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:

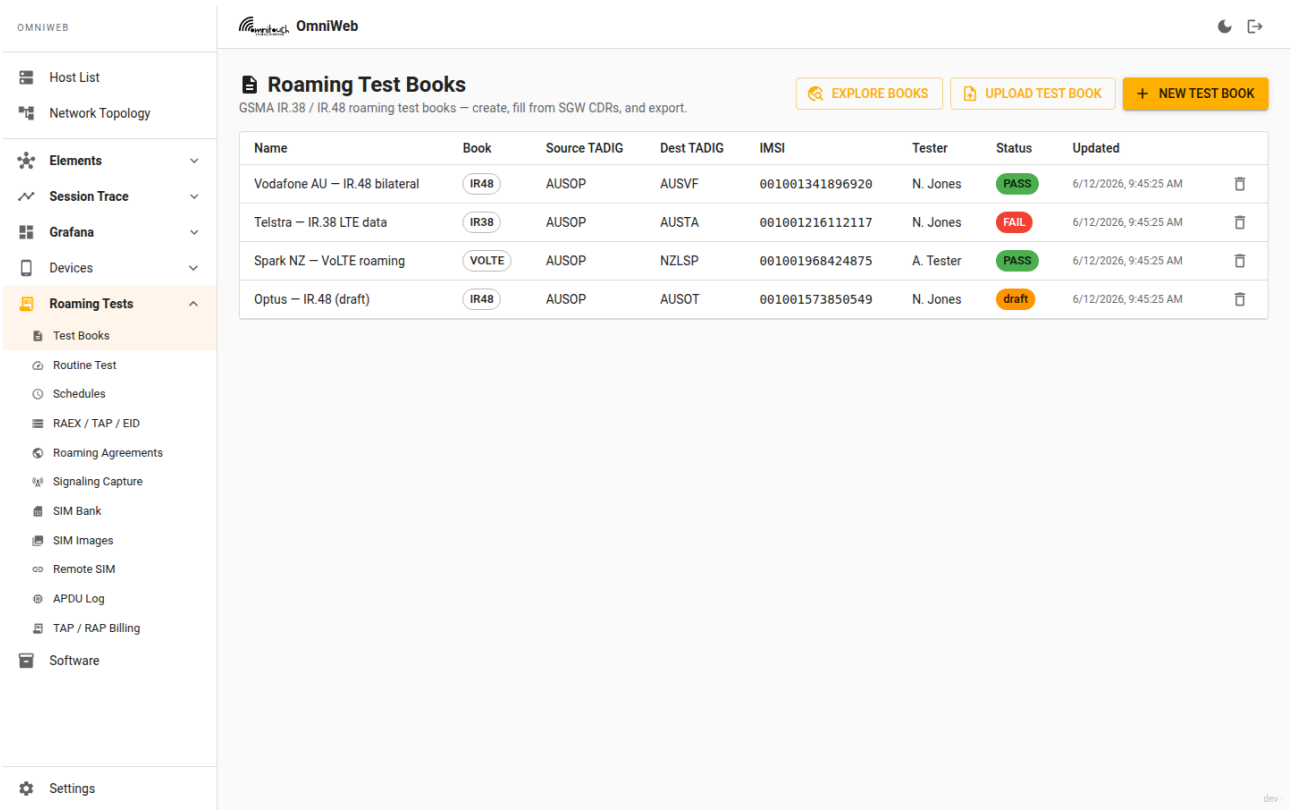
```
SUBSYSTEM=="tty", SUBSYSTEMS=="usb", ATTRS{idVendor}=="05c6",
OWNER="100000", GROUP="100046", MODE="0660"
SUBSYSTEM=="usb", ATTR{idVendor}=="05c6", OWNER="100000",
GROUP="100046", MODE="0660"
```

> 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.

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, drives the individual test cases automatically from a test device, records a pass/fail verdict with supporting evidence for each case, and produces a completed GSMA test-book workbook ready to hand 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), and hands back the same official GSMA spreadsheet a tester would otherwise fill in by hand.



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

Quick Links

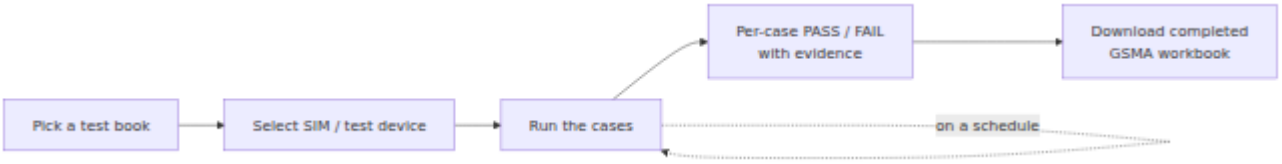
- **Test Books** — The available books (IR.38, IR.48, VoLTE) and every test case they contain
- **Running Tests** — Creating a book, running it, customising the case set, and scheduling routine runs
- **Test Book Explorer** — Inspecting what each book tests and how each case is driven
- **Completed Test-Book Documents** — How OmniRoam generates the filled GSMA workbook

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.

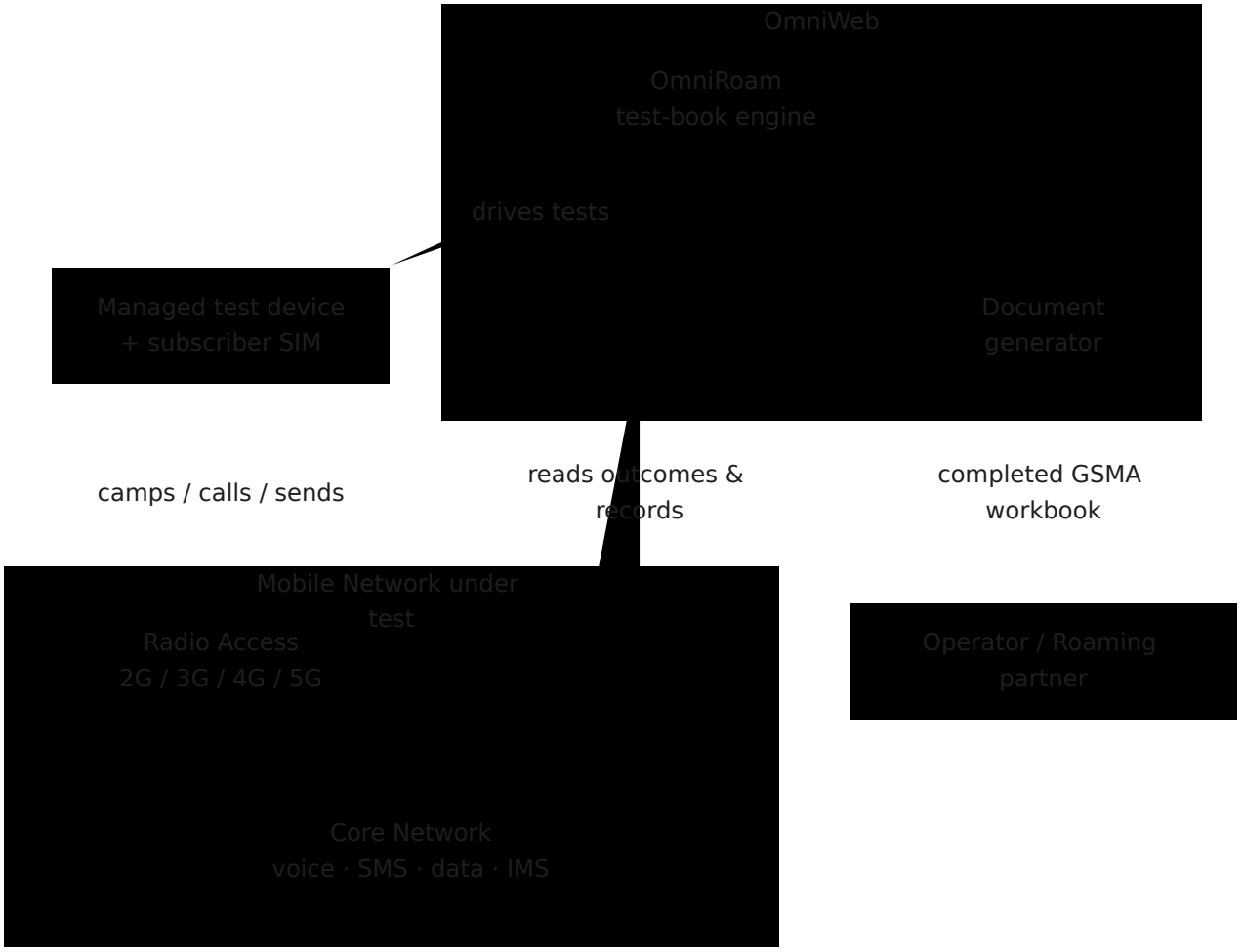
OmniRoam automates three things:

1. **Driving the test** — placing the call, sending the SMS, running the data session, or provisioning the supplementary service on a test device.
2. **Judging the result** — comparing what happened against the GSMA expected result and recording **PASS**, **FAIL** or **NOT PERFORMED**, with the supporting evidence (timings, throughput, network release causes, and so on).
3. **Producing the document** — filling 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: `PASS`, `FAIL`, or `NOT PERFORMED`.
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 cannot be induced from a test device alone (for example, the optional CAMEL prepaid negative tests, or a multimedia-messaging delivery) and 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

OmniWeb Android-control MCP server

An MCP server that wraps the OmniWeb "Android middleware" (the `omniweb-android` Flask service, `/api/android/*`) so an MCP client — Claude Code, Claude Desktop, or a custom client — can drive the lab's Android phones and USB modems.

It is a thin HTTP client over the existing REST API (it imports no Flask internals), so it works against any reachable `omniweb-android` instance.

Tools exposed

Tool	Signature	What it does
`list_devices`	`()`	All known devices (phones + modems), on
`device_info`	`(serial)`	Full details for one device.
`screenshot`	`(serial)` → PNG image	Current screen (universal: scrcpy phones A devices).
`tap`	`(serial, x, y)`	Tap at pixel coordinates.
`swipe`	`(serial, x1, y1, x2, y2, ms=200)`	Swipe/drag/fling.
`input_text`	`(serial, text)`	Type into the focused field.
`key`	`(serial, name)`	Press back/home/recent/enter/del/power/menu/v
`adb_shell`	`(serial, command, timeout=30)` → `{stdout, stderr, return_code}`	Arbitrary `adb shell` (power tool).
`list_scripts`	`()`	The device test-script library.
`run_script`	`(serials, file, params={})`	Fan a script across devices; returns run har

Tool	Signature	What it does
`get_run`	`(run_id)`	Result of a script run (status/stdout/return_
`at_command`	`(serial, command, read_ms=0, timeout_ms=0)`	Raw AT command to a USB modem.
`send_sms`	`(serial, number, text)`	Send an SMS from a modem.

Configuration (environment variables)

Backend target:

- `OMNIWEB_API_BASE` — base URL of the android service. Default `http://127.0.0.1:5002`. The `/api/android` prefix is added automatically.

Backend auth (pick ONE):

- `OMNIWEB_JWT` — a pre-minted backend access token (Bearer), OR
- `OMNIWEB_JWT_SECRET` — the backend's `JWT_SECRET_KEY`. The server then mints a short-lived admin JWT itself and auto-refreshes it. `OMNIWEB_JWT_TTL` (seconds, default 3600) controls its lifetime.

This MCP server's own front door:

- `MCP_TRANSPORT` — `http` (default, streamable HTTP for remote clients) or `stdio` (local `claude mcp add`).
- `MCP_HOST` / `MCP_PORT` — bind address for http transport. Default `127.0.0.1:5005`.
- `MCP_AUTH_TOKEN` — bearer token a remote MCP client must present. **Required to bind to anything other than localhost** — the server

refuses a non-loopback bind without it (fail closed). On `/mcp` the token is checked by middleware; `/` and `/health` are open for health checks.

Run it

venv (self-contained)

```
cd /opt/omniweb-mcp           # or wherever you put server.py + requirements.txt
python3 -m venv venv
venv/bin/pip install -r requirements.txt

export OMNIWEB_API_BASE=http://127.0.0.1:5002
export OMNIWEB_JWT_SECRET=<the backend JWT_SECRET_KEY>
export MCP_TRANSPORT=http
export MCP_HOST=0.0.0.0
export MCP_PORT=5005
export MCP_AUTH_TOKEN=<a long random token>
venv/bin/python server.py
```

systemd (production)

A unit is provided at `packaging/omniweb-android-mcp.service`. It reads `/etc/omniweb/omniweb.env` (for `JWT_SECRET_KEY` → it is re-exported as `OMNIWEB_JWT_SECRET`, see below) plus `/etc/omniweb/android-mcp.env` for the MCP-specific secrets. Create that drop-in (chmod 600):

```
# /etc/omniweb/android-mcp.env
OMNIWEB_JWT_SECRET=<same value as JWT_SECRET_KEY in omniweb.env>
MCP_AUTH_TOKEN=<a long random token>
```

Then:

```
sudo cp packaging/omniweb-android-mcp.service /etc/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable --now omniweb-android-mcp
```

> The backend env file names the JWT secret `JWT_SECRET_KEY`; this server reads `OMNIWEB_JWT_SECRET`. Set `OMNIWEB_JWT_SECRET` explicitly in `android-mcp.env` (copy the value) so the names line up.

Connect from Claude Code

Remote (streamable HTTP), with auth:

```
claude mcp add --transport http android-control
http://<host>:5005/mcp \
  --header "Authorization: Bearer <MCP_AUTH_TOKEN>"
```

Local (stdio), pointing the server at a backend and minting its own JWT:

```
claude mcp add android-control \
  -e OMNIWEB_API_BASE=http://127.0.0.1:5002 \
  -e OMNIWEB_JWT_SECRET=<backend JWT_SECRET_KEY> \
  -e MCP_TRANSPORT=stdio \
  -- /opt/omniweb-mcp/venv/bin/python /opt/omniweb-mcp/server.py
```

Then in Claude Code: `/mcp` to confirm it connected, and try `list_devices`.

Connect from Claude Desktop

`claude_desktop_config.json` — stdio launch:

```
{
  "mcpServers": {
    "android-control": {
      "command": "/opt/omniweb-mcp/venv/bin/python",
      "args": ["/opt/omniweb-mcp/server.py"],
      "env": {
        "OMNIWEB_API_BASE": "http://127.0.0.1:5002",
        "OMNIWEB_JWT_SECRET": "<backend JWT_SECRET_KEY>",
        "MCP_TRANSPORT": "stdio"
      }
    }
  }
}
```

For a remote streamable-HTTP server from Claude Desktop, use an `mcp-remote` bridge or the HTTP server config supported by your Desktop version, pointing at `http://<host>:5005/mcp` with the `Authorization: Bearer <MCP_AUTH_TOKEN>` header.

Security notes

- Device control requires the configured `MCP_AUTH_TOKEN` on the HTTP transport. With no token the server only allows a loopback bind and prints a warning; a non-loopback bind without a token is refused.
- The server authenticates to the backend as an admin (minted JWT or supplied token), so anyone who can reach `/mcp` with the bearer token has full device control. Treat `MCP_AUTH_TOKEN` like a password and front it with TLS (e.g. via nginx) for off-host access.

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**, paste a bearer JWT, and you can try calls live against the running instance.
- **OpenAPI 3 spec:** `/api/openapi.json` — the machine readable document, for importing into Postman/Insomnia or generating clients.

← Back to Operations Guide

What's documented

The spec is **generated at runtime** by introspecting the live route map, so it always matches the endpoints the running instance actually serves — there is

no hand-maintained file to 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.

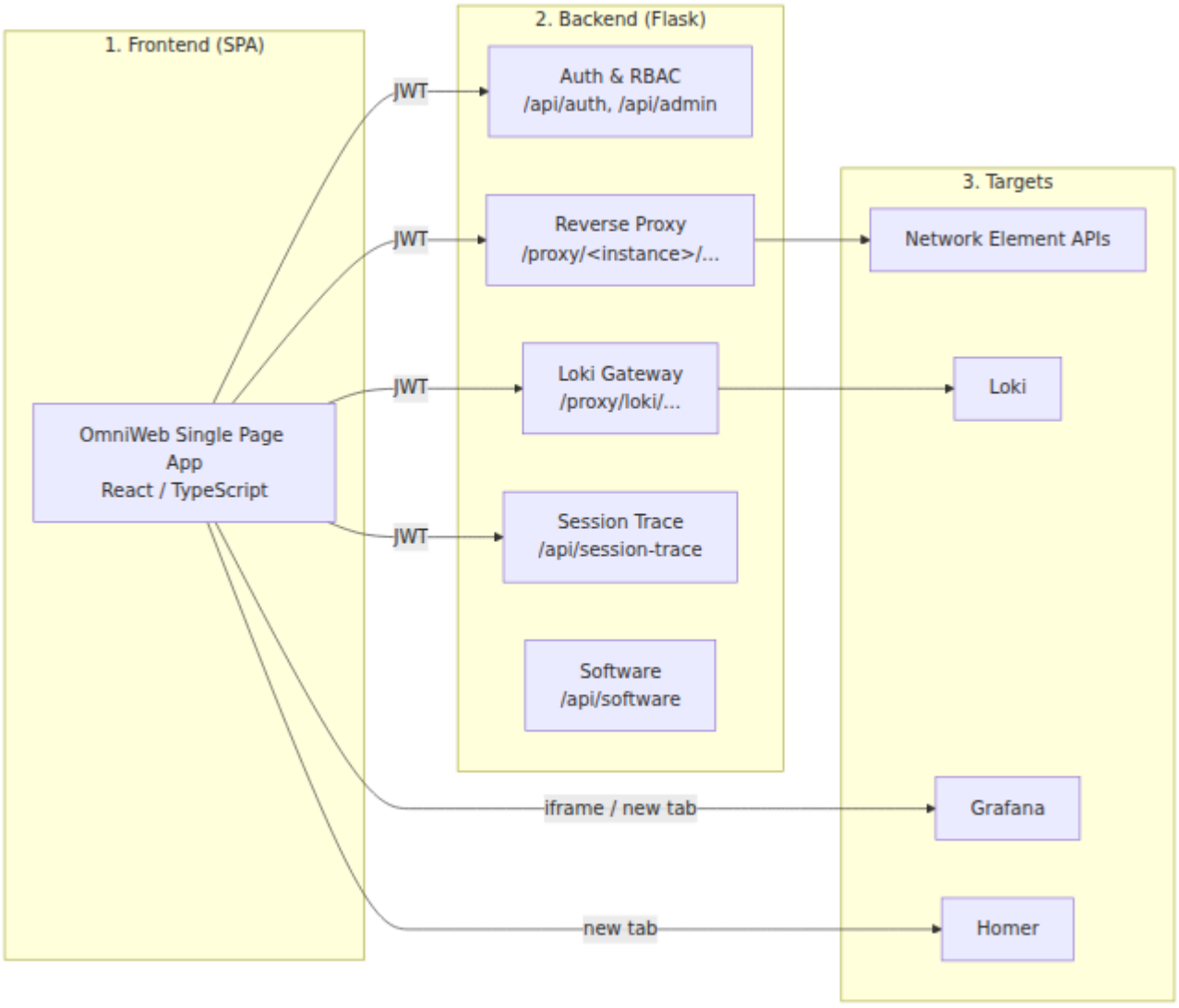
OmniWeb Architecture

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

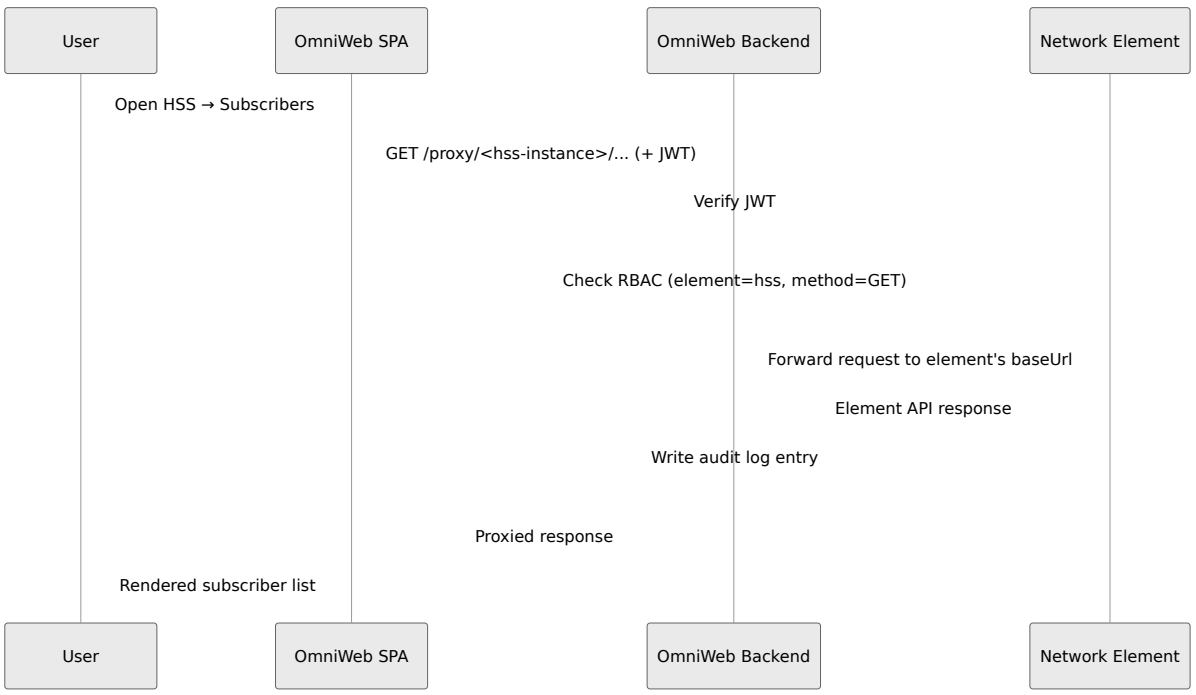
OmniWeb has three moving parts:



Component	Role
Frontend SPA	The user interface. A React/TypeScript single-page application that renders element pages, topology, logs, traces, and embedded dashboards. Served by Vite in development and nginx in production.
Backend (Flask)	The security and integration layer. It authenticates users, enforces RBAC, and reverse-proxies every request to the correct element API, Loki, or Grafana. Runs on port `5001`.
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
`type`	String	Yes	-	Element type. Determines which UI module renders the element. See supported types.
`displayName`	String	Yes	-	Human-readable name shown in the sidebar and topology.
`description`	String	No	-	Longer description shown in tooltips and overview pages.
`instances`	List	Yes	-	One entry per deployed node of this type. See instance parameters below.

Parameter	Type	Required	Default	Description
`defaults.pollIntervalMs`	Integer	No	5000	How often operational pages auto-refresh, in milliseconds.
`defaults.timeoutMs`	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
`id`	String	Yes	-	Unique instance identifier. Used in URLs (`/hss/<id>`) and in the proxy path (`/proxy/<id>/...`).
`name`	String	Yes	-	Display name for the instance.
`baseUrl`	String	Yes	-	The element's management API base URL. The proxy forwards requests here. Self-signed TLS is accepted.
`tags`	List	No	`[]`	Free-form labels (e.g. `primary`, `geo-redundant`) 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 `/proxy/<instance-id>/...`. OmniWeb forwards `GET`/`POST`/`PUT`/`PATCH`/`DELETE` as the element expects; the request/response shape is whatever that element defines.
Log query (LogQL)	All (Logs tab)	Routed through `/proxy/loki/...` 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 nginx and embedded as an iframe. nginx authorises each request against OmniWeb's JWT using `auth_request`. 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 — also authorised via `auth_request`.

Environment Reference

Backend integration endpoints are configured via environment / `backend/config.py`:

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).

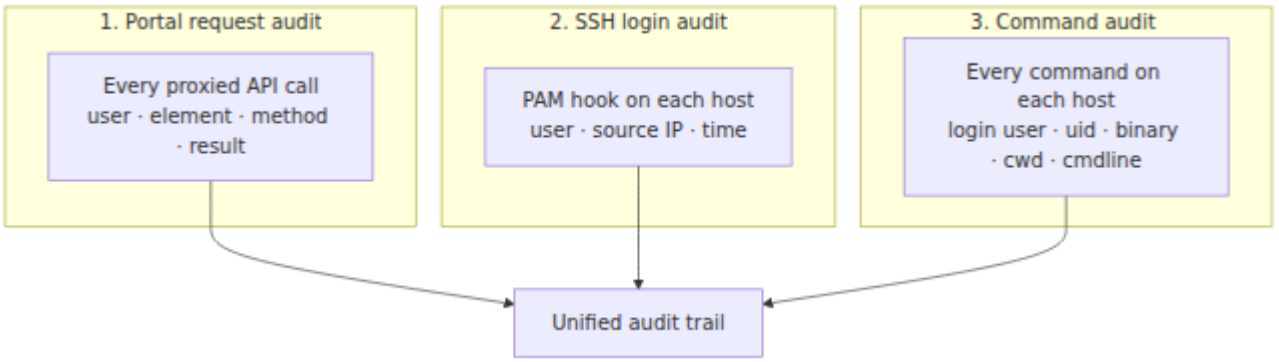
See Authentication & Access Control for the LDAP/SSH and audit details.

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 *through* OmniWeb	Settings → Audit
SSH login audit	Every SSH login *to a managed host*	Settings → Audit (via host callback)
Command audit (snoopy)	Every command *run on a host*	Loki (searchable in element/host logs)

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:

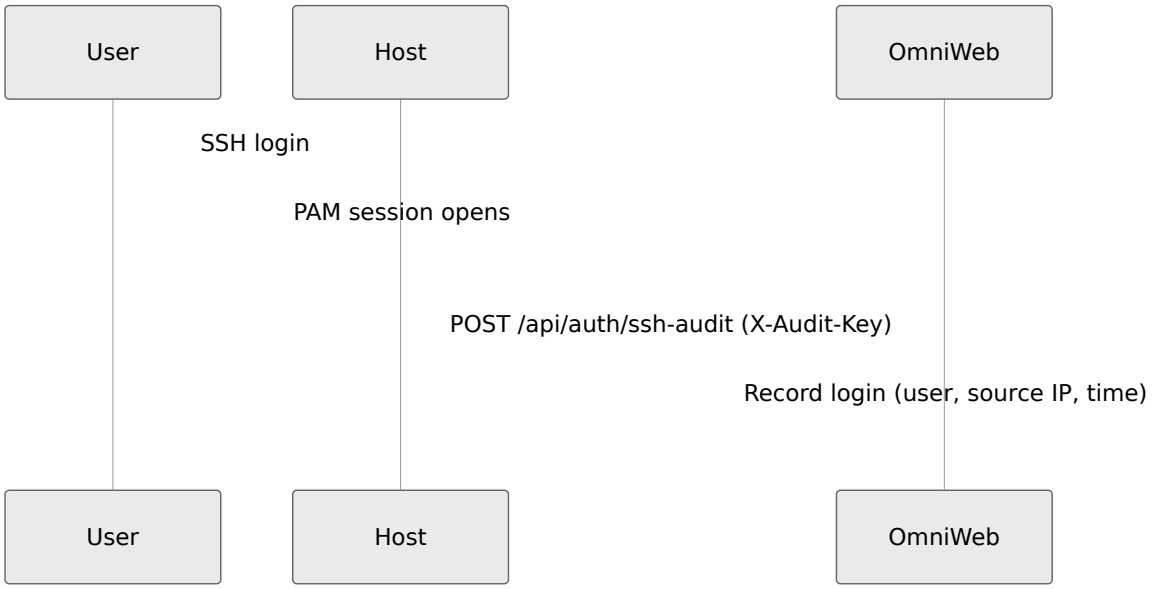
- The **user** who made the request.
- The **element type and instance** targeted.
- The **HTTP method** (which tells you whether it was a view, a change, or a delete).
- The **result** (status code).
- The **timestamp**.

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
`AUDIT_API_KEY`	`omniweb-audit-key`	Shared secret the host PAM hook presents (as `X-Audit-Key`) 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 Logging

For full command-level visibility on the hosts, OmniWeb's hosts run **snoopy**, which intercepts every `execve()` call via `LD_PRELOAD`. Because it works in userspace it functions inside LXC containers without kernel audit support — capturing far more than login events alone.

Snoopy output is routed to `/var/log/omniweb-audit.log` and shipped to Loki, so it is searchable from OmniWeb the same way as any other structured log. Each entry captures:

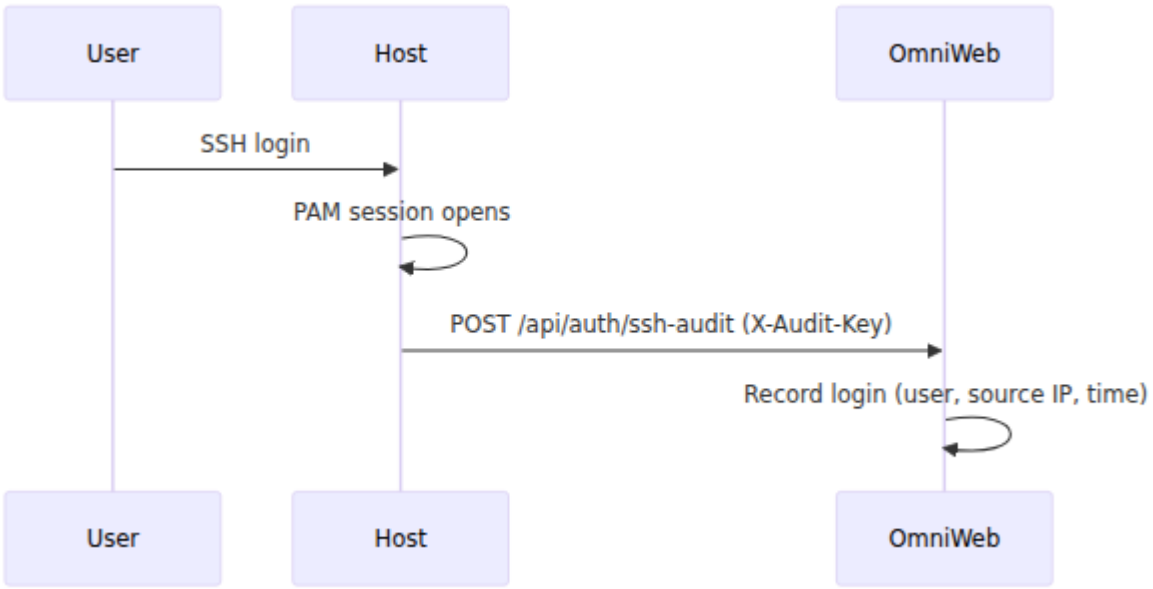
Field	Meaning
`login`	The user who originally logged in — **survives** `sudo`/`su`, so privilege changes don't hide who is acting
`uid`	Effective UID the command actually ran as
`filename`	The binary that was executed
`cwd`	Working directory at the time of execution
(after `]:`)	The full command line

A representative line:

```
2026-04-16T13:18:35 omni-stp01 snoopy[33665]: [login:nickj uid:0
tty:(none) cwd:/home/nickj filename:/usr/bin/sudo]: sudo tail -10
/var/log/omniweb-audit.log
```

The `login:` field is what makes this powerful: even after `sudo` to root, the audit trail still shows the human who ran the command.

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 — where snoopy command audit is searched.
- LDAP & SSH Key Sync — the access side of host activity.
- RBAC — the permissions whose use the portal audit records.

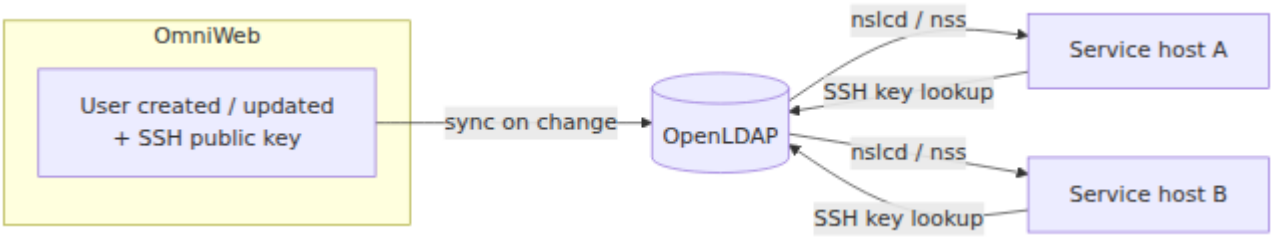
LDAP & SSH Key Sync

OmniWeb's access control can extend beyond the portal to the **hosts** that run the network. By synchronising users and their SSH public keys into an LDAP directory, OmniWeb becomes the single place to manage *who can SSH where* — and, combined with audit logging, who actually did.

This integration is **optional**. If the LDAP settings are not configured, OmniWeb runs normally and simply does not sync.

← Authentication & Access Control · ← Operations Guide

What It Does



When LDAP sync is enabled, every user created or updated in OmniWeb — including any change to their SSH public key — is written to the LDAP directory automatically. Service hosts configured to read from that directory then authorise SSH logins and public-key lookups against it. Add a key in OmniWeb, and the user can SSH to every enrolled host; remove it, and that access is gone everywhere at once.

Configuration

LDAP sync is controlled by four backend settings:

Variable	Default	Description
`LDAP_URI`	`ldap://localhost:389`	OpenLDAP server URI.
`LDAP_BASE_DN`	`dc=omniweb,dc=local`	Base distinguished name of the directory.
`LDAP_ADMIN_DN`	`cn=admin,dc=omniweb,dc=local`	Bind DN OmniWeb uses to write entries.
`LDAP_ADMIN_PASSWORD`	`omniweb-admin`	Bind password. **Change this** — the default must never be used in production.

With these set, OmniWeb writes user entries (with their `sshPublicKey` attribute) into LDAP under the users branch of the base DN.

What Gets Synced

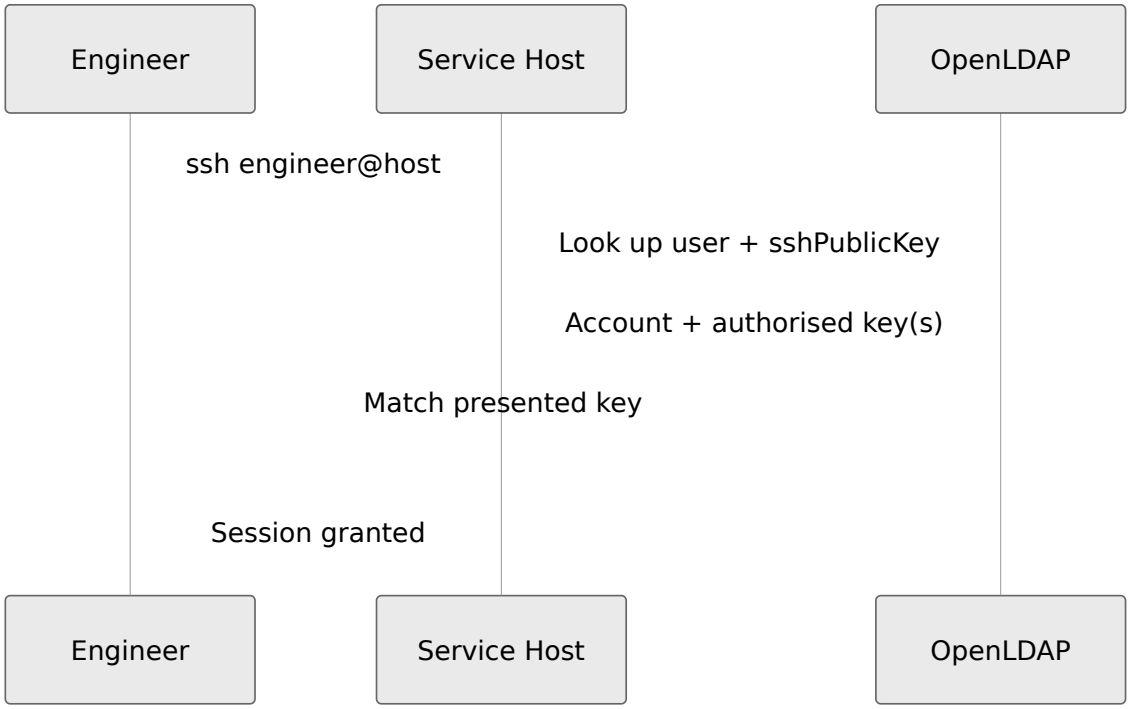
OmniWeb data	LDAP representation
Username	`uid` of the user entry
SSH public key(s)	`sshPublicKey` attribute (via the `ldapPublicKey` object class)
User identity	Standard POSIX account attributes so hosts can resolve the user

Storing SSH keys in LDAP relies on the `openssh-lpk` schema (the `sshPublicKey` attribute and `ldapPublicKey` object class) being present in the directory.

Settings → Users — each user's details including whether an SSH public key is configured. Adding or changing a key here is what syncs to LDAP and, from there, to every enrolled host.

How Hosts Consume It

Each service host is configured to use LDAP as a name service and for SSH key lookup:



On the host side this means:

- **Name resolution** (`passwd`/`group`/`shadow`) is backed by LDAP, so OmniWeb-managed users exist on the host.
- **SSH key authorisation** is served from LDAP, so the keys OmniWeb stores are the keys that grant access — there are no per-host `authorized_keys` files to maintain.

Why This Matters

Centralising host access in OmniWeb gives you:

- **One place to grant/revoke SSH access** across the whole fleet.
- **No scattered key files** drifting out of sync on individual hosts.
- **A single identity** that ties a person's portal access, their SSH access, and — through audit logging — their recorded activity together.

Related

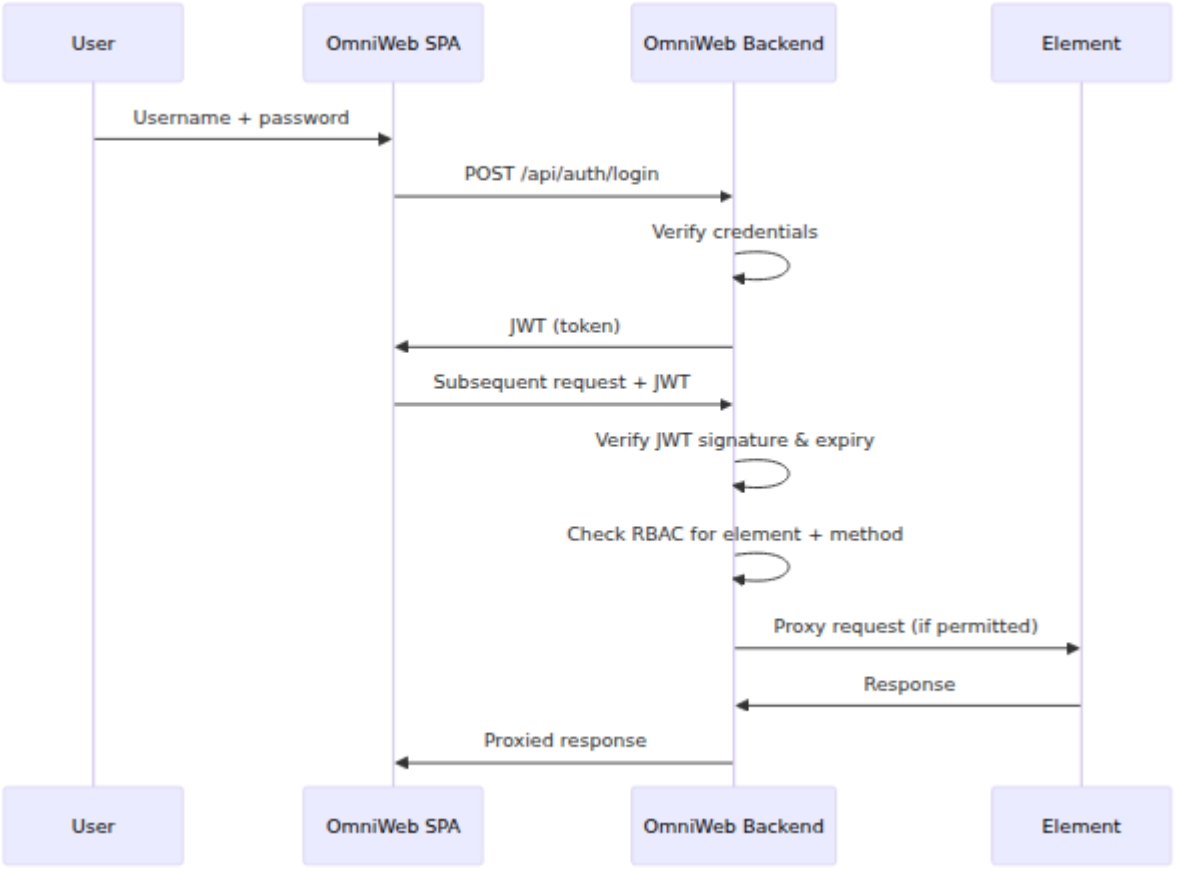
- Audit Logging — record SSH logins and every command run on the hosts.
- RBAC — control what the same users can do *inside* OmniWeb.

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 — and, 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 (`access_token`, HttpOnly)	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.

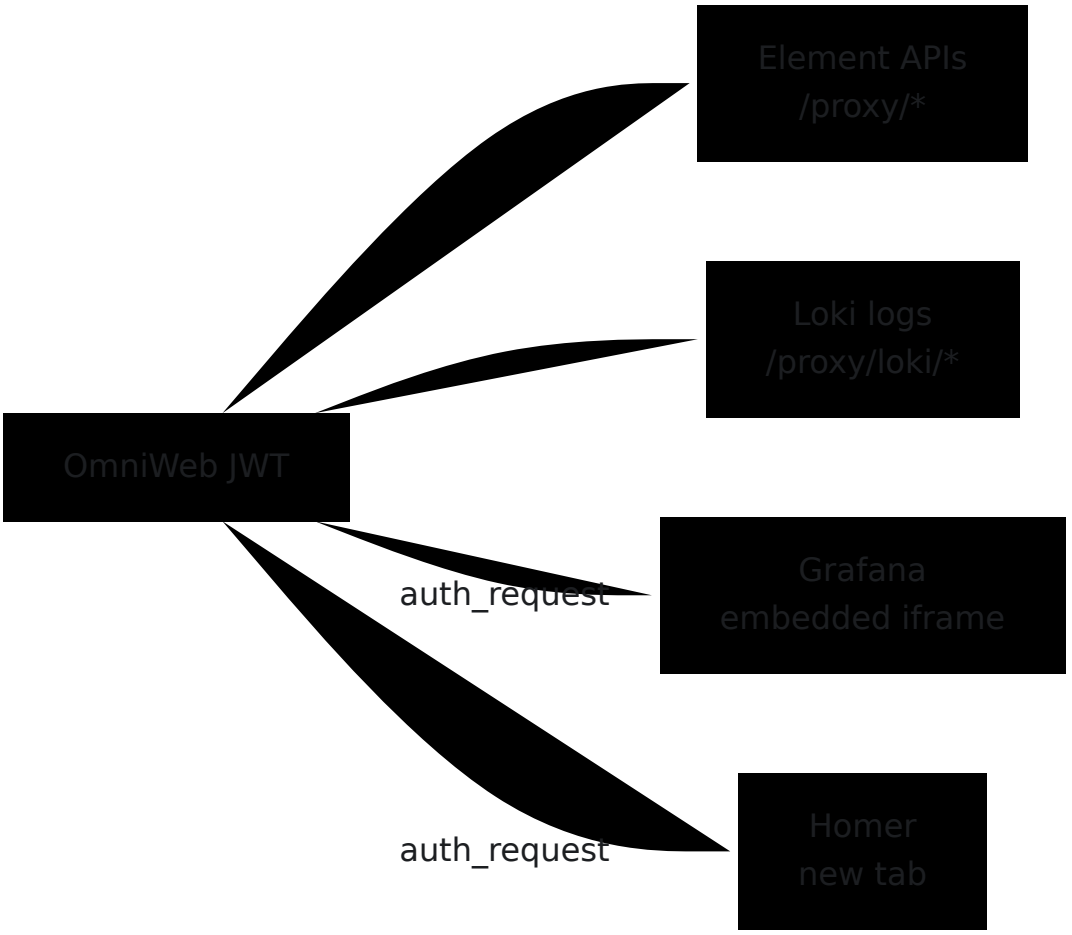
Token Lifetime

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
`JWT_ACCESS_TOKEN_EXPIRES`	`86400` (24 h)	Access-token lifetime in seconds. Lower it for tighter sessions; raise it for fewer re-logins.
`JWT_SECRET_KEY`	*(generated)*	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 nginx, which verifies the JWT using the `auth_request` mechanism before proxying. 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.

← Authentication & Access Control · ← Operations Guide

Permission Levels

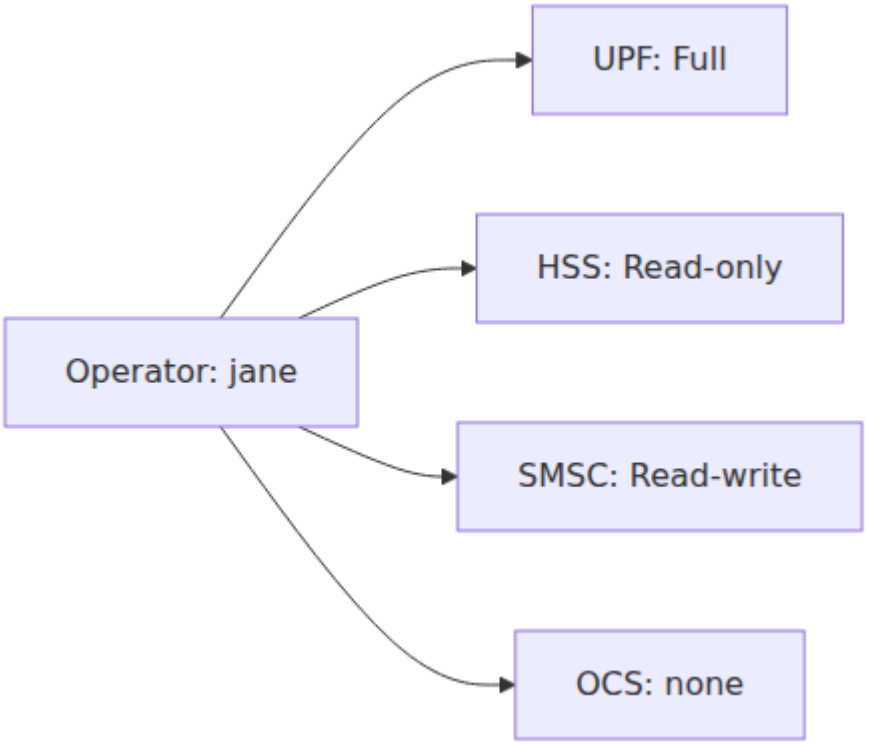
Each element type can be granted to a user at one of three levels. The level is expressed as the set of HTTP methods the user may use against that element through the proxy:

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:



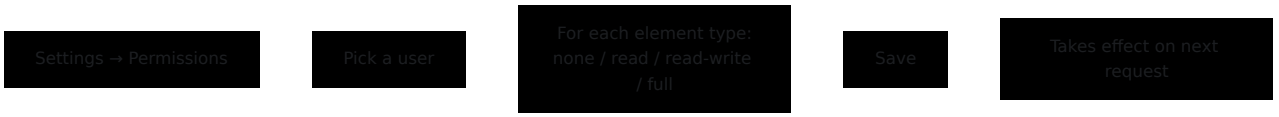
In this example Jane can clear UPF sessions, can view but not change the HSS, can add SMSC routes, and cannot see the OCS at all. This lets you give, say, a dataplane engineer full control of the UPF while keeping subscriber data read-only.

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).

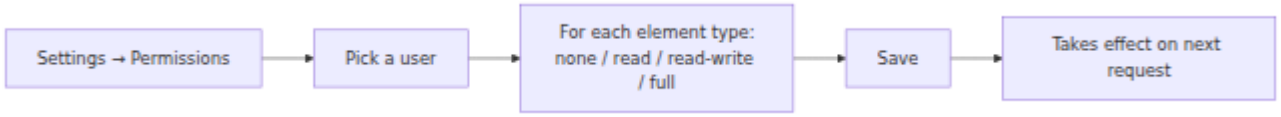


How Enforcement Works

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.

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



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.

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, optional LDAP/SSH-key sync for host access, and a unified audit trail spanning the portal and the hosts it manages.

This area is broken into focused pages:

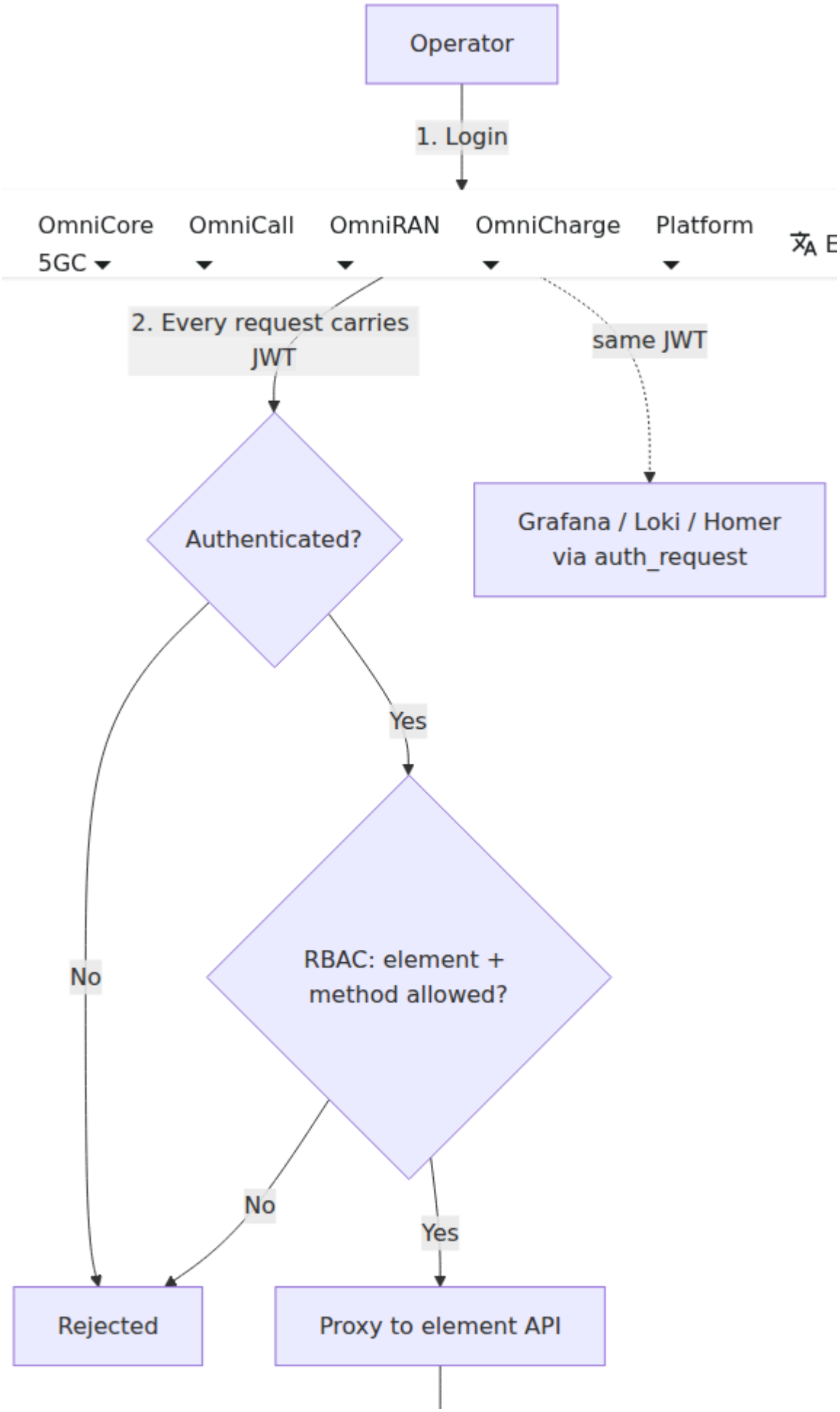
← Back to Operations Guide

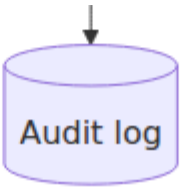
In This Section

- **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.
- **LDAP & SSH Key Sync** — Synchronising users and SSH public keys to LDAP so hosts can authorise SSH logins centrally.
- **Audit Logging** — The three layers of audit: portal request audit, SSH-login audit, and snoopy 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.


Operator
1. Login
OmniCore 5GC
OmniCall
OmniRAN
OmniCharge
Platform
2. Every request carries JWT
same JWT
Authenticated?
Grafana / Loki / Homer via auth_request
Yes
No
RBAC: element + method allowed?
No
Yes
Rejected
Proxy to element API



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 — via LDAP/SSH-key sync	LDAP & SSH
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.

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.

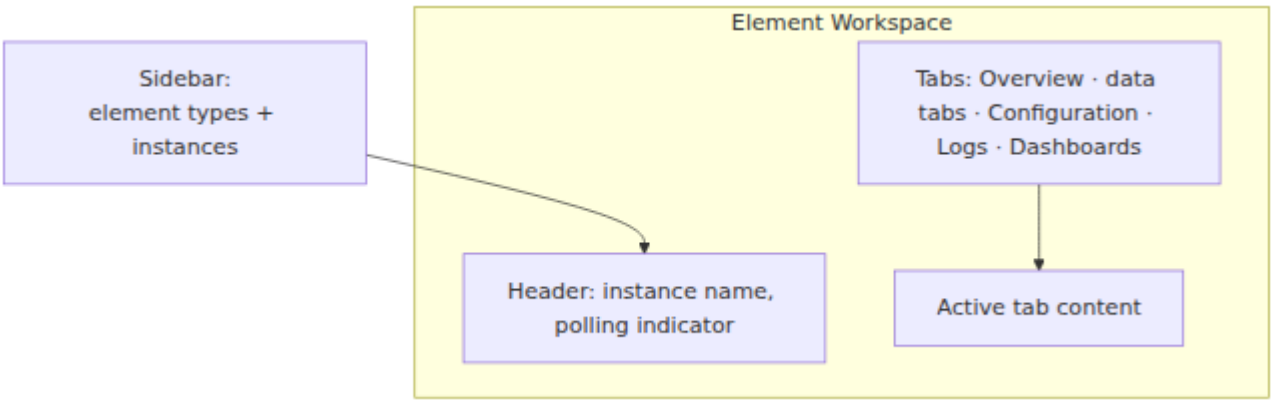
The screenshots below come from specific elements, but they illustrate the *generic* pattern — the same controls appear, with different fields, on every element.

← Back to Operations Guide

Table of Contents

Page Anatomy

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 — 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.

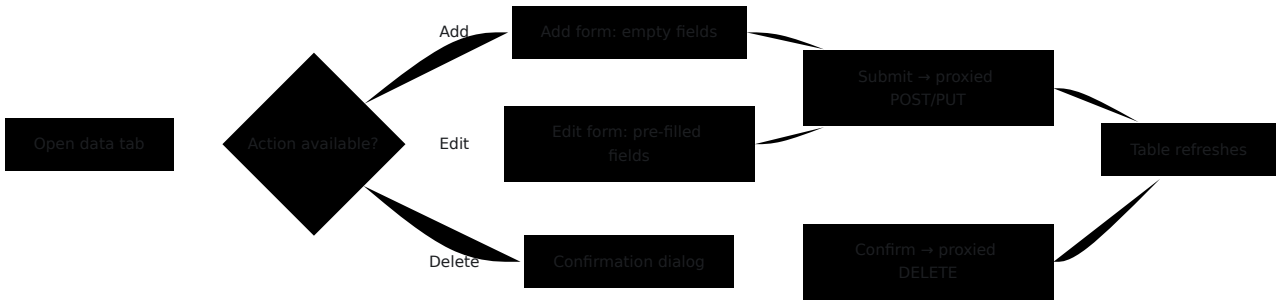
Detail Views

Where a row has more to show, clicking it opens 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 `POST`/`PUT` → 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 `DELETE` → list refreshes

The mechanics — form or confirmation dialog, proxied write, table refresh — are identical. 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 don't 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 a value can't be parsed.
- **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. Pausing polling is useful when you want a stable view while reading 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 for managing 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. Clicking a named instance opens that instance's workspace directly. An instance dropdown also appears in the workspace header, letting you switch between instances without going back to the sidebar — useful when comparing 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

Clicking the element type name in the sidebar (rather than a specific instance) — or selecting **All Instances** from the dropdown — opens 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 drilling 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

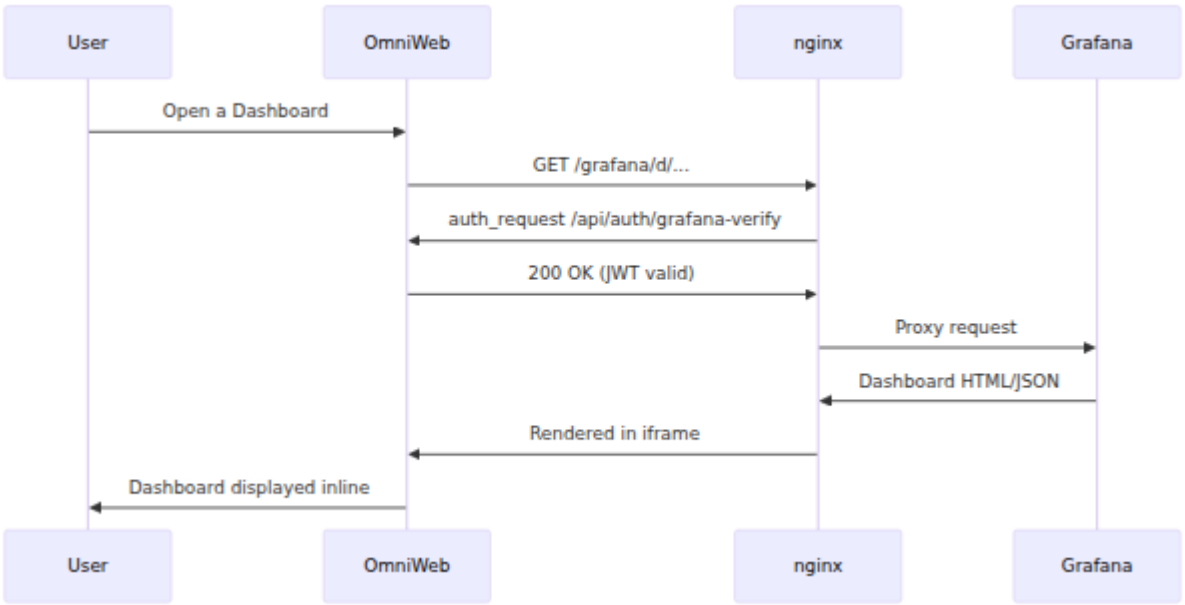
Grafana Dashboards & Statistics

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

← Back to Operations Guide

How It Works

Grafana is served behind nginx 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 using nginx's `auth_request` mechanism against your OmniWeb JWT. There is no second password.



The Grafana instance OmniWeb proxies to is configured by the `GRAFANA_URL` environment variable (default `http://localhost:3000`). In development, Vite proxies `/grafana` to the backend; in production, nginx handles path stripping, proxying, and the `auth_request` check.

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.

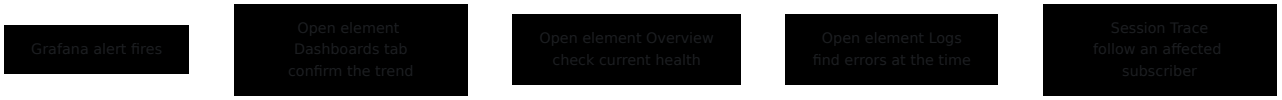
Alerting

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.

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

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 `{ … }`	`{hostname="opt-dev-upf01"}`	Pick which streams (which host/service) to read. Indexed and fast.
Pipeline	After a `|`	`| level="error"`	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 (the Ansible common role configures shipping via Alloy). 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 (snoopy)** — 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. Routed to `/var/log/omniweb-audit.log` and 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.

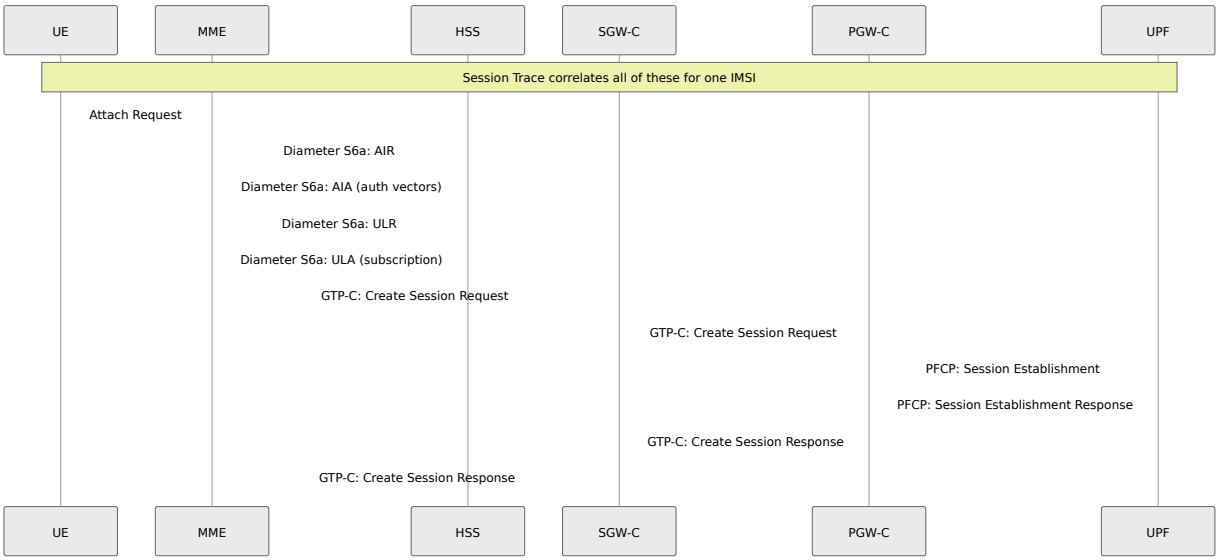
Session Trace — search by IMSI or MSISDN and correlate a subscriber's activity across Diameter, GTP-C, PFCP, SIP, and MAP, with per-protocol tabs across the top.

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
PFCP	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 PFCP 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, useful when you are chasing a protocol-specific problem (e.g. only the SIP exchange of a VoLTE call) rather than following a single subscriber across everything.

Upload PCAP

The **Upload PCAP** tab accepts packet-capture files for offline analysis. Uploaded captures are parsed and indexed and then appear through the same trace interface as live data — useful for analysing captures taken from network taps or element-level traces. The backend tracks each upload as a job you can revisit.

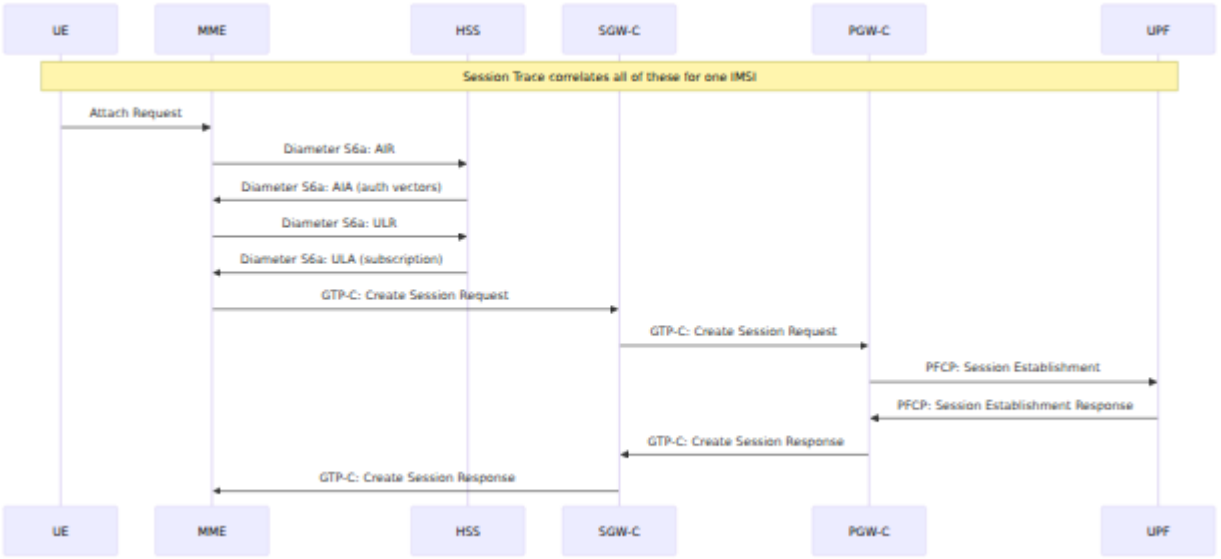
Homer — SIP Capture & Call-Flow Analysis

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 — nothing else changes.



Clicking **Homer** opens the Homer web UI in a new browser tab. Authentication is transparent: nginx verifies the OmniWeb JWT via the same `auth_request` mechanism 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 (Alloy 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 Alloy 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).

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 — 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*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)*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>[,&lt;EARFCN>,&lt;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>,&lt;EARFCN>,&lt;PCI>[,&lt;EARFCN>,&lt;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,&lt;EARFCN>,&lt;PCI>` locks (returns `OK`, lock persists) and `AT+QNWLOCK="common/4g",0` clears it. The `common/lte` *set* forms (`...,1,&lt;EARFCN>,&lt;PCI>` and `...,0`) all return **`ERROR`** on this firmware — `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 << (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`) — it is **not** surfaced in the UI. `AT+QSCAN` is also not supported on this EP06 firmware (it is an RG/RM 5G-series command); the network scan uses `AT+COPS=?` 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
`AT+QCFG="ims"`	Query/enable IMS. `AT+QCFG="ims",1` to force-enable.
`ATD<number>;`	Dial a voice call (trailing `;` = voice, not data).
`ATA`	Answer an incoming call.
`ATH` / `AT+CHUP`	Hang up.
`AT+CLCC`	List current calls: `+CLCC: <id>,<dir>,<state>,<mode>,<mpty>,"<number>",<type>`. Empty `OK` = no calls.
`AT+CLIP=1`	Enable calling-line ID URC (`+CLIP:`) on `RING`.

`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 `min_read`).	`+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.

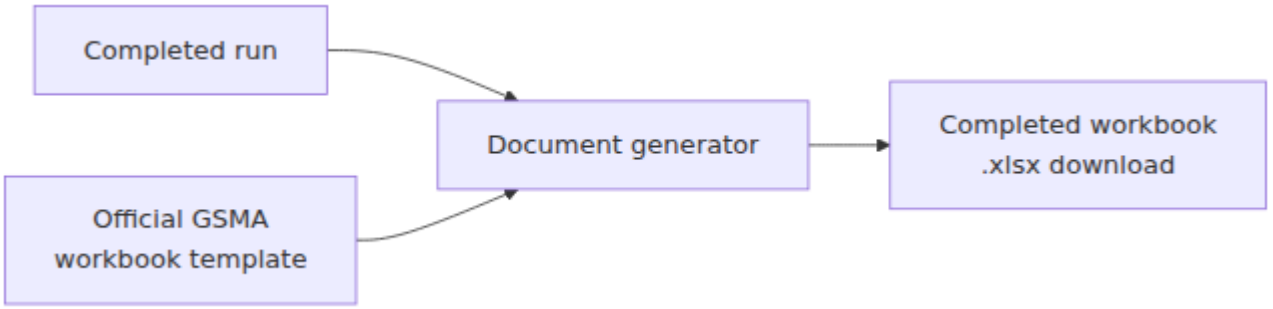
Pragmatic approach used in the UI: the Events tab enables the URCs above, then **polls** the modem on an interval (`AT+CEREG?`, `AT+CLCC`, `AT+QENG`, `AT+CSQ`, `AT+CMGL`) and renders state changes into a timestamped log. Each poll also uses a short `min_read` window so any URC that happens to arrive 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.

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. It is produced with a single **Download Test Book** action and requires 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** — and 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

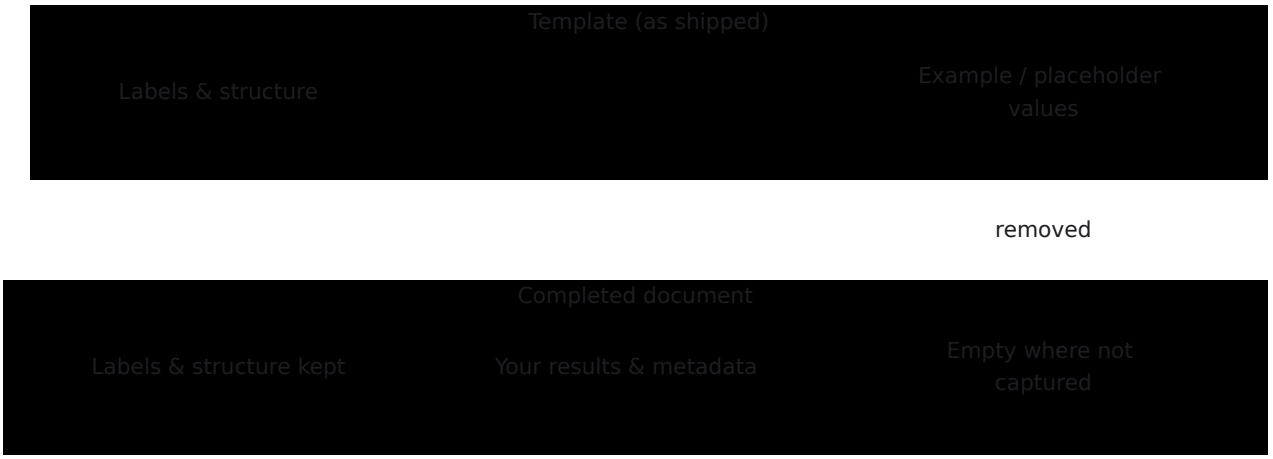
3. Measured values

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 generating 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

The document can be downloaded 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, regenerating after a re-run simply produces an updated document.

Manual Cases in the Document

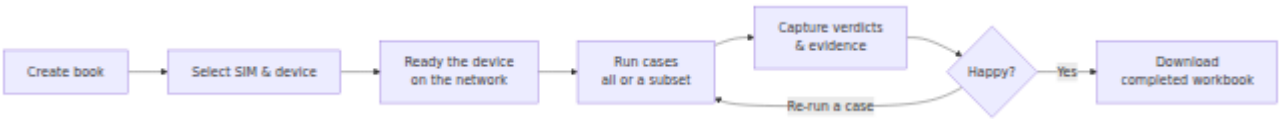
Cases that OmniRoam cannot drive automatically still appear in the workbook with their case number and title, ready for the tester to record a result by hand. This keeps the delivered document complete: the automated cases are filled with real evidence, and the manual cases are presented in the right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

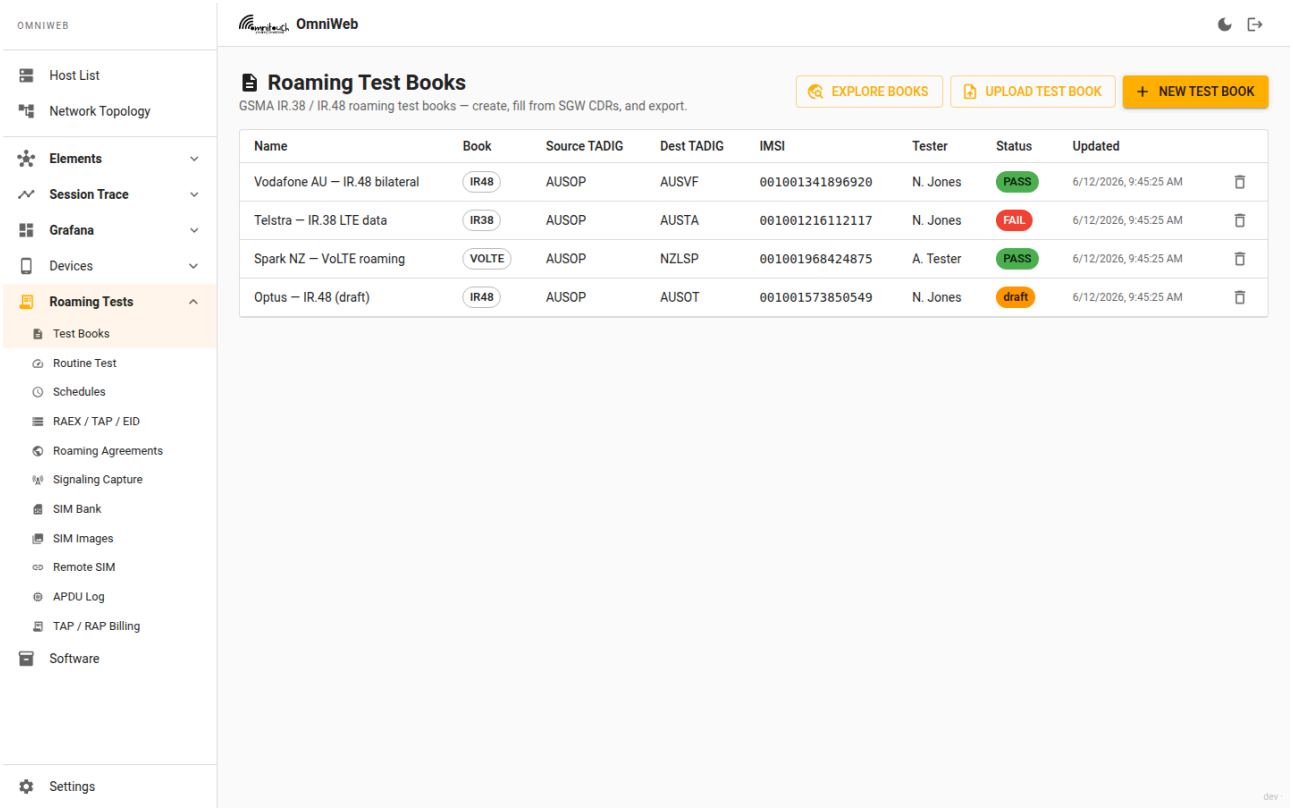
Running OmniRoam Tests

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



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:

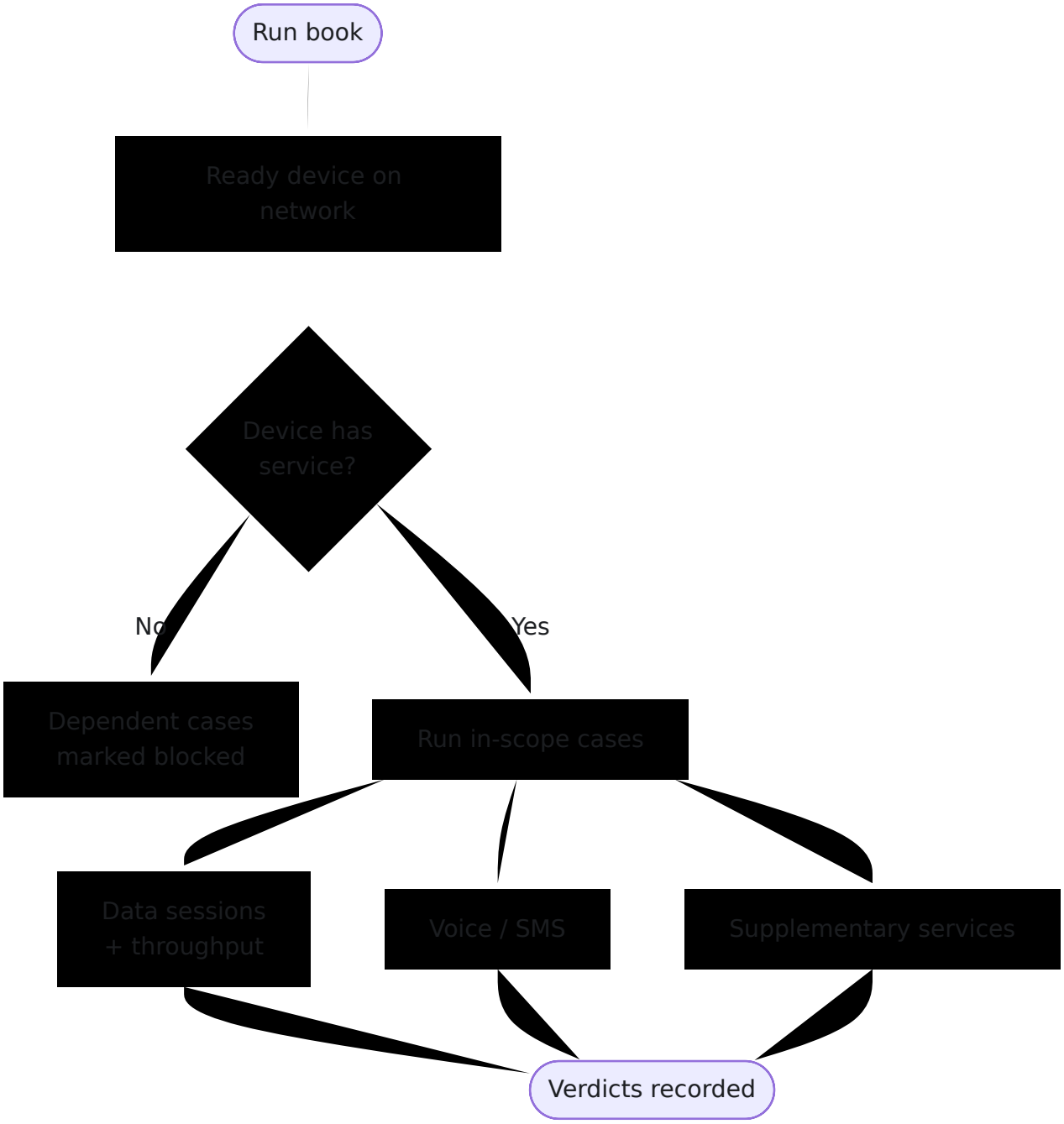
Field	Purpose
Book type	Which GSMA book to run — IR.38, IR.48, or VoLTE. This selects the case set.
Roaming partner / PLMNs	The home (PLMN a) and visited (PLMN b) networks. Used to populate the document header and to pick the correct throughput threshold.
SIM	The subscriber SIM to test with. Its IMSI 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 — presenting the SIM and confirming 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 running 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 relying on values typed in by hand.



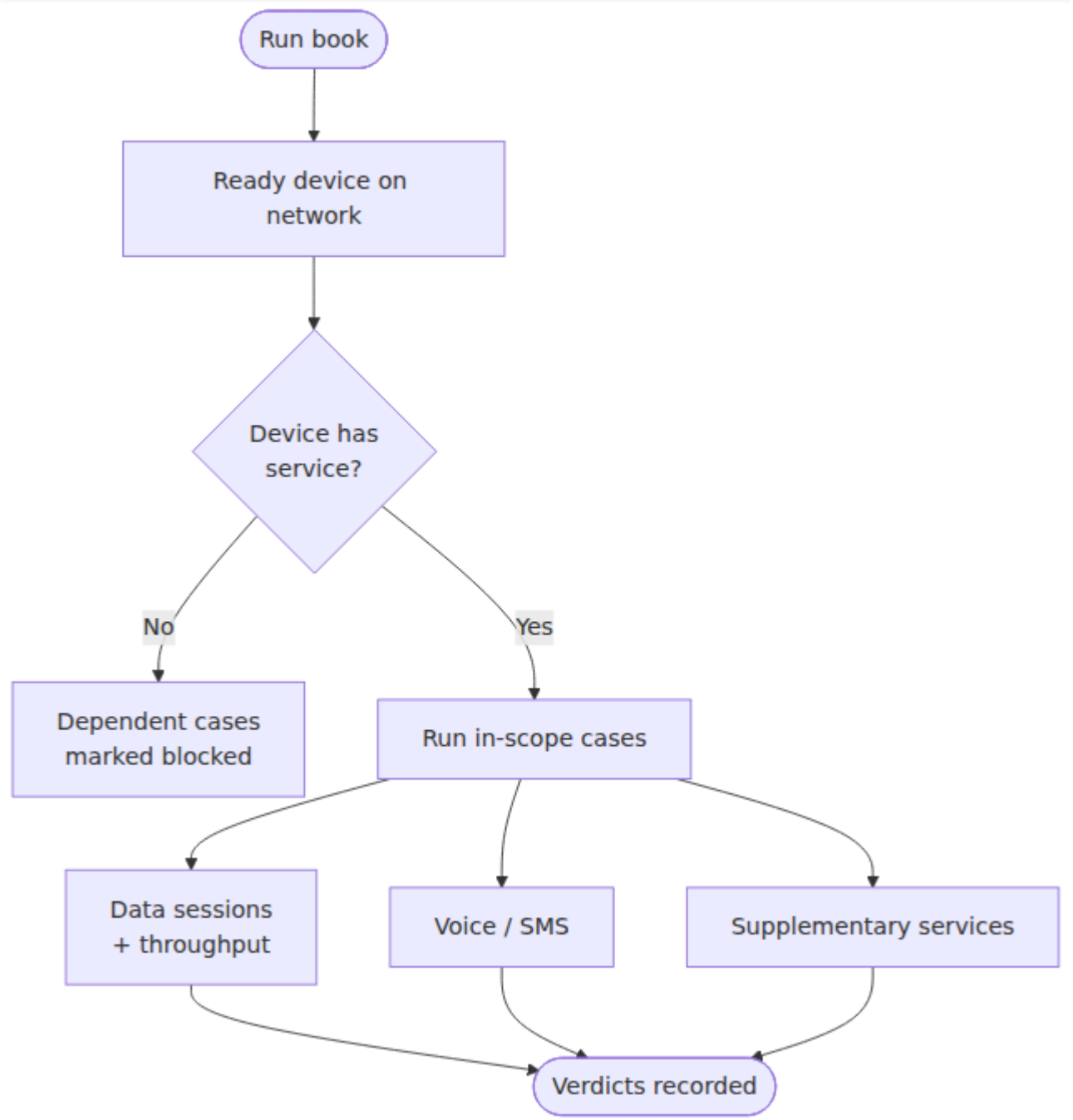
While a book runs, the screen shows each step as it happens, with the current step highlighted and a live pass/fail indication per case.

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, and — for the IR.38 speed case — the measured rate is asserted against the throughput floor for the home/visited distance.



Voice calls

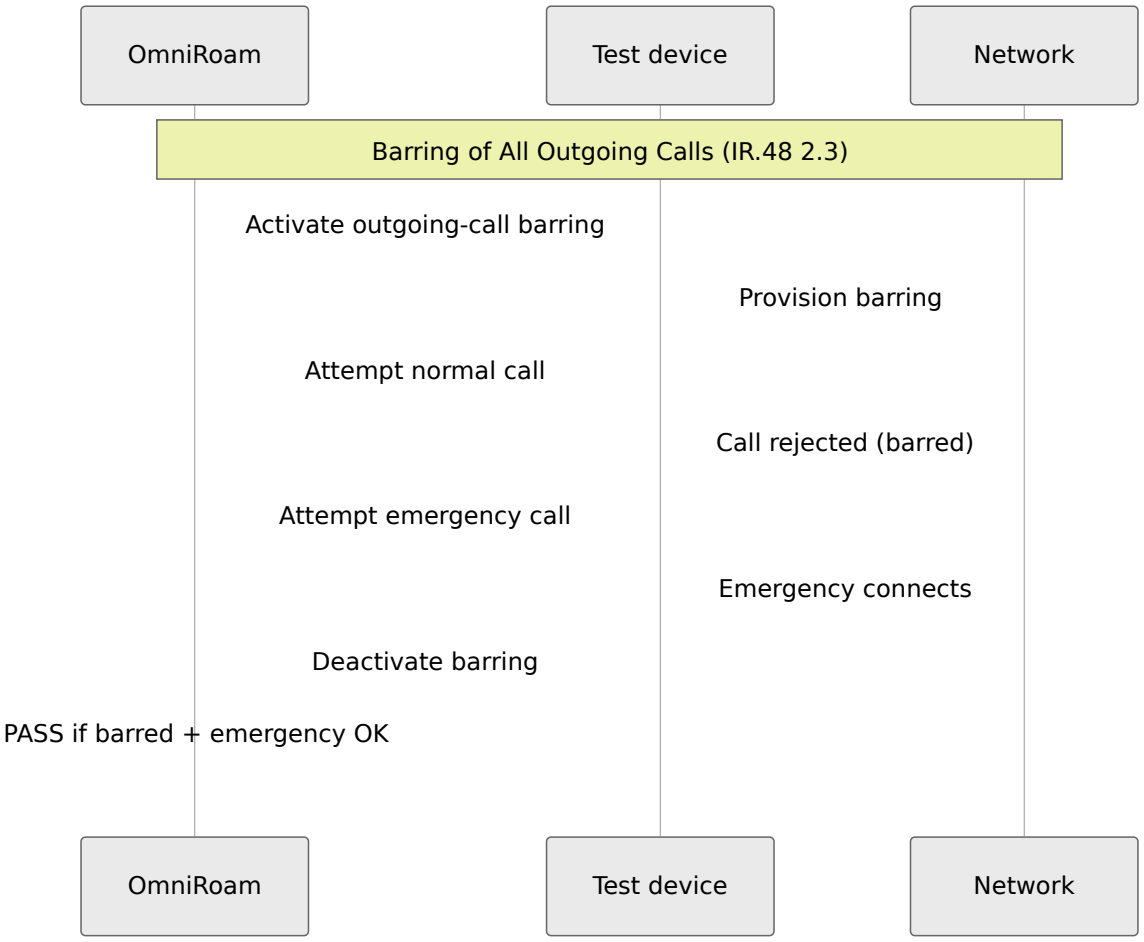
OmniRoam places a call from the device to a lab answering service, confirms the call reaches the active (answered) state, optionally holds it to confirm stability, then clears it. The verdict records the time to connect and whether the call stayed up.

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)

Supplementary services are provisioned over the air from the test device itself — exactly as a subscriber would using the standard service control codes — and then exercised:



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 — confirming 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 — and 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
`omniweb_ir38_routine_pass`	gauge	Latest sche result (1 = p
`omniweb_ir38_routine_check_pass`	gauge	Per-check re latest run (l data/speedt
`omniweb_ir38_routine_download_mbps`	gauge	Download ra on the lates
`omniweb_ir38_routine_upload_mbps`	gauge	Upload rate the latest ru
`omniweb_ir38_routine_latency_ms`	gauge	Latency me latest run
`omniweb_ir38_routine_last_run_timestamp_seconds`	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 can't carry one) and can be gated with a bearer token when required.

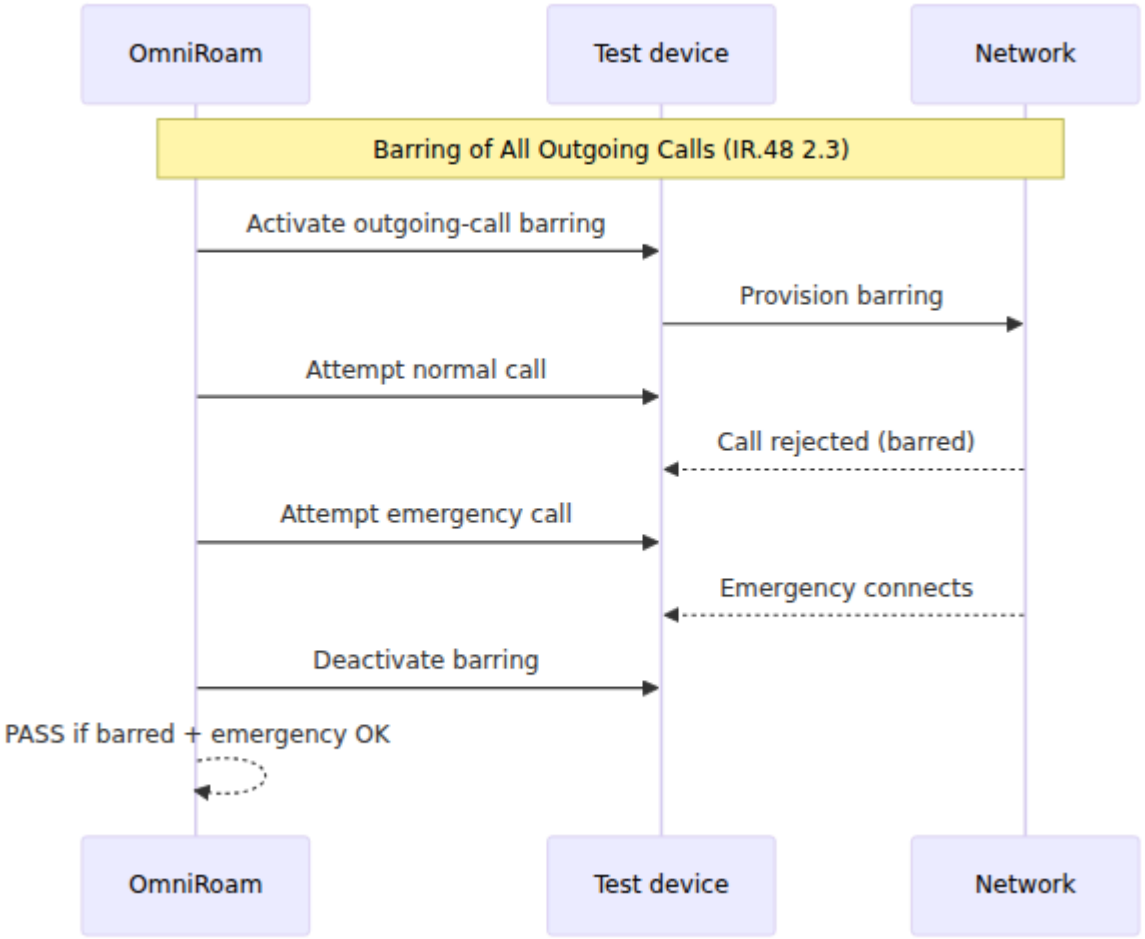
Example alert: roaming voice has been failing on a device for 15 minutes:

```
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.

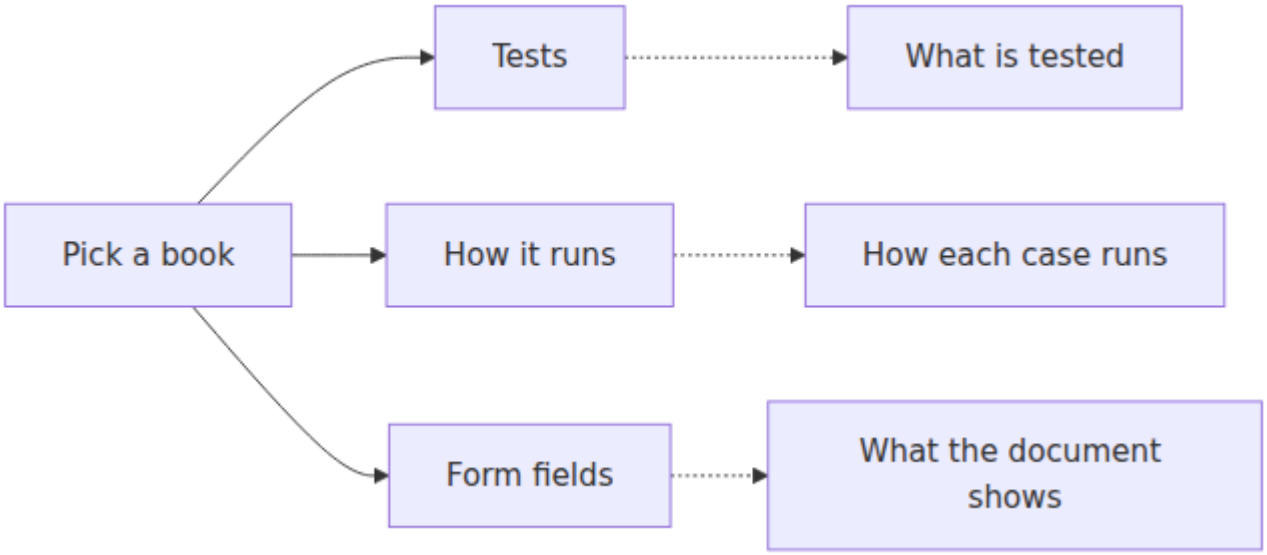
Test Book Explorer

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 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.

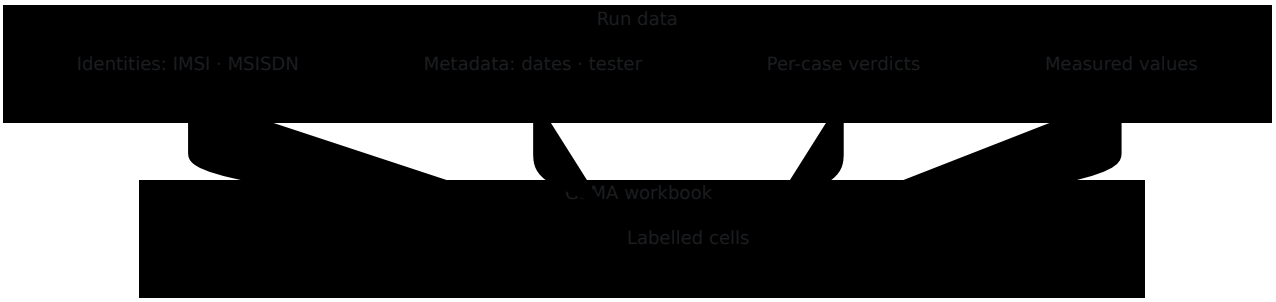
This view is useful for understanding the shape of a run and for confirming that a customised scope will perform the steps you expect.

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 — plus 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`	LTE speed test (data + throughput)	Auto	A data session establishes through the home gateway, carries traffic, and meets the throughput floor.
`3.1.2`	LTE Cancel Location	Auto	The home network can cancel the subscriber's location on the visited network (triggered from the home subscriber server).
`3.1.3`	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.

The status of every supplementary service is reported using the four standard 3GPP flags — **Provisioned**, **Registered**, **Active**, and **Quiescent** (see 3GPP TS 23.011).

VoLTE — Roaming VoLTE Testing

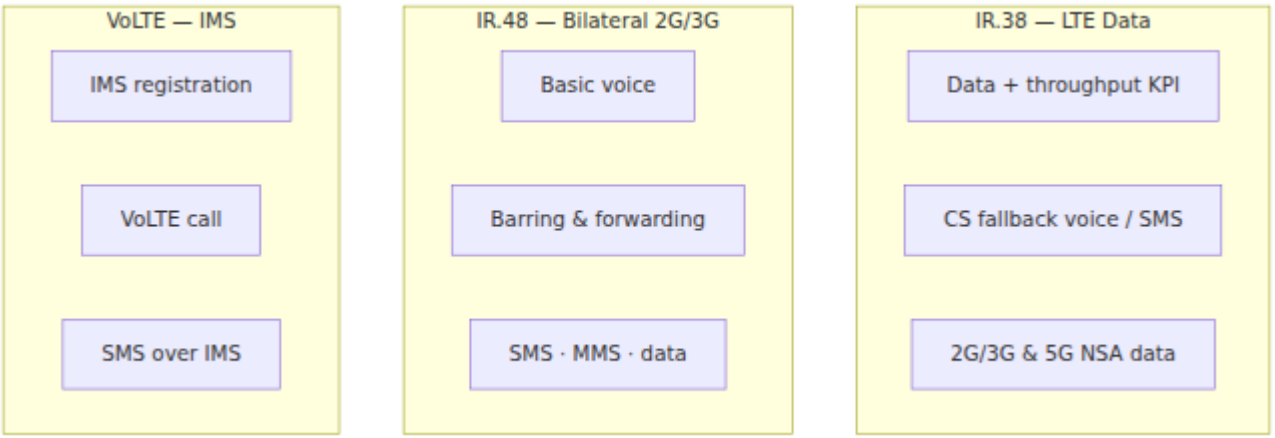
Based on **GSMA BA.65 §5.3**, with VoLTE behaviour per **GSMA IR.92**. Proves the subscriber can register to IMS while roaming and place/receive VoLTE calls and SMS-over-IMS.

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 by sending **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 — except the card is reached **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*, the key store holds the actual **KIC/KID** for that card, and the engine never logs or returns the keys — 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 is Osmocom **pySim** (`pySim.ota`), the same code validated against physical sysmoISIM cards.

```
Command Packet  =  CPL | CHL | SPI | KIc | KID | TAR(3) | CIPHERED
CIPHERED        =  3DES-CBC( CNTR(5) | PCNTR(1) | CC(8) |
APDUs+padding )   IV=0
CC              =  last8( 3DES-CBC( KID,
zeropad(CPL+hdr+CNTR+PCNTR+APDUs) ) )
```

- **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 cryptographic checksum (CC), 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.

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.

Profiles, contexts, and auto-routing

An **OTA profile** is a reusable security preset — it holds **no key material**:

Field	Meaning
`tar`	3-byte Toolkit Application Reference (the target service).
`kic_index` / `kid_index`	Which keyset (1/2/3) → `KIC{n}`/`KID{n}` from the card's key-store row.
`cipher_algo` / `cc_algo`	Ciphering and checksum algorithm (`triple_des_cbc2`, `aes_cbc`, …).
`ciphering`, `rc_cc_ds`, `counter_mode`	SPI: encrypt? CC vs RC vs none? replay-counter policy.
`por`, `por_ciphered`, `por_rc_cc_ds`	POR required? ciphered? signed?

Two profiles are seeded, matching the values validated on sysmoISIM:

- **RFM-USIM-3DES-keyset3** — TAR `B00011`, keyset 3, 3DES, ciphering + CC, POR required + ciphered + 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 — and the right profile, keyset and SELECT are applied without switching 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
`local_pcsc`	a card in a PC/SC reader **on the central host**	synchronous	the directly-attached validation path.
`agent_reader`	a card in a reader **at a remote agent**	synchronous	rides the same agent WebSocket as Test Devices / SIM Bank (`session_apdu` relay).
`smpp`	a **field SIM** over SMPP `submit_sm`	asynchronous	MT SMS to an SMSC; the POR returns later as an MO SMS.

The **Explorer** uses only the in-shell transports (it needs a synchronous POR); **Campaigns** can use any, including SMPP.

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 pySim's 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** (via pySim's `decode_hex`).
4. Edit the **fields** (re-encoded with pySim) 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 can't blast thousands of cards in one request.

Jobs

`src/pages/ota/OtaJobs.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 pySim 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 (inspect the ciphertext + SMS framings).

Method	Route	Purpose
POST	`/decode` · `/encode`	Decode EF bytes ↔ typed fields via pySim (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 — **not** duplicated here.

Engine & dependencies

The crypto, packet structure, CAT/SMS TLVs and the offline file model are reused from **Osmocom pySim** (`pySim.ota` / `pySim.cat` / `pySim.sms` /

`pySim.global_platform` / the `ts_*` filesystem models). The backend modules are:

- `ota_engine.py` — build/decode secured packets; RFM/RAM C-APDU builders; SMS framing.
- `ota_transport.py` — the `local_pcsc` / `agent_reader` / `smpp` adapters.
- `ota_files.py` — the offline FID/path → file index (catalog + `decode_ef` / `encode_ef`).
- `ota.py` — models, RBAC, routes, audit; registered in the `android`-role process in `backend/app.py` (it shares the agent WebSocket for the in-shell transport).

pySim + `smpplib` + `pyscard` must be in the backend venv; see `backend/requirements.txt` (pySim is installed editable from the local checkout that carries the project's fixes).

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.
- SCP80 only — this is **not** SCP81 (no PSK-TLS / HTTPS Admin Agent).

Remote SIM & Virtual SIM

OmniWeb can present **any SIM to any modem over the network**, and can even present a SIM that **does not physically exist** — emulated 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 walking a plastic SIM to a card slot.

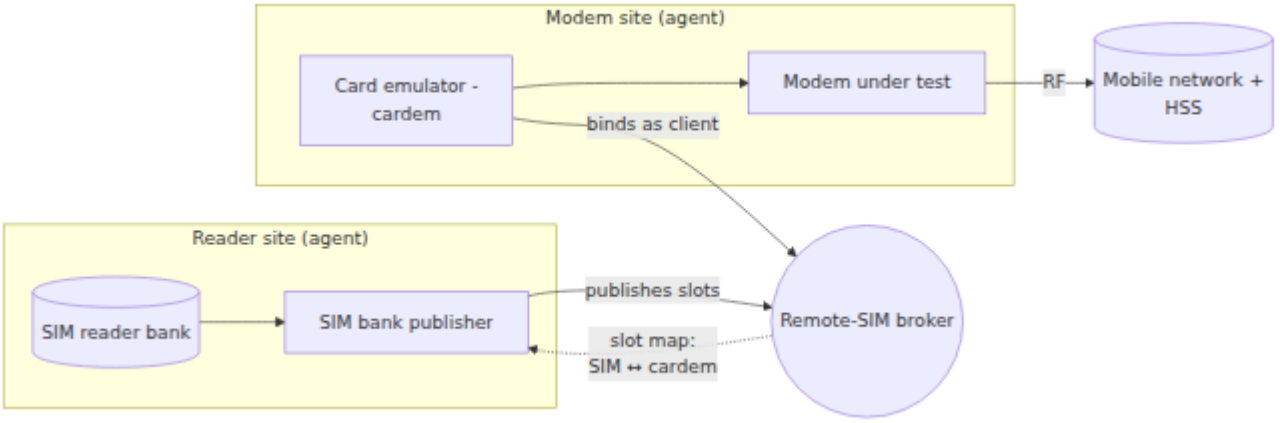
This page is the operations guide for that capability. The **inventory** half — discovering which SIMs are in which readers — is covered in SIM Bank; this page covers **presenting** a SIM (real or virtual) to a modem and using it in a test. Saving a card's file system as a portable profile package — and using one to back a virtual SIM — is covered in SIM Images (SAIP Profile Packages).

Table of Contents

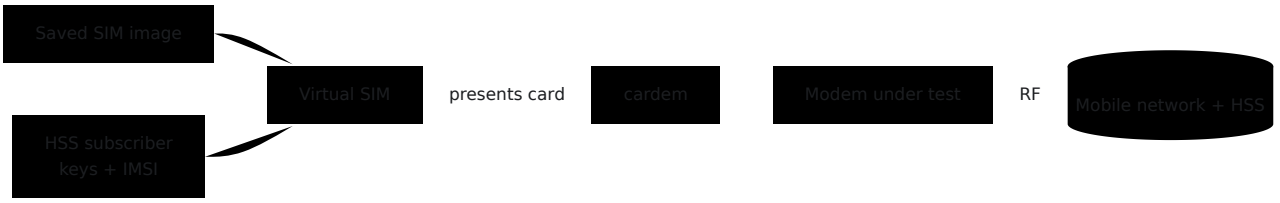
Architecture Overview

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, the reader bank is replaced by a software card built 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 + a subscriber's keys, presented to a cardem exactly like a real SIM — no physical card required.

The SIM Bank — what's where

The **SIM Bank** page lists every reader slot on every connected agent, grouped by agent, so 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** and 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	`ACTIVE` 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**, a card emulator is shown as its own device type (`ngff-cardem`), alongside modems and phones — it is never opened as a phone. Unknown device types are labelled plainly rather than guessed.

Opening the cardem gives 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 — the way you "swap the SIM" without touching hardware.
Remote SIM	Connect or disconnect which remote SIM is presented to this cardem.

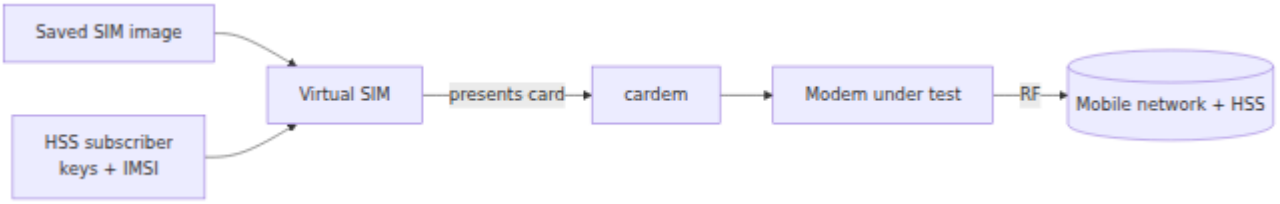
The **SIM source** panel states honestly what is and isn't available: the cardem feeds the modem a *remote* SIM. Presenting a *local* card seated on the emulator board is not offered here yet, and the console says so rather than showing a control that does nothing.

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**. It is built 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 that answers the modem's file reads from the image and answers the network's authentication challenge using the HSS keys — overlaying the chosen subscriber's IMSI. Presented 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 sourcing their physical SIM.
- Exercise 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 — reading that subscriber's IMSI and completing the network's authentication entirely from the virtual card.

> A dedicated **Virtual SIM Reader** element — where you "load" a virtual SIM from a chosen image and subscriber and present it like any other reader slot — is the next step on this capability. Today the virtual card is wired 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 testing; **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 instead of silently running against the wrong card.

See Test Devices and the test-book documentation for the rest of the IR.38 flow.

Troubleshooting

A SIM doesn't appear in the bank

Symptoms: a reader is plugged in but no slot (or an empty slot) shows.

Possible causes & resolution:

- The card isn't seated, or the reader/card pairing can't power the card — the slot shows **Card present** or **Empty**. Re-seat the SIM.
- The reader was just plugged in — readers are re-scanned on hot-plug, but you can force it with **Rescan** on the agent's group.

The modem won't attach after connecting a SIM

Possible causes & resolution:

- The modem hasn't re-read the SIM. Use **Reset modem** on the cardem console to re-present the card.
- The mapping isn't **ACTIVE** — confirm under **Active mappings** on the Remote SIM page that the SIM is bound to the right cardem.
- The subscriber isn't provisioned (or is barred) in the serving network's HSS.

"Card present (unreadable)" on a reader

Symptoms: a card is detected but its identity can't 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 doesn't 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. Re-capture the base image from a known-good SIM if init stalls.

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 — 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 for capturing, exporting, uploading, browsing and diffing SIM images, and for using a saved image to back a virtual SIM. The live reader inventory is covered in SIM Bank.

Table of Contents

Why SAIP UPP

Earlier builds stored a SIM image as a decoded JSON tree used only for diffing. That format was OmniWeb-private and could not be consumed by any external

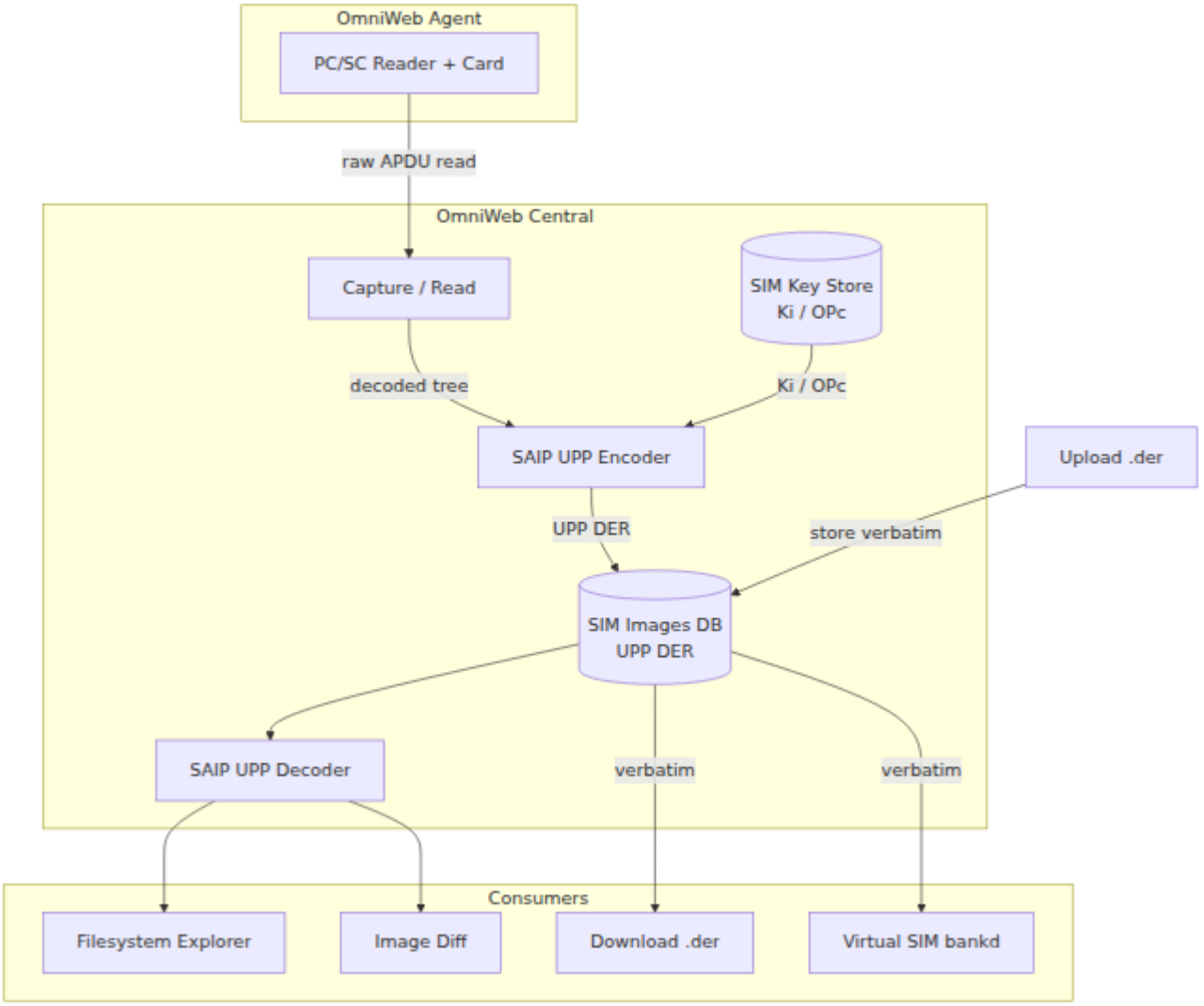
tool.

Storing the image as a **SAIP UPP** instead means:

- **Portability** — the exported `.der` is a standard profile package. It loads in pySim's `saip-tool`, 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-AKAParameter` element carrying Ki and OPc, so the package 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 (folding in Ki/OPc from the SIM Key Store if available), and the package is stored. 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's 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-AKAParameter`	Ki and OPc for Milenage — **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-AKAParameter` 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. To inspect it with pySim's `saip-tool`:

```
saip-tool.py UPP_<iccid>.der dump all_pe
```

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:

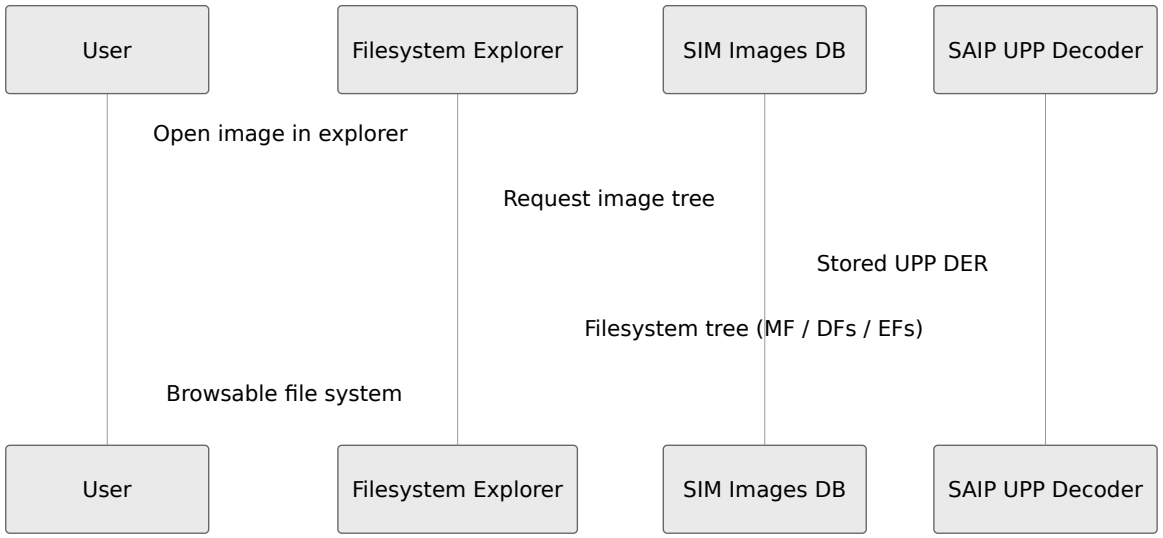
1. Validates the file begins with a SAIP Profile Element tag (a context-class constructed tag — `A0`, `A1` or `B0`).
2. Stores the package verbatim.
3. Decodes the `PE-Header` to populate the ICCID column where present.

An uploaded image is a first-class image: it can be downloaded, browsed in the explorer, diffed, and used to back a virtual SIM exactly like a captured one.

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. The package is decoded 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

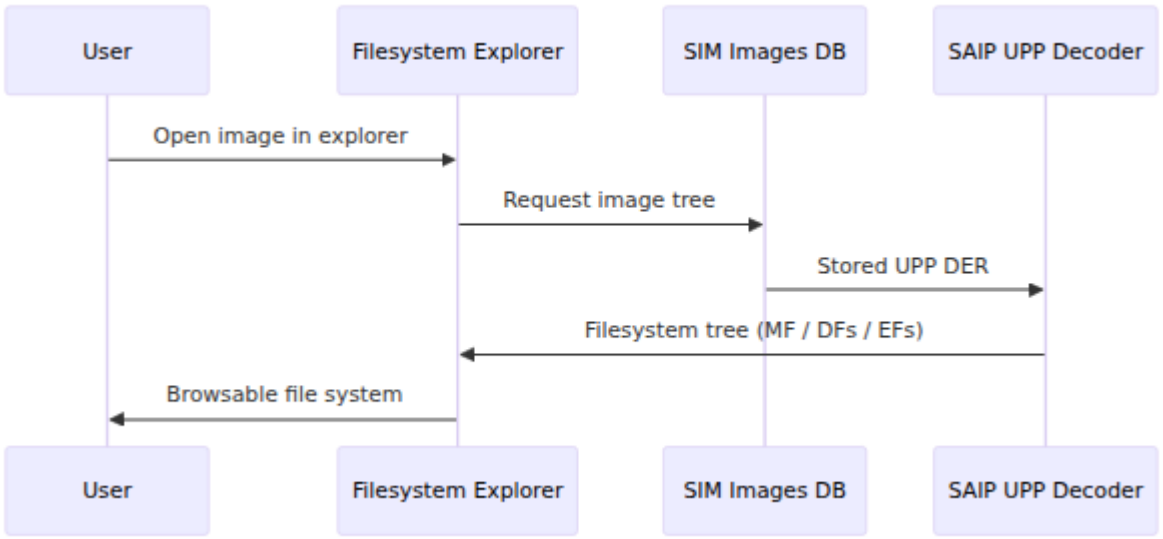
The saved-images page lets you pick two images and diff them. Both packages are decoded to file-system trees and aligned by EF path, so the comparison is file-by-file regardless of capture order:

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 — e.g. 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-AKAParameter`	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 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 osmo-remsim card-emulation/banking layer that lets you **connect any SIM to any cardem over the network** is documented separately in Remote SIM (osmo-remsim); this page covers discovery and inventory. Saving, exporting and reusing a card's file system as a portable profile package is covered in SIM Images (SAIP Profile Packages).

The page is **reader-centric**: one row per physical reader slot, grouped by agent, so 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 lives in `agent/omniweb_agent/simbank.py`. It mirrors the shape of `modem.py` (`list_modems`): enumeration is slow, so results are cached and rescanned on an interval.

For each PC/SC reader on the host:

1. **Enumerate** readers with pyscard (`smartcard.System.readers()`).
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. an osmo-remsim `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 our **in-tree `simcard` module** (`agent/omniweb_agent/simcard.py`), a self-contained set of raw pyscard APDUs (no out-of-tree pySim checkout):
   - 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.

The pySim dependency

The SIM read reuses **Osmocom pySim** (<https://github.com/osmocom/pysim>), which is **GPL**. We keep it as a **separate dependency** in the agent venv — we do **not** copy pySim code into our tree. `simbank.py` only contains our own orchestration plus the plain MCC→country table (data, not code) copied from `sim_record.py`.

Install into the agent venv (`agent/venv`, created with `--system-site-packages`):

```
# pyscard talks to pcscd (apt: pcscd libpcsclite1)
./venv/bin/pip install pyscard

# pySim is NOT on PyPI under this name — install from the Osmocom
source.
# (The PyPI package literally named "pySim" is an unrelated old
project.)
./venv/bin/pip install --no-deps /path/to/pysim     # e.g. a local
clone
```

`--no-deps` is deliberate: the agent venv already has the libraries pySim's card-read path needs (pyscard, pyserial, pycrate via the DIAG deps), and pulling pySim's full dependency tree (twisted/smpp/asn1tools) is unnecessary for reading cards. A host **without** pyscard or pySim still runs the agent fine and

simply reports no SIMs — the SIM bank is an optional per-host capability, like modems.

System packages required on the host: `pcscd` (running) and `libpcsclite1`. The reference reader is an SCM/Identiv SCR35xx (`04e6:581c`). Multiple readers per host are supported.

Inventory → central → UI flow

```
agent host                          central (android-role process)
browser
──────────                          ──────────────────────────────
────────
simbank.list_sims()  ──sims──►     android agent_socket
  (PC/SC + pySim)                    └─
registry.update_sims(agent_id, …)   GET /api/simbank/sims
                                     └─ simbank.record_sims(…)  ─►
DB       GET /api/simbank/readers

POST /api/simbank/agents/<id>/rescan
                       ◄─simbank_rescan──  (forces an uncached
scan)
```

- 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 blueprint is registered **only in the `android` role process** (same constraint as Test Devices), wired in `backend/app.py`.

REST API (`/api/simbank`, JWT, RBAC element type `simbank`)

Method	Route	Purpose
GET	`/sims`	Every SIM across all agents (incl. offline persisted-only), with live present/in-use overlaid.
GET	`/readers`	Every PC/SC reader on every **connected** agent (live), incl. empty and in-use readers.
GET	`/sims/<iccid>`	One SIM.
PATCH	`/sims/<iccid>`	Edit operator-assigned fields (notes today). Creates the row if the ICCID is unseen.
DELETE	`/sims/<iccid>`	Forget a SIM (drops the persisted record; reappears on next scan if still inserted).
POST	`/agents/<id>/rescan`	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
`iccid` (PK)	SIM serial.
`imsi`, `mcc`, `mnc`, `country`, `spn`, `atr`	Last-known identity (only overwritten by non-empty reads).
`notes`	Operator-assigned, editable.
`last_agent_id`, `last_site`, `last_reader_name`, `last_reader_index`	Where it was last (or is currently) seen.
`first_seen`, `last_seen`	Timestamps (UTC).
`remsim_bank_id`, `remsim_slot_nr`, `remsim_mapped_cardem`	**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: osmo-remsim

Phase 2 maps a physical reader/SIM into an osmo-remsim 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's 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.

Software Management & Lifecycle

OmniWeb provides a single place to see and manage the software running across the fleet — package versions, available updates, and licensing — so 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).

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

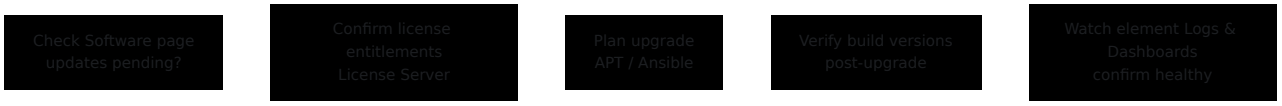
The **License Server** is administered through OmniWeb like any other element, using the same generic patterns (see Common Operations). It surfaces:

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 — useful for auditing 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's 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.

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 — VoLTE registration, signal, connectivity — whose pass/fail results are exported to Prometheus for alerting and dashboards.

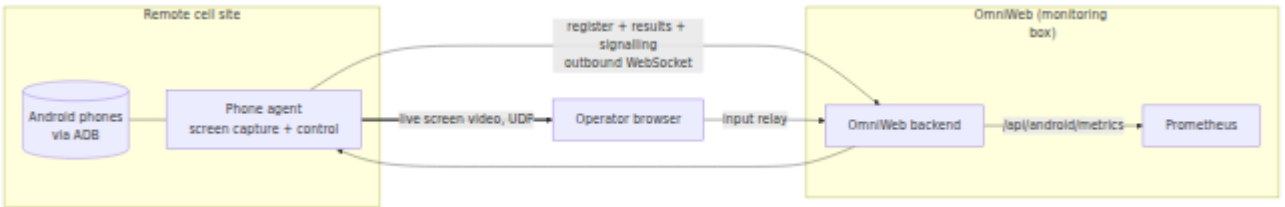
← Back to Operations Guide

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?"* — registered for VoLTE, able to call, text, and reach the internet — without anyone driving 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; input and test results are relayed through central.

> **Network paths matter.** The web UI, input and call setup ("signalling") ride the control plane over HTTPS to the OmniWeb server, but 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 will hang 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 `device_tester` Ansible role.
Central OmniWeb	Device registry, WebRTC signalling relay, script dispatch, results history, Prometheus metrics.
Prometheus	Scrapes scheduled-test results from `/api/android/metrics`.

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 exposing 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

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.

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: 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 the device must be forwarded:

Platform	How to forward the device
Proxmox — VM	`qm set <vmid> -usbN host=<bus>-<port>` to bind a physical port (the mapping survives swapping phones on that port), or `host=<vendor>:<product>` to follow one specific device. Find the ids with `lsusb` on the Proxmox host.
Proxmox — LXC container	Add to `/etc/pve/lxc/<ctid>.conf`: a device-cgroup allow for USB (`lxc.cgroup2.devices.allow: c 189:* rwm`) and a bind-mount of the USB bus (`lxc.mount.entry: /dev/bus/usb/<bus> dev/bus/usb/<bus> none bind,optional,create=dir`), 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 → <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's 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 doesn't, 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's 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 — Cellular validation, Scripts, Shell, Logcat, Files and Record — is alongside.*

> **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's you on another machine or a different operator — with a **Take over** button to claim 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 ±, Power buttons
Quick actions	One-click Airplane toggle, or open the Dialer, Messages or Settings app. These fire in the background — they don't 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 **View second 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. Add more the same way, up to **four screens** at once.

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 — which is named in the panel header so it's 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'd 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've 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 — filterable by source, status and test — without leaving the console. 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's 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, and stopping keeps what you've 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 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. AT commands are relayed agent → modem over its serial (AT) port; every command and response is logged 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 — `screencap` itself is the bottleneck on this hardware) and injects `adb input` for control. You get a click-to-tap, type-to-text view — not smooth video, but 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`).

Programmatic control (MCP)

The same device-control stack is exposed as an **MCP server**, so an MCP client (Claude, scripts) can drive the farm: `list_devices`, `screenshot`, `tap` / `swipe` / `input_text` / `key`, `adb_shell`, `list_scripts` / `run_script`, and modem `at_command` / `send_sms`. It wraps the `/api/android/*` API and is bearer-token gated. Setup and the `claude mcp add …` command are in **mcp/README.md**.

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** — no-argument health checks, run with just the device. These appear in the library, the fan-out runner, and the scheduler.

Examples: Device Connected, VoLTE Registration, Report Signal Strength, Toggle Airplane Mode.

- **Built-in tools** — the parameterised UI tools (Call, Text, Ping, Speedtest) driven from the Cellular Validation panel. They take inputs and are not part of 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 — and 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
`name`	String	Yes	-	Unique label for the schedule. Appears as the `schedule` label on the exported metric.
`script`	String	Yes	-	Filename of a folder script in the scripts directory to run.
`interval_seconds`	Integer	No	300	How often to run the script, in seconds. Minimum 30.
`serials`	List	No	`[]`	Device serials to target. An **empty list means all currently-connected devices**, so the schedule auto-follows whatever phones are plugged in.
`enabled`	Boolean	No	true	Whether the schedule is active. Set `false` to pause it without removing the entry.

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 can't.

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

Companion gauges share the same labels:

Metric	Description
`omniweb_android_test_return_code`	Exit code of the last run
`omniweb_android_test_duration_ms`	Duration of the last run, in milliseconds
`omniweb_android_test_last_run_timestamp_seconds`	Unix time of the last run

Example queries:

```
# Phones currently failing VoLTE registration
omniweb_android_test_pass{script="VoLTE Registration Status"} == 0

# Failing tests grouped by site
count by (site) (omniweb_android_test_pass == 0)

# A test that hasn't run recently (stale agent or device offline)
time() - omniweb_android_test_last_run_timestamp_seconds > 600
```

Troubleshooting

A phone doesn't appear in the fleet

Symptoms: A device is plugged in at a site but isn't listed under Test Devices.

Possible causes:

- The phone isn't authorised for ADB on the agent host
- The agent isn't running or can't reach central
- The phone is in an offline/unauthorised ADB state
- The central **`omniweb-android`** service isn't running (Test Devices runs as its own single-worker service on port 5002; the multi-worker pool does not serve `/api/android/*`)

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 `systemctl status omniweb-android` is active and that nginx routes `/api/android/*` to it (port 5002). 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 — by far the most common cause)
- The agent can't 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 won't 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 can't 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

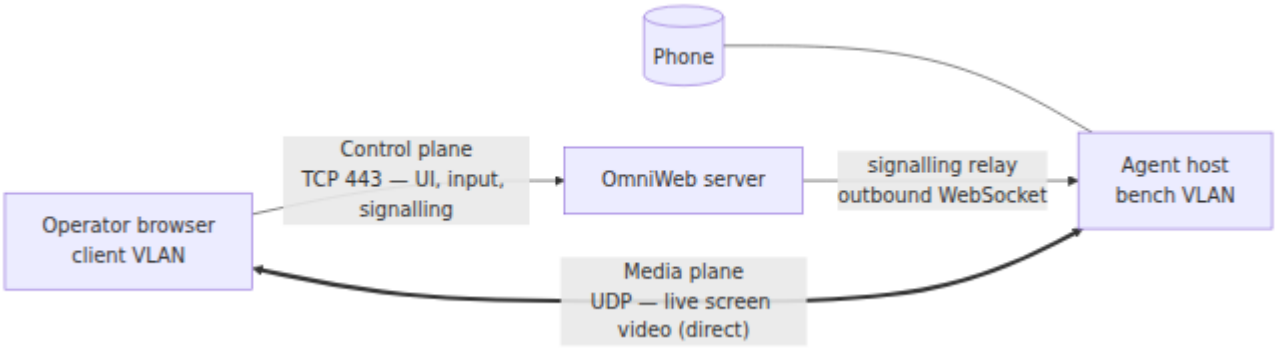
Live phone viewing in **Test Devices** uses **two separate network paths**. Understanding the split is the key to getting it working across firewalls and VLANs — and to diagnosing the single most common symptom, a console that sits on **"Connecting…"** forever.

← Back to Test Devices

The two paths

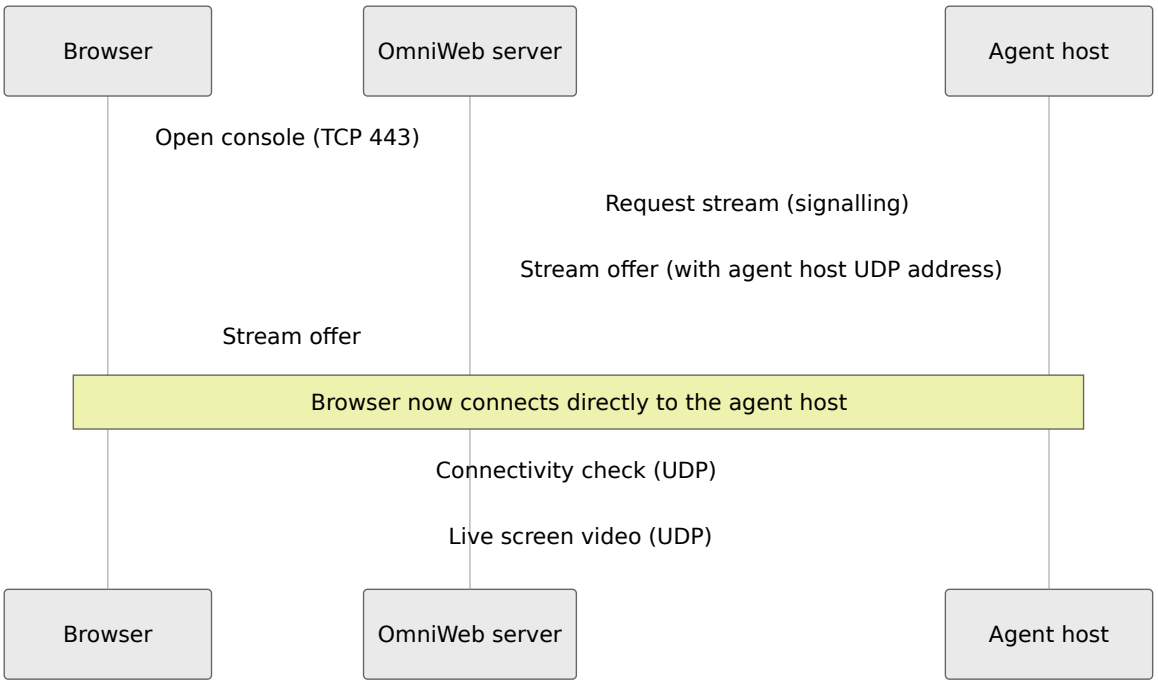
Path	Carries	Direction	Transport	Endpoint
Control plane	The web UI, login, the device list, remote input, test results, and the call setup ("signalling") for a live view	Browser → OmniWeb server	**TCP 443 (HTTPS)**	The OmniWeb server
Media plane	The live screen video itself	Browser ⇄ **agent host** (the 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. A network that allows the browser to reach the *server* but not the *agent host* will load the UI, show the device list, and then hang 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's 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. The agent can instead be configured to draw its media ports from a small fixed range, shrinking 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.

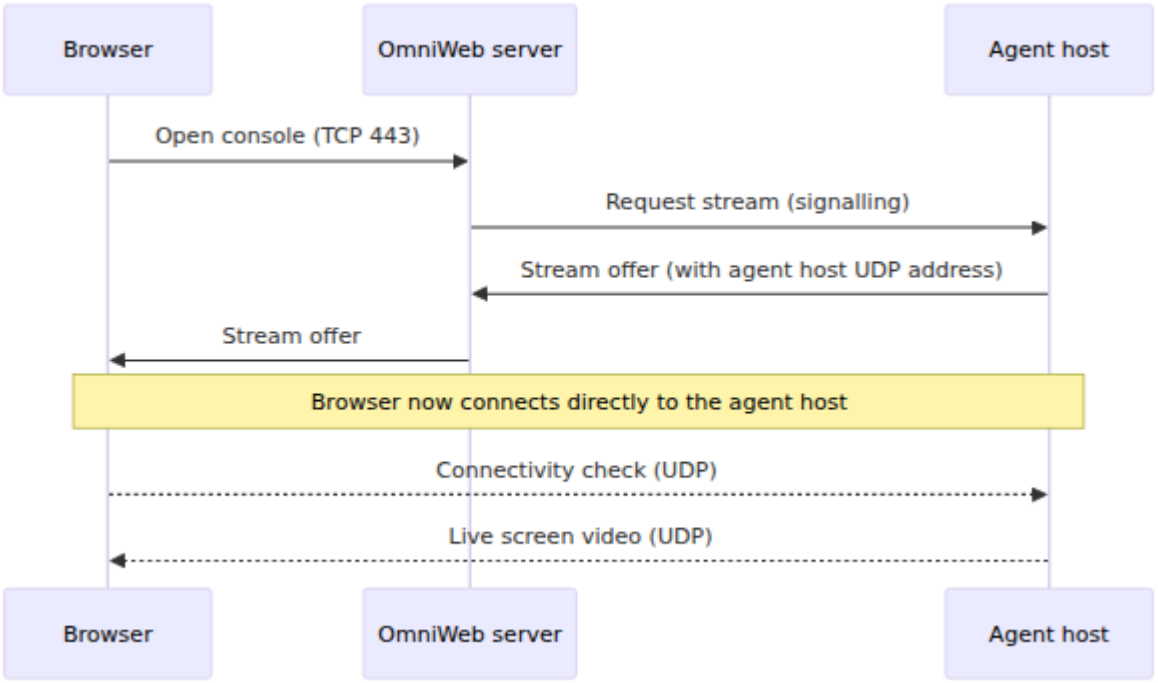
Recognising the problem

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's not this	What you'd actually see if it were
SSL / certificate	The UI, login, device list and signalling all succeed over HTTPS — 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 won't 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 with whoever maintains the benches that the firewall rule's **destination subnet and ports** match the agent hosts and their configured media-port range, and that the rule is applied 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

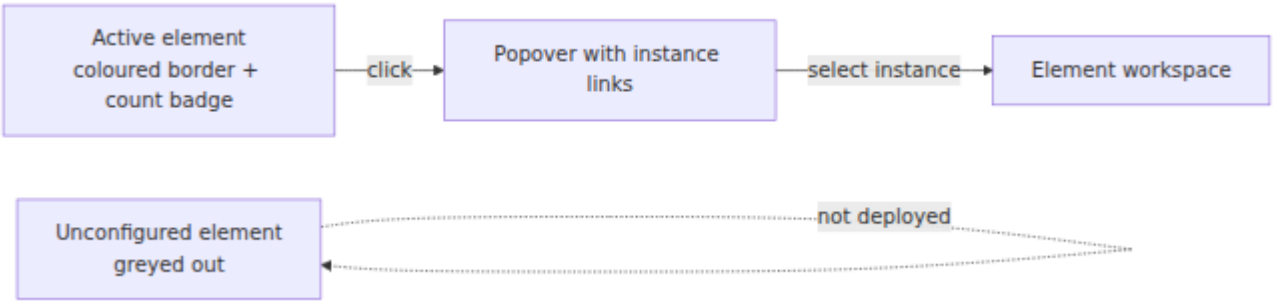
Topology Views

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** — those with instances declared in `omniweb.json` — appear with a coloured border and a badge showing how many instances exist. Clicking one opens a popover with direct links to each instance, so you can 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 isn't 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's 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 — 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, transport — without hunting for bookmarks.