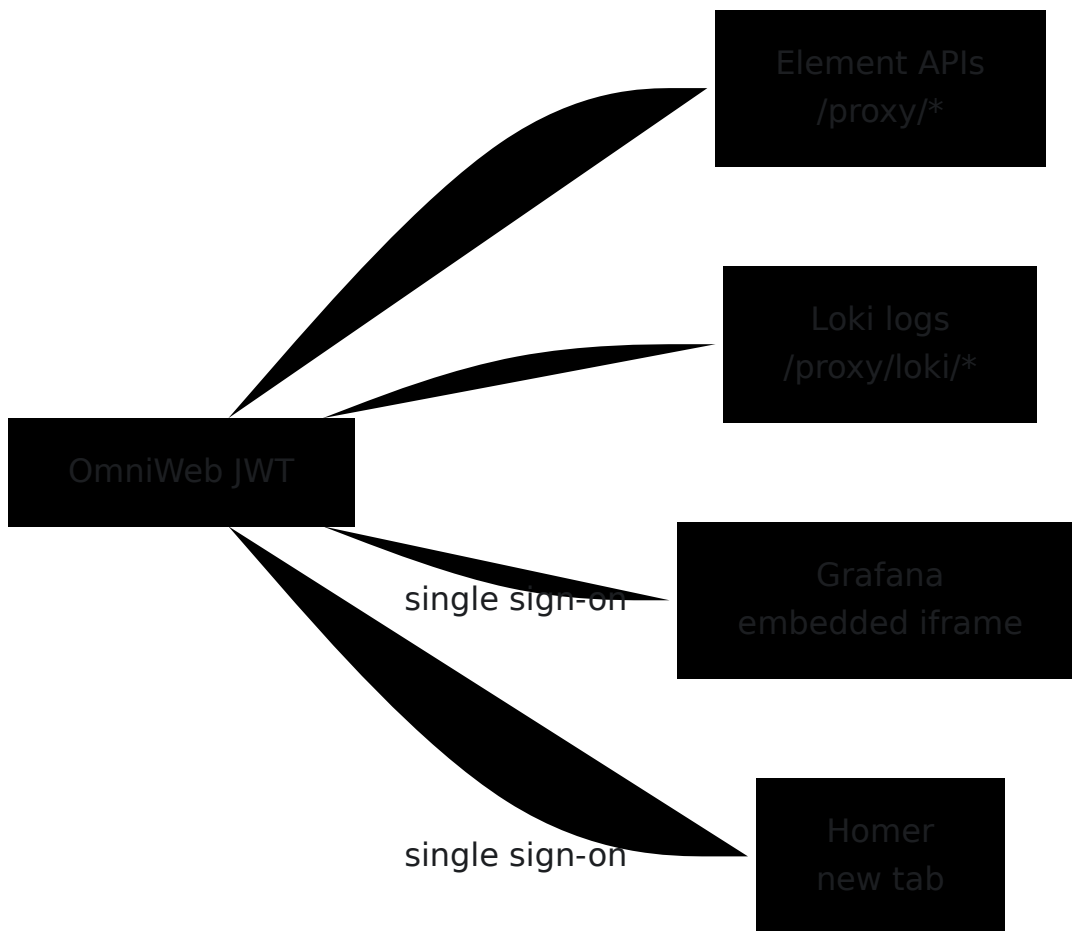
Token Lifetime

The access token has a configurable expiry, **24 hours by default** (`JWT_ACCESS_TOKEN_EXPIRES`, in seconds). When it expires, the operator is returned to the login screen and re-authenticates. There is no silent indefinite session; the lifetime is a deliberate security boundary.

Setting	Default	Description
<code>JWT_ACCESS_TOKEN_EXPIRES</code>	<code>86400</code> (24 h)	Access-token lifetime in seconds. Lower it for tighter sessions; raise it for fewer re-logins.
<code>JWT_SECRET_KEY</code>	<i>(generated)</i>	Secret used to sign and verify tokens. Set explicitly in production so tokens survive a backend restart and are consistent across instances.

One Login, Whole Stack

The same JWT authorises the integrated monitoring tools, so operators never log in twice:



- **Element APIs and Loki** are reached through the backend proxy, which checks the JWT directly.
- **Grafana and Homer** are reached through the authenticated gateway, which verifies your OmniWeb login before proxying, with no second password. See [Grafana](#) and [Homer](#).

This is why a single OmniWeb session governs the entire operational surface: dashboards, logs, traces, and element management all ride on the one token.

First Login & Changing Credentials

A fresh deployment ships with a default administrator account (`admin` / `admin`). **Change this immediately** after first login, via Settings. The default exists only to bootstrap the system and must not survive into production.

Logout & Expiry

Logging out discards the token. An expired or invalid token causes the next request to be rejected and the SPA to return to the login screen. Because element actions are only ever performed through an authenticated, RBAC-checked proxy call, an expired session can never perform a privileged action.

Related

- [Role-Based Access Control](#): what a logged-in user is allowed to do.
- [Audit Logging](#): every authenticated action is recorded.

Role-Based Access Control (RBAC)

Being authenticated tells OmniWeb *who* you are. RBAC decides *what you may do*. Access is granted at a fine grain, **per element type, per HTTP method**, which maps directly to the real operations an operator performs: view, modify, or delete. On top of that, two scopes decide *where*: which **sites** an operator may reach, and which **test devices** they may use. Every request must pass all of them.

[← Authentication & Access Control](#) · [← Operations Guide](#)

Permission Levels

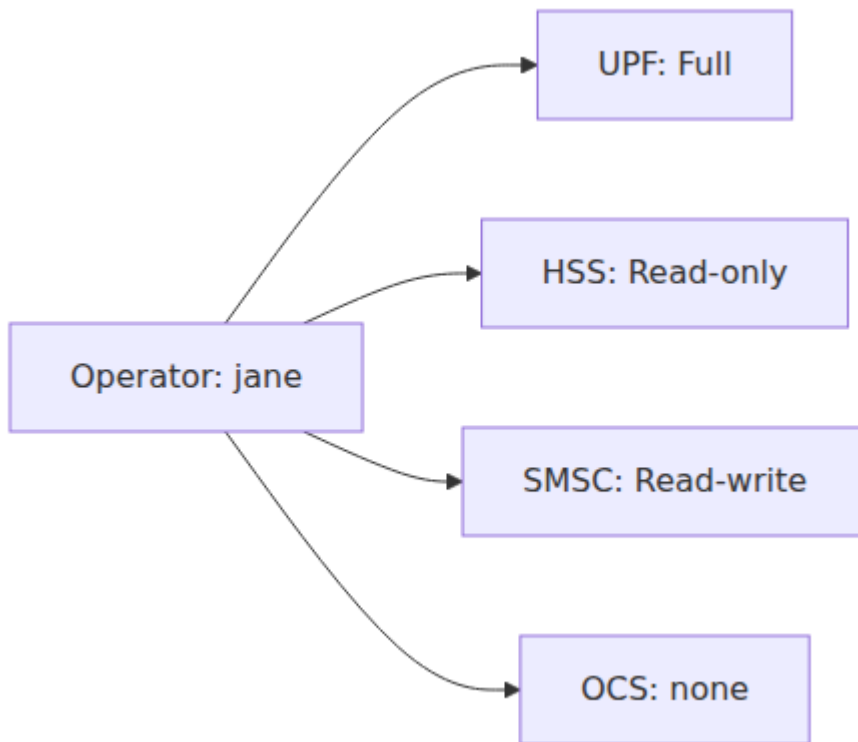
Each element type can be granted to a user at one of three levels. The level is expressed as the set of HTTP methods the user may use against that element through the proxy:

Permission	Methods allowed	What the operator can do
Read-only	GET	View data only, no changes
Read-write	GET, POST, PUT, PATCH	View and modify: e.g. add a subscriber, edit a route, change a QoS rule
Full	GET, POST, PUT, PATCH, DELETE	Everything, including deletion and clearing sessions

Because permissions are keyed to HTTP methods, they line up exactly with the **generic editing model**: an "Add" form is a POST, an "Edit" is a PUT/PATCH, and a "Delete"/"clear" is a DELETE.

Per-Element Granularity

Permissions are assigned **independently per element type**. A user can hold different levels on different elements:

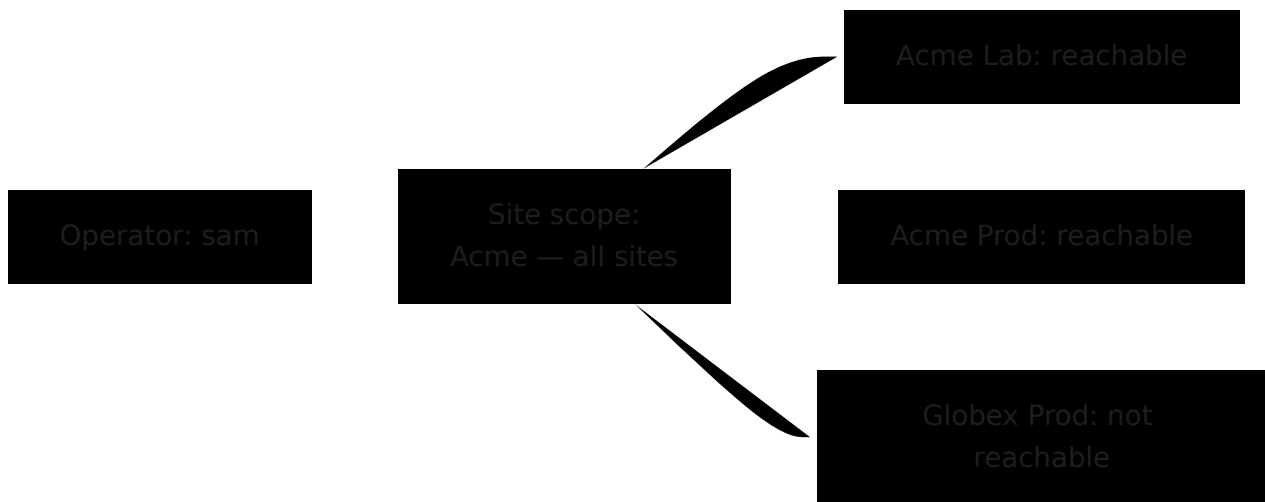


In this example Jane can clear UPF sessions, can view but not change the HSS, can add SMSC routes, and cannot see the OCS at all. This lets you give, say, a dataplane engineer full control of the UPF while keeping subscriber data read-only.

Site Scope (Multi-Site Deployments)

Element permissions answer *what* an operator may do. On a fleet with more than one site, a second, independent axis answers *where*: the set of sites they may reach at all. A request must satisfy both.

By default a user is **unrestricted** — they can reach every site, including sites added later. Scoping a user to one or more sites narrows them to exactly those, and a whole customer can be granted in one go so new sites for that customer are included automatically.



A scoped operator sees only their own sites in the site picker; the others are not listed, and the elements, logs, metrics, software inventory and call traces behind them are refused server-side. Administrators reach every site regardless of scope.

Site scope is managed under **Settings → Users**, from the globe action on a user's row. Tick the sites they may reach, or leave everything unticked to leave them unrestricted.

Test Device Scope (Phone Farm)

The phone farm is a single element type — **Test Devices** — covering every bench, phone and modem, their tools and their scheduled tests. Granting it follows the same per-method rule as any other element:

Permission	What the operator can do with the phones
Read-only	List the benches and devices, watch a phone's screen, read logs, download files and view test results
Read-write	Also type on a phone, send AT commands and SMS, upload files and run a test script
Full	Also remove a device record and clear its assigned metadata

Watching a phone's screen is treated as a **read**, not a change — an operator who may only observe can still see what a bench is doing. Typing on the phone is a separate, write-level action.

Because the whole farm is one element type, a grant is otherwise all-or-nothing. To give someone only *some* of the phones, set a **device scope**: tick the individual devices they may use, grouped by bench. Ticking nothing leaves them unrestricted — every device, including ones added later.

A device-scoped operator sees only their own devices in the fleet list, sees each bench narrowed to those devices (and benches with none of them not at all), and is refused if they name another device directly. Running a script across several phones is refused outright if any one of them is outside their scope, rather than quietly running on the subset — a partial run would read as a complete pass. Editing the schedule list replaces the whole set, so that stays with operators who hold every device.

Site scope and device scope are independent: the phone farm is deliberately site-independent, since a bench is not part of a network deployment.

Both are set under **Settings → Users**, from the globe action on a user's row.

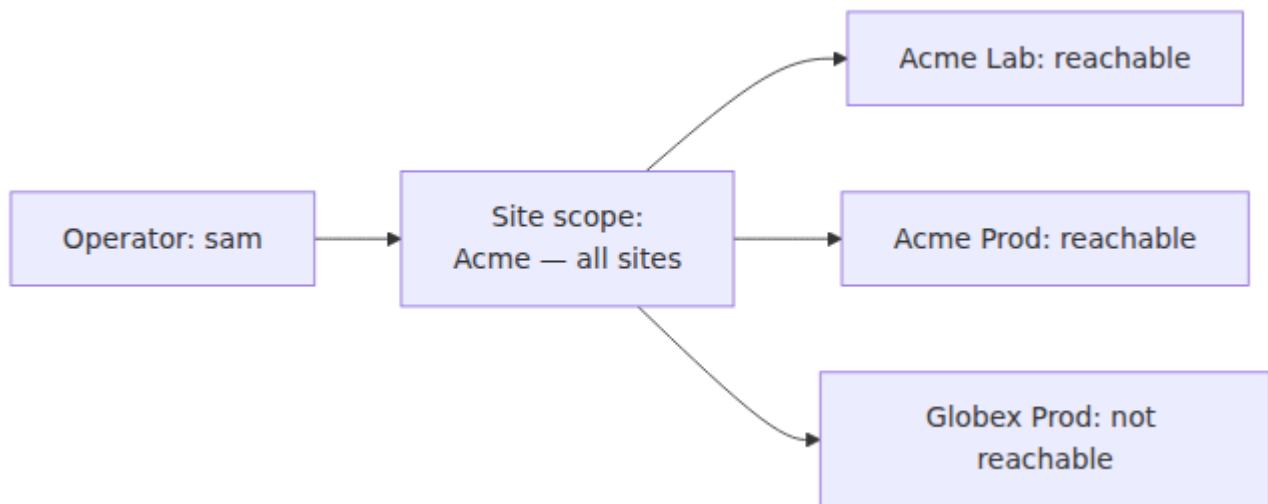
On upgrade: the phone farm previously needed only a login, so every account that existed before this change keeps full Test Devices access, recorded as a normal grant. Tighten it the same way as any other element: revoke **Test Devices** from whoever should not have the phones, and set a device scope for whoever should have only some. Accounts created afterwards start with no access until granted.

Administrators

Administrators have full access to all element types automatically. The admin role is also what grants access to user and permission management. Keep the number of administrators small, and prefer specific per-element grants for everyone else.

Managing Roles & Permissions

Permissions are managed under **Settings → Permissions**. For each user you choose, per element type, whether they have read-only, read-write, or full access (or none).

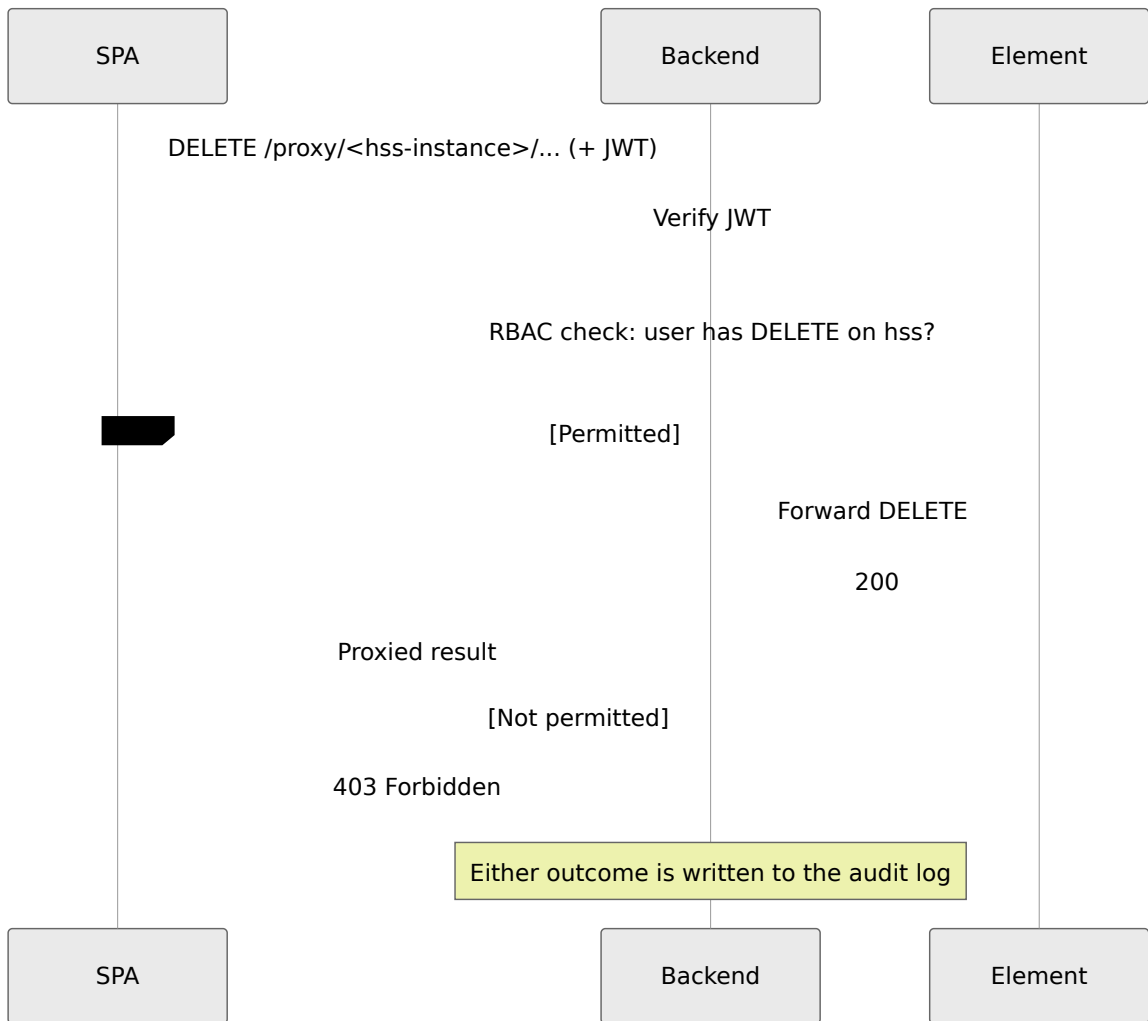


How Enforcement Works

RBAC is enforced in two complementary places:

1. **In the UI:** controls for actions you cannot perform are not shown. If you only have read access to the HSS, you will not see Add/Edit/Delete buttons there.
2. **In the backend:** every proxied request is re-checked against your permissions *server-side*. Even a request crafted to bypass the UI is rejected if your role does not permit that element/method — or if it names an element at a site outside your site scope. The site a request acts on is taken from the request itself, so it is verified rather than trusted.

This "defence in depth" means the UI is a convenience, not the security boundary. The backend proxy is the boundary.



Settings → Permissions: select a user, then grant a level (read-only, read-write, or full) per element type.

Good Practice

- **Least privilege:** grant the lowest level that lets someone do their job; widen only when needed.
- **Read-only by default** for elements an operator only needs to observe.
- **Reserve Full (DELETE)** for those who genuinely need to clear sessions or remove records.
- **Few admins:** use targeted per-element grants instead of handing out admin.
- **Scope by site** on a multi-site fleet: give a customer's operators their own sites, so a mistake cannot reach another deployment.
- **Scope test devices** to the phones someone is responsible for, so a script run cannot disturb another team's bench.
- **Pin service credentials:** an API key can be restricted to fewer sites than its owner, so a key issued for one deployment cannot act on another. A key can only ever be narrower than the account it belongs to, never wider.

Related

- [Login & JWT Sessions](#): authentication that RBAC builds on.
- [Common Operations](#): how permissions shape the everyday UI.
- [Audit Logging](#): both permitted and denied actions are recorded.

Authentication & Access Control

OmniWeb is the single front door to the entire network, so access control is one of its most important responsibilities. It uses JWT-based login, fine-grained role-based access control (RBAC) over every element, SSH-key login for host access, and a unified audit trail spanning the portal and the hosts it manages.

This area is broken into focused pages:

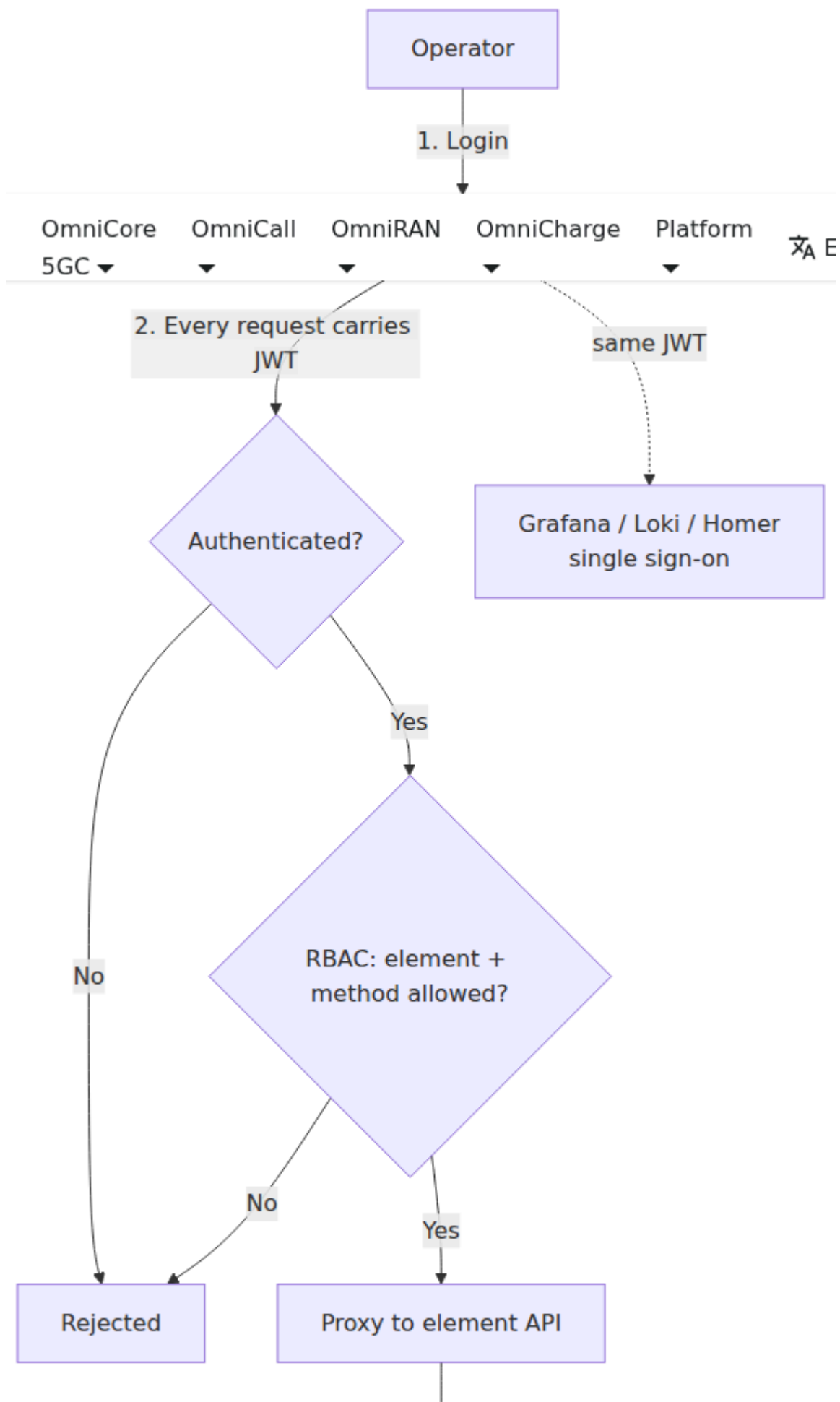
[← Back to Operations Guide](#)

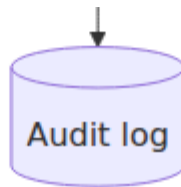
In This Section

- **Login & JWT Sessions:** How operators log in, how the JWT works, token lifetime, and how a single login spans Grafana, Loki, and Homer.
- **Role-Based Access Control (RBAC):** Per-element, per-method permissions; how to grant them; and how they are enforced.
- **SSH Key Login:** Managed hosts fetch each user's SSH public key live from OmniWeb (AuthorizedKeysCommand), so host access is granted/revoked centrally.
- **Audit Logging:** The three layers of audit: portal request audit, SSH-login audit, and command audit.

The Access Control Model

OmniWeb layers four controls. A request must pass each before it reaches an element, and the whole chain is recorded.





Layer	What it controls	Page
Authentication	Who you are, proven by a signed JWT	Login & JWT
Authorisation (RBAC)	What you may do, per element type and HTTP method	RBAC
Host access	Who may SSH where; keys served live from OmniWeb	SSH Key Login
Audit	What actually happened, across portal and hosts	Audit Logging

Why It Matters

Because OmniWeb proxies every element, Grafana dashboard, Loki query, and Homer session, this single access-control chain governs the *entire* operational surface of the network. There is one place to grant access, one place to revoke it, and one trail that records who did what, on the portal and on the underlying hosts.

One trail for everything: API requests, logins, SSH logins, and permission changes, with user, element, method, result, and source IP. See [Audit Logging](#).

Start with [Login & JWT Sessions](#).

Call Trace (SIP Trace)

Call Trace is the OmniWeb tool for voice-call troubleshooting. It records the SIP signalling and the RTP media of calls that cross the IMS. It then lets you search those calls, open a full per-call report, follow the call as a signalling ladder across the network, measure the voice quality, listen to the audio, and download the raw packet capture (PCAP).

Call Trace is a site-level tool. It appears in the sidebar under **Session Trace** as **SIP Trace**. The tool only appears when the capture service is configured and reachable for the selected site.

[← Operations Guide](#)

Contents

- [What it is](#)
- [Architecture](#)
- [Search](#)
- [Call detail](#)
- [Ladder \(Session Trace\)](#)
- [QoS tab](#)
- [Listen](#)
- [PCAP download](#)
- [Active Calls](#)
- [Live Sniffer](#)
- [Remote Capture Agents](#)
- [Recording Rules and Retention](#)
- [Privacy and lawful interception](#)
- [How Call Trace compares to Homer and Session Trace](#)

What it is

Call Trace answers one question in depth: **what happened to this call?** For any recorded call it gives you:

- The two parties, their numbers, their SIP domains, and the display name.
- The SIP IP address of each party, linked to its OmniWeb network element.
- The final SIP response and the disconnect reason (SIP and Q.850).
- The voice quality of each media direction: MOS, jitter, packet loss, and round-trip delay.
- The full SIP message flow (ladder) across every IMS element that touched the call.
- The recorded audio and the raw PCAP.

Call Trace records calls first-hand. Because it captures the SIP Call-ID and the media on the wire, a call maps to its packet capture with no guesswork.

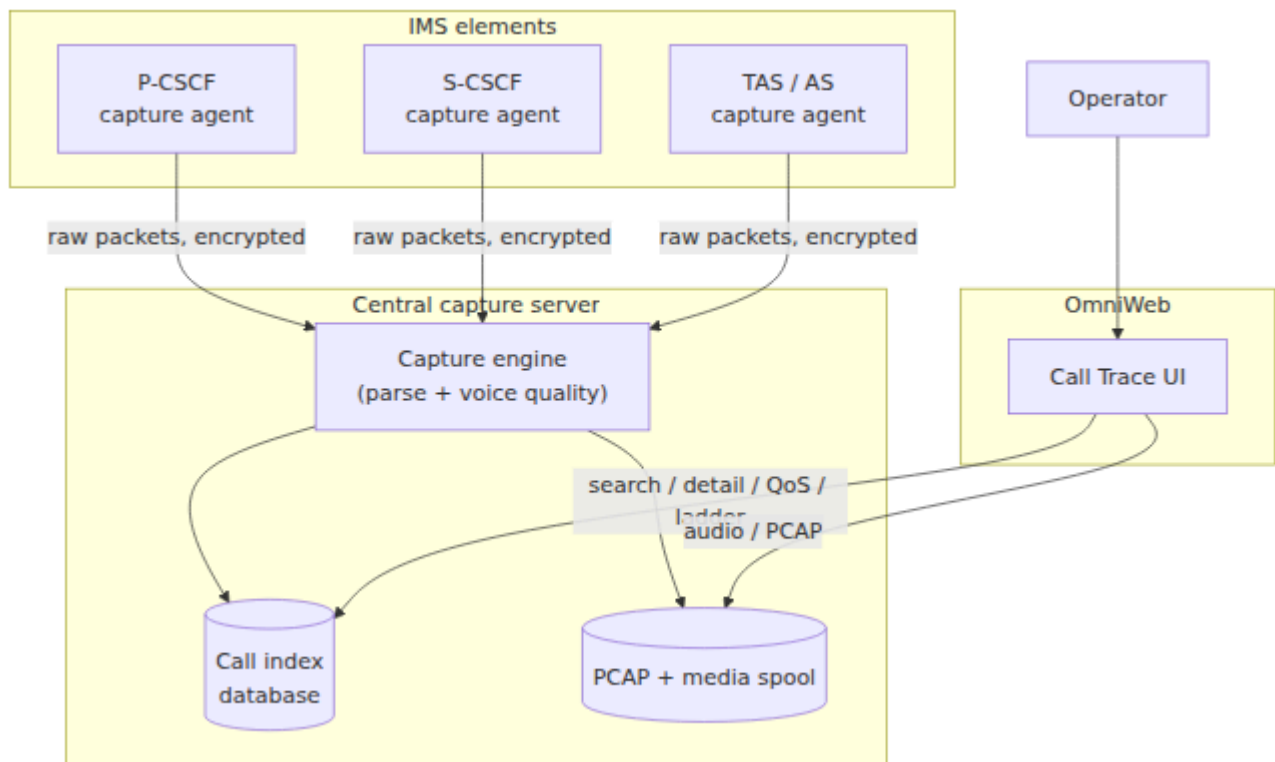
Voice quality is reported as **MOS** (Mean Opinion Score). MOS is a 1 to 5 score. A score of 4.0 or higher is good, 3.0 to 4.0 is fair, and below 3.0 is poor. OmniWeb colours MOS values: green for good, amber for fair, and red for poor. AMR-WB (wideband) calls in a healthy network score about 4.5.

Architecture

Call Trace uses two parts:

1. **Capture agents** run on the IMS elements (P-CSCF, S-CSCF, and the TAS). Each agent captures the SIP and RTP packets it sees on its host. The agents do not store or parse packets locally. They forward the raw packets to the central capture server over an encrypted channel.
2. The **central capture server** parses every packet, computes the voice quality, and writes an index to its database. It holds the packet-capture spool and the recorded media. OmniWeb reads this one server for search, call detail, ladder, QoS, audio, and PCAP.

This design keeps all recorded data on one central server. The IMS elements keep no capture data of their own.



The capture agents give Call Trace a view from many points in the network. A single call is stitched together from every agent that saw it. This is what lets the ladder show the multi-hop SIP flow across the P-CSCF, the S-CSCF, and the TAS.

Search

Open Call Trace to reach the **Search** view. Search finds completed calls in the central call index.

The Search view. Each row stacks the caller and called party (number, domain, name, SIP IP, and user agent) next to per-direction RTP cells that show source IP, codec, three MOS estimates, jitter, and loss.

Filters

Set one or more filters, then select **Search**. All filters combine with AND.

Filter	What it matches
Number	The caller number OR the called number.
Time Range	A preset window (Last 15m to Last 7d) or a custom start and end.
Direction	Inbound, Outbound, or Internal. Direction comes from the role of the caller and called IP.
Min MOS	Keep only calls whose worst-direction MOS is at or above this value. Use this to find good calls, or invert your reading to find bad ones.
Max loss	Keep only calls whose worst-direction packet loss is at or below this value.

Results grid

Each result row shows:

- **Time:** the call start and end.
- **Caller** and **Called:** a stacked cell with the number and domain (num@domain), the display name, the SIP IP (a link to that element in OmniWeb), and the user agent. A **hung up** chip marks the party that sent the BYE.
- **Result:** the final SIP response code and text.
- **RTP A** and **RTP B:** one cell per media direction. Each cell shows the source IP, the codec, three MOS estimates (fixed 50 | fixed 200 | adaptive), the jitter, and the lost-packet count. The three MOS values are colour-coded.
- **Dur (s):** the call duration in seconds.
- **PCAP:** a per-row download button for that call's packet capture.

Select a row to open the [call detail](#).

Call detail

The call detail view is the full report for one call. It opens when you select a search result. The URL carries the call identifier, so you can bookmark or share a call.

The call-detail view. The header shows the Call-ID and context. The summary strip shows Start, Duration, PDD, Ringing, Connected, the last SIP response, and the disconnect reasons. The two-column table shows Identity and voice quality for the Caller (A) and the Called (B) legs.

Header and context

The header shows the wire **Call-ID**. A context block below it shows the call **Direction**, the owning **Sensor** (the capture agent), which party **Hung up**, the **Ended** time, and the **Caller name**.

Summary strip

The summary strip is a one-line overview of the call:

Field	Meaning
Start	When the call started.
Duration	Total call length.
PDD	Post-dial delay: the time from INVITE to the first ringing indication.
Ringing	How long the phone rang before answer.
Connected	How long the call was connected (media flowing).
Last SIP response	The final SIP response code and text.
SIP reason	The SIP disconnect cause, when the capture engine records one.
Q.850 reason	The Q.850 disconnect cause and text, when present.
Caller and Called	The two parties as <code>num@domain</code> .

Identity and voice-quality table

The main table has two columns: **Caller (A)** and **Called (B)**. Every IP links to its OmniWeb network element when the element is known. The table has three groups.

Identity:

Row	Meaning
SIP IP	The party's SIP signalling IP.
RTP IP	The party's RTP media IP.
RTP Port	The RTP media port.
User Agent	The party's SIP user agent.
Codec	The audio codec, for example AMR-WB.
Received packets	The count of received RTP packets.
Lost packets	The count of lost RTP packets.

MOS: three MOS estimators, each shown as `average | minimum`:

Row	Meaning
MOS fixed 50	MOS over a fixed 50-packet window.
MOS fixed 200	MOS over a fixed 200-packet window.
MOS adaptive 500	MOS over an adaptive window up to 500 packets.

Jitter and RTCP: the RTP jitter (max and average), the RTCP jitter, the RTCP loss and fraction-lost count, and the RTCP round-trip delay (max and average).

Ladder (Session Trace)

The **Ladder** tab draws the call as a time-ordered signalling flow. Call Trace stitches the SIP messages together from every capture agent that saw the call. The result is a multi-hop ladder across the IMS elements: for example P-CSCF to S-CSCF to TAS.

The Ladder tab. The columns are the IMS elements the call crossed. Each rung is a SIP message. The ladder shows the INVITE to 100 to 180 to 200 to ACK to BYE flow, with per-message time deltas and the request-to-response round-trip.

Each column is one IMS element, labelled with its role and hostname. Each rung is a SIP message drawn from source to destination. The ladder shows:

- The **per-message time delta** (the gap since the previous message).
- The **request-to-response round-trip** for each response, paired back to its request.
- **Absolute**, **relative**, and **ago** time modes for the time column.
- **Inline packet decode**: select a rung to see the full SIP message text and a hex view.

A **Session Details** panel lists the sessions in the ladder. For a single call there is one session, so the panel stays collapsed.

The ladder uses the same rendering as the network-wide Session Trace Live Flows. See [Logs & Subscriber Tracing](#) for the wider Session Trace tool.

QoS tab

The **QoS** tab shows the voice quality over the life of the call.

The QoS tab. The tables show the PDV (delay) distribution buckets and the loss-in-a-row breakdown for each direction. An interactive delay and MOS quality graph is drawn over the length of the call below the tables.

The QoS tab has three parts:

- An interactive **delay / jitter and MOS quality graph** over the call. The graph is per media direction. You can read the delay and the MOS at any point in the call, which shows where the quality dropped.
- The **PDV (delay) distribution**: how many packets fell into each delay bucket (for example 50 to 70 ms, 70 to 90 ms, and so on up to over 300 ms). A distribution that stays in the low buckets is healthy.
- The **loss-in-a-row** breakdown: how many times 1, 2, 3, and up to 10 packets were lost back to back. Isolated single losses are far less audible than long runs of loss.

Both breakdowns are shown per direction (Caller and Called), so you can tell which leg had the problem.

Listen

The **Listen** panel plays the recorded call in the browser. The audio is decoded from the captured RTP. The caller is the left channel and the callee is the right channel, so you can hear each side on its own.

Listen only works when the call's media was recorded and its codec can be decoded. See [Recording Rules and Retention](#) for how media capture is armed.

PCAP download

Every call has a **Download PCAP** button, in both the search grid and the call-detail header. The download is the raw packet capture (SIP and, when recorded, RTP) for that one call. Use the PCAP to inspect the call in an external tool such as Wireshark.

Active Calls

The **Active Calls** view shows calls that are in progress right now. It reads the live call list from the capture agents and refreshes on a short interval, so rows appear and disappear as calls set up and tear down.

The Active Calls view. Each row is a live call with its state (Connected or Setup), the two parties and their IP addresses, the direction, the growing duration, the codec, the live MOS, and the owning capture agent.

Each row shows the call **State** (Connected, Setup, or Failing), the two parties, the caller-to-called IP path, the **Direction**, the growing **Duration**, the **Codec**, the live **MOS**, the last SIP response, and the owning **Sensor** (capture agent). Turn **Auto-refresh** off to hold the current list.

Live Sniffer

The **Live Sniffer** view shows the health of the central capture server and its capture agents.

The Live Sniffer view. The Central Server panel shows version, uptime, active-call count, throughput, packet drops, and load average. The Remote Capture Agents table lists each agent. The Spool Disk I/O panel shows utilisation and throughput.

The view shows:

- **Central Server:** version and build, uptime, active-call count, capture throughput, packet drops, load average, and the last time a call record was written.
- **Remote Capture Agents:** the per-element agents and their status (see [Remote Capture Agents](#)).
- **Spool Disk I/O:** the utilisation, read and write throughput, IOPS, latency, capacity used, and buffer level of the capture spool disk.
- A live activity feed of the calls in progress.

Watch this view to confirm the capture engine is healthy, is not dropping packets, and has spool disk headroom.

Remote Capture Agents

The **Remote Capture Agents** table (on the Live Sniffer view) lists the capture agents that run on the IMS elements. Each row shows:

Column	Meaning
Agent	The agent hostname. Select it to open that host in the Host List.
Sensor ID	The numeric identifier of the agent in the capture index.
Status	Active, Idle, or Unknown, derived from live calls and recent call records.
Active calls	How many calls the agent is capturing right now.
CDRs	How many call records the agent has produced.
Last seen	When the agent last produced a call record.

Use this table to confirm every IMS element that should capture is capturing. An agent that is unexpectedly Idle or missing means that element's traffic is not being recorded.

Recording Rules and Retention

The **Recording Rules** view controls which calls have their **media** recorded, and shows the retention of the capture spool and the index database.

The Recording Rules view. Rules are grouped by match type: By IP, By number, and By SIP domain (a By SIP header group also exists). Each rule arms RTP media capture, skips a target, or saves the QoS graph data.

Recording rules

By default the capture engine records SIP signalling for every call, but it does **not** record the RTP media. A recording rule arms media capture for a specific target. The capture engine re-reads the rules live, with no restart.

Rules are grouped by how they match a call:

Group	Matches on
By IP	A source or destination IP and mask.
By number	A caller or called number prefix.
By SIP domain	A SIP domain.
By SIP header	An arbitrary SIP header value.

Each rule carries a set of flags:

Flag	Effect
RTP	Arm RTP media capture for the matched target. This is what lets you listen to the call and get the full media PCAP. Keep RTP off globally and arm it per target.
Skip	Drop the matched target entirely: no call record and no capture. Use this for known test traffic.
Graph	Save the fine-grained QoS graph data for the matched target, which drives the QoS graph .
rtcp	Record the RTCP quality reports for the matched target.
enabled	Turn the rule on or off without deleting it.

To arm capture for one target, add a rule in the matching group, fill in the match value, and turn on **RTP**. To stop recording known test traffic, add a rule with **Skip** on.

Retention

The Retention panel shows how long recorded data is kept. Retention is set centrally on the capture server. The panel is read-only in OmniWeb.

The retention settings control two things: the size and age limits of the on-disk spool (split by SIP, RTP, and graph data), and the age limit of the call index and per-stream quality rows in the database. The table below lists the settings the panel reports:

Setting	Meaning
<code>maxpoolsize_gb</code>	Maximum total spool size, in GB. The oldest data is removed when this is reached.
<code>maxpooldays</code>	Maximum age of any spool data, in days.
<code>maxpoolsipsize_gb</code>	Maximum size of the SIP spool, in GB.
<code>maxpoolsipdays</code>	Maximum age of the SIP spool, in days.
<code>maxpoolrtpsize_gb</code>	Maximum size of the RTP (media) spool, in GB.
<code>maxpoolrtpdays</code>	Maximum age of the RTP spool, in days.
<code>maxpoolgraphsize_gb</code>	Maximum size of the QoS-graph spool, in GB.
<code>maxpoolgraphdays</code>	Maximum age of the QoS-graph spool, in days.
<code>cleandatabase_cdr_days</code>	Maximum age of call records in the index database, in days.
<code>cleandatabase_rtp_stat_days</code>	Maximum age of per-stream quality rows in the database, in days.

Media data (RTP) is large. Keep the RTP retention shorter than the SIP retention, and arm RTP capture only for the targets you need.

Privacy and lawful interception

Recording call **media** is a lawful-interception decision. Signalling capture records who called whom and when. Media capture records the **content** of the call: the voice audio. Arm media capture (an **RTP** recording rule) only for targets you are authorised to record, and only for as long as you are authorised.

Keep RTP capture off by default and arm it per target. Set the RTP retention to the shortest period your policy allows. Follow your local privacy law and consent rules for both signalling and media capture.

How Call Trace compares to Homer and Session Trace

OmniWeb has three ways to look at signalling. Reach for the right one:

	Call Trace (SIP Trace)	Homer (SIP Capture)	Session Trace
Scope	SIP and RTP media for IMS calls	SIP only	The whole network: Diameter, GTP-C, PFCP, SIP, MAP
Question it answers	"What did this call do, and how good was the voice?"	"What did this call's SIP flow do?"	"What happened to this subscriber across every element?"
Voice quality (MOS)	Yes	No	No
Recorded audio	Yes	No	No
PCAP download	Yes, per call	No	No
Best for	Diagnosing a VoLTE call's quality, media, and flow	A quick look at a SIP flow	Following one subscriber end to end

Reach for Call Trace when you must judge a call's **voice quality**, hear the audio, or get the media PCAP. Reach for **Session Trace** when you follow a subscriber across the whole stack.

Related

- **Logs & Subscriber Tracing**: the network-wide Session Trace and the shared ladder.
- **Homer: SIP Capture**: the SIP-only capture tool.
- **Network Topology**: the Host List that the capture agents link into.

Common Operations

Every network element in OmniWeb is different on the inside, but the way you *work* with them is deliberately the same. This page documents the patterns shared by all elements: viewing data, editing and changing NF setups, reading configuration, running actions, watching logs, and understanding polling and permissions. Learn these once and they apply to the HSS, the UPF, the SMSC, the MSC, and every other NF.

The screenshots below come from specific elements, but they illustrate the *generic* pattern: the same controls appear, with different fields, on every element.

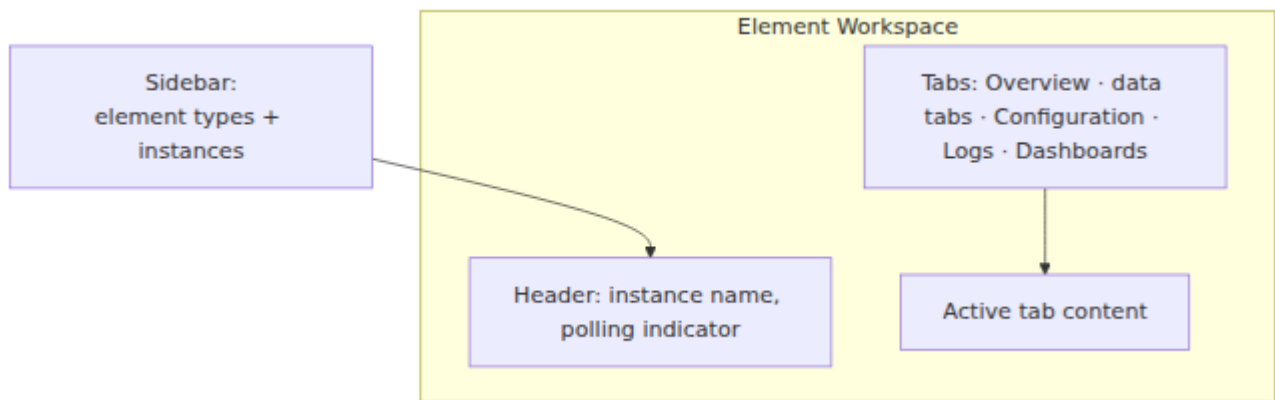
[← Back to Operations Guide](#)

Table of Contents

- [Page Anatomy](#)
- [Viewing Data](#)
- [Detail Views](#)
- [Editing & Changing NF Setups](#)
- [Running Actions \(clear, release, test, ping\)](#)
- [Viewing Configuration](#)
- [Logs on Every Element](#)
- [Dashboards on Every Element](#)
- [Real-Time Polling](#)
- [Permissions & What You Can Change](#)
- [Multi-Instance & Fleet Views](#)

Page Anatomy

Selecting an element instance in the sidebar opens its workspace. Every element workspace shares the same layout:



- **Overview:** the landing tab: health, key counts, and component status for that instance.
- **Data tabs:** the element-specific views (sessions, subscribers, peers, routing, etc.).
- **Configuration:** a structured view of the element's runtime configuration (where exposed).
- **Logs:** a live Loki log viewer scoped to that element.
- **Dashboards:** embedded Grafana dashboards for that element (when Grafana tags match).

Sidebar entries are ordinary links, so you can **Ctrl/Cmd-click** (or middle-click, or right-click → *Open in new tab*) any of them to open that workspace in a new browser tab. This is handy for watching two elements at once.

Viewing Data

Operational data is presented in a consistent, sortable table throughout OmniWeb. Tables provide:

- **Sorting** by column.
- **Pagination** for large result sets.
- **Search / filtering** where the element API supports it (e.g. search by IMSI/MSISDN or UE IP).
- **Skeleton loading** while data is fetched, and a clear **empty state** when there is nothing to show.
- **Row click-through** to a detail view where one exists.

Values are formatted for readability: timestamps as local time, durations as `h m s`, byte counts as KB/MB/GB, and large numbers with thousands separators.

The generic list pattern (here, a subscriber list): searchable, sortable, paginated, with click-through and inline editing. The same table drives sessions, peers, routes, and every other listing.

Tables can render richer cells: here, utilisation bars per row. The pattern is the same; only the columns differ.

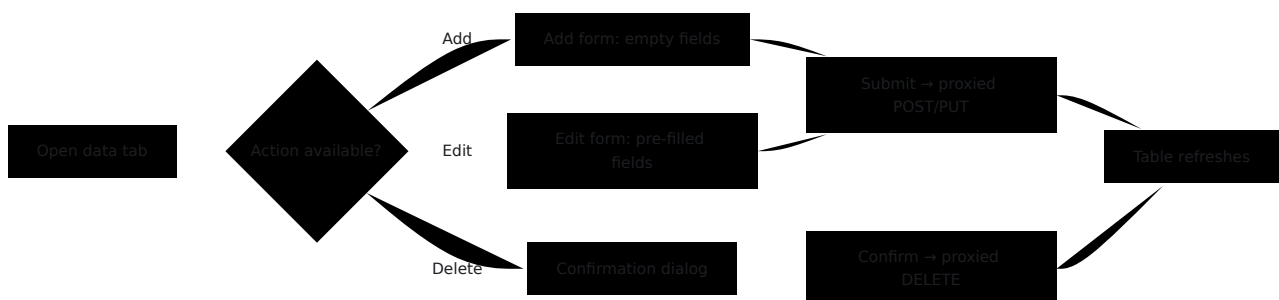
Detail Views

Where a row has more to show, click it to open a detail view. These reuse the same chrome (header, polling, actions) but present the single record in depth, for example an interactive relationship map for a complex object:

A detail view opened from a table row: here, an interactive map of one session's internal rules. Detail views are reached the same way everywhere: click the row.

Editing & Changing NF Setups

Wherever an element's API supports modification, OmniWeb exposes it through a consistent **action form**. The same form component drives "add a subscriber", "add a route", "edit a QoS rule", and "change a profile". Only the fields differ.



Action forms support these field types: **text**, **number**, **select** (dropdown), **boolean** (toggle), **password** (masked, with a show/hide toggle), and **multi-line text**. Each field can be marked required, carry a placeholder, and show

helper text. The form shows a loading state while submitting and reports success or failure via an on-screen notification.

The HTTP method behind each action maps to a permission level. See [Permissions](#).

The same pattern, three elements

These illustrate the *same* mechanic applied to different NFs:

Task	Where	What happens
Add/edit a subscriber	HSS → Subscribers → Add	Fill the form (IMSI, MSISDN, profile, keys) → proxied <code>POST/PUT</code> → list refreshes
Add/change a route	SMSC → Routing → Add (or MSC → Routing)	Fill the rule (pattern, destination, priority) → proxied write → table refreshes
Clear/release a session	UPF → Sessions (or MSC → Calls)	Select the record → confirm → proxied <code>DELETE</code> → list refreshes

The mechanics are identical: form or confirmation dialog, proxied write, table refresh. Only the element and fields change.

Changing routing is the same edit pattern: the routing table is a list you add to and edit; some elements additionally parse destinations into readable chips and offer a simulator to test a change before it goes live.

Running Actions

Beyond CRUD, many elements expose **operational actions** that do not fit a simple form. These are surfaced as buttons that trigger a proxied call and show the result inline. Examples of the same button-action pattern:

Example action	Element	Effect
Force release a call	MSC → Calls	Tears down an active call
OPTIONS ping a SIP peer	MSC → SIP Peers	Tests reachability of a SIP trunk
MAR test a Diameter peer	TWAG → Diameter	Sends a test Multimedia-Auth-Request
Activate/deactivate a redirect	UPF → Walled Garden	Redirects or restores a subscriber
Ping from the node	UPF → Diagnostics	Runs a ping from the element itself
Disconnect a session	TWAG → Sessions	Drops an authenticated WiFi session
Manual paging	MSC → Paging	Triggers a paging request

Destructive actions (release, delete, deactivate) always require explicit confirmation before they run.

Operational actions appear as buttons alongside the data they affect: here, activating/deactivating subscriber redirects and managing whitelisted IPs. Destructive actions prompt for confirmation first.

Viewing Configuration

Elements that expose their runtime configuration have a **Configuration** tab. OmniWeb renders it in a structured, readable layout with a **Raw JSON** toggle for the full underlying document.

Two behaviours are worth knowing:

- **Mixed config formats are normalised.** Some elements return configuration as Python-style dictionaries (`'key': True`, `None`) rather than strict JSON. OmniWeb parses these into a clean view automatically and falls back gracefully if it cannot parse a value.
- **Secrets are masked.** Any field whose name looks sensitive (matching `password`, `secret`, `token`, or `credential`) is displayed as `***` rather than its real value, both in configuration views and in form fields.

The generic configuration view: settings in a structured layout, with a Raw JSON toggle and sensitive values masked. Elements that expose config (UPF, TAS, ePDG, SMSC frontends, OCS, RAN Monitor) all use this pattern.

Logs on Every Element

Every element has a **Logs** tab backed by Loki. It queries that element's service logs by host/label, supports structured filtering, and streams new lines as they arrive. Because the log model is identical across elements, it is documented in full on its own page:

→ **Logs & Subscriber Tracing**

Every element's Logs tab uses the same viewer: structured, colour-coded, live-streamed Loki logs with service and severity filtering.

Dashboards on Every Element

Elements tagged for Grafana show a **Dashboards** tab. OmniWeb discovers matching dashboards by tag and renders them inline, so you can see the statistics for an element without leaving its workspace. See:

→ [Grafana Dashboards & Statistics](#)

Real-Time Polling

Operational pages auto-refresh. The **Polling Indicator** in each page header shows:

- **Active (green dot)** or **paused**.
- The **interval** (default 5 seconds, set per element via `pollIntervalMs`).
- A **spinner** while a fetch is in flight.

- The **timestamp** of the last successful update.

Polling can be toggled per page. Pages built around interactive or one-shot actions (consoles, diagnostics, JSON executors) do not poll by default and are refreshed manually. Pause polling when you want a stable view while you read a large table or while a trace is in progress.

Permissions & What You Can Change

What you can *do* on an element depends on your role. Access is granted **per element type, per HTTP method**:

Permission	Methods allowed	Typical use
Read-only	GET	View data, no changes
Read-write	GET, POST, PUT, PATCH	View and modify (add subscriber, edit route)
Full	GET, POST, PUT, PATCH, DELETE	Everything, including deletion / clearing

Permissions are assigned independently per element: you might have full access to the UPF but read-only on the HSS. If an action is not permitted for your role, its control is unavailable. Administrators have full access to all elements. See [Authentication & Access Control](#) to manage roles, and note that **every proxied request is audit-logged** regardless of role.

Multi-Instance & Fleet Views

Many element types run as multiple instances: several UPFs behind a PGW-C, a pair of SMSCs, redundant STPs. OmniWeb handles this at the element type level: a type with more than one configured instance gets two navigation modes.

Sidebar & Instance Selection

When a type has multiple instances, the sidebar entry expands to list each one by name. Click a named instance to open that instance's workspace directly. An instance dropdown also appears in the workspace header. It lets you switch between instances without going back to the sidebar, which is useful when you compare behaviour across nodes without losing your position on a particular tab.

The instance selector in the top-right of the workspace header. Opening it shows every configured instance plus "All Instances" to jump back to the fleet view, without leaving your current tab.

Fleet View

Click the element type name in the sidebar (rather than a specific instance), or select **All Instances** from the dropdown, to open the **fleet view** for that type. The fleet view shows a summary card per instance with health status and key metrics, so you can assess the whole fleet at a glance before you drill into a specific node.

UPF fleet view: one card per instance, all polled in parallel. Click any instance card or use the dropdown to drill in.

When to Use Each View

Situation	Use
Checking overall fleet health	Fleet view: all instances at a glance
Investigating a specific node	Instance dropdown or sidebar sub-item
Comparing two UPFs session counts	Open each in the instance dropdown in turn
Clearing a session on a specific UPF	Navigate to that instance, then Sessions

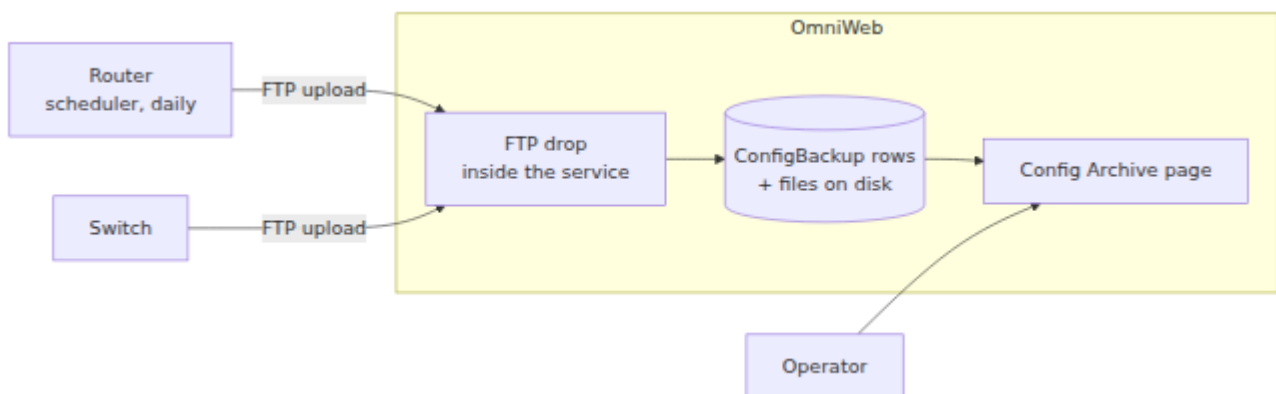
Config Archive: Device Configuration Backups

Config Archive keeps the configuration of every network device that uploads one, and makes the collection searchable, sortable and comparable. A router exports its own configuration on its own schedule and pushes it to OmniWeb; the page shows what arrived, what is inside it, and what changed since last time.

[← Operations Guide](#)

What it is

Routers and switches cannot run a log agent or a backup client, but they can all upload a file. So the archive turns that one capability into a record of every configuration the estate has run.



The drop runs **inside the OmniWeb service**. There is no sidecar process, no extra package, and nothing to install on the host: the listener is part of the application, written against the Python standard library.

Why FTP

RouterOS `/tool fetch` uploads a file over FTP or SFTP. It can also send an HTTP POST, but the body comes from a RouterOS variable, and a variable holds at

most 4 KB, far below a configuration export. FTP is therefore the only transport the devices and OmniWeb both speak for a file of this size.

FTP carries the password and the configuration in clear text. Run it on a management network, and restrict the source addresses at the firewall.

What the drop accepts

The listener implements the part of FTP a device needs to hand a file over, and refuses the rest.

Action	Result
STOR a file	Accepted, wherever the device puts it.
RETR, LIST, NLST, MLSD	Refused. The drop is write-only: nothing can be read back out of it over FTP.
DELE, RMD, RNFR, RNT0	Refused. A device can add a backup and can never remove or rename one.
CWD into a directory that does not exist	Accepted. The tree is virtual, and the path becomes the device name.
MKD	Accepted, and creates nothing. Directories are implied by the uploads.
A path with .. in it	Normalised away before anything touches the disk. The upload lands inside the archive.
An upload past the size limit	Refused with 552, and nothing is written.

Both file kinds are supported, and the archive treats them differently:

- **Text**, such as a RouterOS /export (.rsc), or any .txt, .conf or .cfg. The text is indexed, so it can be searched and compared.

- **Binary**, such as a RouterOS `/system backup`. The file is kept byte for byte for restoring, and is marked binary in the list.

Identity: the address, not the name

The source address is the identity. Everything keys off the address the bytes actually came from: which series an upload joins, what it is compared with, and whose retention counts it. This is TCP, so a sender cannot forge the address without holding it: the handshake has to complete.

The name is only a claim. RouterOS `dst-path=<identity>/<file>` gives the first path element, and a bare `edge-rtr09-20260910.rsc` gives `edge-rtr09`. Any sender can claim any name, so the archive stores it, shows it and searches it, and keys nothing off it.

That is what makes an open drop safe to run:

Someone uploads	What happens
From a router's own address	Joins that router's series, as its next backup.
From a different address, claiming a router's name	Starts its own series beside the real one. It cannot fake a change on the real router, and it cannot flood the real router's history until retention ages the genuine backups away.
From a new address	A new series, announced once in Slack, so an unexpected sender is visible rather than silent.

A source's label starts as whatever its uploads call themselves, which is convenient and untrusted. Pin a truthful name on the source to stop uploads moving it.

The cost of this model: a device that genuinely changes address starts a new series, so its history splits. Pin labels on both, or treat the old series as closed.

Configuration

There is nothing to configure. The drop runs on port 2121, open, as soon as OmniWeb starts. Point a device at it and it uploads.

That is deliberate, not a shortcut. A password is not what protects the archive: FTP sends it in clear, and the routers keep it in the very configurations they upload, so anyone who can read the archive or sniff the management network already has it. What protects the archive is that a sender only ever writes into the series for the address it holds, which TCP makes it hold for real. The firewall is the control that matters; restrict the port to your devices.

Set a password to demand one on top of the address check, or

`OMNIWEB_CONFIG_ARCHIVE_FTP=0` to keep the archive page with no listener at all. The rest of the table is there for the unusual site, and every value below is a sensible default that does not need setting.

Environment variable	Default	Description
OMNIWEB_CONFIG_ARCHIVE_FTP_PASSWORD	(empty)	Set on demar password top of address check. Empty normal to run.
OMNIWEB_CONFIG_ARCHIVE_FTP_OPEN	1	Accept upload without password. Set 0 password required.
OMNIWEB_CONFIG_ARCHIVE_FTP_USER	devbackup	User n the de auther with.
OMNIWEB_CONFIG_ARCHIVE_FTP_PORT	2121	Control Use 21 if the service bind a privileged port.
OMNIWEB_CONFIG_ARCHIVE_FTP_HOST	0.0.0.0	Address listen o

Environment variable	Default	Description
OMNIWEB_CONFIG_ARCHIVE_PASV_MIN	40000	First port for data port. Open the range device
OMNIWEB_CONFIG_ARCHIVE_PASV_MAX	40100	Last port for data port
OMNIWEB_CONFIG_ARCHIVE_ROOT	backend/data/config-archive	Where files are kept, config directory device
OMNIWEB_CONFIG_ARCHIVE_MAX_BYTES	33554432	An upload larger than this is refused
OMNIWEB_CONFIG_ARCHIVE_MAX_TEXT_BYTES	4194304	Text upload size indexed search diff. Large text is on display
OMNIWEB_CONFIG_ARCHIVE_RETENTION_DAYS	365	A backup past this age is removed
OMNIWEB_CONFIG_ARCHIVE_KEEP_MINIMUM	10	Newest backup

Environment variable	Default	Description
		device stay whatever their a
OMNIWEB_CONFIG_ARCHIVE_FTP	1	Set to keep the archive page but never the list
OMNIWEB_CONFIG_ARCHIVE_TIMEOUT	120	Seconds before idle connection or a stalled transfer dropped
OMNIWEB_CONFIG_ARCHIVE_RETRY_SECONDS	20	How often worker lost the tries again. This is how long the driver be unsuccessful after the worker holding recycle
OMNIWEB_CONFIG_ARCHIVE_SLACK_WEBHOOK	(empty)	Slack incoming webhook

Environment variable	Default	Description
		Empty means no notification.
OMNIWEB_PUBLIC_URL	(empty)	Public URL of OmniWeb used to send a Slack message back to the archive.
OMNIWEB_CONFIG_ARCHIVE_MAX_DIFF_BYTES	262144	Largest difference stored; larger changes are truncated from the row.
OMNIWEB_CONFIG_ARCHIVE_METRICS_TOKEN	(empty)	Bearer token for Prometheus must be present to scrape metrics. Empty leaves open.

OmniWeb runs several gunicorn workers, and every worker starts the drop. Exactly one wins the bind and serves; the others stand by and retry, so the listener survives a worker being recycled. The page reports `listening_here` per process, so a `false` on one request is not a fault.

Change tracking and Slack

Every upload is compared with the previous upload **from the same source**, of the same kind, as it arrives. Same kind means the same extension, so a device that sends a text export and a binary backup each night compares each against its own predecessor instead of alternating between them and calling everything a change. Identical bytes mean no change, whatever the file is. Different bytes on a text file produce a unified diff and a line count, both stored on the row, so the page and the notification never recompute them. A binary file that differs is recorded as changed with no diff, because there is nothing readable to show.

What does not count as a change

A device stamps its export with the time it ran. RouterOS opens every `/export` with

```
# sep/10/2026 08:25:00 by RouterOS 7.24.2
```

so two exports of an untouched router differ by that line and nothing else, and it writes the file under a new name each run. Compared raw, every device would report a change every night, post it to Slack, and bury the one signal worth having under a daily false positive per device.

So these lines are removed before anything is compared, and only for comparing: the stored file and the searchable text keep exactly what the device sent.

Ignored	Why
# <date> <time> by RouterOS <version>	The RouterOS export header. Changes on every run. RouterOS 7 writes an ISO date (2026-09-10 11:39:35), RouterOS 6 writes sep/10/2026; the pattern matches the shape they share rather than either rendering.
# software id = ...	Present in some RouterOS exports.
! Last configuration change at ...	Cisco IOS. Changes when anything is written, including by the device.
! NVRAM config last updated at ...	Cisco IOS.
ntp clock-period ...	Cisco IOS. Drifts on its own, with no operator involved.

The file name is not part of the comparison at all: an upload is matched against the previous upload from the same source with the same extension, so a name with a timestamp in it changes nothing.

Set OMNIWEB_CONFIG_ARCHIVE_VOLATILE_PATTERNS to a |||-separated list of regular expressions to replace this list for a vendor that stamps something else.

After changing the patterns, rehash the archive. Each backup stores a hash of its normalised configuration, so a new pattern leaves every stored hash stale: the next upload from each device would no longer match its own history, storing a duplicate file and reporting a change that never happened.

```
sudo -u omniweb /opt/omniweb/backend/venv/bin/python
/opt/omniweb/backend/manage.py \
    rehash_config_archive --dry-run      # report first
sudo -u omniweb /opt/omniweb/backend/venv/bin/python
/opt/omniweb/backend/manage.py \
    rehash_config_archive
```

That comparison drives two things:

- **The page.** Every row shows `+N/-N` when the configuration moved, or a tick when the device sent the same file again. **Changes only** narrows the list to uploads that actually changed something.
- **Slack.** A change posts to an incoming webhook, with the source, the line counts, the first 40 lines of the diff and a link back to the archive. An address heard from for the first time posts once as well, naming the address first and what it claims to be second.

An unchanged nightly backup posts nothing. An estate that backs up every night would otherwise fill a channel with "nothing happened", and everybody would mute it. What arrives instead is only what an operator would want to be interrupted for.

```
*core-rtr01 (10.64.10.1)* configuration changed (+3/-1)
`core-rtr01-2026-09-10-0215.rsc` received 2026-09-10 02:15
Previous backup: 2026-09-09 02:15
```

followed by the diff, and a link to the archive.

Set `OMNIWEB_CONFIG_ARCHIVE_SLACK_WEBHOOK` to a Slack incoming webhook to turn this on, and `OMNIWEB_PUBLIC_URL` to have the message link back to this OmniWeb. With no webhook set, the archive still tracks changes and the page still shows them; nothing is posted.

The POST is fire and forget in a background thread, so a slow or unreachable Slack can never stall an upload or lose a backup.

Check-in and metrics

Two facts, and monitoring needs both:

Fact	Field	What silence means
We heard from it	<code>last_upload_at</code> , bumped on every upload, changed or not	A fault. The device, its schedule or the path here has stopped working.
Its configuration moved	<code>last_change_at</code> , bumped only on a change	Nothing. A device whose configuration never changes is a healthy device.

Alert on the first and never the second, or a stable router pages you every night.

Counters (`upload_count`, `change_count`) live on the source rather than being counted from rows, so retention pruning old backups cannot make them fall. An archive that predates these fields is backfilled from the backups it already holds when the migration runs.

Scraping

`GET /api/config-archive/metrics` returns Prometheus text exposition. It is a plain endpoint with no dependency, and it does not use the session: set `OMNIWEB_CONFIG_ARCHIVE_METRICS_TOKEN` and Prometheus presents it as a bearer. Unset leaves it open, and the exposition names devices and addresses, so set it on anything reachable.


```
scrape_configs:
  - job_name: omniweb_config_archive
    metrics_path: /api/config-archive/metrics
    static_configs:
      - targets: ['<omniweb-host>']
    authorization:
      credentials: '<OMNIWEB_CONFIG_ARCHIVE_METRICS_TOKEN>'
```

Metric	Type	Description
<code>config_archive_last_upload_timestamp_seconds</code>	Gauge	Last upload from this source, changed or not. This is the one to alert on. 0 when it has never uploaded, so the same alert catches a device that never started.
<code>config_archive_last_change_timestamp_seconds</code>	Gauge	Last upload whose configuration differed. 0 when it has never changed.
<code>config_archive_uploads_total</code>	Counter	Uploads received since the source was first seen. Retention does not lower it.
<code>config_archive_changes_total</code>	Counter	Uploads that changed the

Metric	Type	Description
		configuration.
<code>config_archive_retained_backups</code>	Gauge	Backups kept after retention.
<code>config_archive_retained_bytes</code>	Gauge	Bytes kept after retention.
<code>config_archive_sources</code>	Gauge	Sources the archive has ever heard from.
<code>config_archive_ftp_enabled</code>	Gauge	Whether the drop is configured to run.
<code>config_archive_ftp_bound</code>	Gauge	Whether the drop socket is accepting connections on the host. Alert on this one. Any worker can answer it, because it is measured with a local connect.
<code>config_archive_ftp_listening_here</code>	Gauge	Whether the worker that

Metric	Type	Description
		answered this scrape is the one holding the socket. Debugging only: a scrape through nginx lands on an arbitrary worker of the pool, so this is a coin flip. Never alert on it.

Every series carries `source` (the address, the identity) and `label` (the display name, which is only what uploads claim until an operator pins it).

Alerts worth having

```
# A device stopped backing up, or never started. Daily schedule
plus slack.
```

```
(time() - config_archive_last_upload_timestamp_seconds) > 172800
```

```
# Nobody is serving the drop.
```

```
config_archive_ftp_bound == 0 and config_archive_ftp_enabled == 1
```

```
# A device that suddenly changes every run, which usually means
its export
```

```
# carries a timestamp rather than that anyone is editing it.
```

```
increase(config_archive_changes_total[1d]) > 3
```

Retention

Retention runs **per source**, which is the point: one sender filling its own series can never age out another sender's backups. A backup older than the retention window is removed, except that the newest few of each source always stay. A device that stops uploading therefore keeps its last known-good configuration for ever, and age alone never empties a device's history.

Uploads of the same **configuration** share one file on disk, compared the same way changes are: with the export's own timestamp and other volatile lines removed. This matters more than it sounds. A RouterOS export stamps the time it ran into its first line, so the raw bytes differ every single night; keying storage off them would write a fresh copy of an untouched router's configuration every night for ever.

A router that uploads an unchanged configuration every night therefore still gets a row every night, which is what makes "we heard from it" visible, while the bytes are stored once.

Configuring a MikroTik

Two commands, run on the router. Set `<omniweb-host>` to your OmniWeb address. No password is needed, and the credentials below are ignored: the drop identifies a device by the address it uploads from.

```
/system script add name=config-backup policy=read,write,ftp,test
source=":local id [/system identity get name]; :local d [/system
clock get date]; :local t [/system clock get time]; :local base
(\$id . \"-\" . \$d . \"-\" . [:pick \$t 0 2] . [:pick \$t 3 5]);
/export file=\$base; :delay 5s; /tool fetch address=<omniweb-host>
port=2121 mode=ftp user=omniweb password=x src-path=(\$base .
\".rsc\") dst-path=(\$id . \"/\" . \$base . \".rsc\") upload=yes;
:delay 2s; /file remove [find name=(\$base . \".rsc\")];"
```

```
/system scheduler add name=config-backup interval=1d start-
time=02:15:00 on-event="/system script run config-backup"
policy=read,write,ftp,test comment="Nightly config export to
OmniWeb Config Archive"
```

Run it once by hand to prove the path before trusting the schedule:

```
/system script run config-backup
```

A working run prints `status: finished` with the uploaded size, and the device appears in the archive within seconds.

Why it is written like that

Detail	Reason
The whole script is one line	The RouterOS CLI rejects backslash continuations when a command arrives over SSH. A script written across several lines pastes fine in a terminal and fails from automation, with a bare <code>syntax error (line 1 column N)</code> .
<code>policy=read,write,ftp,test</code>	<code>fetch</code> needs <code>ftp</code> and <code>test</code> . Without them the run fails with a policy error and nothing is uploaded.
<code>port=2121</code>	The drop's default. <code>/tool fetch</code> takes the port separately from the address.
<code>user=omniweb password=x</code>	Placeholders. The drop accepts any credentials, because the source address is the identity. <code>fetch</code> still wants the arguments present.
<code>[find name=...] in /file remove</code>	<code>/file remove</code> takes an internal id, not a name. Passing the name directly fails and the export is left on the router, filling its storage over time.
<code>:delay 5s after /export</code>	The export writes asynchronously. Uploading immediately can catch a partial file.
<code>\$d</code> in the file name	RouterOS 7 renders the date as <code>2026-09-10</code> , so the file sorts naturally. The name changes every run and that is fine: the archive matches an upload against the

Detail	Reason
	previous one from the same source, never by file name.

Stagger `start-time` across routers so they do not all upload at once.

To keep a binary backup as well, add `/system backup save name=$base` and upload `$base.backup` the same way. The text export is the one to diff; the binary is the one to restore onto identical hardware.

Any device that can upload over FTP works the same way: point it at the host and port and let it name its own file.

Using the page

Config Archive is in the sidebar, under Host List.

Control	What it does
Silent banner	Names every source that has not uploaded for 48 hours, with how long it has been. This is the check-in, so it fires for a device sending unchanged backups only if it stops sending them.
Source	Filter to one source address. Each entry shows when it last pushed, and marks a silent one.
Name, address or file	Substring search across the claimed name, the source label, the address and the file name.
Text inside the configuration	Search the configurations themselves, for example an IP address, <code>ospf</code> , or a user name. This is the fastest way to find which devices carry a setting.
Column headings	Sort by device, file, received time or size.
Changes only	Narrow the list to uploads that changed the configuration, hiding the unchanged nightly ones.
Change column	<code>+N/ -N</code> when the configuration moved, a tick when the device re-sent the same file.
A row	Opens the configuration, with a diff against that device's previous backup.
Download	The file exactly as the device sent it.
Add a file	Upload a configuration by hand, for a device that cannot push.

The diff is the part that earns its keep: it answers "what changed on this router, and when" without a device login.

API

All endpoints need an authenticated session.

Method and path	Description
GET /api/config-archive/backups	List, with <code>source</code> , <code>address</code> , <code>q</code> , <code>content</code> , <code>changed</code> , <code>ordering</code> , <code>limit</code> and <code>offset</code> .
POST /api/config-archive/backups	Add a backup by hand: multipart <code>file</code> , optional <code>device</code> .
GET /api/config-archive/backups/<id>	One backup, with its text.
GET /api/config-archive/backups/<id>/download	The stored file.
GET /api/config-archive/backups/<id>/diff	Unified diff against the previous backup, or <code>?against=<id></code> .
GET /api/config-archive/sources	One row per source address: backup count, changes, newest upload, size, and every name its uploads have claimed.
PATCH /api/config-archive/sources/<id>/label	Pin a truthful name to a source, so uploads stop moving it.
GET /api/config-archive/status	Drop state and archive totals.
GET /api/config-archive/metrics	Prometheus text exposition. See Check-in and metrics .

`changed=1` returns the changes feed: uploads that differ from the device's previous backup.

`ordering` accepts `received_at`, `claimed_name`, `filename` and `size_bytes`, with a leading `-` to reverse. Any other value falls back to newest first.

Troubleshooting

Symptom	Check
The page warns that the drop is off	<code>OMNIWEB_CONFIG_ARCHIVE_FTP_PASSWORD</code> is unset. Set it and restart OmniWeb.
The device reports a login failure	Confirm the user name and password, then look for <code>config archive ftp: login failed</code> in the OmniWeb log.
The transfer starts and stalls	The passive range is not reachable. Open <code>OMNIWEB_CONFIG_ARCHIVE_PASV_MIN</code> to <code>..._MAX</code> from the device.
Uploads arrive with an odd claimed name	The device sent a bare file name that does not start with its identity. Set <code>dst-path</code> to <code><identity>/<file></code> , or pin a label on the source.
One device appears as two sources	Its address changed, so it started a new series. Pin labels on both to make the relationship obvious.
An unexpected address appears	Somebody or something else can reach the port. Check the Slack "now backing up" message for when it started, and restrict the port.
A device is missing from the list	It has never completed an upload. Run the script by hand on the device and watch its log.

Symptom	Check
Nothing arrives in Slack	Confirm <code>OMNIWEB_CONFIG_ARCHIVE_SLACK_WEBHOOK</code> is set, then check whether the uploads are changes: an unchanged backup is silent by design. Failures are logged as <code>Slack post failed</code> .
Slack posts on every backup	The device stamps something into every export that is not covered by the ignore list. Compare two backups to see what moves, then add a pattern to <code>OMNIWEB_CONFIG_ARCHIVE_VOLATILE_PATTERNS</code> .

DRA: Diameter Routing Agent

The **DRA** is the Diameter routing hub of the core. Rather than every node holding a full mesh of Diameter connections, the EPC and IMS functions connect to the DRA and it relays and routes their signalling: **S6a** between the MME and HSS, **Gx/Gy** to the PCF/PCRF and OCS, **Cx/Dx** and **Sh** for the IMS, and so on. It chooses where each message goes from **routing rules** (matching on Diameter application, realm or AVP) and a peer-selection algorithm, and it can rewrite messages with **transform rules**. It is the Diameter counterpart of an SS7 STP. OmniWeb surfaces the DRA's peers, routing rules, live status and a connection map. Throughput and trend charts are on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

The DRA is a per-instance element page reached from the sidebar when one runs in the site's inventory, with a fleet view (the network map) where several are deployed. All traffic goes through the OmniWeb backend proxy to the DRA's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Network Map

A hub-and-spoke view of the DRA and every Diameter peer it knows, with the connection count at a glance (e.g. *5/6 connected*). Connected peers are linked to the DRA in green. An administratively **disabled** or **disconnected** peer is shown greyed and dashed. It is the fastest way to see, in one picture, which signalling neighbours are up.

Status

The **Status** tab reports the DRA's own identity and configuration: its **Origin-Host**, **realm**, the **listen IP and port** (Diameter is usually 3868), the reported **product name**, and the **peer-selection algorithm** in use (e.g. round-robin). This is where you confirm the DRA is running and configured as expected.

Peers

Every Diameter peer, each row showing the peer's **Origin-Host, IP and port**, reported **product, transport** (SCTP or TCP), **realm**, the negotiated **Diameter applications**, and its **connection state**. Peers can be filtered by name, application, realm or state. Each peer has an **enable/disable** toggle. Disabling a peer administratively takes it out of routing without tearing down its configuration, which is how you drain a neighbour for maintenance.

Rules

The active **routing rules** and **transform rules**, and the **routing algorithm** they run under. Each routing rule matches on a filter (a Diameter application, an AVP, or all traffic) and resolves to a destination: either an explicit set of peers or a named strategy such as *destination-host* or *destination-realm*. This is where you confirm, for example, that S6a is routed to the HSS and Gx to the PCF.

Logs

The **Logs** tab tails the DRA's live log output to troubleshoot a single routing agent without leaving OmniWeb.

Monitoring dashboards

Message rates, per-interface throughput and other time-series for the DRA are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- **MME** and the **HSS**: the DRA relays **S6a** between them.
- The **PCF/PCRF (Gx)** and **OCS (Gy)**: policy and charging peers routed through the DRA.
- The **CSCF** and **HSS**: **Cx/Dx** IMS signalling.
- **STP**: the SS7 routing counterpart to the DRA in the signalling plane.

Entitlement Server: OmniSEP

The **Entitlement Server (OmniSEP)** decides which services a subscriber and their device are allowed to use. When a handset asks whether it may turn on **VoWiFi**, **VoLTE** or **Wi-Fi Calling**, or runs **ODSA** device provisioning to add or activate a device, it checks in with the entitlement server. The server authorises the request over the **entitlement (TS.43)** interface, resolves subscriber data over **Diameter**, and applies the subscriber's **XCAP** service profiles. OmniWeb surfaces its live entitlements, activity, sessions, tokens, Diameter peers and XCAP profiles on one element page; volume and trend charts live on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

OmniSEP is a per-instance element page reached from the sidebar when one runs in the site's inventory. All traffic goes through the OmniWeb backend proxy to the entitlement server's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

The landing tab is a single-screen health read for the entitlement server: its **system status**, the count of active **sessions** and **tokens**, subscribers carrying custom **entitlements**, and the health of its **Diameter** peers. It is the fastest way to confirm the server is up and authorising devices.

Entitlements

The subscribers that carry **custom entitlements**: overrides to the default service set, per subscriber. Open a subscriber to see which services are enabled for them. An operator can **delete all entitlements** for a subscriber to return them to the network default. This is useful for clearing a bad override during a fault.

Activity

The recent **activity** records: a feed of the entitlement requests the server has handled, each showing the subscriber or device, the service requested and the outcome. Open a record for its detail. This is the audit trail for confirming whether a handset actually asked for VoWiFi/VoLTE and what answer it was given.

Sessions

The active **EAP-AKA sessions** the server is holding for device authentication, each with its subscriber identity and state. An individual session can be **expired** from here to force a device to re-authenticate. This is the quickest way to clear a session that has stuck.

Tokens

The active **authentication tokens** the server has issued to devices after they authenticated. Each token can be **revoked** individually, which withdraws a device's authorisation and requires it to authenticate afresh. This is the entitlement equivalent of forcing a re-login.

Diameter

The **Diameter peers** the entitlement server connects to for subscriber data, with each peer's **connection state**. When entitlement decisions start failing for reasons that look like missing subscriber data, this tab is where you confirm the Diameter path is up.

XCAP Profiles

The **XCAP profiles**: the per-subscriber service settings (such as supplementary-service configuration) the entitlement server applies. Open a profile to read it. A profile can be **deleted** to clear a subscriber's stored settings and fall back to the default.

Logs

The **Logs** tab tails the entitlement server's live log output for troubleshooting a single server without leaving OmniWeb.

Monitoring dashboards

Request rates, per-service counters and other time-series for the entitlement server are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- The **HSS** holds the subscriber data the entitlement server's Diameter requests resolve against.
- The **CSCF** and **TAS** deliver the VoLTE/VoWiFi services the entitlement server authorises.
- The **ePDG** and **TWAG** carry the Wi-Fi Calling access the entitlement server enables.
- The **DRA** relays the entitlement server's Diameter signalling to its peers.

ePDG: Evolved Packet Data Gateway

The **ePDG** is the secure gateway for **untrusted Wi-Fi access**: the node behind Wi-Fi Calling (VoWiFi) and untrusted-WLAN data. A handset builds an IPsec/IKEv2 tunnel to the ePDG over any internet connection; the ePDG authenticates the subscriber against the AAA/HSS over Diameter (SWm/SWx), anchors the session to the packet core (S2b to the PGW), and enforces per-subscriber rate limits and geo-IP policy. OmniWeb surfaces the ePDG's live tunnels, its Diameter peers, rate-limiting and geo-IP configuration, and runtime settings; longer-run trends are on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

The ePDG is a per-instance element page reached from the sidebar when one runs in the site's inventory. All traffic goes through the OmniWeb backend proxy, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

A single-screen health read: the ePDG's **status**, the number of **active sessions** (IPsec tunnels), and a summary of the gateway's operational state. This is the quickest confirmation that the ePDG is up and carrying Wi-Fi-calling tunnels.

Sessions

Every active session: the IPsec/IKEv2 tunnels currently established, keyed by subscriber, with the tunnel and bearer state. You can inspect a session to see its detail. This is where you confirm a specific subscriber's Wi-Fi-calling tunnel is up.

Diameter

The Diameter peer connections to the AAA/HSS that carry authentication (SWm/SWx), with each peer's connection state. This is the first place to look when Wi-Fi-calling authentications are failing.

Rate Limiter

The ePDG terminates untrusted-Wi-Fi IPsec/IKEv2 tunnels (SWu) and bridges them to the EPC over S2b, which makes it exposed to any handset (or any source) on the open internet. The **Rate Limiter** is the defence against a client hammering tunnel setup: it throttles IKE flooding so a single misbehaving handset or attacking source cannot exhaust the gateway. The tab shows the enforced **thresholds** alongside a live count of how many **IPs** and **IMSI**s are **currently blocked**. The thresholds are the **per-IP** and **per-IMSI** request limits, each expressed as a number of auth attempts within a rolling **window**, and the **block duration** applied once a limit is breached.

Below the thresholds you can **look up** any IP address or IMSI to see whether it is blocked and how many failures it has accumulated, and **clear** that block so future auth attempts from it are allowed again; a single **Clear All** action drops every block and resets all failure counters at once. This is where an operator confirms why a subscriber's Wi-Fi-calling tunnel is being refused, and releases a block once the offending source has settled.

GeoIP

GeoIP is the ePDG's country-based admission control: how an operator restricts VoWiFi tunnel establishment to permitted countries by matching each incoming tunnel's source IP against a GeoIP database. The tab shows whether filtering is **enabled** and whether the database is **ready** (loaded and usable), the **mode**, and the **country list** the mode applies to. The mode is either **whitelist** (only the listed countries may connect) or **blacklist** (the listed countries are refused). It also shows the **GeoIP database in use** and whether **unknown-country IPs** are allowed or blocked. These are source addresses the database cannot resolve to a country.

Once the GeoIP database file has been updated on disk, the tab can **reload** it so the new mapping takes effect without disturbing established tunnels. This is the control that keeps VoWiFi tunnel setup confined to the regions an operator serves.

Configuration

The ePDG's runtime **configuration**, **rate-limiting** rules (per-subscriber / per-source limits that protect the gateway), and **geo-IP** policy (which source regions are permitted to establish tunnels). These are the controls that shape who may connect and how aggressively.

Logs

The **Logs** tab tails the ePDG's live log output for troubleshooting a single gateway without leaving OmniWeb.

Monitoring dashboards

Tunnel rates, throughput and other time-series for the ePDG are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- The **PGW** anchors the ePDG's user plane over **S2b**.
- The **AAA/HSS** holds the subscriber authentication data the ePDG's Diameter requests resolve against.
- **TWAG**: the *trusted*-Wi-Fi counterpart.

5G Core

The **5G Core** section gives each 5GC network function (NF) its own live dashboard. Each dashboard shows health, registration, and the operational data that NF owns (UE contexts, PDU sessions, IP pools, NRF registrations, policies, charging sessions). It also gives per-NF and per-item actions to clear state. It is the 5GC equivalent of the per-element pages for EPC/IMS functions. It reads each NF's OAM API through the OmniWeb backend proxy.

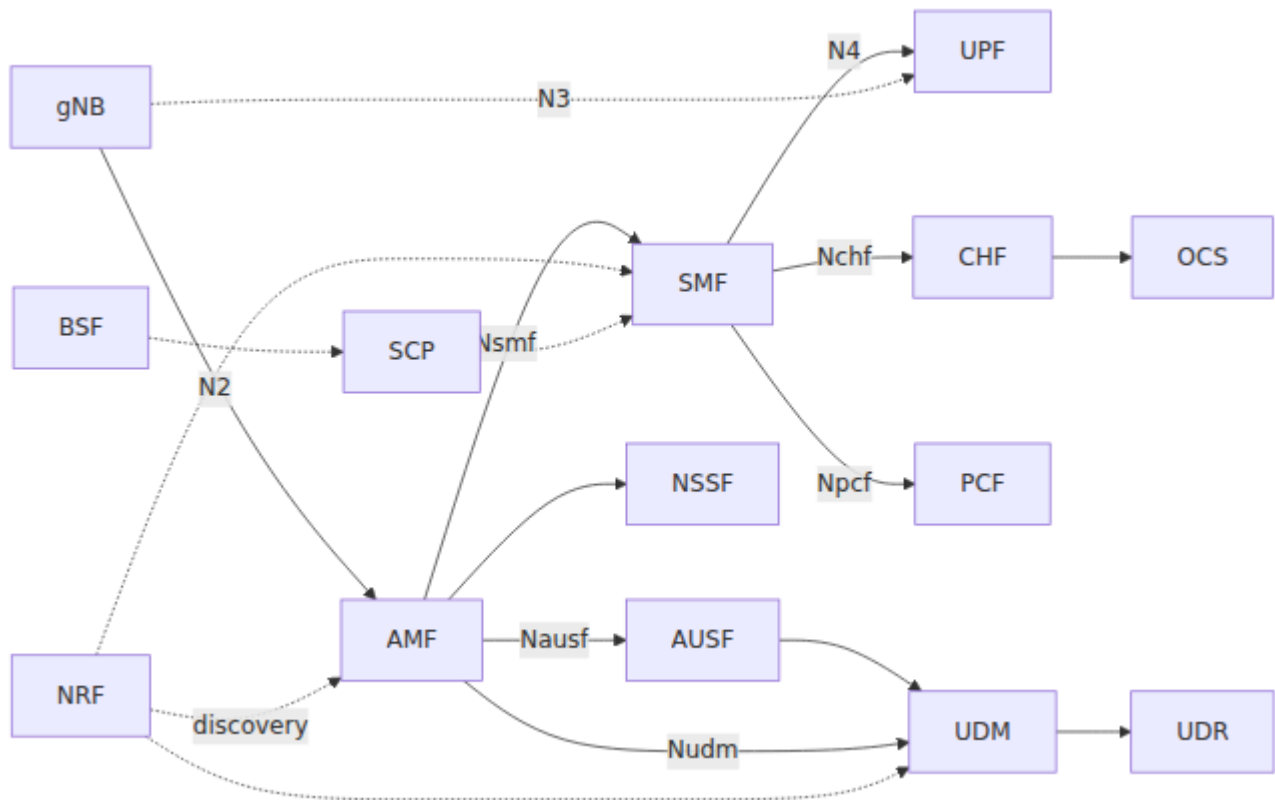
[← Operations Guide](#)

When it appears

The **5G Core** group only shows in the sidebar when the selected site actually runs a 5GC control plane. OmniWeb keys this off the presence of an **AMF and/or SMF** in the site's inventory. A lone UPF or CHF shared with the EPC is not a 5GC and does not show this section.

OmniWeb derives which NFs appear, and the address of each NF's OAM API, **directly from the site's Ansible inventory**: the 5GC NF groups defined in the host file. To add or move an NF is therefore an inventory change with nothing to hand-template. Each NF resolves to a label, description, IP, and port. OmniWeb serves the resolved registry via `/api/5gc/nf-registry`. The dashboards then query each NF's OAM API through the proxy (`/api/5gc/nf/<id>/...`), so all traffic stays behind the single authenticated gateway. See [the configuration model](#). Deployments that run no inventory (for example the hosted service) fall back to a static `fivegc` block in the OmniWeb configuration.

The NFs



Each NF has its own dashboard:

NF	Function	Key data surfaced
NRF	Network Repository Function	Registered NF instances (grouped by type), registered services, subscriptions: see below
AMF	Access & Mobility Management	UE list, connected gNodeBs (with decoded gNB-ID / TAC / PLMN / SST), statistics, maintenance mode
SMF	Session Management	PDU sessions, IP Pools , UPF association, statistics
UDM	Unified Data Management	UE list, statistics
UDR	Unified Data Repository	HSS health, subscribers
AUSF	Authentication Server	AUSF (auth) sessions, statistics
PCF	Policy Control	AM policies, SM policies, statistics
CHF	Charging Function	Charging sessions, CGrateS health, CDR records
NSSF	Network Slice Selection	Slices, NSSAI availability, statistics
BSF	Binding Support	PCF bindings, statistics
SCP	Service Communication Proxy	NF health, discovery cache, statistics

Not every deployment runs every NF. Only the NFs defined in the site's inventory appear. Each NF has its own reference page. Follow the links above.

The SMF's IP Pools tab

As an example of the NF-owned data these dashboards surface: the SMF dashboard has an **IP Pools** tab showing the address pools the SMF allocates UE IPs from, alongside its live **PDU Sessions** and its **UPF Association** (the N4/PFCP link state to the user plane). Together these answer "is the SMF connected to its UPF, what sessions are up, and is it running out of addresses?" without leaving OmniWeb.

A dashboard at a glance

Every NF dashboard follows the same layout:

1. **Health cards** across the top: API Health, NRF Registration (is this NF registered with the NRF?), NF Status, and License. Each polls on an interval. Use the **Polling Indicator** in the header to pause or see the last-update time (polling is a shared pattern: see [Common Operations](#)).
2. **Actions**: NF-level operations (see below).
3. **Data sections**: the API status detail, NRF registration detail, and the NF-specific tables from the table above, each auto-refreshing. Tables auto-decode what they can: PLMN/TAC/SST from Erlang-encoded S1AP terms, timestamps to relative time, statuses to coloured chips, and cross-links (an NF-type value links to that NF's dashboard).

Actions

Dashboards expose actions to clear NF state. Each asks for confirmation first, since they are destructive:

NF	NF-level actions	Per-row actions
AMF	Clear all UE contexts · Clear cache	Deregister a UE
SMF	Release all PDU sessions · Clear cache · PFCP reassociate (re-establish the UPF association)	Release one PDU session
NRF	Deregister all NFs	Deregister one NF · Remove a subscription
AUSF	Clear all auth sessions	Clear one session
UDM	Clear subscriber cache	-
PCF	-	Remove an AM or SM policy

These let an operator recover from stuck state. For example, force an SMF that lost its UPF to reassociate, drop a UE context that will not clear, or deregister a stale NF from the NRF.

The NRF dashboard

The NRF gets a richer view because it is the registry the whole 5GC discovers through:

- **Metric cards:** NF types, total instances, registered (with a percent-of-total bar), and services/subscriptions.
- **Registered NF instances:** grouped by NF type. The dashboard colour-codes each instance card. The card shows registration status, heartbeat timer, service count, and PLMN. Click an instance for its full profile (with an **Open Dashboard** link to that NF and a **Deregister** button).
- **Registered services:** a flat table of every service each instance advertises, its versions, and status.

- **Deregister All NFs**: clears every registration (each NF must then re-register).

This makes the NRF the place to answer "which NFs are up, are they all registered, and what services are they offering?"

Related

- **Network Topology**: the topology views that link into these dashboards.
- **Common Operations**: the polling, configuration, and RBAC patterns shared with every element page.
- **PGW-C / SMF** and **UPF**: the EPC-side session/user-plane pages for a converged EPC+5GC.

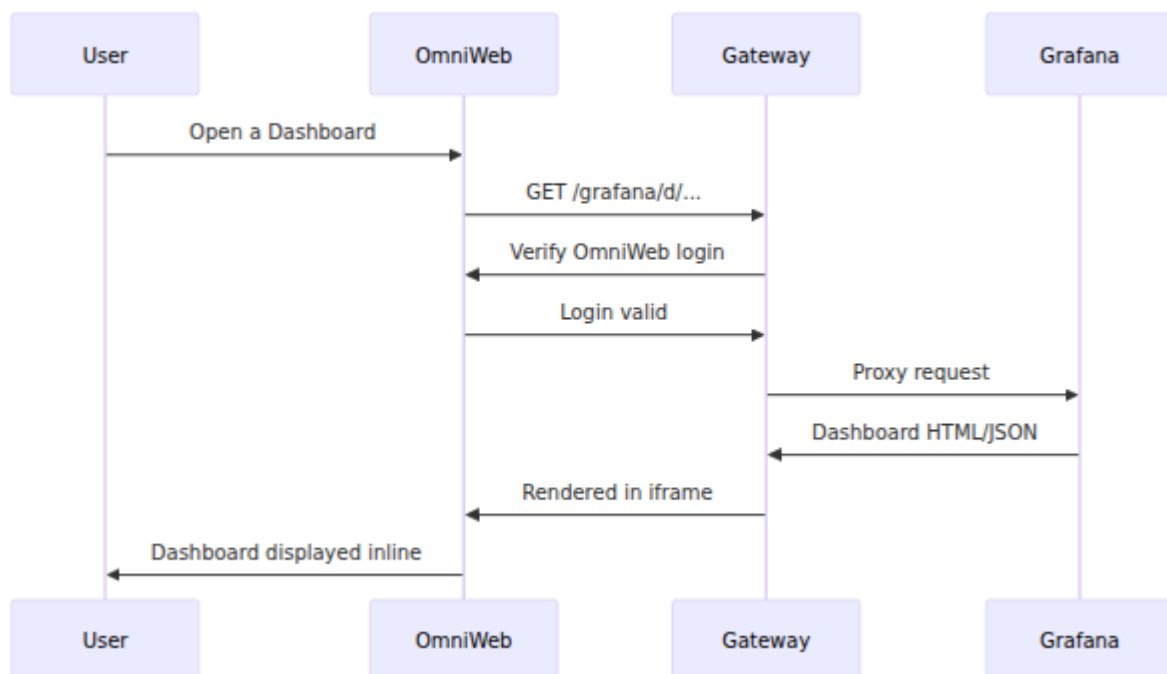
Grafana Dashboards & Statistics

OmniWeb embeds Grafana directly into the portal. Operators can see dashboards and statistics for any element, and for the network as a whole, without a separate login or browser tab.

[← Back to Operations Guide](#)

How It Works

Grafana is served behind the authenticated gateway at the `/grafana/` path and embedded as an iframe inside OmniWeb. Authentication is transparent: when you are logged into OmniWeb, Grafana access is authorised automatically against your OmniWeb login. There is no second password.



The Grafana instance OmniWeb proxies to is configured by the `GRAFANA_URL` environment variable (default `http://localhost:3000`). The gateway handles proxying and the single sign-on check transparently.

Two Ways to Reach Dashboards

1. Per-element Dashboards tab

Elements that are tagged for Grafana show a **Dashboards** tab in their workspace. OmniWeb queries Grafana for dashboards matching that element's tags and renders the relevant one inline. This keeps an element's stats next to its operational data.

Tag matching is case-insensitive. Representative tag sets:

Element	Grafana tags matched
UPF	upf
P-CSCF	pcscf, cscf
I-CSCF	icscf, cscf
S-CSCF	scscf, cscf
PGW-C, MME, SMSC, DRA, OCS, TAS	element-specific tags

To make a dashboard appear under an element, tag it in Grafana with that element's tag (for example, tag a UPF dashboard `upf`).

2. Grafana sidebar

The sidebar's **Grafana → Dashboards** entry lists all available dashboards. Selecting one opens it inline within OmniWeb, and the URL updates to reflect the dashboard path so you can deep-link to a specific dashboard.

Statistics

The embedded dashboards are where network-wide and per-element **statistics** live: session counts and rates, throughput, latency, drop rates, Diameter/PFCP

peer health over time, CDR volumes, registration counts, and capacity trends. OmniWeb's own element pages give you the *current* state (and let you act on it); Grafana gives you the *historical* picture and trends. They are designed to be used together. Investigate a Grafana alert by jumping to the element's operational pages, logs, and traces.

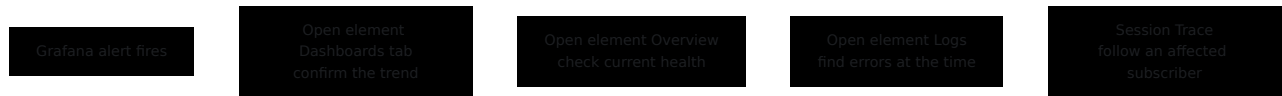
Alerting

Grafana's built-in alerting engine handles all threshold-based alerts. Alert rules are defined in Grafana and can notify via email, Slack, PagerDuty, or webhook. OmniWeb does not duplicate alerting. It provides the operational context (element pages, logs, traces) needed to investigate when an alert fires.

Common alert categories:

Category	Examples
Availability	Element unreachable, Diameter peer down, PFCP association lost
Capacity	IP pool exhaustion, eBPF map utilisation > 80%, VLR subscriber count threshold
Performance	Session setup latency, XDP drop rate, heartbeat failures
Traffic	Abnormal CDR volume, call drop rate, SMS delivery failures

Investigating an Alert: Recommended Flow



1. Confirm the trend in Grafana (Dashboards).
2. Check current state on the element's Overview.

3. Read the element's Logs around the alert time.
4. If subscriber-specific, run a Session Trace. See [Logs & Subscriber Tracing](#).

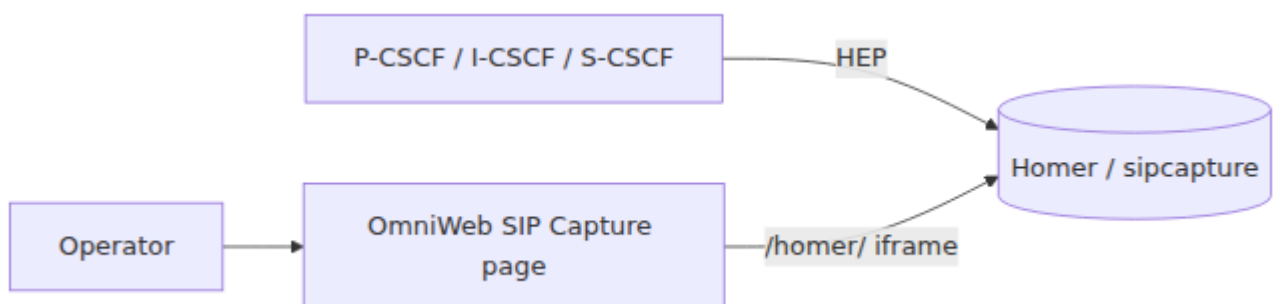
Homer: SIP Capture & Call-Flow Analysis

Homer is a SIP capture and search system (HEP / sipcapture). OmniWeb embeds the Homer web UI directly, so you can search captured SIP traffic and reconstruct a call's signalling ladder without leaving the portal or logging in separately.

← [Operations Guide](#)

What it is

Homer collects SIP messages from the IMS (via the HEP protocol) and stores them searchable by call-id, from/to number, time, and more. Its web UI lets you find a call and draw its full **flow diagram**: every SIP message between the endpoints and the CSCFs, with timings and any failure/hangup cause. OmniWeb does not reimplement any of this. It **embeds the Homer app** in an iframe under **SIP Capture** (nested under Session Trace in the sidebar).



Single sign-on

You do not log into Homer separately. OmniWeb proxies the Homer app at `/homer/` and injects the auth-proxy headers on the way through, so the embedded UI is **already authenticated**, with no Homer login prompt. It is reached through the same authenticated OmniWeb gateway as Grafana and Loki, so one OmniWeb login governs it too.

The **SIP Capture** entry only appears in the sidebar when Homer is **configured and reachable** for the deployment. OmniWeb probes it and hides the entry if it is not present, so you never click into a dead page.

Homer vs Session Trace

OmniWeb has two complementary ways to look at signalling, and it helps to know which to reach for:

	Homer (SIP Capture)	Session Trace
Scope	SIP only, in depth	The whole network: Diameter, GTP-C, PFCP, SIP, MAP
Question it answers	"What did this <i>call</i> 's SIP flow do, message by message, with timing and hangup cause?"	"What happened to this <i>subscriber</i> across every element?"
Best for	Diagnosing a VoIP/VoLTE call, a failed registration, a specific hangup	Following one IMSI/MSISDN end to end through the core

Reach for Homer when you have a call to dissect; reach for Session Trace when you are following a subscriber across the whole stack. Homer is also linked from the [Logs & Subscriber Tracing](#) guide.

Related

- [Logs & Subscriber Tracing](#): cross-network Session Trace and where Homer fits.
- [Grafana Dashboards & Statistics](#): the other embedded monitoring tool, reached the same authenticated way.

HSS: Home Subscriber Server

The **HSS** (Home Subscriber Server) is the master subscriber database for the mobile core. It holds every subscriber's identities (IMSI, MSISDN and SIM/ICCID), the authentication key material that proves who they are, and the EPC, IMS, PCRF and roaming service profiles that state what each subscriber can do on the network. The MME reads it over S6a to authenticate and locate a device and to learn its EPC subscription. The CSCF reads it over Cx for IMS registration and iFC service triggers. The PCRF draws its policy and charging rules from it. An operator uses its dashboard to provision and search subscribers, curate the shared service profiles, and check the HSS's own health.

← Operations Guide

The dashboard is a per-instance element page. It opens from the sidebar when an HSS runs in the site's inventory. All traffic goes through the OmniWeb backend proxy to the HSS's API, so it stays behind the single authenticated gateway. Responses come back in a `{status, response}` envelope. A documentation link in the page header gives in-context help. The page has tabs.

Subscribers

The landing tab is the subscriber directory. Search **by IMSI, MSISDN or ICCID** to find a record. The search also resolves a subscriber **by IP address** when you trace a live session back to its owner. Open a record to show the subscriber overview: a relationship diagram that ties the subscriber to its **identity** (IMSI), its **status** (enabled, and whether IMS is enabled), and the **EPC, IMS** and **roaming** profiles assigned to it. The page shows the profile's APNs beneath the EPC profile. The HSS holds sensitive key values but does not display them. You can update them, but they read back as redacted.

Provisioning

Create or edit a subscriber from the provisioning form. The form brings the identity, credentials, SIM and profiles together in one place:

- **Subscriber identity:** IMSI, a subscriber name, and the **enabled** and **IMS-enabled** switches.
- **Authentication key set:** the authentication algorithm, **Ki** and **OPc** (or **OP**), **AMF**, **SQN**, and the SIP password used for IMS.
- **SIM card:** the **ICCID**, whether it is an eSIM, the PIN/PUK values and the batch and vendor details.
- **Phone numbers:** assign one or more **MSISDNs**. Search existing numbers as you type so you do not create a number twice.
- **Assign profiles:** attach the **EPC**, **IMS** and **roaming** profiles that give the subscriber its service.

You can update existing subscribers the same way. You can delete a subscriber from its record.

SIMs

The SIM inventory holds every **SIM** the HSS knows. Search it **by ICCID**. Each SIM carries its **ICCID**, eSIM flag, batch name and vendor, its transport-key values (**KIC**, **KID**) and eSIM **LPA**, and the full set of **ADM keys**. You can create, edit and delete SIMs. The authentication **key sets** the SIM records pair with are managed on their own tab.

MSISDNs

The MSISDN directory holds the pool of provisioned phone numbers. Search **by digits** for a partial match to find or check a number before you assign it. You can create, edit and delete numbers. You attach numbers to subscribers during provisioning.

Key Sets

The key set directory is the authentication credential store every subscriber record points at. Each entry holds the **algorithm** (Milenage or Digest-MD5), the key material, the **AMF** and the **SQN** sequence counter, and the optional SIM OTA counter. Filter the page **by ID, algorithm or AMF**. You can create, edit and delete key sets.

Key material is write-only. **Ki**, **OPc**, **OP** and the SIP password are never shown — they read back as redacted, and an edit that leaves one blank keeps the stored value rather than clearing it. Deleting a key set that a subscriber still references stops that subscriber authenticating, so check before you remove one.

Service profiles

Service profiles are the reusable building blocks that state what a subscriber can do. A subscriber references them rather than carry its own copy. A change to a profile therefore applies to everyone who uses it.

EPC profiles

The packet-core subscription the MME enforces. An **EPC profile** sets the **network access mode**, the UE-AMBR **downlink/uplink** aggregate bandwidth, the **TAU interval**, and the list of **APN profiles** the subscriber can use. One APN is the **default APN**. Each **APN profile** ties an **APN identifier** to an **APN QoS profile** and a **PCRF profile**. You can mark an APN profile as the default. An EPC profile must reference at least one APN profile. The default APN must be one of the assigned APNs.

Expand an EPC profile to list its APN profiles. **Add APN** attaches an APN to the profile. The unlink action on each row **removes it from this EPC profile only**. The HSS keeps the APN profile itself. You can re-add it later, or other EPC profiles can reuse it. A profile must keep at least one APN, so you cannot remove the last one. The HSS clears the default APN automatically if you remove the APN it pointed at.

APN Profiles

The APN Profiles tab lists every APN profile independently of the EPC profiles that use it. It shows its **APN identifier**, **APN QoS profile** and **PCRF profile**. The **Used By** column names each EPC profile that references the APN. It marks the APN **Orphaned** when no EPC profile references it. An APN reaches this state once you remove it from the last EPC profile that used it. The **Orphaned only** toggle filters the list to just those, so you can find and clean up unused APNs.

You can create profiles here, edit them, and delete them. A new profile starts unattached. To link it to a subscriber's service, add it to an EPC profile. You cannot delete an APN profile that is still linked to one or more EPC profiles. First remove it from those profiles, then delete it from this tab.

APN QoS Profiles

The APN QoS Profiles tab manages the reusable APN-level QoS templates that EPC and APN profiles point at. Every subscriber whose APN references a template gets the same quality of service on its default bearer. Each template

carries a **name** and a **QCI** (the QoS Class Identifier that maps the bearer to a standardised delay and loss behaviour). It also carries an **ARP** (the allocation/retention priority), with **pre-emption capability** and **pre-emption vulnerability** flags. These flags decide whether the bearer can bump, or be bumped by, others when resources are scarce. Each template also carries the **APN-AMBR** uplink and downlink aggregate rates, and whether **online** and **offline charging** are enabled.

You can create, edit, and delete templates. You can also clone a template to derive a new variant. An APN profile references a QoS template rather than carry its own copy. If you edit a template, you re-shape the QoS of every APN that uses it. You cannot delete a template that an APN profile still references. First reassign those APNs.

APN QoS profiles

The per-APN quality-of-service settings an APN profile points at: the **QCI**, **ARP** and pre-emption **capability/vulnerability**, the APN-AMBR **downlink/uplink** bandwidth, and whether **online (Gy)** and **offline (Rf)** charging apply. You can clone a profile to derive a new variant from an existing one.

IMS profiles

The IMS subscription the CSCF reads over Cx. An **IMS profile** carries an **iFC template**: the initial-Filter-Criteria that decide which application servers receive a subscriber's SIP traffic. You can create, edit and delete profiles.

Static IPs

Fixed IP allocations for subscribers that must always receive the same address, managed alongside the EPC subscription data.

PCRF

The policy layer the PCRF applies. A relationship view links the three parts:

- **PCRF profiles:** the named policy a subscriber's APN references, made up of a set of charging rules.
- **Charging rules:** each with a **rule type** and **precedence**, a **QCI** and **ARP**, MBR/GBR **downlink/uplink** limits, and a **rating group** and **service identifier** for charging. A dedicated bearer carries a rule with an MBR or GBR.
- **Flow-information rules:** the traffic flow templates attached to a charging rule that define which packets it matches.

You can create, edit and delete all three. If you edit a charging rule, you affect every profile that uses it.

Roaming

The roaming policy the HSS applies to inbound and outbound subscribers. A **roaming profile** sets a **default data action**, a **default IMS action**, and a list of **roaming rules**. Each rule keys off an **MCC/MNC** and sets the **data** and **IMS action** for that network. You can therefore allow or block specific visited networks while the profile default covers the rest. To add a rule, use the operator lookup to find a network's MCC/MNC from an operator name (from IR.21). Rules are shared. If you edit a rule, you affect every profile that uses it.

EIR

The Equipment Identity Register. Each **EIR rule** matches an **IMEI** by regular expression and sets an **action**: allow or block. You can therefore bar a device model or a specific handset from the network. A device-model search helps build the IMEI pattern. You can create, edit and delete rules.

Status

A single-screen health read of the HSS itself: the **API**, **database** and **license** health, and the **entity counts** for total **subscribers**, **EPC profiles**, **IMS profiles**, **SIMs**, **MSISDNs** and **roaming profiles**. It is the quickest confirmation that the HSS is up and how much it carries. Grafana embeds provide the monitoring and statistics dashboards.

The **Replication** tab shows the status of database replication to any peer HSS instances. The **Logs** tab tails the HSS's live log output. Use it to troubleshoot a single instance without leaving OmniWeb.

Replication

The Replication tab reads the PostgreSQL replication health of the HSS node's subscriber database and draws it as a live topology. Each node shows whether its database is a **primary** or a **standby**. A standby is a node that is in recovery. It replays the primary's changes rather than accept writes of its own. For a primary, the tab draws every connected standby as a downstream link. The link shows the standby's **client address**, the **replication slot** it streams through, its **streaming state**, and the **replication lag** measured against the primary's current WAL position. The tab shows the lag in bytes, so a replica that falls behind stands out at a glance. Whether each replication slot is currently **active** determines whether its link reads as healthy or broken.

An operator uses this tab to confirm that the subscriber database replicates across the HSS cluster: that a change provisioned on one node will reach the others, and that a standby exists to fail over to. A lagging replica shows a growing lag figure. A broken or disconnected replica drops its link. If two or more HSS instances run but none of them replicate, the tab flags the whole cluster as a replication gap. Correct this before it becomes a data-loss risk.

Operations

Two subscriber-level operations act on a live session rather than on the stored record:

- **Cancel-Location (CLR)**: send an S6a Cancel-Location-Request to a subscriber's serving MME. This detaches the subscriber, so it re-attaches and picks up changed subscription data.
- **PCRF Re-Auth**: request that the PCRF re-authorise a subscriber for a given PDN session. This pushes updated policy or charging rules onto an active session.

Related

- **MME**: reads subscriber authentication and EPC subscription data over **S6a**. It is the target of a Cancel-Location.
- **CSCF**: reads the IMS profile and iFC over **Cx** for IMS registration and service triggering.
- **PCRF / PCF**: applies the PCRF profiles, charging rules and flow rules held here. It is the target of a Re-Auth.
- **UDR**: the shared subscriber data store the HSS front-ends in a 5G core.
- [Operations Guide](#): back to the element index.

LCS: Location Services (OmniLCS)

The **LCS** function provides **location services** for the core. It answers "where is this subscriber?" and drives cell-broadcast warnings. It resolves a UE's position from the serving cell (and configured cell geometry), services location requests from the network (immediate and deferred), and pushes **Commercial Mobile Alerts / cell-broadcast** warnings to an area. OmniWeb surfaces the LCS's cell inventory, live and recent location fixes, deferred-location requests, CAP alerts and broadcasts; longer-run trends are on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

The LCS is a per-instance element page reached from the sidebar when one runs in the site's inventory. All traffic goes through the OmniWeb backend proxy, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

A single-screen health read: the LCS **status**, **active** and **completed** location sessions, the number of **cells loaded**, and the **cell-sync** result. This is the quickest confirmation the LCS is up and has current cell geometry.

The overview also shows every connection the LCS holds to the rest of the core, each with its state (**Connected** or **Disconnected**), address and, for SCTP associations, the association id and the time it came up:

- **SBc-AP peers:** the 4G **cell-broadcast** association to each MME. A 4G warning broadcast can only reach the cells of an MME whose SBc-AP association is connected, so this is the first thing to check when a broadcast does not reach handsets.
- **SLs peers:** the **LCS-AP** association to each MME, which carries positioning (E-CID, OTDOA, A-GNSS) between the LCS and the serving MME.
- **Diameter peers:** the SLg/SLh Diameter connections (for example to the DRA), with the peer's product and application ids.
- **CAP feeds:** each connection to an alert source (see [CAP Feeds](#)), with the time of its last heartbeat or poll.

The tiles at the top count the connected peers of each kind against the configured total, turning red when any is down.

Cells

The **cell inventory** the LCS positions against: each cell's id, latitude/longitude, radius, height, TAC/LAC, PCI, EARFCN, azimuth and RAT. You can add, edit and remove cells here. This geometry is what turns a serving-cell id into a position.

Adding or editing a cell

1. Pick the **radio** (2G, 3G, 4G or 5G). The dialog shows the fields that radio uses: LAC for 2G/3G, TAC for 4G/5G, and the matching channel number (ARFCN, UARFCN, EARFCN or NR-ARFCN) and physical cell identity.
2. Enter the **Cell ID** exactly as the network reports it. For 4G, **Build ECI from eNB ID** works out the ECI from the eNB ID and cell number.
3. Enter the **TAC** or **LAC**. Cell broadcasts for this cell go to this area code.
4. Place the cell: click the map, drag the marker, or type or paste the latitude and longitude (`lat, lon` pasted into either box fills both). Set the coverage **radius** and, for a sectorized antenna, the **azimuth**; the map shows the resulting coverage.

Each value is checked against its valid range before the cell can be saved. A CAP alert reaches a cell when the cell's position is inside the alert area, so a cell with no position, or with the wrong TAC or LAC, receives no alerts.

Location

Recent **location fixes**: each fix's subscriber, resolving cell, computed position, accuracy and method. A location can be **requested** for a subscriber from here, and opening a fix shows its authoritative last-known position. This is where an operator confirms a positioning result.

Deferred

The **outstanding deferred-location requests** the LCS is holding. Where the Location tab answers "where is this subscriber *now*", a **deferred MT-LR** (deferred mobile-terminated location request) asks the network to report a subscriber's position **later, when a trigger fires**, rather than immediately: the GMLC arms the request and the report comes back when the condition is met. The common triggers are **UE availability** (report the next time the device becomes reachable, having been idle or unreachable when the request was made), **periodic** location (report at a fixed **interval** for a set **number of reports**), and area/motion events (report when the UE enters, leaves or moves within a designated area). This is how an operator schedules positioning for a subscriber who cannot be found at the moment or who needs to be tracked over time.

Each row lists the deferred session's subscriber by **IMSI**, its **type** (periodic or triggered), the positioning **method**, its schedule (the reporting interval for a periodic session; the area event and poll interval for a triggered one), the **reports left** out of the total, its **status**, the requesting client, when it **started** and the time of the **last fix**. A triggered session with no report limit shows *infinity* reports left.

An operator can **create** a new deferred request from here, naming the target IMSI, the type and the positioning method (cell, E-CID, GNSS or OTDOA):

- **Periodic**: the reporting interval in seconds and the number of reports to send.
- **Triggered**: the area event (**entering**, **leaving** or **being inside**), the area itself (a circle given by its centre and radius in metres, or a polygon of at least three latitude/longitude points), the maximum number of reports (0 for unlimited) and how often the LCS polls the subscriber's position.

An operator can also **cancel** an outstanding session at any time, which tears down the armed request so no further reports are produced. The list polls so a session's status updates as the network arms it and the trigger fires.

CAP Alerts

The **Common Alerting Protocol (CAP)** alerts the LCS has accepted for cell broadcast, in three groups:

- **Pending approval:** alerts waiting for an operator. **Approve** broadcasts the alert to the cells inside its warning area; **Reject** drops it.
- **Active:** alerts being broadcast. **Stop Broadcast** takes an alert off the air on every network it was sent on.
- **History:** alerts that were sent, rejected, **cancelled** (by the issuing authority or an operator) or **expired**.

Each alert shows where it came from (a CAP feed or a manual submission), the CAP identifier and sender, and one line per **language**: the language, the cell-broadcast **message identifier** it goes out on and the text sent to handsets. When an alert has been sent, each line also shows the serial number and the per-network result. The footer gives the network code, the target TACs/LACs, how many cells matched the warning area, and when the alert was received, sent, approved, expires and ended.

Alerts arrive automatically from the **CAP feeds**, or an operator can **Submit Alert** with CAP XML. Not every CAP message is for handsets. The LCS applies the alerting rules below, and a submission that does not qualify is reported rather than queued:

- only **Actual** alerts with **Public** scope are broadcast. Test messages that are not meant for broadcast are ignored;
- for Canadian (CAP-CP) alerts, only information blocks flagged for wireless distribution are broadcast, using their wireless text;
- each language goes out as its own broadcast: the primary language on the configured message identifier (4370 by default) and every other language on the matching additional-language identifier (4383);
- a **Cancel** or **Update** from the issuing authority stops the earlier alert's broadcast, and an alert stops broadcasting when it **expires**;
- the same alert (same sender and identifier) is only processed once.

CAP Feeds

The **CAP feeds** the LCS receives alerts from. Two kinds are supported:

- **NAAD stream (Canada)**: the Pelmorex National Alert Aggregation and Dissemination (NAAD) System. The LCS connects to every streaming site at once (the two sites give redundancy) and expects a heartbeat from each every minute; a site that falls silent is reconnected. Each heartbeat lists the latest alerts, and any that did not arrive on the stream are fetched from the NAAD archive, so an alert issued while a connection was down is still received.
- **CAP Atom feed**: a CAP feed in Atom format, polled at a set interval.

Each feed shows whether it is enabled, its approval setting and, for every site or URL, the **connection state**, the time of the last heartbeat (or poll) and counters: alerts and heartbeats received, alerts **submitted** for broadcast, **ignored** (not for handsets), **cancelled**, **duplicates** (the same alert from the other site) and alerts **backfilled** from the archive.

Add Feed creates a feed. The **Canada — Pelmorex NAAD** button fills in both NAAD streaming sites and both archive addresses. For each feed you set its

name, type, URLs, whether it is enabled and its **approval** behaviour: use the LCS-wide setting, require an operator to approve each alert, or broadcast without approval. Feeds added here can be edited, enabled or disabled, and deleted. Feeds defined in the LCS configuration are marked **From config** and are read-only here.

Broadcasts

The **Broadcasts** tab shows the **cell-broadcast** messages the LCS is pushing, one row per network: the message identifier and its name (test and exercise identifiers are marked **test**), the serial number, the network (2G over CBSP, 3G over SABP, 4G over SBc-AP), the message text, its status and when it was sent. The **History** view lists past broadcasts, including their warning type.

The serial number is shown in hex with its **message code** and **update number**; hover it for the geographical scope. A handset shows a message only once for each message identifier and serial number. A row where the same identifier and serial number were sent more than once is marked **duplicate - handsets ignore**: handsets that received the first send did not display the later ones.

An operator can **stop** an active broadcast from its row; the stop applies to that row's network only, so a message sent on both 3G and 4G is stopped on each separately.

Send Broadcast starts a new one:

- **Message ID:** pick from the 3GPP list (ETWS earthquake/tsunami, CMAS/WEA presidential, extreme, severe, AMBER, monthly test, exercise, operator defined, public safety, state/local test) or enter a custom value. For ETWS identifiers, choose the ETWS **warning type**.
- **Languages:** the message text, plus any number of **additional languages**. Each additional language is sent on the additional-language identifier that matches the chosen one (for example 4370 → 4383), with the same serial number, and the result of each language is reported. ETWS and custom identifiers have no additional-language identifier. Under each text a counter shows the encoding (GSM 7-bit, or UCS-2 when the text has characters such as some accented letters) and the pages used, up to 15.
- **Serial number:** built from the **geographical scope**, a **message code** (0–1023) and an **update number** (0–15). A new alert gets the next message code not yet used for that identifier. **Send as update of...** picks an earlier broadcast and keeps its code with the update number increased by one, so

handsets replace the earlier message. A warning appears when the identifier and serial number were already sent, because handsets would ignore the message.

- **MCC and MNC** of the network being broadcast to (always required, so a broadcast never goes out under the wrong network code), the networks to send on, and the target **TACs** (4G), **LACs** (2G/3G) and **SACs** (3G) for the ticked networks.
- How often and how many times the message repeats (0 repeats until stopped).

Logs

The **Logs** tab tails the LCS's live log output for troubleshooting without leaving OmniWeb.

Monitoring dashboards

Location-request rates and other time-series are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- The **MME** requests positioning over the location interfaces and is the LCS's Diameter peer.
- **CAP alerts** and **broadcasts** feed the cell-broadcast warning system.

License Server: Omnitouch Licensing

The **License Server** is the Omnitouch licensing authority the network functions check against. Every Omnitouch product (the MME, HSS, DRA, PGW-C, UPF, SGW-C, SMSC, CSCF and the rest) validates its entitlement against the License Server before it runs, asking whether a given **licensee** is licensed for that **product** and whether the licence is still within its validity window. The server holds cryptographically **signed licences**, each granting a licensee a set of products with a validity period and metadata, and it answers entitlement queries and records every request it is asked. Because a licence is only meaningful against a trustworthy clock, the server also tracks a **trusted time** source so a licence cannot be revived by rolling a node's clock back. OmniWeb surfaces what is licensed and its health, the log of entitlement requests network functions have made, an on-demand query tool, and the server's live logs.

[← Operations Guide](#)

The License Server is a per-instance element page reached from the sidebar when one runs in the site's inventory, with a fleet view where several are deployed. All traffic goes through the OmniWeb backend proxy to the License Server's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview / Status

A single-screen read of what the server is licensing and whether it is healthy. Summary cards give the count of **total products**, how many are **valid** and **invalid**, and the number of distinct **licensees**. A **trusted time** panel shows the state of the server's clock source: **valid**, **grace period** or **invalid**. When trusted time is unavailable the server cannot safely validate licences at all, and that raises the page's health to a warning or error.

The **Licensed Products** table lists one row per licensed product, each showing the **product**, the **licensee** it is granted to, when it was **issued**, its **start** and **expiry** dates (with a "days remaining" or "expired N days ago" indicator that turns amber within 30 days of expiry and red once past it), and its **valid/invalid** status. Opening a row expands the full detail: the exact validity timestamps, whether the **cryptographic signature** is valid, any product metadata, and the signature itself. This is the quickest confirmation that the network's functions are all licensed and that nothing is about to lapse.

Request Logs

The audit trail of **entitlement requests** the network functions have made against the server. Each row records the **timestamp**, the **source IP** the request came from (which network-function host asked), the **licensee** and **product** it asked about, the **decision** (**allowed** or **rejected**), and, for a rejection, the **reason**. The list can be filtered by decision, by product, and by a partial **source IP** or **licensee** search, and it is paged. This is where you see, for example, that a newly deployed function is being rejected because its product was never licensed, or confirm that a function is successfully checking in.

License Query

An on-demand **entitlement check**. Enter a **licensee** and a **product** and the tool asks the server the same question a network function would (is this licensee entitled to this product right now) and returns the verdict: **valid** or **invalid**, the licence's **valid-to** date, the response **timestamp**, any **metadata**, and the **signature**. It is the direct way to confirm a specific entitlement without waiting for the owning function to check in, and it lists the known product identifiers (for example `omnihss`, `omniupf`, `omnimme`) so you can query the exact one.

Logs

The **Logs** tab tails the License Server's live log output for troubleshooting a single instance without leaving OmniWeb.

Related

- **Every Omnitouch network function:** the MME, HSS, DRA, PGW-C, UPF, SGW-C, SMSC, CSCF and the rest validate their entitlement here; each product's own **Overview / Status** tab shows a licence-health badge derived from this check.
- [Operations Guide](#): back to the element index.

Logs & Subscriber Tracing

OmniWeb gives operators several complementary, real-time views into what the network is doing:

1. **Logs**: structured, live service logs for any single element, powered by Loki.
2. **Session Trace**: a cross-network view that follows one subscriber across every protocol and element (Diameter, GTP-C, PFCP, SIP, MAP).
3. **Homer**: deep SIP/VoIP packet capture and call-flow analysis.

[← Back to Operations Guide](#)

Jump to a Subscriber from an NF

Many network functions show a subscriber, UE, or session by IMSI. These include the **MME** (UEs), **SGW-C**, **PGW-C** and **ePDG** (Sessions), **MSC** and **TAS** (Subscribers), and **SMSC** (Location). In each case a row of quick-link buttons lets you pivot straight to that subscriber elsewhere, without copying the IMSI around:

- **Go to HSS Record** opens the subscriber's provisioning record on the HSS (resolved by IMSI across the configured HSS instances).
- **View Session Trace** opens the subscriber's live signalling ladder in Session Trace, scoped to that IMSI.
- **View Logs** opens *this* element's Logs tab pre-filtered to that IMSI, so you see only the log lines for that subscriber.

Each button only appears when it applies: the IMSI is known, and an HSS or Session Trace is configured for the site. The set therefore adapts to the deployment.

The subscriber quick-links (here on an MME UE): one click to the HSS record, the live signalling ladder, or this element's logs filtered to the subscriber.

Real-Time Logs (Loki)

Every element has a **Logs** tab. It queries the element's service logs from **Loki** and streams new lines as they arrive, so you can watch an element live while you make a change or reproduce an issue.

The Logs tab (here on a UPF): structured, colour-coded log lines streamed from Loki, with service, severity, protocol, and subscriber (IMSI) filters. Every element exposes the same viewer.

How log queries work

The frontend builds a **LogQL** query and sends it through the backend's Loki gateway. Nothing queries Loki directly from the browser. It is proxied (and therefore authenticated and audited) like every other request.



The backend's Loki target is set by the `LOKI_URL` environment variable (default `http://localhost:3100`). Queries use Loki's range-query API with a LogQL expression, a time window, a result limit, and a direction (newest-first or oldest-first).

Selectors vs. pipeline filters

OmniWeb builds structured queries from two kinds of filter:

Filter placement	LogQL position	Example	Use
Selector	Inside the stream selector { ... }	<code>{hostname="omni-eval-upf01"}</code>	Pick which streams (which host/service) to read. Indexed and fast.
Pipeline	After a <code> </code>	<code> level="error"</code>	Filter the matched lines by a structured field.

A typical element Logs tab pins the host/service as a selector and lets you add pipeline filters (severity, component, subscriber identifiers) on top. Because Omnitouch services emit **structured** log lines, you can filter on real fields rather than grepping free text.

Structured logging data

Hosts ship structured logs to Loki (configured by the Ansible common role). Two especially useful structured sources:

- **Service logs:** each NF's own structured output, labelled by `hostname/service`, available on that element's Logs tab.
- **Command audit:** every command executed on a host, including who logged in (surviving `sudo/su`), the effective UID, the binary, the working directory, and the full command line. Shipped to Loki. See [Audit Logging](#) for the field reference.

Using the Logs tab

1. Open any element and select **Logs**.
2. The viewer loads recent lines for that element and begins polling for new ones.
3. Narrow the view with pipeline filters (e.g. severity, or an IMSI/MSISDN if the service logs include it).

4. Adjust the time window to look back at a past event.
5. Pause polling (header indicator) to hold a stable view while reading.

Tip: When investigating one subscriber, filtering element logs by their IMSI/MSISDN is the fast first step; to see the subscriber across *multiple* elements at once, use Session Trace below.

Session Trace: Follow a Subscriber Across the Network

The **Session Trace** engine correlates signalling for a single subscriber across multiple elements and protocols, giving one unified timeline from radio attach through authentication, session establishment, and call/SMS delivery. This is the tool for answering "what happened to this subscriber?" without hopping between element CLIs.

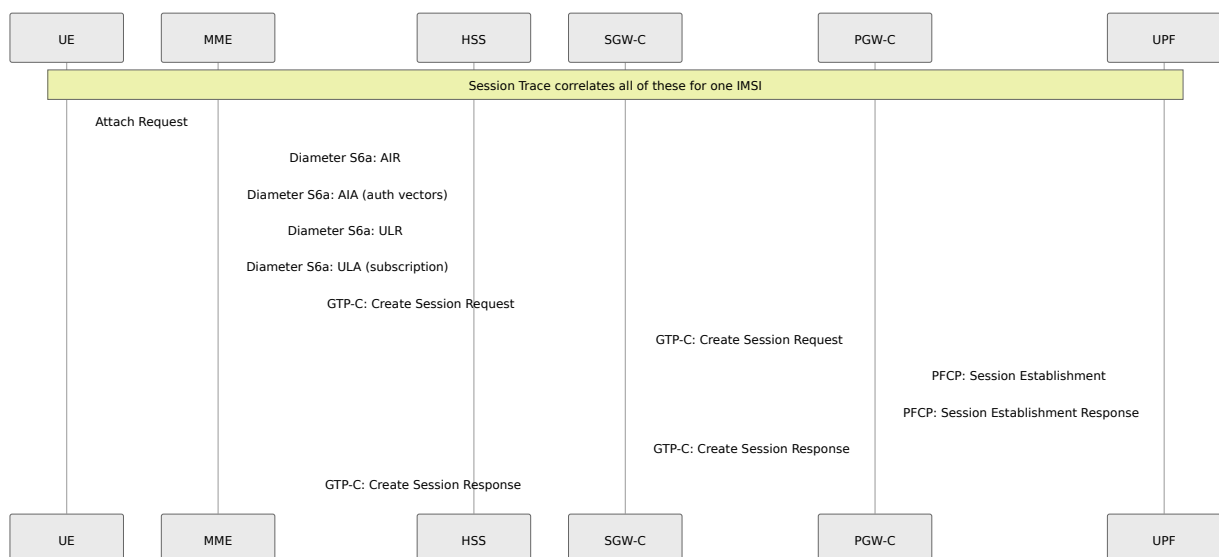
Session Trace: search by IMSI or MSISDN and correlate a subscriber's activity across Diameter, GTP-C, PFCP, SIP, and MAP, with per-protocol tabs across the top.

Supported protocols

Protocol	What it shows	Elements involved
Diameter	Authentication, policy, charging	HSS, DRA, PGW-C, PCRF, OCS, CSCF (Cx/Dx/Rx)
GTP-C	Session management (S5/S8, S11)	SGW-C, PGW-C, MME
PCFP	User-plane session rules (N4)	PGW-C/SMF ↔ UPF
SIP	IMS call control	P-CSCF, I-CSCF, S-CSCF, TAS
MAP	SS7 location, authentication, SMS	MSC, HLR, CAMEL GW, IP-SM-GW

User Trace

The **User Trace** view lets you search by **IMSI** or **MSISDN** and see every correlated protocol interaction for that subscriber across the network. A single attach, for example, surfaces the Diameter authentication, the GTP-C session creation, and the PCFP rule installation as one coherent sequence:



Protocol-specific tabs

Each protocol has its own tab (**Diameter**, **GTP-C**, **PFCP**, **SIP**, **MAP**), plus an **All Protocols** overview. The per-protocol tabs are filtered views. They help when you chase a protocol-specific problem (e.g. only the SIP exchange of a VoLTE call) rather than following a single subscriber across everything.

PCAP Ladder

The **PCAP Ladder** tab turns a packet capture into a ladder: a time-ordered sequence diagram of the signalling. Upload a PCAP or PCAPNG capture (for example from a network tap or an element-level trace) and OmniWeb parses the packets, identifies the signalling nodes by IP address and inferred role (MSC/VLR, HSS, S-CSCF, and so on), and correlates requests with their responses into sessions. The result is drawn as a message ladder across those nodes, ordered in time, so you can read a whole exchange top to bottom at a glance.

PCAP Ladder: an uploaded capture rendered as a ladder across the detected nodes. Click any message to see its decoded frame and hex.

Click a message on the ladder to open its full decode and raw hex, the same detail you would get from a protocol analyser. The tool recognises the signalling families you meet across a mobile core: MAP, CAMEL, TCAP, Diameter, SIP, GTP-C, PFCP, and GSM SMS. Every capture you upload is listed on the tab so it can be re-opened or shared with a colleague rather than passing the file around.

Live Flows

The **Live Flows** tab shows the same ladder view, but sourced live from the network's structured signalling logs (via Loki) rather than an uploaded file. As the network elements emit signalling telemetry, recent per-subscriber call flows appear here automatically, grouped by IMSI, with no capture to upload. Select a flow to reconstruct it into the same ladder as a PCAP, so a live flow is as rich as one read from a file, down to per-message decode and hex.

Live Flows: recent call flows drawn straight from the network's signalling telemetry. Filter by IMSI, node, or protocol, then open a flow to render its live ladder.

Open a flow and OmniWeb reconstructs it into the same ladder as a PCAP, but from the network's own telemetry, with nothing to capture or upload. The

example below is a single subscriber's LTE attach, correlated across every element that touched it: the S1AP/NAS exchange with the **eNodeB**, the Diameter S6a authentication and location update against the **HSS**, and the GTP-C session setup through the **SGW-C** and **PGW-C**. All of this appears on one time-ordered ladder, with per-message request/response round-trip times and full decode a click away.

A live flow opened into its ladder: one subscriber's LTE attach across the eNodeB, MME, HSS, SGW-C and PGW-C, correlating S1AP, Diameter S6a and GTP-C. Click any message for its decode and hex, or any node to focus its exchanges.

If no signalling telemetry is flowing yet, the tab shows an empty state; flows populate on their own as the network elements start emitting telemetry.

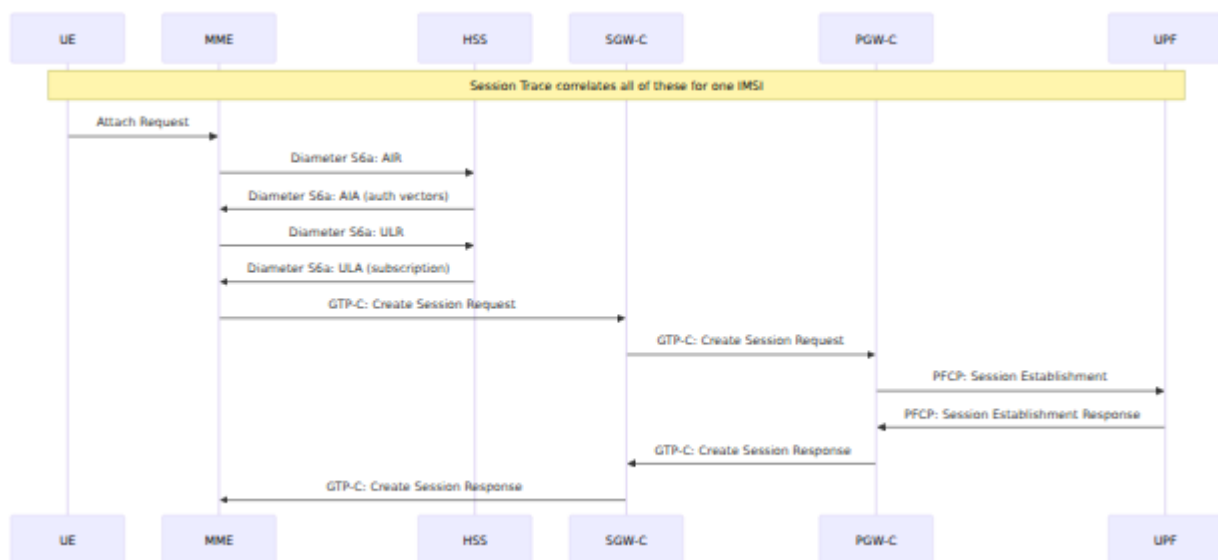
Homer: SIP Capture & Call-Flow Analysis

OmniWeb integrates with **Homer**, the open-source SIP capture and VoIP troubleshooting platform, for deep call-flow visualisation, protocol decoding, and search across captured SIP signalling. Where Session Trace gives a cross-protocol *summary* for one subscriber, Homer gives the full, message-by-message SIP packet detail.

Availability & access

Homer is **optional** and auto-detected. On startup OmniWeb checks `/api/homer/status`:

- If Homer is configured (`HOMER_URL` set on the backend) and reachable, a **Homer** entry appears in the sidebar.
- If Homer is not deployed, the entry is simply hidden, and nothing else changes.



Clicking **Homer** opens the Homer web UI in a new browser tab. Authentication is transparent: the gateway verifies your OmniWeb login via the same single sign-on used for Grafana, so there is no second login. The backend's `HOMER_URL` (and `HOMER_PROXY_SECRET`, when set) configure the integration.

Use cases

Homer is the tool to reach for when a problem is in the SIP layer itself:

- **VoLTE call-flow analysis:** trace SIP INVITE/BYE sequences through P-CSCF, I-CSCF, S-CSCF, and TAS.
- **VoWiFi troubleshooting:** analyse ePDG ↔ P-CSCF SIP interactions.
- **IMS registration debugging:** inspect REGISTER / 401 / REGISTER / 200 OK exchanges.
- **Interconnect issues:** capture SIP-trunk signalling between the MSC and SIP peers.

Session Trace vs. Homer

Both help with troubleshooting, but they answer different questions. They are complementary, not alternatives:

Tool	Best for	Scope	Where it opens
Session Trace	"Follow this subscriber across the whole network"	Diameter, GTP-C, PFCP, SIP, MAP correlated by IMSI/MSISDN	Inline in OmniWeb
Homer	"Show me the exact SIP messages for this call"	SIP/VoIP signalling, full packet decode (VoLTE/VoWiFi/IMS)	New tab (auto-detected)

A common flow is to spot a SIP-related anomaly in Session Trace, then jump to Homer for the detailed call flow.

Troubleshooting

No logs appear on an element's Logs tab

Symptoms: The Logs tab is empty or shows "no results".

Possible causes:

- Loki is not reachable from the backend (`LOKI_URL` misconfigured).
- The host is not shipping logs to Loki (log shipping not running / misconfigured).
- The selected time window predates the available logs, or the host label does not match.

Resolution:

1. Confirm `LOKI_URL` points to a reachable Loki instance.
2. Verify the element's host is shipping logs (check the log-shipping agent on that host).
3. Widen the time window and remove pipeline filters to confirm any lines arrive.

Session Trace returns nothing for a subscriber

Symptoms: A User Trace by IMSI/MSISDN shows no events.

Possible causes:

- The subscriber was inactive in the selected window.
- The identifier is mistyped (IMSI vs. MSISDN).
- Trace sources for the relevant protocol are not feeding the engine.

Resolution:

1. Re-check the identifier and try the alternate one (IMSI ↔ MSISDN).
2. Widen the time window.

3. Confirm with the element's own Logs tab that the subscriber was active at all.

The Homer entry is missing from the sidebar

Symptoms: No **Homer** link appears.

Possible causes:

- Homer is not deployed, or `HOMER_URL` is not set on the backend.
- Homer is unreachable, so `/api/homer/status` reports it as unavailable.

Resolution:

1. Confirm Homer is deployed and `HOMER_URL` is set on the OmniWeb backend.
2. Verify the backend can reach Homer, then reload OmniWeb (detection runs at startup).

MME: Mobility Management Entity

The **MME** is the control-plane anchor of the EPC. It terminates **S1AP** to the eNodeBs, runs the attach, authentication and mobility procedures for every UE, and holds a **UE context** for each attached subscriber. It fetches subscription and authentication data from the HSS over **S6a** (Diameter), sets up bearers on the Serving Gateway over **S11**, and coordinates inter-MME and inter-system handovers over **S10/N26**. In short, it is the node that decides where and how each subscriber attaches, is paged and moves through the network. OmniWeb surfaces the MME's live eNodeBs, UE contexts, SGW and Diameter peers, interface states and handover status. Longer-run trends are on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

The MME is a per-instance element page reached from the sidebar when one runs in the site's inventory, and, where several are deployed, a fleet view for aggregate status. All traffic goes through the OmniWeb backend proxy to the MME's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

A single-screen health read: the **API status** of the MME and a summary of its live load, including connected **eNodeBs**, attached **UEs**, **SGW peers** and the state of its **Diameter** (S6a) peers. It is the quickest way to confirm the MME is up and see how much of the access network it is serving.

eNodeBs

Every eNodeB the MME has an S1AP association with, each row showing the **global eNB ID**, name and connection **state**. Open an eNodeB to show its context: the served cells and TACs and the UEs on it. An operator can **update the MME capacity** advertised to an eNodeB, which adjusts how the eNodeB load-balances UEs across the MME pool (used to drain or weight an MME for maintenance or rebalancing).

UEs

Every attached subscriber's **UE context**, each row keyed by **IMSI** and **GUTI** with its connection and mobility state and serving eNodeB. Opening a UE shows its full context. A range of per-UE actions is available:

- **Page** the UE.
- **Send to idle**: release the radio connection while keeping the context.
- **Offload** the UE to another MME.
- **Request re-attach**: force the UE to re-attach.
- **Send SMS** to the UE.
- **Delete** the UE context: clear a stale or stuck subscriber.

You can also **look up the operator / PLMN** for an IMSI to identify home versus roaming subscribers.

S-GW Peers

The Serving Gateways the MME talks to over **S11** (GTP-C), each row showing the peer's **IP**, its **liveness / reachability**, the GTP **restart (recovery) counter** and the **session count** anchored on that gateway. The counter increments when the S-GW restarts, which tells the MME to re-establish the sessions it was holding. Whether a peer was learned by **DNS** or is a **static** entry is shown alongside the S-GW **selection method** the MME applies for home and roaming subscribers. Open a peer to list the **sessions** it carries, each identified by **IMSI** with its **S11 TEIDs** (the MME- and S-GW-side tunnel endpoints). It is where an operator confirms which S-GWs are up and sees how the UE sessions are distributed across them. This helps when you balance load or diagnose why bearers for a group of subscribers went down with a single gateway.

GUTI reallocation and S10 offload

Two mobility actions are driven by IMSI rather than per row:

- **GUTI reallocation**: reallocate a connected subscriber's **GUTI** to refresh the temporary identity the UE presents (used when rotating identifiers or clearing an identity mismatch).
- **S10 offload**: force-offload a subscriber to **another MME**. This moves a specific subscriber's context off this MME over S10, and complements the eNodeB-capacity control when you drain an MME.

S10

The **S10/N26 inter-MME** interface is the path over which a UE's context is relocated when it moves between MMEs in a pool during a **TAU** or **handover** (and, over **N26**, between the MME and an AMF for EPC-to-5GC interworking). The tab has two parts. **Active handovers** lists the relocations in flight, each showing the **source** and **target** MME and the **IMSI** being moved, so an operator can watch a mobility event proceed or spot one that is stuck. **Peer MME discovery** shows how the MME finds its pool peers: the **DNS / NAPTR** lookups it runs and any **static** peers configured, with each peer's FQDN and address. It is where you confirm inter-MME mobility is healthy and that the MME can actually reach the neighbours it needs to hand contexts to. It complements the per-subscriber **S10 offload** action driven from the UEs tab.

Diameter

The MME's Diameter peers, principally the HSS over **S6a**, each row showing the peer **host**, **realm** and connection **state**. Open a peer to show its detail. This is where you confirm the authentication and subscription path to the HSS is up when attaches are failing.

Interfaces

The **Interfaces** view reports the state of every MME interface in one place: **S1AP** to the eNodeBs, **GTP-C** (S11) to the SGW, **Diameter** for **S6a** (HSS) and **SLg** (location services), **SGsAP** (to the MSC for CS fallback and SMS), **LCS-AP / SLs** (location services), and **S10/N26** (inter-MME and EPC-to-5GC handover). It is the single screen for confirming which of the MME's signalling paths are up and which are down.

Logs

The **Logs** tab tails the MME's live log output to troubleshoot a single MME without leaving OmniWeb.

Monitoring dashboards

Attach rates, UE counts, handover rates and other time-series for the MME are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- The **HSS** holds the subscription and authentication data the MME fetches over **S6a**.
- The **SGW-C** is the control peer over **S11**. The MME drives session and bearer setup on it.
- The **MSC** is reached over **SGsAP** for CS fallback and SMS.
- Other **MMEs** are reached over **S10** for inter-MME mobility, and the **AMF** over **N26** for EPC-to-5GC handover.

Quectel EP06 AT Command Reference (cellular network testing)

Reference for the Test Devices modem console. Targets the **Quectel EP06** (firmware `EP06ELAR04A05M4G`, IMEI `868186042000040`) on the lab container, but applies to the EC25 / EG06 / EG06 / EM06 family. Every command below was tested against the live EP06; example responses are the **real responses captured from that modem** (it was in a `NO SERVICE / SEARCH` state at the time, so signal fields read empty, and the parsers must handle that).

The relay is request/response: `sendModemAt(serial, command, readMs, timeoutMs)` → backend `modem_at` → agent `modem.run_at(serial, command, min_read, timeout)`. `min_read` keeps reading the port for at least that long so async URCs are captured before the call returns. There is **no persistent URC stream**. See "Event logging" below.

Device / SIM info

Command	Purpose	Live response
ATI	Manufacturer / model / firmware	Quectel / EP06 / Revision: EP06ELAR04A05M4G
AT+CGMM	Model	EP06
AT+CGMR	Firmware revision	EP06ELAR04A05M4G
AT+CGSN	IMEI	868186042000040
AT+CIMI	IMSI	(SIM IMSI)
AT+QCCID	ICCID	(SIM ICCID)
AT+CPIN?	SIM PIN state	+CPIN: READY

Signal / measurement

Command	Purpose	Live response
AT+CSQ	RSSI/BER (legacy)	+CSQ: 99,99 (99 = no signal). dBm = $-113 + 2 \cdot \text{rssi}$.
AT+QCSQ	Per-RAT signal	+QCSQ: "NOSERVICE". When attached: +QCSQ: "LTE", <rssi>, <rsrp>, <sinr>, <rsrq>.
AT+QNWINFO	Current RAT / band / channel	+QNWINFO: No Service. When attached: +QNWINFO: "FDD LTE", "<mccmnc>", "LTE BAND 3", 1850.
AT+QENG="servingcell"	Full serving-cell measurements	+QENG: "servingcell", "SEARCH" (idle). See format below.
AT+QENG="neighbourcell"	Neighbour cells	OK (none when not camped).

AT+QENG="servingcell" LTE format

```
+QENG: "servingcell",<state>,"LTE",<is_tdd>,<MCC>,<MNC>,<cellID>,<PCI>,<EARFCN>,<band>,<UL_BW>,<DL_BW>,<TAC>,<RSRP>,<RSRQ>,<RSSI>,<SINR>,<srxlev>
```

- RSRP dBm, RSRQ dB, RSSI dBm (all signed integers).

- SINR is a **converted** value: actual dB = $(1/5)*\text{SINR} - 20$ (range 0..250 → -20..+30 dB).
- cellID/PCI/EARFCN are usually hex/decimal per field; band is the LTE band indicator.

AT+QENG="neighbourcell" LTE format

```
+QENG: "neighbourcell intra","LTE",<EARFCN>,<PCI>,<RSRQ>,<RSRP>,<RSSI>,<SINR>,<srxlev>,...
+QENG: "neighbourcell inter","LTE",<EARFCN>,...
```

Registration / attach

Command	Purpose	Live response
AT+CFUN?	Radio functionality	+CFUN: 1 (full). 0 = minimum/airplane.
AT+CEREG?	EPS registration	+CEREG: 2,2 (2 = searching).
AT+CEREG=2	Enable registration URC with location (TAC/cellID)	OK
AT+CGATT? / AT+CGATT=1 / =0	PS attach state / attach / detach	+CGATT: 0
AT+COPS?	Current operator selection	+COPS: 0 (auto, not registered).
AT+COPS=?	Network scan (operator search, slow)	list of (stat,"long","short","numeric",AcT) tuples.
AT+COPS=0	Automatic operator selection	OK
AT+COPS=1,2," <mccmnc>" [, <AcT>]	Manual select by numeric ID	OK

Cell / band / RAT locking

Command	Purpose	Live response
AT+QNWLOCK=?	Capabilities	+QNWLOCK: "common/4g", <num of cells>,[[<freq>, <pci>],...] and +QNWLOCK: "common/lte",[<action>[,<EARFCN>,<PCI> [,<status>]]]
AT+QNWLOCK="common/4g"	Query cell lock	+QNWLOCK: "common/4g",0 (0 cells = no lock); after lock +QNWLOCK: "common/4g",1,1850,0.
AT+QNWLOCK="common/4g", <n>,<EARFCN>,<PCI>[, <EARFCN>,<PCI>...]	Lock to n cells (works on this firmware)	OK
AT+QNWLOCK="common/4g",0	Unlock	OK (clears the lock).
AT+QNWLOCK="common/lte"	Query single-cell LTE lock (read only)	+QNWLOCK: "common/lte",0,0,0,0.

Firmware behavior (verified live on EP06ELAR04A05M4G): the

`common/4g` form is the one that accepts a lock *action* on this unit.

`AT+QNWLOCK="common/4g",1,<EARFCN>,<PCI>` locks (returns `OK`, lock persists) and `AT+QNWLOCK="common/4g",0` clears it. The `common/lte` set forms (`...,1,<EARFCN>,<PCI>` and `...,0`) all return **ERROR** on this firmware, so `common/lte` is query-only here. The UI therefore uses

`common/4g` for both lock and unlock. | `AT+QCFG="band"` | Query band mask |

`+QCFG: "band",0x8d0,0x1a0880800d5,0x0` = (GSM mask, LTE mask, TDS mask). || `AT+QCFG="band",0,<lte_hex_mask>,0` | **Band lock** (LTE). `0x0`

LTE mask = all bands. | `OK` | | `AT+QCFG="nwscanmode"` | RAT scan mode |
`+QCFG: "nwscanmode",0` (0 = auto, 1 = GSM, 2 = WCDMA, 3 = LTE). | |
`AT+QCFG="nwscanmode",<m>[ ,1]` | Set RAT preference (`1` = effective immediately) | `OK` | | `AT+QCAINFO` | Carrier-aggregation info | `OK` (none aggregated when idle). |

LTE band mask: bit (band-1), i.e. band N → $1 \ll (N-1)$. e.g. B3 = `0x4`, B7 = `0x40`, B28 = `0x8000000`. `0x0` (or all-ones) = all bands enabled.

Firmware note: `AT+QCFG="nwscanseq"` returns `ERROR` on this EP06 firmware (`EP06ELAR04A05M4G`). The UI does **not** surface it. `AT+QSCAN` is also not supported on this EP06 firmware (it is an RG/RM 5G-series command). The network scan uses `AT+C0PS=?` plus `AT+QENG` instead.

Voice calls (VoLTE / IMS)

EP06 voice is **VoLTE-only**: it needs IMS registered over LTE. `AT+QCFG="ims"` on this unit returns `+QCFG: "ims",1,0` (IMS enabled by config, current status 0).

Command	Purpose
<code>AT+QCFG="ims"</code>	Query/enable IMS. <code>AT+QCFG="ims",1</code> to force-enable.
<code>ATD<number>;</code>	Dial a voice call (trailing <code>;</code> = voice, not data).
<code>ATA</code>	Answer an incoming call.
<code>ATH</code> / <code>AT+CHUP</code>	Hang up.
<code>AT+CLCC</code>	List current calls: <code>+CLCC: <id>,<dir>,<state>,<mode>,<empty>,"<number>",<type></code> . Empty <code>OK</code> = no calls.
<code>AT+CLIP=1</code>	Enable calling-line ID URC (<code>+CLIP:</code>) on <code>RING</code> .

`AT+CLCC` call states: 0 active, 1 held, 2 dialing, 3 alerting, 4 incoming, 5 waiting.

SMS (text mode)

Command	Purpose	Live response
AT+CMGF=1	Text mode (vs PDU)	OK
AT+CPMS?	Storage areas / counts	+CPMS: "ME",0,255,"ME",0,255,"ME",0,255
AT+CMGL="ALL"	List messages (text mode)	per message: +CMGL: <index>," <stat>","<sender>","<timestamp>" then body line.
AT+CMGR=<index>	Read one message	+CMGR: "<stat>","<sender>"," <timestamp>" + body.
AT+CMGS=" <number>" then body + Ctrl-Z	Send (handled by agent send_sms).	+CMGS: <mr>
AT+CMGD=<index>[, <delflag>]	Delete message (delflag 4 = all).	OK
AT+CNMI? / AT+CNMI=2,1,0,0,0	New- message indication config	+CNMI: 2,1,0,0,0

PDP / data

Command	Purpose	Live response
AT+CGDCONT?	List PDP contexts (APNs)	+CGDCONT: 1,"IPV4V6","ims",...
AT+CGDCONT=<cid>,"<type>","<apn>"	Set context. types: IP, IPV6, IPV4V6, Non-IP.	OK
AT+CGACT=1,<cid> / =0,<cid>	Activate / deactivate context.	OK
AT+CGPADDR[=<cid>]	Show assigned IP.	+CGPADDR: 1,"<ip>"
AT+CGCONTRDP=<cid>	DNS / gateway / PCO.	+CGCONTRDP: ...
AT+QPING=1,"<host>"	Ping from the modem (URC replies; needs <code>min_read</code>).	+QPING: ...

Event logging (URCs): and the relay limitation

The agent relay is **request/response with a per-call port open**. It opens the serial port, sends one command, reads until OK/ERROR (or `min_read` elapses), then closes. It does **not** hold the port open to stream unsolicited result codes (URCs) back to the browser. So a true live event feed is not possible through this path without agent changes.

URCs worth enabling (each is a one-shot `OK` command):

Command	Enables URC
AT+CEREG=2	+CEREG: registration changes (with TAC/cellID).
AT+CGEREP=2,1	+CGEV: PS/bearer events.
AT+CLIP=1	+CLIP: caller ID on incoming RING.
AT+CNMI=2,1,0,0,0	+CMTI: new SMS indication.
AT+QINDCFG="all",1	+QIND: Quectel indications.

Pragmatic approach used in the UI: the Events tab enables the URCs above, then **polls** the modem on an interval (AT+CEREG?, AT+CLCC, AT+QENG, AT+CSQ, AT+CMGL) and renders state changes into a timestamped log. Each poll also uses a short `min_read` window so any URC that happens to arrive during the read is captured opportunistically. This catches state transitions (registration gained/lost, call started/ended, new SMS, signal change) at the polling cadence, which is sufficient for lab testing. The limitation is documented in the UI.

MSC: Mobile Switching Centre

The **MSC** is the call-control heart of the circuit-switched core. It holds the visitor location register (VLR) for the subscribers currently in its service area, sets up and tears down **voice calls**, carries **SMS**, and manages **mobility** (location updates, paging and handovers) for those subscribers. It reaches the radio side over **RAN/BSC** connections (A-interface, driven over M3UA/SS7), bridges voice to the IMS and interconnect through **SIP peers**, and steers the actual media through one or more **media gateways**. OmniWeb surfaces the MSC's live subscribers, calls, SMS transactions, RAN and SIP peers, routing table and pending pages. Longer-run trends are on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

The MSC is a per-instance element page reached from the sidebar when one runs in the site's inventory, and, where several are deployed, a fleet view for aggregate status. All traffic goes through the OmniWeb backend proxy to the MSC's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

A single-screen health read: the **API status** of the MSC and a summary of what it is currently handling, including registered **subscribers** in the VLR, **active calls**, in-flight **SMS** transactions, live **RAN connections** and the state of the **SIP peers** and **STP** signalling links. It is the quickest way to confirm the MSC is up and see, at a glance, its live load.

Subscribers

Every subscriber the VLR currently holds, each row keyed by **IMSI** and **MSISDN** with its registration and location state. Open a subscriber to show its full VLR context. From here an operator can:

- **Purge** a subscriber from the VLR to clear a stale or stuck registration so the subscriber re-registers cleanly.
- Run a **subscriber action** against a specific subscriber.
- Manage **supplementary services**: inspect and change the call-forwarding, barring and related services active for the subscriber.
- Send **Advice of Charge (AoC)** to the subscriber.
- Send a **silent SMS (Type 0)** or initiate a **silent call**: a location/reachability probe that does not alert the handset.

Calls

Every active call, each row showing the **calling** and **called** parties, the call **state** and its **duration**. Open a call to show its full detail: the legs, the serving RAN connection and the media path. An operator can **force-release** a call to clear a stuck or unwanted call.

SMS

The SMS transactions the MSC is currently handling, each showing the **originator**, **recipient** and **delivery state**, so you can confirm messages are being carried and spot ones that are stuck.

RAN

The radio side of the MSC. It lists the active **RAN connections** (the live A-interface associations carrying subscriber signalling) and the known **BSCs**, learned from the RESET exchanges on the SS7/M3UA links. This is where you confirm the base-station controllers are connected and see which subscribers are on which connection.

SIP Peers

The SIP peers the MSC bridges voice through are the IMS, softswitches and interconnect trunks, each row showing the peer's **address** and its reachability **status**. Open a peer to show its detail. An operator can **trigger an OPTIONS ping** on a peer to actively test reachability, which is the quickest way to prove a SIP path is up when calls to it are failing.

Media Gateways

The media gateways the MSC steers voice bearers through, each row showing the gateway and its **status**, so you can confirm the media plane is available for call setup.

Routing

The MSC's number-routing table holds the rules that decide, by dialled-number prefix, where a call is sent. Each rule shows its **prefix** and **destination**. An operator can:

- **Add a route:** a new prefix and its destination.
- **Remove a route** by prefix.
- **Test a route lookup** for a number: enter a number and see which route it resolves to, to confirm dialling plans before or after a change.

Paging

The pending paging requests are the subscribers the MSC is currently trying to locate over the radio network for an incoming call or SMS. An operator can also **page a subscriber** directly from here.

Logs

The **Logs** tab tails the MSC's live log output to troubleshoot a single switch without leaving OmniWeb.

Monitoring dashboards

Call rates, SMS volumes, subscriber counts and other time-series for the MSC are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- The **HSS** holds the subscriber data the VLR resolves against and where the MSC registers as the serving node.
- The **CSCF / IMS** is bridged over the MSC's **SIP peers** for voice.
- The **BSCs** connect on the radio side over the A-interface, carried on the SS7/M3UA signalling links.

Network Testing (Load Test)

Network Testing drives synthetic signalling load at a live core so you can measure how it behaves under stress. It is powered by the [RAN Simulator](#): virtual eNodeBs and UEs that exercise the full radio→core path (S1AP/NAS, GTP-U, IMS) rather than a single interface.

[← Operations Guide](#)

The **Network Testing** group in the sidebar collects the RAN Simulator load-test tools:

Sub-tab	What it does
RAN Load Test	Recent load-test jobs — attaches, data sessions and IMS registrations run against the core, with their results and a downloadable pcap.
Scenario Schedules	Define recurring scenario runs (e.g. an hourly attach batch), run one now, and browse the run history. Results are exported to Prometheus for alerting.
RAN Simulator	Stand up virtual eNodeBs/gNodeBs and UEs and drive them directly.
RAN Radio	Multi-RAT radio control (5G NR / 2G GERAN / 3G UTRAN).
RANsimware — IMS	IMS registration and VoLTE load across the simulated fleet.

All of these are documented in depth in [RAN Simulator](#).

Pointing a run at a core

When you turn up a cell, OmniWeb offers a dropdown of the core nodes it can dial — every **MME**, **AMF** and **STP** it finds across your site inventories — so you point the run at the right core (`mme_ip` / `amf_ip` / `stp_ip`) instead of typing an IP. With no inventory configured, the field falls back to free text.

Typical use

1. Open **RAN Simulator**, turn up an eNodeB/gNodeB and point it at the core to test.
2. Attach a batch of UEs and run a scenario (attach, data, IMS register, VoLTE, or an outage scenario).
3. Watch the results and, for signalling detail, download the run's pcap.
4. For routine coverage, save the scenario as a **Scenario Schedule** so it runs on an interval and exports pass/fail to Prometheus.

Related

- **RAN Simulator**: virtual eNodeBs/UEs for functional and load testing across the whole radio→core path.
- **Grafana Dashboards & Statistics**: scenario pass/fail and timing are exported to Prometheus; use Grafana for the full picture during a test.

OmniRoam: Roaming Agreements & Email Automation

OmniRoam's **agreements** side turns the paperwork of setting up a roaming partner (the emails, the AA.12/AA.13 forms, the IOT rates, the SIM swaps, and the provisioning work) into a single tracked record per partner. It automates the parts that used to be copy-and-paste. Its headline trick: **forward a partner email to OmniRoam and it reads the thread, plucks out the facts, and updates the agreement for you**. It then generates and e-signs the contracts, files them, and raises the work orders to get the partner launched.

[← Back to Operations Guide](#)

Two different "OmniRoam"s. This document is about the **agreements / email / document** side of OmniRoam (the commercial and provisioning lifecycle with a roaming partner). It is **not** the same as the **roaming test books** (IR.38 / IR.48 / VoLTE conformance testing). Those are covered separately in [OmniRoam: Roaming Test Automation](#). One product, two jobs: *get the agreement signed* (here) and *prove the roaming works* (there).

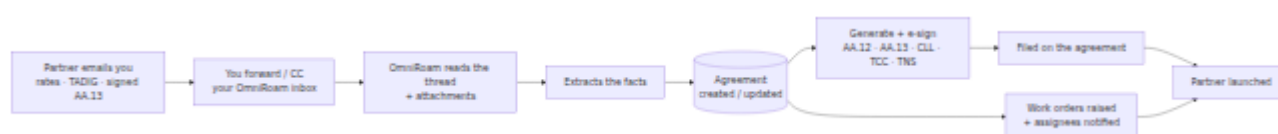
Table of Contents

- [The Big Picture](#)
 - [1. Inbound Email Ingestion: "plucking the data out"](#)
 - [2. Agreements & Partners](#)
 - [3. Automatic Document Generation & Signing](#)
 - [4. Filing & Storage](#)
 - [5. Work Orders & Assignee Notifications](#)
 - [The Full Lifecycle, End to End](#)
 - [What You Set Up Once](#)
-

The Big Picture

Everything in OmniRoam hangs off one record: the **Agreement**, one per roaming partner. An agreement carries the partner's name, country, TADIG(s)/PLMN, its lifecycle **status**, its contacts, the agreed commercial **rates**, the SIMs exchanged, the generated and received **documents**, an activity **timeline**, and the **work orders** that get the partner live.

You can build all of that by hand in the UI. But the point of OmniRoam is that most of it arrives *by email*, and OmniRoam files it for you.



The tenant you are working in is an **Operator** (your own network, the "home" side). Everything is scoped to it: your letterhead and signatory fill the generated documents, your inbound address drives the email automation, and your teams are the work-order assignees.

1. Inbound Email Ingestion: "plucking the data out"

What you do

Negotiating with a partner is an email conversation. To bring it into OmniRoam you do one thing: **forward the email, or CC the OmniRoam inbound address on the thread**. That is it. Your dedicated inbound address (for example `roaming@yourdomain`) is configured once per operator (Settings → paperwork details, `Inbound email`).

- **Forward a whole thread** once and OmniRoam builds the agreement from the history buried in the quoted messages.
- **CC the address on an ongoing negotiation** and every new email keeps the record current. Status advances, contacts and TADIGs fill in, rates land, and attached paperwork gets filed, email by email.

You never fill in a form to do this. The email *is* the input.

What OmniRoam does

Lexical error on line 7. Unrecognized text. ...oaming/inbound-email] HOOK -->
EP -----^

Try again

Step by step, once a mail reaches OmniRoam:

1. **It figures out which operator (tenant) the mail is for** by matching a To/Cc address to an operator's configured inbound address. If nothing matches, the email is logged as `no-operator` and *nothing is changed*: a safe no-op.
2. **It reads the attachments, not just the body.** The agreed rates or the signed terms are often inside an attached **AA.13**, an **IOT rate sheet**, or a spreadsheet. So OmniRoam extracts the text from PDF, Word, Excel and CSV attachments and feeds that to the extractor too. Signature-block logos (PNG/JPG) are ignored.
3. **It extracts the facts** using the ChatGPT API. The extractor is specifically told the outer sender is usually just *forwarding* the thread (often your own roaming manager) and to read the **whole quoted history**. It attributes rates, contacts, addresses and SIMs to whoever actually stated them, never to the forwarder. It returns a structured record: partner name, country, address, TADIG(s)/PLMN, best-fit lifecycle status, agreed rates (only if *both* sides explicitly agreed), partner contacts, exchanged/requested SIMs, whether a signed agreement is attached, and a one-line summary.
4. **It finds the right agreement, or creates one.** Matching is deliberately forgiving so a thread never spawns a duplicate: exact partner name, then a "normalised name" match (so *AT&T Mobility* joins *AT&T*), then a **shared PLMN** (MCC+MNC) match. This is a very high-precision signal, since a PLMN uniquely identifies a network. No match → a brand-new agreement.
5. **It merges in only what's new.** For both new and existing agreements, OmniRoam applies whatever *this* email newly reveals and **never clobbers** existing values:
 - **Status** only ever moves *forward* along the lifecycle (never backwards).

- **Country / address** set if blank or changed.
- **TADIGs** added, but only codes that exist in the IR.21 catalog (so the AI cannot invent one), and never your own home network's codes.
- **Contacts** added, or existing ones *enriched* (blank fields filled in).
- **SIMs** recorded (test SIMs the partner shared) and **SIM requests** (batches someone asked to be shipped to an address).
- **Agreed rates** become a rate tier, only when both sides explicitly agreed, and never a duplicate of an existing tier.
- A **signed AA.12/AA.13** (attached, or stated in the mail) advances the status to *IRA Signed*.

6. **It writes two records.** A friendly **activity-log entry** on the agreement timeline ("Email from *Jane at Partner* (forwarded): agreed data rate; status → rates_agreed, added contact Jane, recorded agreed rates"), and a raw **inbound audit row** (see below).

Processing is **idempotent**: the same email (by Message-ID) is only ever processed once, so CC'ing multiple people or re-forwarding a thread never double-counts.

Three ways the email gets in

You only need one. All three end up calling the same processing logic:

Path	How it works	When to use it
Cloudflare Email Worker	Cloudflare Email Routing forwards the raw message to a small Worker that streams the <code>.eml</code> straight to the OmniRoam webhook.	Domains already using Cloudflare; no mailbox to poll.
IMAP poller	OmniRoam logs into a real mailbox every ~60 s, processes unread messages, and flags them read.	Any standard mailbox (implicit TLS, STARTTLS, or plain).
Provider inbound-parse	A mail provider (Mailjet/SendGrid) POSTs the parsed message + attachments to the webhook.	If you already run inbound-parse.

The webhook (`/api/roaming/inbound-email`) is guarded by a **shared secret**, so only your configured mail path can post to it.

The Inbound Email Log: your audit trail

Every processed email, success *or* failure, writes a row to the **Inbound Email Log** (in the UI under the operator's Emails view). Each row shows the sender, recipients, subject, attachments, whether the **AI extraction succeeded**, exactly what it extracted, and what the email **changed** (or why it did nothing: `no-operator`, `no-partner`). This is your window into the automation: if a rate did not get picked up, the log tells you what the extractor actually saw.

The Inbound Email Log: one row per processed email, with the extraction outcome and the changes it made. This is the audit trail for the whole email→agreement automation.

2. Agreements & Partners

The **Agreements** list is your book of roaming partners for the current operator. Each row is one partner, with its lifecycle **status**, TADIG/country, priority, and who the next action is waiting on ("Us" or "Them").

The Agreements list. Statuses run left-to-right along the lifecycle; the filter bar narrows by status. You create agreements here by hand, or they arrive automatically from an inbound email.

The lifecycle (status)

An agreement moves forward through these stages. The email automation only ever advances the status, so the record reflects the furthest point the conversation has reached:

Enquired → IOT Negotiation → IOT Pending Signature → IRA Negotiating → Awaiting IRA Signatures → IRA Signed → Rates Agreed → SIM Exchange → Testing → Launched (plus Default Launch, Suspended, Terminated).

Opening a partner

Click a partner to open its **detail** page: a set of tabs/sections covering everything about the relationship:

- **Overview & timeline:** status, priority, notes, and the activity log (every email, generated document, signature, and manual note in date order).

- **TADIGs**: the partner's network codes (with MCC/MNC, connection type and direction). A "**find candidates**" action can pull the partner operator's other TADIGs from the IR.21 catalog for you to review and add.
- **Contacts**: roaming/commercial/technical people at the partner.
- **Rates**: the agreed commercial rate tiers (data/SMS/voice, currency, unit).
- **SIMs & SIM requests**: test SIMs exchanged, and batches requested for shipping.
- **Documents**: generated and received paperwork (see below).
- **Routing**: DRA routes, SS7 global titles, DNS and IP ranges for provisioning (much of it pre-populated from the partner's IR.21).

Everything on the detail page is **editable**. The automation gives you a head start, and you correct or add by hand as needed.

The IR.21 catalog

TADIG lookups draw on a **global IR.21 world-operator catalog** keyed by TADIG (organisation, network, MCC/MNC, DNS, IP ranges, SS7 nodes, contacts). This is how OmniRoam validates a TADIG the AI extracted, discovers a partner's sibling TADIGs, and pre-populates routing data.

3. Automatic Document Generation & Signing

Generating a document

On an agreement's **Documents** section you pick a document kind and click **Generate**. OmniRoam renders the real Word form, filled with both sides' details, ready to sign:

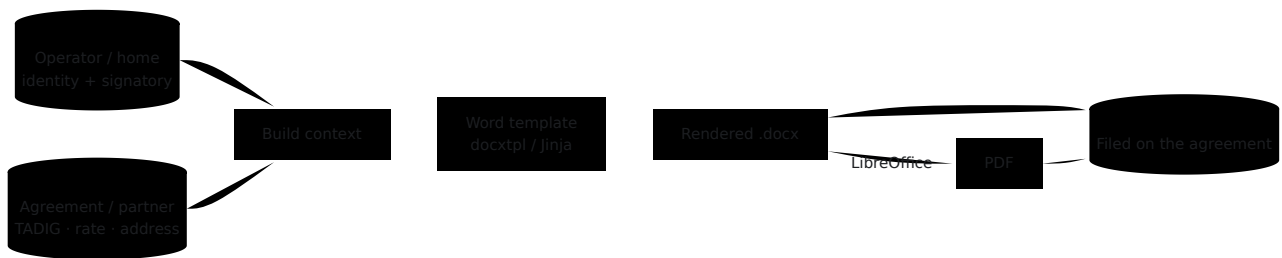
Kind	What it is	Format
AA.12	GSMA AA.12 IREG roaming agreement (the real form, both sides pre-filled)	.docx / PDF
AA.13	GSMA AA.13 common annexes, including the billing/settlement annex (see AA.13 settlement options)	.docx / PDF
CLL	Commercial Launch Letter (generic)	.docx / PDF
CLL (Bilateral / Unilateral, after testing)	Launch letter confirming the start date after successful IREG & TADIG testing	.docx / PDF
CLL (Bilateral / Unilateral, default)	Launch letter for a launch by default, without completing IREG/TADIG testing	.docx / PDF
TCC (VoLTE)	VoLTE Test Call Completion	.docx / PDF
TADIG completion	TADIG completion form	.docx / PDF
TNS Roaming Agreement Form	TNS clearing-house request (delivered as legacy .doc)	.doc
TNS 3G-4G Request	TNS clearing-house request	.docx

How the fields get filled. Each document is a Word template with placeholders (docxtpl / Jinja). OmniRoam fills them from:

- **The home side:** from your **Operator** record: your name, legal name, TADIG, PLMN, country, **signatory & title**, place, and address. Because this

comes from the tenant, each hosted operator gets its *own* letterhead and signatory automatically.

- **The partner side:** from the **Agreement:** partner name, its short-form abbreviation used in the contract body, TADIG(s), PLMN(s), country and address. The partner's own signatory/title default blank, so that line is left ready for them to sign.
- **Derived values:** effective date, direction (bilateral/inbound/outbound), hub company, Diameter realms, currency, and the confirmed IOT rate.



Output format: Word by default, PDF for signing

Generation produces an **editable Word** `.docx` by default. That is the working copy you review and hand-edit if needed. You get a PDF only when you actually need a fixed, final rendering:

- Tick the **PDF output** checkbox before clicking Generate to render a PDF instead of the `.docx`.
- The **e-signature** flow always uses PDF (the signed artifact must not change).
- **Download / preview** a filed document *as PDF* on demand.

A **Roaming Pack** email attaches the AA.12/AA.13 as editable `.docx` so the partner can review and comment. It is not the signing step, so it is not sent as PDF.

AA.13 settlement options (Annex C.3.2)

GSMA AA.13 Annex C.3.2 (billing & settlement) is a **menu of choices** each partner negotiates: which invoice model, which payment method, the settlement threshold and period, payment terms, and interest. Rather than lock

one set of choices into the template, OmniRoam exposes them as a settlement-options panel on the **Documents** section: expand **AA.13 settlement options**, set the values for this partner, then click **Generate AA.13**. The generated annex contains only the options you chose. The alternatives (and the GSMA "delete those which do not apply" scaffolding) are dropped automatically.

Every field has a sensible default, so generating without touching the panel yields the standard terms; change only what differs for a given partner.

Option	Default	Description
Invoice type	Single-network invoice (SNI)	Selects the settlement annex: Single-network (SNI) → Annex C.3.2 H, or Multi-network (MNI) → Annex C.3.2 I. The unused annex is removed.
Payment method	Net settlement	The settlement mechanism: Net settlement, Direct full payment (no conversion), Direct full payment (with conversion), or Netting without conversion . Only the chosen method's clause is kept.
Threshold settlement	Option 1 (Netting)	Option 1 (Netting) applies a single agreed low-balance threshold; Option 2 (Direct payment) uses per-party (Party A / Party B) thresholds.
Threshold	50	Low-balance settlement threshold amount, in the agreement's settlement currency (e.g. SDR). Below this monthly aggregate, settlement may be deferred.
Period start / Period end	January / December	The non-settlement period for balances below the threshold, typically the financial year.
Payment days	60	Days from the invoice date within which the debtor must pay.
Interest home %	8	Annual interest the creditor may charge on overdue balances when the home operator is the creditor.
Interest partner %	8	Annual interest when the partner operator is the creditor.

Option	Default	Description
Send zero-value invoices	Off	When off, the annex states each party will not send monthly invoices with a zero (0.00) amount.
BCE on TAP invoice	Off	When off, the annex states BCE usage will not be included on the TAP Roaming Invoice.

How it works. The choices are passed to generation as overrides on top of the built-in defaults. The AA.13 template resolves them into the finished annex: it selects the right option blocks, fills the amounts/periods/rates, and phrases the toggles. The settlement currency itself comes from the agreement's configured currency, not this panel.

Use case. A partner insists on 90-day terms, a 100-SDR threshold and an April–March financial year: open the panel, set **Payment days** 90, **Threshold** 100, **Period start** April, **Period end** March, and generate. Every other clause stays at the standard terms.

The TNS clearing-house forms are special. The two TNS requests are *not* docxtpl templates. They are the operator's genuine TNS forms with only the fillable fields changed and the right checkboxes ticked, so TNS receives exactly the layout it expects. They are **IOT-gated**: OmniRoam does not generate them until the agreement has a **confirmed (agreed) rate**. Trying earlier returns a clear "a confirmed IOT is required" message. The checkbox states (agreement type, direction, services, LTE interfaces, GRX) are auto-derived from the agreement, and all fields are overridable at generation time.

Headless LibreOffice on the server does the PDF generation (and the legacy `.doc` for TNS). You just get the finished file.

The Documents section of an agreement. Generate a contract on the left; generated and received documents (with signing status) list below, each downloadable.

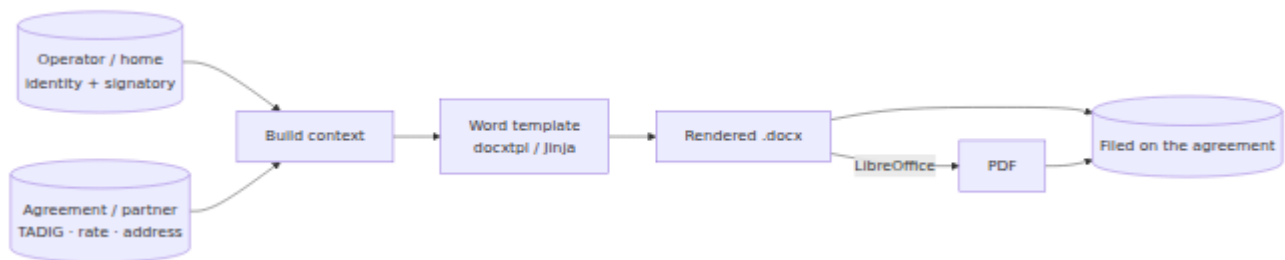
Signing: online or offline

OmniRoam supports two signing routes:

Online e-signature (partner signs in the browser, no login). From a generated AA.12/AA.13 you click **Send for signature**, name the signer and their email, and OmniRoam:

1. Creates a signature request with an opaque token and emails the partner a **secure link** (no account needed).
2. When they open it, OmniRoam marks the request *viewed* and logs it on the timeline. They review the document, optionally correct their signatory name/title/company, and **sign** by typing their name and/or drawing a signature.
3. OmniRoam produces the **signed PDF** and emails it to **both parties** (signer + your operator's return mailbox), files it on the agreement, and logs "Signed by ... on ...".

4. The partner can also **decline** (with a reason), which is recorded too.



Offline signing. Prefer to sign the old way? Download the generated document, sign it externally, and upload the signed copy back onto the agreement. A filename containing *signed/executed/countersigned* is recognised as a signed copy. An inbound email that attaches one advances the status to *IRA Signed* automatically.

The Roaming Pack: one-click first contact

Instead of generating each form separately, **Send roaming pack** emails a partner contact the **AA.12 + AA.13 + IOT offer** together in one message. It generates the contracts as PDFs on the fly and attaches your most recent IOT offer. Pick a contact (or type a name + email, which is backfilled as a contact), and OmniRoam sends the pack with your saved template wording. This is the "let us do a roaming deal" opening email, sent from inside OmniRoam.

Editable email templates

The three automated emails (**signature request**, **signed copy**, and **roaming pack**) use editable templates (per-operator override, else a global default), with placeholders like `{{ signer_name }}`, `{{ partner }}`, `{{ operator }}`, `{{ sign_url }}` and `{{ document_name }}`. Tune the wording once in Settings and every send uses it.

How email actually leaves OmniRoam. All outbound mail goes via the **Mailjet HTTPS Send API**, not SMTP. Hosting platforms (Railway) block outbound SMTP ports, so OmniRoam POSTs to Mailjet over HTTPS instead. Sends are **best-effort and non-blocking**: the UI button returns immediately and the mail goes out on a background thread, so a slow mail server never stalls the app.

4. Filing & Storage

Every document is filed **directly on the agreement** it belongs to. This covers **generated** documents (AA.12, AA.13, CLL, TCC, TNS...) and **received** documents (a signed copy attached to an email, an uploaded IR.21, an IOT sheet). There is no separate document store to manage:

- Documents are stored as blobs **in the database**, tagged with a **kind** (aa12, aa13, aa12_signed, iot, ir21, cll, other, ...), filename, content type, size, and whether OmniRoam **generated** it.
- Signed copies are tagged distinctly (e.g. aa12_signed) so the signed set is always distinguishable from drafts.
- **Inbound attachments are de-duplicated**: an identical copy (same filename + byte length) that arrives again is not re-filed.
- Every generation, upload, signature, and IOT export writes a **timeline entry**, so the agreement's activity log is a complete paper trail of what was produced and received, and when.

To retrieve a document, open the agreement's **Documents** section and download it (generated Word/PDF, or the received original). Signed PDFs produced by the online signing flow are stored on the signature request and also emailed to both parties.

You can also **export the agreed rate as a GSMA RAEX IOT 5.0 XML** file from the agreement. This is the same rate selection the TNS forms use, so the exported IOT and the generated forms always agree.

5. Work Orders & Assignee Notifications

Getting a partner live is *work*: shipping SIMs, changing routing, running IREG/TADIG tests, audits, chasing issues. In OmniRoam that work is tracked as **Work Orders**: the single tracked unit of roaming work. A work order can hang off an agreement (a partner) or stand alone.

The Work Orders view. Each order has a type, priority, status, optional partner link, and an assignee: the party notified when it is assigned.

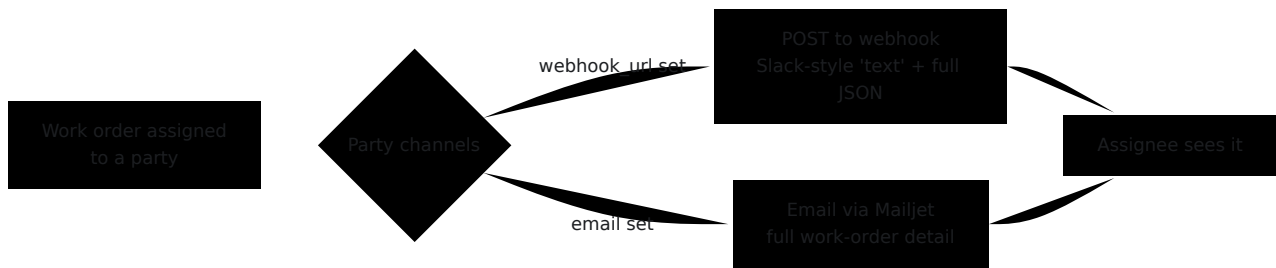
Creating and assigning

A work order has a **type** (SIM Dispatch, Routing, IREG Testing, TADIG Testing, Audit, Issue), a title, priority, status (Open / In Progress / Blocked / Done / Cancelled), an optional partner (agreement) link, free-text details, a due date, and type-specific structured fields (e.g. a SIM Dispatch carries quantity, postal address, carrier and tracking).

You assign a work order to a **party**: a named internal team or external contact (e.g. "Roaming NOC", "TADIG admin") managed per operator in **Settings**. You can set a **default assignee per work-order type**, so a new SIM Dispatch auto-assigns to whoever handles dispatch unless you override it.

The notification: webhook + email

When a work order is **assigned** (created with an assignee, or re-assigned to a new party), OmniRoam **notifies that assignee** through whatever channels the party has configured:



- **Webhook:** if the party has a webhook URL (e.g. a **Slack incoming webhook**), OmniRoam POSTs a message with a human-readable `text` block *and* the full work-order as structured JSON (type, priority, status, partner, the type-specific payload, a link back to the Work Orders page).
- **Email:** if the party has an email address, OmniRoam sends the same detail as an email (via Mailjet), subject `[Work Order] <title> assigned to you.`

A party with **neither** channel set is simply a silent assignee (no-op). Both notifications are best-effort and non-blocking, so a slow webhook or mail server can never stall the app. Re-assignment only notifies when the assignee actually **changes** to a new party (so editing other fields does not re-ping people).

From the assignee's chair: you get a Slack message (or email) the moment a job lands on your team, with everything you need: what to do, for which partner, the SIM count and shipping address (or the routing change, or the test scope), the priority and due date, and a link straight to the work order.

The Full Lifecycle, End to End

Putting it all together, this is the journey from "a partner emailed us" to "the partner is live":

Partner: 'interested in
roaming'

you forward to
OmniRoam inbox

Agreement created
status: Enquired

You Send Roaming Pack
AA.12 + AA.13 + IOT
offer

Partner replies with
agreed rates
+ signed AA.13
attached

CC'd on the thread

Rate tier recorded
signed doc filed
status → IRA Signed /
Rates Agreed

Counter-sign online
or upload offline copy

Work order: SIM
Dispatch
→ Roaming NOC notified

Work order: Routing
DRA / SS7 / DNS
provisioning

Work order: IREG /
TADIG Testing

Generate TNS forms
IOT now confirmed

status → Launched

At every step the agreement's **timeline** records what happened, and the **inbound email log** keeps the raw evidence of what each email contributed.

What You Set Up Once

Most of the automation just needs a little one-time configuration per operator (in **Settings**):

Setting	Why	Where it is used
Paperwork details (legal name, TADIG, PLMN, country, signatory & title , place, address)	Fills the home side of every generated document	Document generation
Inbound email address	The address you forward/CC partner mail to	Email ingestion
Signed-return mailbox	Where offline-signed and signed-copy emails go	Signing
Email templates (signature request, signed copy, roaming pack)	Your wording for the automated emails	Outbound email
Work-order parties + per-type default assignees	Who gets notified for each kind of work	Work orders

With those in place, forwarding a partner email really does become: *email in → agreement out → contracts generated and signed → work orders raised and your team notified.*

See Also

- **OmniRoam: Roaming Test Automation**: the *other* OmniRoam. It runs the GSMA roaming test books (IR.38 / IR.48 / VoLTE) against a live network. This is complementary: sign the agreement [here](#), prove it works [there](#).
- **Common Operations**: the viewing/editing patterns shared across all of OmniWeb.

OmniRoam: Roaming Test Automation

OmniRoam is the roaming test-book automation feature of OmniWeb. It runs the GSMA roaming compliance test books against a live network. It drives the individual test cases automatically from a test device. It records a pass/fail verdict with supporting evidence for each case. It then produces a completed GSMA test-book workbook ready to send to a roaming partner or hub.

In short: OmniRoam turns a manual, multi-hour roaming compliance exercise into a button press (or a scheduled job). It hands back the same official GSMA spreadsheet a tester would otherwise fill in by hand.

OmniRoam has two sides. This document covers the **roaming test books** (IR.38 / IR.48 / VoLTE conformance testing: *does the roaming actually work?*). The other side is the **roaming agreements & email automation**. It turns a partner email into a signed agreement, generated AA.12/AA.13/CLL/TCC/TNS paperwork, and the work orders to get the partner live. That is documented separately in **Roaming Agreements & Email Automation**. One product, two jobs: *get the agreement signed* (there) and *prove the roaming works* (here).

The Roaming Test Books screen. Each row is a book run against a roaming partner, with its current status.

Quick Links

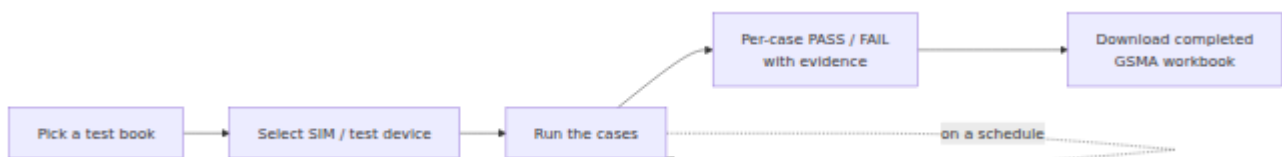
- **Test Books**: the available books (IR.38, IR.48, VoLTE) and every test case they contain
- **Running Tests**: how to create a book, run it, customise the case set, and schedule routine runs
- **Test Book Explorer**: how to inspect what each book tests and how each case is driven
- **Completed Test-Book Documents**: how OmniRoam generates the filled GSMA workbook
- **Roaming Agreements & Email Automation**: the *other* side of OmniRoam. Forward a partner email → agreement created, contracts generated and e-signed, filed, and work orders raised

What OmniRoam Does

A roaming test book is a checklist defined by the GSMA: a list of test cases that prove a subscriber from one network (the **Home PLMN**, "PLMN(a)") receives the expected services while roaming on another network (the **Visited PLMN**, "PLMN(b)"). Historically each case is performed by hand and the result written into a spreadsheet.

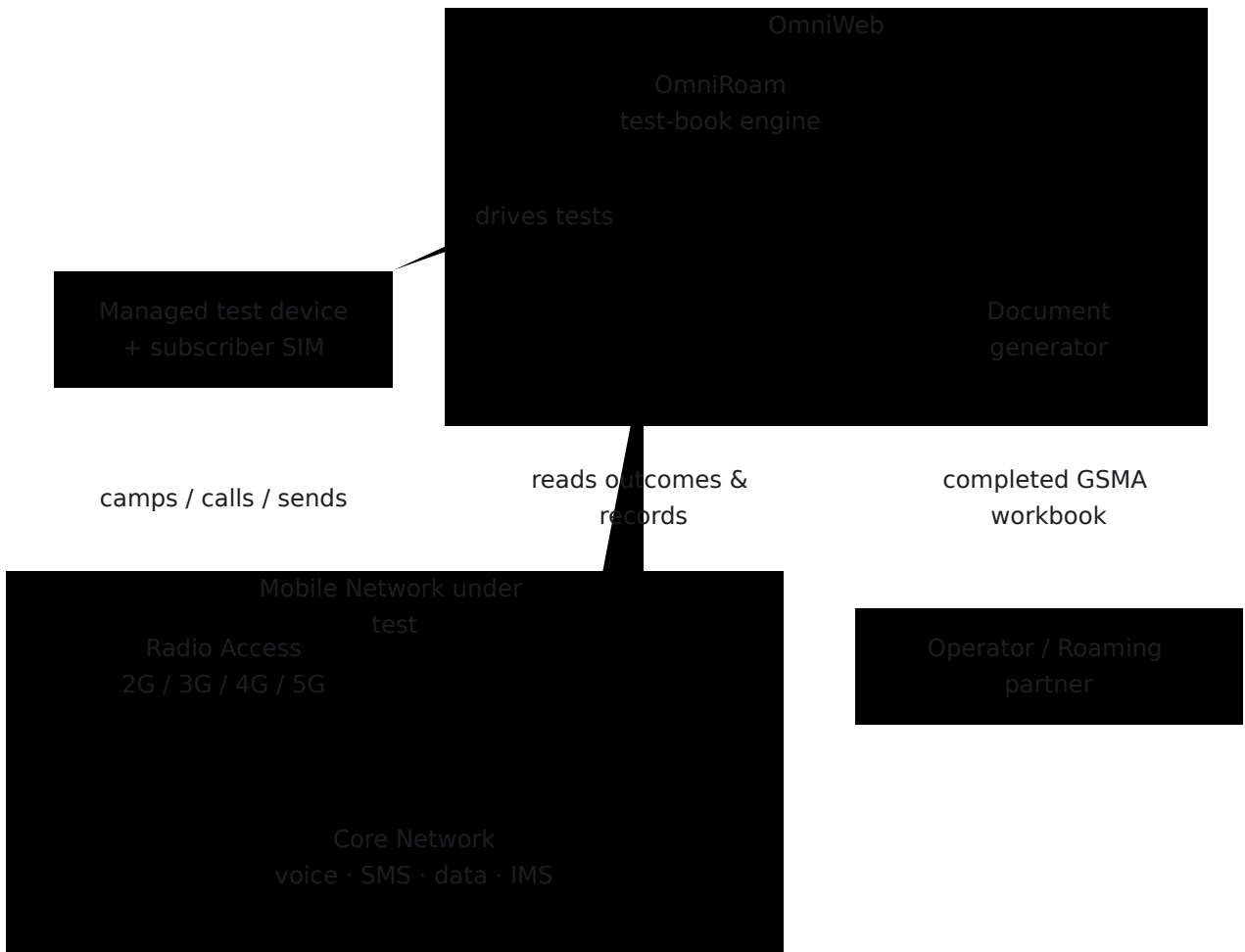
OmniRoam automates three things:

1. **Driving the test:** it places the call, sends the SMS, runs the data session, or provisions the supplementary service on a test device.
2. **Judging the result:** it compares what happened against the GSMA expected result. It records **PASS**, **FAIL** or **NOT PERFORMED**, with the supporting evidence (timings, throughput, network release causes, and so on).
3. **Producing the document:** it fills the official GSMA workbook with the results and all the surrounding metadata (identities, dates, tester details).



Where It Fits

OmniRoam exercises a real subscriber on the real network using a managed test device. Each automated case drives the device, then reads the outcome back from the device and, where applicable, from the network's own records (for example, the serving gateway's charging records for a data session). Nothing is simulated: a call that is reported as connected actually connected.



Key Concepts

Term	Meaning
Test book	A GSMA-defined set of test cases (e.g. IR.38, IR.48, VoLTE). See Test Books .
Test case	A single check within a book, identified by its GSMA number (e.g. IR.48 case 2.3, "Barring of All Outgoing Calls").
Run	One execution of a book (or a chosen subset of its cases) against a device and SIM.
Verdict	The recorded outcome of a case: <code>PASS</code> , <code>FAIL</code> , or <code>NOT PERFORMED</code> .
Evidence	The supporting detail captured for a verdict: timings, throughput figures, network causes, interrogation read-backs, etc.
Home PLMN (a)	The network that owns the subscriber being tested.
Visited PLMN (b)	The network the subscriber is roaming on.

Automated vs. Manual Cases

Most cases in each book are **auto-driven**: OmniRoam performs them end-to-end and records the verdict without operator interaction. A few cases depend on network conditions that you cannot induce from a test device alone (for example, the optional CAMEL prepaid negative tests, or a multimedia-messaging delivery). These remain **manual**. OmniRoam still lists them in the

book and the operator records the result by hand, so the final document is complete.

The [Test Book Explorer](#) shows exactly which cases in a book are automated and how each one is driven.

Standards Referenced

OmniRoam follows the GSMA Permanent Reference Documents and the underlying 3GPP specifications:

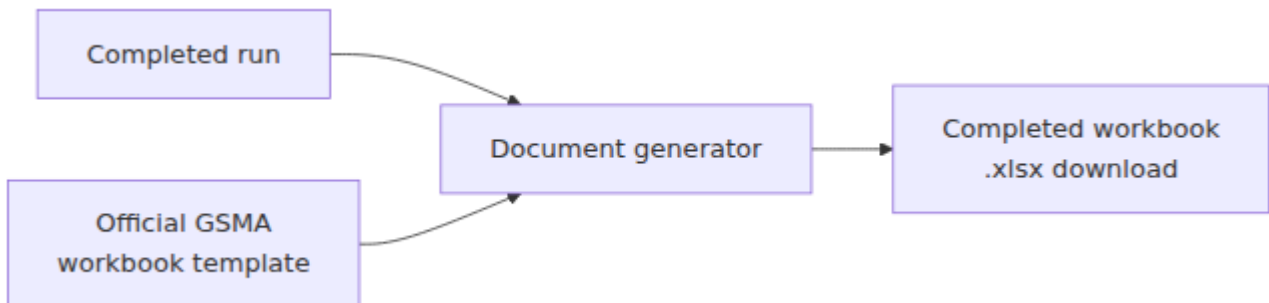
Reference	Applies to
GSMA IR.38	LTE roaming data test book
GSMA IR.48 / IR.24	Bilateral roaming (voice / SMS / MMS / data) test book
GSMA BA.65	Roaming VoLTE testing
GSMA IR.92	IMS profile for voice and SMS (VoLTE behaviour)
GSMA IR.88	LTE roaming guidelines (attach reject causes)
3GPP TS 22.011	Service accessibility (operator-determined barring)
3GPP TS 23.011	Supplementary service status (Active/Registered/Provisioned/Quiescent)
3GPP TS 22.082	Call Forwarding supplementary services
3GPP TS 22.088	Call Barring supplementary services

Completed Test-Book Documents

The end product of an OmniRoam run is a **completed GSMA test-book workbook**: the official GSMA spreadsheet for the book, filled with the run's results and all the surrounding detail, ready to send to a roaming partner or hub. OmniRoam produces it with a single **Download Test Book** action. It needs no manual transcription.

What You Get

OmniRoam takes the official GSMA template for the book and fills it in place, so the result is the same workbook a partner expects to receive (same sheets, same case numbering, same layout), but populated with real results instead of blank fields.



What Gets Filled In

Three categories of information are written into the workbook.

1. Per-case results

For every case in the book, the workbook's result cell is set to the recorded verdict (**PASS**, **FAIL**, or **NOT PERFORMED**). The case's comments cell is filled with the supporting evidence captured during the run.

In the document	Comes from
Test case result (PASS / FAIL / NOT PERFORMED)	The case verdict
Test case comments	The evidence: timings, throughput, network causes, interrogation read-backs, etc.

2. Identities and metadata

The header and operator-information sections are filled from the run:

In the document	Comes from
IMSI	The test SIM
MSISDN	The test SIM
Home PLMN (a) / Visited PLMN (b)	The roaming partner / PLMNs chosen for the book
Date of tests	The run date
Tester name, phone, email	The tester details on the book
APN (where applicable)	The data configuration used

3. Measured values

For the data and supplementary-service cases, the specific measured values the GSMA sheet asks for are written into their dedicated cells:

In the document	Comes from
Data session start / end times	The data session
Data volumes sent / received	The serving gateway's charging records
Session duration	The data session
Throughput / mean data rate	The throughput measurement
Service success indications (Yes/No)	The case outcome

Clean, Presentable Output

The blank GSMA templates ship with example and placeholder content in the fill-in cells: sample timings and reference notes intended to guide a manual tester. OmniRoam removes that placeholder content when it generates the document, so the completed workbook shows only your real results and empty fields where a value was not captured. It never carries the vendor's sample data into your delivered report, and it never overwrites the GSMA labels, headings, or structure.



When to Generate

You can download the document at any point. It always reflects the current state of the run. A typical flow is to run the book (or scheduled routine), review the per-case verdicts, re-run any cases that need attention, and then download the finished workbook once the results look right. Because the export reflects the live run, a re-export after a re-run simply produces an updated document.

Manual Cases in the Document

Cases that OmniRoam cannot drive automatically still appear in the workbook with their case number and title, ready for the tester to record a result by hand. This keeps the delivered document complete: the automated cases are filled with real evidence, and the manual cases are presented in the right place for the tester to finish.

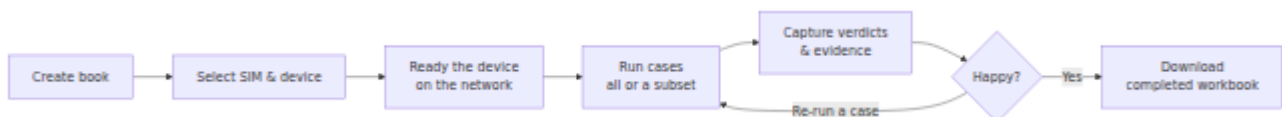
See [Running Tests](#) for how verdicts are produced and [Test Book Explorer](#) for the exact cell-by-cell mapping of what gets filled.

Running OmniRoam Tests

This guide covers the full lifecycle of a roaming test: creating a book, running it, watching the cases execute, customising which cases run, re-running a single case, and scheduling routine unattended runs.

- [The Test Lifecycle](#)
- [Creating a Test Book](#)
- [Running a Book](#)
- [How Automated Cases Are Driven](#)
- [Customising a Run](#)
- [Re-running a Single Case](#)
- [Routine and Scheduled Runs](#)
- [Reading the Results](#)

The Test Lifecycle



Creating a Test Book

The Roaming Test Books screen lists every book with its type, the home and visited networks, the subscriber identity, the tester, and the current status.

A test book is created from the **Roaming Test Books** screen. When creating a book you choose:

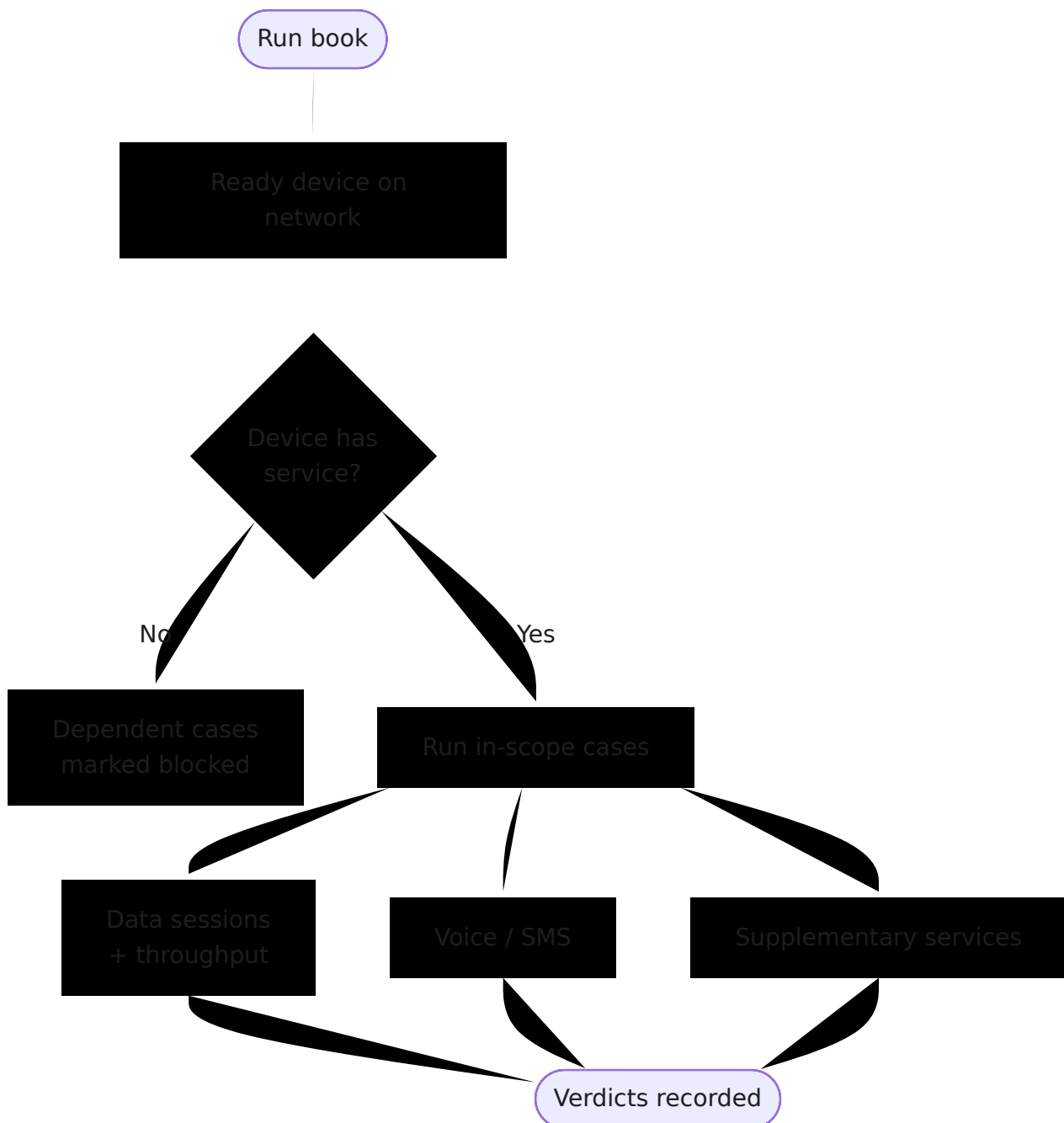
Field	Purpose
Book type	Which GSMA book to run: IR.38, IR.48, or VoLTE. This selects the case set.
Roaming partner / PLMNs	The home (PLMN a) and visited (PLMN b) networks. Used to populate the document header and to pick the correct throughput threshold.
SIM	The subscriber SIM to test with. Its IMSI and MSISDN are read and used throughout the run and in the final document.
Tester details	Name, phone, and email of the person responsible, written into the completed workbook.

Once created, the book lists every case from the chosen GSMA book, each starting in a "not yet performed" state.

Running a Book

Running a book performs each in-scope case in turn. Before the test cases run, OmniRoam readies the test device on the network. It presents the SIM and confirms the device has attached and has service. This is a **prerequisite gate**: if the device cannot get service, the dependent cases are reported as blocked rather than failed, because there is no point in a data or voice test with no bearer.

As part of readying the device, OmniRoam **reads the subscriber identity off the modem and fills the book automatically**: the **IMSI** is read from the SIM, the **home network (HPLMN)** is derived from it, and the **visited network (VPLMN)** is read from the network the device is currently camped on. These populate the book's header, and therefore the completed document, with what was actually tested, rather than with values typed in by hand.



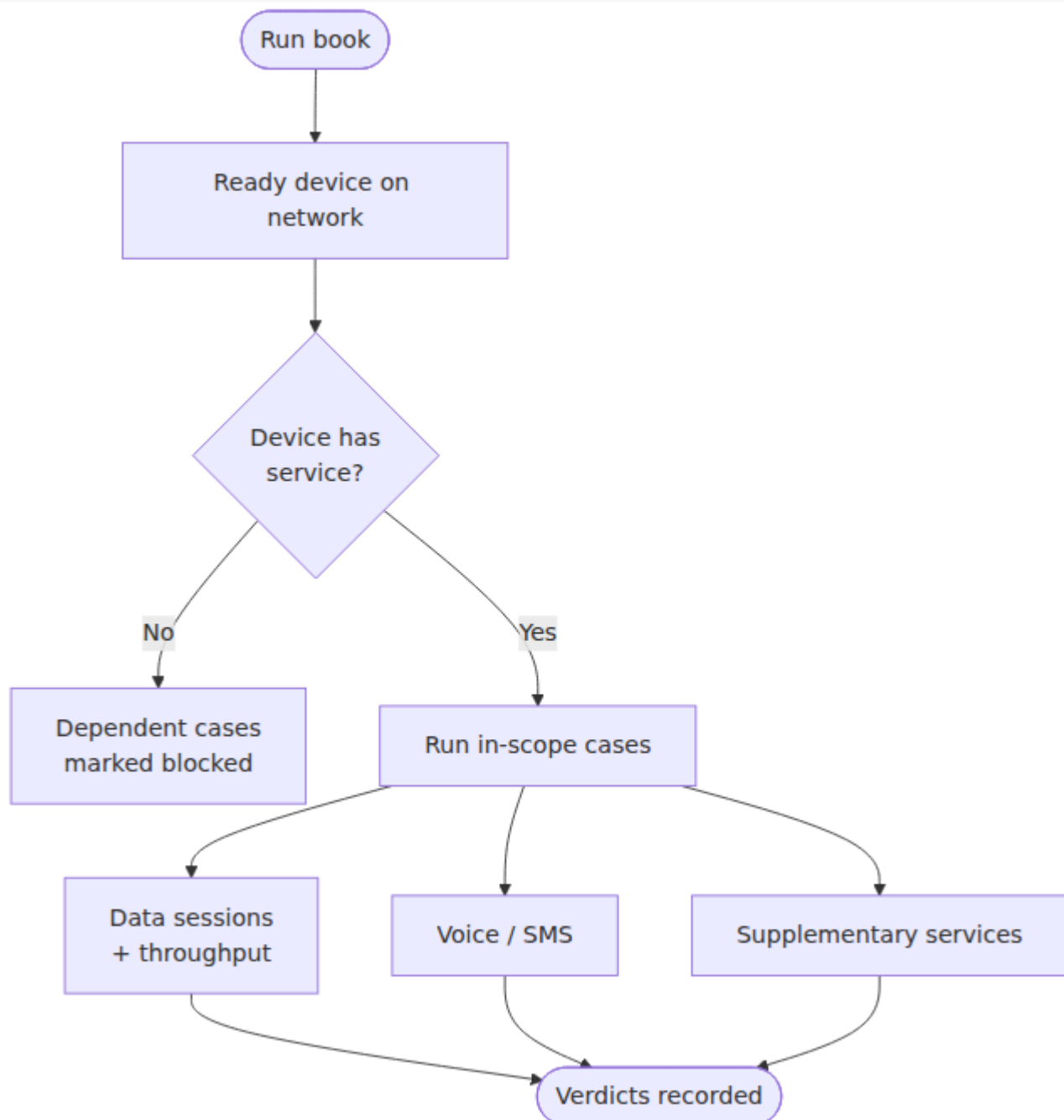
While a book runs, the screen shows each step as it happens, with the current step highlighted and a live pass/fail indication per case.

How Automated Cases Are Driven

Each automated case drives the test device, observes what the network does, and compares the outcome against the GSMA expected result. A case only passes on positive evidence. A step that cannot be confirmed is recorded as a failure with the reason, never a silent pass.

Data sessions and throughput

For data cases, OmniRoam establishes a packet data session, runs traffic to measure the achieved download/upload rate and latency, then disconnects so the network writes a clean charging record for the session. The data volumes and timings are read back from the serving gateway's charging records. For the IR.38 speed case, the measured rate is asserted against the throughput floor for the home/visited distance.



Voice calls

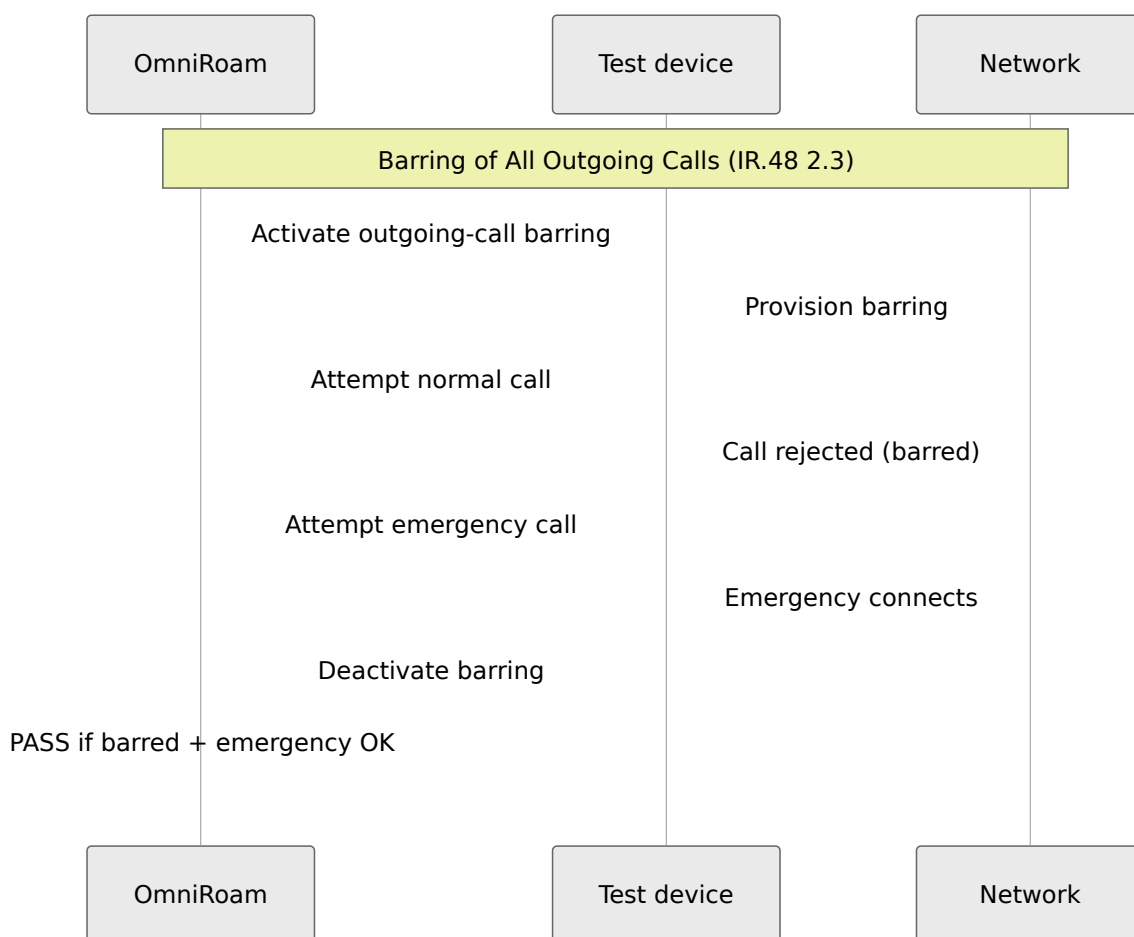
OmniRoam places a call from the device to a lab answering service, confirms the call reaches the active (answered) state, optionally holds it to confirm stability, then clears it. The verdict records the time to connect and whether the call stayed up.

SMS

OmniRoam sends a message with a unique body to a lab answering service and waits for the reply. Delivery of the reply confirms mobile-originated and mobile-terminated SMS end-to-end. A missing reply is a clear failure with the reason, not a crash.

Supplementary services (call barring and forwarding)

OmniRoam provisions supplementary services over the air from the test device itself, exactly as a subscriber would with the standard service control codes, and then exercises them:



For **Call Forwarding on No Reply**, OmniRoam registers the forwarding to a target number with a ring timer, then interrogates the service and confirms the network stored the forward-to number and timer that were registered.

Cancel Location (IR.38 3.1.2)

Some cases are driven from the network side rather than the device. For **LTE Cancel Location**, OmniRoam asks the home subscriber server to cancel the subscriber's location on the visited network, and records a pass when the request is accepted. This confirms the home network can withdraw a roaming registration.

IMS registration (VoLTE)

For VoLTE, OmniRoam first confirms the device is registered to IMS. This gates the IMS-dependent cases: if registration is not confirmed, the VoLTE call and SMS-over-IMS cases are reported as blocked.

Customising a Run

A run does not have to cover the whole book. Using the **test picker**, the operator selects exactly which cases to run. This is useful for:

- Re-validating only the cases that previously failed.
- Running just the voice cases, or just the data cases, during fault-finding.
- Skipping cases that depend on an arrangement not yet in place (for example, a CAMEL agreement).

Cases left out of the scope keep their existing verdict and are untouched.

Re-running a Single Case

Every case has a re-run control. Selecting it runs just that one case again, against the same device and SIM, and updates only that case's verdict. This is the fastest way to confirm a fix (for example, re-running the throughput case after a transport change) without repeating the whole book.

Routine and Scheduled Runs

OmniRoam can run tests **unattended on a schedule** so a roaming relationship is continuously validated rather than checked once.

- **Routine checks** run a compact set of health checks against a device (data, speed, voice, and SMS) and report a combined pass/fail. These are the same drives used by the full books, packaged as a quick recurring confidence check.
- **Schedules** run a chosen routine on a recurring interval automatically. Each scheduled run records its own result, building a history over time so a regression (for example, throughput dropping below the floor, or SMS delivery failing) is caught and visible without anyone running a test by hand.



Setting Up a Schedule

Schedules are managed on the **Schedules** screen under Roaming Tests. A schedule defines:

Setting	Purpose
Name	Identifies the schedule (and labels its metrics and history).
Modem(s)	The test device(s) the routine runs against.
Checks	Which checks the routine performs: data, speedtest, voice, SMS.
Interval	How often the routine runs (for example, every hour).

Once saved, the schedule runs automatically at its interval with no further interaction. The **Routine Test** screen shows the run history: each run's combined pass/fail, the measured speeds, and the per-check detail. A failing run surfaces the same evidence as an interactive run.

Prometheus Monitoring

Scheduled routine results are exported as **Prometheus metrics**, so roaming health can be alerted on and dashboarded alongside the rest of the network, the same way the scheduled tests on test devices are monitored. The exposition is served at `/api/ir38/routine/metrics` (reads the latest scheduled run per schedule and modem):

Metric	Type	Me
<code>omniweb_ir38_routine_pass</code>	gauge	Latest sched result (1 = p
<code>omniweb_ir38_routine_check_pass</code>	gauge	Per-check re latest run (l data/speedt
<code>omniweb_ir38_routine_download_mbps</code>	gauge	Download ra on the lates
<code>omniweb_ir38_routine_upload_mbps</code>	gauge	Upload rate the latest ru
<code>omniweb_ir38_routine_latency_ms</code>	gauge	Latency me latest run
<code>omniweb_ir38_routine_last_run_timestamp_seconds</code>	gauge	Unix time of scheduled r

Every series carries `schedule`, `serial`, `modem`, and `imsi` labels (plus `check` on the per-check metric), so you can alert per schedule, per device, or per home subscriber. The endpoint needs no login (scrapers cannot carry one) and can be gated with a bearer token when required.

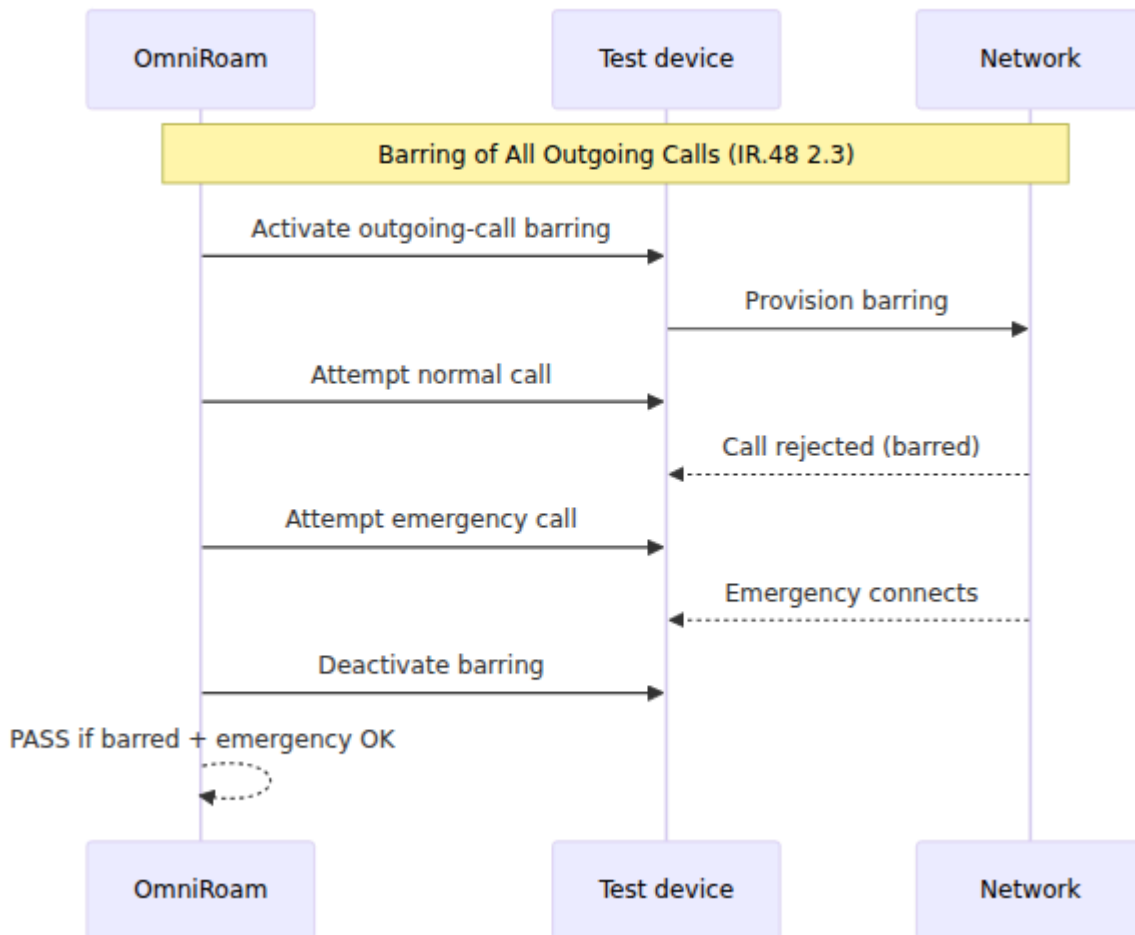
Example alert: roaming voice has been failing on a device for 15 minutes:

```
omniweb_ir38_routine_check_pass{check="voice"} == 0
```

Scheduled results accumulate into a history that can be reviewed at any time, and a failing run surfaces the same evidence as an interactive run.

Reading the Results

Each case shows its verdict (**PASS**, **FAIL**, or **NOT PERFORMED**) together with the evidence behind it:



- **PASS**: the GSMA expected result was confirmed, with the supporting figures.
- **FAIL**: the expected result was not met; the reason is recorded (for example, "throughput below floor", "no echo reply", or the network release cause for a call that did not connect).
- **NOT PERFORMED**: the case was not run, or is a manual case awaiting a tester.

When the results look right, generate the completed document. See [Completed Test-Book Documents](#).

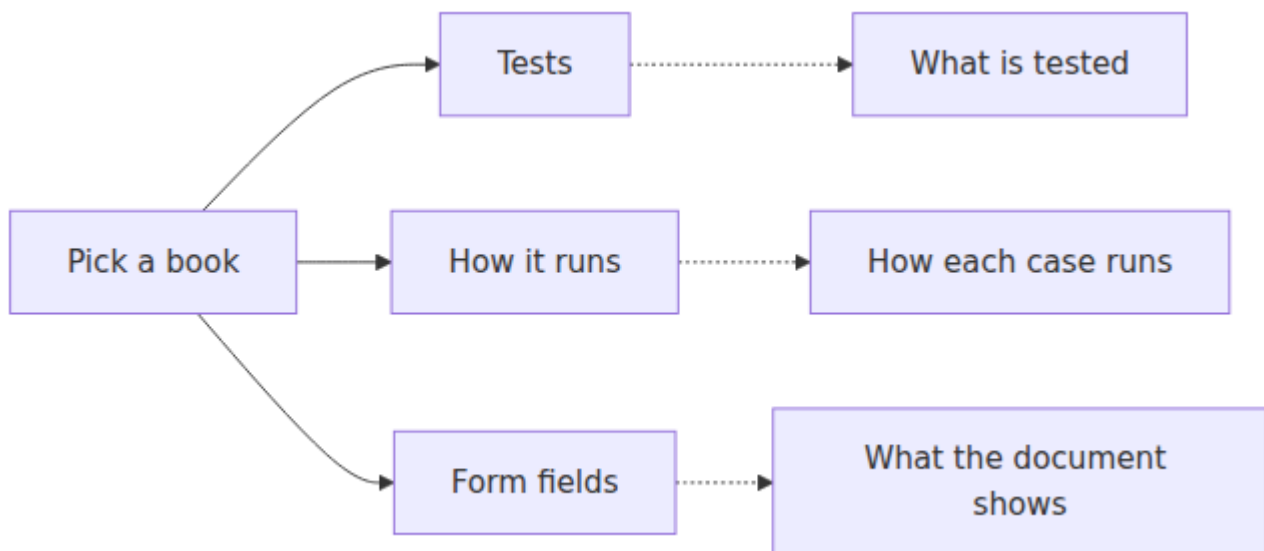
Test Book Explorer

The Test Book Explorer is a read-only reference view that shows, for each test book, exactly what it tests and how. It answers three questions an operator typically asks before trusting an automated run:

1. **What cases are in this book, and which are automated?**
2. **How is each case actually performed?**
3. **What ends up in the completed document, and where does it come from?**

The Explorer is reached from the Roaming Test Books screen via **Explore books**, and lets you switch between IR.38, IR.48, and VoLTE.

The Explorer showing the IR.48 book. Each case is listed with whether it is automated or manual and a description of how it is driven.



Tests

The **Tests** view lists every case in the book with:

- the **case number** and **title** (matching the GSMA specification),
- whether the case is **automated** or **manual**, and
- a plain description of **how it is driven**: for example "data session + charging-record check", "voice call", "barring activation and call attempt", or "tester records the result" for manual cases.

This is the quickest way to see, at a glance, how much of a given book OmniRoam performs for you and what remains a manual check.

How It Runs

The **How it runs** view shows the **step sequence** a full run of the book will perform, in order: readying the device, the data sessions, the throughput check, the voice and SMS drives, the supplementary-service drives, and so on. It reflects the cases currently in scope, so it is an accurate preview of what a run will do before you start it.

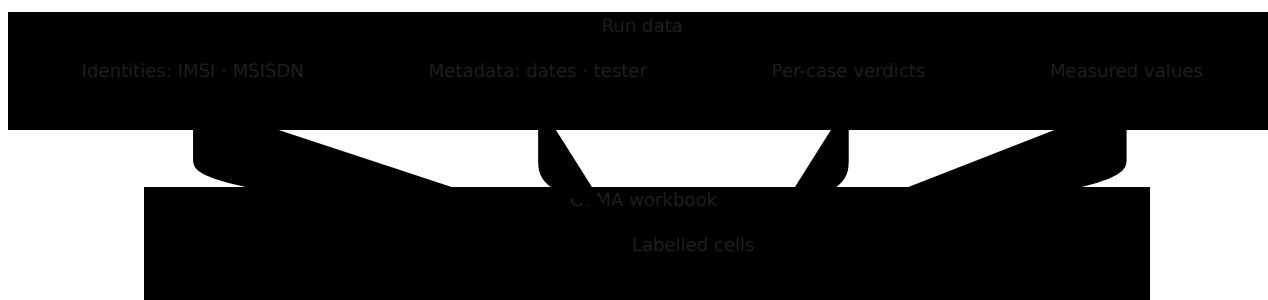
This view is useful for understanding the shape of a run and for confirming that a customised scope will perform the steps you expect.

Form Fields

The **Form fields** view lists every cell in the completed GSMA workbook that OmniRoam fills, each annotated with:

- the **sheet and cell** in the GSMA template,
- the **GSMA field label** (read from the template itself), and
- the **source** of the value: for example the subscriber MSISDN, the test date, a per-case pass/fail verdict, or a measured value such as data volume.

Because this list is derived from the same logic that produces the document, it cannot drift out of step with the actual export: what the Explorer says will be filled is exactly what gets filled. See [Completed Test-Book Documents](#) for the document itself.



Download Template

The Explorer also offers a **Download Template** action, which downloads the blank GSMA workbook for the selected book: the official template with the vendor's example/placeholder content removed, ready to fill in by hand if needed. This is the empty counterpart to the completed document a run produces (see [Completed Test-Book Documents](#)).

Why It Matters

A roaming partner accepting a completed test book needs confidence that the results are real and that the document faithfully represents what was tested. The Explorer makes the mapping transparent: every automated case can be

traced from the GSMA case number, through the way it is driven, to the exact cell it lands in on the final document.

OmniRoam Test Books

OmniRoam ships three GSMA roaming test books. Each book is the official GSMA test set, with cases identified by their GSMA case numbers so the completed document maps one-to-one onto the published specification.

- [IR.38: LTE Roaming Data Test](#)
- [IR.48: Bilateral Roaming \(Voice / SMS / MMS / Data\)](#)
- [VoLTE: Roaming VoLTE Testing](#)

For how each automated case is actually driven and judged, see [Running Tests](#). To browse a book's cases interactively, use the [Test Book Explorer](#).

In the tables below, **Driven** indicates whether OmniRoam performs the case automatically (**Auto**) or leaves it for the tester to record (**Manual**).

IR.38: LTE Roaming Data Test

Based on **GSMA IR.38 v3.0**. Proves that a roaming subscriber can attach for data on the visited LTE network, reach services through the home network, and receive acceptable throughput. It also proves the circuit-switched fallback voice and SMS, and 2G/3G data, where those are in scope.

Case	Title	Driven	What it proves
3.1.1.1	LTE speed test (data + throughput)	Auto	A data session establishes through the home gateway, carries traffic, and meets the throughput floor.
3.1.1.2	LTE Cancel Location	Auto	The home network can cancel the subscriber's location on the visited network (triggered from the home subscriber server).
3.1.1.3	LTE Operator Determined Barring	Manual	Operator-determined barring applied in the home subscriber record is honoured.
3.2.2	CS Fallback: Mobile Originating Voice Call	Auto	A subscriber-originated voice call succeeds via CS fallback.
3.2.3	CS Fallback: Mobile Terminating Voice Call	Manual	A call to the subscriber succeeds via CS fallback.
3.2.4	SMS over SGs	Auto	Mobile-originated and terminated SMS works over the SGs interface.
3.3.1	Data Access from 2G/3G using PGW in HPLMN	Auto	A 2G/3G data session reaches services through the home gateway.
3.4	Data Access from 4G + 5G NSA over S8	Auto	Packet data is available and reaches the internet (confirmed by a data session with connectivity check).

Cancel Location (3.1.2)

OmniRoam automates Cancel Location by asking the home subscriber server to cancel the subscriber's location on the visited network, then confirming the request was accepted. This is the home-network-initiated removal of the roaming registration (per the location-cancellation procedure in [3GPP TS 29.272](#)).

The Throughput KPI

Case [3.1.1](#) does more than confirm a data session. It measures the achieved download rate and asserts it against the IR.38 throughput floor. The floor depends on the geographic distance between the home and visited networks (longer paths permit lower rates):

Protocol	Distance < 4000 km	Distance ≥ 4000 km
HTTP	≥ 11 Mbit/s	≥ 5 Mbit/s
FTP	≥ 13.5 Mbit/s	≥ 6 Mbit/s

OmniRoam estimates the distance from the home and visited network identities and selects the correct threshold automatically; the operator can override the distance or the protocol when the defaults do not match the test setup. Recording a speed without testing it against the floor is never treated as a pass.

IR.48: Bilateral Roaming (Voice / SMS / MMS / Data)

Based on **GSMA IR.48 v2.1** (internally the IR.24 test set). The classic 2G/3G bilateral roaming book: basic calls, supplementary services, SMS, and packet data, exercised for a subscriber roamed into the visited network.

Case	Title	Driven	What it proves
2.1	MS1(a) calls MS2(a), both roamed in VPLMN(b)	Auto	A basic voice call establishes and holds.
2.2	CAMEL negative test (MOC, prepaid, no agreement)	Manual	A prepaid call is correctly barred where there is no live CAMEL agreement.
2.3	Barring of All Outgoing Calls (BAOC)	Auto	With outgoing-call barring active, normal calls fail while emergency calls still succeed.
2.4	Call Forwarding on No Reply (CFNRY)	Auto	No-reply call forwarding can be registered and interrogated, with the stored configuration confirmed.
2.5	Mobile Originated / Terminated SMS	Auto	An SMS sent from the roamed subscriber is delivered and a reply is received.
2.6	CAMEL negative test (SMS-MO, prepaid)	Manual	Prepaid SMS behaviour where there is no live CAMEL agreement.
2.7	Intranet / ISP access using home gateway	Auto	A data session reaches the internet/intranet through the home gateway.
2.8	MMS using home gateway and MMSC	Manual	A multimedia message is delivered via the home MMSC.

Supplementary Services in IR.48

Two supplementary-service cases are fully automated:

- **BAOC (2.3): Barring of All Outgoing Calls.** OmniRoam activates outgoing-call barring on the test subscriber, confirms that a normal outgoing call is rejected by the network, confirms that an emergency call still connects (the required exception), then removes the bar to leave the SIM clean.
- **CFNRY (2.4): Call Forwarding on No Reply.** OmniRoam registers no-reply call forwarding to a target number with a ring timer, then interrogates the service (the equivalent of dialling the `*#61#` status code) and confirms the network stored the forward-to number and timer that were registered. Observing the live divert requires an inbound call from a third party, so that final leg is left for a manual check; the provisioning and interrogation are automated.

OmniRoam reports the status of every supplementary service using the four standard 3GPP flags: **Provisioned**, **Registered**, **Active**, and **Quiescent** (see [3GPP TS 23.011](#)).

VoLTE: Roaming VoLTE Testing

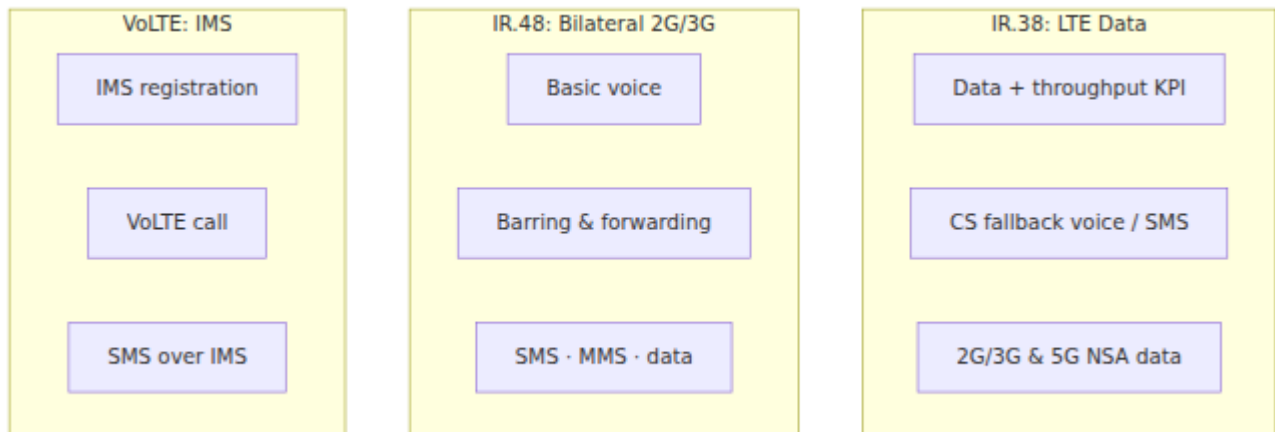
Based on **GSMA BA.65 §5.3**, with VoLTE behaviour per **GSMA IR.92**. Proves the subscriber can register to IMS while roaming and place/receive VoLTE calls and SMS-over-IMS.

Case	Title	Driven	What it proves
ims_reg	IMS registration over roaming	Auto	The device registers to IMS while roaming (the prerequisite for all VoLTE services).
mo_call	VoLTE mobile-originated call	Auto	A subscriber-originated VoLTE call establishes and holds.
mt_call	VoLTE mobile-terminated call	Manual	A call to the subscriber over VoLTE establishes.
sms_ims	SMS over IMS (SMSoIP)	Auto	SMS is delivered over the IMS network.
emergency	VoLTE emergency call	Manual	Emergency calling works over VoLTE (never auto-dialled).

The IMS Registration Gate

VoLTE services depend on IMS registration. OmniRoam runs `ims_reg` first as a **gate**: if the device is not IMS-registered, the IMS-dependent cases (`mo_call`, `sms_ims`) are reported as blocked rather than failed. There is no value in dialling a VoLTE call when IMS is down, and a blocked result tells the operator exactly why.

How the Books Compare



Book	Primary domain	Best for
IR.38	LTE / EPC data	Validating a new LTE roaming data relationship and throughput
IR.48	2G/3G circuit-switched + packet	Validating voice, supplementary services, SMS and basic data
VoLTE	IMS	Validating VoLTE roaming once IMS interconnect is in place

OTA SIM Management: over-the-air RFM/RAM

The **OTA** module manages SIMs **over the air**: it reads and edits files on a remote card. It sends **secured packets** (ETSI TS 102 225 / 3GPP TS 31.115, "03.48" / SCP80) that carry Remote File Management (RFM) and Remote Application Management (RAM) commands. The card verifies and decrypts each packet, executes the commands, and returns a signed **Proof-of-Receipt (POR)**.

In practice it gives you the same file-level editing as the offline **SIM Filesystem Explorer**: browse the file tree, decode an EF, edit the fields, write it back. The difference is that OTA reaches the card **remotely**: through a PC/SC reader on the central host, a reader at a remote **agent**, or over **SMPP** to a SIM in the field.

It builds on the same per-ICCID key store as the **SIM Key Store**. A profile says *which keyset and algorithms*, and the key store holds the actual **KIC/KID** for that card. The engine never logs or returns the keys. It returns only the resulting ciphertext and the POR.

What a secured packet is

A secured packet protects the command APDUs end-to-end, independent of the (plain) bearer. The engine has been validated against physical ISIM cards.

```
Command Packet = CPL | CHL | SPI | KIc | KID | TAR(3) | CNTR(5)
                | PCNTR(1)
                | RC/CC/DS | APDUs
```

CPL counts every octet after itself; **CHL** counts from SPI up to and including the RC/CC/DS. When ciphering is on, everything from CNTR to the end is enciphered and zero-padded to the cipher block size, with **PCNTR** recording how many octets were added.

- **TAR** (Toolkit Application Reference, 3 bytes) routes the packet to a service on the card: a specific RFM application or the RAM (card manager).
- **SPI** flags select ciphering, the integrity check, the counter policy, and what the PoR must contain.

- **KIc / KID** index which transport (ciphering) and authentication keyset the card uses; the **values** come from the key store per ICCID.

Security settings in full

Every switch below is settable on a profile, and every combination is implemented. A card vendor quotes these as four hex octets — MSL, PoR, KIc, KID — and a profile accepts them that way: paste `15213505` into the **SPI** field and every switch is set from it. The field also works in reverse, showing the four octets updating as you change a switch, so a setting can be read back against a personalisation sheet.

SPI octet	Setting	Choices
MSL	Integrity	none, redundancy check (RC), cryptographic checksum (CC)
MSL	Ciphering	on / off
MSL	Counter	no counter; present but unchecked; must be higher; must be higher by one
PoR	Requirement	never; always; only on error
PoR	Response integrity	none, RC, CC
PoR	Ciphering	on / off
PoR	Bearer	SMS-DELIVER-REPORT or SMS-SUBMIT
Klc	Ciphering algorithm	DES-CBC, triple DES-CBC 2-key, triple DES-CBC 3-key, DES-ECB, AES-CBC
Klc	Key version	1-3
KID	Checksum algorithm	DES-CBC, triple DES-CBC 2-key, triple DES-CBC 3-key, AES-CMAC
KID	Redundancy algorithm	CRC 16, CRC 32
KID	Key version	1-3

The KID octet means different things either side of the MSL: when the integrity choice is a **CC** it names a checksum algorithm, and when it is an **RC** the same bits name a CRC. The profile editor swaps the choices to match, and the RC length follows — two octets for CRC 16, four for CRC 32.

Key versions stop at 3 because a card record holds three keysets; the SPI itself allows 1-15.

Parameters per keyset

A card is personalised with an algorithm **per keyset** — keyset 1 might be triple DES with a CRC while keyset 3 is AES with a CMAC. That belongs with the keys, not with the profile that happens to select them, so each keyset in the SIM key store carries its own SPI.

When a keyset records one, its **algorithms and key versions** are used in place of the profile's, because a packet built with anything else is rejected by the card. Everything that is genuinely the operator's choice — the TAR, the PoR policy, the counter policy, whether to cipher, the command format — still comes from the profile. The preview reports which settings came from where, so a profile that disagrees with a card shows up before anything is sent rather than as a failed delivery afterwards.

A keyset with no SPI recorded changes nothing: the profile stands as written.

A **Proof of Receipt** is decoded against the same settings, including verifying its RC or CC. The status octet is reported in the spec's own words — *PoR OK, RC/CC/DS failed, CNTR low, CNTR high, CNTR blocked, ciphering error, unidentified security error, insufficient memory, more time needed, TAR unknown, insufficient security level*.

Remote command format

Command scripts default to the **compact** format: the C-APDUs one after another. A profile can select the **expanded** format instead (ETSI TS 102 226), which wraps each C-APDU in a 22 TLV inside an AA command scripting template, with indefinite-length coding (AE) as an option. The card replies with an AB or AF template holding one 23 R-APDU per command, so a failing script can be traced to the command that failed rather than only the last status word. Some TARs accept nothing else.

The APDUs inside are an ordinary C-APDU script: SELECT + READ BINARY for an RFM read, SELECT + UPDATE BINARY for a write, GlobalPlatform DELETE /

INSTALL for RAM.

On the wire (an SMS-PP) the packet rides in the TP-User-Data with a UDH Command Packet Identifier (IEI 0x70), PID 0x7F (SIM data download), DCS 0xF6 (8-bit, message class 2). In-shell it is delivered as a CAT ENVELOPE(SMS-PP-Download) APDU and the PoR comes back synchronously.

A packet too long for one short message is split automatically, either with the 8-bit concatenation header or — over SMPP — with the `sar_*` parameters so the SMSC reassembles it.

Profiles, contexts, and auto-routing

An **OTA profile** is a reusable security preset. It holds **no key material**:

Field	Meaning
<code>tar</code>	3-byte Toolkit Application Reference (the target service).
<code>kic_index</code> / <code>kid_index</code>	Which keyset (1/2/3) → <code>KIC{n}</code> / <code>KID{n}</code> from the card's key-store row.
<code>cipher_algo</code> / <code>cc_algo</code>	Ciphering and checksum algorithm (<code>triple_des_cbc2</code> , <code>aes_cbc</code> , ...).
<code>ciphering</code> , <code>rc_cc_ds</code> , <code>counter_mode</code>	SPI: encrypt? CC vs RC vs none? replay-counter policy.
<code>por</code> , <code>por_ciphared</code> , <code>por_rc_cc_ds</code>	POR required? ciphared? signed?

Two profiles are seeded, matching the values validated on sysmolSIM:

- **RFM-USIM-3DES-keyset3**: TAR `B00011`, keyset 3, 3DES, ciphering + CC, POR required + ciphared + CC. Operates inside **ADF.USIM**.
- **RFM-SIM-3DES-keyset2**: TAR `B00010`, keyset 2. Operates on the **MF-rooted** 2G tree (DF.GSM / DF.TELECOM).

Different applications need different services, so the explorer **routes the profile and the SELECT context automatically per file**, from the file's application:

File application	Context	Profile (by TAR)	Pre-SELECT before the ENVELOPE
ADF.USIM	usim	TAR B00011	SELECT ADF.USIM (by AID)
DF.GSM, DF.TELECOM	sim	TAR B00010	none (MF-rooted)
ADF.ISIM	isim	TAR B00013	SELECT ADF.ISIM (by AID)

Each context maps to one of your profiles (defaulted by matching TAR, overridable in the explorer). Pick a card and click any file (USIM, GSM or TELECOM). OmniWeb applies the right profile, keyset and SELECT without a switch of anything. ISIM works as soon as you create a profile for it (its TAR varies by card).

Transports

A transport delivers the secured packet and, where the bearer allows, collects the POR. The pluggable adapters share one interface:

Kind	Reaches	POR	Notes
<code>local_pcsc</code>	a card in a PC/SC reader on the central host	synchronous	the directly-attached validation path.
<code>agent_reader</code>	a card in a reader at a remote agent	synchronous	rides the same agent WebSocket as Test Devices / SIM Bank (<code>session_apdu</code> relay).
<code>smpp</code>	a field SIM over SMPP <code>submit_sm</code>	asynchronous	MT SMS to an SMSC; the PoR returns later as an MO SMS, or on the same connection when bound as a transceiver. See SMPP parameters below.
<code>http_admin</code>	a field SIM over HTTP (SCP81)	asynchronous	a trigger SMS makes the card dial the Remote Admin Server over TLS-PSK; scripts + responses flow there. See HTTP OTA below.

The **Explorer** uses only the in-shell transports (it needs a synchronous PoR); **Campaigns** can use any, including SMPP.

SMPP parameters

An OTA short message differs from an ordinary one in a handful of parameters, and an SMSC will accept a message that gets them wrong and quietly never deliver it to the card. All of them are settable:

Parameter	Default	Why it matters
<code>esm_class</code>	<code>0x40</code>	Marks the user data as starting with a header. Without it the card never sees the Command Packet Identifier.
<code>protocol_id</code>	<code>0x7F</code>	(U)SIM data download.
<code>data_coding</code>	<code>0xF6</code>	8-bit, message class 2 — what routes the message to the SIM rather than the inbox.
<code>dest_addr_subunit</code>	<code>0x02</code>	The destination subunit is the SIM. The parameter most often missing when a packet is accepted and never arrives.
<code>user_message_reference</code>	unset	Carried through to the MO PoR by most SMSCs, which is how an asynchronous PoR is matched back to its job.
<code>registered_delivery,</code> <code>validity_period,</code> <code>priority_flag,</code> <code>service_type,</code> <code>schedule_delivery_time,</code> <code>replace_if_present_flag,</code> <code>sm_default_msg_id</code>	SMSC defaults	Ordinary submit_sm controls.

Long packets segment three ways: the 8-bit concatenation header in the user data (the default), the `sar_msg_ref_num` / `sar_total_segments` / `sar_segment_seqnum` parameters so the SMSC reassembles, or a single `message_payload` where the SMSC supports it. Any other optional parameter can be added by name.

Before sending, a transport will report settings that are valid but will not reach a SIM — a missing header flag, a data coding that lands in the inbox, a destination subunit that is not the SIM.

Discovering a card's parameters

A card cannot be asked which TAR and security settings it was personalised with. When a batch arrives with keys but no documentation, **Profiles** → **Discover parameters** finds them by probing: it tries each of the card's keysets against the plausible parameter space and reports what the card accepts. A result can be turned straight into a profile with **Use**.

This needs an in-shell transport — a store-and-forward bearer cannot answer each probe.

Two things make a naive sweep useless, and the discovery run handles both:

- **A counter that is too low is rejected in silence.** Where the security level says *counter must be higher*, a card given a counter at or below its stored value returns nothing at all — no error, no Proof of Receipt. That is indistinguishable from wrong keys or a wrong TAR. Discovery therefore starts the counter high; since a counter only ever increases, starting high costs nothing.
- **The PoR byte decides where the answer comes back.** With bit 6 clear the card answers in the ENVELOPE response; with it set the card raises a proactive SEND SHORT MESSAGE instead and the ENVELOPE response is empty. Watch only one path and a correct answer is invisible, so both are searched.

A status of **CNTR low** is treated as a success rather than a failure: it means the keys, the TAR, the algorithms and the security level were all accepted, and only

the counter was wrong. Discovery stops there and resolves the counter by doubling, which takes a handful of round trips instead of continuing the sweep.

If nothing is accepted, that genuinely does not tell you which parameter was wrong — silence is the card's response to every kind of mistake.

Every accepted probe advances the card's counter, and a counter never goes back down.

Delivering through EF_SMS

A packet can also be written straight into a record of `EF_SMS` (`3F00/7F10/6F3C`) on a card in a reader, with the record marked unread. The card processes it exactly as though it had arrived over the air — no SMSC and no radio. This is the practical way to exercise a script against a card on a desk or in a bank.

Explorer

`src/pages/ota/OtaExplorer.tsx`: the remote SIM file editor.

1. Pick a **target ICCID** (from the SIM Key Store) and a **transport**.
2. Browse the **file catalog**: every known EF from the offline file model (ADF.USIM, ADF.ISIM, DF.GSM, DF.TELECOM), grouped by application and searchable by code / name / description.
3. Select a file → **Read over OTA**. The secured RFM read is delivered, the POR decoded, and the EF rendered: POR status, last SW, raw hex + ASCII, and a **typed field form** (decoded to named fields).
4. Edit the **fields** (re-encoded automatically) or the **raw hex**, then **Write over OTA**. The secured RFM `UPDATE` is sent and the file is **read back** automatically so you see the on-card state after the write.

Reads/writes are gated by the `ota` control permission; everything is recorded as a **job**.

Campaigns

`src/pages/ota/OtaCampaigns.tsx`: a saved operation fanned out across a set of SIMs: **target query** → **operation** → **profile** → **transport**.

- **Target query** is evaluated against the **SIM Bank** inventory at run time: ICCID list / prefix, IMSI prefix or range, agent, bank. An empty query deliberately matches nothing (no accidental fan-out).
- **Operation** is the RFM/RAM command (read/update binary or record, GP delete / get-status / install).
- **Run** creates one **job** per matched card. SMPP campaigns are fire-and-forget (the POR arrives later as an MO SMS).

A run is bounded (`CAMPAIGN_MAX_TARGETS`) so a fat-finger query cannot blast thousands of cards in one request.

Jobs

`src/pages/ota/0taJobs.tsx`: every OTA delivery (explorer reads/writes and campaign runs) with its status and POR. The detail drawer shows the **secured request packet** (ciphertext), the **Proof-of-Receipt**, and the decoded result (commands executed, last SW, response data). Card secrets are **never** stored on a job: only the ciphertext and POR.

REST API (/api/ota, JWT, RBAC

element type ota)

Method	Route	Purpose
GET	/profiles · POST · /profiles/<id> PUT/DELETE	OTA security profiles.
GET	/campaigns · POST · /campaigns/<id> GET/PUT/DELETE	Campaigns.
GET	/campaigns/<id>/targets	Preview which SIMs the query currently matches.
POST	/campaigns/<id>/run	Execute: one job per matched card.
GET	/jobs, /jobs/<id>	Job history (filter by campaign_id / iccid).
GET	/files/catalog	The browsable EF catalog (offline file model).
POST	/explore/read	Read one EF over OTA; returns the job + decoded EF.
POST	/execute	Run one operation (RFM update, RAM, raw) against one card.
POST	/preview	Build the secured packet without sending — returns the ciphertext, the SMS framings, the SPI breakdown, the counter used and the next one.

Method	Route	Purpose
POST	<code>/spi</code>	Decode four SPI octets into the named settings, or build the octets from settings. Pure arithmetic; no card or key material involved.
POST	<code>/discover</code>	Probe a card for the TAR and SPI it accepts, given only its keys. Needs an in-shell transport; advances the card's counter.
POST	<code>/decode</code> · <code>/encode</code>	Decode EF bytes ↔ typed fields (offline).

RBAC mirrors `simbank/ir38`: admins always pass; otherwise a JWT permission with `element_type == "ota"` grants view, and `POST` in its methods grants control. Reads are treated as control (they transmit a packet to the card). Routes are auto-documented in the [API Reference](#).

Database schema

Table	Row	Key columns
ota_profiles	one security preset	name, tar, kic_index/kid_index, algorithms, SPI flags.
ota_campaigns	one saved fan-out	name, profile_id, target_query (JSON), operation (JSON), transport (JSON), status.
ota_jobs	one delivery to one card	iccid, operation, request_hex (ciphertext), response_hex (POR), por_status, result (JSON), status.

Keys live in the existing `simbank_keys` table (the **SIM Key Store**), referenced per ICCID by the profile's keyset index. They are **not** duplicated here.

How it works

The OTA engine handles the crypto, secured-packet structure, CAT/SMS TLVs and the offline file model needed to build and decode 03.48/SCP80 RFM/RAM packets. It runs in the agent-role backend service and shares the agent connection for the in-shell transport. Delivery is pluggable across the local-reader, remote-agent-reader, and SMPP transports.

Security notes

- Card secrets (KIC/KID) are read from the key store to build packets but are **never** written to the audit log or returned in any response. Only the ciphertext request and the POR are stored.

- Writing a file is destructive; the explorer confirms before each write and reads back the result.
- The secured packet is genuine 03.48: tamper a byte and the card rejects it (the CC fails); with ciphering on, the plaintext APDUs never appear on the wire.

HTTP OTA (SCP81 / RAM-over-HTTP)

Alongside SCP80 over SMS/reader, OmniWeb speaks **SCP81** — HTTP-based OTA per **GlobalPlatform Card Spec v2.3 Amendment B**. Instead of one secured SMS, the card opens an **HTTP admin session over TLS-PSK** and runs a whole script.

```
trigger SMS (SCP80) → card -BIP OPEN CHANNEL→ TLS-PSK → Remote
Admin Server
                                     RAS serves command scripts, collects
responses, ends with 204
```

- **Trigger** — an `http_admin` op sends a secured SMS carrying the Amendment B *admin session trigger* (RAS host/URI, `psk_id`, retry policy) over the same SMPP bearer, then queues an **admin session**.
- **TLS-PSK** — a pre-shared key **derived from the card's KIC/KID** OTA keyset; no certificates. TLS profile per **GlobalPlatform Amendment B §3.3.2** (the spec ETSI TS 102 226 defers to): TLS 1.2 with `TLS_PSK_WITH_AES_128_CBC_SHA256` / `TLS_PSK_WITH_NULL_SHA256`, falling back to the RFC 4279 suites for TLS 1.0/1.1 cards. The `NULL` (integrity-only) suites are valid here — the script is already SCP80-secured underneath.
- **Admin loop** — the card POSTs to the RAS, which serves the queued command script(s) and collects each response until the session completes.
- **Admin Sessions** view (`/ota/admin-sessions`) shows each session live: state, script progress, and the command→response ladder. The linked **job** carries the final result, exactly like the other transports.

The crypto (PSK derivation, secured trigger) stays in the OTA sidecar; the public-facing **RAS** (`omniweb-ota-scp81`, an Elixir/OTP service) holds no keys — it terminates TLS-PSK and pulls scripts from the backend over localhost. Works for classic UICC RFM/RAM and eUICC (SGP.02). See `docs/ota-specs/HTTP_OTA_PLAN.md`.

PGW-C: PDN Gateway (Control Plane)

The **PGW-C** is the control-plane half of the EPC **PDN Gateway**. It is the anchor point for every subscriber's data session: it terminates **S5/S8** to the SGW-C, sets up and holds each **PDN connection**, allocates the UE its IP address from an operator **IP pool**, and selects and controls the user plane (**UPF**) over **Sx/PFCP**. It also runs the session's policy and charging over Diameter: **Gx** to the PCF/PCRF for policy and QoS, and **Gy** to the OCS for online charging. It also raises **CDRs** for offline charging. In the 5G core the equivalent function is the **SMF**. OmniWeb surfaces the PGW-C's live sessions, IP pools, UPF and Diameter peers, a topology map and session history, and diagnostic tools. The Grafana dashboards reached from the same page hold the longer-run trends.

[← Operations Guide](#)

The PGW-C is a per-instance element page reached from the sidebar when one runs in the site's inventory, with a fleet view where several are deployed. All traffic goes through the OmniWeb backend proxy to the PGW-C's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

A single-screen health read: the **API status** of the PGW-C and a summary of its live state (active session count, IP-pool utilisation, UPF peers and Diameter peers). It is the quickest way to confirm the node is up and carrying sessions.

Sessions

Every active PGW session, each row keyed by subscriber **IMSI** and **MSISDN** with its assigned **UE IP** and **APN**. Sessions can be searched by IMSI, UE IP, MSISDN or APN. Opening a session shows its full state: the PDN connection and bearers, the serving SGW and selected UPF, and the live Gx/Gy policy and charging context. You can edit a session here. An operator can adjust live state such as the session **AMBR** (which pushes a new APN-AMBR down to the UPF) and the requested **Gx event triggers** that control location-change reporting. An operator can also **delete** a session to force-release a stuck or misbehaving subscriber. Because the operator applies edits directly to live state, they take effect immediately.

Where several PGW-Cs are deployed, the same tab in the **fleet view** lists every session across all of them in one table, with the serving PGW-C shown against each row. The search is passed to every instance, so an operator can find a subscriber without knowing which PGW-C is anchoring them; opening a row goes to that session on its own instance. An instance that cannot be reached is named above the table, so a short list is never mistaken for a quiet network.

IP Pools

The operator IP pools the PGW-C allocates UE addresses from, each showing its **size**, **allocated** count and **utilisation**. Opening a pool lists the individual allocated addresses. Here you confirm a pool is not exhausted (an exhausted pool is a common cause of session-setup failures) and see which subscriber holds a given address.

UPF Peers

The user-plane (UPF) peers the PGW-C controls over **Sx/PFCP**, each showing its **IP address** and PFCP **association state**. Opening a peer shows its details. This is where you confirm the PFCP associations that carry every session's user plane are up.

Diameter

The Diameter peers the PGW-C talks to (the **PCF/PCRF** over **Gx** for policy and the **OCS** over **Gy** for online charging), each with its **hostname** and **connection state**. Opening a peer shows its details. When sessions are being rejected or throttled, here you confirm the policy and charging paths are up.

History

A searchable, paged log of session lifecycle **events**: *created*, *modified*, *deleted*, *handover* and *failed*. You can filter events by type and search them by IMSI, MSISDN, IP address or TEID. Here you reconstruct what happened to a specific subscriber's session over time, including failures.

Topology

A map of the PGW-C and the neighbours it works with: the SGWs it anchors sessions for, the UPFs it controls, and its Diameter peers. It gives a one-picture view of which signalling and user-plane neighbours the node is connected to.

UPF Selection

The rules that decide which UPF a new session is placed on, together with the current **health status** of each candidate UPF and the **PCO** (Protocol Configuration Options) configuration the PGW-C returns to the UE. This is where you confirm why sessions are landing on a particular UPF and check that all candidates are healthy.

UE Search

A direct lookup of a single subscriber by **IMSI**, **MSISDN** or **IP address**, returning that UE's information on the PGW-C. It is the fastest way to find one subscriber's session without scrolling the full Sessions list.

Charging

The live **online-charging (Gy/OCS)** state of every active session, each row keyed by **IMSI** and **MSISDN** and searchable by either. At a glance each session shows whether **Gy** online charging is on (with its **session id**, and the **OCS host** and **realm** it talks to), whether **offline (Gz)** accounting is also running, and whether the OCS has pushed the subscriber into a **walled garden** redirect. Opening a session expands its full charging detail: the per-URR **granted** quotas against the **used** volume (uplink, downlink and total octets), the **final-unit action** the OCS set for when credit runs out, the offline (Gz) counters, and any redirect target. This is how you read a subscriber's live credit-control state: whether they have credit, how much of their grant they have consumed, and why they may have been throttled or redirected.

Break glass: suspend online charging. At the top of the tab is a node-level switch that suspends online charging across the whole PGW-C. Use it when the OCS cannot answer and subscribers would otherwise lose service.

While the switch is armed, the PGW-C sends no CCR-I on attach and no CCR-U on live sessions. It releases walled gardens and un-gates gated bearers, so

traffic is carried without the OCS authorising it. Offline charging is unaffected: usage is still written to CDRs and can be rated later.

To arm the switch, give it two things:

Field	Required	Description
Suspend for	Yes	How long the switch stays armed. It disarms itself when the time runs out. The node sets the longest value it accepts.
Reason	Yes	Why charging is suspended. It is recorded against the override.
Armed by	No	Who armed it. Defaults to your address.
Send a Gy CCR-T on detach	No	Off by default. Turn it on to report usage the OCS never authorised.

Give a real reason. The reason and the operator are recorded, and they outlive the shift that armed the switch.

The panel shows the time left while the switch is armed. Press **Resume charging** to disarm it at once. The switch also disarms itself when its time runs out, which is what stops a forgotten override from becoming permanent free service.

CDRs

The offline **charging records** the PGW-C writes for billing: one row per session **start**, **interim** and **stop** event, most-recent first, and filterable by **IMSI** or **MSISDN**. Each record carries the subscriber's **IMSI**, **MSISDN** and **IMEI**, the **APN** and **QCI**, the **charging id** that ties a session's records together, the **uplink/downlink octets** counted for that interval, the **S-GW**, **PGW** and **UE** IP addresses, and the **source file** the record was read from. Here you reconcile a subscriber's data usage against the offline (Gz) accounting the operator bills on, and trace which record file a given session landed in.

Gy Test

An operator tool that issues a **Gy Credit-Control-Request** against the OCS to validate the online-charging path end to end. You supply an **IMSI** (and optionally an MSISDN and a requested/used octet count) and pick the request type. Use **Initial (CCR-I)** to open a session and request units, **Update (CCR-U)** to report usage and ask for more, or **Terminate (CCR-T)** to report final usage and close it. Then send it. The PGW-C defaults the rating group, Service-Context-Id and Auth-Application-Id. The parsed **Credit-Control-Answer** comes back on the same screen. Unlike the other tabs this one **does not fetch anything on load**: it is a form you submit. When online charging is misbehaving, it is the quickest way to prove the OCS is reachable and granting credit.

P-CSCF Monitor

The status and metrics of the P-CSCF discovery monitor. The PGW-C returns a P-CSCF address to VoLTE devices during PDN setup, and this tab confirms that discovery is healthy so IMS registration can proceed.

Logs

The **Logs** tab tails the PGW-C's live log output to troubleshoot a single node (session-setup failures, PFCP and Diameter errors) without leaving OmniWeb.

Monitoring

Session-count trends, throughput and other time-series for the PGW-C are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

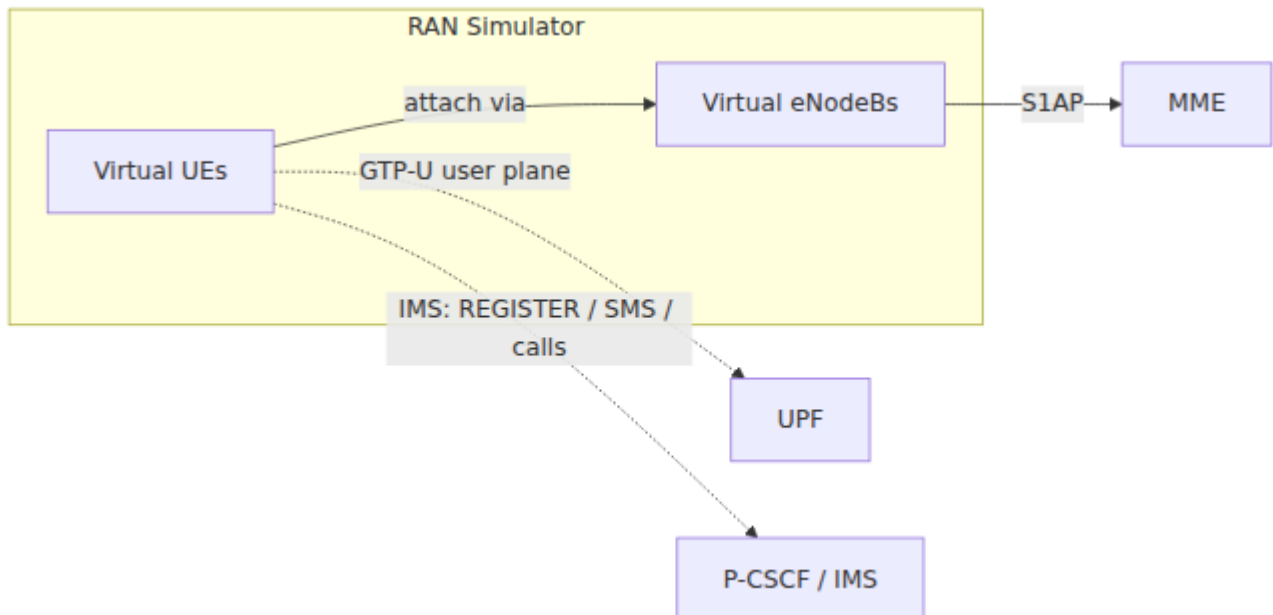
- The **SGW-C** is the control peer over **S5/S8**; it anchors the access-side bearers the PGW-C connects to the PDN.
- The **UPF** is the user plane the PGW-C selects and controls over **Sx/PFCP**.
- The **PCF/PCRF** provides policy and QoS over **Gx**.
- The **CHF/OCS** performs online charging over **Gy**; the PGW-C also raises CDRs for offline charging.

RAN Simulator

The **RAN Simulator** lets you exercise a live core network without physical radios or handsets. It stands up virtual **eNodeBs** and **UEs** that speak real S1AP/NAS to the MME and GTP-U to the gateways, so you can drive attaches, data sessions, IMS registrations, SMS and calls. Use it for functional testing, load testing, or reproducing field scenarios, all from OmniWeb.

[← Operations Guide](#)

What It Is



The simulator runs alongside the core it targets. In OmniWeb it appears as its own **RAN Simulator** area (Network Map, eNodeBs, UEs, IMS, Load Test, Scenario Schedules). Because a simulated UE behaves like a device, its UEs also appear in **Test Devices** so you can drive them from the same place as real handsets.

By default it starts with **no eNodeBs**. You turn up exactly the topology you want.

eNodeBs

Turning up an eNodeB attaches it to the MME (S1 Setup). An eNB can be connected to **more than one MME** (an MME pool). The Network Map shows the full set of MMEs an eNB is homed to.

Turn-up is **fire-and-forget**: OmniWeb asks the simulator to bring the eNB up and returns; the S1 link establishes in the background (watch the Network Map / eNodeBs list to confirm it came up).

Disconnecting an eNodeB

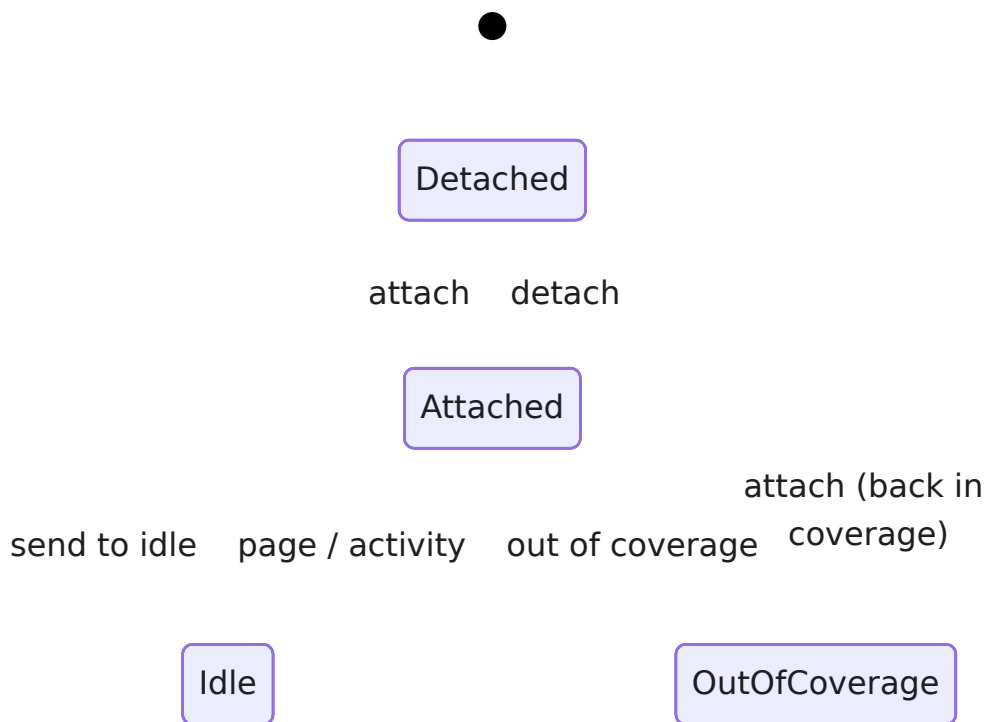
You can take an eNB down two ways, which model different real-world events:

Action	Models	Effect on UEs
Clean S1 disconnect	An orderly eNB shutdown / S1 reset	UEs are released cleanly and survive . They re-attach when a link is available
Drop link	The eNB abruptly losing its S1 (fibre cut, power loss)	The S1 association is aborted; UEs survive the eNB going away and re-attach rather than being torn down

Critically, taking an eNB down does **not** destroy its UEs. The UEs persist and re-attach, so you can simulate transport outages without rebuilding your fleet of subscribers.

UEs

Each simulated UE has a lifecycle you can drive directly:

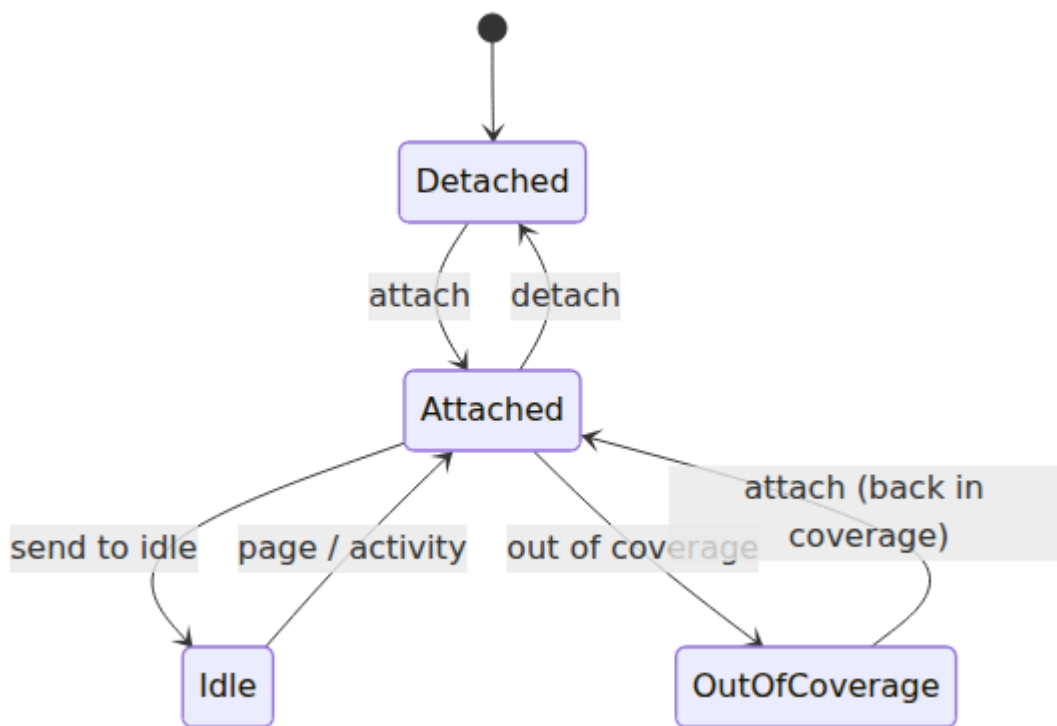


- **Attach / Detach:** join or leave the network.
- **Send to idle:** move to ECM-IDLE (as a real UE does between activity). Paging or new activity brings it back.
- **Out of coverage:** an *unclean* loss of radio (the UE drops without signalling), to test how the core handles a UE that simply vanishes. Attaching brings it back into coverage.

For each UE the simulator shows the **serving MME** it is attached to: the MME Group ID and MME Code (decoded from the UE's GUTI). So in a pooled deployment you can see which MME picked up each subscriber.

IMS registration & location

A simulated UE can register to IMS and place SMS/calls. When it registers, it includes its **cell location** (`P-Access-Network-Info` with the `utran-cell-id-3gpp` for its serving cell), exactly as a real UE does. So the S-CSCF's third-party REGISTER propagates that location to the SMSc, TAS, and other application servers. This lets you test location-dependent IMS behaviour end to end without real radios.



Driving UEs from Devices

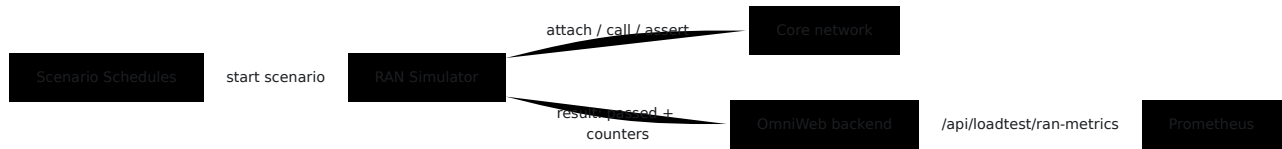
Because a simulated UE is just another device, it shows up in **Test Devices** tagged as a simulator, and can be filtered to there. You use the **same UE control panel** whether you open a UE from the RAN Simulator's Network Map or from the Devices fleet. It is one place to attach/detach, idle, drive out-of-coverage, register to IMS, send an SMS or place a call, and watch the action log.

Filter Test Devices to the RAN Simulator UEs to drive emulated subscribers alongside real handsets.

Scheduled Scenarios

A **scenario** is a saved test recipe: a named sequence of UE actions (attach, idle, wake, IMS register, call, SMS, handover, detach) that the simulator plays against the core. The simulator ships a library of presets, grouped by category (for example attach lifecycle, VoLTE, and negative reject cases). Run a scenario once from the Load Test view, or run it on a repeating schedule to monitor the core continuously.

The **Scenario Schedules** page (sidebar: Load Test → Scenario Schedules) manages the repeating runs. For each schedule you choose a scenario and how often to run it. OmniWeb records the pass or fail result of every run and exports it to Prometheus, so you can alert on it and chart it.



Runs are **serialised**. OmniWeb runs one scheduled scenario at a time, because a scenario can start many UEs and puts real load on the core. If a run is still active when the next schedule is due, OmniWeb waits and starts it on the next free cycle. Use the **Run now** button on a saved schedule to start it immediately.

Schedule parameters

Field	Type	Required	Default	Description
Name	String	Yes	-	Unique label for the schedule. Also the <code>schedule</code> label on the exported metric.
Scenario	String	Yes	-	The preset scenario to run. The picker groups scenarios by category.
Every (s)	Integer	No	3600	How often to run the scenario, in seconds. Minimum 60.
UE count	Integer	No	100	Number of subscribers the scenario runs.
RPS	Number	No	10	Target number of new scenarios per second.
Concurrency	Integer	No	50	Maximum number of scenarios in flight at once.
APNs	List	No	scenario default	APNs to attach, as a comma-separated list. Overrides the scenario default.

Pass and fail

A scheduled run **passes** when the simulator reports every assertion passed and no subscriber scenario aborted. A run **fails** if any assertion fails or any scenario aborts. A scenario with no assertions passes as long as no scenario

aborts. If OmniWeb cannot start the run, or the run does not finish in time, OmniWeb records the run as an **error**.

The page shows a **Recent runs** table with the result, the assertion count, the number of aborted scenarios, and the duration of each run.

Metrics

OmniWeb exposes the latest result of each schedule in Prometheus format at `/api/loadtest/ran-metrics`. Prometheus scrapes it as the `ransim_scenarios` job, which the monitoring role configures automatically.

Metric: `ransim_scenario_pass` **Type:** Gauge **Description:** Result of the most recent scheduled run: `1` for pass, `0` for not passed **Labels:**

- `schedule`: the schedule name
- `scenario`: the scenario id that ran
- `category`: the scenario category
- `status`: the run outcome (`passed`, `failed`, or `error`)

Companion gauges share the same labels:

Metric	Description
<code>ransim_scenario_duration_ms</code>	Duration of the last run, in milliseconds
<code>ransim_scenario_assertions_passed</code>	Passed assertions in the last run
<code>ransim_scenario_assertions_failed</code>	Failed assertions in the last run
<code>ransim_scenario_scenarios_aborted</code>	Aborted subscriber scenarios in the last run
<code>ransim_scenario_last_run_timestamp</code>	Unix time of the last run

Example queries:

```
# Scenarios whose last run did not pass
ransim_scenario_pass == 0

# Total assertion failures across all scheduled scenarios
sum(ransim_scenario_assertions_failed)

# A schedule that has not run in the last two hours
time() - ransim_scenario_last_run_timestamp > 7200
```

Grafana dashboard

The monitoring role ships a **RAN Simulator Scenarios** dashboard (Grafana folder **RAN**). It reads the metrics above and shows a pass or fail timeline, the latest result for each schedule, assertion failures, scenario freshness, and run duration. Filter by category, scenario, or schedule at the top. See [Grafana](#).

Typical Uses

- **Functional testing:** turn up an eNB and a UE, attach, bring up a data session, register to IMS, send an SMS, then confirm each core element behaves.
- **Load testing:** bring up many UEs to generate attach/session/IMS load against the core (see the Load Test view).
- **Scenario reproduction:** model outages (drop link, out of coverage) and pooled-MME behaviour to reproduce and diagnose field issues.
- **Continuous monitoring:** schedule a scenario to run on a timer and alert on the pass or fail metric (see [Scheduled Scenarios](#)).

Related

- [Test Devices](#): real handsets and simulated UEs, driven from one place.
- [Session Trace](#): correlate the signalling the simulator generates.

Remote SIM & Virtual SIM

OmniWeb can present **any SIM to any modem over the network**. It can also present a SIM that **does not physically exist**: OmniWeb emulates it from a saved card image plus the subscriber's keys held in the HSS. This turns a rack of SIM readers, card emulators ("cardems") and modems at one or more sites into a shared pool. Any tester can drive any subscriber profile against any modem without a walk to carry a plastic SIM to a card slot.

This page is the operations guide for that capability. The **inventory** half discovers which SIMs are in which readers. [SIM Bank](#) covers that half. This page covers how to **present** a SIM (real or virtual) to a modem and use it in a test. [SIM Images \(SAIP Profile Packages\)](#) covers how to save a card's file system as a portable profile package and how to use one to back a virtual SIM.

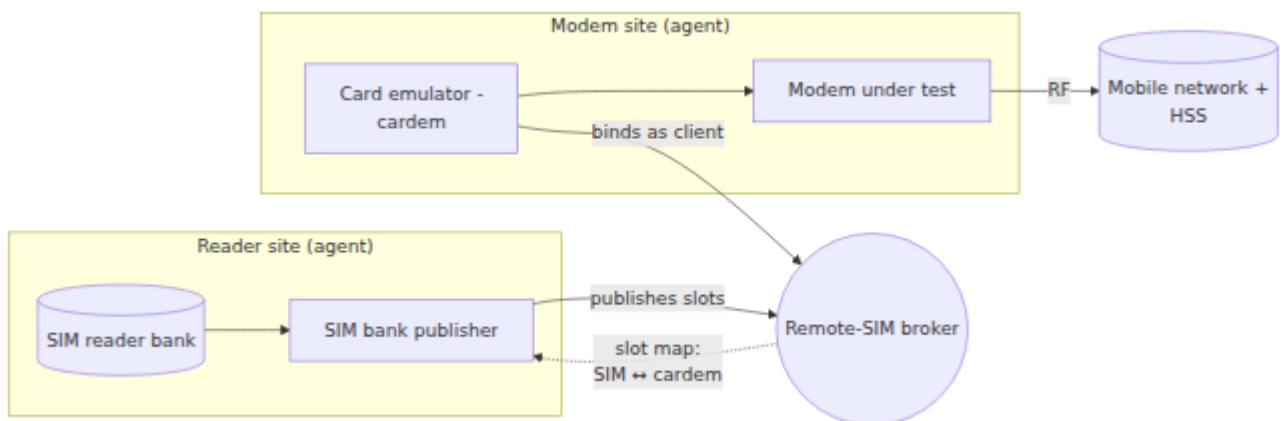
Table of Contents

- [Architecture Overview](#)
- [Concepts](#)
- [The SIM Bank: what is where](#)
- [Remote SIM: connect a SIM to a cardem](#)
- [The Cardem Console](#)
- [Virtual SIM: a subscriber with no physical card](#)
- [Using a SIM in a Roaming Test Book](#)
- [Troubleshooting](#)

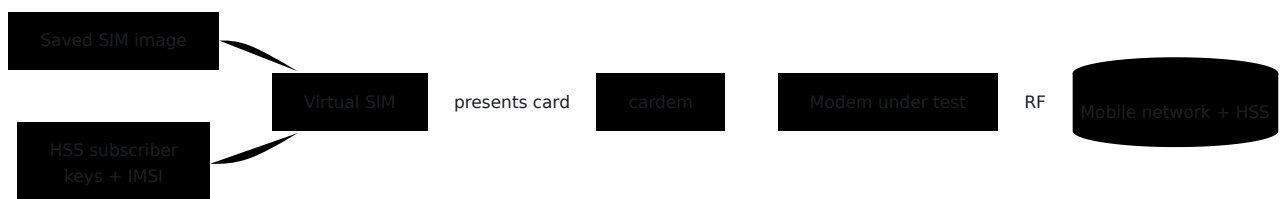
Architecture Overview

A SIM reader bank, one or more card emulators, and the modems under test connect to OmniWeb through its device agents. A central broker maps a chosen

SIM slot to a chosen cardem; the cardem then presents that SIM to the modem plugged into it.



For a **virtual** SIM, a software card replaces the reader bank. OmniWeb builds this software card from a saved SIM image and the subscriber's keys. Everything else is identical:



Concepts

Term	Meaning
SIM bank	A set of smart-card readers on an agent, published so their SIMs can be presented elsewhere. Each populated reader is one slot .
cardem	A card-emulator board (USB) that pretends to be a SIM towards a modem plugged into it. It plays back whatever card it is mapped to.
Slot map	The binding "present this SIM slot to that cardem". Creating it connects the two across the network; removing it disconnects them.
Modem under test	The modem behind a cardem. It attaches to the live network using whichever SIM the cardem is presenting.
Virtual SIM	A software card built from a saved SIM image plus a subscriber's keys. OmniWeb presents it to a cardem exactly like a real SIM. No physical card is required.

The SIM Bank: what is where

The **SIM Bank** page lists every reader slot on every connected agent, grouped by agent. The bank layout is the truth: empty, unreadable, and in-use slots all show, not only populated ones.

Each row is one physical reader. The status chip shows whether a card is present and readable. The row then shows the SIM's identity (ICCID, IMSI, MCC/MNC, country, operator name). Readers held by an active emulation session show as **In use**. Persisted SIMs whose reader is currently offline appear under **Previously seen**. They reappear when the agent reconnects with the SIM inserted.

Status chip	Meaning
SIM	A card is present and its identity was read.
Card present	A card is physically seated but its identity could not be read (e.g. an incompatible reader/card pairing).
In use	The reader is held by a remote-SIM/emulation session right now.
Empty	No card in the slot.

Per-row actions open the **SIM Filesystem** explorer or the **APDU Terminal** for that exact slot, and **Rescan** re-reads an agent's readers on demand.

Remote SIM: connect a SIM to a cardem

The **Remote SIM** page is where you bind a SIM to a cardem. Bindings work **across agents and sites**: the SIM can be in a reader at one location and the cardem/modem at another.

Steps

1. **Start the SIM bank** on the agent that holds the readers (**Start bankd**). This publishes its reader slots.
2. **Start the cardem client** for the card emulator you want to use (**Start client**). This binds the cardem so a SIM can be mapped to it.
3. Under **Connect a SIM**, pick a published slot and choose a cardem to bind it to. The mapping is created and shows under **Active mappings** as **ACTIVE**.
4. The cardem now presents that SIM to its modem. Reset the modem (see [the cardem console](#)) so it reads the newly presented SIM and attaches.

Global Reset tears down all mappings and returns every cardem to an idle state. Use it to recover from a confused test bench.

Field (Active mappings)	Meaning
State	<code>ACTIVE</code> once the cardem is presenting the SIM.
SIM (bank/slot)	Which agent's bank and slot the card lives in.
ICCID	The SIM presented.
Cardem (client/slot)	Which cardem (and which agent) is presenting it.
By	The operator who created the mapping.

The Cardem Console

In **Devices**, OmniWeb shows a card emulator as its own device type (`ngff-cardem`), next to modems and phones. It is never opened as a phone. OmniWeb labels unknown device types plainly rather than guesses them.

Open the cardem to get a dedicated console for the emulator and its modem.

Control	What it does
Start client	Binds the cardem as a remote-SIM client so a SIM can be mapped to it.
Reset modem	Re-presents the emulated card, which makes the modem re-read the SIM and re-run attach. This is the way you "swap the SIM" without a touch to the hardware.
Remote SIM	Connect or disconnect which remote SIM is presented to this cardem.

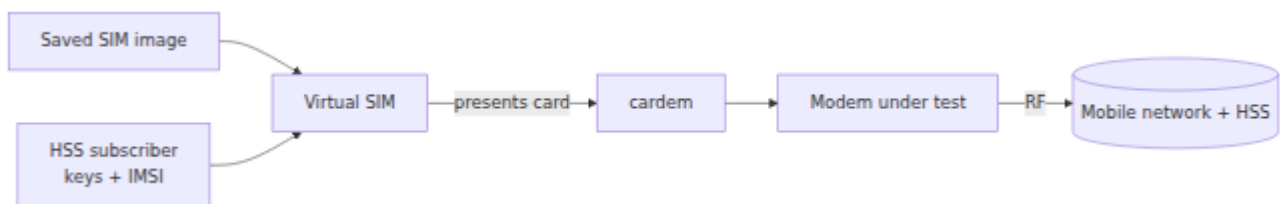
The **SIM source** panel states honestly what is and is not available: the cardem feeds the modem a *remote* SIM. The console does not yet offer to present a *local* card seated on the emulator board. The console says so rather than shows a control that does nothing.

Virtual SIM: a subscriber with no physical card

A **virtual SIM** lets a modem attach as a subscriber **you do not have a SIM for**. OmniWeb builds it from two things you already have:

1. A **base SIM image**: a saved snapshot of a real card's file structure (any compatible test SIM works as the base; see **SIM Images**).
2. The subscriber's **identity and keys from the HSS**: the IMSI and authentication keys of a subscriber already provisioned in the network.

OmniWeb combines them into a software card. This card answers the modem's file reads from the image and answers the network's authentication challenge with the HSS keys. It overlays the chosen subscriber's IMSI. Through a cardem, the modem cannot tell it from a plastic SIM: it reads the IMSI, runs authentication, and **attaches to the live network**.



When to use it

- Reproduce a specific subscriber's roaming or attach behaviour without a source for their physical SIM.
- Test many provisioned subscribers against one modem, back to back.
- Test profiles that only exist in the HSS (newly provisioned, pre-shipment).

What it proves. A modem on the bench has attached to the live network (registered, home network) as an HSS subscriber for which no plastic SIM exists. It read that subscriber's IMSI and completed the network's authentication entirely from the virtual card.

A dedicated **Virtual SIM Reader** element is the next step on this capability. There you "load" a virtual SIM from a chosen image and subscriber and present it like any other reader slot. Today OmniWeb wires the virtual card through the same Remote SIM broker and cardem path described above.

Using a SIM in a Roaming Test Book

Roaming **Test Books** (GSMA IR.38 / IR.48) can drive the SIM presentation as part of a run. In a test book's **Run Tests** panel you can choose:

- a **SIM (by ICCID / IMSI)**: tracked by identity, its current reader resolved automatically at run time, and
- a **Cardem modem**: the cardem the SIM is emulated to.

Present SIM to cardem establishes the mapping before you test. **Go: Run All Tests** presents the SIM first (when both a SIM and a cardem are selected), then reads the IMSI and runs the data cases. If the SIM is not currently in a reader, or the cardem's client is not running, the panel says exactly why. It does not run against the wrong card silently.

See [Test Devices](#) and the test-book documentation for the rest of the IR.38 flow.

Troubleshooting

A SIM does not appear in the bank

Symptoms: a reader is plugged in but no slot (or an empty slot) shows.

Possible causes & resolution:

- The card is not seated, or the reader/card pairing cannot power the card. The slot shows **Card present** or **Empty**. Re-seat the SIM.
- The reader was just plugged in. OmniWeb re-scans readers on hot-plug, but you can force it with **Rescan** on the agent's group.

The modem does not attach after you connect a SIM

Possible causes & resolution:

- The modem has not re-read the SIM. Use **Reset modem** on the **cardem console** to re-present the card.
- The mapping is not **ACTIVE**. Confirm under **Active mappings** on the Remote SIM page that the SIM is bound to the right cardem.
- The subscriber is not provisioned (or is barred) in the serving network's HSS.

"Card present (unreadable)" on a reader

Symptoms: a card is detected but its identity cannot be read.

Resolution: that reader cannot talk to that card (commonly a voltage/protocol mismatch on inexpensive readers). Move the SIM to a different reader in the bank.

Virtual SIM: modem reads the card but does not attach

Possible causes & resolution:

- The chosen subscriber must be **enabled** and fully provisioned (with a data profile) in the HSS for the serving network.
- The base image must come from a compatible card type. If init stalls, re-capture the base image from a known-good SIM.

SCEF: Service Capability Exposure Function

The **SCEF** (Service Capability Exposure Function), Omnitouch's **OmniSCEF**, is the network's **northbound gateway for the Internet of Things**. It exposes core-network capabilities to external Application Servers over the **3GPP T8** RESTful API defined in [TS 29.122](#), so a machine-to-machine platform can subscribe to events about a fleet of devices and exchange small amounts of data with them without ever speaking Diameter or GTP. It fronts two families of service. **Monitoring Events** ask the network to report when a device becomes reachable, loses connectivity, changes location and so on. **Non-IP Data Delivery (NIDD)** carries small non-IP payloads to and from constrained NB-IoT devices over the control plane. Internally the SCEF reaches the MME over the **T6a** Diameter interface ([TS 29.128](#)). OmniWeb surfaces its status, the T8 resources an Application Server has created, and the live NIDD sessions it is carrying.

[← Operations Guide](#)

The dashboard is a per-instance element page reached from the sidebar when a SCEF runs in the site's inventory. All traffic goes through the OmniWeb backend proxy to the SCEF's T8 API, so it stays behind the single authenticated gateway. A documentation link in the page header gives in-context help. The page is organised into tabs.

The SCS/AS ID model

Every T8 resource (every monitoring subscription and every NIDD configuration) is **scoped to the SCS/AS identity** the Application Server registered under (the *Services Capability Server / Application Server*). The T8 API has **no endpoint that enumerates the SCS/AS IDs** in use, so OmniWeb cannot discover them for you: the operator **sets the SCS/AS ID** on the page,

and OmniWeb remembers it **across the SCEF tabs** so you set it once and it carries from Overview to Monitoring Events to NIDD. Until an SCS/AS ID is set, the scoped tabs prompt for one rather than show an empty list. The resources exist under an identity you have to name. Enter the SCS/AS ID an Application Server was onboarded with (for example `iot-platform-01@omnitouch`) to work with that platform's subscriptions and configurations.

Overview

A single-screen health read of the SCEF itself: its **status**, **product** name, **licence** state and the **Diameter host** it uses for T6a towards the MME. Beneath it is the **application context** (the SCS/AS ID field described above) and, once an SCS/AS ID is set, the count of **monitoring subscriptions** and **NIDD configurations** currently held for that Application Server. It is the quickest confirmation that the SCEF is up and how much a given platform is carrying. Monitoring and statistics dashboards are provided as Grafana embeds, reached from the same page.

Monitoring Events

The **monitoring event subscriptions** an Application Server has created for the current SCS/AS ID. Each subscription asks the network to report a class of event and POST it to the Application Server; each row shows:

- **Event type:** the monitoring type, such as **UE Reachability**, **Loss of Connectivity**, **Location Reporting**, **Availability after DDN Failure**, **Roaming Status** or **Number of UEs in an Area**.
- **Target:** the device or group the subscription watches, identified by an **External ID**, an **MSISDN** or an **External Group ID** for a whole fleet.
- **Notification destination:** the **callback URL** the SCEF POSTs each event report to. It must be an absolute `http(s)` address the Application Server exposes.
- **Max Reports** and, for a reachability subscription, the **reachability type** (`SMS` or `DATA`). Together with the monitoring **duration/expiry**, these bound how many events are sent and until when the subscription lives.

Subscriptions can be created, edited and deleted; deleting one stops the Application Server receiving those event reports.

NIDD

The **NIDD configurations** are the Non-IP Data Delivery bindings an Application Server has created for the current SCS/AS ID. A configuration authorises small non-IP payloads to be exchanged with a device over the control plane. Each row shows:

- **Target:** the device by **External ID**, **MSISDN** or **External Group ID**.
- **DNN / APN:** the Data Network Name (APN) the NIDD PDN connection uses (for example `nidd.iot.omnitouch`).
- **Status:** `ACTIVE`, or terminated (`TERMINATED`, `TERMINATED_UE_NOT_AUTHORIZED`).
- **Max Packet:** the **maximum packet size** in bytes the non-IP payload may reach.
- **RDS:** whether the **Reliable Data Service** is enabled. It adds a lightweight acknowledged transport with source/destination ports over the non-IP link.

A configuration also carries a **PDN establishment option** that decides what happens when downlink data arrives while the device has no NIDD PDN

connection up: **wait for the UE** to establish one, **indicate an error**, or **establish the PDN connection** by paging the device.

Downlink data delivery

From a NIDD configuration you can **deliver a downlink payload** to the device. Enter the payload as text. The SCEF **base64-encodes** it before it is sent over NIDD, and each delivery can carry a **priority** and a **maximum latency**. Recent deliveries are listed with their **delivery status**: `SUCCESS`, `BUFFERING`, `SENDING`, or one of the failure reasons such as `FAILURE_UE_NOT_REACHABLE`. You can cancel a queued or buffered delivery. This is how a platform pushes a command or a firmware nudge to a constrained device that has no IP address.

NIDD Sessions

The **live T6a NIDD sessions** are the active Non-IP Data Delivery bindings the SCEF holds with the MME right now. Unlike the NIDD configurations, which are T8 resources scoped to an SCS/AS ID, these are **server-global runtime state**: they are **read-only** and are not scoped by SCS/AS ID, so this tab shows every active session regardless of which application context you are viewing. Each row shows the device's **IMSI**, the **APN** the NIDD PDN connection uses, the **EPS Bearer Identity (EBI)**, the session **state** (**ACTIVE**, **IDLE**), the **T6a session id**, the serving **MME host** and **MME realm**, and the **configuration id** the session was set up against. It is the fastest way to see which devices actually have a live non-IP binding, and through which MME.

Logs

The **Logs** tab tails the SCEF's live log output for troubleshooting a single instance without leaving OmniWeb.

Monitoring dashboards

T8 request rates, event-report throughput and NIDD delivery counts for the SCEF are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- **MME**: the SCEF reaches it over **T6a** to set up NIDD PDN connections and to gather the monitoring events reported here. The MME host and realm each session uses are shown on the NIDD Sessions tab.
- **HSS**: holds the subscription data (External Identifiers, NIDD authorisation) the SCEF's monitoring and NIDD services depend on.
- **PGW-C / SMF** and **UPF**: carry the ordinary IP data plane. NIDD deliberately bypasses them to move small non-IP payloads over the control plane instead.
- [Operations Guide](#): back to the element index.

SGW-C: Serving Gateway (Control Plane)

The **SGW-C** is the control-plane half of the EPC Serving Gateway. It anchors each subscriber's bearers between the access network and the packet gateway: it terminates **S11** to the MME (session and bearer signalling), **S5/S8** to the PGW-C (the PDN connection), and programmes the user plane (SGW-U) that forwards traffic on **S1-U** to the eNodeB. In short, it is the node that holds the live bearer state for every attached subscriber and keeps the MME, PGW and user plane in step. OmniWeb surfaces the SGW-C's live sessions, session counts and logs; longer-run trends (throughput, session growth) are on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

The SGW-C is a per-instance element page reached from the sidebar when one runs in the site's inventory. Where several are deployed, it also provides a fleet view for aggregate status. All traffic goes through the OmniWeb backend proxy to the SGW-C's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

A single-screen health read: the **API status** of the SGW-C, the **active session count**, and a summary derived from the live sessions: the number of unique **MMEs** (distinct S11 peers), unique **PGWs** (distinct S5/S8 peers), and the **total bearers** currently anchored. It is the quickest way to confirm the SGW-C is up, how many sessions it holds, and how many MMEs and PGWs it is working with.

Sessions

Every anchored session, each row keyed by subscriber **IMSI** and **MSISDN** with its connection **status**. Opening a session shows its full bearer state: the **S11** control endpoints (MME-side and SGW-side F-TEIDs), each **PDN connection** (APN, default bearer, and the S5/S8 control F-TEIDs to the PGW), and each **bearer** (EBI, the S1-U and S5/S8 user-plane F-TEIDs, and its QoS). This is where you confirm a specific subscriber's bearers are established and see which MME and PGW are serving them.

Where several SGW-Cs are deployed, the same tab in the **fleet view** lists every session across all of them in one table, with the serving SGW-C shown against each row, so an operator can find a subscriber without knowing which SGW-C is anchoring them. Opening a row goes to that session on its own instance. An instance that cannot be reached is named above the table, and so is the per-instance row limit when one instance holds more sessions than the combined view pulls.

Logs

The **Logs** tab tails the SGW-C's live log output for troubleshooting a single gateway without leaving OmniWeb.

Monitoring dashboards

Throughput, session-count trends and other time-series for the SGW-C are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- **MME**: the control peer over **S11**; it drives session and bearer setup on the SGW-C.
- The **PGW-C** anchors the PDN connection over **S5/S8**.
- The **SGW-U / UPF** is the user plane the SGW-C programmes to forward traffic on S1-U.

SIM Card Tools: Filesystem, APDU & eSIM

OmniWeb does more than discover and present SIMs. It gives you a set of **low-level card tools** that reach through any reader on any agent to talk to the card directly. You can browse and edit its file system, send raw APDUs by hand, watch the live APDU trace of a remote-SIM session, and manage eSIM profiles on an eUICC. This page is the operations guide for those tools.

← Operations Guide

These tools sit next to the existing SIM features and reuse them. Start here for the bigger picture:

- **SIM Bank**: the inventory: which SIMs are in which readers across every agent. Every tool on this page targets a reader from that inventory, and the SIM Bank's per-slot actions deep-link straight into them.
- **SIM Images (SAIP Profile Packages)**: how to save a read-out file system as a portable image, and how to browse or diff saved images.
- **Remote SIM & Virtual SIM**: how to present a SIM (real or virtual) to a modem over the network through card emulators.
- **OTA SIM Management**: the same file-level editing, but it reaches a card **over the air** with secured (03.48 / SCP80) packets instead of a local reader.

All four tools below live in the sidebar under the **SIMs** group.

One reader, one consumer at a time

Only one thing at a time can drive a physical PC/SC reader. A tool on this page may need a reader that something else already holds: another operator who reads it, or a live remote-SIM banking session that presents it to a modem. The tool then shows **who holds it and why**, and offers an explicit **Take over**. There is no silent kick: when you take over, OmniWeb cancels the other session (and, for a banking session, disconnects that SIM↔cardem link). To edit or send is a control action gated behind the SIM-bank control permission. Without it you can view readers but not drive cards.



SIM Filesystem Explorer

Pick a reader and **read the whole file tree** of its card: the MF, DF.GSM / DF.TELECOM, ADF.USIM and ADF.ISIM and all their EFs, with every readable file decoded into named fields. A full read touches many files, so a progress bar shows the walk ("read X of Y files: EF...").

- **Search** the tree by file code (FID), short name, or description. A search for "Emergency Call Codes" finds EF.ECC, not just 6FB7.
- **PIN-protected files are flagged, never read**: the explorer will not burn PIN retries.
- **Edit an EF** (with the control permission): the detail pane offers a schema-driven form (decoded fields, typed) where available, and falls back to raw hex. A write is destructive and asks for confirmation. Where an EF needs authentication, use **Verify CHV/ADM** first. It tests a PIN/ADM key against the card (it reports remaining tries and never burns the last retry). Once it is proven, the explorer reuses it for the write. The SIM key store offers saved keys for a card's ICCID automatically.
- **Save image** snapshots the decoded tree as a named **SIM Image**. **Compare with image** diffs the live card against a saved one. **Saved**

images opens the image library. You can also open a saved image *as* a tree (`?image=<id>`) to browse a card you no longer have in a reader.

The reader must be free (not held by a remote-SIM session), or you can Take over as above.

APDU Terminal

The APDU Terminal is a **hand console for the card**: type a raw APDU in hex, send it to the card in a chosen reader, and see the response decoded. The decode shows the command's INS name, the status word and its meaning, and an expanded interpretation of any response data (an FCP TLV tree, decoded EF fields, or an AUTHENTICATE result). Canned buttons (SELECT MF, STATUS, READ BINARY, GET RESPONSE, ...) fill common commands. Input is whitespace/colon-tolerant and validated (whole bytes, at least CLA/INS/P1/P2).

Two modes:

Mode	Behaviour	Use for
Single-shot	Each APDU is self-contained; the card resets to the MF between sends.	One-off probes.
Session	Opens a persistent, exclusive card session so state (selection, authentication) persists across APDUs: a SELECT then a READ BINARY works, and a VERIFY shares its auth with a later UPDATE. The reader stays held until you close the session.	Stateful sequences.

The SIM Bank's per-slot **APDU** action deep-links here with that exact reader preselected. Same one-consumer/Take-over rules apply.

APDU Log

Where the terminal is you driving one card, the **APDU Log** is a passive **live trace** of the command/response APDUs exchanged between a **modem and a (remote) SIM** through the remote-SIM / cardem chain. Turn on **capture** and it tails the exchange: each row pairs a `modem→card` command with its decoded status word, plus the same response decoding as the terminal (FCP tree, EF fields, ATR/reset events). You can pause the view (capture keeps running), filter by command/hex/SW, and auto-scroll.

Use it to see what a modem actually asks the SIM during an attach or an **IR.38 test** run: which files it selects, where a VERIFY fails, whether an AUTHENTICATE succeeds. You do not have to instrument the modem. It pairs naturally with **Remote SIM**, which sets up the modem↔SIM link the log then traces.

eSIM (LPA)

The **eSIM** tool manages an **eUICC** through the agent's **Local Profile Assistant (LPA)**. An eUICC is a physical eSIM chip in one of your readers. It holds several profiles, but only one is active at a time.

- **eUICC card**: shows the chip's **EID**.
- **Installed profiles**: every profile on the card: its state (active/disabled), nickname, provider, name, and ICCID. Per profile you can **enable** (make it the active eSIM), **disable**, **rename** (nickname), or **delete**. A delete removes the profile from the eUICC and is irreversible, so it confirms first.
- **Download a profile**: pull a new profile from an operator's SM-DP+ and install it onto the card. Use a whole **activation code** (`LPA:1$sm_dp.example.com$MATCHING-ID`) or a split **SM-DP+ address + Matching ID**, with an optional confirmation code.

This is for eSIM (eUICC) cards specifically. For classic SIM/UICC file editing use the Filesystem Explorer above. To edit a card **remotely** use **OTA**.

SIM Toolkit (STK)

The **SIM Toolkit** client drives STK applets on a SIM card. It sends a terminal profile to the card, then presents each proactive command as an interactive screen: menus, text display, single-key entry, and text input. Use it to navigate operator-resident services, trigger OTA configuration flows, or test SIM-resident applet logic.

See **SIM Toolkit** for the full operations guide.

Related

- **SIM Bank**: the reader/SIM inventory these tools target.
- **SIM Images**: save, browse and diff read-out file systems.
- **Remote SIM & Virtual SIM**: present a SIM to a modem over the network (the link the APDU Log traces).
- **OTA SIM Management**: the same file editing, over the air.
- **SIM Toolkit**: interact with STK applets on a card in a reader.

SIM Images: SAIP Profile Packages

A **SIM Image** is a saved snapshot of a card's whole file system. OmniWeb stores each image as a **SAIP Unprotected Profile Package (UPP)** in DER encoding. This is the same interoperable format eUICC profiles use, defined by the TCA *eUICC Profile Package: Interoperable Format Technical Specification*. One captured card becomes one portable `.der` file that any SAIP-aware tool can read.

This page is the operations guide. It covers how to capture, export, upload, browse and diff SIM images, and how to use a saved image to back a **virtual SIM**. **SIM Bank** covers the live reader inventory.

Table of Contents

- [Why SAIP UPP](#)
- [Architecture Overview](#)
- [What is in an Image](#)
- [Capturing an Image](#)
- [Exporting an Image](#)
- [Uploading an Image](#)
- [Browsing an Image in the Explorer](#)
- [Diffing Two Images](#)
- [Backing a Virtual SIM with an Image](#)
- [Profile Element Reference](#)
- [Troubleshooting](#)

Why SAIP UPP

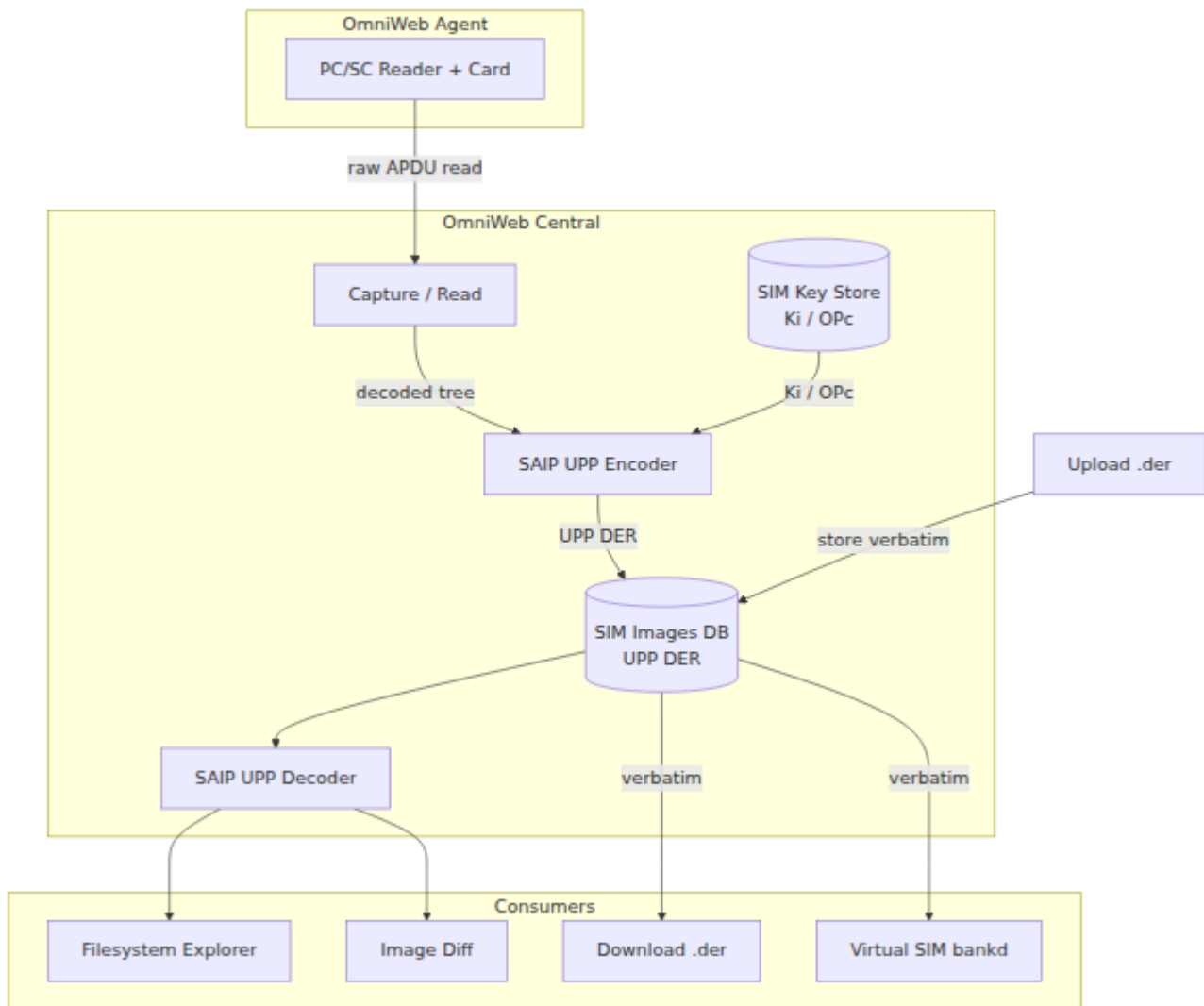
Earlier builds stored a SIM image as a decoded JSON tree used only for diffing. That format was OmniWeb-private and no external tool could consume it.

When OmniWeb stores the image as a **SAIP UPP** instead, you get:

- **Portability:** the exported `.der` is a standard profile package. It loads in SAIP profile validators and any SAIP-compliant toolchain.
- **One format everywhere:** the same `.der` an operator downloads is what backs a virtual SIM and what the file-system explorer renders. There is no second, divergent representation.
- **Reprovisioning:** when the card's authentication keys are known (see [Capturing an Image](#)), the UPP includes a `PE-AKAPparameter` element that carries Ki and OPc. The package then describes a complete, installable profile rather than a read-only dump.

The package is a raw concatenation of DER-encoded Profile Elements. It is **not** encrypted (it is an *Unprotected* Profile Package). Treat an exported `.der` with the same care as any file holding Ki/OPc. See [Troubleshooting](#).

Architecture Overview



Capture reads the card, the encoder produces a UPP (it folds in Ki/OPc from the SIM Key Store if available), and OmniWeb stores the package. Every consumer reads back from that single stored package. The explorer and diff decode it to a tree on demand. Download and the virtual-SIM bankd use the bytes verbatim.

What is in an Image

A captured image holds, for the card's standard MF / ADF.USIM layout:

Element	SAIP Profile Element	Contents
File system	PE - GenericFileManagement	Every reachable EF: its File Control Parameters (FCP) and raw content (transparent EFs) or every record (linear/cyclic EFs)
Identity	PE -Header	The card's ICCID
Authentication	PE -AKAPParameter	Ki and OPc for Milenage. Present only when the card's keys are in the SIM Key Store

Files are created with `PE-GenericFileManagement` (one operation group per DF) rather than the standardised SAIP file-system templates: a card read recovers raw bytes, not the template a profile creator originally used, so the generic representation is always exact.

Capturing an Image

Capture happens from the **SIM Filesystem** explorer or the SIM Bank slot actions:

1. Read a card's file system in the explorer.
2. Choose **Save image** and give it a label.

The decoded file system is encoded to a UPP and stored. If a **SIM Key Store** entry exists for the card's ICCID with both **Ki** and **OPc**, those keys are written into a `PE-AKAPParameter` so the resulting package is fully installable. Without a key-store entry the package still captures the complete file system; only the authentication element is omitted.

Identity columns (ICCID, IMSI, card type, file count) are derived from the file system at capture time so the image list and diff header never have to re-decode the package.

Exporting an Image

Each row in the saved-images list has a **download** action that serves the stored package verbatim as `UPP_<iccid>.der`.

The downloaded `.der` is a standard SAIP UPP and can be inspected with any SAIP-compliant profile tool.

How it works: The file is the exact bytes stored at capture (or upload), with no re-encoding on download, so a round-trip download is byte-identical to what is stored.

Uploading an Image

The saved-images page has an **Upload .der** action that accepts a SAIP UPP file and stores it as a new image. Use it to bring in profiles generated elsewhere: for example the GSMA TS.48 *Generic eUICC Test Profile* packages, or a UPP produced by another SAIP tool.

On upload, OmniWeb:

1. Validates the file begins with a SAIP Profile Element tag (a context-class constructed tag: `A0`, `A1` or `B0`).
2. Stores the package verbatim.
3. Decodes the `PE-Header` to populate the ICCID column where present.

An uploaded image is a first-class image: it can be downloaded, browsed in the explorer, diffed, edited, and used to back a virtual SIM exactly like a captured one.

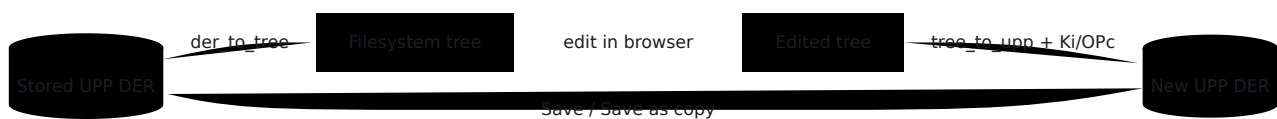
Editing an Image

A saved image is a **file**, not a card. To edit one changes the stored profile package only and **never writes to any SIM**. From the saved-images list, the **Edit filesystem** action opens the **SIM Filesystem** explorer in edit mode (`?image=<id>&edit=1`). No reader and no agent card access are involved.

You can:

- **Edit a file's content:** change any EF's bytes. As with a live card, the editor offers a schema-driven **Form**, a whole-record **JSON** view, and a **Raw hex** escape hatch. There is no PIN/ADM prompt (a file has no access conditions to satisfy).
- **Add a file:** create a new EF (transparent or record-structured) or DF under any DF/MF/ADF, and give its FID, structure and initial content.
- **Delete a file:** remove any EF or DF (the MF root is the whole image and cannot be deleted).
- **Rename** the image (metadata only) from the list.

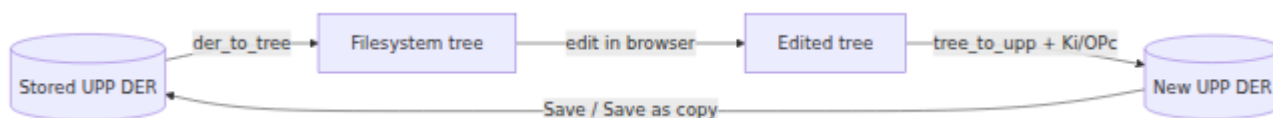
The browser holds changes until you **Save image** (overwrite in place) or **Save as copy** (fork a new image and leave the original untouched). On save OmniWeb re-encodes the edited file-system tree to a SAIP UPP. It folds in Ki/OPc from the **SIM Key Store** when known. It preserves verbatim any Profile Elements the tree does not model (PIN, PUK, Security Domain, templates), so an edit never drops the rest of the profile.



Form / JSON field editing offline. To decode an EF into named fields and re-encode them uses the same file model the live editor uses, run on any online agent, but with **no card**: the image already holds the bytes, so the agent only runs the codec. If no agent is online, editing falls back to Raw hex, which needs nothing external.

Browsing an Image in the Explorer

Every saved image has a **Browse in filesystem explorer** action. It opens the **SIM Filesystem** explorer against the stored package. **No physical card and no reader are needed.** OmniWeb decodes the package back into the same file-system tree the explorer renders for a live card, so you can inspect every EF's raw content offline.



Diffing Two Images

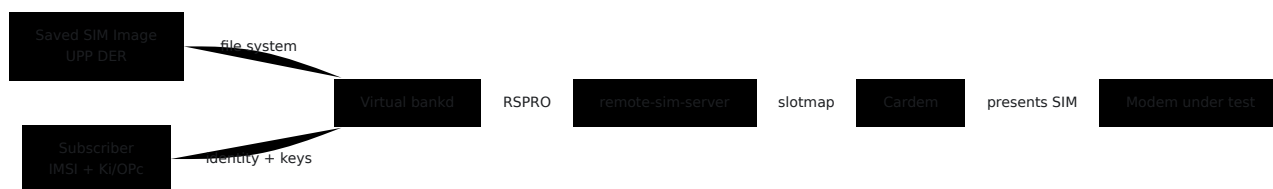
The saved-images page lets you pick two images and diff them. Both packages are decoded to file-system trees and aligned by EF path, so the comparison is file-by-file regardless of capture order:

Status	Meaning
same	EF present in both with identical content
changed	EF present in both, content differs
only_a	EF present only in the first image
only_b	EF present only in the second image

For a **changed** EF the diff reports the differing byte ranges (and per-record deltas for record-structured files), so you can see exactly which bytes moved: for example a test SIM before and after a personalisation change, or two operators' SIMs side by side.

Backing a Virtual SIM with an Image

A saved image can stand in for a physical card when presenting a **virtual SIM**. In the virtual-SIM composer, saved images appear alongside the captured rspro images as selectable file-system sources. Pair one with a subscriber identity (from the HSS or the SIM Key Store) and OmniWeb launches a virtual **bankd** backed by that image's file system, with the subscriber's Ki/OPc overlaid for authentication.



The file system the modem reads is replayed from the image; the identity and keys are overlaid from the chosen subscriber, so one image can back many subscribers.

Profile Element Reference

The UPP is a sequence of Profile Elements, each a DER TLV whose context-class tag identifies its type. OmniWeb writes and reads the subset below; other elements in an uploaded package (PIN, PUK, Security Domain, file-system templates) are preserved on storage and skipped when decoding to a file-system tree.

Tag	Profile Element	Used by OmniWeb
A0	PE-Header	ICCID; profile metadata
A1	PE- GenericFileManagement	The file system (FCP + content)
A4	PE-AKAPParameter	Ki and OPc (Milenage)
AA	PE-End	End-of-package marker
B0	PE-MF	(Templated MF: read on upload, not emitted)

For the full element catalogue and encoding rules, see the [TCA eUICC Profile Package: Interoperable Format Technical Specification](#).

Troubleshooting

Exported image has no authentication keys

Symptoms: A downloaded `.der` has no `PE-AKAParameter`; a virtual SIM backed by the image cannot authenticate without keys supplied separately.

Possible causes:

- No SIM Key Store entry existed for the card's ICCID at capture time.
- The key-store entry was missing Ki or OPc.

Resolution:

1. Add the card's Ki and OPc to the SIM Key Store, keyed by its ICCID.
2. Re-capture the image. The new capture folds the keys into a `PE-AKAParameter`.
3. Alternatively, supply the keys at virtual-SIM presentation time (from the HSS or an explicit override).

Upload rejected as "does not look like a SAIP UPP"

Symptoms: Uploading a file returns a validation error.

Possible causes:

- The file is not a SAIP UPP: it does not begin with a Profile Element tag (`A0`, `A1` or `B0`).
- The file is a *protected* profile package (encrypted), not an *unprotected* one.

Resolution:

1. Confirm the file is an **Unprotected** Profile Package in DER encoding.
2. Verify it begins with a `PE-Header` (`A0`). Most well-formed UPPs do.
3. If the package is protected/encrypted, it cannot be stored or browsed; only unprotected packages are supported.

Browsing an uploaded image shows files by FID, not by name

Symptoms: EFs in the explorer are labelled by file identifier (e.g. EF.6FXX) rather than a friendly name.

Possible causes:

- A card read recovers file identifiers and raw content, not the symbolic names a profile creator used. OmniWeb maps the well-known USIM EFs to friendly names; any EF outside that map is shown by its FID.

Resolution: No action is needed. The FID is the authoritative identifier and the raw content is exact. This is expected for non-standard or vendor-specific files.

Handling exported packages safely

An exported UPP is **unencrypted** and, when keys were available at capture, contains Ki and OPc. Treat a downloaded .der as sensitive key material:

- Do not send it over untrusted channels.
- Delete local copies when no longer needed.
- Prefer keeping images inside OmniWeb (where access is gated by the SIM Bank permission) over distributing .der files.

SIM Bank: distributed SIM inventory

The **SIM Bank** shows every SIM in every smart-card reader across every OmniWeb agent. Each agent scans its local PC/SC readers, reads each SIM's identity, and reports the inventory to central exactly like it already reports modems/phones as devices. Central aggregates the inventory per agent and persists a database of SIMs keyed by ICCID, so the "what SIMs are where" view survives reader and agent disconnects.

This is the inventory half of a distributed remote-SIM testbed. The remote-SIM card-emulation/banking layer lets you **connect any SIM to any cardem over the network**. [Remote SIM](#) documents that layer separately. This page covers discovery and inventory. [SIM Images \(SAIP Profile Packages\)](#) covers how to save, export and reuse a card's file system as a portable profile package.

The page is **reader-centric**: one row per physical reader slot, grouped by agent. Empty, unreadable and in-use slots all show. The bank layout is the truth, not just the populated slots.

How the agent reads a SIM

The agent scanner mirrors the modem scanner: enumeration is slow, so results are cached and rescanned on an interval.

For each PC/SC reader on the host:

1. **Enumerate** the PC/SC readers attached to the host.
2. **Probe** each reader with an *exclusive* connect.
 - No card → `present=false`.
 - Connect raises a **sharing violation** → the reader is held by another process (e.g. a remote-SIM `bankd` or a card-emulation session) → `present=true, in_use=true`, and **no read is attempted**. A busy reader never crashes the scan.
 - Card present and free → read it.
3. **Read** the SIM identity with a self-contained set of raw APDUs:
 - UICC-vs-classic SIM detection (select `3F00`; `6E00` → classic GSM, so use `CLA=A0` and classic selection mode),
 - EF.ICCID (`2FE2`), EF.IMSI (`6F07`), EF.SPN (`6F46`),
 - EF.AD (`6FAD`) for the MNC length (so MCC/MNC are split correctly),
 - a **safe** default-PIN (`0000`) VERIFY on a locked card. It probes the retry counter first (a non-decrementing empty VERIFY) and **refuses** if only one retry remains, so it **never** burns the last retry or touches the PUK (a SIM bank must not brick cards),
 - the card ATR.

Each reader becomes a dict:

```
{
  "reader_name": "Identiv SCR35xx USB Smart Card Reader ...",
  "reader_index": 0,
  "present": true,
  "in_use": false,
  "iccid": "8961...", "imsi": "505...", "mcc": "505", "mnc": "01",
  "country": "Australia", "spn": "Telstra", "atr": "3b9f96..."
}
```

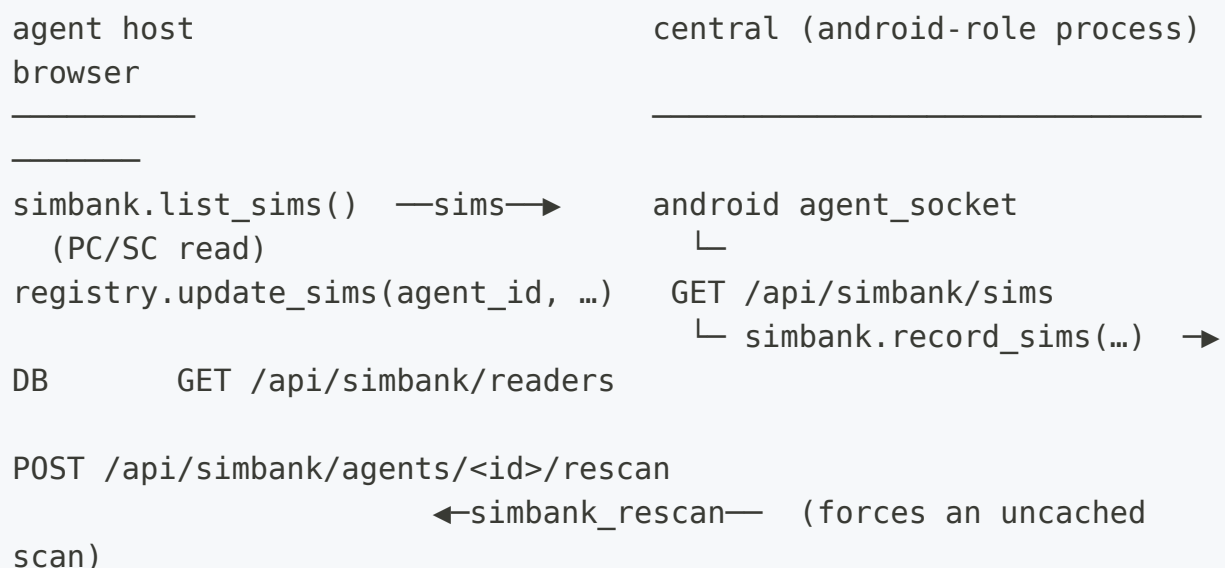
The list is sent to central as a `sims` message (alongside the device list) on register and whenever it changes; a `simbank_rescan` command forces an immediate uncached scan.

Card-reading capability

A host reads SIMs through its PC/SC smart-card stack. Where deeper card decoding is needed, the agent uses an optional, separately-licensed smart-card library kept isolated from the rest of the agent (it is not merged into OmniWeb's own code). A host **without** the smart-card stack still runs the agent fine and simply reports no SIMs. The SIM bank is an optional per-host capability, like modems.

The reference reader is an SCM/Identiv SCR35xx. Multiple readers per host are supported.

Inventory → central → UI flow



- The agent dials **out** to central over the same WebSocket it uses for phones (`/api/android/agent`). SIM inventory rides that connection. There is no new transport.

- Central holds the **live** per-agent reader snapshot in the in-memory `android.Registry` (`update_sims` / `all_sims`), and mirrors each ICCID into the persistent `simbank_sims` table via `simbank.record_sims`.
- Because this depends on the single-process agent Registry, the SIM-bank API is served **only by the agent-role backend process** (same constraint as Test Devices).

REST API (`/api/simbank`, JWT, RBAC element type `simbank`)

Method	Route	Purpose
GET	<code>/sims</code>	Every SIM across all agents (incl. offline persisted-only), with live present/in-use overlaid.
GET	<code>/readers</code>	Every PC/SC reader on every connected agent (live), incl. empty and in-use readers.
GET	<code>/sims/<iccid></code>	One SIM.
PATCH	<code>/sims/<iccid></code>	Edit operator-assigned fields (notes today). Creates the row if the ICCID is unseen.
DELETE	<code>/sims/<iccid></code>	Forget a SIM (drops the persisted record; reappears on next scan if still inserted).
POST	<code>/agents/<id>/rescan</code>	Force an immediate uncached PC/SC rescan on one agent and return its fresh inventory.

RBAC mirrors `ir38/android-control`: admins always pass; otherwise a JWT permission with `element_type == "simbank"` grants view, and `POST` in its methods grants control.

Frontend

`src/pages/simbank/SimBank.tsx`: a searchable, filterable table of all SIMs: status (present / in use / offline), agent/site, reader, ICCID, IMSI, MCC/MNC, country, SPN, editable notes, last seen. Per-agent **Rescan** button. Routed at `/simbank` and linked from the sidebar under **Roaming Tests** > **SIM Bank**.

Database schema

`simbank_sims`: one row per ICCID ever seen:

Column	Notes
<code>iccid</code> (PK)	SIM serial.
<code>imsi</code> , <code>mcc</code> , <code>mnc</code> , <code>country</code> , <code>spn</code> , <code>atr</code>	Last-known identity (only overwritten by non-empty reads).
<code>notes</code>	Operator-assigned, editable.
<code>last_agent_id</code> , <code>last_site</code> , <code>last_reader_name</code> , <code>last_reader_index</code>	Where it was last (or is currently) seen.
<code>first_seen</code> , <code>last_seen</code>	Timestamps (UTC).
<code>remsim_bank_id</code> , <code>remsim_slot_nr</code> , <code>remsim_mapped_cardem</code>	Phase-2 reserved (nullable, unused).

Empty and in-use readers carry **no** ICCID, so they are not persisted as SIMs. They only appear in the live `/readers` snapshot.

Future: Remote SIM

Phase 2 maps a physical reader/SIM into a remote-SIM bank so a SIM in one site can be presented to a modem/phone in another (remote SIM). The hooks are already in place. Search the code for `# TODO remsim`:

- `simbank_sims.remsim_bank_id / remsim_slot_nr / remsim_mapped_cardem` (nullable columns, surfaced in `to_dict` and the TS `SimRow` type as `null`),
- the agent's per-reader `in_use` flag already distinguishes a reader held by a `bankd`/emulation session. That is the reader phase 2 will map,
- `PATCH /api/simbank/sims/<iccid>` is where editing the remsim mapping will be wired.

Nothing here drives remsim today; the shape is reserved so the mapping slots in without reworking the inventory hot path.

Site Editor: Site Configuration Viewer

The **Site Editor** shows the raw configuration file for the currently selected site: the YAML host/inventory file that defines that deployment's network functions, infrastructure, and settings. It is a read-only, syntax-highlighted view, so you can see exactly what OmniWeb is working from for a site without opening a shell on a host.

[← Operations Guide](#)

What it shows

OmniWeb drives every site from a per-site configuration file. That file lists the site's elements (which NFs are present and at what addresses), its infrastructure hosts, its monitoring role, its `fivegc` block (see [5G Core](#)), and so on: everything the sidebar, topology, element pages, and dashboards derive from.

The Site Editor loads that file for the site chosen in the top-bar site dropdown and displays it in a full editor pane (Monaco, the same editor VS Code uses) with YAML highlighting, line numbers, folding, and a minimap. The header shows the customer, the site, and the source filename.

```
Parse error on line 3: ... SE -->|/api/sites/{customer}/{site}/ra -----
--^ Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND',
'SUBROUTINEEND', 'PIPE', 'CYLINDEREND', 'DIAMOND_STOP', 'TAGEND',
'TRAPEND', 'INVTRAPEND', 'UNICODE_TEXT', 'TEXT', 'TAGSTART', got
'DIAMOND_START'
```

Try again

If no site is selected, the page prompts you to pick one from the dropdown first.

Read-only by design

The Site Editor is a **viewer**: the pane is read-only (marked with a **read-only** chip in the header). It is there to *inspect* the effective configuration, not to change it.

Site configuration is not edited from the browser. As noted throughout this guide, OmniWeb deployment and configuration is managed via **Ansible**. Do not edit configuration files directly on hosts, and adding or changing an element is a configuration change made through that pipeline (see [the configuration model](#)). The Site Editor lets you read the result of that pipeline for the selected site.

When to use it

- **Confirm what a site is configured with:** which elements, hosts, and 5GC NFs OmniWeb thinks are present, and at which addresses.
- **Troubleshoot the UI against the config:** if an element, a sidebar entry, or a 5GC section is missing or pointing at the wrong host, the raw file is the source of truth to check against.
- **Reference the exact filename and contents** when preparing an Ansible change.

Related

- [Architecture: the configuration model](#): how site config drives OmniWeb and how it is managed.
- [5G Core](#): the `fivegc` block of the site config and what it surfaces.
- [Network Topology](#): the topology and Host List views built from the same site configuration.

SMPP Gateway

The **SMPP Gateway** bridges external SMPP peers to the messaging core. External applications and interworking partners, the **ESMEs** (External Short Message Entities), connect over SMPP either as clients the gateway dials out to or as servers the gateway accepts binds from, and the gateway carries their short messages to and from the SMSC. Each connection is an SMPP **bind** (transceiver, transmitter or receiver). The gateway keeps those binds up, authenticates and polices them, and hands traffic on to the messaging core. OmniWeb surfaces the live bind status, the configured client and server peers, and the gateway's logs.

[← Operations Guide](#)

The SMPP Gateway is a per-instance element page reached from the sidebar when one runs in the site's inventory, with a fleet view where several are deployed. All traffic goes through the OmniWeb backend proxy to the gateway's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

A single-screen health read of the aggregate bind statistics: the **total** number of binds and the **connected / disconnected** breakdown split across **client** (outbound) and **server** (inbound) binds. It is the quickest way to confirm the gateway is up and that its peers are bound, and to spot at a glance if any expected connection has dropped.

Bind Status

Every enabled bind, client and server, with its current **connection status**, the underlying **session details** and per-bind **metrics**. This is the live view of who is bound right now. From here an operator can act on a bind:

- **Drop** an active bind. For a client bind this terminates the outbound connection. For a server bind it disconnects all active client connections. Use it to clear a stuck or misbehaving connection.
- **Reconnect** a client bind. This forces a client (outbound) bind to redial its remote host, which is the quickest way to bring an outbound peer back after a network blip.

Client Peers

The configured outbound SMPP **client** connections are the peers the gateway dials out to. Each shows its **name**, the remote **host** and **port**, the **system ID** it binds with, and the **bind type** (transceiver, transmitter or receiver). Client peers are fully editable here: an operator can **add** a peer, **edit** an existing one (host, port, credentials, bind type, enquire-link interval, TPS limit and the other bind parameters), and **delete** one that is no longer needed. Enabling a peer brings its outbound bind into service. Disabling it takes the peer out without discarding its configuration.

Server Peers

The configured inbound SMPP **server** authorisations are the credentials and rules for peers that bind *into* the gateway. Each entry defines a **system ID** and password, the **allowed bind types**, and the guard rails that police an inbound bind: an **IP whitelist** of permitted source addresses and a **source-address whitelist** of permitted originating-address patterns, plus a **TPS limit**. Server peers are editable here in the same way (**add**, **edit** and **delete**), so an operator can onboard a new inbound partner, tighten its whitelists, or revoke it. Enabling and disabling an authorisation controls whether that partner is allowed to bind.

Logs

The **Logs** tab tails the gateway's live log output to troubleshoot a single gateway (bind failures, authentication rejects and dropped connections) without leaving OmniWeb.

Monitoring

Message rates, throughput and other time-series for the SMPP Gateway are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- The **SMSC** is the messaging core the gateway hands short messages to and from.
- The **IP-SM-GW** carries SMS over IMS for VoLTE devices, and complements the SMPP path to external peers.

SMSC: Short Message Service Centre

The **SMSC** (Short Message Service Centre) is the store-and-forward heart of SMS in the core. It **accepts** messages from the network (over SS7 (MAP) from the mobile side, over Diameter, and over SMPP from application front-ends). It then **routes**, **translates** and **delivers** them. It holds any message it cannot deliver immediately and retries until the message is delivered, expires or is dropped. Along the way it tracks the delivery state of every message, translates calling and called numbers, federates with peer SMSCs so a message can be handed to whichever node can reach the recipient, and drives bulk SMS **campaigns**. OmniWeb surfaces all of this from a single per-instance page: live message state, subscriber locations, front-end registrations, Diameter and federation health, the routing and translation rule sets, and campaign management.

[← Operations Guide](#)

The SMSC is a per-instance element page reached from the sidebar when one runs in the site's inventory, with a fleet view where several are deployed. All traffic goes through the OmniWeb backend proxy to the SMSC's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

The landing tab is a single-screen health read for the node: current message counts and delivery state, front-end and peer health at a glance, and a **search by MSISDN** that looks across both stored messages and subscriber locations. It is the quickest way to answer "what is happening to this subscriber's SMS right now" from one box.

Messages

Every message the SMSC is holding or has processed, each row showing the sending and receiving numbers, the message **state** (pending, delivered, failed, expired, dropped or auto-replied), the number of **delivery attempts**, and its age. You can filter messages by state, by **SMSC**, and to include **dead-letter** or **unrouted** traffic, and search by MSISDN, node name or message ID. Select a message to open its detail, including the decoded plaintext. An operator can **reroute** a message to force it back through routing, for example after fixing a routing rule.

Frontends

The application and SMPP **front-ends** currently registered with the SMSC are the clients that submit and receive messages. Each entry shows its hostname, when it was **last seen**, and when its registration **expires**, so you can confirm the message-handling front-ends are connected and their registrations are live.

Locations

The **subscriber locations** the SMSC has learned are the last known serving location for each subscriber by **IMSI** and **MSISDN**. The SMSC uses this information to reach a recipient. You can search it by IMSI or MSISDN and filter it to include expired entries. It is where you confirm the SMSC knows where a subscriber is before it can deliver to them.

Diameter

The Diameter peer connections the SMSC uses for messaging, each row showing the peer's **realm**, **IP address**, **transport**, connection-initiation direction, reported **product name**, negotiated **application IDs** and **connection state**. This is where you confirm the Diameter path the SMSC depends on is up.

Federation

The **federation** view shows how this SMSC cooperates with peer SMSCs so a message can be handed to whichever node can reach the recipient. It reports **this node**, the count of **healthy**, **degraded** and **unhealthy** peers, the number of **cached federated messages** (highlighted when non-zero), the **discovery** method in use (DNS domain or static peers), and a per-peer table of node name, status, recent failures, last successful health check and the front-ends each peer offers. It is the fastest way to see whether the SMSCs in a federation reach each other.

Routing

The **SMS routing rules** that decide where each message goes, listed by **priority**. Each rule matches on a **calling** and **called** number pattern (and optionally a source SMSC or source type such as SS7 or circuit-switched), and resolves to a **destination** (a target SMSC or an **ENUM** lookup) with a description and an enabled state. Rules are fully editable: an operator can add, edit, enable, disable or delete them, and export and re-import the rule set.

Translations

The **number-translation rules** that rewrite calling and called numbers as messages pass through: they normalise formats, apply prefixes and the like. Each rule is listed by **priority** with its calling and called match, optional source SMSC, a **continue** flag (so several rules can apply in sequence), a description and an enabled state. As with routing, the rules are editable (add, edit, enable, disable, delete), and you can export and re-import them.

Campaigns

Bulk SMS **campaigns** are created and driven from this tab. A **New Campaign** is defined with a name and description, a **sender** number, the source SMSC, a **recipient list**, an **audience** (active recipients only or everyone in the list), a **drip** rate in messages per second, a message **validity** window, an optional **deliver-after** time and the **message template**. Once created, a campaign is controlled from here (**start**, **pause**, **resume** or **cancel**). You can inspect its **targets** to see the per-recipient send and delivery state as the campaign progresses.

Campaign Lists

The recipient **lists** that campaigns send to. Lists are fully managed here: create a **New Recipient List** with a name, description and an initial set of recipients, **append** more recipients later, edit, or delete a list. You supply recipients as **CSV**, so you can import an existing contact list directly rather than enter it by hand.

NAPTR Test

The **NAPTR Test** tab is an operator tool that resolves an **E.164 number (or domain) through ENUM/NAPTR DNS** to see how the SMSC will route a message towards it. Enter the phone number and the **ENUM domain** to query (`e164.arpa` by default) and submit. Nothing is fetched until you do. The SMSC reverses the number's digits into the ENUM zone, looks up the **NAPTR** records published there, and returns them in **order** and **preference**. Each record carries its **flags**, **service**, the rewrite **regexp** and **replacement**, and the resolved **result domain** the record points the message at. It is where you confirm the ENUM data behind a routing rule really resolves a number to the URI or route you expect, before you rely on it in production.

MWI

The **MWI** tab sends a **Message Waiting Indication** update for a subscriber: the notification that sets or clears the "waiting messages" flag on a handset, typically after a voicemail is left or collected. Give the **destination MSISDN** to notify and the **source MSISDN** it comes from, pick the **indication type** (voicemail, fax, email or other), and set whether the indicator should be **active** (on) or cleared (off), with an optional **message body** and originating **SMSC**. It is a submit form: the update is only sent when you post it. This is the direct way to raise or clear a subscriber's waiting-message indicator by hand.

System Logs

The **System Logs** tab is the SMSC's own API-backed log view. You can filter it by **level** (debug, info, warning, error) and by a **limit** on the number of lines returned, so you can pull just the recent errors or a fuller trace when you investigate a specific problem, without leaving OmniWeb.

More tools

Alongside the tabs above, the SMSC page offers a few focused tools:

- **Submit:** a form that sends a one-off SMS directly from OmniWeb (from, to and message text), to test delivery to a specific recipient.
- **Operations:** node-wide maintenance actions, including clearing the registered **front-ends**, stored **messages**, **routing** rules and **translation** rules. Each is a destructive action and asks for confirmation first.

Logs

The **Logs** tab tails the SMSC's live log output to troubleshoot a single node without leaving OmniWeb. For a filtered, API-backed view of the same node, use **System Logs** above.

Monitoring

Message throughput, delivery-success rates, queue depth and other time-series for the SMSC are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- **HSS**: holds the subscriber data and routing information the SMSC's deliveries resolve against.
- The **DRA**: relays the Diameter signalling the SMSC uses for messaging.
- **STP**: the SS7 signalling path over which the SMSC exchanges MAP messages with the mobile network.
- The **CSCF/TAS**: the IMS side that originates and terminates messaging for VoLTE subscribers.

Software Management & Lifecycle

OmniWeb provides a single place to see and manage the software running across the fleet: package versions, available updates, and licensing. The lifecycle of the network's software is visible from the same portal used to operate it.

[← Back to Operations Guide](#)

Software Page (APT Repository)

The **Software** page is an APT repository browser for Omnitouch packages. It connects to the configured APT repository server and shows, for the packages it manages:

- **Available packages** in the repository.
- **Versions** offered, including the latest available.
- **Update status**: whether a newer version exists than what is deployed.

This lets operators see at a glance which Omnitouch components have updates pending and which versions are current, providing the inventory side of the software lifecycle. Package installation/upgrade on hosts is performed through the standard APT/`dpkg` tooling (and is typically automated via Ansible); OmniWeb gives the visibility layer over what is published and current.

The Software page: each Omnitouch package with its latest repository and cache versions and an in-sync indicator that flags where a newer version is published than is cached/deployed (here `omni-upf` and `omni-msc` are out of sync).



Syncing Packages to the Cache

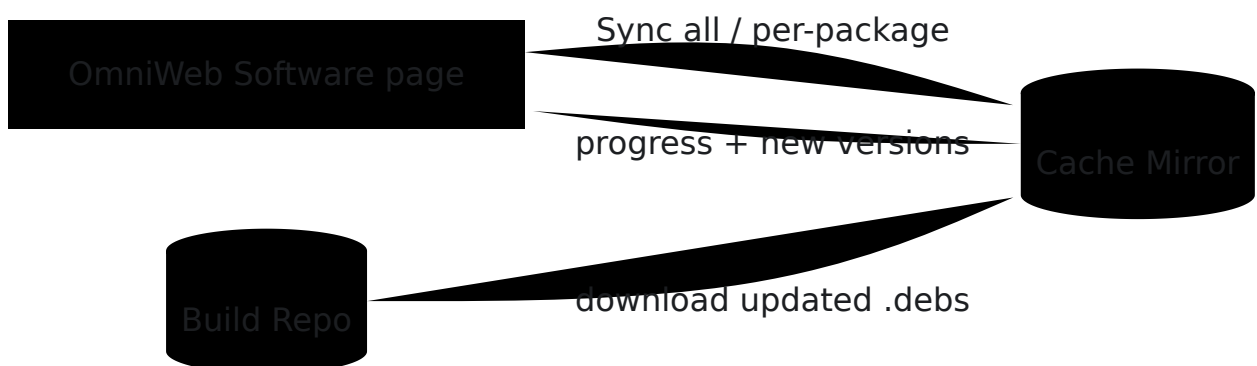
Deployments that run an on-site **APT cache mirror** (for airgapped or bandwidth-limited sites) keep a local copy of the packages published on the build repository. Over time the cache falls behind. The Software page flags each package that is **out of sync** (a newer version is published than is cached). From the same page you can pull those updates into the cache without a move away from OmniWeb:

- **Sync all:** pull every updated package from the repo into the cache.
- **Sync** (per row): pull just one package. Click several and they **queue**: they download one after another, and OmniWeb regenerates the package index

once at the end (not per package), so the cache is never left half-updated.

While a sync runs, the page shows live progress (which package is downloading, how many are queued, and a streaming log). It refreshes the version comparison when it finishes.

A sync in progress: `omni-msc` downloading with `omni-upf` queued, the phase banner, and the live log. When the queue drains, the cache index is regenerated once and the in-sync status updates.



The sync runs on the cache mirror itself (OmniWeb triggers it and streams the progress); the mirror pulls the updated `.debs` from the build repo and

regenerates its index. Hosts at the site then install the updated packages from the cache as usual.

Versioning

OmniWeb reports its own build version (version, commit, and build date) so operators can confirm exactly which build of the portal they are running. This is the reference point when reporting issues or coordinating an upgrade of the portal itself.

License Management

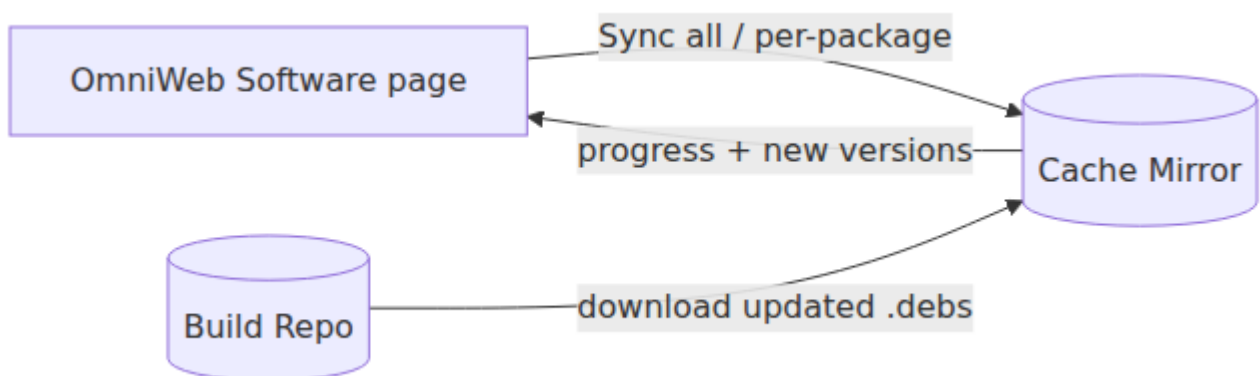
The **License Server** is administered through OmniWeb like any other element, using the same generic patterns (see [Common Operations](#)). It surfaces:

License Server overview: current license status and validity, including trusted-time state.

View	Description
Overview	Current license status, validity dates, and feature entitlements.
Request Logs	Historical license requests, with filtering. This is useful to audit what was checked and when.
Query	Look up the license status for a specific licensee/product on demand.

Licensing ties into the software lifecycle: entitlements determine which products and features a deployment is authorised to run, and the request logs give an audit trail of license activity over time.

Lifecycle at a Glance



1. **See what is available:** the Software page shows pending updates.
2. **Confirm entitlements:** the License Server shows what the deployment is licensed for.
3. **Apply:** upgrades run via APT/Ansible on the hosts.
4. **Verify:** confirm versions, then watch element **Logs** and **Dashboards** to confirm a healthy result.

SS7: Signalling Stack (STP / HLR / IP-SM-GW / CAMEL-GW)

The **SS7 stack** (OmniSS7) terminates SIGTRAN transport (SCTP associations carrying **M3UA**) and routes SS7/MAP traffic across the signalling network. A single service provides the whole stack, and OmniWeb presents it as four element views for its different roles: **STP** (Signalling Transfer Point, the SS7 router), **HLR** (Home Location Register), **IP-SM-GW** (IP Short Message Gateway) and **CAMEL-GW** (CAMEL Gateway). All four are backed by the same shared client, so the SIGTRAN transport and routing they share is the same underlying stack seen from different angles.

[← Operations Guide](#)

Each view is a per-instance element page reached from the sidebar when an SS7 stack runs in the site's inventory. All traffic goes through the OmniWeb backend proxy to the stack's API, so it stays behind the single authenticated gateway. Responses come back in a `{status, response}` envelope. The documentation link in each element page header (the in-context help button) opens this page. Each view is organised into tabs. Message rates and other time-series are presented as Grafana dashboards reached from the element page. OmniWeb does not duplicate those charts natively.

The stack at a glance

The SIGTRAN transport underneath every role is the same, so the **Peers** and **SCTP Connections** tabs appear in all four views and read the same data.

The **Peers** tab reports M3UA peer status (`/api/m3ua-status`): every M3UA/M2PA peer, with its **ASP state**, **association state**, local and remote **endpoints**, **routing context**, protocol and mode, and per-peer **uptime** statistics. This is where you confirm each signalling neighbour's application-server process is up and in service.

The **SCTP Connections** tab lists the underlying transport (`/api/sctp-connections`): every SCTP association with its **status**, **role**, local and remote **IP/port**, inbound and outbound **stream** counts, **bytes** and **messages** sent and received, **last activity**, and any last error, plus aggregate totals across all connections. When a peer shows down, check this layer to see whether the SCTP association itself is established.

STP: routing

The **STP** view is the SS7 router. Alongside the shared transport tabs it carries the **Routing** tab, plus **Network Map**, **Overview** and an **SS7 Events** feed.

The **Routing** tab shows the full M3UA routing configuration (`/api/routing`): the configured **peers**, the **point-code (PC) routes** and the **global-title (GT) routes**.

- **Point-code routes** direct traffic to a destination point code (`dest_pc`) via a target **peer**, with a **network indicator**, point-code **mask** and **priority**. Operators can **add** a PC route (`POST /api/routing/pc-routes`, giving the destination point code and target peer) and **delete** one (`DELETE /api/routing/pc-routes/{dest_pc}-{peer_id}`).
- **Global-title routes** match a GT **prefix** and forward to a target **peer** with a **priority** and optional SSN, translation-type and nature-of-address details and a description. Operators can **add** a GT route (`POST /api/routing/gt-routes`, giving the prefix and target peer) and **delete** one (`DELETE /api/routing/gt-routes/{prefix}:{peer_id}`).

This is where you confirm, and adjust, how the STP forwards SS7 messages toward each destination.

HLR

The **HLR** view surfaces the register's active subscribers and the link to the HLR back-end, alongside the shared transport tabs, an **Overview** and an **SS7 Client** tab.

The **Subscribers** tab lists the active subscriber register (`/api/subscribers`): every subscriber who has sent an **UpdateLocation** and not yet a **CancelLocation**, each row showing **IMSI**, **VLR**, serving **MSC**, registration time and **duration**, with counts of unique VLRs and MSCs. An operator can **clear** all tracked active subscribers (`DELETE /api/subscribers`) to reset the register's view.

The **HLR Links** tab checks connectivity to the configured HLR API (`/api/hlr-links`): the **status** (up, degraded or down), measured **latency**, reported **version** and the target **host**. It is the quickest way to confirm the stack can reach the HLR back-end.

IP-SM-GW

The **IP-SM-GW** view is the short-message gateway. It carries the SMSc subscriber tracker and the link to the SMSc back-end, alongside the shared transport tabs and an **Overview**.

The **SMSc Links** tab checks connectivity to the configured SMSc API (`/api/smsc-links`): **status** (up, degraded or down), **latency**, reported **version** and target **host**. It is the SMS-path equivalent of the HLR link check.

The **SMSc Subscribers** tab lists the subscribers tracked by the SMSc subscriber tracker (`/api/smsc-subscribers`): each with **MSISDN**, **IMSI**, **HLR GT** and **VLR GT**, **messages sent** and **received**, status and last update, plus active/failed and unique-HLR counts. An operator can **clear** all tracked SMSc subscribers (`DELETE /api/smsc-subscribers`).

CAMEL-GW

The **CAMEL-GW** view surfaces the CAMEL sessions the gateway is running, alongside the shared transport tabs and an **Overview**.

The **CAMEL Sessions** tab lists all active CAMEL sessions managed by the gateway (`/api/camel-sessions`): each session's **state**, **CAP version**, **service key**, **IMSI**, **calling** and **called** numbers, start time and **duration**, with counts of sessions initiated and answered. It is where you watch live CAMEL call handling in real time.

Status

The **Status** endpoint (`/api/status`) reports the overall node health that the **Overview** tabs summarise: the node identity and **role**, whether the service is **ok**, its **uptime**, and a peer summary of how many signalling peers are active out of the total. This is the first read to confirm the stack is running and its peers are in service.

Related

- **STP**: the SS7 routing element (point-code and global-title routes).
- **HLR**: the Home Location Register element (active subscriber register and HLR link).
- **IP-SM-GW**: the IP Short Message Gateway element (SMSc subscribers and link).
- **CAMEL-GW**: the CAMEL Gateway element (live CAMEL sessions).
- The **MSC/VLR** raises the UpdateLocation and CAMEL signalling this stack routes and tracks.
- The **SMSc** is the back-end the IP-SM-GW's SMSc link and subscriber tracker resolve against.
- [← Operations Guide](#)

SIM Toolkit (STK) Client

OmniWeb includes an interactive **SIM Toolkit** client. Use it to navigate and drive STK applets on any physical SIM card inserted in a connected agent's PC/SC reader. The STK client sends a terminal profile to the card, then presents each proactive command as an interactive screen. You respond as a terminal would, so the applet advances to its next step.

This tool sits in the sidebar under **SIMs - SIM Toolkit**, alongside the [APDU Terminal and SIM Filesystem Explorer](#).

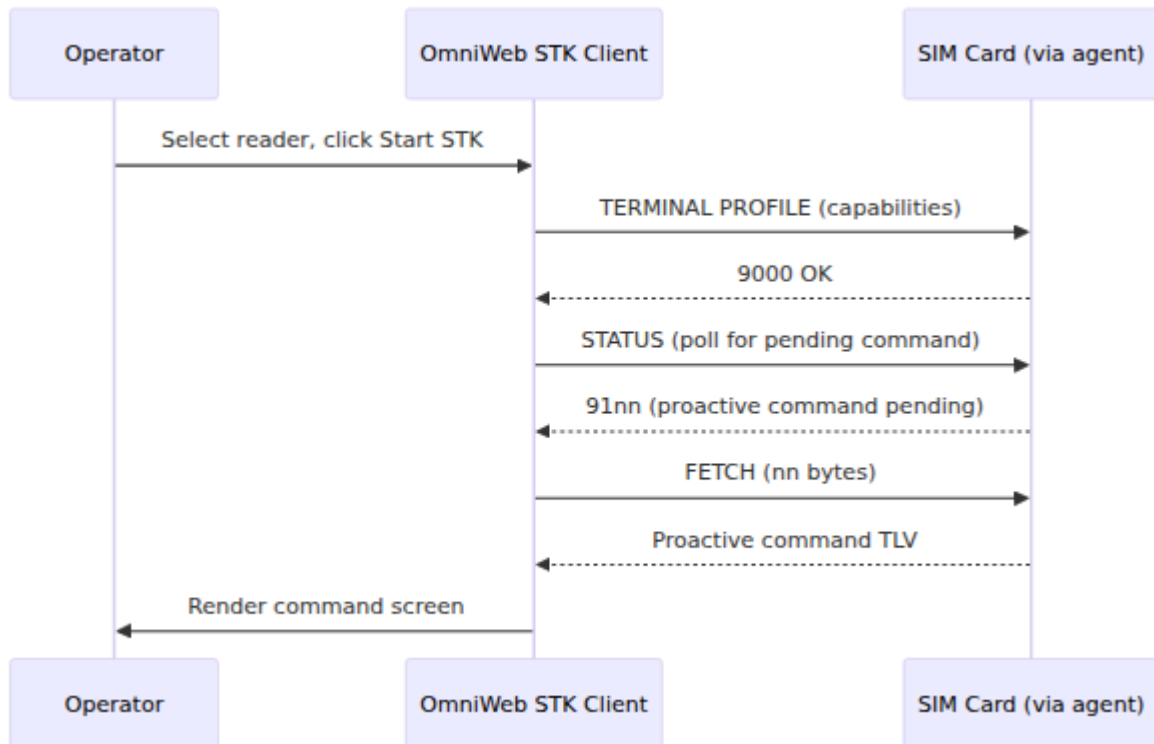
What is SIM Toolkit?

SIM Application Toolkit (STK) is a standard defined in [ETSI TS 102.223](#) that allows a SIM card to initiate actions on the terminal. A SIM applet issues **proactive commands** - requests such as "display this text" or "show this menu" - and the terminal responds with the user's selection or input. Common use cases include operator top-up menus, balance enquiry, OTA configuration triggers, and SIM-resident service portals.

Requirements

- A reader with a SIM card that contains an STK applet.
 - The **SIM bank control** permission. Without it, you can see readers but cannot open sessions or drive cards.
 - The reader must not be held by another tool. If it is, OmniWeb shows who holds it and offers a **Take over** option. Taking over cancels the other session.
-

Opening a Session



1. Select the reader from the **Reader** drop-down. Only readers with a card present appear.
2. Click **Start STK**. OmniWeb opens an exclusive session on the card, sends a terminal profile, and polls for the first proactive command.
3. Interact with each screen as it appears. OmniWeb sends a TERMINAL RESPONSE after each step and fetches the next command automatically.
4. The session ends when the card issues an END SESSION command, or when you click **End session**.

Command Screens

OmniWeb renders each proactive command as an interactive screen. The table below lists the supported command types.

Command	Description	User action
DISPLAY TEXT	The card shows a message.	Click OK to acknowledge, or Back to go to the previous step.
SELECT ITEM / SET UP MENU	The card presents a numbered list of options.	Click the item you want to select.
GET INKEY	The card asks for a single keystroke.	Type one character and click Send , or choose Yes / No for yes-no prompts.
GET INPUT	The card asks for a text string of a specified length.	Type the text and click Send . The field shows the minimum and maximum character count.

If the card issues a command type that the client does not render interactively (for example SEND SHORT MESSAGE or SET UP CALL), OmniWeb shows the raw command data and sends an **OK** response automatically so the session can continue.

Session History

A **Session history** panel appears on the right side of the screen once you start interacting. It records each command type and the response you gave (selected item, input text, or OK/Back). Use it to retrace the path through a menu tree.

Ending a Session

The session ends in one of three ways:

- **Card-initiated end:** the applet issues an END SESSION proactive command. OmniWeb closes the session automatically and shows a completion message.
- **Operator end:** click **End session** at any time. OmniWeb sends a session-terminated result to the card and closes the reader session.
- **Navigation backward past the root:** clicking **Back** on the first screen sends a backward-move result to the card. The card decides how to respond - it may show a previous screen or end the session.

After the session ends, the reader becomes available again for other tools.

Relation to Other SIM Tools

Tool	Purpose
SIM Toolkit (this page)	Drive STK applets interactively.
APDU Terminal	Send raw APDUs by hand, one at a time or in a persistent session.
SIM Filesystem Explorer	Browse and edit the card's file system.
APDU Log	Passively trace commands between a modem and a remote SIM.

Troubleshooting

Start STK fails with "TERMINAL PROFILE failed"

The card rejected the terminal profile or the agent could not reach the card. Check that:

- The card is seated correctly in the reader.
- No other tool holds the reader. If it does, use **Take over**.
- The agent that owns the reader is online (check the SIM Bank page).

The session stops after the first command

Some SIM applets end the session immediately if the terminal profile does not declare the capabilities the applet needs. The client declares a broad set of capabilities, but some cards check for specific bits. If this happens, use the [APDU Terminal](#) in session mode to send TERMINAL PROFILE and FETCH manually and examine the raw proactive command data.

Back returns an unexpected screen

The **Back** action sends result code 0x11 (backward move) to the card. The card controls what this means. Some applets return to the previous menu; others restart the session from the root menu; a few ignore backward move and re-issue the same command.

GET INPUT rejects my text

The card sets a minimum and maximum character count. The input field shows both. If the card also restricts to digits only, the field label says **digits only**. Enter a value within the required range.

TAS: Telephony Application Server

The **TAS** (Telephony Application Server) is the IMS application server that delivers the subscriber's telephony experience. It handles **call control** and **supplementary services** (call forwarding, barring, waiting, hold and the rest), runs **conferences**, takes and plays back **voicemail**, records **CDRs** for every call, resolves **routing** and number translation, and talks to the **OCS** over Diameter so that calls are charged in real time. Where the CSCF decides *how* a call is signalled, the TAS decides *what* the service does. OmniWeb surfaces its live calls, subscriber registrations, gateways, Diameter peers and configuration on one element page. Call-volume and trend charts live on the Grafana dashboards reached from the same page.

[← Operations Guide](#)

The TAS is a per-instance element page reached from the sidebar when one runs in the site's inventory. All traffic goes through the OmniWeb backend proxy to the TAS's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Overview

The landing tab is a single-screen health read for the application server: the number of **active calls**, **registered subscribers**, running **conferences** and stored **voicemail** records, the health of the **Diameter** peers and **SIP gateways**, and the **licence** status. It is the fastest way to confirm the TAS is up and carrying traffic.

Active Calls

Every call currently in progress, each row showing the calling and called parties, the **direction**, the call **state** and its **duration**. Select a call to open its detail so you can see how a specific call is legged and where it sits in the dialplan. It is the first place to look when a subscriber reports a call in trouble right now.

CDR

The **call detail records** the TAS has written: one row per completed call, with the parties, **start time**, **duration** and **disposition**. Open any record for its full detail. CDRs are the accounting and after-the-fact troubleshooting trail: use them to confirm a call happened, how long it ran and how it ended.

Subscribers

The subscriber **registrations** the TAS knows about, each with the subscriber **identity**, registration **state** and current **contact**. Open a subscriber for the full registration detail. This is where you confirm a subscriber is registered to the application server before chasing why a service is not working for them.

Diameter

The **Diameter peers** the TAS connects to, including the path to the **OCS** for charging and the **HSS** for user data, with each peer's hostname and **connection state**. Open a peer for its detail. When charging or subscriber lookups fail, this tab is where you confirm the underlying Diameter paths are up.

Gateways

The **SIP gateways** the TAS uses to reach other networks and trunks, each showing its name and current **status**. Open a gateway for its detail. Use this tab to confirm the interconnect gateways are registered and available before diagnosing failed off-net calls.

Conferences

The active **conferences** and their participants. Open a conference to see who is in it, and act on it in place: **lock** or **unlock** the bridge, **kick** a participant, turn **video on or off**, or **destroy** the conference entirely. This is the operator's live control over a running bridge.

Voicemail

The stored **voicemail** records, per subscriber. Each record can be **deleted** individually, for example to clear a message left in error or to free a full mailbox.

OCS Test

The active **OCS (Online Charging System) sessions** the TAS is holding, so you can see which calls are drawing against a real-time balance. An individual charging session can be **terminated** from here. This is useful for clearing a session that has stuck open after its call has gone.

Routing

The **dialplan templates** the TAS routes calls against. The list shows each template; open one to read its content. This is the reference for how numbers are matched and where calls are sent. Read it alongside the **Number Translation** tool to confirm a given number routes as intended.

Configuration

The **application configuration** the TAS is running, grouped by application. Open an entry to read its settings. This is the read reference for confirming the server is configured as expected without leaving OmniWeb.

Prompts

The **text-to-speech prompt recordings** the TAS plays to callers, the announcements and menu prompts, with the status of each. Use this tab to confirm the expected prompts are present and rendered.

Tools

Alongside the tabs above, the TAS page carries a set of on-demand tools for validating routing and subscriber data. Each takes a small form, runs the request against the application server through the OmniWeb proxy, and shows the result in place.

Call Simulator

Places a test call through the TAS to validate routing and media. Enter a **destination** (and optionally a source number, source IP and a forced disposition). The simulator walks the call through the dialplan **without ringing anyone**. It returns the resolved **call type** (MT, MO or emergency), the on-/off-net decision, every **dialplan variable** it set, and a step-by-step list of **processing notes**, so you can see exactly how a number would be handled before a real call is placed. The OCS authorisation and HLR lookup can each be skipped to isolate a stage. A separate **echo-test call** action rings a destination and loops the party's own audio back, which confirms the live voice path to a handset.

Sh Query

Queries the HSS over the **Sh** interface for a subscriber's user data. Enter the subscriber's **public identity** and choose the **data reference** to fetch, from the full **repository data** to a specific element such as the IMS user state, the serving S-CSCF name, the initial filter criteria, the MSISDN or the IMSI. The tool returns the transparent and repository service data the HSS holds. Use it to confirm what the application server sees for a subscriber when a service is not behaving as expected.

HLR Lookup

Performs an **HLR lookup** over SS7 MAP (a Send Routing Information request) to resolve a subscriber's routing and roaming state. Enter the **MSISDN** in international format and the tool returns the HLR response together with the **dialplan variables** the result would set on a call. This is how you confirm a number's current network location before diagnosing why calls to it route the way they do.

Number Translation

Tests how a dialled number is **normalised and translated** by the TAS routing rules. Enter a **country code**, the **phone number** and an optional **disposition**. The tool returns the translated number the dialplan would use. Read it alongside the **Routing** tab to confirm a given number is matched and rewritten as intended.

ESL

An interactive **console** for issuing Media Server-style commands to the TAS engine for diagnostics. Type a command and press Enter. The output is printed back in the console, and Arrow Up/Down recalls previous commands. Example commands such as `status`, `sofia status`, `show channels` and `show registrations` are offered as one-click chips. It is the lowest-level view of the running engine, for confirming live channels, gateway state and registrations directly.

Logs

The **Logs** tab tails the TAS's live log output to troubleshoot a single application server without leaving OmniWeb.

Monitoring dashboards

Call volumes, per-service counters and other time-series for the TAS are presented as Grafana dashboards, reached from the element page. OmniWeb

does not duplicate those charts natively.

Related

- The **CSCF** signals the calls the TAS applies services to. The two together deliver IMS voice.
- The **HSS** holds the subscriber and Sh user data the TAS resolves against.
- The **OCS** rates and charges the TAS's calls in real time over Diameter.
- The **DRA** relays the TAS's Diameter signalling (Sh, Gy) to those peers.

Test Devices

Test Devices turns real Android handsets at remote sites into a remotely-controllable test farm. From OmniWeb you can watch a phone's screen live, drive it (tap, swipe, type, hardware keys), place calls and send texts, and run scheduled health checks. These checks cover VoLTE registration, signal and connectivity. OmniWeb exports their pass/fail results to Prometheus for alerts and dashboards.

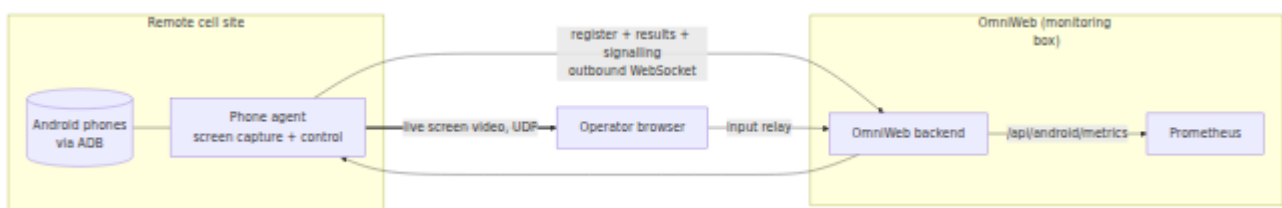
[← Back to Operations Guide](#)

It exists to answer the question a NOC keeps asking about a remote cell site: *"is a real phone actually working on this network right now?"* The phone must be registered for VoLTE and able to call, text and reach the internet. Nobody has to drive to the site.

Architecture

Each host with phones plugged in via ADB runs a lightweight **agent**. The agent dials *out* to central OmniWeb, so sites behind NAT need no inbound access. Live video uses **WebRTC** (UDP) so it survives lossy, high-latency links far better than TCP screen-sharing. Central relays the input and test results.

Network paths matter. The web UI, input and call setup ("signalling") ride the control plane over HTTPS to the OmniWeb server. However, the live screen video streams **directly between the operator's browser and the agent host over UDP**. It does not pass through the server. If a browser can reach the server but not the agent host over UDP, the console hangs on "Connecting...". See [Network Requirements](#) for the firewall rule to request and how to recognise the problem.



Component	Role
Phone agent	Discovers ADB devices, captures the screen, injects input, runs host test scripts. One per site; installed by the <code>device_tester</code> Ansible role.
Central OmniWeb	Device registry, WebRTC signalling relay, script dispatch, results history, Prometheus metrics.
Prometheus	Scrapes scheduled-test results from <code>/api/android/metrics</code> .

Device types

The farm is no longer phones-only. Every connected device is one of three kinds, auto-detected by the agent and shown with its own badge (and the Android robot logo for phones) in the fleet:

Kind	What it is	Console
Android phone	A normal handset (Android 5.0+ / API 21+) over ADB	Live screen + full control
USB modem	A cellular module that exposes AT ports (e.g. Quectel EP06)	AT-driven cellular console: status, calls, SMS, cell lock, VoLTE
Old-Android dongle	A cheap LTE dongle running old Android (API < 21, e.g. UZ801) too old for the live-screen view	Poll-based screencap view + control
RAN Simulator UE	A virtual UE from the RAN Simulator , with no hardware	UE control panel: attach/detach, idle, out-of-coverage, IMS register, SMS, calls

Clicking a device opens the console appropriate to its kind. Inside any console the **Swap phone** button (top-right) switches to another device without going back to the fleet grid.

Simulated UEs from the RAN Simulator appear in the fleet tagged as simulators, so you can drive emulated subscribers alongside real handsets. The Test Devices filter lets you narrow to just the RAN Simulator UEs. They use the **same UE control panel** shown in the RAN Simulator's own Network Map, so there is one place to attach, idle, drop out of coverage, register to IMS, and send SMS/calls. See **RAN Simulator** for the full UE lifecycle.

Adding a device to a site

A device joins the fleet automatically once the **agent host** at that site can see it over ADB. There are two steps. First, enable ADB on the handset. Then, if the agent runs in a VM or container, pass the USB device through to it.

1. Enable ADB on the phone

1. On the handset, open **Settings** → **About phone** and tap **Build number** seven times to unlock **Developer options**.
2. In **Settings** → **System** → **Developer options**, turn on **USB debugging**.
3. Plug the phone into the agent host with a **data-capable** USB cable.
4. The phone shows an *"Allow USB debugging?"* prompt with the host's RSA key. Tick **Always allow from this computer** and accept. (Without the tick, the prompt returns every time it re-connects and the device shows as `unauthorized`.)

A normal handset needs nothing more. The test scripts and live screen use only standard ADB. Keep the screen unlocked for the first connection so the prompt can be accepted.

2. Pass the USB device through to the agent host

Skip this if the agent runs on bare metal. If the agent host is a virtual machine or container, the hypervisor owns the USB port, so you must forward the device:

Platform	How to forward the device
Proxmox (VM)	<code>qm set <vmid> -usbN host=<bus>-<port></code> to bind a physical port (the mapping survives swapping phones on that port), or <code>host=<vendor>:<product></code> to follow one specific device. Find the ids with <code>lsusb</code> on the Proxmox host.
Proxmox (LXC container)	Add to <code>/etc/pve/lxc/<ctid>.conf</code> : a device-cgroup allow for USB (<code>lxc.cgroup2.devices.allow: c 189:* rwm</code>) and a bind-mount of the USB bus (<code>lxc.mount.entry: /dev/bus/usb/<bus> dev/bus/usb/<bus> none bind,optional,create=dir</code>), then restart the container. Re-plugging a device onto a different bus needs the matching bus mounted.
VMware	With the VM running, VM → Removable Devices → &lt;phone> → Connect (or add a USB controller and attach the device in the VM settings).

USB modems and old-Android dongles forward the same way. They appear as the same USB device class, so a bus that is passed through carries phones, modems and dongles alike.

3. Confirm it appears

On the agent host, `adb devices` should list the handset as `device` (not `unauthorized` or `offline`). Within a few seconds it appears in the **Fleet View** for that site. If it does not, see [Troubleshooting](#).

Fleet View

The **Test Devices** view lists every connected handset across all sites, grouped by site. Each card shows the model, manufacturer, Android version, screen resolution, network operator, battery and build. At a glance you can see which phones are online and healthy.

The fleet groups devices by site (e.g. "Cell Site 12", "Lab North"). Each card shows live status, battery and operator. Select phones and a script to fan a test out across them. Click a card to open its console.

Labelling a device. The pencil on a fleet card opens **Edit device metadata**, where you can give a device a friendly **name** and record its **operator**, **IMSI** and **MSISDN**. These are operator-assigned (modern Android hides the IMSI/MSISDN from ADB) and stick to the device even when it is offline. The name is what shows in the console headers and device pickers.

The sidebar's **Test Devices** entry expands to show each site (with a device count), the **Agents** view, the script library, the scheduler and the device tools, so you can jump straight where you need.

Running tests across devices

Select one or more phones, choose a script, and **Run on selected** fans it out in parallel. Results stream into a table, one row per device with a pass/fail badge, exit code, duration and expandable output.

Agents

The **Agents** view (sidebar → Test Devices → Agents) lists every bench connected to this server, one row per site host, with its online status, IP, the devices it carries, when it registered, and its software version. It answers "which sites are actually reporting in, and from where?" at a glance, and is where you confirm a newly-installed bench has dialed in.

Device Console

Clicking a phone opens its **console**: a live view of the screen with full remote control.

*The console streams the phone's screen live. Click to tap, drag to swipe, type to enter text. A **network-quality** badge (top-right of the screen) shows green/amber/red at a glance. Click it for the full stream stats (resolution, FPS, bitrate, 5-second packet-loss %, jitter, RTT). The hardware-key bar (back, home, recents, volume, power) and quick actions (Airplane, Dialer, Messages, Settings) sit below. A tabbed tool panel sits alongside: Cellular validation, Scripts, Shell, Logcat, Files and Record.*

One viewer at a time. A device streams to one operator at a time. If someone else is already viewing it, the console shows "*In use by <user> (<IP>)*". The IP lets you tell whether it is you on another machine or a different operator. A **Take over** button claims the stream. A dropped or reloaded console reconnects to its own session automatically rather than fighting for it.

Remote control

Action	How
Tap	Click anywhere on the screen
Swipe	Click and drag
Type	Click the screen, then type on your keyboard
Hardware keys	Back, Home, Recents, Volume $\pm$, Power buttons
Quick actions	One-click Airplane toggle, or open the Dialer, Messages or Settings app. These fire in the background. They do not run a test or block the console.

The streamed screen is the live device. Any app or web page renders and is fully drivable from the console. Here a test phone has a page open in the browser, controlled remotely:

Viewing more than one phone

To watch several handsets at once (both ends of a test call, say), open one phone's console and click **Extra phone** (top-right). Pick another phone from the list and its live screen opens alongside the first, with its own full controls: tap/swipe/type, hardware keys, quick actions, listen-to-audio and stream stats. The list stays open so you can keep adding, and the screens tile across the page — there is no fixed limit, so add as many as the bench has.

To open a whole group in one go, tick the phones you want on the Test Devices list and click **View selected**: the first becomes the primary device (the one the tools column acts on) and the rest tile alongside it.

Each extra screen carries its own picker to **swap** which phone it shows and an × to close it. The arrangement is captured in the page address (`?with=...`), so a specific multi-phone layout can be bookmarked or shared.

The tabbed tool panel (Cellular, Scripts, Shell, ...) always acts on the **primary** phone, the one you opened. The panel header names it so it is clear where a script or command will run.

Console tools

Alongside the screen, a **tabbed tool panel** keeps the per-device tools together. Switch tabs without losing each tool's state:

Tab	What it does
Cellular	The call / text / ping / speedtest validation tools (below)
Scripts	Run a library script on this device and see its full run history, with a one-click re-run on any past run
Shell	An interactive command/response console for the device
Logcat	Live device log stream with text + level filtering
Files	Browse, download, upload, rename and delete files on the device
Record	Capture a tap/swipe sequence as replayable script lines

Cellular Validation panel

Built-in tools for the checks an operator runs by hand, each taking the inputs you would expect:

Tool	Inputs	What it does
Call	Number, hold time	Dials, confirms the call goes off-hook, holds, hangs up
Text	Number, message	Sends an SMS and confirms it left the device
Ping	Host, count	Pings a host over the device's data connection
Speedtest	Test URL	Measures download throughput over the device's data link

The **Number** box remembers numbers you have recently called or texted (kept in your browser only, not on the server) and offers them as a dropdown, so re-dialling a test number is a click.

Scripts

The **Scripts** tab runs any library script against this one device and keeps its full run history without leaving the console. You can filter that history by source, status and test. Every past run has a **re-run** button, so repeating a check (or a script that just failed) is one click. Scripts that take inputs re-prompt for them.

Shell

The **Shell** tab is an interactive command/response console for the device. Type a command, see its output and exit code, with ↑/↓ to recall history. It is the device analogue of the modem AT console, for the times a quick command is faster than writing a script.

Logcat

The **Logcat** tab streams the device's live system log. Filter by free text and minimum level (Verbose → Fatal). Levels are colour-coded. **Start/Stop** controls the stream. Stopping keeps what you have captured on screen so you can scroll back through it.

Files

The **Files** tab is a file manager for the device. Browse directories (starting at `/sdcard`), **download** a file to your machine, **upload** a file to the device, and **rename**, **delete** or create folders. Useful for pulling a capture or diagnostic file off a device, or for pushing a config onto it.

Gesture recording

Toggle **Record**, perform a tap/swipe sequence on the screen, and the console emits the equivalent `adb input` lines. Paste them into a folder script to turn a manual UI flow into a repeatable test.

USB Modem console

A USB cellular modem (Quectel EP06 and similar) appears in the fleet as a **modem** and opens a cellular console instead of a screen. The agent relays AT commands to the modem over its serial (AT) port. It logs every command and response to the **Terminal** tab as an audit trail.

Tab	What it does
Status	One-shot read of model / firmware / IMEI / IMSI / SIM, signal (CSQ + per-RAT QCSQ), operator, EPS registration, attach, IMS, IP, APN, DNS/PCO
Actions	Set APN (preset or custom, with PDP / IP type), attach/detach data, show IP + DNS, ping, send SMS, radio (airplane / reboot)
Networks	Operator scan (AT+COPS=?) and a serving/neighbour cell measurement (AT+QENG) → EARFCN / PCI / band / RSRP / RSRQ / SINR table
Cell lock	Lock to a specific EARFCN (+PCI) (AT+QNWLOCK), band lock (AT+QCFG="band"), RAT preference, or release back to auto
Calls	Dial / answer / hang up, an AT+CLCC call table, and an Enable / Disable VoLTE control (AT+QCFG="ims")
SMS	Read the inbox (AT+CMGL), delete, and send
Events	Enables the relevant URCs and polls them into a timestamped log (registration / call / signal / new-SMS changes)
Terminal	Raw AT entry plus the full request/response log

The full AT-command reference, with real EP06 responses, is in [modem-at-commands.md](#).

VoLTE: the EP06 is VoLTE/IMS-only for voice. *Enable VoLTE* sets AT+QCFG="ims",1; an AT+CFUN=1,1 reboot may be needed to (de)register IMS, and it only registers if the network + SIM provision it and the correct carrier MBN is selected.

Old-Android dongles (screencap mode)

Cheap ~\$10 LTE "MiFi" dongles (e.g. the UZ801, Qualcomm msm8916) run **Android 4.4 (API 19)**, which is too old for the live-screen view. These devices fall back to a **screencap** console: the agent polls `adb screencap` for frames (a few per second, because `screencap` itself is the bottleneck on this hardware) and injects `adb input` for control. You get a click-to-tap, type-to-text view. It is not smooth video, but it is enough to drive it.

The agent smooths over the rough edges of these dongles on the host side:

- **ADB unlock:** they ship in RNDIS mode. A udev-triggered service `curl`s the dongle's `usbdebug.html` to switch it into ADB mode. See [agent/host-setup/mifi-dongle](#).
- **Auto-English:** they ship in Chinese. The agent sets `persist.sys.language` to `en-US` the first time it sees an API-<21 device (disable with `OMNIWEB_DONGLE_AUTO_ENGLISH=0`).

Scripts

Tests are bash scripts that run on the **agent host** (so host tools like `grep/awk` are available) and target a device via ADB. There are two kinds:

- **Folder scripts:** health checks that appear in the library, the fan-out runner, and the scheduler. Some run with only the device (Device Connected, VoLTE Registration, Report Signal Strength). Others take optional inputs (`@param`). Ping Test takes a host and a count. Data Throughput Test takes a URL. The connectivity and voice tests (Ping, Data Throughput, Send SMS, Place Call) are folder scripts, so you can schedule them as well as run them by hand. Two airplane-flap recovery tests confirm that a device recovers after it loses radio. **Airplane Flap → Ping Recovery** turns airplane mode on, then off, waits for data to re-attach, then pings. **Airplane Flap → IMS Recovery** does the same, then confirms VoLTE re-registers.

- **Built-in tools:** the app-open shortcuts (Dialer, Messages, Settings) and the plain Airplane toggle in the console quick-actions. Each does one UI action with no pass or fail result, so it is not in the library or scheduler.

The script library lists the no-argument folder scripts. Each shows its description and timeout. Click one to view its source (read-only). Recent runs appear below.

A script reports its result by **exit code**: 0 is a pass, anything else a fail. Its captured output is shown in the results table.

Scheduled Tests

Scheduled tests are defined declaratively in `schedules.json` in the scripts directory. The file is re-read on every cycle, so edits take effect without a restart. Each entry runs a folder script against its target devices on an interval.

Configuration

```
{
  "schedules": [
    {
      "name": "VoLTE Registration",
      "script": "volte_registration.sh",
      "interval_seconds": 120,
      "serials": [],
      "enabled": true
    }
  ]
}
```

Parameter	Type	Required	Default	Description
<code>name</code>	String	Yes	-	Unique label for the schedule. Appears as the <code>schedule</code> label on the exported metric.
<code>script</code>	String	Yes	-	Filename of a folder script in the scripts directory to run.
<code>interval_seconds</code>	Integer	No	300	How often to run the script, in seconds. Minimum 30.
<code>serials</code>	List	No	<code>[]</code>	Device serials to target. An empty list means all currently-connected devices , so the schedule auto-follows whatever phones are plugged in.
<code>params</code>	Object	No	<code>{}</code>	Inputs for a script that declares <code>@param</code> values. Keys are the parameter names. For example, Ping

Parameter	Type	Required	Default	Description
				Test accepts <code>host</code> and <code>count</code> .
<code>enabled</code>	Boolean	No	true	Whether the schedule is active. Set <code>false</code> to pause it without removing the entry.
<code>exclusive</code>	Boolean	No	false	Reserve the phone for this test's duration. While an exclusive test is running, no other scheduled test starts on that device, and the exclusive test itself waits for the device to be free before it begins. Use it for disruptive tests — an airplane-mode flap or a reboot — that would corrupt another test (say a VoLTE call) running on the same handset at the same time.

Example: VoLTE check across a site

```
{
  "name": "VoLTE Registration",
  "script": "volte_registration.sh",
  "interval_seconds": 120,
  "serials": [],
  "enabled": true
}
```

How it works: Every 120 seconds the scheduler runs `volte_registration.sh` against every connected phone. The script checks the device is voice-registered (`IN_SERVICE`) on LTE with VoPS (Voice-over-PS) support, and exits `0` if VoLTE is registered. The result becomes a Prometheus gauge per device.

Use case: Continuously prove that real handsets at a site can register for VoLTE, and alert the moment one cannot.

Example: Ping every phone

```
{
  "name": "Ping Test",
  "script": "ping_test.sh",
  "interval_seconds": 120,
  "serials": [],
  "params": { "host": "8.8.8.8", "count": "5" },
  "enabled": true
}
```

How it works: Every 120 seconds the scheduler runs `ping_test.sh` on every connected phone. The `params` supply the script inputs. The script pings `8.8.8.8` five times over the device data link and exits `0` if it gets replies. Change `host` to test a specific target.

Use case: Prove that real handsets at a site have working data, and alert on packet loss.

Automatic pause while a phone is in use

Opening a phone's live screen pauses the scheduled tests **for that phone**. A test fired at a handset someone is watching and tapping would fight them for the screen and record a meaningless pass or fail, so the scheduler skips any phone that has a live viewer and picks it up again as soon as the view is closed.

- The pause is per phone. Every other phone a schedule targets keeps running on its normal interval.
- A skipped run is not counted as done: the schedule fires on its next cycle once the phone is free, rather than waiting out a full interval.
- A test that was already running when the screen was opened is left to finish.
- Close the phone view (or navigate away) and testing resumes within about a minute. If the browser disappears without closing cleanly, the hold lapses on its own.
- **Run now** is unaffected — an explicitly requested run always goes ahead.

The phone's Scheduled Tests tab shows a banner while the pause is in effect, and the fleet marks a held phone **in use** with who has it.

Metrics

Central OmniWeb exposes the latest result of each scheduled test, per device, in Prometheus format at `/api/android/metrics`. Prometheus scrapes it as the `android_device_tests` job (configured automatically by the monitoring role).

Metric: `omniweb_android_test_pass` **Type:** Gauge **Description:** Result of the most recent scheduled run: `1` for pass, `0` for fail **Labels:**

- `phone`: device model (e.g. `SM-A536E`)
- `serial`: device serial number
- `site`: the agent's site label (e.g. `Cell Site 12`)
- `agent`: the agent host the device is attached to
- `script`: the script that ran
- `schedule`: the schedule name

- `status`: the run outcome (`passed`, `failed`, `timeout`, or `error`)

Companion gauges share the same labels:

Metric	Description
<code>omniweb_android_test_duration_ms</code>	Duration of the last run, in milliseconds
<code>omniweb_android_test_last_run_timestamp</code>	Unix time of the last run

A separate gauge reports which phones are currently held by a viewer, so a staleness alert can tell a deliberately paused phone from a broken test:

Metric	Description	Labels
<code>omniweb_android_device_in_use</code>	<code>1</code> while someone has the phone's screen open (its scheduled tests are paused), <code>0</code> otherwise	<code>serial</code> , <code>phone</code> , <code>site</code> , <code>agent</code>

Example queries:


```
# Phones currently failing VoLTE registration
omniweb_android_test_pass{schedule="VoLTE Registration"} == 0

# Failing tests grouped by site
count by (site) (omniweb_android_test_pass == 0)

# A test that has not run recently (stale agent or device offline)
time() - omniweb_android_test_last_run_timestamp > 600

# The same, ignoring phones someone currently has open (tests are
# paused for those)
(time() - omniweb_android_test_last_run_timestamp > 600)
  unless on (serial) omniweb_android_device_in_use == 1

# Phones currently in use
omniweb_android_device_in_use == 1
```

Grafana dashboard

The monitoring role ships a **Phone / Device Tests** dashboard (Grafana folder **Devices**). It reads the metrics above and shows, per phone and schedule:

- a pass or fail state timeline,
- the latest result for each phone and test,
- test freshness (time since the last run, to catch a stalled agent),
- run duration over time.

Filter the dashboard by site, phone, or test at the top. The dashboard is provisioned automatically. See [Grafana](#).

Troubleshooting

A phone does not appear in the fleet

Symptoms: A device is plugged in at a site but is not listed under Test Devices.

Possible causes:

- The phone is not authorised for ADB on the agent host
- The agent is not running or cannot reach central
- The phone is in an offline/unauthorised ADB state
- The central Test Devices service is not running (Test Devices is served by its own dedicated backend service, separate from the main request pool)

Resolution:

1. Confirm the phone shows as `device` (not `unauthorized/offline`) in ADB on the agent host, accepting the RSA prompt on the handset if needed.
2. Check the agent service is running and that its configured central URL and token are correct.
3. On central, confirm the Test Devices service is running and that the gateway routes `/api/android/*` to it. If the agent registered but the fleet is empty, this service is the usual cause.
4. Verify the phone's USB connection; re-seat the cable if it shows as offline.

The console screen stays on "Connecting"

Symptoms: The device console opens but the screen never appears, while the rest of the UI (fleet list, device details, input) works normally and the device shows **online**.

Possible causes:

- UDP is blocked between the browser's network and the agent host (the live video streams **directly** browser → agent host, not via the server, and this is by far the most common cause)
- The agent cannot start screen capture on that handset

This is **not** an SSL or MTU problem: TLS is proven working (the UI loaded), and with MTU issues the video would connect and *then* tear rather than never starting. The tell-tale sign is a device that shows **online** in the fleet yet whose live view alone will not start.

Resolution:

1. Confirm the browser's network can reach the agent host over **UDP** (the video path). This usually means a firewall/VLAN rule permitting the operator's subnet to the bench hosts over UDP. See **Network Requirements** for the exact rule to request and how to confirm the fix.
2. Check the agent log for screen-capture errors. Some handsets cannot have their screen captured over ADB.

VoLTE / signal checks report "not readable"

Symptoms: VoLTE or signal scripts fail with the telephony state unreadable.

Possible causes:

- The handset restricts telephony status to privileged access

Resolution:

1. Use test handsets that expose telephony status to ADB. Vendor-locked consumer devices may hide it without privileged access.

Test Devices: Network Requirements

Live phone viewing in **Test Devices** uses **two separate network paths**.

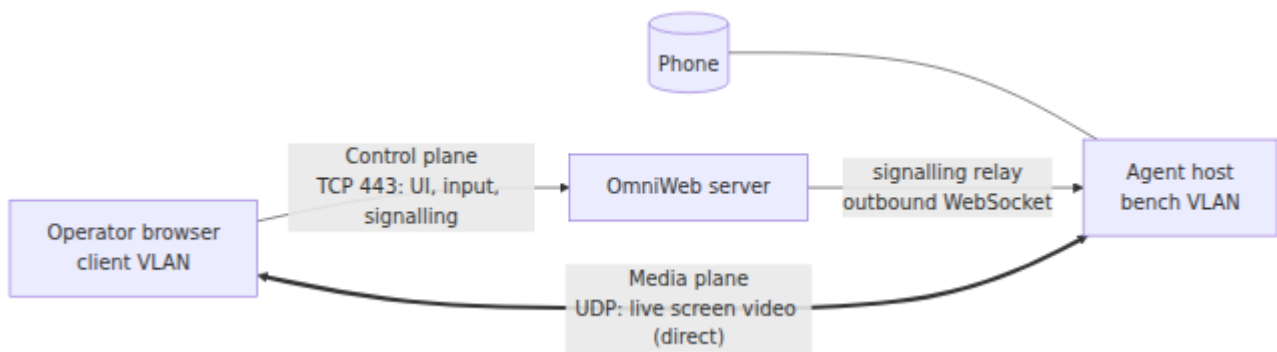
Understand the split to make it work across firewalls and VLANs. The split also helps you diagnose the single most common symptom, a console that sits on "**Connecting...**" forever.

[← Back to Test Devices](#)

The two paths

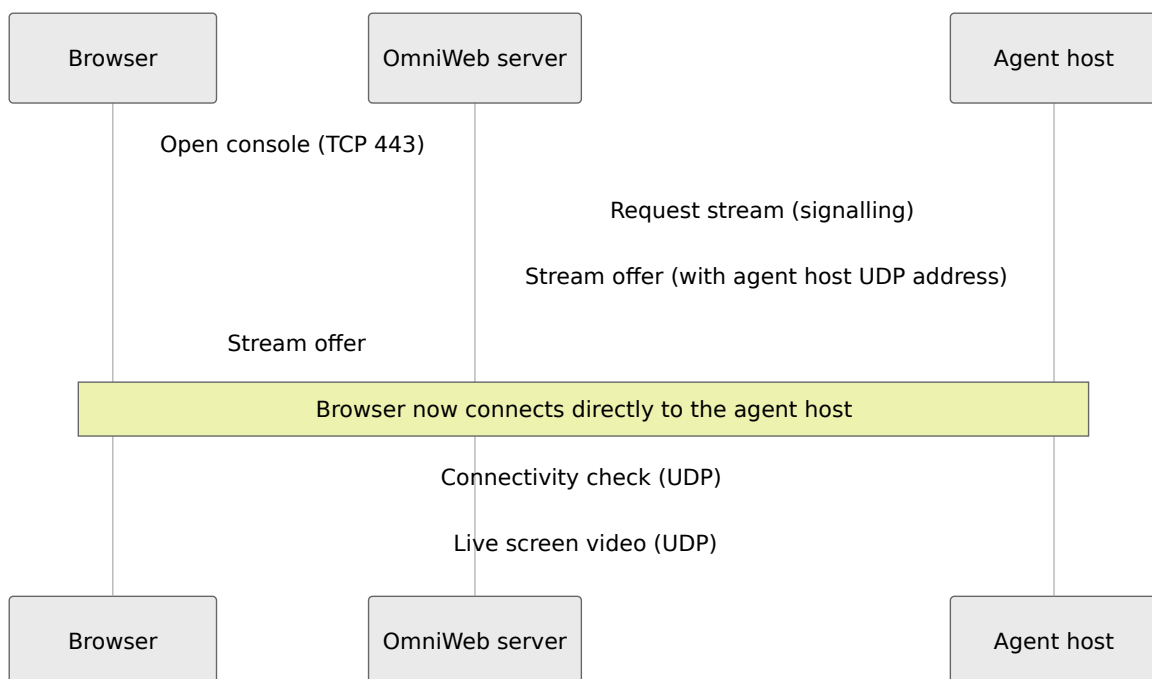
Path	Carries	Direction	Transport	Endpoint
Control plane	The web UI, login, the device list, remote input, test results, and the call setup ("signalling") for a live view	Browser → OmniWeb server	TCP 443 (HTTPS)	The OmniWeb server
Media plane	The live screen video itself	Browser ⇌ agent host (the bench the phone is plugged into)	UDP (WebRTC)	The agent host, not the server

The crucial point: **the live video does not flow through the OmniWeb server**. Once the server has brokered the connection, the video streams **directly between the operator's browser and the agent host** over UDP. If a network allows the browser to reach the *server* but not the *agent host*, it loads the UI, shows the device list, and then hangs on "Connecting..."



Why the media path is direct

Live screen video is sent peer-to-peer with **WebRTC over UDP** so it survives the lossy, high-latency links typical of remote cell sites far better than relaying it through a central server over TCP. The trade-off is that the operator's browser needs a **direct UDP path to the agent host**. There is no media relay in the middle to fall back on.



Firewall requirements

For an operator to view phones at a site, **both** paths must be permitted from the operator's network to the relevant hosts.

Control plane (usually already open)

Field	Value
Action	Permit (stateful)
Protocol	TCP
Source	Operator / client VLAN(s)
Destination	OmniWeb server
Destination port	443

If the OmniWeb UI loads at all, this path is already open. No change needed.

Media plane (the one that is usually missing)

Field	Value
Action	Permit (stateful , so return video is allowed automatically)
Protocol	UDP
Source	Operator / client VLAN(s), where browsers run
Destination	The device-bench host subnet (the agent hosts), e.g. 10.0.20.0/24
Destination ports	32768–60999 (the default Linux ephemeral range the agent draws media ports from)

The browser always initiates the media connection, so a **stateful** permit on client → agent host UDP also allows the agent's return video automatically. No STUN/TURN or other ports are involved.

Do not change port 443 for this. Signalling already rides the control plane. Only the UDP media path to the agent hosts needs opening.

Recommended: pin the media port range

Opening the whole ephemeral range (32768–60999) is broad. You can instead configure the agent to draw its media ports from a small fixed range. This shrinks the firewall rule to a tidy block:

Field	Value
Action	Permit (stateful)
Protocol	UDP
Source	Operator / client VLAN(s)
Destination	Device-bench host subnet
Destination ports	50000–50100 (pinned range)

This is the preferred form to hand a network team. 100 ports to one subnet is far easier to approve and audit than the full ephemeral range. The pinned range is set on the agent host. Coordinate the exact range with whoever maintains the benches so the firewall rule and the agent configuration match.

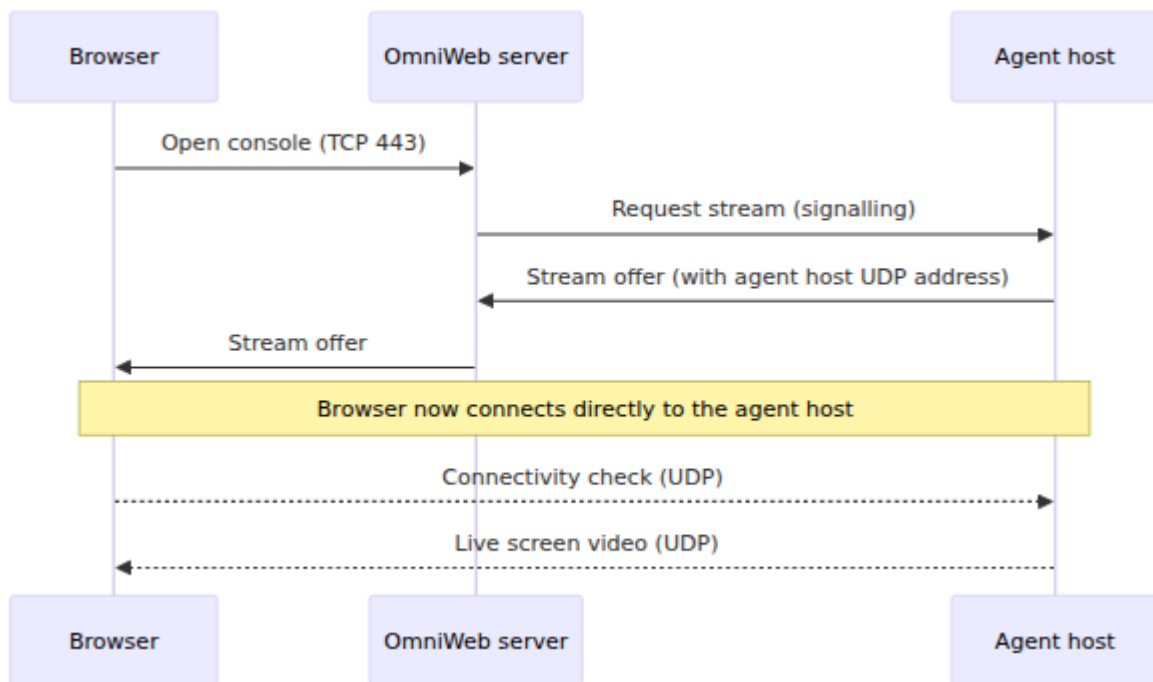
Recognising the problem

Symptom: the device console opens, but the screen never appears. It stays on "**Connecting...**" indefinitely. Everything else works: the fleet list, device details, even remote input setup.

This is almost always the media-plane UDP path being blocked. It is **not** a TLS/certificate problem and **not** an MTU problem:

Suspected cause	Why it is not this	What you would actually see if it were
SSL / certificate	The UI, login, device list and signalling all succeed over HTTPS, so TLS is clearly working	The page itself would fail to load, or show a certificate warning
MTU / fragmentation	MTU only matters once video packets flow. Here no video packets ever arrive	Video would connect , then stall or tear, not sit on "Connecting..."
Agent / phone down	The device shows online in the fleet with live battery/operator data	The device would be missing from the fleet, or marked offline
UDP path blocked ✓	The browser can reach the server (UI loads) but not the agent host over UDP	Permanent "Connecting...", exactly this symptom

A useful tell: the device tile shows the phone as **online** (so the agent and phone are healthy and the server can see them), yet the live view alone will not start. That combination points squarely at the media-plane firewall path.



Confirming the fix

After the firewall rule is applied, re-open the device console. If the screen appears within a few seconds and the **network-quality** badge (top-right of the live view) goes green, the media path is working. The badge's stream stats (resolution, FPS, bitrate, packet-loss %, jitter, RTT) confirm video is flowing end-to-end.

If it still hangs, verify two things with whoever maintains the benches. First, the firewall rule's **destination subnet and ports** must match the agent hosts and their configured media-port range. Second, the rule must apply to the operator's **actual** source VLAN (a laptop on guest Wi-Fi is a different source than the corporate LAN).

Multiple operator locations

Each operator network that needs to view phones must have the media-plane rule for its own source VLAN. The rule is per source subnet → bench subnet, so adding a new office, VPN pool, or site means adding its subnet as an additional **Source** on the same UDP permit. Sites whose operators only ever view phones from the bench LAN itself need no inter-VLAN rule at all.

Network Topology & Single Pane of Glass

OmniWeb's topology views are the "single pane of glass" entry point to the whole network. They show how the elements connect, which ones are deployed, and let you jump straight from a node on the map into that element's workspace.

[← Back to Operations Guide](#)

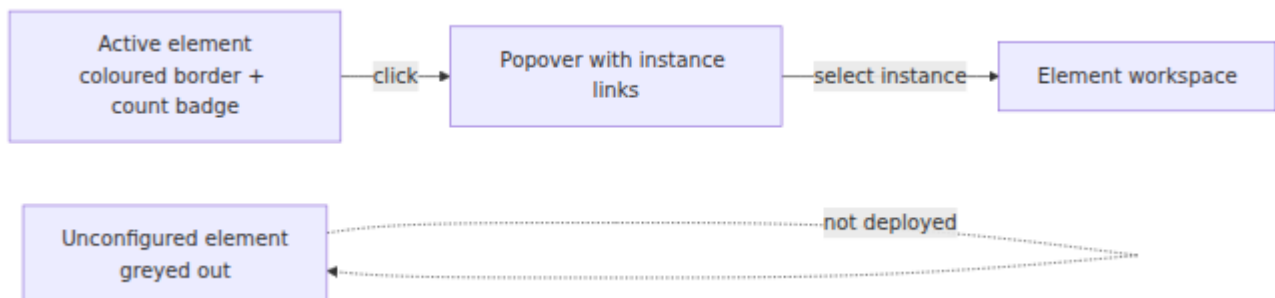
Topology Views

The landing/topology page presents the network across four domain tabs. Each is an interactive map of the relevant elements and their interfaces.

EPC topology: UE → eNB → MME → SGW-C → PGW-C → UPF, with Diameter signalling through the DRA to the HSS. Active elements show a coloured border and an instance-count badge.

Tab	Elements shown
Core (EPC)	UE → eNB → MME → SGW-C → PGW-C → UPF; DRA ↔ HSS; OCS; TWAG; RAN Monitor
IMS	P-CSCF → I-CSCF → S-CSCF; DRA ↔ HSS; TAS; IMS SMSc → SMSC → SMPP GW; OCS; Entitlement (OmniSEP)
CS Core	BTS → BSC → MSC; STP ↔ HLR / CAMEL GW / IP-SM-GW; HSS; SMSC; OCS
5G Core	gNB → AMF → SMF → UPF; NRF, SCP, AUSF, UDM, UDR, PCF; CHF → OCS; NSSF; BSF

Reading the Map



- **Active elements** are those with instances declared in `omniweb.json`. They appear with a coloured border and a badge that shows how many instances exist. Click one to open a popover with direct links to each instance. You can then navigate from the topology straight into an element's pages.
- **Unconfigured elements** appear greyed out, so the map doubles as a quick inventory of what is and is not deployed.
- **Connections** indicate the interface between elements (e.g. SIP, Diameter, PFCP, GTP-C, SS7), distinguishing signalling types.

This makes the topology page the natural starting point: see the shape of the network, spot what is deployed, and drill in.

Element-Level Topology

Some elements also provide their own focused topology view. The **PGW-C → Topology** tab, for example, shows that node's relationships to its SGW-C, UPF, and Diameter peers, annotated with live session counts and utilisation. This gives a localised single-pane view for that element's neighbourhood.

PGW-C topology: SGW-C, UPF, and Diameter peer relationships with per-peer session distribution.

Host List & Infrastructure

Beyond the network functions themselves, the **Host List** page is the single pane of glass for the *infrastructure* that runs them. It lists all infrastructure hosts by category, each with a direct link to its web UI:

Category	Examples
Monitoring	Grafana, Prometheus, Loki
SIP Capture	Homer Web UI
Charging	OCS Web UI
DNS	DNS servers
Virtualisation	Proxmox hypervisor consoles
Out-of-band	iDRAC server management
Routers	MikroTik WebFig, Peplink
Microwave	SAF, Ubiquiti AirFiber links
NAS	Synology DSM

The Host List: every infrastructure and element host by category, each with a direct link to its web UI (Grafana, Prometheus, Loki, element APIs, and more).

This gives operators one place to reach every supporting system: monitoring, hypervisors, out-of-band management, and transport. They do not need to hunt for bookmarks.

TWAG: Trusted WLAN Access Gateway

The **TWAG** (Trusted WLAN Access Gateway) brings trusted Wi-Fi access into the mobile core. It authenticates Wi-Fi devices with EAP-AKA/AKA' against the AAA/HSS over Diameter, hands each authenticated client an IP address by DHCP, and anchors its data back to the packet core over a GTP tunnel to the PGW. So a subscriber on Wi-Fi is served the same APN, policy and charging as on the cellular network. Its dashboard is the place to see who is connected, whether the Diameter and GTP paths are healthy, and to clear individual sessions.

[← Operations Guide](#)

The dashboard is a per-instance element page reached from the sidebar when a TWAG runs in the site's inventory. All traffic goes through the OmniWeb backend proxy to the TWAG's API, so it stays behind the single authenticated gateway. It is organised into tabs.

Overview

The landing tab is a single-screen health read: active and total **sessions**, connected/disconnected **Diameter peers**, accepted **subscribers**, the number of **access points**, **DHCP** lease count, the **GTP** tunnel status and its PGW address, the **authentication mode** in use (e.g. EAP-AKA), aggregate **traffic** in and out, uptime, and a live **recent-activity** feed of authentications and session starts/stops.

Sessions

Every authenticated Wi-Fi session, each row showing the subscriber **identity** (the network access identifier), **status**, assigned **IP**, client **MAC**, **SSID**, serving **NAS IP**, session **duration**, and **bytes in/out**. An operator can **delete** a session to force-disconnect a client, for example to clear a stuck or misbehaving device.

Access Points

The WLAN access points (NAS devices) currently connected to the gateway. Each can be **removed** individually when an access point is decommissioned or needs to be cleared.

Accounting

The accounting sessions the TWAG has recorded: the usage records raised for authenticated clients. Individual accounting sessions can be **deleted**.

Diameter

The Diameter peer connections to the AAA/HSS that carry authentication, with connected and disconnected counts. This tab also offers a **Diameter test**. It sends a test Multimedia-Auth-Request (MAR) to confirm the authentication path to the AAA/HSS is up, which is the quickest way to prove Diameter connectivity when authentications are failing.

DHCP

The DHCP leases the TWAG has issued to Wi-Fi clients, with the address, client and lease state for each, so you can confirm clients are being addressed and see how much of the pool is in use.

Logs

The **Logs** tab tails the TWAG's live log output for troubleshooting an individual gateway without leaving OmniWeb.

Actions

Each action asks for confirmation first:

- **Delete a session:** force-disconnect one authenticated Wi-Fi session (Sessions tab, per row).
- **Delete an access point:** clear a single access point / NAS entry (Access Points tab, per row).

- **Delete an accounting session:** remove a single accounting record (Accounting tab, per row).
- **Diameter test:** send a test MAR to validate the Diameter path to the AAA/HSS (Diameter tab).

Related

- The **PGW** anchors the TWAG's user plane over GTP. See the packet-core elements.
- The **HSS** holds the subscriber authentication data the TWAG's Diameter requests resolve against.
- **ePDG**: the untrusted-Wi-Fi counterpart for Wi-Fi calling.

UPF: User Plane Function

The **UPF** (User Plane Function) is the packet-forwarding node of the mobile core. It carries every subscriber's actual traffic: it receives GTP-U tunnels from the access network on **N3** (S1-U in the EPC), forwards packets to and from the data network, applies per-flow QoS and volume accounting, and buffers downlink packets when a device is idle. It has no logic of its own. The control plane (an **SGW-C** in the EPC over **Sxa**, or an **SMF** in a 5G core over **N4**) programmes it entirely over **PCF**. The control plane installs the packet-detection, forwarding, QoS and usage-reporting rules that describe how the UPF handles each session's traffic. Omnitouch's **OmniUPF** implements the forwarding path as an **eBPF/XDP dataplane** in the kernel, so the rules the control plane installs become entries in eBPF maps that the XDP program consults for every packet. OmniWeb surfaces the UPF's PCF associations and sessions, the live eBPF map contents, dataplane statistics, routing, downlink buffers, walled-garden state and configuration. It reaches the throughput and trend charts on the Grafana dashboards from the same page.

[← Operations Guide](#)

The UPF is a per-instance element page reached from the sidebar when one runs in the site's inventory, with an instance selector where several are deployed and a fleet view for aggregate status. All traffic goes through the OmniWeb backend proxy to the UPF's API, so it stays behind the single authenticated gateway. The documentation link in the page header (the in-context help button) opens this page. It is organised into tabs.

Fleet and instance selection

Where a site runs more than one UPF, the **fleet view** lists every instance with its health at a glance. This is the fastest way to confirm the whole user plane is up before you drill into one node.

From any tab, the **instance selector** in the page header switches which UPF the page reads. Everything below (sessions, associations, eBPF maps, statistics) is scoped to the selected instance, so you pick the node once and the whole page follows it.

Overview

A single-screen health read of the UPF: its **health** status, the number of **PFCP associations** it holds with control-plane peers, the total **active sessions** across those peers, and the **XDP action counters**: how many packets the dataplane has passed, dropped, redirected, transmitted or aborted, which is the quickest confirmation that traffic is actually flowing. Beneath it, each **PFCP peer** is listed with its association state, session count and any missed heartbeats, and the **eBPF map sizes** show the capacity of the PDR, FAR, QER and URR maps and the maximum session count. A search box jumps straight to a session by **UE IP** or **TEID**, and a **Node Report** action sends a PFCP Node Report Request to the first active control-plane peer.

Sessions

Every **PFCP session**: the per-bearer / per-PDU-session user-plane state the control plane has installed. Each row is keyed by its **local and remote SEID** and shows the session's **IMSI, APN/DNN**, the **association** (which control-plane peer owns it), and the count of **packet-detection, forwarding-action, QoS-enforcement** and **usage-reporting** rules it carries. Search **by UE IP or TEID** to find the session serving a specific device or tunnel.

Where several UPFs are deployed, the same tab in the **fleet view** lists every PFCP session across all of them in one table, with the carrying UPF shown against each row. UE IP and TEID are searched on every UPF; IMSI and APN/DNN are matched in the combined list. Opening a row goes to that session's detail on its own instance. A UPF that cannot be reached is named above the table, so a short list is never mistaken for an idle user plane.

Session detail

Opening a session shows its full rule state. A **session rule map** draws how each **Packet Detection Rule (PDR)** references the **Forwarding Action Rule (FAR)**, **QoS Enforcement Rule (QER)** and **Usage Reporting Rules (URR)** it applies. Click any node to highlight just the rules connected to it, so you can trace a single flow's path through the dataplane. Beneath the map, four tables give the detail:

- **Packet Detection Rules (PDR)**: how the UPF classifies incoming packets. This covers direction (uplink / downlink), the UE IP or inbound TEID they match, the SDF filters (protocol and port ranges), and which FAR, QER and URRs they resolve to.
- **Forwarding Action Rules (FAR)**: what happens to matched packets. The action is **Forward**, **Drop**, **Buffer** or **Notify CP**, with the outbound GTP-U tunnel details (remote IP and TEID) and network instance. You can edit FARs in place.
- **QoS Enforcement Rules (QER)**: the per-flow QoS. This covers the **QFI**, the uplink and downlink **gate** status (open / closed) and the **maximum** and **guaranteed bitrates**. You can edit QERs in place.

- **Usage Reporting Rules (URR)**: the volume counters. These are the current and last-reported uplink and downlink bytes and the report sequence number, which is what the UPF feeds back to the control plane for charging and quota.

From the session header you can also **activate or remove a walled-garden redirect** for that specific session (see Walled Garden below).

Associations

The **PFCP associations**: the Sxa/N4 relationships the UPF holds with its control-plane peers (the SGW-C or SMF). Each row shows the peer's **remote IP and port**, its **session count**, whether it is currently **associated**, the number of **missed heartbeats**, and when it last **connected** or **disconnected**. Here you confirm the user plane and control plane are in step. Here you also spot a peer whose heartbeats are lapsing.

Statistics

The dataplane's real-time numbers. **eBPF Map Usage** shows how full each map is (sessions, PDR, FAR, QER and URR) as used-versus-capacity bars that turn amber and red as they approach their limit, so you can see a UPF nearing its dimensioned capacity before it starts rejecting sessions. **XDP Statistics** breaks out the pass / drop / redirect / TX / aborted packet counters, and the **Metrics** section surfaces the UPF's Prometheus counters as individual figures.

A capacity check reads the same used-versus-capacity model. At a glance you can tell whether the constraining resource is the session table or one of the rule maps.

Routing

The routes that steer subscriber traffic to and from the UPF. The **Active UE IP Routes** table lists the routes the UPF currently holds for allocated UE IP pools, and a **Sync Routes** action reconciles them with the dataplane. Below it, an **FRR console** gives an in-context vtysh terminal against the UPF's routing daemon. To inspect routing state, type a `show` command (for example `show ip route` or `show ip ospf neighbor`), with tab-completion and command history. To make a change, switch to **configure mode**. This is how the UPF advertises its UE pools into the surrounding network without leaving OmniWeb.

Buffers

The **downlink packet buffers**. When downlink data arrives for a device that is idle (no active tunnel), the dataplane holds those packets against the relevant FAR and asks the control plane to page the device. Once the tunnel is re-established, the UPF flushes the buffered packets (this is the user-plane side of **Downlink Data Notification**). The tab shows the **total buffered packets** and **total FARs** holding packets, the **per-FAR** and **total** buffer limits, and a **lookup by FAR ID** to see how many packets a specific FAR is holding. Buffers can be **flushed** individually or **cleared** in bulk.

eBPF Maps

A direct browser onto the **eBPF maps** that are the UPF's dataplane. These are the raw kernel-side entries the XDP program consults for every packet, split across tabs:

- **Uplink PDR**: keyed by inbound TEID, which maps a tunnel to its PDR and FAR.
- **Downlink PDR**: keyed by UE IP, which maps a subscriber's downlink to its PDR and FAR.
- **FAR**: the forwarding actions, with decoded action, remote IP, outbound TEID and outer-header creation.
- **QER**: the QoS entries, with QFI, gate status and bitrate limits.
- **URR**: the usage counters, with current and last-reported uplink/downlink volumes.

Click any row to **edit the raw map entry**, which is a low-level troubleshooting and repair tool. PFCP normally drives the maps entirely, so you hand-edit an entry for diagnosis rather than routine operation.

Configuration

A read-only view of how the UPF is configured: its **network** identity (node ID, PFCP address and port, API port, N3 and N9 addresses), the **features** it has enabled (FTUP, UEIP, the route manager and its type, and the XDP attach mode: native, generic or offloaded), the **PFCP heartbeat** interval, timeout and retry settings, and the **dataplane** N3/N9 addresses. A **Raw JSON** toggle shows the full config document for support. This is where you confirm a UPF is dimensioned and addressed as expected.

Diagnostics

Runs a **ping from the UPF node** to a target host. Enter a target and a packet count, and the raw ping output comes back with a success/failure verdict. It is a quick way to confirm the UPF has data-network reachability (for example to a DNS server or upstream gateway) from where the packets actually leave, without shelling into the node.

Walled Garden

The **walled garden**: restricted forwarding for sessions that must reach only a captive portal before they are authenticated or entitled. When you activate a redirect on a session's SEID, it rewrites the downlink so the device goes to a **redirect URL / portal IP** instead of the open internet. The tab lists every **active redirect** with the UE IP, portal IP, redirect URL, gNB IP and the TEIDs involved. To release the session to normal forwarding, remove the redirect. A **whitelist** of IPs or CIDRs names the destinations a walled-garden session is still allowed to reach directly (for example the portal and its assets). You can also activate and remove redirects per-session from the session detail page.

Logs

The **Logs** tab tails the UPF's live log output for troubleshooting a single instance without leaving OmniWeb.

Monitoring dashboards

Throughput, per-interface packet rates, session-count trends and other time-series for the UPF are presented as Grafana dashboards, reached from the element page. OmniWeb does not duplicate those charts natively.

Related

- **SGW-C**: the EPC control peer that programmes this UPF (as the SGW-U) over **Sxa/PFCP** and forwards traffic on **S1-U / N3**.
- **PGW-C / SMF**: anchors the PDN connection and, as the SMF, programmes the 5G user plane over **N4/PFCP**.
- **MME**: drives bearer setup that ultimately installs the sessions seen here.

- **SCEF**: carries small non-IP payloads over the control plane, and deliberately bypasses this user plane.
- **Operations Guide**: back to the element index.

