SERIAL CABLES



PCIE GEN6 · NVME · CXL · OUT-OF-BAND

Hydra Command Center

User's Manual

VERSION 1.0

© Serial Cables, LLC · Englewood, CO · serialcables.com

CONTENTS

Table of Contents

Hydra Command Center — Introduction 3
1. Getting Started & Connection 8
2. Dashboard 10
3. Slot Control 13
4. Environment 20
5. NVMe-MI 23
6. Diagnostics 29
7. Quarch: Power Analysis 37
8. Quarch: Hot-Plug Lab 44
9. Quarch: Hot Swap 53
10. Quarch: Signal Breaking 54
11. Automation Guide 55
12. Vendor Defined Messages (VDM) 61
13. Slot Device Firmware Updates 64
14. Event Log 68
15. Run Manifest 70
16. CXL OOB Management (Technology Preview) 71
17. Host Agent (in-band) 80
18. Workloads 89

Hydra Command Center — Introduction

Welcome to the **Hydra Command Center**, the control application for the **Serial Cables PCIe Gen6 HYDRA 8-Bay JBOF enclosure**.

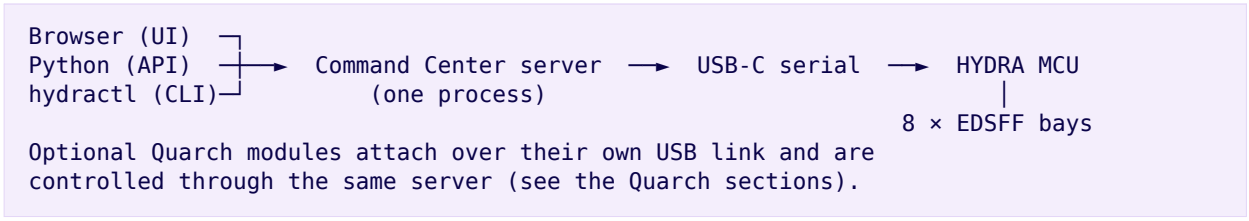
The Command Center gives you three ways to work with the same enclosure, all served by a single program:

Surface	What it is	When to use it
Web UI	An interactive dashboard in your browser	Day-to-day operation, exploration, live monitoring
REST API	An HTTP interface with interactive docs	Automation from Python, LabVIEW, Jenkins, or any language
CLI (`hydractl`)	A command-line tool	Shell scripts, CI pipelines, quick one-liners

Every function in the UI has a matching API endpoint and CLI command. Anything you learn to do by clicking, you can later script.

How it connects

The HYDRA enclosure is managed through **one USB Type-C serial interface** to its on-board MCU (115200 baud, 8N1). The Command Center server owns that serial connection and arbitrates it, so the UI, API, and CLI can all be used at the same time without conflicting.



Installation

Requirements. Python 3.10 or newer, and HYDRA MCU firmware **0.9.0 or newer**. HYDRA units shipped with firmware 0.0.8 must be updated before the Command Center's out-of-band features will work; the firmware image and its SHA-256 checksum are on the HYDRA product page. To check your version, look at the Dashboard **System** card (*Version*) or run `hydractl system version`.

Everything is on the **HYDRA product page on the Serial Cables website**. The Command Center is not published on PyPI, so always install the file you downloaded, never `serialcables-hcc` by name.

Linux (recommended): one command

Download the **Linux install kit** (`serialcables-hcc-<version>-linux.zip`), unzip it on the machine cabled to the enclosure's USB management port, and run:

```
sudo sh install-hcc.sh
```

Add `--with-agent` if this same machine is the host whose PCIe links reach the drives: the *host agent* is then installed, enrolled and started too. The installer needs internet access for the Python dependencies, and supports Ubuntu 22.04+/Debian 12+ and RHEL/Rocky/Alma 9+ with systemd. It:

- installs Python 3.10+ and its venv support if missing, and puts the Command Center (with Quarch support) in `/opt/serialcables/hcc`;

- runs it as the `hcc` systemd service under its own `hcc` user, which can open the enclosure's serial port and Quarch USB modules;
- picks the enclosure's serial port automatically: the only `/dev/serial/by-id/` device, or among several the one that is the HYDRA's controller (USB ID `0000:0002`, named `RENESAS CDC USB Demonstration`). With two enclosures attached, choose with `--serial-port`, or connect from the *Connection* page;
- serves the network with **TLS and an API token**: it creates a self-signed certificate and a token in `/etc/hcc`, and opens the port in `ufw`/`firewalld` if one is active;
- sets up `hydractl` to reach this server with no extra configuration (members of the `hcc` group; the user who ran `sudo` is added).

It finishes by printing the link to open, for example `https://192.168.1.20:8600/?token=…`. Accept the browser's self-signed certificate warning once. Commands you will use later:

Command	Does
`sudo hcc-setup status`	Show the browser link, token and certificate fingerprint again
`systemctl status hcc` / `journalctl -u hcc -f`	Service state / live log
`sudo sh install-hcc.sh` (newer kit)	Upgrade; the token, certificate, settings and data are kept
`sudo hcc-setup --serial-port /dev/serial/by-id/…`	Change a setting (also `--port`, `--local`, `--renew-cert`)
`sudo sh install-hcc.sh --uninstall [--purge]`	Remove (`--purge` also deletes settings and data)

`--local` keeps the server on this machine only (127.0.0.1, plain HTTP, no token needed). Settings live in `/etc/hcc/hcc.env` if you prefer to edit them (any `HCC_*` setting from *Starting the server*); restart with `sudo systemctl restart hcc`. Re-running the installer keeps your edits. An upgrade stops the service while it installs, so don't run one during a workload, capture or firmware update.

The installer never takes over things it didn't create: an existing `hcc` account, a `hydractl` or `hcc-setup` already on the PATH, or your own Quarch udev rule are left alone (an existing account or `hcc.service` stops the install with a message saying what to move aside).

The self-signed certificate is a server certificate only (it can't sign others). For a certificate from your own CA, set `HCC_TLS_CERT` and `HCC_TLS_KEY` in `/etc/hcc/hcc.env` and restart; host agents then enroll with `--tls-ca` instead of `--pin`.

Windows 10/11 (x64): one double-click

Download the **Windows install kit** (`serialcables-hcc-<version>-windows.zip`), unzip it on the PC cabled to the enclosure's USB management port, and double-click **`install-hcc.cmd`**. It asks for administrator rights once (UAC), then asks whether the drives are connected to this PC: if so, it installs the *host agent* too. It keeps its window open at the end so you can read the result. With internet access, it:

- installs Python 3.12 from python.org (checking the installer's signature) if the PC has no Python 3.10 or newer, and puts the Command Center (with Quarch support) in `C:\Program Files\SerialCables\HCC`;
- runs it in the background as the scheduled task `\SerialCables\HYDRA Command Center`, which starts with Windows (as SYSTEM) and, every 5 minutes, starts it again if it has stopped;
- keeps settings, the token, a self-signed TLS certificate, data and the log (`logs\hcc.log`) in `C:\ProgramData\SerialCables\HCC`, readable only by SYSTEM and Administrators (and, for `client.env` and the certificate, the user who installed it);
- picks the enclosure's serial port automatically: the only USB serial port, or among several the HYDRA's controller (USB ID `0000:0002`); Bluetooth COM ports are ignored;
- adds a firewall rule for the port on **Domain and Private** networks. A PC on a network Windows calls *Public* stays closed, so a double-click offers to change it to Private, and a run with options prints the command;

- puts `hydractl` and `hcc-setup` on the PATH (open a new terminal).

It ends with the same browser link as on Linux. From a prompt, the kit takes the same options as `hcc-setup` (`install-hcc.cmd --serial-port COM5`). Later, from an **Administrator** prompt: `hcc-setup status`, `hcc-setup restart`, and `install-hcc.cmd --uninstall [--purge]`. Quarch modules additionally need Quarch's USB driver (installed with Quarch Power Studio). From a prompt, add `--with-agent` if this PC is also the host whose PCIe links reach the drives: it installs the *host agent* here too and enrolls it with this server. An upgrade also upgrades an installed agent. If the agent needs fio, install it in `C:\Program Files\fio`, then run the kit again.

Things to know on Windows:

- The server runs as **SYSTEM** (on Linux it runs as its own unprivileged user). It is protected by the token and TLS, but treat the PC like any machine that exposes hardware control on the network.
- `COM` numbers can change if the enclosure is plugged into a different USB port. If HCC stops finding it, re-run `install-hcc.cmd --serial-port COMn` (Device Manager > Ports shows the new number), or connect from the Connection page.
- The installer refuses an existing `C:\ProgramData\SerialCables\HCC` that was not created by an administrator: a folder planted there by another account could pre-set the token or certificate.
- If Group Policy forces a PowerShell execution policy, the unsigned `install-hcc.ps1` may be blocked; ask IT to allow it or run the steps in *Running in the foreground* instead.

Running in the foreground (any OS)

Install the wheel into Python 3.10+ and start the server yourself (see *Starting the server* below):

```
pip install ./serialcables_hcc-<version>-py3-none-any.whl

# Optional: enable the Quarch module features (Power Analysis and the
# Hot-Plug Lab) if you purchased Quarch hardware:
pip install "quarchpy>=2.2"
```

To upgrade, download the newer wheel from the same page and install it the same way.

License

The Command Center is free of charge but proprietary — it is not open-source software. Use is governed by the `LICENSE` file installed with the package (`pip show -f serialcables-hcc` lists its location): you may use it within your organization and script against its API and CLI freely, but not redistribute or modify it. Third-party components are listed in `THIRD_PARTY_NOTICES.md`.

Starting the server

```
# With real hardware (replace COM3 with your port; /dev/ttyUSB0 on Linux):
hydractl serve --serial-port COM3

# No hardware yet? Explore everything with the built-in simulator:
hydractl serve --demo
```

Then open **http://127.0.0.1:8600** in your browser.

Navigation and the Active Bay

The left navigation is grouped into **Monitor**, **Control**, **Protocol Lab**, **Quarch**, **Workloads**, **Host**, and **System**. If the optional QuarchPy library is not installed (and you are not in demo mode), the Quarch group shows a locked heading — hover over it to see what to install. The header carries a global **Active Bay** selector: pick a bay once — or click it on the enclosure front panel — and every bay-scoped page (NVMe-MI, VDM, CXL CCI, Firmware, the Diagnostics I2C tools, and Bay Detail) targets that bay. **Alt +**

1…8 sets the active bay from the keyboard, and **Alt + H** shows the keyboard-shortcut overlay. Clicking a bay on the Dashboard opens its **Bay Detail** view with that DUT's controls, health, telemetry, and recent protocol activity in one place.

A **light/dark theme** toggle and a **compact density** toggle sit in the header; the theme defaults to dark for bench use.

Useful options (`hcc-server` takes the same ones):

- `--serial-port COM3` (`-p`) — the HYDRA serial port.
- `--host 0.0.0.0` — allow other machines on your LAN to reach the UI/API. **On any non-local bind, token authentication is enforced.** If you don't set a token, the server generates one for that run and prints it at startup, together with a browser URL of the form `http://<host>:8600/?token=<token>` (the browser keeps the token in a cookie after that). Scripts send `Authorization: Bearer <token>`. For a token that stays the same across restarts, use `--token` or `HCC_API_TOKEN`.
- `--tls-cert` / `--tls-key` (PEM files) — serve HTTPS. Recommended on any LAN bind; without them the token crosses the network in plaintext.
- `--port 8700` — change the HTTP port (default 8600).
- `--no-auto-connect` — start the server without opening the serial port.

Environment variables can be used instead of flags: `HCC_SERIAL_PORT`, `HCC_PORT`, `HCC_HOST`, `HCC_DEMO`, `HCC_API_TOKEN`, `HCC_TLS_CERT`, `HCC_TLS_KEY`, and `HCC_DATA_DIR` (where logs and records are stored; default `hcc_data`, relative to the directory the server is started from; the Linux install kit's service uses `/var/lib/hcc`).

Feature maturity in this release

Out-of-band NVMe-MI drive management, enclosure control, the in-band Host Agent, and Workloads are fully supported; Workload presets not yet run end to end on hardware carry a **BENCH** marker (see §18). **CXL OOB management and I3C discovery are Technology Previews** (§16, §6). Slot-device firmware update ships with an on-screen caution (see §13). The VDM Template Studio is a demonstration feature, and the SAS page (§3) is experimental.

Manual sections

1. *Getting Started & Connection*
2. *Dashboard*
3. *Slot Control*
4. *Environment*
5. *NVMe-MI*
6. *Diagnostics*
7. *Quarch: Power Analysis*
8. *Quarch: Hot-Plug Lab* — signal routing, hot-swap timing and cycling, signal breaking, and the hardware glitch engine
9. *Quarch: Hot Swap* — now part of the Hot-Plug Lab
10. *Quarch: Signal Breaking* — now part of the Hot-Plug Lab
11. *Automation Guide*
12. *Vendor Defined Messages*
13. *Slot Device Firmware Updates*
14. *Event Log*
15. *Run Manifest*
16. *CXL OOB Management (Technology Preview)*
17. *Host Agent (in-band)*
18. *Workloads*

Each section describes the UI first, then ends with **API Reference** and **CLI Reference** addendums covering the same functions.

1. Getting Started & Connection

The **Connection** page attaches the Command Center to your HYDRA enclosure. You will usually visit it once per bench setup.

Before you begin

1. Make sure the enclosure runs HYDRA MCU firmware **0.9.0 or newer** (see *Introduction → Installation*).
2. Connect the HYDRA's **USB Type-C management port** to your computer.
3. Windows shows the enclosure as a COM port (e.g. `COM3`); Linux as `/dev/ttyUSB0` or similar. No driver is normally required.
4. Start the server: `hydractl serve` (add `--demo` to practice without hardware).

Using the Connection page

Open the UI and choose **Connection** in the left navigation.

1. **Rescan Ports** — lists every serial port on the host with its description. Your HYDRA is typically described as a USB serial device.
2. Pick your port from the **Serial port** dropdown.
3. Leave **Demo mode (simulator)** off for real hardware. Switch it on to use the built-in simulator instead — every page then runs against a realistic fake enclosure, which is perfect for learning this manual.
4. Click **Connect**. The status line turns green (● `Connected — COM3`, or ● `Connected — simulator` in demo) and the header chip across the whole UI changes to **CONNECTED · COM3** (or **DEMO MODE**).
5. **Disconnect** releases the serial port (for example, to use a terminal program directly).

The **Connection Settings** card shows the fixed link parameters (115200 baud, 8N1, no flow control), the serial timeout, the telemetry poll interval, and the API base URL for scripting.

> **Tip:** If the server was started with `--serial-port COM3` (or `HCC_SERIAL_PORT=COM3`), it connects automatically at startup — the Connection page will already show green.

Troubleshooting

- **Port missing from the list** — check the USB cable and click *Rescan Ports*. On Linux, confirm your user is in the `dialout` group.
- **Connect fails** — another program (PuTTY, another Command Center) may hold the port. Only one process can own it; close the other program.
- **Connected but pages show no data** — verify the enclosure is powered and the correct port was chosen; try Disconnect → Connect.
- **Red "Not connected" banner on every page** — click **Reconnect** in the banner, or **Connection settings** to come to this page.

API Reference — Connection

Base URL: `http://<host>:8600/api/v1`

On a server bound to a non-local address, add the header `Authorization: Bearer <token>` (see *Introduction → Starting the server*).

All responses use the envelope `{"success": true, "data": ...}` or `{"success": false, "error": "..."}`.

Method	Path	Body	Purpose
GET	`/connection/status`	—	Current connection state (`connected`, `demo`, `port`, `baudrate`)
GET	`/connection/ports`	—	Enumerate host serial ports
POST	`/connection/connect`	`{"port": "COM3", "demo": false}` (both optional)	Open (or re-open) the MCU link. Omitted fields keep their previous value, so pass `"demo": false` to leave demo mode
POST	`/connection/disconnect`	—	Release the serial port
GET	`/health`	—	Server liveness + connection summary

```
import httpx
api = httpx.Client(base_url="http://127.0.0.1:8600/api/v1")
ports = api.get("/connection/ports").json()["data"]
api.post("/connection/connect", json={"port": ports[0]["device"], "demo": False})
```

Errors: attempting any enclosure operation while disconnected returns HTTP **409** with `"Not connected to a HYDRA enclosure"`.

CLI Reference — Connection

```
hydractl ports                      # list serial ports
hydractl connect --port COM3 --hardware   # real hardware (--hardware leaves demo mode)
hydractl connect --demo             # connect to the simulator
hydractl disconnect                 # release the port
hydractl status                     # server + connection status
```

Global options go **before** the command name, e.g. `hydractl --json slot list`:

- `--url http://otherhost:8600` (or `HCC_URL` env var) — target a remote server.
- `--json` — machine-readable output for scripting.

When the server is token-protected (any non-local bind), set `HCC_API_TOKEN` to its token and `hydractl` sends it automatically.

2. Dashboard

The **Dashboard** is your at-a-glance view of the whole enclosure. It is the first page you see and the one you will keep open on a bench monitor.

What you see

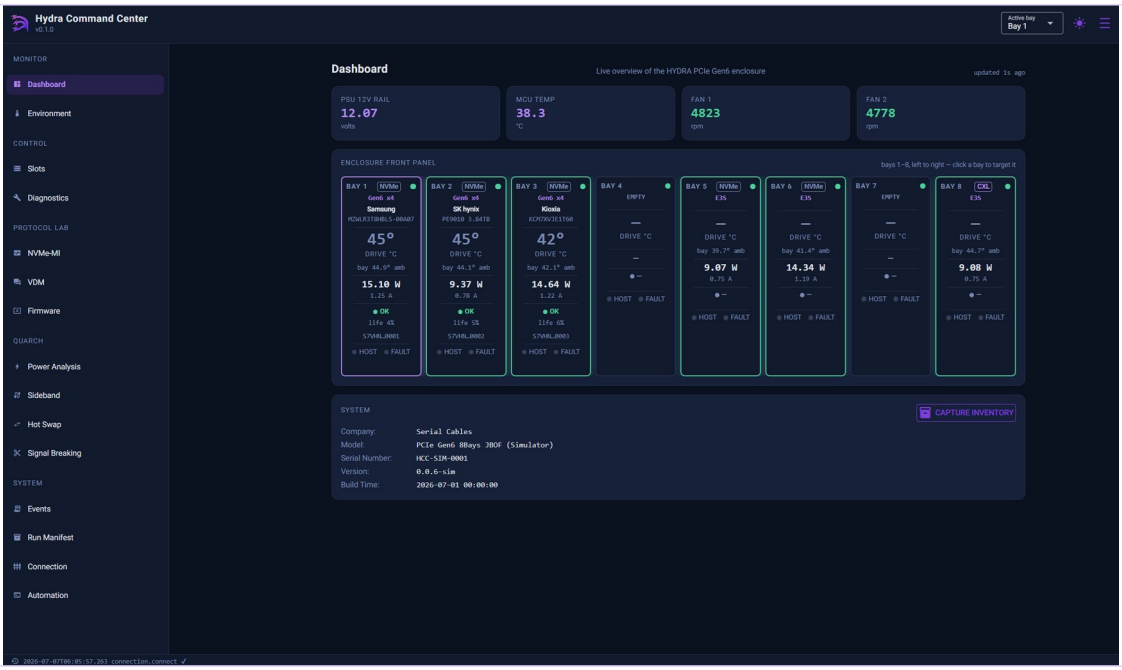


Figure 2.1 Dashboard. Per-bay temperature, power, and health tiles update live across all eight bays.

Headline tiles

Four live tiles across the top, refreshed every couple of seconds. A small indicator at the top right reads **live**, or **updated Ns ago** (in amber when polling has stalled), so a frozen value never looks current.

- **PSU 12V RAIL** — the main supply voltage. Healthy systems read close to 12.0 V under load.
- **MCU TEMP** — the management controller's own temperature.
- **FAN 1 / FAN 2** — measured fan speeds in RPM.

Enclosure Front Panel

A live picture of the eight bays, laid out like the physical enclosure:

- **Green border with a glow** — drive present and powered. **Grey border** — present but unpowered.
- **"EMPTY"** — no drive detected in the bay. Drives behind a U.3 interposer show as e.g. **SAS · interposed** instead (see §3).
- **Red border** — the bay's fault LED has been commanded on.
- **Accent-colored border** — the current Active Bay.
- Each occupied bay shows a class badge (**NVMe**, **CXL**, **SAS**), the negotiated link (e.g. *Gen6 ×4*) or EDSFF type, the drive's own temperature, the bay ambient temperature, power draw (W and A), health, and HOST/FAULT LED indicators. §3 describes each field.

Click any bay to open its **Bay Detail** view — a single-DUT page gathering that bay's power/reset controls, live drive health, a temperature and power sparkline, and its recent VDM exchanges. Clicking a bay also sets it as the global Active Bay for the protocol pages.

System card

Identity information read from the enclosure: company, model, serial number, firmware **Version**, and **Build Time**. Quote these when contacting support.

Capture Inventory

The **Capture Inventory** button (top-right of the System card) sweeps the whole enclosure — firmware/serial plus every populated bay's drive serial number and health summary — into one timestamped **run manifest**. Capture this at the start of every test campaign and save the JSON with your results; it is the reproducibility record ("what exactly was installed when I ran this?"). The sweep queries each populated drive over NVMe-MI, so allow a few seconds on a full enclosure. When it finishes, the **Run Manifest** page opens with the result (see *Run Manifest*).

Reading it like an operator

- All bays green and temperatures in the 30–50 °C range: normal.
- A bay that shows power but 0 W draw usually means the drive is present but held in reset or PWRDIS — check the Slot Control page.
- Fans reading 0 RPM with the system powered is a fault; check the Environment page and the physical fans.

API Reference — Dashboard data

The dashboard is a composition of endpoints you can call yourself:

Method	Path	Purpose
GET	`/system/info`	Full identity + per-slot summary (presence, temp, volts, amps, watts)
GET	`/system/version`	Identity fields only (fast)
GET	`/slots`	Per-bay live view used by the front panel
GET	`/environment/telemetry/latest`	Most recent telemetry sample (tiles)
GET	`/inventory`	One-shot run manifest (enclosure + all drive SNs + health). ? `include_health=false` to skip health queries

```
info = api.get("/system/info").json()["data"]
print(info["model"], info["firmware_version"])
for slot in info["slots"]:
    if slot["present"]:
        print(f"Bay {slot['slot_number']}: {slot['temperature']} degC")
```

CLI Reference — Dashboard data

```
hydractl system info          # identity + slot summary table
hydractl system version       # identity only
hydractl slot list            # the front-panel view as a table
hydractl env show             # the sensor sweep behind the tiles
hydractl inventory            # run manifest as a table
hydractl --json inventory --no-health > manifest.json
```

Put `--json` **before** the command for scripting:

```
hydractl --json system info | python -c "import json,sys; d=json.load(sys.stdin);
print(d['firmware_version'])"
```

3. Slot Control

The **Slots** page is laid out like a physical control panel: **rocker toggle switches** for on/off state and **push-buttons** for momentary reset commands, so it reads and operates like the physical hardware it drives. It controls each of the eight drive bays: payload power, identification LEDs, resets, and the dual-port / PWRDIS signal lines.

Page layout, top to bottom:

1. **Bulk actions** (all bays at once).
2. **Enclosure front panel** — the eight bays drawn as tall SSD-shaped slots, left to right 1–8, mirroring the physical chassis. Each bay shows a live **device readout** (see below); **click** a bay to target it, **double-click** to open its full Bay Detail page.
3. **Per-bay controls** for the selected (Active) bay.

Every toggle and button has a **mouse-over tooltip** describing its function.

Interposed bays (U.2/U.3 via interposer)

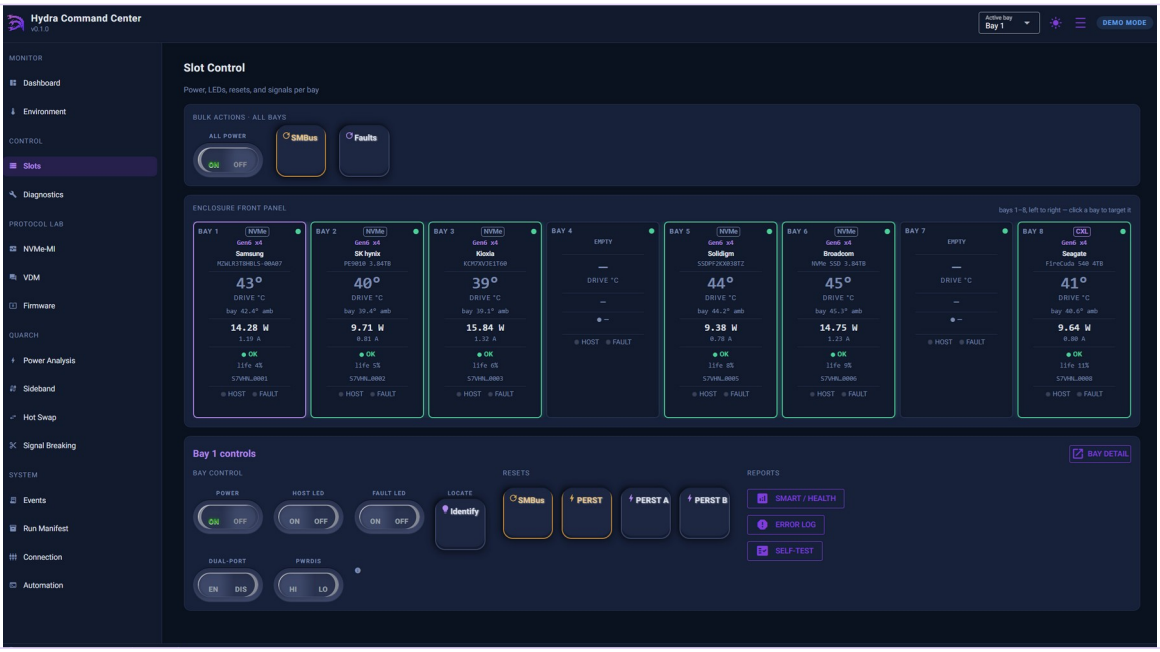


Figure 3.1 Slot Control. Rocker and momentary controls per bay, with live power, LED and PWRDIS readback.

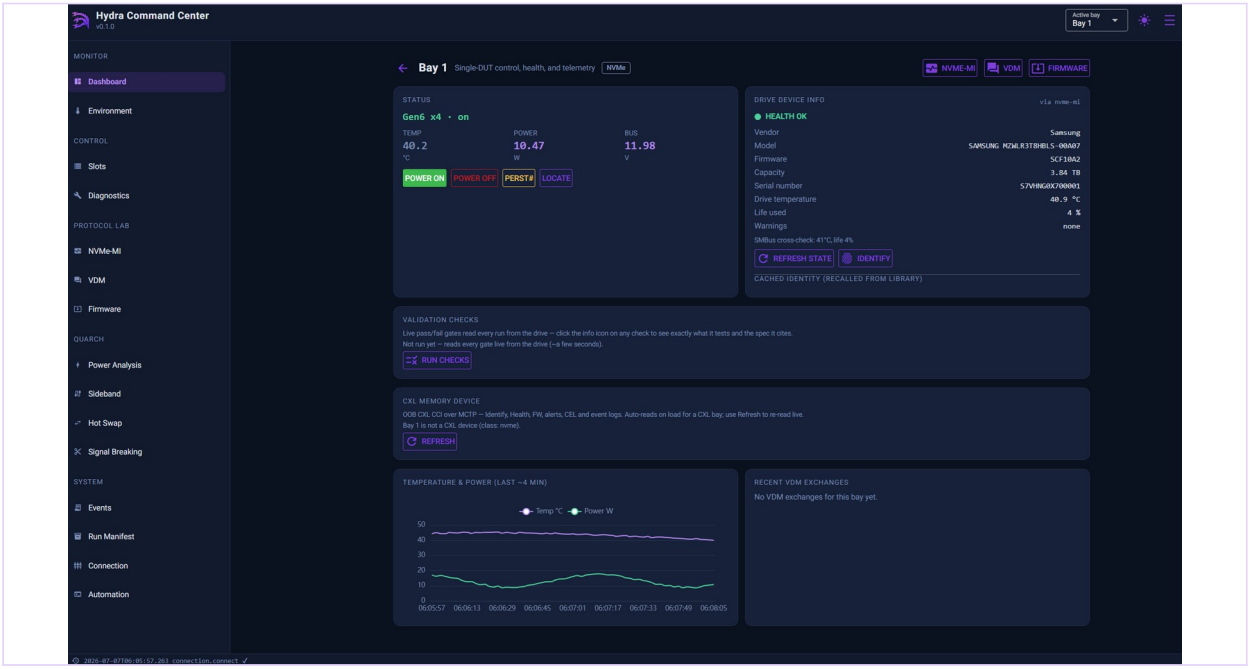


Figure 3.2 Bay Detail. Per-bay identity, link, and device-library match for a single drive.

A drive mounted through a **U.3 interposer** does not assert bay-level presence, but HCC detects it automatically: the interposer's on-board IO expander is probed on every slot refresh, and an interposed drive shows on the front-panel tile as `**SAS · interposed**` (or the detected type) with a class badge and its real power draw, instead of EMPTY. Drive type comes from the connector's interface-detect pins: the **SAS** pattern is verified on hardware; other patterns are reported with `drive_type_verified: false` until proven on hardware.

The probe is also available directly — `hydractl slot interposer <bay>` or `GET /slots/{bay}/interposer` — returning expander presence, drive presence at the drive connector, drive type, and the interposer's board-ID (e.g. U3X4, the single-port x4 lane-order variant; a 2×2 dual-port variant identifies differently). Power/current telemetry and PWRDIS control work normally on interposed bays regardless of type.

The SAS page

> **Experimental.** The SAS page has not been validated across the range of SAS drives and is provided as-is in this release.

When an interposed **SAS drive** is detected, a **SAS** entry appears in the sidebar (Control group) — it disappears when no SAS drive is present (in `--demo`, bay 7 simulates an interposed SAS drive). The page shows one card per SAS bay: the interposer identity (board variant and raw expander registers), a live stat row (power, current, bay-ambient temperature, PWRDIS state, presence), confirm-gated **PWRDIS** control, and a **Sideband scan** that probes the well-known I2C addresses so a drive that terminates the sideband (e.g. PMBus on some vendors' parts) is found immediately. SAS SMART-equivalents (Log Sense pages) are **in-band only** — attach the drive to a SAS HBA for those; the page states this explicitly.

Per-slot device readout

Each bay tile shows both enclosure and **drive-reported** information, gathered in the background so it never slows the interactive controls:

Field	Meaning
Vendor / Model	Drive vendor (Samsung, SK hynix, Kioxia, Micron, Solidigm, Broadcom, ...) and model number, from **Identify Controller**.

Field	Meaning
DRIVE °C (large)	The drive's own composite temperature, from NVMe-MI. Thermal-colored (amber ≥70 °C, red ≥80 °C).
bay … amb	The enclosure bay's ambient thermal sensor — distinct from the drive's internal temperature.
W / A	Live power and current for the bay.
health dot + OK/WARNING/CRITICAL	Derived from the drive's SMART critical-warning flags and spare-vs-threshold.
life %	Percentage of rated endurance used.
spare %	Available spare capacity.
serial	Drive serial number (truncated; hover for the full value).

A background **round-robin poller** refreshes one bay's device data per tick, so all eight refresh over roughly 15–25 seconds without saturating the single MCU serial link. Static identity (the serial number) is read once when a drive is detected and cached; dynamic health is refreshed each cycle. Values appear within a cycle of a drive being inserted or powered on.

Data sources, for resilience:

- **NVMe-MI over MCTP** (primary) — health/temp/spare on any enterprise NVMe-MI drive.
- **Identify Controller**, tunneled over the NVMe Admin Command Message (NMIMT=2h) — vendor, model, firmware revision, and capacity (queried once per drive and cached, since identity never changes). *(A drive that doesn't support the tunnel simply shows no identity and still reports health. Disable with `HCC_DEVICE_IDENTIFY=0`.)*
- **NVMe Basic Management over SMBus** (raw I2C, address 0x6A) — a universal cross-check and fallback for temperature / life / SMART warnings on drives with limited MCTP support. *(A secondary source. Values obtained this way are labelled via smbus** on the Bay Detail page, which also shows the SMBus cross-check alongside the NVMe-MI values. Disable the probe with `HCC_DEVICE_SMBUS_PROBE=0`.)*
- **CXL Get Health Info + Identify Memory Device** — a bay detected as a CXL Type-3 device fills the same tile fields from its own SMART-equivalent: device temperature, life used, health flags (media / hardware-replacement / maintenance), FW revision, and capacity. The health dot maps media-not-normal or hardware-replacement to CRITICAL.
- **SAS drives** (via U.3 interposer) show what the sideband physically offers — power, presence, drive type. SCSI's SMART-equivalents (Log Sense temperature / percentage-used / error counters) are **in-band only**; per- drive telemetry appears only if the drive terminates the sideband (vendor- dependent, e.g. PMBus).

The full per-drive detail — vendor/model/firmware/capacity, the SMBus cross-check, and the warning list — is on each bay's **Bay Detail** page (click a bay on the Dashboard), and the identity is captured into the *Run Manifest*.

The device library (identity & traceability)

Every drive identity the poller and inventory sweeps observe is folded into a durable, **serial-keyed device library** — a single JSON index persisted under the data directory (`HCC_DATA_DIR`). It gives you cross-session traceability for FA / RMA without re-paying the Identify tunnel on every test.

> **Identity only — never health.** The library stores **static identity** (vendor, model, firmware-at-capture, capacity) keyed to the drive's serial. Dynamic values — health, temperature, link, counters — are **never** cached; you always read those live. The rule is an allowlist, so a new dynamic field can't accidentally leak into the record.

Population is deliberately conservative:

- A drive whose serial reads but whose Identify has not yet succeeded still gets a **weak, serial-only** record, so the library is useful even for a drive that does not answer the admin tunnel; a later successful Identify upgrades it to **strong** and backfills the details. Confidence never downgrades.
- A **serial seen live in two bays at once** (a cloned or duplicate serial) is flagged **`conflict`** — never silently merged, and never served as one drive's identity for another. Bogus serials (all-zero, all-F, blank) are refused so they can't collapse many drives into one record.
- Bay moves and firmware changes are recorded as observations; a departed drive's bay binding is cleared, and bindings are invalidated on an enclosure power-cycle / PERST.

Cached identity on the Bay Detail page

Each **Bay Detail** page has a **CACHED IDENTITY (recalled from library)** panel, separate from the live state block, driven by an **Identify** button. Pressing it **resolves** the bay: it **re-reads the drive's serial live**, then serves the identity cached against *that just-read serial*. Because the lookup keys on the freshly-read serial (not a stale binding), a **swapped drive can never resolve as the previous one** — and if the same serial is found live in another bay, the panel shows an **IDENTITY CONFLICT** callout instead of a trustworthy result. The panel labels each result **NEW** or **KNOWN** and clears itself when it detects the bay changed.

Selecting a bay

The live front panel sits at the top of the page. **Click a bay** to select it — the selected bay is highlighted with an accent ring, the controls card switches to it ("Bay 3 controls"), and it becomes the global Active Bay used by the protocol pages too. **Alt + 1…8** selects from the keyboard. The controls card re-syncs its switches to the newly selected bay immediately (and refreshes them every 2 seconds) so the toggles never misrepresent which bay they target.

Per-bay controls

The **Bay Control** cluster is a bank of rocker toggle switches; **Resets** are momentary push-buttons. A **Bay Detail** button on the control card opens the full single-DUT detail view (`/bay/N`) for the selected bay — the same page reachable by clicking a bay on the Dashboard, or by **double-clicking a bay tile** here on the Slots page (a single click only selects, since the tiles double as the target selector for the controls below).

The card's third cluster, **REPORTS**, opens on-demand reads for the selected bay: **SMART / Health** (see *NVMe-MI*), **Error Log**, and **Self-Test** (see *Diagnostics*).

Toggles (Bay Control)

- **Power** (ON/OFF) — switches the bay's payload power. This toggle tracks the drive's **live** power state. Powering a bay off is equivalent to pulling the drive's power, so treat it like a surprise removal unless the host has been prepared.
- **Host LED** (ON/OFF) — the drive's host-controlled indicator. On real hardware the enclosure lights it on **every linked bay** as a presence lamp, so don't rely on it for locating a specific drive.
- **Fault LED** (ON/OFF) — the amber fault indicator. The front panel turns that bay's border red while the fault LED is commanded on. Also used by **Identify (Locate)**, precisely because the host LED is already lit on every populated bay.
- **Dual-Port** (EN/DIS) — drives the dual-port enable line. Enable when using dual-port EDSFF drives with two hosts. Shows the enclosure's **live readback**.
- **PWRDIS** (HI/LO) — the PWRDIS pin. Shows the enclosure's **live readback**. HIGH disables the drive's power at the drive level even though the bay has power — useful for emulating power-fail conditions on a single drive.

> **Live vs. last-commanded:** the **Power**, **Dual-Port**, and **PWRDIS** toggles show the enclosure's **live readback** — the same values as the `dual_port` / `pwrdis` fields of `GET /slots` and the Dual / PWRDIS columns of `hydractl slot list`. If the enclosure has not reported a readback for the bay, Dual-Port and PWRDIS appear **dimmed (unknown)**; they never fall back to showing the value you

last commanded. The MCU cannot read back LED state, so the **Host LED** and **Fault LED** toggles show the value **last commanded** from the Command Center, and appear dimmed (unknown) until you command them this session. The ⓘ icon next to the toggles summarises this. After an enclosure reset the hardware reverts to firmware defaults.

Push-buttons (Resets & Locate)

- **Identify** (Locate) — **blinks** the bay's **fault LED for 60 seconds**, then turns it off automatically, so a technician can physically find the drive in a rack. The fault LED is used (not the host LED) because the hardware keeps the host LED lit on every linked bay as a presence lamp. The fault LED indicator on that bay's front-panel tile blinks while identify is active.
- **SMBus** — pulses reset on the bay's SMBus/management interface only. Use when NVMe-MI communication with a drive stops responding; the data path is unaffected.
- **PERST** — pulses PCIe reset to the drive (both ports).
- **PERST A / PERST B** — reset a single port of a dual-port drive.

Recovering a bay's management

If a bay stops answering, its status reads unknown and its LEDs may flash or stay on, use **Recover** rather than **SMBus**. A drive can hold the bay's bus with a Hot-Join or IBI request, and an SMBus reset on its own can start that hold again. Recover checks the bay first and changes nothing if it is healthy. Otherwise it releases the bus, resets, and re-checks, up to four times, then tells you what it found and whether management came back.

Recover never touches drive power or PERST#, so a live PCIe link stays up. On a bay behind a Quarch breaker card, choose the bay's module: the card masks the enclosure's reset, so the pulse has to come from the module. Releasing the bus needs an MCU firmware with the I3C test commands; on other firmware Recover uses the SMBus reset alone and says so. If management doesn't return, hot-plug the drive or power-cycle the bay.

Front-panel LED indicators

Each bay tile shows two simulated LED indicators mirroring the physical enclosure: **HOST** (green when lit) and **FAULT** (red when lit). They reflect the last-commanded LED state; the fault LED **blinks** during an Identify. **Both LEDs default to OFF after any power cycle** — the hardware clears them when payload power is removed, so they must be re-commanded after a power-off/on or an enclosure reset. The **buzzer** behaves the same way (defaults off; see *Environment*).

Bulk actions

The **Bulk Actions** card sits **above** the front panel and applies to all bays at once:

- **All Power** (toggle) — ON powers every bay; OFF powers every bay down (with a confirmation, since it interrupts any running test).
- **SMBus** (button) — SMBus-reset every bay (confirmed).
- **Faults** (button) — clear the fault LED on every bay.

API Reference — Slots

Method	Path	Body	Purpose
GET	`/slots`	—	Live merged view of all bays (includes a `device` block per bay)
GET	`/slots/power`	—	Power state per bay

Method	Path	Body	Purpose
GET	`/slots/devices`	—	Per-bay drive device info from the background poller (cached; no serial cost)
GET	`/slots/devices/{slot}`	—	Device info for one bay
GET	`/slots/{slot}/interposer`	—	U.3 interposer probe (expander, drive presence, drive type, board ID)
POST	`/slots/power`	`{"slot": 3, "state": "on"}` — for `"off"` add `"confirm": true`	Bay payload power (`slot` may be `"all"`)
POST	`/slots/host-led`	`{"slot": 3, "state": "on"}`	Host LED
POST	`/slots/fault-led`	`{"slot": 3, "state": "off"}`	Fault LED
POST	`/slots/identify`	`{"slot": 3, "seconds": 60}` (1–3600; one bay)	Blink the fault LED to locate the drive, then auto-off
POST	`/slots/smbus-reset`	`{"slot": 3}`	SMBus reset pulse
POST	`/slots/recover-management`	`{"slot": 3, "quarch_con_string": null, "attempts": 4}`	Bring back a bay whose management stopped answering; nothing is changed on a healthy bay
POST	`/slots/ssd-reset`	`{"slot": 3, "channel": "a", "confirm": true}`	PERST# pulse (`channel` optional: `"a"`/`"b"`); always confirm-gated
POST	`/slots/dual-port`	`{"slot": 3, "enabled": true}`	Dual-port enable line
POST	`/slots/pwrdis`	`{"slot": 3, "level": "high", "confirm": true}`	PWRDIS level (`"high"`/`"low"`); `"confirm"` is required for `"high"`
GET	`/devices/library`	—	The whole serial-keyed library: every device record + current bay→serial bindings
GET	`/devices/library/{serial}`	—	One device record by serial (404 if never seen)
GET	`/devices/resolve/{slot}`	—	Resolve a bay: live serial reconfirm, then cached identity (`no-store`)

Validation errors (bad slot, bad state, or a destructive call without `"confirm": true`) return HTTP **422**; a command the MCU did not acknowledge returns HTTP **502**.

`/devices/resolve/{slot}` returns `{"resolved": false, "reason": ...}` for an empty/unreadable bay, or `{"resolved": true, "serial_number", "model", "firmware", "identity", "identity_confidence", "origin", ...}` — plus `"conflict": true, "conflict_bay"` when the same serial is live in two bays. A **CXL** bay resolves through its bay-keyed CXL identity instead (see *chapter 16*): `serial_number` is `null`, `origin` is `cxl-identify`, and a device whose CXL mailbox doesn't answer returns `{"resolved": false, "reason": "cci_not_answering"}`.

```
# Power-cycle bay 3 with a settle delay
api.post("/slots/power", json={"slot": 3, "state": "off", "confirm": True})
time.sleep(2)
api.post("/slots/power", json={"slot": 3, "state": "on"})
```

CLI Reference — Slots

```
hydractl slot list                      # table of all bays
hydractl slot power 3 off -y            # bay 3 power off (prompts unless --confirm/-y)
hydractl slot power all on              # every bay on
hydractl slot hled 5 on                 # host LED on bay 5
hydractl slot fled 5 off                # fault LED off
hydractl slot identify 5                # blink the fault LED 60 s to locate bay 5
hydractl slot identify 5 --seconds 30   # custom duration
hydractl slot smbus-reset 3             # SMBus reset pulse
hydractl slot recover 3                 # bring back lost management
hydractl slot recover 3 --quarch USB:QTL3311-01-006   # bay behind a breaker card
hydractl slot ssd-reset 3               # PERST# both ports (prompts; -y to skip)
hydractl slot ssd-reset 3 --channel a   # PERST# port A only (prompts; -y to skip)
hydractl slot dualport 3 on             # dual-port enable
hydractl slot pwrdis 3 high             # assert PWRDIS (prompts; -y to skip)
hydractl slot interposer 7              # U.3 interposer probe

hydractl device library                 # list every device + current bay bindings
hydractl device resolve 3               # live-reconfirm bay 3's serial, then cached
identity
```

4. Environment

The **Environment** page is the enclosure's health station: live temperature and power charts, fan control, the audible alarm, and reference clock status.

Live charts

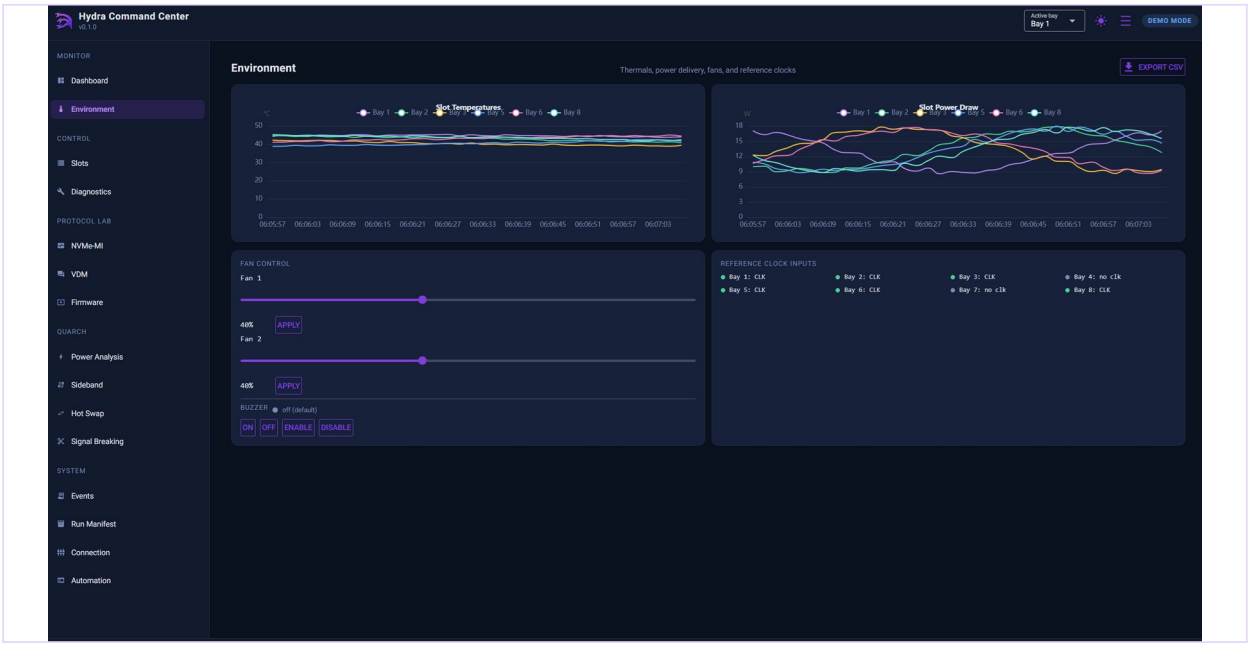


Figure 4.1 Environment & Telemetry. Rolling per-bay temperature and power charts with manual fan control.

Two rolling charts (about the last four minutes at the default 2-second poll interval):

- **Slot Temperatures (°C)** — one line per occupied bay.
- **Slot Power Draw (W)** — one line per occupied bay.

Empty or unpowered bays are omitted automatically. Hover for exact values; each bay keeps a consistent color across both charts.

The charts are a **live view** backed by the server's in-memory window (last ~30 minutes at the default poll interval). For report-grade records, every sample is also **persisted to disk** as NDJSON under the server's data directory (`HCC_DATA_DIR`, default `hcc_data` in the directory the server was started from; one file per day), and the in-memory window can be downloaded as CSV with the **Export CSV** button at the top right of the page (or via the API/CLI below).

Fan control

Each fan has a slider (0–100 % PWM duty, 5 % steps) and an **Apply** button. The slider's starting position is not read back from the hardware, and setting a duty below 20 % asks for confirmation.

- The enclosure ships with automatic thermal behavior; manual duty settings are for testing (e.g. acoustic tests, forced-cooling experiments).
- After setting a duty cycle, watch the Dashboard RPM tiles to confirm the fans responded.
- **Do not leave fans at low duty with hot drives.** Watch the temperature chart while experimenting.

Buzzer

Four buttons plus a state indicator (grey = off, amber = sounding). The indicator reads *off (default)* until you command the buzzer this session:

- **On / Off** — sound or silence the buzzer immediately.
- **Enable / Disable** — arm or disarm the buzzer function itself (a disabled buzzer stays quiet even during alarm conditions).

The buzzer **defaults to OFF** and, like the slot LEDs, reverts to off after a system power-off or enclosure reset — re-command it if you want it sounding.

Reference Clock Inputs

A grid of the eight bays showing whether each detects a PCIe reference clock (green dot = clock present). A powered bay with a drive but **no clock** will never link-train — check the host cabling and the paddle card.

API Reference — Environment

Method	Path	Body / Params	Purpose
GET	`/environment`	—	One full sensor sweep (temperatures, voltages, currents, powers, fan speeds)
POST	`/environment/fan`	`{"fan_id": 1, "duty_cycle": 60}`	Fan PWM duty (fan 1–2, duty 0–100)
POST	`/environment/buzzer`	`{"state": "off"}`	`"on"`, `"off"`, `"enable"`, `"disable"`
GET	`/environment/clocks`	—	Clock-present boolean per bay
GET	`/environment/telemetry`	`?limit=100`	Server's in-memory sample history
GET	`/environment/telemetry/latest`	—	Most recent sample
GET	`/environment/telemetry.csv`	`?limit=900`	History as CSV (one row per sample, one column per sensor)
WS	`ws://<host>:8600/ws/telemetry` (not under `/api/v1`)	—	Push stream: every new sample as JSON. On a token-protected server, send the Bearer header

Sensor sweep shape:

```
{
  "temperatures": {"slot_1": 43.5, "mcu": 38.2},
  "voltages":     {"psu_12v": 12.05, "slot_1": 12.01},
  "currents":     {"slot_1": 1.21},
  "powers":       {"slot_1": 14.53},
  "fan_speeds":   {"fan_1": 4800, "fan_2": 4750}
}
```

```
# Log bay temperatures to CSV every 5 s
import csv, time
with open("temps.csv", "w", newline="") as f:
    writer = csv.writer(f)
    while True:
```

```
        env = api.get("/environment").json()["data"]
        writer.writerow([time.time()] + [env["temperatures"].get(f"slot_{n}") for n in
range(1, 9)])
        f.flush()
        time.sleep(5)
```

CLI Reference — Environment

```
hydractl env show                # one sensor sweep as a table
hydractl env watch               # live refreshing table (Ctrl-C to stop)
hydractl env watch --interval 5  # slower refresh
hydractl env fan 1 60            # fan 1 to 60% duty
hydractl env buzzer off          # silence the buzzer
hydractl env clocks              # clock inputs per bay
hydractl --json env telemetry --limit 100 > history.json
hydractl env export -o soak.csv  # in-memory window as CSV for reports
```

For runs longer than the in-memory window, use the persistent NDJSON files the server writes continuously under its data directory — they survive restarts and cover the entire session. Files are named `telemetry-YYYYMMDD.ndjson`; set `HCC_TELEMETRY_LOG=0` to disable them.

5. NVMe-MI

> **NVMe-MI, VDM and CXL CCI share one Protocol Console.** All three follow the same loop — build an MCTP frame → send it to a bay → decode the response — so they live together at `**/console**`. The context bar at the top of the console has three protocol buttons, **NVMe-MI**, **VDM** and **CXL CCI**. The **NVMe-MI** and **VDM** nav entries still exist; each deep-links into the console with its protocol preselected (`/console?proto=nvme-mi` / `?proto=vdm`), and the old `/nvme-mi` / `/vdm` URLs redirect there. This section documents the NVMe-MI side; *section 12* covers VDM and *section 16* covers CXL CCI.

The **Protocol Console** talks to the drives themselves — not just the enclosure — using the NVMe Management Interface carried over MCTP through the HYDRA MCU. This works **out-of-band**: no host PCIe link or OS driver is involved.

The console at a glance

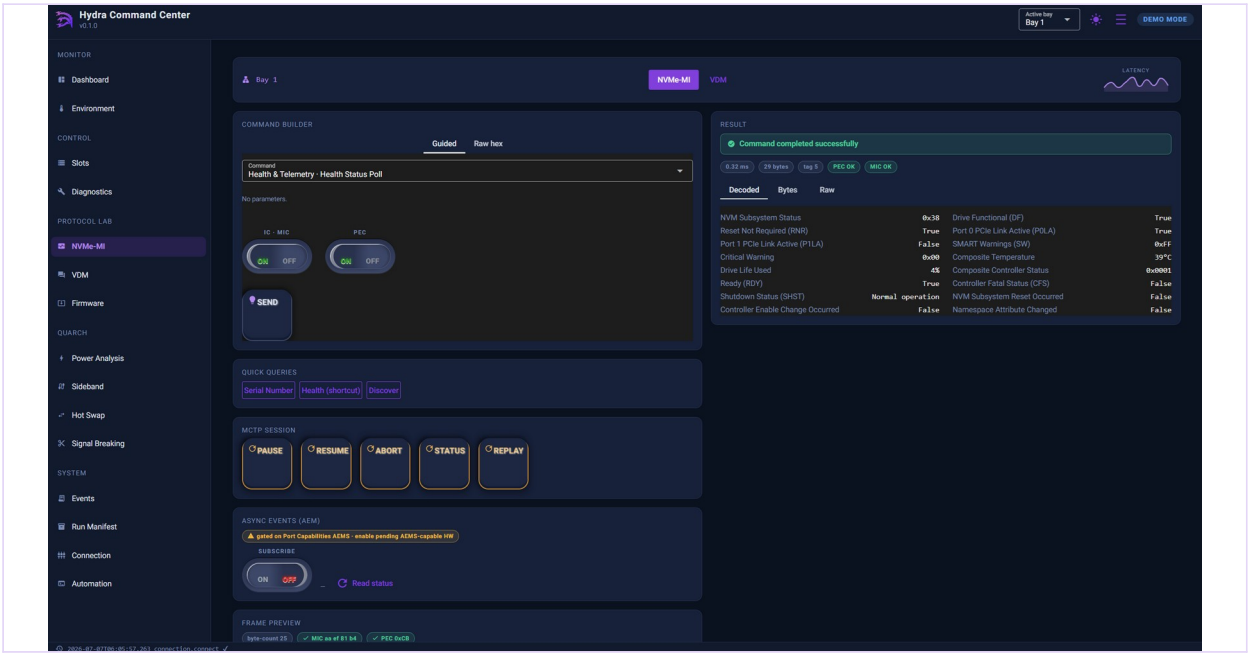


Figure 5.1 Protocol Console — NVMe-MI. Guided command builder and raw-hex expert tab with a decoded verdict banner.

- **Context bar** — the current target bay (with a warning if no drive is detected there), the three protocol buttons, and a **latency sparkline** that trends the response time of the active bay's recent exchanges.
- **Command Builder** — a *Guided* tab (pick a command from a searchable catalog, fill named typed fields, no hex typing) and a *Raw hex* expert tab (freeform bytes, parsed-as-you-type). **IC** and **PEC** rocker toggles set the frame's integrity fields; an **ADMIN TUNNEL** badge marks commands (Identify, SMART / Health Log) that carry an NVMe admin command inside NVMe-MI.
- **Frame Preview** — the exact bytes that will be sent, as an annotated, role-tinted byte grid with computed byte-count / MIC / PEC chips.
- **Result** — the **RESULT** panel: a plain-English **verdict banner** (GOOD/WARN/BAD from the transport checks + NVMe-MI status), a telemetry strip (latency, byte count, tag, PEC/MIC), and **Decoded / Bytes / Raw** tabs. The *Bytes* tab renders the response as the same annotated grid + ASCII, with **Copy hex**; the *Raw* tab shows the structured JSON, exactly as the API returns it.
- **Exchange History** — every NVMe-MI and VDM exchange of this session, below the builder and result (see *Exchange History*).

Choosing a target

Every query on this page targets the **Active Bay** shown in the header (set it there, with **Alt + 1...8**, or by clicking a bay on any enclosure panel). The page shows the current target and warns if that bay has no drive detected. The selection follows you across NVMe-MI, VDM, Firmware, and I2C — select the DUT once.

> **Not an NVMe drive?** A bay may hold a **CXL Type 3 memory device** that answers out-of-band on a different MCTP message type. *Section 16* covers device-class autodetect and the parallel CXL CCI read path.

Queries

Queries come in two places: the **Guided** catalog in the Command Builder, and the **Quick Queries** card.

Guided catalog (fully decoded)

Open **Command Builder → Guided** and pick a command. The catalog is grouped into **Health & Telemetry**, **Topology** and **Identity**. Each command is built on the host as a standards-compliant NVMe-MI request, sent as a raw MCTP frame, and every response field is decoded:

Group	Command	What it returns
Health & Telemetry	**Health Status Poll**	NVM Subsystem Health Status Poll, with each field named and explained.
Health & Telemetry	**SMART / Health Log**	The full SMART / Health Information Log (see below). Admin tunnel.
Topology	**Subsystem Info**	NVMe-MI version, number of ports.
Topology	**Controller List**	Controller IDs behind the endpoint.
Topology	**Port Info**	Port type, MTU sizes.
Identity	**VPD Read**	Vital Product Data.
Identity	**Identify Controller**	Vendor, model number, serial, firmware revision and capacity, read with the NVMe Identify Controller admin command carried over NVMe-MI. Admin tunnel.

Quick Queries

The **QUICK QUERIES** card holds three one-click shortcuts. They have no host-built frame to preview:

- **Serial Number** — the drive's serial number, parsed by the enclosure MCU.
- **Health (shortcut)** — quick health values parsed by the enclosure MCU: composite temperature, spare threshold, critical warning flags.
- **Discover** — runs a full discovery sweep of the bay in one click.

SMART / Health log (on-demand deep read)

The full **NVMe SMART / Health Information Log** (Log Page 02h) is available on demand — it is **not** part of the background poller. Besides the Guided catalog entry, the *Slots page* **Bay Control** panel has a **REPORTS → SMART / Health** button that opens a pop-out with the fully parsed log for the active bay, organized into Reliability, Temperature, Endurance & Usage, and Thermal Management sections, with a health summary at the top. The pop-out can **save the raw 512-byte log** (`.bin`) or the parsed data (`.json`) to a file.

Fields include composite + per-sensor temperatures, available spare & threshold, percentage used, **data written/read (TBW)**, host read/write commands, **power-on hours**, **power cycles**, **unsafe**

shutdowns, **media & data-integrity errors**, error-log entry count, warning/critical temperature time, and thermal-management throttle counters. (Requires a drive that supports NVMe admin commands over NVMe-MI.)

Results of every query appear in the **RESULT** panel: the verdict banner, the telemetry strip, and the **Decoded / Bytes / Raw** tabs (*Raw* is the API JSON).

MCTP session control

The **MCTP SESSION** card has five buttons for the MCU's MCTP engine on the active bay — **PAUSE, RESUME, ABORT, STATUS, REPLAY**. Use these when a drive's management endpoint wedges mid-transaction: *ABORT* clears the state machine, *REPLAY* re-sends the last frame for debugging.

> **If every query suddenly returns "No response packets received":** the bay's management path has wedged — this can happen when many MI reads are issued back-to-back without pacing. Recovery is one **SMBus reset** on the bay (Slots page → Resets → SMBus, or `hydractl slot smbus-reset <bay>`); queries resume immediately. When scripting bulk reads, leave a beat (~1 s) between successive MI commands rather than firing them back-to-back.

Async events (AEM)

The **ASYNC EVENTS (AEM)** card enables or disables NVMe-MI Asynchronous Event Messages on the active bay with a **SUBSCRIBE** toggle, and **Read status** reports whether each of the drive's ports advertises AEM support. A warning chip on the card summarises the rules: "gated on Port Capabilities AEMS · subscribe only; refused when the drive reports no AEM support". This is **subscribe-only**: HCC configures the drive but does not yet receive or display the events themselves. A subscribe request is **refused** when the drive reports no AEM support. From the CLI: `hydractl mi aem 3` (status) and `hydractl mi aem 3 --on` / `--off`.

Raw MCTP frame

For protocol work, open **Command Builder → Raw hex**, paste space-separated hex bytes of a complete MCTP-over-SMBus frame and send it. The response packets are shown byte-for-byte. This is the same path the decoded queries use — great for experimenting with new commands before scripting them.

> **A raw send is always one packet.** Every drive's 2-Wire MCTP link starts at a **64-byte transmission unit** (and returns to it on PERST, power-off or an SMBus reset); a single packet whose MCTP payload is larger is silently dropped. The admin-tunnel commands (Identify Controller, SMART / Health Log) are 72-byte requests: HCC's own commands send them fragmented, but their **preview is one 72-byte packet**, so the preview (Guided and Raw hex), `hydractl mi preview`, the preview API and the raw send result all carry a warning — *exceeds the 64-byte baseline transmission unit* — and a raw send of that frame gets no response unless the bay has negotiated a larger unit.

> **Enclosure devices are protected.** A raw frame whose destination byte (the first byte) is `50`, `51`, `80`, `81` or `a0`–`af` is refused: those addresses are the enclosure's own bay devices (IO expander, power monitor, FRU/VPD EEPROMs), so the frame would write to bay infrastructure, not the drive. The NVMe-MI endpoint is normally `3a`.

> **Drive support varies.** Not every SSD implements every NVMe-MI command; an "Unsupported Command" status comes from the drive, not the enclosure.

Exchange History

The **EXCHANGE HISTORY** panel lists every NVMe-MI and VDM exchange of the current server session — bay, protocol and command, status, and latency in ms. Each row offers:

- **Replay** — resend the identical request frame to the same bay.
- **Load** — load the request frame into the **Raw hex** builder to edit and resend.
- **Export** — download the exchange as **JSON**, **CSV**, or **BIN** (the request frame).
- **Select for diff** — tick two rows and click **Diff selected** for a side-by-side comparison of the request frames, response frames and decoded fields.

The same log is available from the API (`GET /console/exchanges`) and the CLI (`hydractl console log`).

Command support probe

The endpoint can be asked **which optional commands it implements** — the NVMe-MI *Optionally Supported Command List* (Read NVMe-MI Data Structure, DTYP `04h`). The console's guided builder uses this to **flag** a command the drive does not advertise, so you know before sending. The result is cached per bay (per connection generation) and refreshable.

> **A hint, not a gate.** The console **warns but does not block** — it still lets you send a "not supported" command, because the drive itself rejects anything it truly does not implement, and this list is **command-set scoped** (it reflects MI-command-set optionality only). On a truncated or mis-framed reply the probe returns *inconclusive* and fails open (nothing is marked unsupported). Do not use it to gate admin-tunnel commands.

API Reference — NVMe-MI

Method	Path	Purpose
GET	`/nvme-mi/{slot}/serial-number`	Quick query: serial number (MCU-parsed)
GET	`/nvme-mi/{slot}/health`	Quick query: health values (MCU-parsed)
GET	`/nvme-mi/{slot}/health-poll`	Decoded Health Status Poll
GET	`/nvme-mi/{slot}/subsystem`	Decoded subsystem info
GET	`/nvme-mi/{slot}/controllers`	Decoded controller list
GET	`/nvme-mi/{slot}/port-info?port=0`	Decoded port info
GET	`/nvme-mi/{slot}/vpd?offset=0&length=256`	VPD read
GET	`/nvme-mi/{slot}/discover`	Full discovery sweep
GET	`/nvme-mi/{slot}/identify`	Identify Controller (vendor/model/serial/firmware/capacity)
GET	`/nvme-mi/{slot}/smart-log`	Full SMART/Health log — parsed fields + 512-byte `raw_hex`
GET	`/nvme-mi/{slot}/supported-commands`	Capability probe (DTYP 04h): optional opcodes + per-catalog-command verdict. `?refresh=true` bypasses the cache
GET	`/nvme-mi/{slot}/aem`	AEM support and current configuration per port

Method	Path	Purpose
POST	`/nvme-mi/aem`	`{"slot": 1, "enable": true}` — subscribe/unsubscribe (refused when the drive reports no AEM support)
POST	`/nvme-mi/raw`	`{"slot": 1, "frame_hex": "3a 0f ...", "timeout": 5}` — one packet; the result adds `transmission_unit` and a `warning` when its payload is over 64 bytes
POST	`/nvme-mi/session`	`{"slot": 1, "action": "abort"}` (pause/resume/abort/status/replay)
GET	`/nvme-mi/catalog`	The command catalog (groups, commands, typed params)
GET	`/nvme-mi/{slot}/preview?command=health-poll`	Build (without sending) a command's annotated frame. `port`/`offset`/`length` and `integrity_check`/`include_pec` refine it. `transmission_unit` gives the packet's payload size vs the 64-byte baseline; `warning` is set when it exceeds it
GET	`/console/exchanges?bay=&protocol=&limit=`	The MCTP exchange log (both protocols)
GET	`/console/exchanges/{id}/export?fmt=json|csv|bin`	Export one exchange (bin = the request frame). Returns the raw file, not the JSON envelope

Decoded query responses carry a `fields` map, a `status`, the `raw_data_hex` payload, and a `pretty` text block, plus `latency_ms`, a `status_decoded {severity, name, description}`, and `response_frame_hex` (the full MCTP response frame, for the byte grid):

```
import httpx

api = httpx.Client(base_url="http://127.0.0.1:8600/api/v1", timeout=30)

health = api.get("/nvme-mi/3/health").json()["data"]
if health["success"]:
    print(health["composite_temperature_celsius"], "degC")

poll = api.get("/nvme-mi/3/health-poll").json()["data"]
print(poll["pretty"])
```

CLI Reference — NVMe-MI

```
hydractl mi catalog                # list commands + their parameters
hydractl mi preview 3 health-poll   # show a command's frame byte-map (no send)
hydractl mi preview 3 port-info --port 1
hydractl console log --bay 3        # the MCTP exchange log
hydractl console export 7 --fmt bin -o bay3.bin   # export exchange #7
hydractl mi sn 3                    # serial number, bay 3
hydractl mi health 3                # quick health values
hydractl mi health-poll 3           # full decoded health poll
hydractl mi smart-log 3             # full SMART/Health log (endurance, temps, power)
hydractl mi smart-log 3 --raw bay3.bin   # also save the raw 512-byte log to a file
hydractl mi supported-commands 3         # capability probe (04h) — which commands the
drive advertises
```

```
hydractl mi supported-commands 3 --refresh   # bypass the cached probe
hydractl mi subsystem 3
hydractl mi controllers 3
hydractl mi port-info 3 --port 0
hydractl mi vpd 3 --offset 0 --length 256
hydractl mi discover 3
hydractl mi session 3 abort         # MCTP session control
hydractl mi aem 3                   # AEM support per port
hydractl mi aem 3 --on              # subscribe (refused without AEM support)
hydractl mi raw 3 "3a 0f 11 21 01 00 00 fc 84 00 00 00 04 45 00 00 cd 21 ec 1e c1"
```

Every command emits JSON with the global flag placed **before** the subcommand: `hydractl --json mi health 3`.

6. Diagnostics

The **Diagnostics** page holds the enclosure's self-test, direct I2C/SMBus access for advanced debugging, and system-level maintenance actions.

Built-In Self Test (BIST)

Click **Run BIST**. The enclosure scans its on-board devices (temperature sensors, fan controller, power monitors, mux) and reports PASS/FAIL for each, listed with its I2C address. Run this first whenever the enclosure behaves strangely — a failing on-board device explains a lot quickly.

System maintenance

- **System Power On / Off** — the enclosure's payload power as a whole (all bays at once). The management MCU stays alive either way. **System Power Off** asks for confirmation.
- **Reset Enclosure** — restarts the MCU. A confirmation dialog guards this: the management link drops for a few seconds and all commanded state (LEDs, PWRDIS, fan duty) reverts to firmware defaults. Drives on a host data path are disturbed only if your configuration powers them from the reset domain.

I2C / SMBus access

Direct register access to devices behind each bay — the sharpest tool in the box, intended for interposer/paddle-card debugging and vendor support sessions.

Fields:

- **Address (hex)** — the **8-bit (write-form)** device address, e.g. `3a` (the NVMe-MI endpoint).
- **Bay** — which slot's bus segment to route to.
- **Register (hex)** — starting register for reads.
- **Length** — bytes to read (1–128).
- **Write data (hex bytes)** — space-separated bytes for writes.

Read returns the bytes as hex and as a decimal array. **Write** asks for confirmation, then reports the byte count. Results appear in the panel below the form, in the exact shape the API returns.

> **Enclosure devices are write-protected.** Writes to the enclosure's own bay devices — `50`/`51` (IO expander), `80`/`81` (power monitor) and `a0`–`af` (FRU/VPD EEPROMs) — are always refused. Reads of those addresses are allowed.

Reading the bus map: some addresses answer on *every* bay — populated or empty — because they are the enclosure's own per-bay devices (the power monitor at `80`, the IO expander at `50`, the FRU EEPROM at `a0`). When scanning for a *drive's* sideband devices, scan an empty bay first as the control: anything that answers there is enclosure-side, not the drive. On a bay with a U.3 interposer, the interposer's IO expander answers at `5e` (presence/interface-detect/board-ID — see the *Slots chapter*).

> **Caution:** writing registers on a live drive's management interface can confuse the drive. Know your device datasheet before writing.

I3C access

The **I3C ACCESS** card exposes the MCU's raw I3C layer with the same shape as the I2C tools plus a **CCC engine** for MIPI I3C Basic v1.1.1 Common Command Codes. Requires an I3C-capable device in the bay.

- **Write** (`i3cw`) — private write of up to 128 bytes to an address behind a bay. Success requires the firmware to echo every byte back. Asks for confirmation.
- **Write-Read** (`i3cwr`) — write (typically a register pointer), then read up to 128 bytes. Asks for confirmation.
- **Address** — the target's **dynamic address in 8-bit (write) form**. After `ENTDAA` assigns the 7-bit address `0x30`, enter `60`.
- **CCC** (`i3c_ccc`) — run a CCC chosen from the catalog list (type to search by name: `GETPID`, `SETMWL`, ...); the API and CLI also accept a hex id. Parameters are validated against the firmware's own table *before* anything is sent, and direct-read CCCs (`GETPID`, `GETBCR`, `GETDCR`, ...) return their bytes. Broadcast and direct-write CCCs ask for confirmation, and so does **`GETACCMST`** (Get Accept Mastership): although it is a direct read, it hands the active-controller role to the target, so the enclosure may lose control of that bay's I3C bus. From the API and CLI, any hex id not in the catalog also needs confirmation. Every other direct-read CCC runs immediately. The full catalog — id, kind (broadcast / direct write / direct read) and parameter counts — is available via `hydractl i3c catalog` or `GET /diagnostics/i3c/ccc/catalog`. A hex id that isn't in the catalog is passed to the enclosure firmware as-is (and needs confirmation, since its effect is unknown); the firmware may refuse it, in which case the result reports `error_kind: not_found` (`[Error] Command not found`).

The I3C page (Technology Preview)

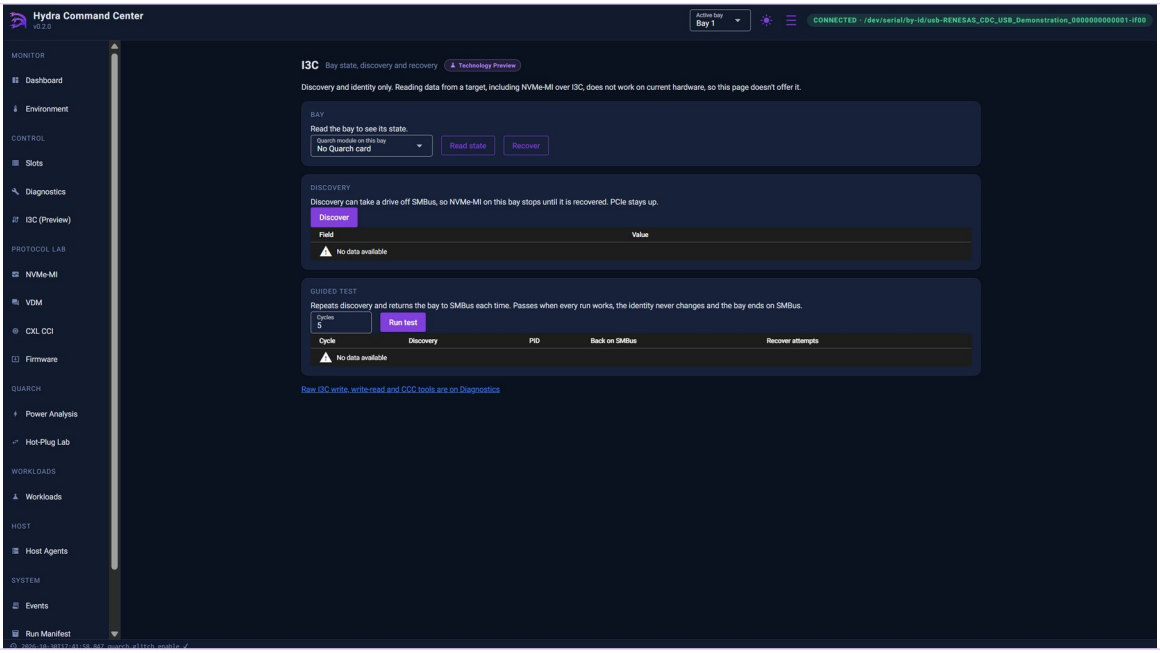


Figure 6.1 I3C (Technology Preview). Read state, Discover, Run test and Recover controls for a bay on the I3C bus.

Control > I3C (Preview) gathers what works today on one page. Pick a bay from the header, then:

- **Read state** shows the bus lines and whether management answers.
- **Discover** assigns a dynamic address and decodes the identity: PID fields, BCR capabilities and DCR.
- **Run test** repeats discovery and returns the bay to SMBus each time. It passes when every run works, the identity never changes and the bay ends on SMBus. From the CLI: `hydractl i3c test <bay> --cycles 5`, or `POST /diagnostics/i3c/test`. `hydractl i3c state <bay>` and `GET /diagnostics/i3c/state/{slot}` read the state.
- **Recover** returns a bay to SMBus (see *Slots*).

The preview covers discovery and identity only. Reading data from a target, including NVMe-MI over I3C, does not work on current hardware, so the page does not offer it. Direct-read CCCs on the card

above may return no data for the same reason. An I3C drive can drop off SMBus for short periods and return on its own.

Bringing a bay up on I3C (API and CLI)

These need an MCU firmware with the I3C test commands (`i3cdaa`, `i3clines`, `i3crecover`).

Step	CLI	API
Read SCL/SDA by GPIO; safe on a held bus	`hydractl i3c lines <bay>`	`GET /diagnostics/i3c/lines/{slot}`
Release a held bus	`hydractl i3c recover <bay>`	`POST /diagnostics/i3c/recover`
Assign a dynamic address, report PID/BCR/DCR	`hydractl i3c daa <bay>`	`POST /diagnostics/i3c/daa`

Recover and DAA ask for confirmation. **DAA is refused, with nothing sent, unless the lines read idle**, because an I3C transfer on a held segment can hang the MCU.

SCL high with SDA low means the drive is holding a Hot-Join or IBI request. The enclosure only runs I3C during a command, so that request waits, and the bay's management and status reads stop while it does. On a drive with a modal sideband, the first I3C traffic also leaves SMBus mode, so NVMe-MI stops until an SMBus reset. A drive can re-assert within seconds of a recovery; if it won't stay released, hot-plug it.

Behind a Quarch breaker: I3C works, with one difference. The card's SMRST# override masks the enclosure's own SMBus reset, so `hydractl slot smbus-reset` has no effect on that bay. Pulse it from the module with a raw command, low then back high, and always put it back to HIGH:

```
hydractl quarch cmd <con> "SIGnal:SMRST#:DRIve:CLOsed LOW" --confirm
hydractl quarch cmd <con> "SIGnal:SMRST#:DRIve:CLOsed HIGH" --confirm
```

I3C writes and write-reads aimed at the enclosure's own bay devices in their 7-bit I3C form (`28` expander, `40` power monitor, `50`–`57` EEPROMs) are refused, the same protection the I2C tools apply.

Failures come back classified in `error_kind`: `timeout` (nothing at that address), `bus_change` (the MCU's I3C controller failed to set up the transfer), `invalid_data` (the firmware rejected the write data), `error_code` (the controller rejected the transfer), or a parameter error. With current firmware a common failure shape is `Failed (Error: 0xNNN)` with a firmware-internal code: **`0x12c`** is the normal "no I3C target there" result (empty bay or SMBus-only drive); other codes indicate a bus-level fault worth capturing. Every result also carries `raw_response` — the verbatim firmware reply — so free-text console output (e.g. `ENTDAA`'s discovered PID/BCR/DCR report) is never lost.

> **Caution:** address-management CCCs (`RSTDAA`, `ENTDAA`, `SETNEWDA`, `SETAASA`, `SETDASA`) reassign dynamic addresses on the bay's bus — a device can stop answering at the address you knew until re-enumerated. HCC has no bay bus-mode switch, but a drive that has been assigned an I3C dynamic address may stop answering on SMBus/NVMe-MI until it is returned to I2C mode — an **SMBus reset** on the bay (Slots → Resets → SMBus) restores it.

Drive Error Log (FA / RMA triage)

The **Error Information Log** (LID `01h`) records a drive's recent command errors — read it out-of-band over NVMe-MI when triaging a failed or suspect DUT. Open it from the *Slots page* **Bay Control** panel: **REPORTS → Error Log** opens a pop-out for the active bay (a sibling of the SMART / Health dialog), with raw and JSON export.

Each entry decodes:

- **Status Code Type** — generic, command-specific, **media-and-data-integrity**, or path — so a **media defect (RMA)** is distinguishable at a glance from a host/command problem. Media entries are flagged red.

- **Status Code** name (e.g. Unrecovered Read Error), plus **Do Not Retry**.
- **LBA** (never truncated — it is triage-critical), **NSID**, submission-queue and command IDs, and command-specific info.

Entries are sorted **most-recent-first**. A clean drive reads as an explicit **"No errors logged"** healthy banner, not an empty table — historical entries are not themselves a fault, but a red **"N media"** chip highlights media errors that warrant attention.

Device Self-Test

Device Self-Test (admin `0x14` + Self-Test Log LID `06h`) runs the drive's own **short** or **extended** internal qualification out-of-band, or **aborts** one in progress. Open it from the Slots page: **REPORTS → Self-Test**.

> The start (or abort) command is a single request, sent at whatever MCTP transmission unit the bay is currently running — it does not reset the unit, so it works on drives that don't implement the unit setting. If the send fails at a negotiated unit above 64 bytes (the drive may have reset to 64 on a PERST HCC didn't issue, e.g. a host reboot), HCC drops the bay back to 64 and reports the failure; retry the command. Resets HCC issues itself (bay power off/on, PERST#, SMBus reset, PWRDIS, enclosure power-off or reset) and a drive the poller sees removed already return the bay to 64 in HCC's books, so a self-test right after them works first time. A start is refused while a firmware download is in progress on the same bay.
>
> **Gated.** Starting a self-test **requires an explicit confirmation** and is refused when the drive's **Identify OACS** bit reports no self-test support. If the OACS read is inconclusive, the start is allowed (fails open) — the confirmation remains the interlock. The *command support probe* is MI-command-set scoped and never lists this admin command. "Already running" is refused with a clear reason rather than double-started.

The dialog polls progress and shows the **latest completed result** with its power-on-hours stamp, in red for any non-zero result code (including aborts). The extended-test confirmation quotes the drive's own **EDSTT** estimate of the test duration. Pass/fail interpretation lives in the **self-test** validation gate: only result codes `0x5`/`0x6`/`0x7` are device **failures** — an aborted or sanitize-interrupted run is **inconclusive** (REVIEW), not a failure.

Validation checks

The **VALIDATION CHECKS** card on each bay's **Bay Detail** page turns the drive read surface into **pass/fail qualification gates**. Click **Run Checks** and every gate reads the drive's state **live** (identity is recalled from the *device library* only for attribution). The seven gates:

Gate	What it checks
link	PCIe link training vs. the drive's max (and an optional expected Gen/width). Asserts GOOD/BAD.
thermal	The **current** composite temperature vs. the drive's own warning threshold (WCTEMP), or 70 °C if it reports none. A `limit_c` target replaces either. The hottest sensor is shown but not gated, because NVMe sets no per-sensor limit. Lifetime warning/critical-temp counters are shown as context, not gated.
reliability	Critical-warning flags, media errors, and available spare vs. the drive's own threshold.
endurance	Percentage of rated life used vs. limit.
firmware	Active firmware revision vs. an expected value (from the live Firmware Slot log).
traceability	Resolved (live-serial-reconfirmed) model + serial vs. an expected DUT manifest.

Gate	What it checks
self-test	The latest Device Self-Test result.

When an enrolled host agent can see the drive, three in-band agreement gates are added — **host-identity**, **host-link** and **host-smart** — comparing what the host sees with the out-of-band view (see *chapter 17*).

Each gate returns **GOOD / WARN / BAD**, and the card rolls them up into one banner. The rollup is **three-state and fail-closed**:

Rollup	Meaning
PASSED	At least one GOOD and no BAD, WARN, or incomplete gate.
REVIEW	A WARN with no BAD or incomplete.
FAILED	Any gate is BAD.
INCOMPLETE	A gate could not verify (state unreadable, DUT unresolved).

> **Incomplete is never a pass.** An unread or unresolved bay reports **INCOMPLETE**, never a green PASS — the dangerous false-positive is closed by design. Failures and incomplete gates are listed first so they jump out.

> **Point-in-time verdicts.** Each run judges the drive's state **now** (the API reports `scope: "point-in-time"`). Delta signals — changes in the unsafe-shutdown count, cumulative over-temperature time, and self-test freshness — are shown as context but not gated; HCC does not yet compare against a per-drive baseline. On a CXL bay, the **cxl-events** gate's verdict is marked `provisional` (see *chapter 16*).

You can also run a single gate, and supply expected targets (link Gen/width, expected firmware) from the CLI or API.

> **Hostless setups:** the management plane needs no host, but the PCIe data link does — a drive powered by the enclosure alone does not train, so the **link** gate reports WARN ("PCIe link not active"). That is the correct verdict, not a fault: run the link gate with a host attached, and everything else works fully hostless.

What each check tests

Every advertised test explains itself with a spec cite. Each check in the panel carries an **info icon** — click it to open a pop-up describing exactly what that gate tests, what it reads **live** from the device, and the governing spec citation. The mapping:

Gate	Reads live	Governing spec
link	NVMe-MI Port Information	PCIe Base Spec — Link Status Register; DMTF DSP0235
thermal	SMART / Health Log, Identify Controller	NVMe Base Spec — Log 02h (Composite / Sensor temperatures), WCTEMP and CCTEMP
reliability	SMART / Health Log	NVMe Base Spec — Log 02h (Critical Warning, Available Spare, Media Errors)

Gate	Reads live	Governing spec
endurance	SMART / Health Log	NVMe Base Spec — Log 02h (Percentage Used)
firmware	Firmware Slot Information Log	NVMe Base Spec — Log 03h (Active Firmware / Slot revisions)
traceability	Identify Controller	NVMe Base Spec — Identify Controller (SN bytes 04–23, MN bytes 24–63)
self-test	Device Self-test Log	NVMe Base Spec — Log 06h (Self-test Result)
cxl-health	CXL Get Health Info (opcode 4200h)	CXL 2.0 §8.2.9.5.3.1 Get Health Info, Table 181
cxl-events	CXL Get Event Records (opcode 0100h)	CXL 2.0 §8.2.9.1.1 Get Event Records, Table 159
cxl-traceability	CXL Identify Memory Device (opcode 4000h)	CXL 2.0 §8.2.9.5.1.1 Identify Memory Device, Table 175

The CXL gates only appear on a CXL bay (see *chapter 16*).

API Reference — Diagnostics

Destructive calls without `"confirm": true` (or `?confirm=true`) return **422**.

Method	Path	Body	Purpose
POST	`/diagnostics/bist`	—	Run self test; returns `{"Device@0xNN": "PASS"}` map
GET	`/diagnostics/clocks`	—	Clock inputs (same as Environment)
POST	`/diagnostics/i2c/read`	`{"address": "3a", "slot": 1, "register": "00", "length": 8}`	Read bytes
POST	`/diagnostics/i2c/write`	`{"address": "3a", "slot": 1, "data_hex": "00 01 02", "confirm": true}`	Write bytes (enclosure bay devices refused)
POST	`/diagnostics/i3c/write`	`{"address": "60", "slot": 1, "data_hex": "a5 5a", "confirm": true}`	I3C private write
POST	`/diagnostics/i3c/write-read`	`{"address": "60", "slot": 1, "read_len": 8, "write_data_hex": "00", "confirm": true}`	I3C write-then-read
POST	`/diagnostics/i3c/ccc`	`{"slot": 1, "command": "ENTDAA", "params_hex": "", "confirm": true}`	Run a MIPI CCC by name or hex id. `confirm` is required for broadcast and direct-write CCCs, for `GETACCMST` (it hands over the active-controller role), and for any hex id not in the catalog; other direct-read CCCs (`GETPID`, `GETBCR`, `GETDCR`, ...) need none
GET	`/diagnostics/i3c/ccc/`	—	Every supported CCC with

Method	Path	Body	Purpose
	`catalog`		kind + param counts
GET	`/slots/{slot}/interposer`	—	U.3 interposer probe: drive presence/type (SAS/NVMe), board id
POST	`/system/power`	`{"state": "off", "confirm": true}`	Enclosure payload power (`confirm` required for off)
POST	`/system/reset?confirm=true`	—	MCU reset
POST	`/system/diagnostics`	—	Alias of BIST under the system tag
GET	`/nvme-mi/{slot}/error-log?entries=64`	—	Error Information Log (01h): recent command errors, media-decoded
GET	`/nvme-mi/{slot}/self-test`	—	Latest self-test result + in-progress status
GET	`/nvme-mi/{slot}/self-test/supported`	—	Whether the drive advertises self-test (Identify OACS)
POST	`/nvme-mi/self-test`	`{"slot": 3, "kind": "short", "confirm": true}`	Start (`short`/`extended`) or `abort` a self-test
GET	`/checks/{slot}`	—	Run every validation gate on a bay (rollup + per-gate verdicts)
GET	`/checks/{slot}/{gate}`	—	Run one gate (422 on an unknown gate)
POST	`/checks/{slot}`	`{"targets": {"link": {"gen": 5, "width": 4}, "firmware": {"expected": "..."}}}`	Run all gates with expected targets
POST	`/checks/{slot}/{gate}`	`{"targets": {...}}`	Run one gate with expected targets

Only the `/checks` routes set `Cache-Control: no-store` — a verdict is a live, ephemeral read.

```
import httpx

api = httpx.Client(base_url="http://127.0.0.1:8600/api/v1", timeout=30)

bist = api.post("/diagnostics/bist").json()["data"]
failures = {device: status for device, status in bist.items() if status != "PASS"}
assert not failures, f"BIST failures: {failures}"

read = api.post("/diagnostics/i2c/read",
                json={"address": "3a", "slot": 1, "register": "00", "length": 8})
print(read.json()["data"]["data_hex"])
```

CLI Reference — Diagnostics

Destructive commands prompt for confirmation; add `--confirm` (or `-y`) in scripts.

```
hydractl system diag                # run BIST (also: hydractl i2c bist)
hydractl system power off           # enclosure payload power (prompts; add --confirm/-
y in scripts)
```

```
hydractl system reset --yes         # MCU reset without prompt
hydractl i2c read 3a 1 00 --length 8
hydractl i2c write 3a 1 "00 01 02"  # prompts; add --confirm/-y in scripts

# I3C — argument order: address, bay, [read length,] data
hydractl i3c write 60 1 "a5 5a"              # prompts; add --confirm/-y in scripts
hydractl i3c wr 60 1 8 00                    # write 00, read 8 bytes back (prompts)
hydractl i3c ccc 1 GETPID                    # direct-read CCC by name (no prompt)
hydractl i3c ccc 1 SETMWL --params "02 00" -y   # broadcast CCC with params (prompts
without -y)
hydractl i3c ccc 1 GETACCMST -y              # direct read, but hands over bus
mastership (prompts without -y)
hydractl i3c ccc 1 8d                        # hex id also accepted (8d = GETPID, no
prompt)
hydractl i3c catalog                         # list every supported CCC

# U.3 interposer / SAS detection
hydractl slot interposer 3                   # expander, presence, drive type

# Drive error log, self-test, and validation gates
hydractl mi error-log 3                     # recent command errors (media-decoded)
hydractl mi error-log 3 --entries 32 --raw bay3-errlog.bin
hydractl mi self-test 3                      # show latest result + progress
hydractl mi self-test 3 --start short --yes  # start a short self-test (confirm
required)
hydractl mi self-test 3 --start abort --yes  # abort a running self-test
hydractl check 3                             # run every validation gate on bay 3
hydractl check 3 thermal                     # run a single gate
hydractl check 3 link --gen 5 --width 4      # gate link against an expected target
hydractl check 3 firmware --expected-fw 1.2.3
```

7. Quarch: Power Analysis

> **Optional hardware.** The Quarch chapters (this one and *chapter 8*) apply only if you use optional Quarch modules with your HYDRA. Quarch support needs Quarch's own Python library, installed on the HCC server host:
>
> `pip install "quarchpy>=2.2"`
>
> Quarch modules connect over their **own USB link**, separate from the HYDRA's management port. Until quarchpy is installed, the navigation shows a locked **QUARCH** heading (tooltip: "Install quarchpy (≥ 2.2) to enable") in place of **Power Analysis** and **Hot-Plug Lab**, and the Quarch API answers HTTP **501** with an install hint. In `--demo` mode both pages work against simulated modules (`SIM:QUARCH-PPM` and `SIM:QUARCH-BREAKER`), with or without quarchpy.

The **Power Analysis** page measures the real power a drive draws, live and in recorded captures, using a Quarch power module or breaker fixture in the drive's power path. Rails and sideband pins are shown together, so you can see what the power did at the moment a pin moved.

Connecting a module

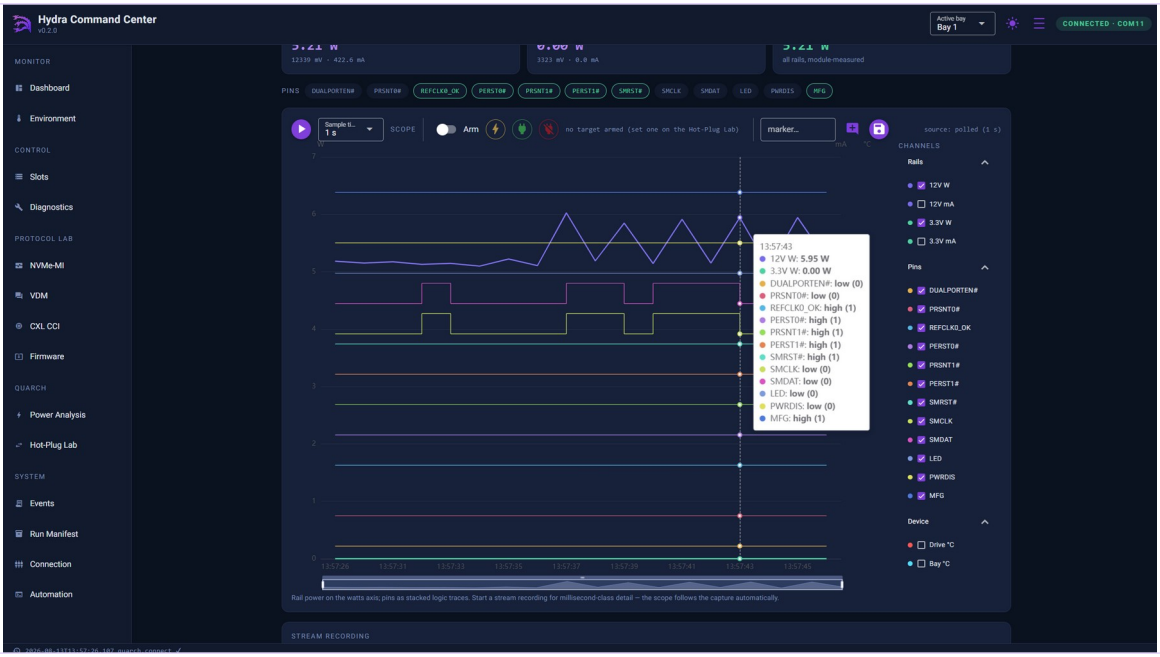


Figure 7.1 Power Analysis. Per-channel rail tiles, a combined rails-and-pins scope, and non-blocking QIS stream recording.

Both Quarch pages start with the same **module bar**:

1. **Scan** discovers attached Quarch modules and fills the dropdown.
2. Select your module.
3. **Connect** opens it. The module's identity appears as chips (**part / serial / fixture**) with the product name in the status line.

The connection lives in the server, so the CLI and API can use the same module immediately. The reverse also holds: **a module already connected anywhere (the other Quarch page, the API, or the CLI) is adopted automatically when a Quarch page loads**, so you connect once per session, not once per page. Click **Disconnect** (the same button) to close it.

The bar also carries a **Raw Torridon command** expansion: a small console for the module's native command set (HELP lists it) with a scrolling response log. Anything the module can do beyond HCC's typed controls is reachable there.

> **Caution:** the raw console is an expert passthrough; HCC does not check what a command does. **Queries** (commands ending in `?`, such as `*IDN?` or `RUN:POWer?`) are sent at once. **Any other command** changes module state, so the console asks for confirmation ("Send raw command?") each time you send one. For example, a confirmed raw `RUN:POWer DOWN` removes drive power immediately.

Scan results to know about:

- A module HCC already holds open is listed with its connection string and marked `(connected)`.
- A module held by **another application** (Quarch Power Studio, or a QIS service left running) is listed as **LOCKED**, "held by another application". Close that application, or restart the HCC server, then scan again.

Linux: USB permissions (udev)

On Windows the Quarch module works with the vendor USB driver and needs no extra setup. On **Linux**, the module (USB vendor `16d0`) needs udev permissions. Without them a scan finds nothing, or connect fails with an access error. This is separate from the MCU management port (*chapter 1*), which only needs `dialout` membership.

The Linux install kit does this for you: `install-hcc.sh` installs `/etc/udev/rules.d/70-hcc-quarch.rules`, which gives Quarch modules to the `hcc` group only (the service and the operators in that group), because a module can cut drive power; `--no-quarch` skips it. Otherwise, Quarch's helper writes the rule and reloads it — run it with the Python that has quarchpy installed (your virtual environment's, if you use one):

```
sudo /path/to/venv/bin/python -m quarchpy.run debug --fixusb   # create and apply the rule
/path/to/venv/bin/python -m quarchpy.run debug                 # re-run WITHOUT sudo to verify
```

Or install the rule by hand. Create `/etc/udev/rules.d/20-quarchmodules.rules`:

```
SUBSYSTEM == "usb", ATTRS{idVendor}=="16d0", MODE="0666"
SUBSYSTEM == "usb_device", ATTRS{idVendor}=="16d0", MODE="0666"
```

(CentOS: append `GROUP="users"` to each line.) Then apply it:

```
sudo udevadm control --reload
sudo udevadm trigger
```

These steps follow Quarch's *USB FAQ*. If a scan still finds nothing, unplug the module's USB cable, plug it back in, and scan again.

Live measurements: every rail, every pin

The measurement tiles are built from the connected fixture's **real channel list**: one tile per analog rail (for example **12V** and **3.3V**) showing watts with mV and mA beneath, plus a **Total Power** tile summing the rails (captioned "all rails, V × I computed by HCC"). Below the tiles, a **PINS** strip shows the live level of every digital sideband pin, highlighted when high.

Every channel comes from one module round-trip, so a full sweep costs the same as a single reading. The tiles fill with one reading when the module connects; they keep updating while the live monitor runs (next section).

Measurement notes:

- Modules report current in different units (power modules in mA, breaker fixtures in µA). HCC reads the unit and always presents **mA**.
- The tiles and the polled scope compute each rail's watts as **V × I** from the module's voltage and current readings; Total Power is the sum of those rails. A stream recording's power columns are computed by the module itself.
- A pin's *level* is not its *routing*. The Hot-Plug Lab's signal matrix shows how a pin is routed (*chapter 8*); the PINS strip shows what the wire is actually doing, pin overrides included.
- Every sweep is also published as `quarch.<rail>.voltage_mv`, `quarch.<rail>.current_ma` and `quarch.<rail>.power_w` channels (plus `quarch.total_power_w` and `quarch.pin.<name>`) for the Workloads monitor and the `/channels` API (*chapter 18*). These are different from `GET /quarch/channels`, which returns the fixture's channel catalog.

The Scope

The **Scope** plots every channel over time: rail power as solid lines on the watts axis, and digital pins stacked beneath as logic-analyzer step traces.

Monitor transport. One reading lands on the scope when the module connects. Click ▶ on the scope bar to start the live monitor and pick a **Sample time** of 250 ms, 500 ms, 1 s, 2 s or 5 s. Click ■ to stop it.

CHANNELS panel. The panel on the right groups the channels; tick a channel to show it, untick it to hide it:

Group	Channels	Shown by default
Rails	Each rail's watts (solid line), and each rail's current in mA (dashed, same colour)	Watts yes; mA no
Pins	Every digital pin as a stacked logic trace	Yes
Device	**Drive °C** (the Active Bay's drive temperature), **Bay °C** (the enclosure's sensor for that bay), and **fio kIOPS** when a host-agent fio job is running on that bay (*chapter 17*)	No

Data source. While the live monitor runs, the scope follows the polled sweep at the chosen Sample time. **While a stream recording is running, the scope switches to the recording's capture**, downsampled for display. The `source:` label at the right of the scope bar shows which feed is in use: `polled (1 s)`, or the recording id with its rate and stride (for example `1:20 shown`). The recording is where hot-plug edges, SMBus transactions and power transients become visible. Zoom with the slider under the chart or with the scroll wheel; for full resolution, download the recording's CSV.

Markers and Save

Actions taken while the monitor runs (plug, pull, glitch, routing changes, and cross-subsystem events such as a bay power-cycle) appear as labelled vertical markers on the trace. An action that failed is never marked. To add your own marker, type a label in the **marker…** box and press **Enter** or the add button; the marker also goes into the event log.

The **Save** (disk) button writes the current scope buffer, with every channel and all markers, to a downloadable CSV in the recordings list (mode `snapshot`). Markers are stored as `# annotation` lines at the top of the file and in its metadata. Save works with or without a stream recording.

> **Start the monitor before the event.** The scope buffers data only while the live monitor runs (plus the single reading taken on connect). If the monitor was stopped when something happened, the scope has nothing to save from that moment.

Stimulus strip

The scope bar also carries four stimulus controls, so you can act while watching the trace:

- **Arm**: nothing fires until Arm is on. Arm switches itself off after **5 minutes**.
- **Glitch** (⚡): repeats the module's armed glitch target, shown beside the buttons as `target: <signal> (<ms> ms)`. The target is the last timed glitch sent to that module (`hydractl quarch glitch …` or `POST /quarch/break/glitch`, *chapter 8*), or the glitch signal of an armed Workloads run (*chapter 18*). With no target, the label reads `no target armed (send a host-timed glitch first)` and the button is disabled.
- **Plug** and **Pull**: connect or disconnect the drive.

> **Plug, Pull and Glitch act immediately, with no confirmation dialog.** Arm is the safety gate. Leave it off unless you intend to fire.

While a timed plug/pull cycle run is active (*chapter 8*), the strip shows its progress and a **stop** button. Stopping a run always re-plugs the drive.

Stream recordings (high-rate capture)

Stream Recording starts a named capture that the server writes to disk while you keep working. On live hardware it uses the module's hardware streaming through Quarch's headless QIS service, which HCC starts automatically on first use. The capture contains every fixture channel: each rail's voltage and current, per-rail and total power computed by the module, and **every digital sideband pin as a 0/1 channel**. A hot-swap capture therefore shows exactly when PRSNT# or PERST# moved relative to the rails.

1. Pick a **Sample time** (`250us`, `1ms`, `10ms`, `100ms` or `1s`; other rates such as `55ms` work through the API and CLI), optionally type a **Note**, and click **Start recording**. The button becomes **Stop recording**, and the status line shows the elapsed time and file size.
2. The page lists the **8 newest** recordings, each with a **download** button (CSV, columns exactly as the module streamed them). `hydractl quarch rec-list` shows all of them.
3. Long soaks are supported: there is no duration limit, and an optional auto-stop (API and CLI) can end an unattended run after up to 7 days.

Where recordings are stored:

- Recordings persist under `quarch-recordings/` in the HCC data directory. The data directory is set by `HCC_DATA_DIR` and defaults to `hcc_data`, relative to the folder the server was started from.
- **On live hardware the recording path must not contain spaces** (a QIS limitation). If it does, starting a recording fails with an error; set `HCC_DATA_DIR` to a folder without spaces and restart the server.
- Nothing is deleted automatically. Delete a recording with `DELETE /quarch/recordings/{id}?confirm=true`.

Module control stays available during a recording: routing, plug and pull, and live measurements keep working, so you can drive the events you are capturing from the same tool.

> One recording per module at a time. While HCC holds a module for streaming, other applications (including Quarch Power Studio) cannot open it, and the reverse is also true.

Record a series (quick polled capture)

For quick checks, set **Samples** (1–600) and **Interval** (0.05–10 s) and click **Record**. **Samples × Interval may not exceed 300 seconds.** The page waits for the capture to finish, then draws the trace with **min / avg / max** watts. Use Stream Recording for anything transient or long-running.

API Reference — Power Analysis

Method	Path	Body / Params	Purpose
GET	`/quarch/availability`	—	Is quarchpy installed? Demo mode? Which modules are connected?
GET	`/quarch/scan`	—	Discover modules
POST	`/quarch/connect`	`{"con_string": "SIM:QUARCH-PPM"}`	Open a module
POST	`/quarch/disconnect`	`{"con_string": "..."}`	Close a module
GET	`/quarch/channels`	`?con_string=...`	The fixture's channel catalog (name, kind, unit, max)
GET	`/quarch/channels/measurements`	`?con_string=...`	One sweep: every rail (mV / mA / W), total W, and pin levels
GET	`/quarch/power/measurements`	`?con_string=...`	One 12V voltage / current / power reading (compatibility view)
POST	`/quarch/power/record`	`{"con_string": "...", "samples": 60, "interval": 0.5}`	Blocking polled capture with min/avg/max (samples 1–600, interval 0.05–10 s, samples × interval ≤ 300 s)
POST	`/quarch/recordings`	see below	Start a stream recording (returns immediately)
GET	`/quarch/recordings`	—	List all recordings, newest first
GET	`/quarch/recordings/{id}`	—	One recording's status and metadata
POST	`/quarch/recordings/{id}/stop`	—	Stop an active recording (the file is kept)
GET	`/quarch/recordings/{id}/tail`	`?offset=0&max_points=500`	New rows since `offset` from a running or finished capture (`max_points` ≤ 5000)
GET	`/quarch/recordings/{id}/csv`	—	Download the capture (CSV)
DELETE	`/quarch/recordings/{id}`	`?confirm=true`	Delete a capture (confirm-gated; a running recording must be stopped first)
POST	`/quarch/command`	`{"con_string": "...", "command": "RUN:POWer?"}`	Raw Torridon command. Queries (ending in ?) need no confirm; any other command needs `"confirm": true` (otherwise 422)

Stream recording body. Only `con_string` is required:

```
{
  "con_string": "SIM:QUARCH-PPM",
  "resample": "1ms",
  "note": "idle soak",
  "auto_stop_s": 3600,
  "header_v3": true,
```

```
  "power_enable": true,
  "power_total_enable": true,
  "averaging": 0
}
```

Field	Default	Meaning
`resample`	`"1ms"`	Stream rate, `<number><us|ms|s>`
`note`	`""`	Free-text label (up to 200 characters)
`auto_stop_s`	none	Stop automatically after 1 s – 7 days (604800 s)
`header_v3`	`true`	Units in the CSV column headers
`power_enable`	`true`	Per-rail power columns, computed by the module
`power_total_enable`	`true`	Total power column
`averaging`	`0`	Module averaging (1–32768); `0` keeps the module's default

> **`/quarch/command` is an expert passthrough.** HCC does not check what a command does, so every command that is not a query (does not end in ?) requires `"confirm": true` and is otherwise refused with HTTP 422. Queries such as `*IDN?`, `RUN:POWer?` and `MEAS:…?` run without it. A confirmed raw `RUN:POWer DOWN` removes drive power immediately.

If quarchpy is missing (and the server is not in demo mode), Quarch endpoints return HTTP **501** with an install hint. Calling a module that is not connected, or passing an out-of-range value, returns HTTP **422**.

```
# Copy-pasteable against a --demo server; on hardware use e.g. "USB:QTL3311-01-XXXX"
modules = api.get("/quarch/scan").json()["data"]
con = modules[0]["con_string"]                    # "SIM:QUARCH-PPM" in demo
api.post("/quarch/connect", json={"con_string": con})

sweep = api.get("/quarch/channels/measurements", params={"con_string": con}).json()
["data"]
print(sweep["total_power_w"], sweep["analog"]["12V"])

rec = api.post("/quarch/power/record",
               json={"con_string": con, "samples": 20, "interval": 0.5},
               timeout=60).json()["data"]
print(rec["stats"])   # {'power_min_w': ..., 'power_max_w': ..., 'power_avg_w': ...}
```

CLI Reference — Power Analysis

Examples use the demo module so you can copy and paste them against a `--demo` server. On hardware, use your module's connection string from `hydractl quarch scan`, for example `USB:QTL3311-01-XXXX`. For JSON output, put the global `--json` option before the command: `hydractl --json quarch measure SIM:QUARCH-PPM`.

```
hydractl quarch scan
hydractl quarch connect SIM:QUARCH-PPM
hydractl quarch channels SIM:QUARCH-PPM           # fixture channel catalog
hydractl quarch measure SIM:QUARCH-PPM            # every rail + pin levels
hydractl quarch power SIM:QUARCH-PPM              # 12V view (compatibility)
hydractl quarch record SIM:QUARCH-PPM --samples 60 --interval 0.5
```

```
# Stream recordings (QIS hardware streaming on live modules)
hydractl quarch rec-start SIM:QUARCH-PPM --resample 1ms --note "idle soak" --auto-stop
3600
hydractl quarch rec-list                          # ids (narrow terminals may
truncate: use --json)
hydractl quarch rec-stop <id>
hydractl quarch rec-export <id> -o soak.csv       # default file name: <id>.csv

hydractl quarch cmd SIM:QUARCH-PPM "RUN:POWer?"    # raw query (no prompt)
hydractl quarch cmd SIM:QUARCH-PPM "RUN:POWer UP" -y   # non-query: prompts unless -y
hydractl quarch disconnect SIM:QUARCH-PPM
```

`rec-start` options:

Option	Default	Meaning
`--resample`	`1ms`	Stream rate, e.g. `250us`, `1ms`, `1s`
`--note`	empty	Free-text label
`--auto-stop N`	none	Stop after N seconds
`--no-header-v3`	header v3 on	Omit units from the CSV column headers
`--no-power`	on	Omit the per-rail power columns
`--no-power-total`	on	Omit the total power column
`--averaging N`	`0`	Module averaging 1–32768; `0` = module default

Recording ids have the form `rec-<yyyymmdd>-<hhmmss>-<module>`, for example `rec-20260101-120000-sim-quarch-ppm`. There is no CLI delete command; use `DELETE /quarch/recordings/{id}?confirm=true`.

8. Quarch: Hot-Plug Lab

> **Optional hardware.** The Hot-Plug Lab needs a Quarch breaker / drive-control module and `pip install "quarchpy>=2.2"` on the HCC server host. In `--demo` mode it works against the simulated module `SIM:QUARCH-BREAKER`. See *chapter 7* for installation, the locked **QUARCH** navigation heading, and the shared module bar (Scan → select → Connect).

A drive's connector carries more than power and PCIe lanes: **sideband signals** such as PERST#, the SMBus pair, PWRDIS and the presence pins. A Quarch breaker module sits inline on these signals and on the power rails, and can connect, disconnect, delay or glitch each one.

The **Hot-Plug Lab** (`/quarch/hotplug`) is where you use it. The workflow is configure, then execute:

1. **Configure** every signal, power rails included: which timed source it follows, or whether it is broken.
2. **Shape the timing** of the six timed sources.
3. **Execute**: plug, pull, run timed plug/pull cycles, or glitch.

The page has three areas below the module bar: the **transport strip** (always visible while you scroll), the **SIGNAL MATRIX** with the glitch engine, and the **SOURCES** panel.

Plug, pull and timed cycling

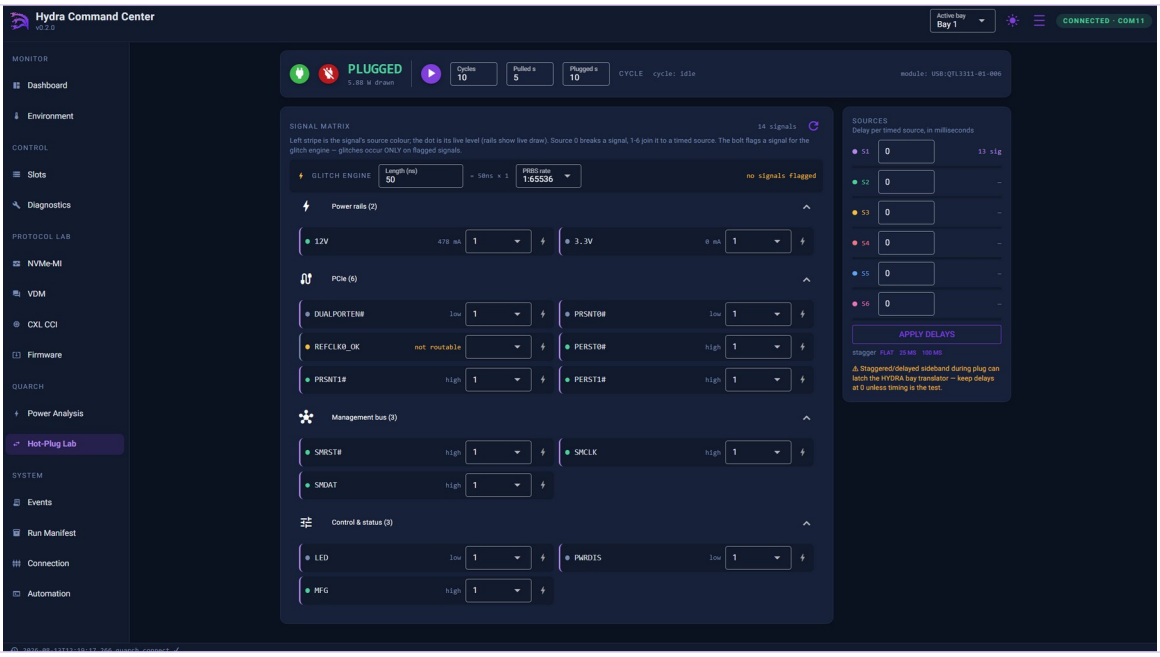


Figure 8.1 Hot-Plug Lab. Fixture-discovered signal matrix with per-source delays and the hardware glitch engine.

The transport strip stays pinned at the top of the page while the matrix scrolls.

- **Plug** (green round button) connects the drive; **Pull** (red round button) disconnects it.

> **Plug and Pull act immediately, with no confirmation dialog.** The Lab is an expert bench tool: Plug, Pull, choosing source 0 and the glitch bolt all act at once. Starting a cycle run is the one signal-matrix or transport action that asks for confirmation (the module bar's raw Torridon console also confirms any command that is not a query).

- The **state readout** shows **PLUGGED** (green) or **PULLED** (red), and below it the module's total draw, for example `14.45 W drawn`. The draw is the proof that a swap actually happened: a pulled drive draws next to nothing.
- **Timed cycling** runs repeated pull/plug cycles on the server:
  - **Cycles** (1–100000), **Pulled s** and **Plugged s** (0.5–3600 s each) set the run. Each cycle pulls, waits *Pulled s*, plugs, then waits *Plugged s*.
  - ▶ opens a **"Start endurance cycling?"** dialog showing the cycle count, both dwell times and the estimated total duration. Click **Run cycles** to start.
  - A progress bar and a label (`cycling 3/10 · pull`) track the run. When it ends, the label shows `cycle run done: 10/10`, or the error if the run failed.
  - ■ stops the run. **A stopped, finished or failed run always leaves the drive plugged.**
  - The manual Plug and Pull buttons are disabled while a run is active. Only one run per module can be active at a time.
- **ENUM** shows the latest hot-plug enumeration timing that the host agent measured for the Active Bay (plug → link-up, plug → device visible). See *chapter 17*.
- The **module** label on the right shows the connected module, and turns red with `unreachable` if the module stops answering.

A cycle run uses whatever signal routing and source timing are configured at the time, so set those up first. The Power Analysis page shows the same run's progress and a stop button on its stimulus strip (*chapter 7*).

The signal matrix

The **SIGNAL MATRIX** loads automatically when a module connects. Click the refresh button in its header to re-read signals, routing and source timing from the module. The header also shows a summary such as `13 signals · 1 broken`.

Signals are grouped into collapsible sections: **Power rails**, **PCIe**, **Management bus** and **Control & status**. Each row shows:

- A **left stripe** in the colour of the signal's source. The SOURCES panel uses the same colours.
- A **live-level dot and readout**, refreshed every 5 seconds: `high` or `low` for pins (dot lit when high), and the live current such as `1200 mA` for rails (dot lit when the rail is drawing current).
- A **source dropdown**: `0 · broken`, 1 … 6, `7 · immediate`, `8 · always on`. Choosing a value re-routes the signal immediately.
- A **bolt** button that flags the signal for the hardware glitch engine (*below*).

A **broken** row (source 0) shows its name in red strikethrough with a red dot. A **monitor-only** channel, which the module can measure but not route, shows an amber dot and `not routable`.

Signals are discovered, not assumed

Different Quarch fixtures expose different signals under different names. On connect, HCC reads the fixture's own channel list and builds the matrix from **the fixture's real signal names**, power rails first. For example, the E3 breaker fixture used with HYDRA bays reports these sideband signals:

```
DUALPORTEN#  PRSNT0#  REFCLK0_OK  PERST0#  PRSNT1#  PERST1#
SMRST#  SMCLK  SMDAT  LED  PWRDIS  MFG
```

`REFCLK0_OK` is monitor-only on that fixture. A PCIe-slot or U.2 fixture reports a different set. Whatever the fixture reports is what the matrix, the API and the CLI accept; any other name is rejected as an unknown signal.

The demo module `SIM:QUARCH-BREAKER` shows a generic set instead:

```
12V  3.3V  PERST  CLKREQ  WAKE  SMBCLK  SMBDAT  DUALPORTEN
PWRDIS  PRSNT  REFCLK  3V3AUX  12V_POWER
```

(In the demo, `3.3V` shows as `not routable`.) The demo examples in this chapter therefore use names such as PERST and SMBCLK; on the E3 fixture use `PERST0#` and `SMCLK`.

What "sources" mean

Quarch modules route each signal onto a **source**:

Source	Meaning
`0`	ALWAYS_OFF: the signal is broken (disconnected)
`1–6`	Timed sources: connected, and each can be delayed independently on plug and pull (*Source timing*)
`7`	IMMEDIATE: switches with no timing applied
`8`	ALWAYS_ON: permanently connected, whatever the plug state

Moving a signal to a different timed source changes which timing group it belongs to. Routing it to `0` disconnects it entirely (*Breaking signals*). The API and CLI also accept `NONE` as an alias for `0`.

A bay behind a QTL3311 breaker that looks dead: this card doesn't follow the enclosure's control pins, so PWRDIS and SMRST# stay asserted until the module overrides them. The drive then draws only a few mA, won't link and doesn't answer, although the module reads `PLUGGED`. It isn't a hardware fault. Run `examples/quarch_bay_bringup` to set both overrides. They survive an HCC restart but not an enclosure power loss, so run it again after one.

Pin overrides: the module can also actively drive a pin high or low, independent of its routing (`SIGnal:<name>:DRIve:OPEn|CLOsed HIGH|LOW`). The matrix dropdown shows *routing*, not overrides. If a pin's live level doesn't match its routing, check for an override with the **Raw Torridon command** console in the module bar.

Source timing

Real connectors have staggered pin lengths: grounds mate first, then power, then the short signal pins. The **SOURCES** panel reproduces this by delaying each timed source.

- **S1–S6** each have a delay field in milliseconds (0–60000), a count of the signals currently assigned to that source (`3 sig`), and a bar showing the delay relative to the longest one. The fields load from the module when it connects.
- **Apply delays** sends all six delays to the module.
- The **stagger** presets **flat**, **25 ms** and **100 ms** fill S1–S6 with 0, 1, 2 ... 5 steps of that size. They only fill the fields; nothing reaches the module until you click **Apply delays**.

To build a scenario, assign signals to sources in the matrix, then give the sources delays. For example: rails on S1 at 0 ms, sideband pins on S3 at 100 ms, and PERST# on S5 at 250 ms. The timing stays in effect for every following plug and pull, and for cycle runs, until you change it.

Two timing features are available through the API and CLI only:

- **Pin bounce**: per-source bounce shaping (length, period, duty), set with `POST /quarch/hotswap/timing` or `hydractl quarch hotswap-timing --bounce`.
- **Named timing profiles**: `instant`, `fast`, `normal` and `slow` set a 0, 25, 100 or 500 ms step across S1–S6, via `POST /quarch/hotswap/configure` or `hydractl quarch hotswap-config`.

HYDRA caution: staggered or delayed sideband during a plug can latch the HYDRA bay's level translator and disturb the enclosure's shared management bus: the other bays stop answering until the drive is pulled again. Keep all delays at 0 (**flat**) unless timing is what you are testing, and treat a management-bus stall right after a staggered plug as this effect.

Breaking signals

Breaking is targeted fault injection: disconnect **one specific signal** while everything else stays connected, and watch how the drive and host cope. Where a pull disturbs the whole connector, a break is a scalpel.

- **Break** a signal: choose `0 · broken` in its source dropdown. The drive sees a genuine physical disconnect on that pin until you restore it.
- **Restore** it: choose a source again (usually the timed source it was on, such as `1`).

In the Lab, choosing source 0 breaks the signal immediately, with no confirmation dialog. The Lab is an expert bench tool. This applies to power rails too: breaking 12V cuts drive power. (Through the API and CLI, routing a signal to source 0 requires confirmation.)

Host-timed glitches (break, wait, restore) are available through the API and CLI: `POST /quarch/break/glitch` and `hydractl quarch glitch`, with a duration of 1–60000 ms. The call returns after the signal is restored. The timing is kept by the server, so short durations are approximate; for precise, sub-millisecond glitches use the *hardware glitch engine*. The most recent host-timed glitch on a module also becomes the target of the **Glitch** button on the Power Analysis page's stimulus strip (*chapter 7*), so you can repeat it while watching the scope.

The Lab has no separate Break, Restore or timed-Glitch buttons; use the source dropdown, or the API and CLI.

The hardware glitch engine

Beyond host-timed glitches, the module carries a **hardware glitch engine** (Quarch's native model). Its controls are in the **GLITCH ENGINE** row at the top of the signal matrix:

1. **Flag signals.** Each matrix row's bolt button toggles that signal's glitch flag. Glitches occur **only on flagged signals**; the flag is the selection. The row lists the flagged signals (`flagged: PERST0#`).
2. **Configure the engine** (applies to all flagged signals):
   - **Length (ns)**: the hardware builds a glitch length as a step size (50 ns, 500 ns, 5 µs, 50 µs, 500 µs, 5 ms, 50 ms or 500 ms) × a count of 1–31, so lengths run from 50 ns to 15.5 s. HCC snaps your request to the nearest achievable value and shows how it was built, for example `= 500ns × 4` for 2000 ns. A new length is sent when you leave the field.
   - **PRBS rate**: the pseudo-random glitch rate, `1:2` (aggressive) through `1:65536` (sparse). A new rate is sent as soon as you pick it.
3. Glitching runs on the module's own timing while signals are flagged, producing nanosecond-class events that no host-timed path can. Unflag every signal to stop.

Warning: in the Lab, clicking a bolt starts hardware glitching on that signal **immediately**, with no confirmation dialog. (Through the API and CLI, flagging requires confirmation.) Changing the length or rate while signals are flagged changes the live glitching at once, with no confirmation anywhere.

Test recipes

Signal names below are from the E3 breaker fixture; your fixture's list may differ (see *Signals are discovered, not assumed*).

Signal	Break or glitch it to test…
PERST0#	Surprise-reset recovery and link retraining
SMCLK / SMDAT	Management-path loss while the data path stays up. Breaking both also shows whether a host problem is (or is not) SMBus-related; restore afterwards
PRSNT0#	Presence-detect handling and hot-plug controller logic: a drive that is electrically present but reports absent
12V	Brown-out and power-loss protection (use short glitches)
PWRDIS	Drive power-disable semantics
REFCLK (on fixtures that route the reference clock)	Clock-loss behaviour and SRIS/SRNS clocking assumptions

Start with long glitches (easy to see and to correlate with host logs), then shorten toward the timing corner you care about.

Verify wiring before a campaign. Connect the module and check that every expected signal is present in the matrix, on the source you expect, and not broken.

A typical hot-swap test:

1. Click **flat**, then **Apply delays** (or apply the scenario under test).
2. **Pull**, and verify that the host handles the surprise removal cleanly.
3. Wait for your host-side settle time.
4. **Plug**, verify re-enumeration (the **ENUM** readout shows the timing if a host agent is enrolled), then check drive health on the *NVMe-MI page* (`hydractl mi health <bay>`).
5. Repeat with the source timings you need to cover, or hand the sequence to a timed cycle run.

> **Safety:** breaking or glitching power signals on a drive with active writes can cause data loss on drives without power-loss protection. That may be exactly what you are testing, but do it with expendable data.

API Reference — Hot-Plug Lab

All bodies take `con_string`, the module's connection string (for example `SIM:QUARCH-BREAKER`, or `USB:QTL3311-01-XXXX` on hardware). **Confirm** means the call is refused with HTTP **422** unless the body includes `"confirm": true`.

Method	Path	Body / Params	Confirm	Purpose
GET	`/quarch/sideband/signals`	`?con_string=...`	—	The fixture's sideband pin names (pins only)
GET	`/quarch/sideband/status`	`?con_string=...`	—	Source of every sideband pin
GET	`/quarch/signals/routable`	`?con_string=...`	—	Every routable signal, power rails first
GET	`/quarch/signals/routing`	`?con_string=...`	—	Source of every routable signal (monitor-only channels answer `FAIL…`)

Method	Path	Body / Params	Confirm	Purpose
POST	/quarch/sideband/set	{"signal": "PERST", "source": "2"}	**Yes for `"0"`**	Route a signal or rail to source "0"–"8". Source "0" breaks it and needs "confirm": true; other sources do not
GET	/quarch/hotswap/timing	?con_string=...	—	Delay and bounce settings of sources 1–6
POST	/quarch/hotswap/timing	{"delays": {"1": 0, "3": 100}, "bounce": {...}}	No	Set per-source delays and optional pin bounce
POST	/quarch/hotswap/configure	{"speed": "instant"}	No	Named profile: instant / fast / normal / slow (0 / 25 / 100 / 500 ms step)
POST	/quarch/hotswap/run	{"action": "pull", "confirm": true}	**Yes**	Plug or pull once ("plug" / "pull"). A plug or pull the module refuses returns HTTP 422
GET	/quarch/hotswap/status	?con_string=...	—	Current state (PLUGGED / PULLED)
POST	/quarch/hotswap/cycle	{"cycles": 100, "pull_dwell_s": 5, "plug_dwell_s": 10, "confirm": true}	**Yes**	Start a timed cycle run on the server (returns immediately)
GET	/quarch/hotswap/cycle	?con_string=...	—	Cycle run status
POST	/quarch/hotswap/cycle/stop	{"con_string": "..."}	No	Stop the active run; always re-plugs
POST	/quarch/break/break	{"signal": "PERST", "confirm": true}	**Yes**	Break a signal (route to source 0)
POST	/quarch/break/restore	{"signal": "PERST", "source": "1"}	No	Restore a signal to a source (the recovery direction)
POST	/quarch/break/glitch	{"signal": "PERST", "duration_ms": 250, "source": "1", "confirm": true}	**Yes**	Host-timed break → wait → restore; arms the Power page's Glitch button
GET	/quarch/glitch/engine	?con_string=...	—	Glitch-engine length, PRBS rate and every signal's flag
POST	/quarch/glitch/engine	{"length_ns": 2000, "prbs": 16}	No	Set the length (snapped) and/or PRBS rate; affects flagged signals at once
POST	/quarch/glitch/enable	{"signal": "PERST", "enabled": true, "confirm": true}	**Yes for `"enabled": true`**	Flag / unflag a signal. Flagging starts hardware glitching immediately and needs "confirm": true; unflagging

Method	Path	Body / Params	Confirm	Purpose
				`("enabled": false)` does not
POST	`/quarch/command`	`{"command": "RUN:POWer?"}`	**Yes for non-queries**	Raw Torridon passthrough (*chapter 7*). Queries (ending in ?) are open; any other command needs `"confirm": true`

Calls without a confirm gate: restore, routing to a live source (`/sideband/set` with `"1"–"8"`), unflagging a glitch signal and cycle stop are ungated on purpose, because they are the recovery direction. `/glitch/engine` changes the length and rate of any glitching already running on flagged signals. Raw `/quarch/command` queries (ending in ?) are also ungated; any other raw command needs `"confirm": true`, because HCC does not check what it does (a raw `RUN:POWer DOWN` removes drive power immediately).

Limits:

Parameter	Range
Source delay (`delays`)	0–60000 ms, keys `"1"–"6"`
Bounce (`bounce`)	`{"<source>": {"length_ms": …, "period_ms": …, "duty_pct": …}}`; `length_ms` and `period_ms` 0–60000, `duty_pct` 0–100
`cycles`	1–100000
`pull_dwell_s`, `plug_dwell_s`	0.5–3600 s (defaults 5 and 10)
Host-timed glitch `duration_ms`	1–60000 ms (default 100)
Glitch-engine `length_ns`	50 ns – 15.5 s, snapped to step (50 ns – 500 ms) × count (1–31)
Glitch-engine `prbs`	2, 4, 8 … 65536 (rate 1:N)

`POST /quarch/hotswap/timing` needs at least one delay or bounce entry and returns the resulting timing of all six sources. The cycle status returns `running`, `cycle`, `cycles`, `pull_dwell_s`, `plug_dwell_s`, `started_at`, `stopped_at`, `last_action` and `error`; stopping when no run is active returns HTTP 422. A module answer of `FAIL…` to a routing, plug/pull or glitch-engine call is returned as HTTP 422, not as success.

The examples below run against a `--demo` server. On the E3 fixture, use `PERST0#` in place of `PERST`.

```
con = "SIM:QUARCH-BREAKER"          # hardware: e.g. "USB:QTL3311-01-XXXX"
SIGNAL = "PERST"                    # E3 breaker fixture: "PERST0#"
api.post("/quarch/connect", json={"con_string": con})

routing = api.get("/quarch/signals/routing", params={"con_string": con}).json()["data"]
print("Broken:", [s for s, src in routing.items() if src == "0"] or "none")

# Build a scenario: PERST on source 5, delayed 250 ms; everything else on S1 at 0 ms
api.post("/quarch/sideband/set", json={"con_string": con, "signal": SIGNAL, "source":
"5"})
api.post("/quarch/hotswap/timing",
         json={"con_string": con, "delays": {"1": 0, "5": 250}})
```

Endurance cycling on the server (preferred: the run keeps going if your script exits, and always ends plugged):

```
import time

api.post("/quarch/hotswap/cycle",
         json={"con_string": con, "cycles": 100, "pull_dwell_s": 5,
               "plug_dwell_s": 10, "confirm": True})
while True:
    status = api.get("/quarch/hotswap/cycle", params={"con_string": con}).json()
["data"]
    print(f"cycle {status['cycle']}/{status['cycles']} ({status['last_action']})")
    if not status["running"]:
        break
    time.sleep(15)
```

The same 100 cycles as a client loop, when you need a check after every cycle:

```
import time

for cycle in range(100):
    api.post("/quarch/hotswap/run",
             json={"con_string": con, "action": "pull", "confirm": True})
    time.sleep(5)
    api.post("/quarch/hotswap/run",
             json={"con_string": con, "action": "plug", "confirm": True})
    time.sleep(10)
    health = api.get("/nvme-mi/3/health").json()["data"]
    assert health["success"], f"Drive lost after cycle {cycle}"
```

Glitch-duration sweep with a drive check after each glitch:

```
for ms in (1000, 500, 250, 100, 50):
    api.post("/quarch/break/glitch",
             json={"con_string": con, "signal": SIGNAL,
                   "duration_ms": ms, "confirm": True},
             timeout=ms / 1000 + 30)
    health = api.get("/nvme-mi/3/health").json()["data"]
    print(ms, "ms ->", "OK" if health["success"] else "DRIVE LOST")
```

CLI Reference — Hot-Plug Lab

Examples use the demo module and its signal names, so you can copy and paste them against a `--demo` server. On hardware, use your module's connection string (for example `USB:QTL3311-01-XXXX`) and your fixture's names (for example `PERST0#`, `SMCLK`). For JSON output, put the global `--json` option before the command: `hydractl --json quarch hotswap-cycle SIM:QUARCH-BREAKER`.

`hotswap`, `hotswap-cycle --cycles`, `break`, `glitch`, `glitch-enable … on`, and `sideband-set` with source `0` ask for confirmation; add `-y` (or `--confirm`) for scripted runs. The restore direction never prompts: `sideband-set` to any other source, `restore`, `glitch-enable … off` and `hotswap-cycle --stop`. `hotswap-timing`, `hotswap-config` and `glitch-engine` also act without a prompt.

```
hydractl quarch connect SIM:QUARCH-BREAKER

# Signals and routing (sideband pins)
hydractl quarch signals SIM:QUARCH-BREAKER                  # fixture's pin names
hydractl quarch sideband SIM:QUARCH-BREAKER                 # routing table
hydractl quarch sideband-set SIM:QUARCH-BREAKER PERST 2     # move PERST to source 2
hydractl quarch sideband-set SIM:QUARCH-BREAKER SMBCLK 0 -y # break the SMBus clock
hydractl quarch sideband-set SIM:QUARCH-BREAKER SMBCLK 1    # restore it
```

```
# Source timing
hydractl quarch hotswap-timing SIM:QUARCH-BREAKER                          # show S1-S6
hydractl quarch hotswap-timing SIM:QUARCH-BREAKER --set '1=0,3=100'        # per-source
delays (ms)
hydractl quarch hotswap-timing SIM:QUARCH-BREAKER --bounce
'3:length=5,period=1,duty=50'
hydractl quarch hotswap-config SIM:QUARCH-BREAKER instant                  # named
profile

# Plug, pull and cycling
hydractl quarch hotswap SIM:QUARCH-BREAKER pull -y          # disconnect the drive
hydractl quarch hotswap SIM:QUARCH-BREAKER plug -y          # connect it
hydractl quarch hotswap-status SIM:QUARCH-BREAKER           # PLUGGED / PULLED
hydractl quarch hotswap-cycle SIM:QUARCH-BREAKER --cycles 100 --pull-dwell 5 --plug-
dwell 10 -y
hydractl quarch hotswap-cycle SIM:QUARCH-BREAKER            # no options = status
hydractl quarch hotswap-cycle SIM:QUARCH-BREAKER --stop     # stop (always re-plugs)

# Breaking and host-timed glitches
hydractl quarch break SIM:QUARCH-BREAKER PERST -y                    # break
hydractl quarch restore SIM:QUARCH-BREAKER PERST --source 1          # restore
hydractl quarch glitch SIM:QUARCH-BREAKER PERST --duration 250 -y    # 250 ms glitch

# Hardware glitch engine
hydractl quarch glitch-engine SIM:QUARCH-BREAKER --length-ns 2000 --prbs 16
hydractl quarch glitch-enable SIM:QUARCH-BREAKER PERST on -y   # starts glitching NOW
hydractl quarch glitch-engine SIM:QUARCH-BREAKER               # show config + flags
hydractl quarch glitch-enable SIM:QUARCH-BREAKER PERST off     # stop (no prompt)
```

Command	Options
`hotswap-timing <con>`	No options shows the timing. `--set '1=0,3=100'` sets delays (source=ms); `--bounce '<source>:length=…,period=…,duty=…'` shapes one source's bounce
`hotswap-cycle <con>`	No options shows status. `--cycles N` starts a run; `--pull-dwell S` (default 5), `--plug-dwell S` (default 10); `--stop` ends the run; `-y` skips the prompt
`hotswap <con> plug|pull`	`-y` skips the prompt
`break <con> <signal>`	`-y` skips the prompt
`restore <con> <signal>`	`--source N` (default 1)
`glitch <con> <signal>`	`--duration MS` (1–60000, default 100), `--source N` to restore to (default 1), `-y`
`glitch-engine <con>`	No options shows the engine. `--length-ns N`, `--prbs N`
`sideband-set <con> <signal> <source>`	`-y` skips the prompt (asked for source `0` only)
`glitch-enable <con> <signal> on|off`	`-y` skips the prompt (asked for `on` only)

`signals` and `sideband` list the sideband pins only. To see the routing of the power rails as well, use `GET /quarch/signals/routing`. `sideband-set`, `break`, `restore`, `glitch` and `glitch-enable` accept the rails too (for example 12V).

Older single-purpose pages remain reachable at `/quarch/sideband`, `/quarch/hotswap` and `/quarch/breaking`.

9. Quarch: Hot Swap

Hot swap is now part of the **Hot-Plug Lab** (*chapter 8*). Plug, pull and timed plug/pull cycling are covered in *Plug, pull and timed cycling*, and per-source timing (pin-length emulation) in *Source timing*. The API and CLI commands are listed in the chapter's *API Reference* and *CLI Reference*.

10. Quarch: Signal Breaking

Signal breaking and glitching are now part of the **Hot-Plug Lab** (*chapter 8*). Breaking, restoring and host-timed glitches are covered in *Breaking signals*, nanosecond-class glitching in *The hardware glitch engine*, and per-signal test ideas in *Test recipes*.

11. Automation Guide

Everything you have done in this manual by clicking can be scripted. This section pulls the API and CLI addendums together into working automation patterns. The in-app **Automation** page shows live links to the interactive API docs served by your own instance.

The three scripting surfaces

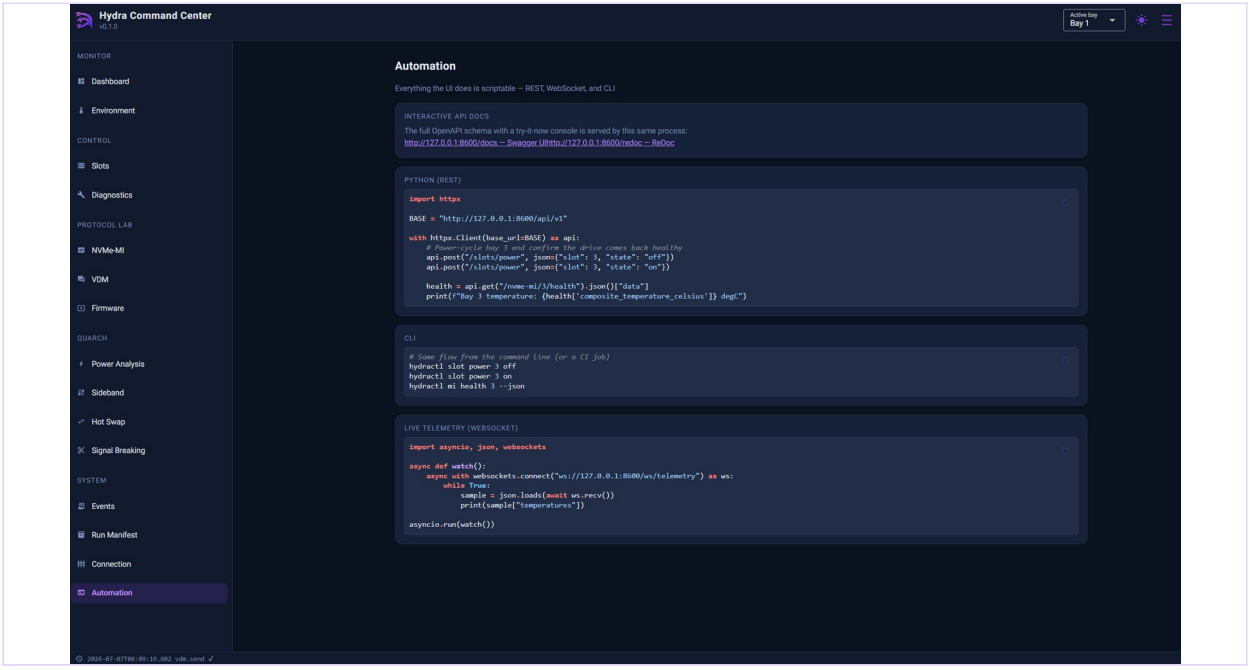


Figure 11.1 Automation. Live links to the interactive REST API docs served by your own instance.

1. **REST API** — `http://<host>:8600/api/v1/...`. Language-agnostic. Interactive docs at `/docs` (Swagger UI, try-it-now) and `/redoc`.
2. **WebSocket** — `ws://<host>:8600/ws/telemetry` pushes every telemetry sample as JSON; no polling.
3. **CLI** — `hydractl` with the global `--json` flag (placed before the command, e.g. `hydractl --json inventory`) gives shell scripts and CI jobs the same data as the API.

All API responses share one envelope:

```
{"success": true,  "data": { ... }}
{"success": false, "error": "human-readable message"}
```

Two endpoints return raw payloads instead of the envelope, because they are file downloads: `/console/exchanges/{id}/export` and `/environment/telemetry.csv`.

HTTP status codes:

Code	Meaning
401	Missing or wrong API token (non-local bind; see below)
404	Unknown resource (e.g. an exchange id that does not exist)
409	Not connected, or the request conflicts with current state
422	Invalid parameters — **or** a destructive call sent without

Code	Meaning
	`"confirm": true`
501	Optional feature unavailable (Quarch)
502	MCU did not acknowledge

Remote access & tokens

- **Local bind** (the default, `127.0.0.1`) needs no token.
- **Non-local bind** (e.g. `hydractl serve --host 0.0.0.0`) enforces a bearer token on every HTTP request and WebSocket connection. Set it with the `HCC_API_TOKEN` environment variable or `--token`; if neither is set, the server generates one and prints it at startup.
- **Scripts and API clients** send `Authorization: Bearer <token>`.
- **Browsers** open `http://<host>:8600/?token=<token>` once; the UI then stays authenticated for that browser session.
- **`hydractl`** reads `HCC_API_TOKEN` automatically — export it on the client machine, and point the CLI at the server with `HCC_URL` or `--url`. For a self-signed server certificate (the install kits' default), set `HCC_CA_CERT` to a copy of it (`/etc/hcc/tls/cert.pem`, or `C:\ProgramData\SerialCables\HCC\tls\cert.pem`, on the server). On the server itself none of this is needed: with no `HCC_URL` set, `hydractl` reads the installed server's `client.env` (URL, token, certificate) — and sends that token only to that server.

A token sent over plain HTTP can be read by anyone on the network path. For any non-local deployment, serve over TLS with `--tls-cert <cert.pem> --tls-key <key.pem>` (or `HCC_TLS_CERT` / `HCC_TLS_KEY`) and use `https://` URLs.

```
export HCC_API_TOKEN="<token printed by the server>"
export HCC_URL="https://hydra-lab-01:8600"
export HCC_CA_CERT="$HOME/hydra-lab-01.pem"     # only for a self-signed certificate
hydractl --json system info
```

Python quickstart

```
import os
import httpx

headers = {}
if os.environ.get("HCC_API_TOKEN"):           # only needed for a non-local server
    headers["Authorization"] = f"Bearer {os.environ['HCC_API_TOKEN']}"

api = httpx.Client(base_url="http://127.0.0.1:8600/api/v1",
                   headers=headers, timeout=30)

def call(method: str, path: str, **kwargs):
    body = api.request(method, path, **kwargs).json()
    if not body["success"]:
        raise RuntimeError(body["error"])
    return body["data"]

# Ensure we're connected (demo or hardware)
status = call("GET", "/connection/status")
if not status["connected"]:
    call("POST", "/connection/connect", json={"port": "COM3"})

# Power-cycle every populated bay and verify health
```

```
for slot in call("GET", "/slots"):
    n = slot["slot_number"]
    if not slot["present"]:
        continue
    call("POST", "/slots/power", json={"slot": n, "state": "off", "confirm": True})
    call("POST", "/slots/power", json={"slot": n, "state": "on"})
    health = call("GET", f"/nvme-mi/{n}/health")
    print(f"Bay {n}: {health['composite_temperature_celsius']} degC")
```

Live telemetry over WebSocket

```
import asyncio, json, websockets

async def watch():
    async with websockets.connect("ws://127.0.0.1:8600/ws/telemetry") as ws:
        while True:
            sample = json.loads(await ws.recv())
            hot = {k: v for k, v in sample["temperatures"].items() if v and v > 55}
            if hot:
                print("OVER-TEMP:", hot)

asyncio.run(watch())
```

Against a non-local server, pass the token as an `Authorization: Bearer <token>` header on the WebSocket connection.

Shell / CI patterns

```
# Gate a CI stage on enclosure health
hydractl --json system diag | python -c '
import json, sys
results = json.load(sys.stdin)
bad = {device: status for device, status in results.items() if status != "PASS"}
sys.exit(1 if bad else 0)'

# Nightly telemetry snapshot
hydractl --json env telemetry --limit 900 > "telemetry-$(date +%F).json"

# Locate a drive for a technician
hydractl slot identify 5 && echo "Bay 5's locate LED is flashing (60 s)"
```

pytest integration

```
import httpx, pytest

@pytest.fixture(scope="session")
def hydra():
    client = httpx.Client(base_url="http://127.0.0.1:8600/api/v1")
    assert client.get("/connection/status").json()["data"]["connected"]
    return client

def test_all_present_drives_are_healthy(hydra):
    for slot in hydra.get("/slots").json()["data"]:
        if slot["present"]:
            health = hydra.get(f"/nvme-mi/{slot['slot_number']}/health").json()["data"]
            assert health["success"]
```

```
            assert (health["critical_warning"] or 0) == 0
```

Run artifacts: audit log, telemetry files, manifests

Three primitives turn a test session into a reproducible test record:

Event audit log — every state-changing command from any surface (UI click, API call, CLI) is timestamped and recorded, in memory and as NDJSON files under the server's data directory (`HCC_DATA_DIR`, default `./hcc_data`). Use it to correlate enclosure actions with host-side analyzer traces or dmesg (see *chapter 14*):

```
hydractl events --limit 50           # recent commands, colorized
hydractl events --follow             # live tail during a test
hydractl events --category quarch    # only Quarch actions
# API: GET /events?category=slots&since=<epoch>   (cursor-style polling)
```

Telemetry persistence — every sample is appended to `hcc_data/telemetry-YYYYMMDD.ndjson` (disable with `HCC_TELEMETRY_LOG=0`). For quick spreadsheet work, export the in-memory window as CSV:

```
hydractl env export -o thermal-soak.csv
# API: GET /environment/telemetry.csv
```

Run manifest — capture what was installed at run start (see *chapter 15*):

```
hydractl --json inventory > run-042-manifest.json
# API: GET /inventory      UI: Dashboard -> Capture Inventory
```

A typical campaign wrapper:

```
run_id = "run-042"
manifest = call("GET", "/inventory")          # what's installed
# ... execute the test (slot power cycles, Quarch hot swaps, ...) ...
events = call("GET", "/events")               # what was commanded, when
# archive manifest + events + hcc_data/telemetry-*.ndjson with your results
```

Example scripts

A set of ready-to-run lab-automation scripts ships as the **examples zip on the HYDRA product page**. Download it, unzip it anywhere, and run the scripts with Python 3.8 or newer:

- **Stdlib only** — no `pip install`; every script uses a small shared helper (`_common/hydra_client.py`).
- **Server address** — by default the scripts connect to `http://127.0.0.1:8600`. Pass `--host` and/or `--port` to target another server; an explicit `--host` or `--port` always wins. When neither is given, the scripts use the `HCC_URL` environment variable if it is set (for example `https://hydra-lab-01:8600`). `HCC_URL` is the way to reach a server running HTTPS; for a self-signed certificate (the `hcc-setup` default) also set `HCC_CA_CERT` to a copy of the server's certificate, as for `hydractl`.
- **Token** — for a token-protected server, pass `--token` or set `HCC_API_TOKEN` (the same variable `hydractl` reads); `--token` wins when both are given.
- **Try them on the simulator first** — start `hydractl serve --demo`, then run e.g. `python smoke_test/smoke_test.py --demo`. If it prints PASS, every other script will run the same way.

- **CI-style exit codes** — `0` all checks passed, `1` one or more validation failures, `2` setup/connection error (nothing was validated). `--json` emits exactly one JSON document on stdout (progress goes to stderr). A value that couldn't be read is never reported as PASS, and an interrupted run never passes.
- **Safe by default** — scripts that change enclosure state print a dry-run plan and exit `0` unless you pass `--yes`. When they finish or are interrupted they put back what they changed (bay power, Quarch plugs, signal routing and rail setpoints, recordings, watches and jobs) with retries; if a restore fails they say so loudly and exit `1`. Firmware commits, cleared CXL event logs and fio-written data can't be undone, and each script's plan says so. Read each script's README before pointing a destructive script at real hardware.
- **License** — the examples are provided as sample code; you may adapt them for use within your organization.

Script	What it does
`smoke_test`	First script to run: server, connection, bays and environment are alive.
`slot_inventory`	Bay population with model/serial/link and device-library match.
`link_check`	Negotiated PCIe link vs. expected Gen/width; catches downtraining.
`health_snapshot`	NVMe-MI health + SMART threshold audit per bay.
`telemetry_capture`	Live telemetry to CSV plus an event-log watch.
`health_trend`	Long-duration SMART trending and degradation flags.
`validation_gates`	Runs the server-side `/checks` gates; JUnit output.
`cxl_device_check`	CXL CCI identify / health / firmware / events (CXL Type 3 device).
`raw_nvme_mi`	Protocol passthrough: discover / catalog / preview / raw / VDM / I2C.
`self_test_campaign`	Device Self-Test launch and poll across bays.
`smart_delta_stress`	SMART before/after delta across a stress window.
`thermal_soak`	Fan-commanded thermal stress with an abort ceiling.
`hotplug_endurance`	Bay power-cycle endurance loop.
`firmware_update`	Validate → download → commit → verify drive firmware state machine.
`quarch_power_margin`	Rail margining / glitch injection (Quarch module).
`quarch_bay_bringup`	Breaker-paddle pin profile + drive power/health verify (Quarch breaker).
`quarch_stream_capture`	High-rate power + sideband capture to CSV (Quarch module).
`workload_run`	Run a saved workload end to end and exit by its verdict.
`i3c_ccc_probe`	I3C target identification with a per-step health check.
`host_inband_check`	Host-side link / AER / SMART vs. the out-of-band view (host agent).
`host_fio_stream`	Canned fio profile on the host, streamed live to CSV (host agent + fio).
`host_aer_watch`	AER counter + kernel-log watch over a window (host agent).
`host_reset_recovery`	Host-initiated controller / link / bus reset with recovery

Script	What it does
	timing (host agent).

Tips

- **Develop against `--demo`.** The simulator honors the full API, so scripts written against it run unchanged on hardware.
- **One server, many clients.** Never open the serial port from your script; go through the server so the UI keeps working while automation runs.
- **Long operations** (Quarch recordings, discovery sweeps) can exceed default HTTP timeouts — raise your client timeout rather than retrying.
- **Destructive calls need `"confirm": true`.** Slot power-off, resets, raw writes and similar calls return `422` without it — add it deliberately, never by default.
- The OpenAPI schema at `/openapi.json` can generate typed clients (`openapi-python-client`, `openapi-generator`) if you prefer generated SDKs.

12. Vendor Defined Messages (VDM)

> **VDM lives in the unified Protocol Console.** The **VDM** nav entry deep-links into `/console?proto=vdm` — the same console as *NVMe-MI*, with the VDM builder, frame preview, and response viewer. The old `/vdm` URL redirects there.

The **VDM** tab sends MCTP **Vendor Defined Messages** to a drive's management endpoint and shows the decoded response. VDMs are the escape hatch the MCTP standard reserves for vendor-private protocols — drive vendors use them for diagnostics, debug unlocks, telemetry extensions, and manufacturing commands that have no standard equivalent.

Two VDM formats exist (DMTF DSP0236 §13):

Format	MCTP type	Vendor identifier
Vendor Defined — PCI	`0x7E`	2-byte PCI-SIG vendor ID (e.g. Samsung `0x144D`)
Vendor Defined — IANA	`0x7F`	4-byte IANA enterprise number

Everything after the vendor identifier is **vendor-defined** — the Command Center transports and displays it faithfully but cannot interpret it; consult your drive vendor's VDM documentation for payload semantics.

Sending a VDM

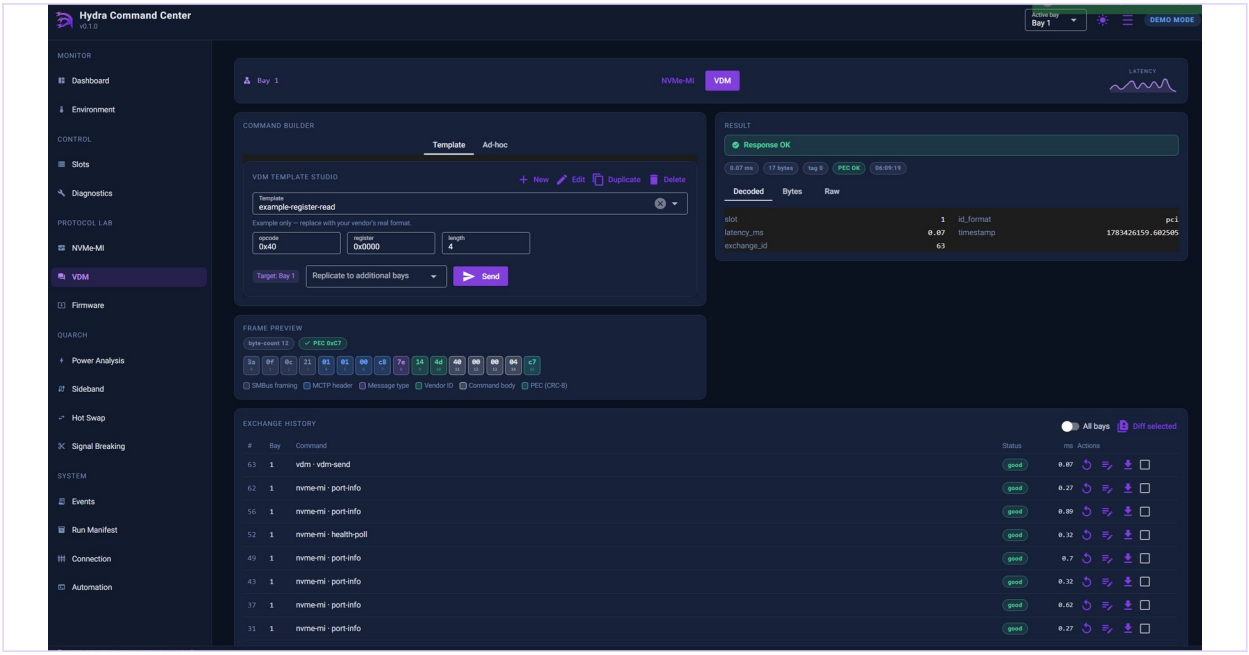


Figure 12.1 Vendor Defined Messages. Build a MCTP frame, send it to the active bay, and read a decoded response.

Open the **Ad-hoc** tab of the Command Builder (the **Template** tab is the Template Studio — see below).

1. Confirm the **Active Bay** in the header (shared across all bay-scoped pages; set with the header selector, **Alt + 1…8**, or an enclosure-panel click).
2. Choose a **Vendor preset** (Samsung, SK hynix, Kioxia, Micron, Solidigm, Broadcom, …) or type any **Vendor ID** in hex. Presets fill the ID field; you can still edit it.
3. Select the **ID format** — PCI (0x7E) is what NVMe drive vendors use almost universally.

4. Enter the **Payload** as space-separated hex bytes (may be empty; some discovery VDMs are header-only).
5. Click **Send VDM**. A confirmation dialog ("Send vendor-defined message?") repeats the bay, vendor ID, ID format and payload, because a vendor-defined message is opaque to HCC and can change drive state. Click **Send VDM** in the dialog to transmit, or cancel.

The **Frame Preview** under the builder shows the exact bytes before you send. The **Result** panel shows a verdict banner from the transport checks, a telemetry strip (latency, byte count, tag, PEC) and **Decoded / Bytes / Raw** tabs — the decode includes the message type, the echoed vendor ID, and the body as hex and ASCII. Every exchange lands in the **History** list below; **Load frame** puts a past request back into the Ad-hoc fields. History is shared with NVMe-MI (*chapter 5*) and exportable via `/console/exchanges`.

> **Demo mode:** the simulator answers every VDM with a status byte `00` followed by an echo of your request body — useful for validating your payload framing before touching hardware.

VDM Template Studio (demonstration feature)

The **Template** tab lets you describe a vendor command once as named, typed fields (and optionally a response map), then fill it in and send it without typing hex — to the Active Bay plus any bays chosen under **Replicate to additional bays**. **Send** asks for confirmation first; the dialog names the template and every target bay.

> **In this release the Template Studio is a demonstration feature.** VDM formats are vendor-proprietary, so the Command Center ships no vendor templates; only the built-in **Load example** template is provided, for use with `--demo`. Use the **Ad-hoc** tab for real drives.

Templates are stored in the HCC data directory. The matching API (`/console/vdm-templates…`) and CLI (`hydractl vdm template …`) serve the same demonstration use; their `send` is confirm-gated.

MCTP details worth knowing

- The vendor ID is transmitted big-endian per DSP0236.
- VDM traffic shares the per-bay MCTP session (and message-tag sequence) with NVMe-MI, so interleaving VDM and NVMe-MI commands is safe.
- Whether a drive answers a VDM at all is vendor policy; an unsupported vendor ID typically times out or returns a vendor-specific error body.

API Reference — VDM

Method	Path	Body / Params	Purpose
GET	`/vdm/vendors`	—	Well-known PCI vendor IDs
POST	`/vdm/send`	`{"slot": 1, "vendor_id": "0x144D", "payload_hex": "01 02", "id_format": "pci", "timeout": 5, "confirm": true}`	Send and decode one VDM exchange — confirm-gated (vendor-defined effects); without `"confirm": true` it returns 422
GET	`/vdm/history`	`?limit=20`	This session's exchanges
GET	`/vdm/{slot}/preview`	`?vendor_id=0x144D&payload_hex=01+02&id_format=pci`	Build (without sending) the annotated VDM frame

```
result = api.post("/vdm/send", json={
    "slot": 3,
    "vendor_id": "0x1E0F",          # Kioxia
    "payload_hex": "de ad be ef",
    "confirm": True,
}).json()["data"]

print(result["response"]["payload_hex"])
print(result["latency_ms"], "ms")
```

CLI Reference — VDM

```
hydractl vdm vendors                              # known PCI vendor IDs
hydractl vdm send 3 0x144D "01 02 aa" -y          # PCI-format VDM to bay 3
hydractl vdm send 3 0x0000ABCD "01" --iana -y     # IANA-format VDM
hydractl vdm preview 3 0x144D "01 02"             # frame byte-map (no send)
hydractl --json vdm history --limit 20            # session exchange log as JSON
```

send asks for confirmation; `--confirm`/`-y` skips the prompt in scripts.

13. Slot Device Firmware Updates

The **Firmware** page updates a drive's firmware **out-of-band** — through the enclosure MCU over MCTP, with no host PCIe link required. Under the hood it tunnels the standard NVMe admin commands (Firmware Image Download `0x11`, Firmware Commit `0x10`) carried inside NVMe-MI as **NVMe Admin Command Messages** — the same admin tunnel the Identify and SMART / Health reads use — streaming the image as a queue of small MCTP messages.

This chapter covers **drive (SSD) firmware**. The enclosure's own MCU firmware is not updated from HCC — follow the instructions published with the firmware image on the HYDRA product page.

HCC requires MCU firmware 0.9.0 or newer. Enclosures shipped with 0.0.8 must be updated before HCC's out-of-band features work. Download the image and its published SHA-256 from the HYDRA product page.

⚠ **Not yet run on live hardware.** The NVMe-MI admin-command transport this feature uses is proven on enterprise SSDs from several vendors, but firmware download and commit over it have **not yet been run on live hardware**. Use an **expendable drive** for your first download and expect to tune chunk size / inter-chunk delay. A download without a commit changes nothing on the drive.

The safety model

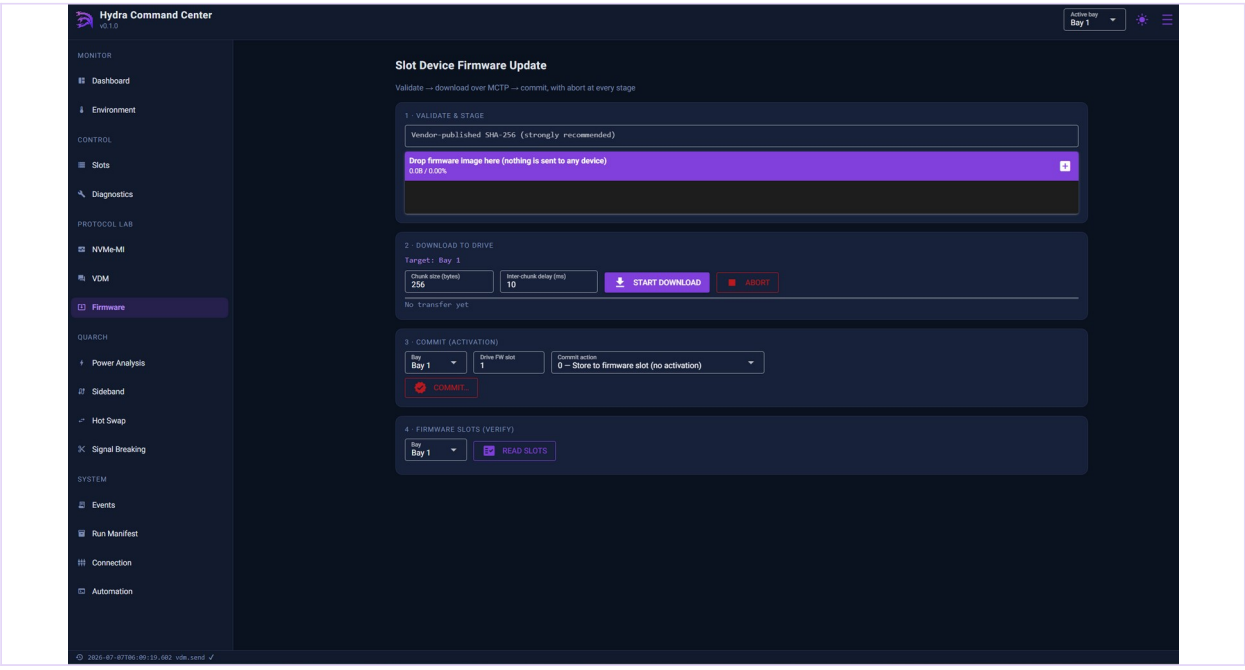


Figure 13.1 Slot Device Firmware. Validate, download, commit, and verify firmware state machine per bay.

Corrupt firmware bricks drives, so the workflow is three separate gates:

1. **Validate & stage** — the image is checked *before anything is sent*: non-empty, under the 64 MiB sanity cap, dword-aligned, SHA-256 computed and (strongly recommended) compared against the vendor-published hash. Only a fully valid image is staged; an invalid one cannot be downloaded.
2. **Download** — first the bay's MCTP transmission unit is brought to the 64-byte baseline: HCC reads it, sets it only if it isn't already 64, and reads it back to confirm (other reads may have negotiated it higher, and a drive may silently drop packets of the wrong size). If 64 can't be confirmed the download is refused before any chunk is sent. The bay is then held at 64 until the transfer ends — nothing renegotiates it, and a Device Self-Test start on that bay is refused meanwhile. Then

chunks go out strictly in order; the MI status of **every chunk** is checked and the first rejection aborts the entire transfer. The staged file is hash-re-verified at read time. Downloading **never activates anything** — the image just lands in the drive's staging area. One transfer at a time per server; **Abort** stops cleanly between chunks.

3. **Commit** — a separate, explicitly confirmed action chooses the drive firmware slot and commit action. The default action (`0`) only stores the image to a slot without activating it; activation actions get an extra red warning in the confirmation dialog. Commit is a single request sent at the bay's current transmission unit (no reset to 64). If it fails at a negotiated unit above 64 the bay is dropped back to 64 and the failure is reported — it is never re-sent automatically; retry it. (After a reset HCC issued itself — bay power, PERST#, SMBus reset, PWRDIS, enclosure power-off/reset — HCC already knows the bay is back at 64, so this only happens after a reset it didn't see.)

Every stage is recorded in the *event audit log*.

Using the Firmware page

0 · Acknowledge

The page opens behind an acknowledgement gate, **Firmware update — read before proceeding**, which summarizes the live-hardware status and the safety model above. Click **I understand — proceed** to reveal the four stages.

1 · Validate & Stage

Paste the vendor-published SHA-256 (if you have it), then drop the firmware file onto the upload zone. The validation report lists each check with PASS/FAIL and the computed SHA-256. Only a green **VALID & STAGED** result unlocks step 2.

2 · Download to Drive

The download targets the header's **Active Bay** (shown on the card). Optionally tune **chunk size** (dword-aligned, 4–952 B; the page defaults to 256 B, while the API and CLI default to the 952 B maximum) and **inter-chunk delay** (default 10 ms — raise it if the transfer stalls or chunks are rejected), then **Start Download**. The progress bar tracks chunks in real time; **Abort** stops the transfer between chunks with nothing committed.

> **Set expectations on duration:** the transfer rides the enclosure's SMBus management link (64-byte MCTP packets), so a multi-megabyte image is thousands of round-trips — plan on **tens of minutes**, not seconds. This is a property of the 2-wire link, not the tool; leave the session undisturbed and watch the progress bar or `GET /firmware/status`.

3 · Commit (Activation)

Select the bay, the **drive firmware slot** (check your drive's slot count via the Firmware Slot log), and the **commit action**:

Action	Effect
0	Store downloaded image to slot (no activation) — default, safest
1	Store to slot **and activate on next reset**
2	Activate the image already in the slot on next reset
3	Store to slot and **activate immediately**

Commit… opens a confirmation dialog; actions 1–3 carry a red warning because they change what firmware the drive runs.

4 · Verify (Firmware Slots)

After a commit, the page **reads the drive's Firmware Slot Information Log back** (LID `03h`) into a **FIRMWARE SLOTS (VERIFY)** card, so you can confirm what actually happened rather than trusting the commit's return code alone. The card shows the **active slot**, the **slot pending activation on next reset**, and the **revision stored in each** of the up-to-seven slots. The card has its own **Bay** selector and a **Read Slots** button, so you can read any bay's slot log at any time, not only after a commit. A drive that reports no active firmware slot shows a red **NO ACTIVE IMAGE** state.

The verdict is honest about **deferred vs. immediate** activation — it never reports a green success for an image that has not started running:

Commit action	Verify verdict
0 (store only)	**STORED — NOT ACTIVATED**
1 / 2 (activate on next reset)	amber **RESET REQUIRED** — still running slot *X* until reset
3 (activate immediately)	green **ACTIVE** with the running revision

The pending (next-reset) slot always renders, even when a slot's revision string is empty, so a staged image is never invisible. This is a **read-back** of the slot log — a read miss reads as "slot read-back failed", never as a commit failure.

Suggested first-hardware procedure

1. Validate with the vendor hash; download with default settings to an expendable drive.
2. Commit with action `0` only; read the **Firmware Slots** log back (the verify card, `hydractl mi fw-slots`, or `GET /nvme-mi/{slot}/firmware-slots`) to confirm the image landed in the intended slot.
3. Commit action 2, PERST the bay (*Slot Control*), and verify the new firmware revision — read the slot log again (the active slot should now be the newly activated one) and cross-check via Identify.
4. Only then consider actions 1/3 for routine use.

API Reference — Firmware

Method	Path	Body	Purpose
POST	`/firmware/validate`	multipart: `file`, optional `expected_sha256`	Validate + stage; returns report with `sha256`
POST	`/firmware/download`	`{"slot": 1, "sha256": "...", "chunk_size": 952, "inter_chunk_delay_ms": 10}`	Start background transfer of a staged image. `chunk_size` 4–952 (dword-aligned), default 952
GET	`/firmware/status`	—	Job progress: state, chunks_sent/total, bytes, error
POST	`/firmware/abort`	—	Stop between chunks; nothing committed
POST	`/firmware/commit`	`{"slot": 1, "firmware_slot": 1, "commit_action": 0, "confirm": true}`	Firmware Commit (confirm required)
GET	`/nvme-mi/{slot}/firmware-slots`	—	Firmware Slot Info (LID 03h): active slot, next-on-reset slot, per-slot revision + 512-byte `raw_hex`

```
# Scripted update with progress
import time
import httpx

api = httpx.Client(base_url="http://127.0.0.1:8600/api/v1", timeout=60)
VENDOR_SHA = "<vendor-published sha256>"

with open("fw.bin", "rb") as f:
    report = api.post("/firmware/validate",
                      files={"file": ("fw.bin", f)},
                      data={"expected_sha256": VENDOR_SHA}).json()["data"]
assert report["valid"], report["checks"]

api.post("/firmware/download", json={"slot": 3, "sha256": report["sha256"]})
while True:
    job = api.get("/firmware/status").json()["data"]
    print(f"{job['chunks_sent']}/{job['chunks_total']}")
    if job["state"] != "downloading":
        break
    time.sleep(1)
assert job["state"] == "downloaded", job["error"]

api.post("/firmware/commit", json={"slot": 3, "firmware_slot": 1,
                                   "commit_action": 0, "confirm": True})
```

CLI Reference — Firmware

```
hydractl fw validate fw.bin --sha256 <vendor-hash>     # check only, send nothing
hydractl fw update 3 fw.bin --sha256 <vendor-hash>     # validate + download w/
progress
hydractl fw update 3 fw.bin --chunk 512 --delay-ms 25  # tuning knobs for bring-up
hydractl fw status                                     # job progress
hydractl fw abort                                      # stop between chunks
hydractl fw commit 3 --fw-slot 1 --action 0 --confirm  # store only (safe)
hydractl fw commit 3 --fw-slot 1 --action 2 --confirm  # activate on next reset
hydractl mi fw-slots 3                                 # read back the slot log (active
+ per-slot revision)
hydractl mi fw-slots 3 --raw bay3-fwslots.bin          # also save the raw 512-byte log
```

14. Event Log

The **Events** page (under **System** in the navigation) is Hydra Command Center's audit trail: every state-changing command — from the UI, the REST API, or the CLI — is recorded with a millisecond timestamp. This is the timeline you align against a host-side PCIe analyzer capture or a `dmesg` log when answering "at T = ..., I pulled bay 3 — what did the link do?"

What gets recorded

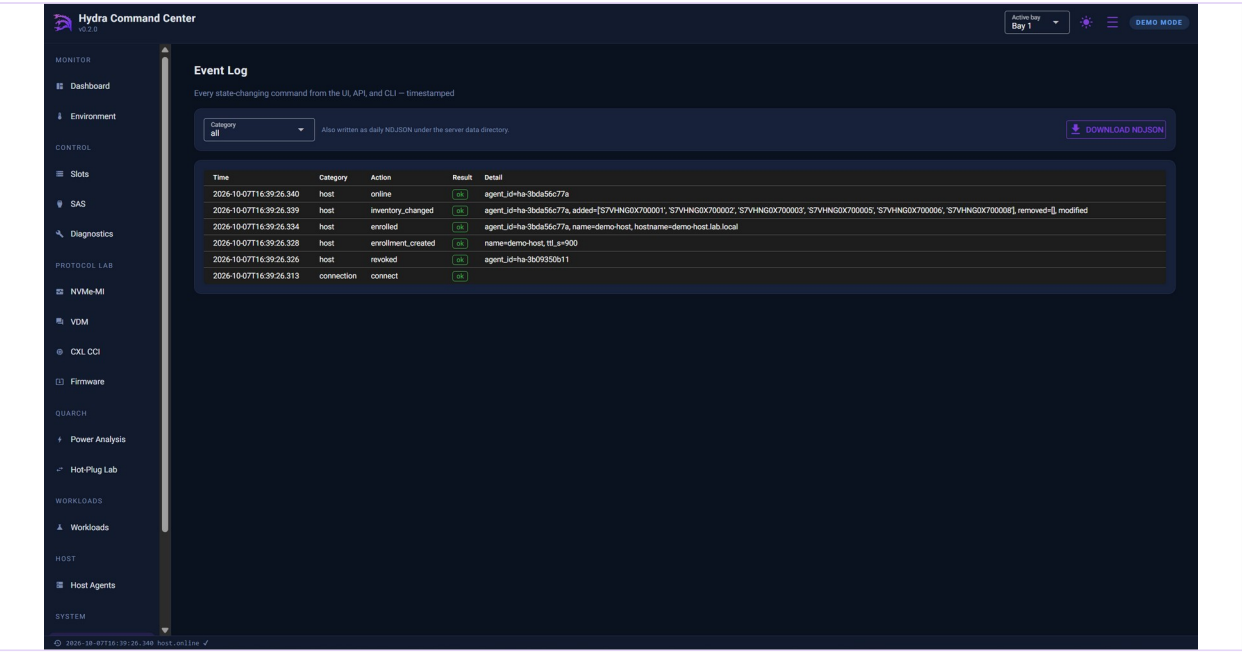


Figure 14.1 Event Log. Every state-changing command from any surface, timestamped and correlated.

Any command that changes enclosure or device state: connect/disconnect, system power/reset, slot power/LED/reset/dual-port/PWRDIS, fan and buzzer, NVMe-MI raw sends, session control and AEM subscribe, VDM sends, firmware validate/download/abort/commit, device self-test start/abort, I2C/I3C writes and CCCs, CXL event-clear and console sends, Quarch actions and recordings, and host-agent and workload actions. Read-only queries are generally not logged.

Each entry has a timestamp, the surface-independent category and action, the parameters, and the result (`ok` or `error` with the message).

Using the page

- The table shows the 400 most recent commands, newest first, refreshing live.
- The **Category** filter narrows to one subsystem: `connection`, `system`, `slots`, `environment`, `nvme_mi`, `vdm`, `firmware`, `diagnostics`, `cxl`, `quarch`, `quarch_recorder`, `host`, or `workload`.
- **Download NDJSON** saves a newline-delimited JSON file (`application/x-ndjson`, one event per line, oldest first) for attaching to a test report. It honours the Category filter: the file is named `hydra-events.ndjson` with **all** categories selected, or `hydra-events-<category>.ndjson` (for example `hydra-events-firmware.ndjson`) when filtered. It includes up to the 5,000 most recent events — the size of the server's in-memory event buffer.

Beyond the in-memory view, the server continuously writes every event to a daily newline-delimited JSON file — `events-YYYYMMDD.ndjson` under the data directory (`HCC_DATA_DIR`, default `./hcc_data`). These files survive restarts and cover the full session, so they are the record of choice for a qualification run.

The footer **activity strip** on every page shows the single most recent action, so you always have quiet confirmation of what just happened without opening this page.

API Reference — Events

Method	Path	Params	Purpose
GET	`/events`	`?limit=`, `?category=`, `?since=`	Audit entries (newest last). `since` is epoch seconds for cursor-style polling.

```
# Tail new events since the last one seen
import time
import httpx

api = httpx.Client(base_url="http://127.0.0.1:8600/api/v1", timeout=30)
cursor = 0.0
while True:
    fresh = api.get("/events", params={"since": cursor}).json()["data"]
    for e in fresh:
        print(e["time"], e["category"], e["action"], e["result"])
    if fresh:
        cursor = fresh[-1]["timestamp"]
    time.sleep(1)
```

CLI Reference — Events

```
hydractl events                    # recent commands
hydractl events --limit 200
hydractl events --category firmware # one subsystem
hydractl events --follow           # live tail (Ctrl-C to stop)
hydractl --json events > run-042-events.json
```

15. Run Manifest

The **Run Manifest** page (under **System**) captures a timestamped snapshot of *what is installed* — the enclosure's firmware and serial number plus each populated bay's drive identity (vendor, model, firmware, serial) and health summary — in a single action. Capture one at the **start of every test campaign** and archive the JSON with your results; it is the reproducibility record that lets you (or a customer) know exactly which drives and firmware produced a given result months later.

Capturing

Click **Capture Now** (or the **Capture Inventory** button on the Dashboard). The sweep queries every populated bay over NVMe-MI, so allow a few seconds on a full enclosure. The latest manifest is shown as:

- **Enclosure** — model, serial number, firmware version.
- **Bays table** — columns **Bay**, **Present**, **Vendor**, **Model**, **FW**, **Capacity**, **Drive SN**, **Temp °C** and **Life %**.

Download JSON saves the manifest. The JSON additionally carries each bay's EDSFF type, power state, and the full health summary, including the critical-warning flags. If you capture more than one during a session, the earlier ones are listed under **Session History** with their timestamp and drive count; retrieve the full snapshots with `GET /inventory/history` to compare a pre-test and post-test manifest.

Per-bay query failures are recorded inside the manifest (e.g. a drive that doesn't answer) rather than aborting the whole capture, so a manifest is always produced.

API Reference — Run Manifest

Method	Path	Params	Purpose
GET	`/inventory`	`?include_health=true`	Capture a manifest now
GET	`/inventory/history`	`?limit=`	Manifests captured this session

```
manifest = api.get("/inventory").json()["data"]
print(manifest["enclosure"]["version"])
for bay in manifest["bays"]:
    if bay["present"]:
        print(bay["bay"], bay.get("serial_number"),
              bay.get("health", {}).get("percentage_used"))
```

CLI Reference — Run Manifest

```
hydractl inventory                       # manifest table
hydractl --json inventory > run-042.json # archive with results
hydractl inventory --no-health           # skip per-drive health queries (faster)
```

16. CXL OOB Management (Technology Preview)

The **CXL** page reads a **CXL Type 3 memory device** the same way the *NVMe-MI* page reads an SSD — **out-of-band**, through the enclosure MCU over MCTP, with no host CXL link or OS driver involved. Where NVMe-MI rides MCTP message type `0x04`, CXL management uses the device's **CCI** (Component Command Interface) carried over MCTP message type `0x08`.

Optional per device. Not every bay exposes an OOB CXL CCI endpoint — a plain NVMe SSD advertises only `0x04`, and even a CXL device may present no out-of-band CCI at all. So the first job on this page is always to **detect** what the bay actually answers before reading anything.

Technology Preview. CXL OOB management is a Technology Preview in this release. Detect, Identify, Health, firmware info, alert thresholds, timestamp and the `cxl-health` / `cxl-traceability` gates have been validated against one CXL Type 3 memory module class. **Event records (`0100h`) and the Command Effects Log (`0401h`) have not yet been decoded from a real device** — the decoders are verified against reference data only, and the `cxl-events` gate verdict carries `provisional: true` with a decode note until they are. Treat event and CEL results as indicative, not authoritative.

Following along in demo mode. In `--demo` the simulated CXL Type 3 device sits in **bay 8**, so the examples in this chapter use bay 8.

Device-class autodetect

Before any capacity or health read, the CXL page probes **which MCTP message types the bay's endpoint answers** — an MCTP *Get Message Type Support* request — and classifies the device:

Class	Meaning
nvme	Only NVMe-MI (`0x04`) — an ordinary SSD.
cxl	Only CXL CCI (`0x08`) — a CXL-only memory device.
both	Both `0x04` and `0x08` are advertised.
unknown	Neither — or the endpoint did not answer (check EID / bay power / wiring; this is **not** a definitive "no CXL").

The FM-API type (`0x07`) is reported separately when present.

Advertise vs. answers — the go/no-go

Detection reports **two distinct facts**, and they are not the same:

- **`cxl_present`** — the endpoint **advertises** CXL CCI (`0x08` appears in the supported message-type list).
- **`cxl_live`** — the CCI **mailbox actually answered** a command. Advertising `0x08` only means a responder *should* be there; that responder is often a separate firmware layer that may be not-ready or timing out. To confirm, detect sends one benign CCI command — **any** well-formed response frame, even an *Unsupported* return code, proves the mailbox parsed and replied.

`cxl_live` is the go/no-go. A bay that advertises CXL CCI but does not answer (`advertised but NOT answering`) cannot be read yet — recheck the EID, bay power, and wiring. Only a live mailbox unlocks Identify and Health.

The detect result is **cached per bay** (per connection generation); it is a static per-endpoint capability, not a live verdict, so it is not re-probed on every call. Force a fresh probe with `--refresh` (CLI) or `?refresh=true` (API) — for example after swapping a drive.

Detect also discovers the endpoint's **profile** and applies it to every later CCI command on that bay: the SMBus address it answers at (`smbus_addr`; `null` = the default) and whether it requires the MCTP integrity check (`ic_required`). A device that silently drops requests without the integrity check is detected and handled automatically.

Identify Memory Device

Identify Memory Device (CCI opcode `4000h`) reports the device's static identity and geometry:

Field	Meaning
FW revision	Firmware revision string (device-controlled text).
Total capacity	Total device capacity (reported in 256 MiB units; shown in GB).
Volatile	Volatile capacity.
Persistent	Persistent capacity.
LSA size	Label Storage Area size, in bytes.
Poison list max	Maximum number of poison-list records the device tracks.

Parsing targets the **CXL 2.0** payload layout and is bounds-checked: a longer 3.0/4.0 payload is tolerated (later revisions only append fields), and a shorter one degrades to partial values rather than failing.

Get Health Info

Get Health Info (CCI opcode `4200h`) reports the device's current condition. The card leads with an **overall health verdict** — HEALTHY or ATTENTION:

> **The verdict is not the Health Status byte alone.** A device can report a clear Health byte while Media Status reads *All Data Lost* — that is never "healthy". The overall verdict is HEALTHY only when the Health byte is clear **and** media status is Normal **and** neither Life Used nor Temperature is at Warning or Critical.

Field	Meaning
Media status	Normal, Not Ready, Write Persistency Lost, All Data Lost, and the persistency-loss variants.
Life used	Life Used % (0–100; `0xFF` = not implemented) plus a Normal / Warning / Critical tri-state.
Temperature	Device temperature in °C (signed) plus its Normal / Warning / Critical tri-state.
Dirty shutdowns	Count of dirty (unsafe) shutdowns.
Corrected errors (vol / pers)	Corrected volatile and persistent error counts.

Three **Health Status** conditions surface as explicit callouts when set: **Maintenance needed**, **Performance degraded**, and **Hardware replacement needed**.

Ad-hoc CCI console

The **CXL CCI** entry under **Protocol Lab** in the sidebar opens the console's CXL tab (`/console?proto=cxl`), the interactive front end: preset picker with typed parameter fields, a live annotated frame preview, and the same history/replay loop as NVMe-MI. A badge warns when the active bay isn't detected as CXL — but never blocks sending, since undetectable devices are exactly what an ad-hoc console is for.

Beyond the structured commands above, the ad-hoc console sends any mailbox command to a bay — typed **presets** for the standard CXL 2.0 command set (each with validated parameters, e.g. `get_log` takes a log UUID + offset/length for CEL walks) plus a **raw opcode** escape hatch for vendor commands (C0h+) with a free-form hex payload.

```
hydractl cxl console-catalog                # every preset + params + gating
hydractl cxl send 8 get_health_info
hydractl cxl send 8 get_event_records -p log_type=2
hydractl cxl send 8 raw -p opcode=0xC0A0 -p payload="01 00" --confirm
```

API: `GET /cxl/console/catalog`, `POST /cxl/{slot}/console/preview` (build + annotate the frame without sending), `POST /cxl/{slot}/console/send`.

Two rules, both enforced server-side:

- **Set-family presets and every raw opcode require `--confirm`** (`"confirm": true` on the API) — they can change device state. This includes the `clear_event_records` preset, which erases a device event log.
- **Detection never blocks the console** — a bay that fails CXL autodetect can still be sent to (that's often exactly when you need ad-hoc access); the UI shows a warning rather than disabling anything.

Responses decode the **CXL Return Code** (never NVMe status codes): `0` success, `0x0001` "Background Command Started" (poll `bg_op_status`), and the full CXL 2.0 code table by name. Frame previews annotate the CCI layout — category nibble, little-endian opcode pair, 21-bit payload length with the bit-23 background flag, and on responses the return code

- vendor extended status.

The `--ic` flag (force integrity check)

Some devices require the MCTP **integrity check** (IC=1, a CRC-32C MIC on the message) and silently drop requests without it. **Detect** normally discovers this requirement and applies it automatically to every later command on the bay (see the endpoint profile under *Device-class autodetect*).

`--ic` (CLI) or `?ic=true` (API) forces IC=1 on a single read as a **manual override** — useful if a device's requirement changes after detect cached its profile. Re-run `hydractl cxl detect N --refresh` first; reach for `--ic` only if reads still time out.

The override is accepted by `identify`, `health`, `fw-info`, `events`, `alerts`, `timestamp` and `send`.

Firmware info

Get FW Info (CCI opcode `0200h`) reports the device's firmware slot state:

Field	Meaning
Slots supported	Number of firmware slots the device implements (2–4).
Active slot	The slot whose firmware is currently running.
Staged slot	The slot armed for next activation — `none` (0) when nothing is staged.
Online activation	Whether the device can activate firmware without a reset.

Field	Meaning
Slot N FW	The firmware revision string held in each slot (the active one is marked).

An **active slot of 0 is flagged invalid** — the device did not report a valid running slot, not "slot zero".

Event records

Get Event Records (CCI opcode `0100h`) reads the device's event log. CXL keeps **four separate logs, one per severity class**, and every record lives in the log that matches its own severity:

Log	Severity
0	Informational
1	Warning
2	Failure
3	Fatal

Select which log to read with `--log 0|1|2|3` (CLI) or `?log=` (API); the default is log 0. Each record decodes:

- **Type** — **General Media**, **DRAM**, or **Memory Module**, derived from the record's event UUID.
- **Severity** — matching the log it came from.
- **Condition flags** — **Hardware Replacement Needed** (the key RMA signal), plus **Permanent**, **Maintenance Needed**, and **Performance Degraded**.
- **Event handle** — the device's handle for the record.
- **Timestamp** — **device-relative** unless a *Set Timestamp* command has been issued to align the device clock to wall time.

The reader **pages automatically** on the device's *More Records* flag, so a full log is returned completely in one call; the number of pages fetched is surfaced as `pages_read` (up to 64 pages; `paging_capped: true` if the cap is reached). If the device's log **overflowed**, that is reported as `overflow: true` with `overflow_error_count`.

Clearing event records

Clear Event Records (CCI opcode `0101h`) empties the whole selected log.

> ⚠ **Destructive.** This **erases the device's event evidence**. It requires an explicit confirmation — `--confirm` on the CLI (otherwise `hydractl` prompts) and `confirm=true` on the API (the endpoint refuses without it).

> **Capture before you clear.** Run `hydractl cxl events N --log T` and **save the output first** — those records are the FA / RMA evidence, and the device copy is destroyed by the clear. There is no undo.

Command support (CEL)

The device can be asked **which CCI commands it actually implements** — read from its **Command Effects Log** (Get Supported Logs, then Get Log). This is the CXL analog of the NVMe *command support probe*, and it reports each known command as **supported** or **unsupported**:

- The **Bay Detail** CXL card shows the count as a **Commands (CEL)** row reading `N/M supported` (the row is omitted when the probe fails).
- `hydractl cxl commands N` lists each command and its support.
- The `cxl-health` gate warns if the mandatory *Get Health Info* command is absent from the CEL.

CEL support is **advisory** — nothing is disabled or blocked because of it. If the device cannot return its CEL, support is reported as **unknown**.

A successful probe is **cached per bay** (per connection generation); a failed probe is retried on the next call. Force a fresh read with `--refresh` (CLI) or `?refresh=true` (API).

Alert thresholds

Get Alert Configuration (CCI opcode `4201h`) reads the device's **own** critical and programmable-warning thresholds. This is **read-only** — the tool never writes them.

Field	Meaning
Life used critical / warning	Life-used percentage thresholds.
Over-temp critical / warning	Over-temperature thresholds (signed °C).
Under-temp critical / warning	Under-temperature thresholds (signed °C).
Corrected volatile warn	Corrected-volatile-error warning threshold.
Corrected persistent warn	Corrected-persistent-error warning threshold.

Each **warning** threshold carries a **validity flag**; a threshold the device marks invalid is not shown as a real value. These thresholds are most useful as **context** read alongside the live *Get Health Info* values — they tell you the levels at which the device *itself* will raise a warning.

Device timestamp

Get Timestamp (CCI opcode `0300h`) reads the device's running timestamp counter, in **nanoseconds**. Its main use is **correlating event-record timestamps** to a common reference. As with the event records, the **epoch is device-relative** unless a *Set Timestamp* command has aligned the device clock to wall time.

Weak CXL identity (resolve)

`cxl resolve` folds a CXL bay into the *device library* — but **honestly weak**, and deliberately so:

- It is **bay-keyed** (`CXL-BAY-N`). The CCI path exposes no serial number.
- When the module's **VPD EEPROM** is readable, its vendor, model, part number and VPD serial are recorded as **descriptive** identity — but never used as the key, because it is not yet confirmed that the EEPROM travels with the module rather than with the bay's carrier.
- Its confidence is **weak and never upgrades**.
- A later read that disagrees with the first record (VPD serial, model, capacity or firmware) sets **`bay_content_changed`**, which stays set.
- A swap to an **identical** module with no readable VPD cannot be detected.

It only resolves when the CCI mailbox is **actually answering**: a bay that is not CXL returns *not resolved* (`not_a_cxl_bay`), and a bay that advertises CXL CCI but whose mailbox is silent returns *not resolved* (`cci_not_answering`) — never a hollow all-empty entry.

> **Not a substitute for serial traceability.** Use this only to note "a CXL module was seen in this bay"; do not rely on it across a swap or a restart. For durable FA / RMA traceability there is no OOB path on CXL today.

Using the CXL card

On each bay's **Bay Detail** page (click a bay on the Dashboard), the **CXL MEMORY DEVICE** card **auto-reads on page load** — no click needed. For a CXL bay it runs the whole sequence automatically:

1. **Autodetect** the bay's device class and CCI presence (using the cached device-class probe, so it is cheap).
2. If the bay is not a CXL device, it shows a one-line "not a CXL device" note and stops.
3. If CXL CCI is advertised but the mailbox is not answering, it warns and stops (check EID / power / wiring).
4. Otherwise it reads **Health**, **Identify**, the **firmware summary**, the **alert thresholds**, a **CEL command-support count** (how many known CCI commands the device implements), and **all four event logs** (Informational / Warning / Failure / Fatal) live and renders them, with the HEALTHY / ATTENTION verdict on top.

A **Refresh** button forces a fresh re-read (it re-runs the device-class detect too). If the enclosure is not connected, the card shows **"Not connected..."** instead. The card is only meaningful on a CXL bay; on a plain NVMe bay it just shows the one-line class note.

Validation gates

A CXL bay is qualified from the **same** place as an NVMe drive — the **Validation Checks** panel on the Bay Detail page, `hydractl check`, and the `/checks/{slot}` API (see *Diagnostics → Validation checks*). The difference is automatic: when the bay is a CXL device, **Run Checks selects the CXL gate set** instead of the NVMe SMART / link / self-test gates (those do not apply and would only read *incomplete*). The rollup result carries the bay's **`device_class`**.

Three CXL gates run:

cxl-health

Gates the **live** *Get Health Info* read:

- **BAD** on a media **data-loss** status (Write Persistency Lost, All Data Lost, or any of the persistency-loss variants — `0x02`–`0x07`), **hardware-replacement-needed**, **life-used Critical**, or **temperature Critical**. An optional `temp_max_c` target (like the NVMe thermal gate) also fails the gate when the current temperature exceeds that ceiling.
- **WARN** on **maintenance-needed**, **performance-degraded**, **life/temp Warning**, the device's own **corrected-error warning flags**, or if the mandatory Get Health command is absent from the CEL.
- **INCOMPLETE** when health is unreadable — and, importantly, when **Media = Not Ready** (`0x01`). That is the **transient cold-boot / media-init state**, not a defect, so it reads as incomplete (re-run once ready) and is **never a failure**.

Dirty-shutdown count and the raw corrected-error counts are **context** (shown alongside the verdict), not gated point-in-time — they are lifetime/delta signals, so a baseline-delta gate is the follow-on, exactly as with the NVMe unsafe-shutdown signal.

cxl-events

Reads the **Failure** (log 2) and **Fatal** (log 3) event logs and fails (**BAD**) if either contains **any records** — the CXL analog of the NVMe error -log gate. A clean Health Status byte can coexist with logged failure/fatal events, so **health alone is not a pass**. Both logs empty is **GOOD**; unreadable logs are **incomplete**.

Until event decode is validated on a real device, every GOOD or BAD `cxl-events` verdict carries `provisional: true` and a decode note.

cxl-traceability

Informational only. A CXL Type 3 device has no OOB serial to gate identity against, so this reports the weak, bay-keyed identity — firmware revision, the VPD serial when readable (descriptive only), and the *bay-content-changed* note — as context, never pass/fail.

> A **both**-class endpoint (rare — advertises NVMe-MI *and* CXL CCI) runs **both** gate sets. The **PASSED / REVIEW / FAILED / INCOMPLETE** rollup is identical to the NVMe path, and an unread or unresolved bay is never a green PASS.

Auto-labeled bay tiles

Once a bay's device class is known, it is **labeled automatically** — no manual step. The enclosure front-panel tiles on the *Dashboard* and the *Slots page*, and the **Bay Detail** header, each show a small class badge:

Badge	Bay
NVMe	An ordinary NVMe SSD.
CXL	A CXL Type 3 memory device.
CXL+NVMe	An endpoint advertising both.

The badge is driven by the same cached device-class *detect* as everything else on this page, so a **mixed NVMe / CXL population is labeled at a glance** with nothing to configure.

Caveats

- **Weak OOB identity.** CCI exposes no serial; the module VPD serial (when readable) is recorded descriptively only, so CXL identity stays bay-keyed — not durable traceability across a swap or a restart.
- **Technology Preview scope.** Validated on one CXL Type 3 module class; other devices may differ in timing or optional-command support. Report anomalies to Serial Cables support.
- **Event decode is provisional and partial.** Each record shows type, severity, handle and condition flags; the event-UUID byte order and the event-specific fields (physical address, channel / rank) are not yet decoded or confirmed against a real device.

API Reference — CXL

Method	Path	Purpose
GET	`/cxl/{slot}/detect`	Device-class autodetect + OOB-CCI presence. `?refresh=true` bypasses the cached probe
GET	`/cxl/{slot}/identify`	Identify Memory Device (4000h). `?ic=true` forces the MCTP integrity check
GET	`/cxl/{slot}/health`	Get Health Info (4200h). `?ic=true` forces the MCTP integrity check
GET	`/cxl/{slot}/firmware`	Get FW Info (0200h): slot count, active/staged slot, per-slot revisions. `?ic=true` supported
GET	`/cxl/{slot}/events?log=0`	Get Event Records (0100h) for one severity log (`log` 0–3). `?ic=true` supported
POST	`/cxl/{slot}/events/clear?`	Clear Event Records (0101h) —

Method	Path	Purpose
	`confirm=true&log=0`	**destructive**, refuses without `confirm=true`
GET	`/cxl/{slot}/commands`	Command Effects Log capability probe: per-command supported/unsupported. `?refresh=true` bypasses the cache
GET	`/cxl/{slot}/alerts`	Get Alert Configuration (4201h): read-only critical/warning thresholds. `?ic=true` supported
GET	`/cxl/{slot}/timestamp`	Get Timestamp (0300h): device timestamp counter in ns. `?ic=true` supported
GET	`/cxl/{slot}/resolve`	Weak, bay-keyed CXL library resolve (no OOB serial); `resolved:false` when not CXL or the mailbox is silent

The **CXL validation gates** (`cxl-health`, `cxl-events`, `cxl-traceability`) are not separate endpoints — they run through the shared `/checks/{slot}` routes documented in *Diagnostics → Validation checks*, and are auto-selected when the bay's `device_class` is CXL. Pass a `temp_max_c` ceiling via the POST body's `targets` (`{"cxl-health": {"temp_max_c": 70}}`).

`detect` returns `device_class`, `message_types`, `message_type_names`, `cxl_present`, `cxl_live`, `nvme`, `fm_api`, `smbus_addr` and `ic_required`. An inconclusive probe returns `{"success": false, "error": ...}` — read that as "could not confirm", not as a definitive "no CXL".

```
det = api.get("/cxl/8/detect").json()["data"]
if det.get("cxl_present") and det.get("cxl_live"):
    ident = api.get("/cxl/8/identify").json()["data"]
    health = api.get("/cxl/8/health").json()["data"]
    print(ident["fw_revision"], health["media_status_name"],
          "HEALTHY" if health["overall_healthy"] else "ATTENTION")
elif det.get("cxl_present"):
    print("advertises CXL CCI but the mailbox is not answering")
else:
    print("not a CXL device:", det.get("device_class"))
```

CLI Reference — CXL

```
hydractl cxl detect 8               # device class + OOB CCI presence (advertise vs
answers)
hydractl cxl detect 8 --refresh     # bypass the cached probe (e.g. after a drive swap)
hydractl cxl identify 8             # Identify Memory Device (capacities, FW, LSA,
poison list)
hydractl cxl health 8               # Get Health Info (verdict, media, life, temp,
errors)
hydractl cxl identify 8 --ic        # manual IC=1 override (re-run detect --refresh
first)
hydractl cxl health 8 --ic          # manual IC=1 override on the health read
hydractl cxl fw-info 8              # Get FW Info (slots, active/staged, per-slot
revisions)
hydractl cxl events 8              # Get Event Records — Informational log (default)
hydractl cxl events 8 --log 2      # read the Failure log (0 Info, 1 Warn, 2 Fail, 3
Fatal)
hydractl cxl clear-events 8 --log 2 --confirm   # DESTRUCTIVE — clear the Failure log
(capture it first)
hydractl cxl commands 8            # CEL capability probe — which CCI commands the
device implements
```

```
hydractl cxl commands 8 --refresh  # bypass the cached CEL probe
hydractl cxl alerts 8              # Get Alert Configuration (read-only threshold
context)
hydractl cxl timestamp 8           # Get Timestamp (ns) — for event-record correlation
hydractl cxl resolve 8             # weak, bay-keyed CXL library entry (no OOB serial)
```

Add the global `--json` flag for scripting: `hydractl --json cxl health 8`.

To run the CXL **validation gates**, use the shared check command — on a CXL bay it auto-selects the CXL gate set:

```
hydractl check 8                   # runs cxl-health / cxl-events / cxl-traceability on
a CXL bay
hydractl check 8 cxl-health        # a single CXL gate
```

17. Host Agent (in-band)

Everything else in this manual reads the enclosure **out-of-band**: the MCU, over USB, talks to the drives' sideband. The **Host Agent** adds the other half of the picture — what the **Linux host that owns the PCIe links** sees: `/dev/nvmeX`, the trained link, the kernel's AER counters, fio throughput, the admin queue, and how long a hot-plugged drive takes to come back. HCC joins the two views by drive **serial number**: the bay's OOB identity on one side, the host's in-band device on the other.

The agent is a single-file, dependency-free Python program (`hcc-agent.pyz`, Python 3.8+ standard library only) run by systemd on the host. It **dials out** to HCC — nothing listens on the host — and it only ever runs **typed jobs** with a fixed command line; there is no way to send it a shell command. Its token authorises only its own routes on HCC: never slot power, resets, I2C or firmware. To set it up, see *Installing and enrolling the agent* below.

Installing and enrolling the agent

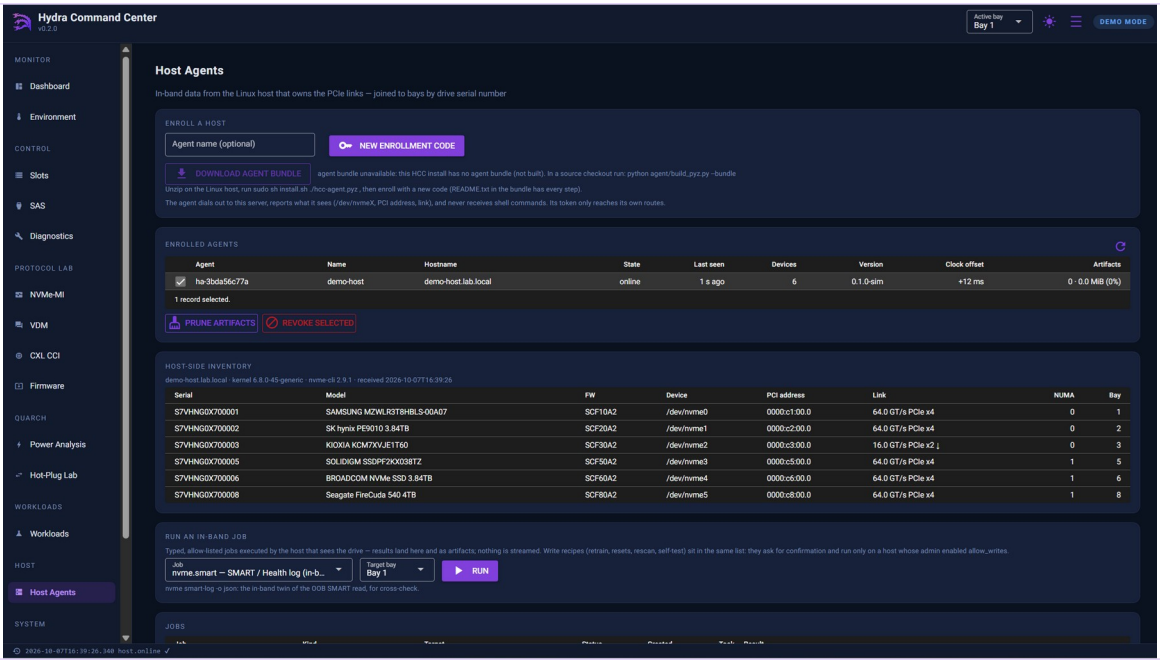


Figure 17.1 Host Agent. In-band host-side view joined to the OOB bay identity by drive serial number.

HCC serves the agent bundle itself; there is nothing to install with `pip`. The Linux host needs `python3` 3.8 or later and root access.

1. Get the bundle and install it

Download the bundle from HCC in any of three ways:

- **UI: Host Agents → Enroll a host → Download agent bundle**. The caption beside the button shows the agent version (`agent v<version>`). If this HCC installation has no agent bundle, the button is disabled and the caption gives the reason (`agent bundle unavailable: …`).
- **CLI:** `hydractl host agent-bundle` writes `hcc-agent-bundle-<version>.zip` to the current directory (add `-o FILE` to choose the output file). It checks the download against the server's SHA-256 and refuses to write a mismatched file. With `hydractl --json host agent-bundle` it prints the path, agent version, size, SHA-256 and file list as JSON.
- **API:** `GET /api/v1/hosts/agent/bundle` returns the zip. `GET /api/v1/hosts/agent/bundle/info` returns, without downloading, `agent_version`, `filename`, `bytes` and `sha256` of the zip, and `files[]` (each member's `name`, `bytes` and `sha256`). Both answer HTTP **503** when no bundle is available.

The download is `hcc-agent-bundle-<version>.zip`, containing:

File	Purpose
`hcc-agent.pyz`	The agent: one file, Python 3.8+ standard library only, licence included
`install.sh`	Installer (Linux)
`install-agent.cmd`, `install-agent.ps1`	Installer (Windows; see *Windows hosts*)
`hcc-agent.service`	The systemd unit
`hcc-agent-writes.conf`	Optional systemd drop-in for write recipes (see below)
`LICENSE`	Licence terms
`README.txt`	Short install notes

> **Same machine as HCC?** If HCC was installed with the Linux install kit on this host, skip this whole section: `sudo hcc-setup --with-agent` installs, enrolls and starts the agent in one step (*chapter 0*). With the Windows install kit, `install-hcc.cmd --with-agent` does the same.

Copy the zip to the Linux host and unzip it (`unzip`, or `python3 -m zipfile -e hcc-agent-bundle-<version>.zip hcc-agent-bundle`).

2. Create an enrollment code in HCC

On **Host Agents** → **Enroll a host**, optionally type a name for the agent and click **New enrollment code**. Under the code, the page shows the complete install command for the host, with this server's address, the code and (on HTTPS) the server's certificate fingerprint already filled in. From the CLI: `hydractl host enroll --name NAME`.

The code is **single-use** and **expires in 15 minutes**.

3. Install, enroll and start: one command

In the unzipped folder on the host, run the command the page showed:

```
sudo sh install.sh --url https://<hcc-host>:8600 --code <code> --pin <sha256>
```

It installs the agent at `/usr/local/bin/hcc-agent` with the `hcc-agent` systemd unit, enrolls the host (the code goes over stdin, never onto the command line of a running process), and starts the service. It also notes any missing tool the jobs use (`nvme-cli`, `fio`, `pciutils`). Running it through `sh` works even if your unzip tool dropped the execute permission.

The agent stores its token in `/etc/hcc-agent/config.json` (mode `0600`).

Certificate options (pass one):

- `--pin <sha256>` — for HCC's self-signed certificate (the Linux install kit's default): the agent **pins exactly that certificate** and refuses to enroll if the server presents any other. The fingerprint is on the Host Agents page and in `sudo hcc-setup status`.
- `--tls-ca ca.pem` — trust a private CA that signed the HCC certificate.
- `--insecure` — pin whatever certificate the server presents on first use (and check it on every later connection); compare the fingerprint it logs with the server's.
- Plain `http://` URLs work, but the agent token then crosses the network in clear text.

Also: `--allow-writes` (see below) and `--name NAME`. To upgrade, re-run `sudo sh install.sh` from a newer bundle with no options; an enrolled agent is restarted automatically. The two-step form also still works: `sudo sh install.sh`, then `sudo hcc-agent enroll --url … --code …` and `sudo systemctl enable --now hcc-agent`.

The agent appears under **Enrolled agents** and shows as online within a few seconds of the service starting; it then sends a heartbeat every 10 seconds.

Windows hosts

Steps 1 and 2 are the same. Then, in a Command Prompt in the unzipped folder, run the Windows command the page shows under the Linux one:

```
install-agent.cmd --url https://<hcc-host>:8600 --code <code> --pin <sha256>
```

Unzip the bundle to a local folder first: run from a network share, the script the elevated prompt runs could be swapped on the way. It asks for administrator rights once (UAC), then:

- uses an all-users Python 3.8+ under `C:\Program Files` (or installs Python 3.12 from python.org after checking its pinned SHA-256 and its signature). A per-user or Store Python is never used, because the agent runs as SYSTEM;
- installs the agent in `C:\Program Files\SerialCables\HCC-Agent`;
- enrolls the host (the code goes over stdin), keeping the token in `C:\ProgramData\SerialCables\HCC-Agent\config.json`. That folder is readable by SYSTEM and Administrators only. If it already exists and was not created by an administrator, the installer refuses to use it;
- starts the scheduled task `\SerialCables\HCC Host Agent`: SYSTEM, at boot, restarted within 5 minutes if it stops. Its log is `C:\ProgramData\SerialCables\HCC-Agent\logs\hcc-agent.log`.

The certificate options and `--name` are the same as on Linux (a `--tls-ca` file is copied into the locked folder). `--allow-writes` enables the two write recipes Windows has: device self-test and namespace rescan. To upgrade, run `install-agent.cmd` from a newer bundle with no options. To remove the agent (add `--purge` to delete the enrollment as well):

```
"C:\Program Files\SerialCables\HCC-Agent\install-agent.cmd" --uninstall
```

What a Windows host can do

The Windows agent talks to drives through the inbox `stornvme` driver (no vendor driver, no nvme-cli). It reports the same shapes as a Linux host, so gates, Workloads and the Hosts page work unchanged — for what it offers:

Capability	Windows	Notes
Inventory (serial, model, firmware, disk, BDF)	✅	serial read with Identify Controller
Link speed/width, max payload	✅	from Windows device properties, which can stay at the old value after a link downtrains (see below)
Identify, SMART, error log, firmware slots, self-test log	✅	
Get Features	FID 02h and 04h only	stornvme refuses 07h, 0Ch and others. On 04h, the composite's over-temperature threshold only: `sensor` and `threshold` are Linux only
fio workloads	✅ when fio is installed under `C:\Program Files\fio`	the agent never takes fio from PATH; a write needs `confirm=true`, an NVMe disk with the targeted serial, and no partitions
Telemetry log (host / controller)	✅	raw NVMe log page
Telemetry regenerate (host-new)	—	the inbox driver doesn't pass the field that asks the drive to build a new log, so the job fails with that reason and

Capability	Windows	Notes
		nothing changes. Tested on Windows 10 19045
Kernel log watch + `host.dmesg_capture`	✅ from the Windows Event Log	System log storage providers (stornvme, disk, partmgr, volmgr, WHEA-Logger) and Kernel-PnP device events for PCI devices, one line each: `[EventLog] <provider> <id>: <message>`
AER watch + `pci.aer_snapshot`	✅ as **WHEA events**	Windows has no AER counter API: the agent counts WHEA-Logger PCIe error events for the drive (and its upstream port) since boot. Shown as "WHEA events" in the run log and on the Hosts page; non-fatal errors may appear as correctable or fatal, depending on what WHEA records
Block I/O watch	✅	Windows disk performance counters
SMART health watch	✅	the same log read as the SMART job, through the inbox driver
Hot-plug enumeration timing	✅ plug to device	the "link" step is meant to come from the controller's PnP *device started* event (Linux: the driver attach line); it has not yet been seen on real hardware, so expect plug-to-device timing only
Device self-test start	✅ with `allow_writes`, once per agent start	through the inbox driver's protocol command (Windows 10 1903 or later). A later start fails with `ReturnStatus 0x8 (throttled request)` until the agent restarts
Device self-test abort	—	fails with the same status. A short test runs to completion, and an extended one may take hours, so start one deliberately
Rescan namespaces	✅ with `allow_writes`	re-enumerates the controller's PnP node; refused while any of its disks has partitions
Link retrain, secondary bus reset, controller reset	—	Linux only (needs config-space writes)
Admin-queue trace, `nvme.subsys`, `pci.config_dump`, `cxl.list`	—	Linux only

HCC refuses a job a host can't run before it is sent, naming the missing capability; a Workloads definition that needs one fails validation with `CAPABILITY_MISSING`.

What to expect on Windows

Performance. At large block sizes Windows matched Linux on the same Gen6 ×4 drive: 19.3 and 19.5 GB/s at 128k. With one fio job, small-block IOPS reached about half of Linux at high queue depth: about 300k against 600-665k at QD32 and QD128. Windows was level or ahead at low queue depth. The figures come from `nvme2x-perf-profile`; compare your own drives the same way.

A downtrained link. The agent's link reading comes from Windows device properties and can keep showing the old speed after a downtrain, so the host-side link gate can't see it. The drive's own sideband report does, and that is what the link gate reads. A sudden fall in throughput with a clean error log is the other sign.

Drive missing after boot. A drive that links late can miss the host's boot-time scan, on Linux as well as Windows. HCC shows it healthy over sideband with no host device. A hot-plug of the bay brings it back.

Dual-boot clocks. Linux stores UTC in the hardware clock and Windows reads it as local time, so after a Linux boot the Windows clock, and Event Log times, can be hours off until Windows resyncs. To stop that, make Windows treat the hardware clock as UTC.

Hot-plug and a stuck agent. A hot-plug during an inventory scan can leave a drive call blocked inside Windows. The agent keeps sending heartbeats and logs a warning, its inventory stops updating, and only a reboot clears the blocked call.

Allowing write recipes (optional)

By default the agent **refuses every write recipe** (link retrain, bus reset, controller reset, rescan, device self-test). To opt a host in:

- **At enrollment:** add `--allow-writes` to the `install.sh` command (or to `hcc-setup --with-agent`). It enrolls with writes allowed and installs the systemd drop-in in the same step.
- **On an enrolled host:**

1. Set `"allow_writes": true` in `/etc/hcc-agent/config.json`.
2. Install the systemd drop-in that lets the service write to sysfs, then restart:

``bash  sudo install -D -m 0644 hcc-agent-writes.conf /etc/systemd/system/hcc-agent.service.d/writes.conf  sudo systemctl daemon-reload && sudo systemctl restart hcc-agent  ``

To take it back, remove both — set `allow_writes` to `false` (or delete it) and delete `/etc/systemd/system/hcc-agent.service.d/writes.conf` — then run the same `daemon-reload` and `restart`.

Host Agents page

The page has six cards.

Enroll a host — an optional agent name and **New enrollment code**, which mints a single-use code valid for 15 minutes and shows the one-line install command to run on the host (server address, code and, on HTTPS, `--pin` with this server's certificate fingerprint filled in). **Download agent bundle** (the agent version is shown beside it; disabled, with the reason, when no bundle is available) downloads the bundle described in *Installing and enrolling the agent*.

Enrolled agents — every enrolled agent with its state (online when it has heart-beaten inside the liveness window), last-seen age, device count, version, clock offset against HCC, and its **artifact usage** against the per-agent quota. *Revoke selected* removes an agent (its token stops working at once and its watches stop); *Prune artifacts* runs retention and quota eviction now.

Host-side inventory — what the selected agent's host sees, one row per NVMe controller: serial, model, firmware, `/dev/nvmeX`, PCI address, link (current/max), NUMA node, and the bay when the serial matches an OOB binding. A mismatch between the OOB and in-band link is the first thing to look for after a hot-plug or a Quarch signal break.

Run an in-band job — the typed job catalogue. Pick a kind and a bay (or a host, for host-level kinds), set the kind's parameters, and *Run*. Jobs are queued for the agent, which long-polls for them; results land in the Jobs table within a second or two and any artifact (nvme-cli JSON, a config-space dump, a telemetry blob, a kernel log) is downloadable from the job dialog. **Write recipes** sit in the same list, marked in the catalogue: they ask for confirmation, run only on a host whose admin enabled `allow_writes`, and are refused by the host itself when a namespace is mounted or held or when a reset would hit other functions under the same port. When more than one host is online a **Host** picker appears; HCC never chooses a host silently for a drive two hosts can see.

Jobs — recent jobs, newest first, with a one-line result per kind (link and AER for a config dump, temperature and wear for SMART, recovery time and AER delta for a reset). Click a row for the full summary and artifact.

Watches — opt-in monitors the agent keeps running until stopped: AER counters on change and every 30 s when quiet, a filtered kernel log, block I/O rates, SMART health every 30 s, and the NVMe admin-queue tracepoints. A watch keeps running through a reboot of the host until you stop it, and a workload uses one that's already running. Raw events stay in each watch's own ring; only **transitions** (a counter that moved, a link change, a reset line, a failed admin command) reach the Events timeline. A watch the host cannot run (no journalctl, read-only tracefs) shows as `failed` with the reason.

Bay Detail → Host view (in-band)

Each bay's **Bay Detail** page has a **Host view (in-band)** card showing what the host sees for the drive in that bay, joined by serial number: the host and its state, the device node, namespaces, PCI address, link and link capability, driver / state, NUMA node, and when the inventory was received. Quick-read buttons (**SMART**, **Identify**, **PCI config**, **AER**) run the matching in-band job and show the result inline. When no agent is enrolled, the card links to **Host Agents**; when the bay has no OOB serial yet, it asks you to resolve the bay first.

Job catalogue

Kind	Target	Needs	What it does
`nvme.identify`, `nvme.smart`, `nvme.error_log`, `nvme.fw_log`, `nvme.get_feature`, `nvme.subsys`	bay	nvme-cli	In-band reads shipped raw and decoded by HCC; `nvme.smart` is normalised to the OOB key names so the gates compare like with like
`pci.config_dump`	bay	sysfs	Config space from sysfs, decoded by HCC: link, AER, DPC, ACS
`pci.aer_snapshot`	bay	sysfs	Kernel AER counters for the device and its root port
`fio.profile`	bay	fio	A canned workload streamed live at 1 Hz to the Workloads Monitor (chapter 18) and the Quarch scope; write profiles need confirm and a blank namespace
`pci.link_retrain`, `pci.secondary_bus_reset`, `nvme.reset_controller`, `nvme.rescan`	bay	**host opt-in**	Write recipes: confirm + `allow_writes`; link / AER / controller state before and after, recovery timed to link-up and `live`
`nvme.self_test`	bay	**host opt-in** + nvme-cli	Start (short / extended) or abort a device self-test; poll `nvme.self_test_log`
`nvme.self_test_log`	bay	nvme-cli	Current operation, completion %, results — same shape as the OOB self-test status
`nvme.telemetry_log`	bay	nvme-cli	Binary telemetry artifact; the stored host-initiated log by default, `host-new` regenerates (confirm), data area 1 by default, sized from the header before a larger pull, ≤ 8 MiB
`host.dmesg_capture`	host	journalctl	Kernel log of this boot, storage-filtered, per-class

Kind	Target	Needs	What it does
			counts, newest 4 MiB
`cxl.list`	host	cxl-cli	The CXL memdevs the host enumerated: memdev, serial, host, sizes, NUMA node (opportunistic)

Agreement gates

When a bay's drive is visible from a host, `hydractl check` and the **Validation Checks** card on Bay Detail (*chapter 6*) add three **host gates** — `host-identity`, `host-link`, `host-smart` — that compare the OOB report with the in-band one. A link the enclosure says is x4 Gen5 and the host trained at x2 is the classic finding. `host-link` also reports the Max_Payload_Size and Max_Read_Request_Size the host configured on the drive ("· MPS 256 of 1024 B, MRRS 256 B"); it informs, it does not judge.

> **When a Gen6 drive reads far below its link.** Two host settings cap bandwidth long before PCIe does, and neither shows in the link speed:
>
> - **The IOMMU.** With DMA translation on (`ls /sys/class/iommu` lists `dmar*` devices and `/proc/cmdline` has no `iommu=pt`), a single drive on a client CPU can top out near 12 GB/s. On the HYDRA bench a Gen6 ×4 drive behind a Gen5 ×8 host link read at 12.1 GB/s with translation on and 20.5 GB/s (22.8 GB/s with four fio jobs) after booting with `iommu=pt`. - **The payload size.** A BIOS that leaves every hop at a 128-byte Max_Payload_Size is common; `pci=pcie_bus_perf` lets Linux raise it to what the whole path supports. On that bench it was not the limit, but the host-link gate shows it so you can rule it out.
>
> Both are kernel command-line parameters (`/etc/default/grub`, `GRUB_CMDLINE_LINUX_DEFAULT`, then `sudo update-grub` and a reboot); weigh `iommu=pt` against your platform's DMA-isolation needs.

Hot-plug enumeration timing

With a `kernel` watch running, a bay power-on or a Quarch plug is joined to the host's `pciehp: Link Up` line and to the moment the device appears in inventory. The result is a `host.enumeration` event, an **ENUM** readout on the Hot-Plug Lab strip, and `hydractl host enumerations`. Plugging a bay by hand starts no timing: HCC has no plug time to measure from.

> **Timestamps.** Stream samples (`/hosts/streams/{id}`) and watch events (`/hosts/watches/{id}/events`) carry HCC time in `t` — the agent's stamp minus the clock offset the heartbeat measured — with the agent's own stamp kept in `t_src`. A sample the agent did not stamp is server-stamped and marked `t_source: server`. Every live source is also published as a pipeline channel: `GET /api/v1/channels`, `hydractl channels list|follow`. A channel id can hold one series per bay, job or watch; `GET /api/v1/channels/{ch}` reads the newest and reports `series`, so narrow with `?tag=slot=3` (or `job=`, `watch=`) before trusting a cursor across runs.

> **Stopping a streaming job early.** A workload stop (or any caller of the job cancel) marks a running fio job `cancel_requested`; the agent learns of it from its next stream POST reply (`"stop": true`), kills fio and posts a `done` result with `summary.partial: true` and `elapsed_s`. A stopped run is therefore never reported as a completed profile. Non-streaming jobs are seconds-scale and simply finish; a queued job is failed at once.

Data policy

Nothing streams by default. The agent heart-beats every 10 s and re-posts its inventory only when it changes. Jobs are on demand with a 256 KiB summary and an 8 MiB artifact cap; artifacts live under a per-agent quota (default 256 MiB) and a retention window (default 30 days), pruned hourly and on

demand. fio is the one live stream, at 1 Hz, in a bounded ring. Watches are opt-in, rate-limited per watch, and echo only transitions.

Multiple hosts

Every serial may be seen by several hosts (a dual-port drive). HCC lists all of them; a job or watch on such a drive **requires** an explicit host (`agent_id` in the API, `--agent` on the CLI, the Host picker in the UI). With exactly one online host the choice is automatic. The validation gates cannot ask, so they compare against the lowest agent id and record how many hosts saw the drive (`hosts_seen`).

API Reference

All under `/api/v1/hosts`. Operator token for everything below; the agent's own token reaches only `/agents/<its id>/…`.

Method	Path	Purpose
GET	`/hosts`	Enrolled agents
GET	`/hosts/agent/bundle`	Download the agent bundle (`hcc-agent-bundle-<version>.zip`); 503 when no bundle is available
GET	`/hosts/agent/bundle/info`	`agent_version`, `filename`, `bytes`, `sha256`, and `files[]` (`name`, `bytes`, `sha256` per member)
POST	`/hosts/enrollments`	Mint a one-time enrollment code
GET	`/hosts/devices`	serial → host bindings
GET	`/hosts/bay/{slot}`	Host-side view of a bay
GET / DELETE	`/hosts/agents/{id}`	One agent / revoke it
GET	`/hosts/agents/{id}/inventory`	Its latest inventory
GET	`/hosts/jobs/kinds`	The job catalogue
POST / GET	`/hosts/jobs`	Dispatch a job (`kind`, `params`, `slot` or `serial`, `agent_id`) / list jobs
GET	`/hosts/jobs/{id}?wait=N`	One job, long-poll up to 30 s
POST	`/hosts/jobs/{id}/cancel`	Cancel a job. A queued job fails at once; a running fio job stops at its next sample and ends `done` with `summary.partial: true` (the record keeps `cancel_requested: true`); other running kinds finish normally. Cancelling a finished job is a no-op
GET	`/hosts/jobs/{id}/artifact`	The raw artifact
GET	`/hosts/artifacts`	Per-agent artifact usage
POST	`/hosts/artifacts/prune`	Retention + quota eviction now (`agent_id` optional)
GET	`/hosts/streams`, `/hosts/streams/{job}`	fio streams (cursor); `/ws/hosts/stream` for the UI
GET	`/hosts/watches/kinds`	The watch catalogue
POST / GET	`/hosts/watches`	Start / list watches
GET / DELETE	`/hosts/watches/{id}`	One watch / stop it
GET	`/hosts/watches/{id}/events?since=N`	Its raw events (cursor)
GET	`/hosts/enumerations`	Hot-plug enumeration timings

Write recipes return **422** without `confirm=true` and **409** when the host has not opted in; a drive visible from several online hosts returns **409** naming them until `agent_id` is given.

CLI

```
hydractl host list | show | revoke | inventory | devices | bay
hydractl host agent-bundle [-o FILE]
hydractl host enroll --name NAME
hydractl host kinds
hydractl host run KIND --slot N | --serial S [--agent ID] [-p key=value]… [--confirm]
[--follow]
hydractl host jobs | job ID | artifact ID -o FILE
hydractl host cancel JOB_ID              # stop a queued or running (fio) job
hydractl host artifacts [--agent ID]     hydractl host prune [--agent ID]
hydractl host stream ID | streams
hydractl host watch kinds | start KIND | stop ID | list | events ID [-f]
hydractl host enumerations [--slot N]
```

Scripts: the `host_inband_check`, `host_fio_stream`, `host_aer_watch` and `host_reset_recovery` examples from the examples download (see *chapter 11*).

18. Workloads

A **workload** is a saved test you can run again: which bay (and which Quarch module), which steps in which order, which channels to watch, which gates to read at the start and the end, and what to export. The Workloads page is where you configure one, run it, watch it live and read its verdict; every button has a REST route and a `hydractl workload` command behind it, so the same run scripts from Python, LabVIEW, Jenkins or a shell.

Three rules shape the whole feature:

- **Errors are counted and shown, never auto-failed.** A correctable AER that moved during a run is a *review* verdict with the delta on the tile, in the log, in the report. Only a gate you declared for the end of the run, or a stop guard you armed, can produce *failed*.
- **STOP TEST is never gated.** It stops the run inside a second, unwinds what the run started (fio cancelled, watches stopped, Quarch signals restored, recordings stopped, a self-test aborted), reads the closing gates and lands `stopped early` with a verdict. It is not a failure — but it is not a pass either: `hydractl workload run --wait` exits 2 for a run that did not complete, whatever its gates said.
- **Destructive steps need an ARM.** Host write recipes, fio write profiles, bay power and bay resets, Quarch glitches, signal breaks and hot-swap cycles, and device self-tests unlock only for one run, for a limited time (15 minutes by default), after a confirm that lists every destructive step; the arm clears when the run ends.

> **Hardware-validated presets.** Every preset carries a `bench_proven` flag. A preset not yet run end to end on hardware shows a **BENCH** pill in the Library, and loading one shows a validation warning. Presets run normally — the flag is information, not a lock.

The page

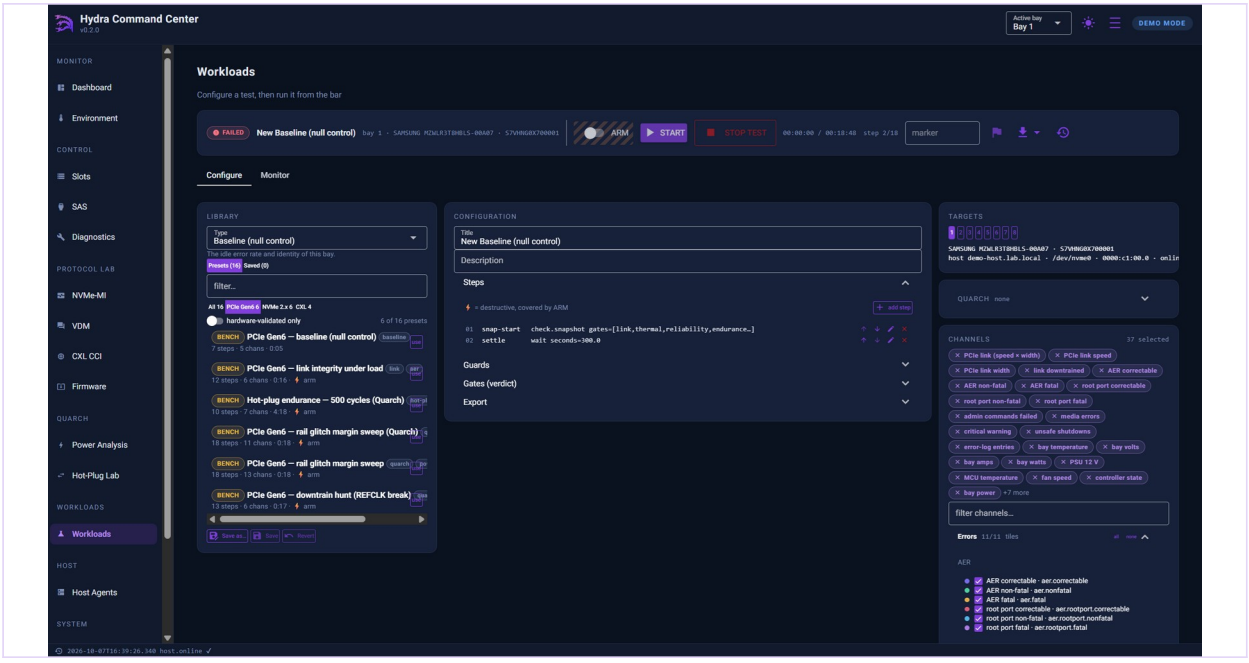


Figure 18.1 Workloads. Configure a saved test, run it, and watch it live.

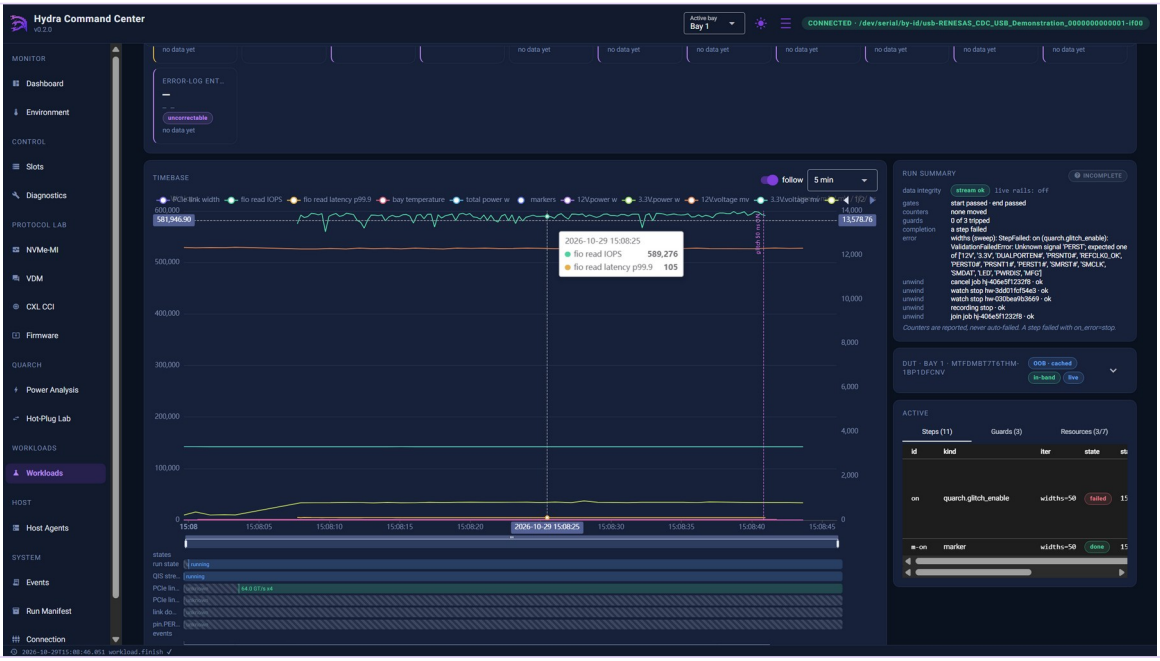


Figure 18.2 Workload Monitor. Run-control bar with live state chip, channel tiles, and gate readouts.

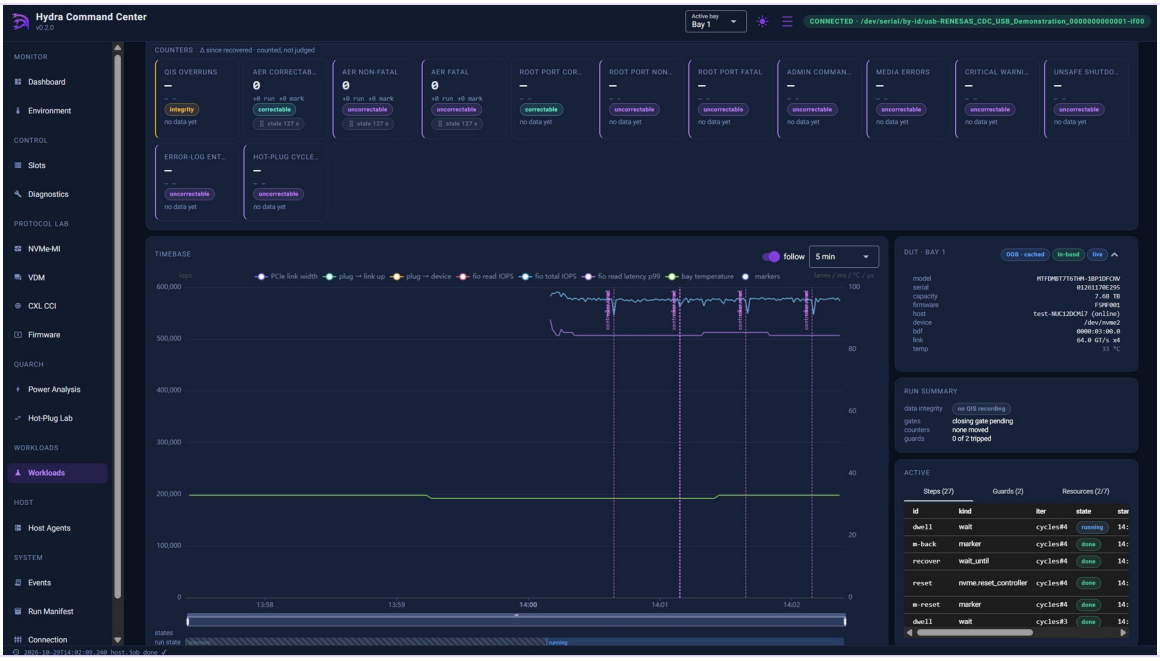


Figure 18.3 Workload Monitor — Verdict. Closing gates and the run's final pass / review / fail verdict.

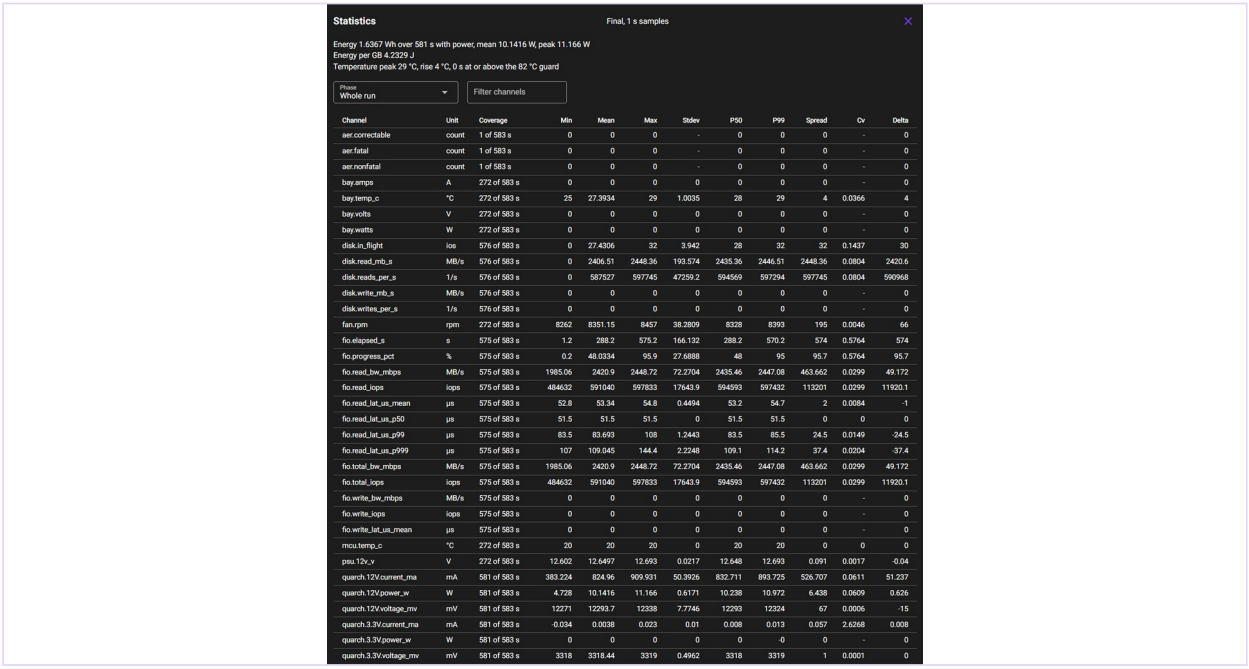


Figure 18.4 Statistics panel. Per-channel coverage, min, mean, max, standard deviation, P50/P99, spread and CV for the run or one phase.

Run-control bar (always visible): the state chip (`IDLE`, `ARMED`, `RUNNING`, `STOPPING`, `STOPPED EARLY`, `DONE · PASS / REVIEW / FAIL / INCOMPLETE`, `ABORTED`, `FAILED`), the template name and target bay, the **ARM** switch (a switch, never a button; the confirm lists every destructive step), **Start**, **STOP TEST**, **Force abort** (appears after 15 s of stopping), the elapsed / estimated clock and step progress, the marker box, the **download ▾** export menu (JSON manifest, CSV, NDJSON, Report (HTML) — enabled once the run has finished) and the **run history** drawer.

Configure tab: three columns. The **Library** card on the left holds the **Type** select (always visible), a **Presets (n)** / **Saved (n)** segment and one filter box. The Presets segment adds a family filter (All · PCIe Gen6 · NVMe 2.x · CXL, with counts; it follows the type you pick and never changes the draft), a **hardware-validated only** switch, and a **BENCH** pill on every preset not yet run end to end on hardware (hover the pill for the tooltip "not yet run end to end on hardware"). **Save as...**, **Save** and **Revert** sit at the bottom of the card. **Configuration** in the middle: title, description, then the sections **Steps** (open; add, edit, move, remove; loops and sweeps nest), **Guards** (`channel op value → stop|mark`), **Gates** (the verdict, at start and end) and **Export**. The right rail: **Targets** (bay, agent and DUT binding), **Quarch** (collapsed until a module is bound or required) and **Channels**: every group collapsed with `selected/total` and the Monitor panel it feeds (scope, tiles, strips, timeline) in its header, the groups of the chosen type expanded and listed first, all / none in the header, the fat groups split into captions (fio throughput, fio latency, block I/O; AER, SMART / NVMe; temperatures, bay rails, enclosure), channels not yet published folded at the end of their group, and the current selection as removable chips above the filter. Groups such as `errors`, `link`, `quarch` or globs like `quarch.*.power_w` still ride through as authored. The validation strip revalidates on every edit and names each problem by path. **Start** stays disabled until validation is clean, the draft is saved (Start runs a saved template), and — for a template with destructive steps — the run is armed; hover Start for the blocking reason. A preset not yet validated on hardware also shows a `BENCH_PRESET` warning ("This preset has not been run end to end on hardware"), which never blocks. Save as a custom template (preset ids are reserved; a custom may fork a preset and keep its `preset_key`).

The Monitor offers every channel available to the run, not only the ones the preset charts. Selecting a channel only shows or hides it; its data keeps collecting. Quarch rails and pins need a module bound to the run. Hover over an enabled channel to emphasise its line on the chart and dim the rest.

Statistics. The **Statistics** button in the Monitor header opens a side panel with each channel's coverage, min, mean, max, standard deviation, P50, P99, spread, CV and change, for the whole run or one phase of it, plus energy and temperature figures where those channels exist. From the CLI: `hydractl workload stats <run> [--phases]`; from the API: `GET /workloads/runs/{id}/statistics`. The JSON manifest includes them too.

Each phase with I/O also gives IOPS/W and J/GB, shown when you pick that phase. A step whose throughput swings is marked unsteady: its mean is not a sustained rate.

While a run is going the figures are marked provisional. They come from the stored samples, so they are not device latency percentiles, and spikes between samples are not seen. Always read the coverage: `294 of 328 s` means the figures describe only those seconds. Statistics are informational and never change a verdict.

Qualification Report. For a finished run, pick **Qualification Report (HTML)** from the Monitor's export menu, or run `hydractl workload qualify <run> --document QR-0042 --operator "A. Tester" -o report.html`. It is one file you can file, send or print to PDF (the page is set up for A4 landscape):

1. **Summary and findings.** The result, best figures, power, and anything a reader must not miss: moved counters, a guard that fired, a bad snapshot even if the run went on to pass, time at warning temperature, low coverage and unsteady steps. Findings are informational, and the verdict stands.
2. **Device and setup.** The drive, enclosure, host, OS and software versions, recorded when the run started. Runs from before this version say the device was not recorded.
3. **Results by phase.** Power vs Performance: IOPS, MB/s, mean and peak W, IOPS/W, J/GB and P99 latency for each step. Peak W is the highest 1 s sample.
4. **Health before and after.** Each gate at each snapshot, drive temperature sensors and error counters.
5. **Evidence and sign-off.** Charts, full statistics, steps and the event list, then document number, customer, tester and reviewer. Anything you leave blank stays blank for a pen.

Generate the report within a day of the run: per-step figures come from stored samples, which are kept for about 25 hours. After that the report says the data has gone.

Monitor tab, top to bottom: a slim run header (the run picker, `run id · bay · DUT model`, the start → end stamps or "started … · running" with a progress bar, and the verdict chip — the state, elapsed and step live in the sticky bar above), the **Counters** card (tiles with Δ-since-start and Δ-since-marker, captioned with the marker they count from; counted, never judged; correctable and uncorrectable carry their own colours), then ONE **Timebase** card whose follow switch and window select ("5 min", "1 h", "6 h", "24 h", "whole run" — the long ones read the aggregate tier and say "hot + rollup") govern three views on one shared time axis: the scope (a true time axis with run markers), the state strips directly under it (link, Quarch pins, QIS stream status) and the event timeline (steps, markers, gate snapshots, guard trips, warn/bad rows), so a vertical line at one instant reads straight down. Under it the streamed **parsed-log table** (kernel lines, AER and SMART counter moves, link states, admin traces, enumeration timings, gate checks, guards, markers; filter by source / class / level / text — the filter bar folds away —, follow the tail, pin rows, download as CSV). The right rail: the **DUT** card, the **run summary** with the verdict, any tripped guard (rule · observed · time · action) right under it, and the **Data-integrity** chip (QIS stream ok / overrun / stopped, and the live rail feed's state), and the **Active** card with Steps / Guards / Resources tabs (a tripped guard turns its tab red). Once a run ends the summary moves to the top of the rail and the DUT card collapses to one line. On a laptop the rail widens; below that everything stacks.

Run history (the drawer): every run with its state, verdict and completion; open one into the Monitor, delete a finished one.

Presets

Sixteen ship with HCC; `hydractl workload templates` lists them. Presets marked *host* need an enrolled host agent (*chapter 17*). *host (writes)* means the run writes through the host: a write recipe (which also needs the host's write opt-in) or a write fio profile (which needs a blank namespace).

Preset	What it does	Needs
`pcie-gen6-baseline`	null control: gates, watches, 5 min idle, gates	host
`pcie-gen6-link-integrity`	seq-read load, five link retrains with markers	host (writes), ARM

Preset	What it does	Needs
`pcie-gen6-rail-margin-sweep`	QIS recording, random-read load, PERST glitch widths 50 ns → 500 µs on the hardware glitch engine	host, Quarch, ARM
`pcie-gen6-downtrain-hunt`	ten REFCLK breaks under load, restored to the captured source	host, Quarch, ARM
`nvme2x-perf-profile`	block-size and queue-depth sweeps	host
`nvme2x-admin-conformance`	identify, logs, features, self-test	host (self-test = ARM)
`nvme2x-reset-recovery`	ten controller resets under load, recovery timed marker → first IOPS	host (writes), ARM
`nvme2x-selftest-under-load`	short self-test polled every 30 s under load	host, ARM
`nvme2x-write-endurance`	24 × 10 min random write	host (writes), ARM
`cxl-t3-health-watch`	detect, identify, health every 30 s, alert config	CXL bay
`cxl-t3-event-sweep`	every event log, then the events gate	CXL bay
`cxl-t3-inband-inventory`	`cxl list -M` vs the OOB identify	CXL bay, host with cxl-cli
`cxl-t3-power-thermal`	QIS recording around health reads	CXL bay, Quarch
`hotplug-endurance`	500 Quarch hot-swap cycles with every enumeration timed	host, Quarch, ARM
`thermal-soak`	8 h of 10-min sequential-write jobs, SMART sampled between, bay-temperature stop guard	host (writes), Quarch, ARM
`quarch-power-margin`	PERST glitch-width sweep on the hardware glitch engine under load, with a QIS recording	host, Quarch, ARM

Presets that have run end to end on hardware have `bench_proven: true`: `pcie-gen6-baseline`, `pcie-gen6-link-integrity`, `pcie-gen6-rail-margin-sweep`, `nvme2x-admin-conformance`, `nvme2x-perf-profile`, `nvme2x-reset-recovery`, `nvme2x-selftest-under-load`, `nvme2x-write-endurance` and `cxl-t3-health-watch`. The Library shows a **BENCH** pill on every other preset, the **hardware-validated only** switch hides those, and the validation strip shows a `BENCH_PRESET` warning when one is loaded. The flag is visible in `hydractl --json workload templates` / `hydractl --json workload show`. A saved copy of a preset is your own template and carries `bench_proven: false`.

Live SMART

Most NVMe presets start a `smart` watch, which reads the drive's SMART log every 30 s. The Monitor shows media errors, error-log entries, unsafe shutdowns, thermal throttle events and critical warning as tiles, the last by flag name rather than a number, and drive temperature, hottest sensor, each temperature sensor the drive reports, available spare and life used as channels. The composite temperature can sit far below the hottest sensor, and NVMe leaves what each sensor measures to the vendor, so sensors keep their numbered names. Throttle events count host-controlled thermal management only: a drive can also throttle internally, and those counters won't show it. A moved counter lands in the run log. Like AER, these are counted and shown, and never fail a run on their own.

Presets that trace the admin queue leave the watch out, so its reads stay out of the trace, and so does `hotplug-endurance`, because the drive keeps leaving. A read that fails skips one sample. If the reads keep failing, the tile goes stale after 45 s. Failed admin commands are not tracked live.

Reading sustained throughput

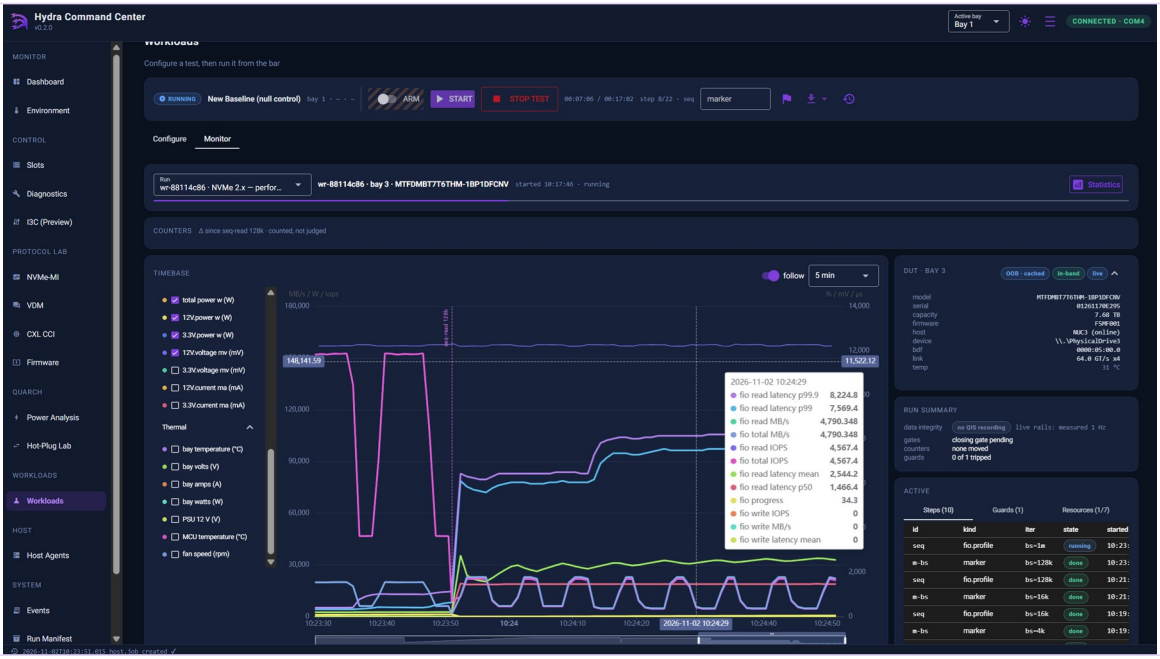


Figure 18.5 Power vs Performance. IOPS, MB/s, mean and peak watts, IOPS/W, J/GB and P99 latency by phase, as read from the stored samples.

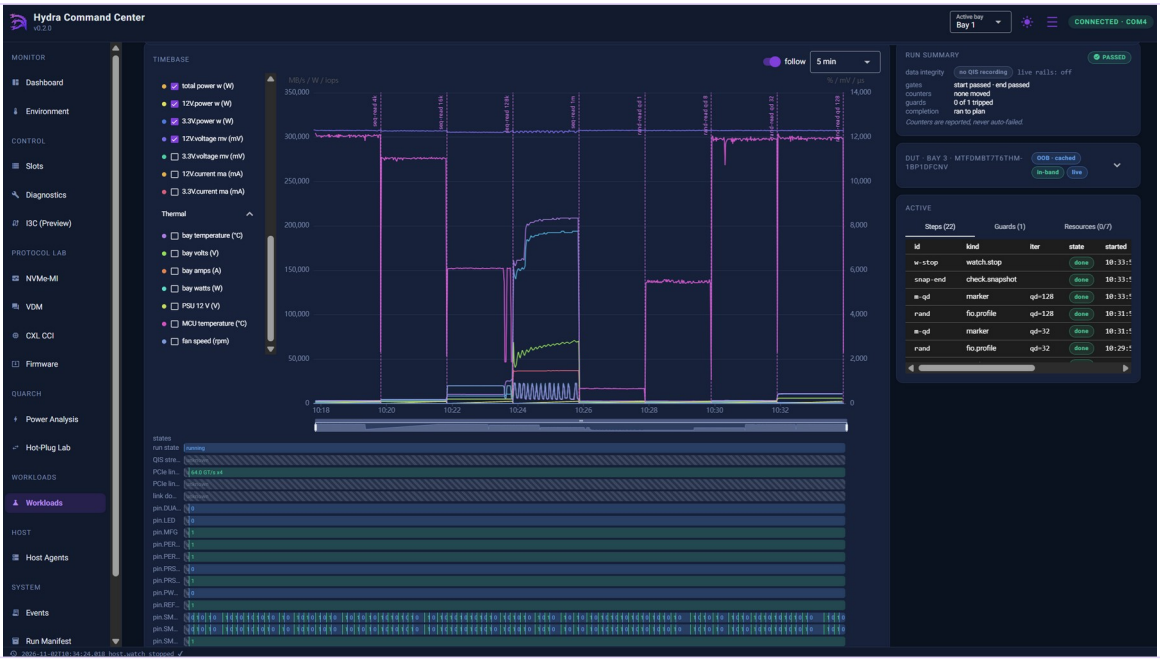


Figure 18.6 Power vs Performance — detail. A single sequential step expanded, showing the throughput cycle against drive temperature.

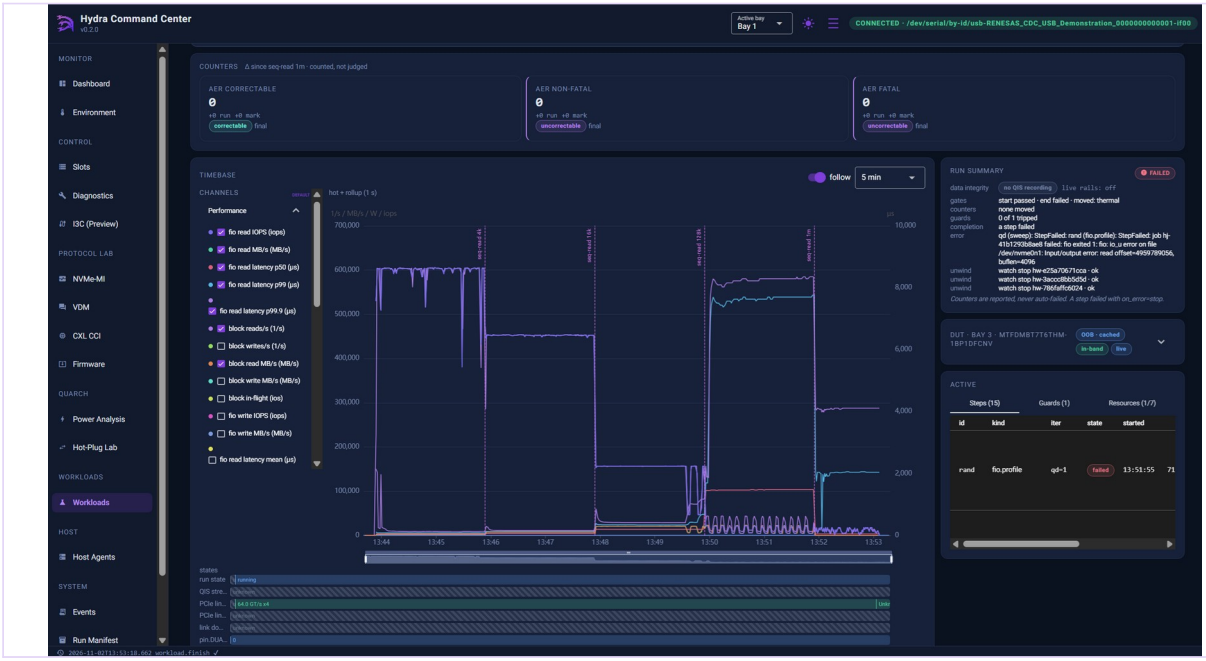


Figure 18.7 Sustained throughput — Linux. The ~24 s peak/trough cycle on our Gen6 ×4 bench drive: about 22.9 GB/s, then 5-6 GB/s, averaging about 12 GB/s.

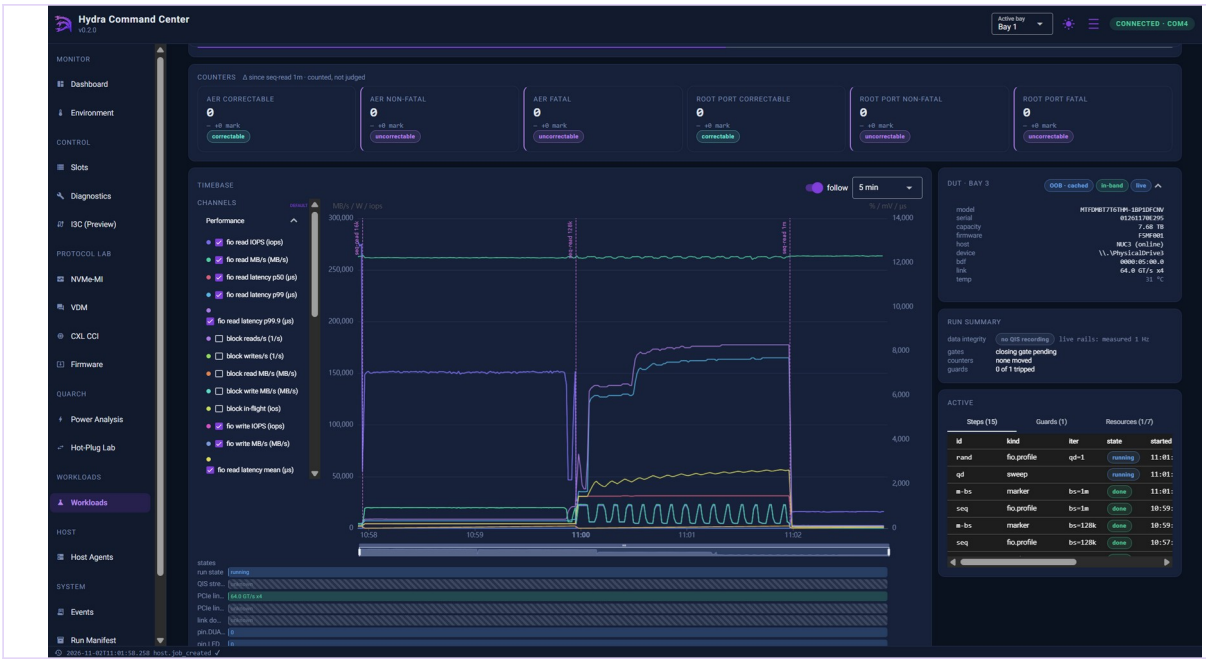


Figure 18.8 Sustained throughput — Windows. The same cycle reproduced under Windows, consistent with the Linux run.

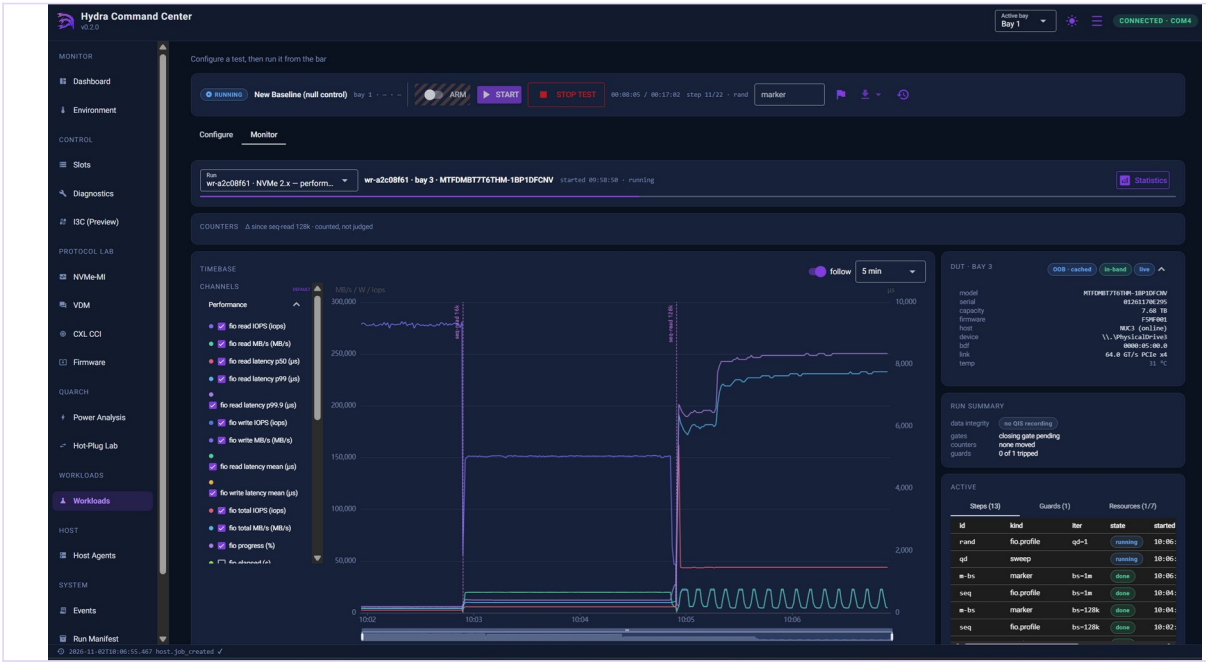


Figure 18.9 fio channels. fio throughput, latency and block-I/O channel groups plotted on the Monitor timeline.

A short read shows a drive's peak, and a long one shows what it holds. In a 120 s sequential 1 MB step, a Gen6 ×4 drive on our bench repeated a cycle of about 24 s on both Linux and Windows: about 22.9 GB/s at 24 W, then 5-6 GB/s at 13-14 W, averaging about 12 GB/s. So quote a peak as a peak, and read the step's chart as well as its mean. Check the drive's temperature sensors too: the hottest can sit far above the bay temperature. On this drive one sensor passed 70 °C while the bay read 31 °C.

The thermal gate compares the composite temperature to the drive's own warning threshold, because that is what NVMe defines it against, or to 70 °C, the spec's recommended value, if the drive reports none. A hot sensor is reported but not gated. For a specific limit, set `limit_c` on the gate.

Running

```
hydractl workload run pcie-gen6-baseline --slot 1 --wait          # exit 0
passed/review, 1 failed, 2 incomplete or stopped early
hydractl workload run quarch-power-margin --slot 1 --quarch <con_string> --arm --ttl
600 --wait
hydractl workload run my-custom --slot 2 --set fio.runtime_s=120 --note "after FW 1.2"
--no-start
                                  # --set naming a step that doesn't exist is refused
hydractl workload arm wr-8f2a1c4d --confirm      # idle → armed for the TTL
hydractl workload start wr-8f2a1c4d
hydractl workload mark wr-8f2a1c4d "before glitch"
hydractl workload stop wr-8f2a1c4d --reason "seen enough"   # never gated; --force
skips the unwind
hydractl workload status wr-8f2a1c4d --follow
hydractl workload log wr-8f2a1c4d --follow --source aer,gate --level warn,bad --grep
PERST
hydractl workload active
```

`--arm` is the explicit confirmation on `workload run`: without it, a template with destructive steps stops with exit code 2 and arms nothing. `workload arm` requires `--confirm`.

The `workload_run` example from the examples download (see *chapter 11*) — `python workload_run.py <template> --slot N [--yes]` — is the scripting shape of the Start button: the plan is printed and nothing is actuated without `--yes` when the template needs an arm; Ctrl-C stops early through the server.

One workload runs at a time (a second start answers 409 `RUN_ACTIVE`); an unarmed start of a destructive template answers `NOT_ARMED` or `ARM_EXPIRED`. If HCC restarts mid-run, the run is recovered at startup: the record's actuations are unwound and it lands `aborted` naming anything left asserted.

Verdicts, gates, guards, counters

- **Gates** are the validation checks (`link`, `reliability`, `thermal`, `self-test`, `host-identity`, `host-link`, `host-smart`, the `cxl-*` set). The start snapshot is the baseline; the end snapshot decides. A gate that could not be established reads *unknown — precondition not established*, never *good*.
- **Guards** watch a channel while the run is live: `stop` is your pre-armed STOP TEST (the run lands `guard_tripped`, which is *failed* only because you said so), `mark` drops a marker and re-arms.
- **Counters** (AER, SMART, admin failures, hot-plug fails, QIS overruns) snapshot at start and end; a moved one is *review*. A counter first seen mid-run baselines at its first sample inside the run.
- The **verdict** is `passed | review | failed | incomplete`; `auto_fail` is pinned false in every record and report.

Exports and the report

```
hydractl workload export wr-8f2a1c4d                          # JSON manifest to stdout
hydractl workload export wr-8f2a1c4d --format csv -o series.csv --cadence-ms 1000
hydractl workload export wr-8f2a1c4d --format ndjson -o samples.ndjson
hydractl workload export wr-8f2a1c4d --format html -o run.html
hydractl workload report wr-8f2a1c4d -o report.html            # written under the run
folder too
hydractl workload qualify wr-8f2a1c4d --document QR-0042 -o qualification.html
hydractl workload artifacts wr-8f2a1c4d [--get report.html -o …]
hydractl workload series wr-8f2a1c4d --channels bay.temp_c,fio.read_iops --csv
```

The **manifest** is one diffable object: the definition as run, component versions captured when the run started, the run block, steps, gate snapshots, guards, counters with their class, markers, unwind, channel URLs and the artifacts list. The **CSV** is the aligned bucket table at the run's cadence (a units row, every cell through the formula guard); it reports `coverage`, `evicted` and `rollup` honestly when the hot store no longer held the run's oldest points and the aggregate tier filled them. The **HTML report** is one self-contained file: header and versions, the DUT, the gate table, "Counters observed — shown, not judged", guards, server-rendered SVG charts (the page embeds its on-screen scope as a PNG when you use **Report (HTML)** in the export menu), steps, the first 200 series rows and the event list. Print CSS is A4 landscape. The **Qualification Report** is the version for filing or sharing, described above under the Monitor.

The live feed and the aggregate tier

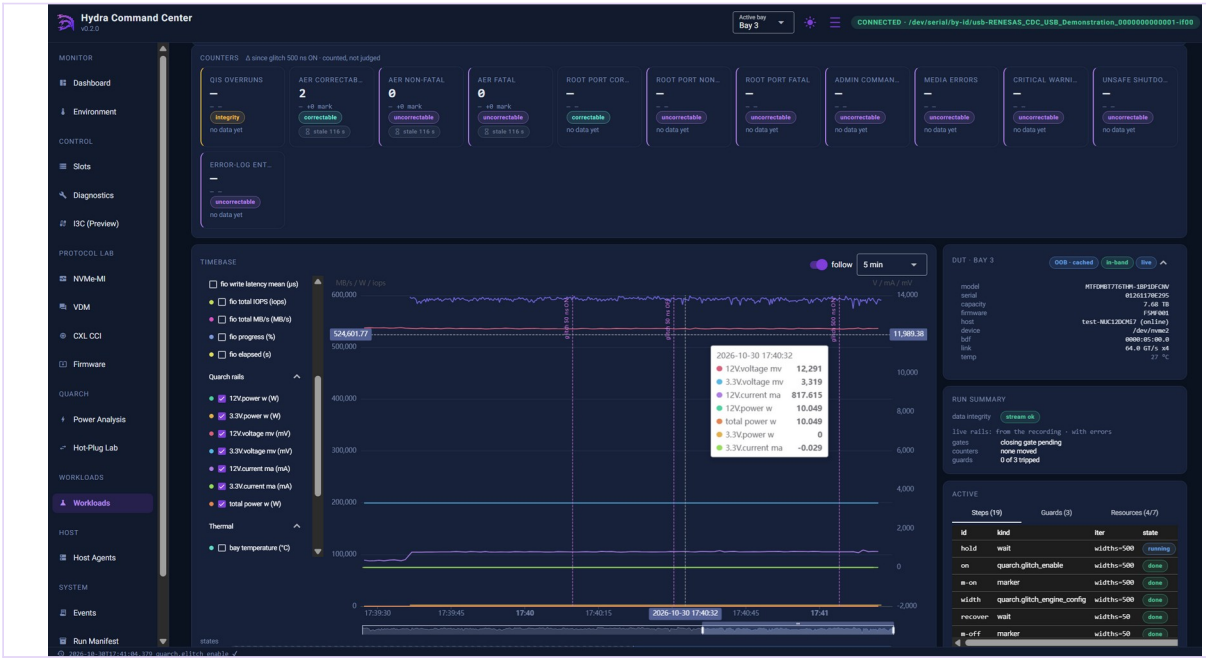


Figure 18.10 Quarch rails — overview. Live quarch.<rail>.voltage_mv / current_ma / power_w channels streamed into the Monitor once a module is bound.

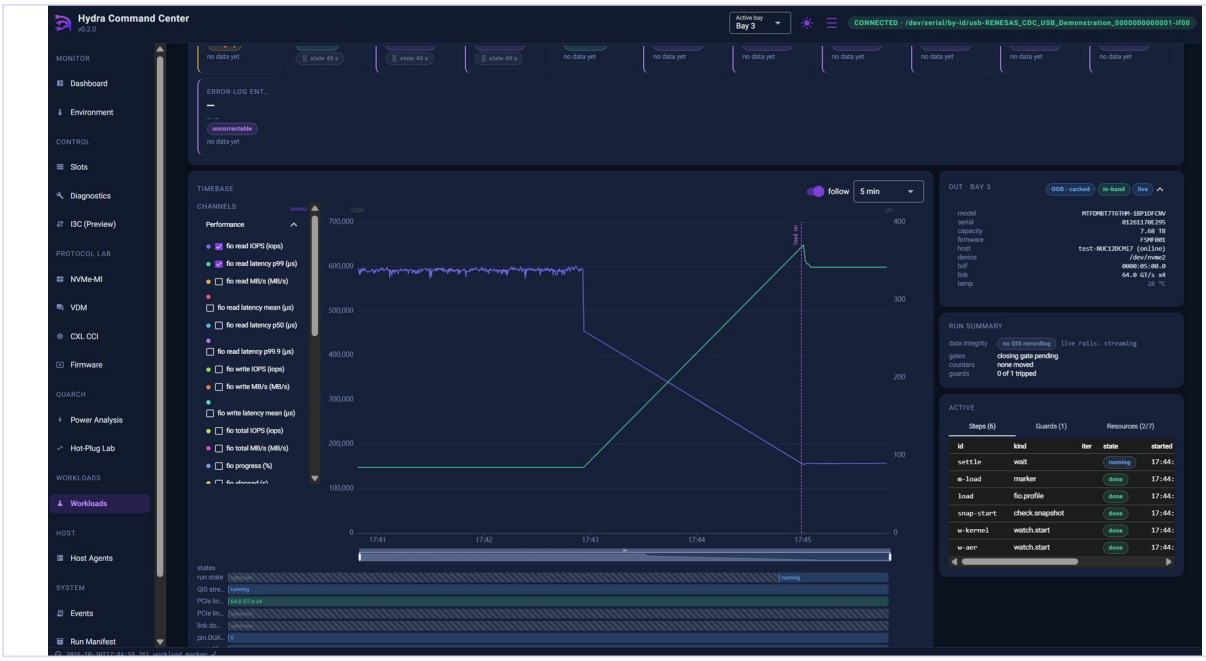


Figure 18.11 Quarch rails — detail. A single rail expanded against the run timeline.

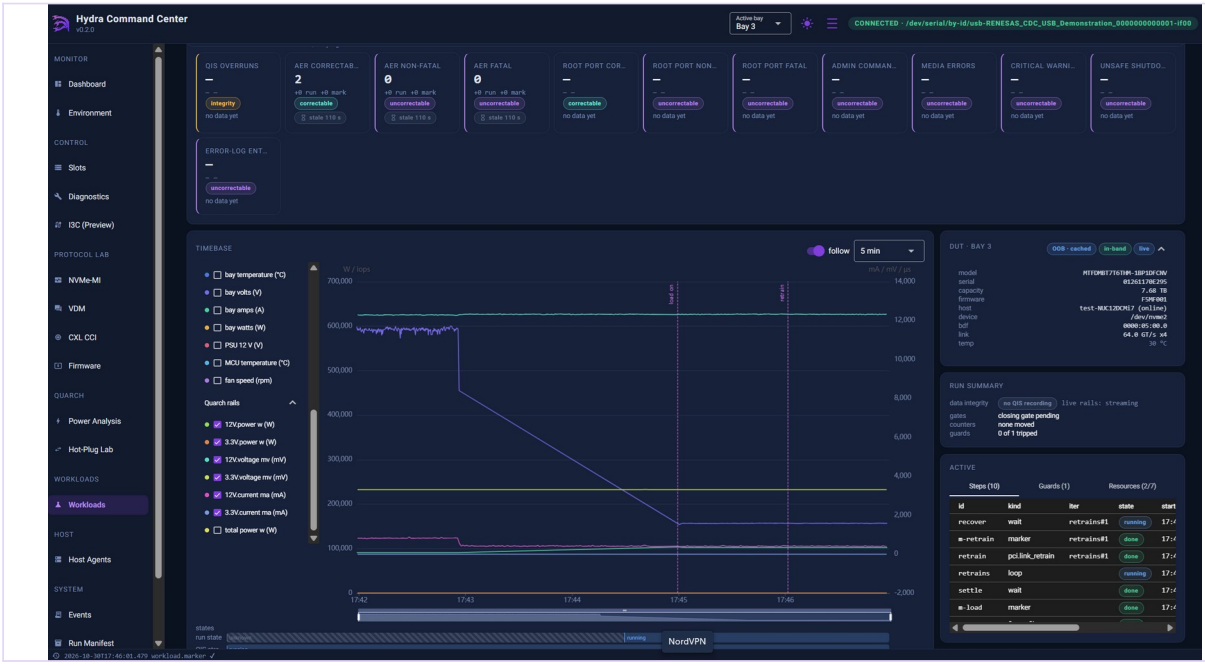


Figure 18.12 Quarch rails — QIS recording. Rails sourced from the QIS recording's own CSV tail while a recording is active.



Figure 18.13 Quarch rails with fio load. Rail power overlaid against a concurrent fio throughput channel.

While a run with a Quarch target is running, HCC polls the module's stream (`stream text 1`) once a second on the module's own handle and publishes the rails as `quarch.<rail>.voltage_mv | current_ma | power_w` and `quarch.total_power_w`. The feed runs only with a live consumer; a connected module with nobody watching costs no traffic. During a real QIS recording the module's stream belongs to the recording, so the rails come from the recording's own CSV tail instead ("live rails: from the recording" on the Data-integrity chip). `integrity.qis_stream_status` and `integrity.qis_overrun` come from `streamRunningStatus()` once a second while recording: an overrun means the power trace has a hole, and the chip goes red.

Every channel is folded into 1 s and 10 s buckets on disk, 24 h back:

```
hydractl channels rollup bay.temp_c --hours 24 --cadence-s 10 --tag slot=1
hydractl channels rollup quarch.12V.power_w --hours 2 --cadence-s 1 --csv
```

GET /api/v1/channels/{ch}/rollup?from&to&cadence_s&tag serves the same; the scope's "6 h" and "24 h" windows read it. The tier lives under the HCC data directory (hcc_data by default; set HCC_DATA_DIR to change it) in pipeline/rollup/{1s,10s}/<channel>/<hour>.ndjson, one file per channel-hour (8 MiB each) under a 6 GiB ceiling for the whole tier: past it the oldest hours go first, across every channel, and hydractl channels integrity reports the bytes held, the files, and what was dropped and why (dropped_disk_full, dropped_file_full, dropped_bad_stamp, dropped_series_cap).

API and CLI parity

Action	REST	CLI
list / show / validate / save templates	GET/POST/PUT/DELETE /workloads/templates…, POST …/validate, POST …/{id}/copy	workload templates / show / validate / create / copy / delete
step kinds, test types	GET /workloads/kinds, GET /workloads/types	workload kinds / types
create / arm / start / stop / marker / delete a run	POST /workloads/runs, POST …/{id}/arm ({"confirm": true, "ttl_s": N}), POST …/{id}/disarm|start|stop|marker, DELETE …/{id}	workload run / arm / disarm / start / stop / mark / rm
runs, the live one, a run (with wait)	GET /workloads/runs[?state&template_id&slot], GET …/active, GET …/{id}?wait=	workload status [--follow --wait] / active
series table, parsed log	GET …/{id}/series, GET …/{id}/log?since&source&level&class&slot&q	workload series / log
exports, reports, artifacts	GET …/{id}/export?format=, POST …/{id}/report, POST …/{id}/qualification, GET …/{id}/artifacts[/{name}]	workload export / report / qualify / artifacts
statistics	GET …/{id}/statistics?channels=	workload stats [--channels] [--phases]
the live run feed	WS /ws/workloads/{id} (state, samples, log, counters, dropped, verdict)	— (use status --follow + log --follow)
channels and the aggregate tier	GET /channels, GET /channels/{ch}, GET /channels/{ch}/rollup, GET /channels/integrity, WS /ws/channels	channels list / follow / rollup / integrity

All routes need the operator token; an agent's token never reaches them.