



HD Administrator Manual

Hyperscale Dataset Store — Deployment and Operations Guide —
v2.0.5

Table of Contents

1. HD Administrator and Deployment Manual	4
1.1 System Overview & Architecture	4
1.2 Deploying an Instance	6
1.3 Bare-Metal Host Preparation for Accelerated Storage Nodes	8
1.4 Networking & Storage Configurations	9
1.5 Security & TLS/PKI Requirements	10
1.6 Troubleshooting & Operational Tasks	11
2. What's new	12
2.1 2.0.5	12
2.2 2.0.4	13
2.3 2.0.2	13
3. HD System Architecture	15
3.1 Holistic System Topology	15
3.2 Component Architecture Breakdown	15
3.3 Host Requirements for Storage Nodes with SPDK/DPDK	17
3.4 Further Reading	20
4. HD Ecosystem Overview	21
5. Installation	22
5.1 HD — Administrator's Guide	22
5.2 install	36
5.3 Networking	42
5.4 install/ansible	44
5.5 install/helm/fermihdi	50
5.6 Installing HD on Talos Linux	52
6. Using HD	58
6.1 Querying HD	58
6.2 The SQL Server	62
6.3 The GraphQL Services	66
6.4 The MCP Server	68
6.5 Ingestion	70
7. Operations	76
7.1 HD: Operators & Admins Manual	76
7.2 Observability: metrics, logs and traces	82
7.3 The DMS WebUX	90
7.4 The Front Doors: Reverse Proxies, Authentication and Identity	96

7.5	The BloomRF Filter Node	99
7.6	Sandbox Data Management System (DMS) Operators & Admins Manual	103
7.7	FermiHDI HD Wire Protocol — Network & MTU Guide	105
7.8	Global User Guide	108
8.	Development	109
8.1	HD SDK — Developer Guide	109
8.2	SDK	114
8.3	Global Developer Guide	153
8.4	Global Test Guide	154
8.5	FermiHDI End-to-End Test Documentation	155
9.	Procurement and security	158
9.1	Security and development practices	158
9.2	Data handling and opt-in	160
9.3	Third-party software and licences	163

1. HD Administrator and Deployment Manual

Installing an instance? [install/admin-guide.md](#) is the guide to that: requirements, sizing, the cluster file, all three install paths and every option they take. This manual is the platform reference behind it — architecture, networking and storage configuration, the PKI model, and operational tasks.

1.1 System Overview & Architecture

HD — the Hyperscale Dataset Store — is developed and sold by FermiHDI Limited. It is an optimised storage engine for the records of a hyperscale dataset: one measured in billions of records and, at rest and unreduced, in terabytes to yottabytes. This manual covers deploying and operating an HD instance.

HD is an ultra-high-performance Data Management System designed for massive-scale analytics. It achieves extreme throughput by blending bare-metal acceleration technologies like DPDK (networking) and SPDK (storage) with a secure, Go-based control plane. This architecture bypasses traditional kernel bottlenecks, enabling a zero-copy data path where information flows from the network interface directly to storage or CPU cache layers.

1.1.1 Architectural Principles

The system is segmented into three distinct, isolated network planes:

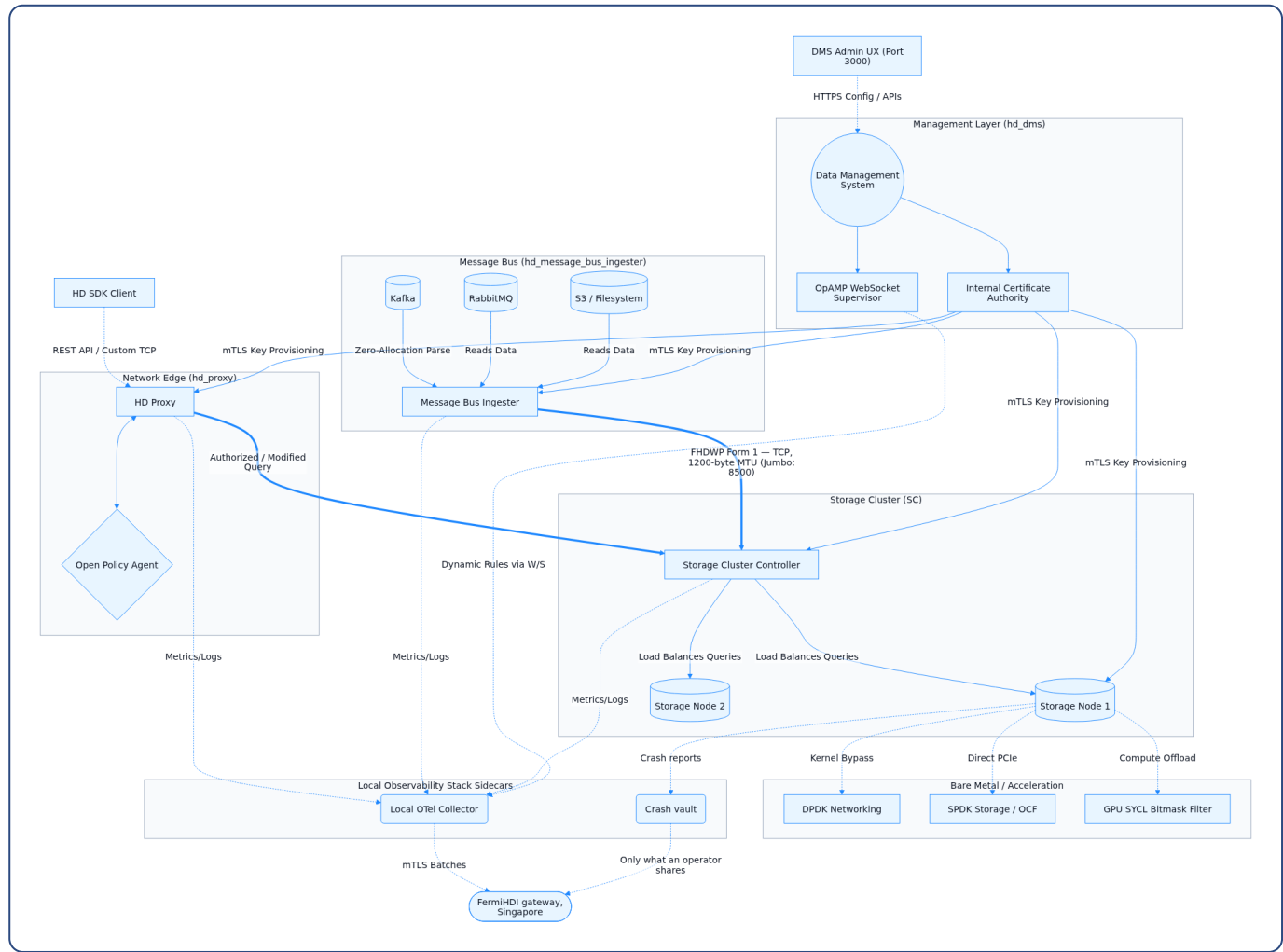
1. **Control Plane (Port 6986):** An HTTPS mTLS network for all orchestration traffic. This includes node registration, configuration distribution, policy updates, and telemetry management. User data records never traverse this plane.
2. **Data Plane (Port 6987):** A high-speed, unidirectional network for binary data streaming using the proprietary FermiHDI HD Wire Protocol (FHDWP). Data flows from Storage Nodes directly back to the requesting client.
3. **Internal DMS/CA Plane (Ports 6985, 6989):** Secure channels used exclusively by the Data Management System (DMS) components to bootstrap nodes and provision cryptographic identities.

1.1.2 Core Components

The platform is composed of several specialized microservices:

- **Data Management System (DMS):** The central source of truth and orchestration.
- **DMS-Core:** The Go-based engine that manages node topology, distributes schemas, and handles licensing.
- **DMS-CA:** An integrated Certificate Authority that provisions short-lived mTLS certificates to all nodes, forming the cluster's root of trust.
- **DMS-WebUX:** A Vue.js/Astro web interface (Port 3000) for system administration, configuration, and monitoring.
- **OTEL Collector:** Receives every node's metrics, logs and traces over OTLP and re-exports them wherever an administrator configures — push (OTLP, syslog, Prometheus remote write, Kafka) or, for Prometheus, pull from a fixed port. See the [observability page](#) for what each node type actually emits.
- **HD Proxy:** The ingress gateway for all data queries. It enforces Policy-Based Access Control (PBAC) using an embedded Open Policy Agent (OPA), rewrites queries based on security rules, and forwards authorized requests to the storage layer.
- **Storage Cluster Controller (SCC):** A C++ load balancer and orchestrator for a group of Storage Nodes. It receives queries from the Proxy and ingest data from the Message Bus Ingestor, distributing the workload across its assigned nodes. It uses NORM (NACK-Oriented Reliable Multicast) over UDP port 6988 for internal data distribution.
- **Storage Node (SN):** The C++-based data persistence and retrieval engine. It uses DPDK and SPDK to interact directly with networking and NVMe hardware, bypassing the kernel. It can offload filtering to GPUs (via SYCL) and hashing to specialized hardware like NVIDIA Bluefield DPUs.
- **BloomRF Filter Node (optional):** An engine role, at most one per storage cluster, that takes the SCC's ingest broadcast like a Storage Node and keeps a filter per batch instead of the records. At query time the SCC asks it which batches cannot match and leaves those unread. It persists nothing and is never allowed a false negative; see the [filter node page](#).
- **Message Bus Ingestor:** A high-performance C++ service that connects to Kafka, RabbitMQ or MQTT, or reads files from a directory or an S3-compatible store (JSON, CSV, Parquet, ORC, Avro, Protocol Buffers, XML), maps records onto the dataset schema and streams them to an SCC in batches; see the [ingestion page](#).
- **Query services:** Front doors onto the Proxy — the SQL server (`hd_sql`: PostgreSQL wire, Arrow Flight SQL, HTTP), the GraphQL services (`hd_graphql`, `hd_ipflow_graphql`), the MCP server for AI assistants (`hd_mcp`) and the Data Explore web front end. Each carries the SDK, receives results straight from the Storage Nodes, and converts addresses at its own boundary; see the [querying page](#).
- **HD SDK:** Client-side libraries (C++, Python, Go, JavaScript, Rust and others) that manage query execution. The SDK performs heavy computational work (SQL, Vector Search, Graph Math) locally on the data returned from the Storage Nodes, preventing computational bottlenecks within the storage cluster, and hands the developer the Arrow table itself.

1.1.3 System Topology Diagram



1.2 Deploying an Instance

There are four supported ways to stand up an instance, and all four are driven from one file — `install/cluster.yaml` — which describes the whole deployment. [install/admin-guide.md](#) is the guide to all of them; what follows is the summary.

Path	Use it for	Read
<code>install/ scripts</code>	preparing hosts: hugepages, VFIO, isolcpus, IOMMU	install/installer.md
<code>install/ansible/</code>	deploying the DMS and every node container across hosts	install/ansible.md
<code>install/helm/fermihdi</code>	the same deployment on Kubernetes	install/helm.md
<code>install/fermihdi-talos-*</code>	machines running Talos Linux	install/talos.md

```
install/fermihdi-discover          # what this host has
install/fermihdi-configure         # what it should run
install/validate-cluster cluster.yaml # is that coherent
sudo install/fermihdi-prepare cluster.yaml --apply # make it so

cd install/ansible && ansible-playbook site.yml # the DMS, then the nodes
```

1.2.1 What a node is actually given

Deploying by hand is a matter of getting one contract right, so it is worth stating plainly: **a node holds no configuration of its own**. What it is given is an identity, and everything else — record size, which SCC to talk to, which devices to use, the hash election — comes back from the DMS in the registration response.

```
docker run -d --name sc-1-scc --network host \
-e FERMIHDI_NODE_NAME=sc-1-scc \
-e FERMIHDI_DMS_URL=https://dms-core:6986 \
-e FERMIHDI_BOOTSTRAP_TOKEN="$SCC_TOKEN" \
-e FERMIHDI_CA_CERT_PEM="$(cat dms-ca-bundle.pem)" \
-e MASTER_CPUSSET=2-7 \
-e FERMIHDI_CONTROL_PLANE_SUBNET=10.147.0.0/24 \
-e FERMIHDI_DATA_PLANE_SUBNET=10.150.0.0/24 \
-v /dev/hugepages:/dev/hugepages -v /dev/hugepages1G:/dev/hugepages1G \
--ulimit memlock=-1:-1 --cpuset-cpus 0-1 \
cr.fermihdi.io/hd/scc:2.0.3
```

Four things about that command generalise to every node:

- **--cpuset-cpus is the overflow cores, not the poll cores.** `MASTER_CPUSSET` is what the process pins its reactors to, and those cores must stay out of the container's cgroup — the container runtime shim inherits it and will otherwise be scheduled onto them.
- **Both hugepage mounts.** `/dev/hugepages` is the 2 MB pool DPDK's EAL needs for networking; `/dev/hugepages1G` is the 1 GB pool SPDK uses for storage and its ring buffers. One mountpoint carries one page size, so they are separate.
- **--ulimit memlock=-1:-1.** The HD engine pins its arena; the default 64 KB stops it at startup. In Kubernetes this is the `IPC_LOCK` capability.
- **The bootstrap token is per node,** and the CA bundle is how the node knows it is talking to your DMS and not something else.

The Ansible layer builds exactly this command from `cluster.yaml`, which is the argument for using it rather than maintaining the above by hand.

1.2.2 Talos Linux

A machine running Talos has no shell, no SSH and no container runtime to run `docker run` against, so the host-preparation step cannot run on it. It is applied to it instead, over the Talos API, by a set of tools that read the same `cluster.yaml` and ask the same questions:

```
install/fermihdi-talos-discover -n 10.147.0.20 --format=tsv > facts.tsv
install/fermihdi-configure --facts facts.tsv --append cluster.yaml
install/validate-cluster cluster.yaml
install/fermihdi-talos-prepare cluster.yaml --apply
install/fermihdi-talos-k8s cluster.yaml -o values.yaml
helm install fermihdi install/helm/fermihdi-talos -f values.yaml
```

The interview and the checks are the bare-metal ones unchanged. What differs is that hugepages, driver binding and kernel modules become machine configuration, and that the kernel command line — `isolcpus`, the IOMMU, the 1 GB hugepage pool — is part of the boot image rather than a file, so applying it replaces the image and reboots the machine. [install/talos.md](#) is the whole of that path.

1.2.3 Sandbox Environment

For development and testing, a pre-configured sandbox environment is available. This automates the deployment of the entire stack.

```
cd ./tests/sandbox_dms
./start_sandbox.sh
```

To shut down and clean up the sandbox environment:

```
cd ./tests/sandbox_dms
docker compose down -v --remove-orphans
```

1.3 Bare-Metal Host Preparation for Accelerated Storage Nodes

For maximum performance, Storage Nodes (SN) should be deployed on bare-metal hosts with specific kernel and system configurations to support SPDK and DPDK. These steps must be performed on the host machine before launching the SN container.

1.3.1 IOMMU Configuration

The IOMMU (Input-Output Memory Management Unit) must be enabled in the system firmware (BIOS/UEFI) and via kernel parameters to allow direct device access (VFIO).

CPU Vendor	GRUB Parameter (/etc/default/grub)
AMD	<code>amd_iommu=on iommu=pt</code>
Intel	<code>intel_iommu=on iommu=pt</code>

After editing GRUB configuration, update and reboot: `sudo update-grub && sudo reboot`.

1.3.2 Hugepages Allocation

DPDK and SPDK require pre-allocated hugepages for efficient memory management. Both 2MB and 1GB pages are necessary.

Add the following to your GRUB command line: `default_hugepagesz=2M hugepagesz=2M hugepages=1024 hugepagesz=1G hugepages=4`

This allocates 1024 pages of 2MB (2 GiB total) for DPDK and 4 pages of 1GB (4 GiB total) for large DMA buffers.

1.3.3 CPU Isolation

To prevent OS scheduler noise from interfering with the data plane, isolate the CPU cores that will be assigned to the Storage Node.

Add the following to your GRUB command line (example reserves CPUs 2-15): `isolcpus=2-15 rcu_nocbs=2-15 nohz_full=2-15`

1.3.4 Host Setup

`install/fermi-hdi-prepare` applies all of the above from `cluster.yaml`: the hugepage pools per NUMA node, the `hugetlbfs` mounts, the `vfio-pci` bindings and the GRUB command line. It is idempotent, it reports what it would change before changing anything, and it never reboots — `--verify` is the pass to run after one. That is the supported path, and the one the Ansible layer drives.

```
install/validate-cluster cluster.yaml
sudo install/fermi-hdi-prepare cluster.yaml      # dry run
sudo install/fermi-hdi-prepare cluster.yaml --apply
install/fermi-hdi-prepare cluster.yaml --verify  # after the reboot
```

`scripts/host_setup.sh` remains for working on a machine by hand — deciding what it can do, or fixing one thing at a time:

```
# Check the current system status
sudo bash scripts/host_setup.sh status

# Configure hugepages and add to /etc/fstab
sudo bash scripts/host_setup.sh hugepages

# Bind an NVMe device to the vfio-pci driver for SPDK
sudo bash scripts/host_setup.sh bind 0000:01:00.0

# Get a recommended GRUB command line for the host
sudo bash scripts/host_setup.sh grub-advice
```

1.4 Networking & Storage Configurations

1.4.1 Networking

Port Assignments

Port	Plane	Purpose
3000	Management	DMS WebUX
6985	Internal DMS	Secure channel for DMS component communication
6986	Control Plane	mTLS API for node registration, config, and telemetry
6987	Data Plane	Unidirectional FHDWP binary streaming from SN to client
6988	Auxiliary	NORM Multicast for SCC-to-SN data distribution
6989	Internal CA	DNS CA Network
6997	Control Plane	The filter node's answer to the SCC: which batches cannot match
5432 / 31337 / 8081	Client	The SQL server: PostgreSQL wire, Arrow Flight SQL, HTTP
8080	Observability	Prometheus metrics and health endpoints on all nodes

FHDWP MTU Configuration

The FermiHDI HD Wire Protocol (FHDWP) on the Data Plane (port 6987) has strict MTU requirements to prevent IP fragmentation.

Mode	MTU (bytes)	Condition
Standard	1200	Default. Safe for all environments, including cloud and container overlays (VXLAN).
Jumbo Frame	8500	Admin-elected. Requires end-to-end network support for jumbo frames (≥ 8550 bytes).

Records are never split across frames. Standard mode packs records into 1133 bytes of payload per frame, Jumbo Frame mode into 8433. A record wider than the payload travels alone in a frame of its own; the data plane runs over TCP, so that frame is segmented to the network's MTU like any other. The widest record a dataset may declare is 8433 bytes, in either mode.

[!WARNING] The MTU setting is **instance-wide**. All nodes in a cluster must use the same MTU. Mixing modes will lead to packet loss and data corruption. Do not enable Jumbo Frame mode in cloud environments (AWS, GCP, Azure) as they do not provide reliable end-to-end support.

To enable Jumbo Frame mode, the entire data path must be verified first.

Pre-flight Checklist: 1. Verify host NIC MTU: `ip link show <interface>` (should show `mtu 9000`). 2. Verify path MTU between nodes: `ping -M do -s 8472 <destination_ip>`. This must succeed without fragmentation. 3. For Docker, verify network MTU: `docker network inspect <network_name> | grep -i mtu`.

Changing the MTU on a running instance requires a **full cluster restart**.

1.4.2 Storage Node Configuration

Storage Nodes are configured via a local `config.json` file.

Example `config.json`:

```
{
  "node_name": "us-east-sn-01",
  "dms_url": "https://dms-core:6986",
}
```

```

"command_port": 6986,
"production_port": 8000,
"production_interface": "0000:04:00.0",
"drive_strategy": "fill_sequential",
"storage_devices": [
  { "name": "nvme_0", "address": "0000:01:00.0", "type": "nvme" }
],
"use_ocf": true,
"ocf_cache_devices": ["nvme_0"],
"ocf_core_devices": ["hdd_bulk_1"],
"filter_device_selector": "gpu",
"hash_offload_platform": "nvidia_bluefield"
}

```

Key Parameters: * `dms_url`: The address of the DMS for registration. * `production_interface`: The VFIO PCIe address of the network card for DPDK. * `storage_devices`: A list of storage devices, identified by their PCIe address. * `use_ocf`: Set to `true` to enable the Open CAS Framework, a caching tier that places fast devices (`ocf_cache_devices`) in front of slower, larger ones (`ocf_core_devices`). * `filter_device_selector`: Offloads query filtering to `gpu`, `cpu`, or `host`. * `hash_offload_platform`: Specifies hardware for cryptographic hash offloading.

1.5 Security & TLS/PKI Requirements

1.5.1 mTLS and Certificate Authority

HD operates on a zero-trust model, with all inter-node communication secured by mutual TLS (mTLS).

- **DMS-CA:** The DMS includes an integrated Certificate Authority which acts as the root of trust for the entire cluster.
- **Node Registration Lifecycle:**
 - a. A new node starts and connects to the DMS Control Plane (port 6986) using standard TLS.
 - b. The node sends an authenticated Certificate Signing Request (CSR) to the DMS.
 - c. The DMS-CA signs the CSR and returns a short-lived X.509 certificate to the node.
 - d. The node uses this certificate to authenticate itself for all subsequent communication within the cluster.
- **Criticality:** The DMS-CA is critical for cluster operation. If the DMS-CA is lost, the entire cluster must be rebooted to regenerate the trust chain.

1.5.2 Policy-Based Access Control (PBAC) with OPA

The **HD Proxy** is the enforcement point for data governance, using Open Policy Agent (OPA).

- **Policy Management:** Policies are written in Rego and managed through the **Security > Governance** tab in the DMS WebUX.
- **Dynamic Updates:** When a policy is updated in the DMS, it is pushed to all active Proxies via a `POST /v1/proxy/config` webhook, which triggers an atomic in-memory swap of the ruleset.
- **Query Enforcement:**
 - a. A client sends a query to the query front door — the customer's reverse proxy, Traefik in the reference deployment.
 - b. The front door authenticates the caller and sets the identity headers (`X-Forwarded-User`, `-Role`, `-Groups`, `-Email`), stripping any the client sent; see the [front doors page](#).
 - c. The HD Proxy receives the request and evaluates it against the active OPA policy.
 - d. Instead of rejecting queries, the Proxy rewrites them:
 - **Restricted Fields:** If a user is not allowed to see a field (e.g., `credit_score`), the proxy overwrites that part of the query payload with null bytes (`\x00`), making it cryptographically unmatchable.
 - **Enforced Fields:** The proxy applies hard constraints (`\xff`) to enforce filters (e.g., `tenant_id = 'abc'`) without dropping the connection.

1.6 Troubleshooting & Operational Tasks

1.6.1 Troubleshooting

- **Node Quarantine:** If a Storage Node begins emitting `Bad Batch Notice (BBN)` control messages, it can be quarantined via the DMS WebUX. The SCC will automatically re-route queries to healthy replicas.
- **Data Plane Sync Issues:** If an SN falls out of sync on the NORM multicast network (UDP 6988), rebooting the node will trigger a cold-boot synchronization process over the Control Plane to ensure data integrity before it is marked as `Ready`.
- **AVX-512 Illegal Instruction Crashes:** On systems using Windows Hyper-V, including Docker Desktop on Windows, the hypervisor can report AVX-512 support that the guest does not actually have, and an engine role then exits with `SIGILL` at startup. This affects that one virtualisation environment; it does not occur on bare-metal Linux or on a hypervisor that reports CPU features correctly.

Run engine roles on bare-metal Linux or a correctly configured hypervisor. Where the environment cannot be changed, contact FermiHDI support for images built for a host without AVX-512 — the published images are built for the full instruction set and the alternative is supplied rather than produced locally.

- **Viewing Logs:** To view the logs for a specific component in a Docker Compose environment:

```
docker compose -f ./tests/sandbox_dms/docker-compose.yml logs -f dms-core
```

1.6.2 Operational Tasks

- **Observability:** All nodes expose Prometheus metrics and health endpoints on port **8080**. The system is designed to integrate with a central OpenTelemetry (OTel) Collector for logs, traces, and metrics, and holds every crash report in a crash vault on the DMS host.
- **SDK Resource Management:** The HD SDK uses asynchronous C++ reactors. To prevent resource leaks and freezes, especially in garbage-collected languages like Python, developers **must** clean up connections properly. Use context managers (with `client:`) or explicitly call `client.stop()` in a `finally` block.
- **License Activation:** An Enterprise License must be imported via the DMS WebUX to activate full functionality, including node registration and telemetry.

2. What's new

New features and new functionality in each release of HD, newest first. Defect corrections are in each component's release notes and are not repeated here.

2.1 2.0.5

2.1.1 New features

- **Chess notation fields.** A `chess_move` field stores a move in standard algebraic notation as a 16-bit code, and a `chess_fen` field stores a full board position in 32 bytes. Queries show both as notation, and position filters (a piece on a square, the side to move) are evaluated on the storage nodes. See [Chess notation](#).
- **Chess board view in Data Explore.** Data Explore replays a game from its starting position and its list of moves on an interactive board, and filters games by position.
- **Array fields.** A field can hold a fixed-width list of values. Filters that ask whether some element matches, or whether none does, run on the storage nodes; the filter node prunes batches for the "some element matches" form. Every query service returns arrays as native lists. See [Arrays](#).
- **Enumerations and 8-bit text.** A dataset can declare named value lists. Fields store compact codes and every query service shows and accepts the labels. ASCII text fields are stored one byte per character. See [Enumerations](#).
- **Intra-record time compression.** A time field can be stored as a small delta from another time in the same record, in a unit of your choosing, and every query service returns the full timestamp. See [Time fields](#).
- **Opt-in cloud sharing.** A single, deployment-wide switch decides whether anonymised operational telemetry is shared with FermiHDI. It is off by default and enforced at the deployment's own collector. See [Data handling and opt-in](#).
- **Crash reporting and the crash vault.** Crashes in the engine roles, DMS-Core, the Proxy and the WebUX are captured, held and symbolicated on the DMS host. They are shared only as an operator decides, and can be exported for air-gapped support.
- **Bare-Docker multi-host deployment.** The installer can render plain `docker run` scripts for each host from the cluster file, for deployments with no orchestrator.
- **Signed images with provenance.** Every image is signed, carries labels naming its source repository, commit and branch, and ships with vulnerability exploitability statements. See [Security and development practices](#).
- **Ubuntu 26.04 image variants.** Every service is published as an OCI Rock and as an Ubuntu 26.04 image, each for AVX-512 and AVX2 processors.
- **Agent Skills.** Every installer tool, the sandbox, troubleshooting, hardware sizing, schema authoring, ingestion setup and the query services have an Agent Skill an AI coding assistant can load to drive them conversationally. Published at `downloads.fermihdi.io/skills/`. See [Agent Skills](#).

2.1.2 New functionality

- **Dry-run ingestion.** The ingester's `--dry-run` mode reads, decodes and maps records against a dataset schema exactly like a real ingest, but registers with no DMS, connects to no SCC and needs no licence — verify a data source and a schema are compatible before a real deployment sees either. See [Verifying a source before it feeds a deployment](#).
- **Jumbo frames end to end.** `FERMIHDI_JUMBO` enables 8500-byte data-plane frames on the ingester, the SCC and the storage nodes alike. A record wider than a standard frame's payload now travels whole in a frame of its own.
- **Closing the last batch.** The ingester closes an idle batch after a timeout, set manually or derived from the time span of the files it reads. The last batch of a file drop no longer waits for the next delivery. See [Closing the last batch](#).
- **Exact 64-bit values.** Integers and nanosecond timestamps larger than 2^{53} are stored exactly from every file format and from quoted JSON values.
- **Larger filters.** A query whose filter exceeds 64 KB, such as a long list of OR conditions, now reaches the storage nodes.

- **Time-series windows on the batch field.** Data Explore windows and buckets a time series on the dataset's batch identifier field.
- **Schema import from a share link.** The WebUX imports a dataset schema from the schema generator's share link.
- **Filters that survive restarts.** The filter node keeps its filters on local storage, and every node fetches its configuration again when it restarts.
- **Filter node licensing.** An SCC takes an `HD_BLOOMRF` licence slot to run its filter in active mode, and runs it passive when no slot is available. The filter node itself needs no slot.
- **Syslog export over TLS or plain.** The WebUX's syslog exporter has a transport setting (TLS, plain, or by port), and each message carries the configured facility, the record's severity, host name and service. See [OTEL Collector](#).
- **Hardened Kubernetes defaults.** The management and query services run without privilege escalation or raw-socket access and under the default system-call filter. The chart generates the internal access token when none is supplied.

2.2 2.0.4

2.2.1 New features

- **The filter node.** An optional BloomRF service per storage cluster. In active mode it tells the SCC which batches a query cannot match, so they are never read; passive mode measures the same without acting on it. See [The filter node](#).
- **The SQL server.** A read-only SQL front door over the PostgreSQL wire protocol, Arrow Flight SQL and HTTP, for BI tools, notebooks and JDBC, ODBC and ADBC clients. See [The SQL server](#).
- **Talos Linux.** A cluster whose machines run Talos Linux is installed from the same cluster file, over the Talos API.
- **File and RAM storage.** Storage nodes can run on a file or a RAM device as well as on NVMe drives.
- **Online licence activation.** Licences are activated against the FermiHDI licence service, encrypted and bound to the DMS they were issued for, and capacity is enforced.
- **Installer archive.** The installer and its tooling are published as a versioned download.
- **Query cancellation and open queries.** A client can cancel a running query through the Proxy, and a query may omit its time range to read the whole dataset.
- **Batch-level query policy.** Policy can restrict which batches a query is allowed to read.

2.2.2 New functionality

- **File and object ingestion.** The ingester reads Parquet, ORC, Avro, Protobuf and XML from directories and S3-compatible object stores, sorting each file's records into batches by the schema's batch field. See [Ingestion](#).
- **IPv4 and IPv6 addresses.** Address fields are converted at every query service boundary, in both directions, exactly.
- **Blob fields** are supported by the SDK and every query service.
- **In-place certificate renewal.** Node certificates renew themselves before they expire, without a restart.
- **Separate hugepage pools.** Networking uses 2 MB hugepages and storage 1 GB hugepages, each discovered at start.
- **Filter node provisioning in the WebUX,** with the filter mode set per deployment and each SCC's filter state shown.
- **Query stream lifetime.** The query services hold the client stream open for the life of a query, with an explicit completion signal and a liveness signal on idle connections.
- **Excluded batches reported.** When the filter node excludes batches, the SCC reports them to the client, so a result is known to be complete.

2.3 2.0.2

2.3.1 New features

- **The installer.** Discovery, configuration, validation and preparation of a whole cluster from one cluster file, deployed with Ansible on bare metal or with Helm on Kubernetes.

- **Query priority.** A query carries a quality-of-service weight. Storage nodes schedule work by it with weighted fair queuing, and policy can override it.
- **Policy-based access control in the Proxy.** Policy can inject filters into a query and adjust its priority.
- **Verified batches.** A batch is finalised only after every storage node that received it has verified it, and each batch is recorded with its closing hash.
- **Direct NVMe storage.** Storage nodes drive NVMe devices from user space, with RAID-0 across devices and support for SATA, SAS, Fibre Channel, iSCSI, RAM and NVMe over Fabrics.
- **Observability endpoints.** Every engine node exposes health, recent logs and metrics, and exports metrics over OpenTelemetry.

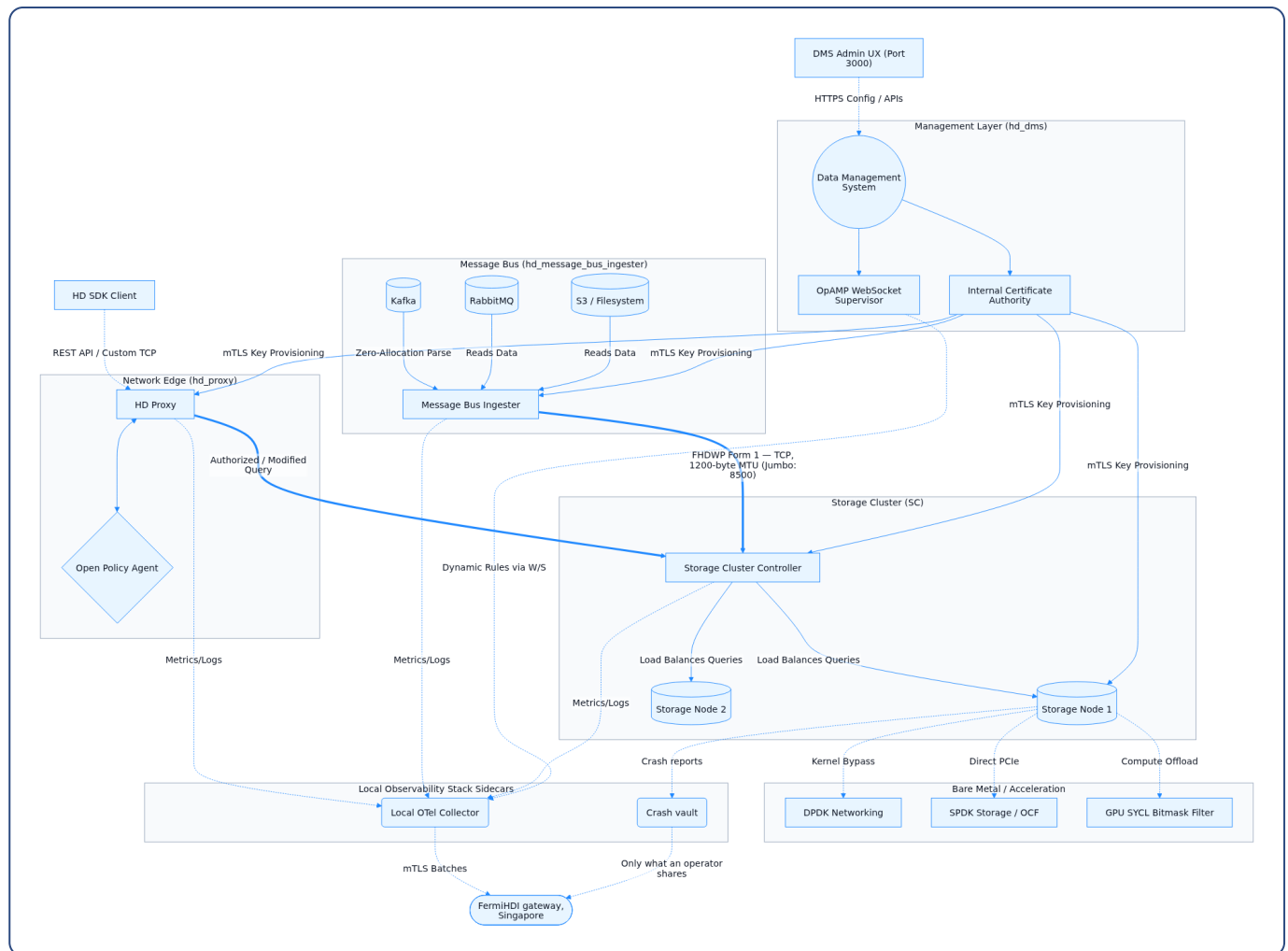
2.3.2 New functionality

- **Topology-aware CPU placement.** Cores are allocated by NUMA node, cache domain and physical core.
- **Faster, larger queries.** Wide queries stream their batch lists, and the query relay on the storage nodes was redesigned with bounded memory and admission control.
- **Configurable chain hashing,** including the XXH3 chain with a tunable block size.
- **Finer timings.** Latency is reported in microseconds.

3. HD System Architecture

HD by FermiHDI is an ultra-high-performance, bare-metal acceleration Data Management System designed to bypass the Linux kernel for maximum networking and storage throughput.

3.1 Holistic System Topology



3.2 Component Architecture Breakdown

3.2.1 Data Management System (DMS) `hd_dms`

The authoritative root of the clustered ecosystem (written in Go / Astro). - **Node Registration & Topology**: Maintains the complete physical layout of `hd_scc` and `hd_storage_node` agents. Nodes check-in and register their capabilities. - **DMS-CA (Root of Trust)**: An integrated Certificate Authority that automatically provisions short-lived X.509 certificates to support strict mTLS across all control-plane connections. The data plane (FHDWP) is not encrypted; see [Networking](#). - **Config & Schema Engine**: Synchronizes JSON schemas, Open Policy Agent (OPA) governance models, and dynamic telemetry instructions.

3.2.2 HD Proxy `hd_proxy`

A highly concurrent Go application acting as the PBAC (Policy Based Access Control) firewall and query scatter/gather router. - **Client Interface**: The target for `hd_sdk` connections in proxy mode (a client in direct mode registers with the DMS as an

`HD_CLIENT` node and submits to the SCCs itself), reached through the customer's query front door, which authenticates the caller, sets the identity headers the policy reads and presents a client certificate to the Proxy (see the [front doors page](#)). - **OPA Governance (PBAC)**: The proxy evaluates queries dynamically utilizing an embedded Open Policy Agent component mapping Traefik/OAuth headers to enterprise PBAC policies synced seamlessly from the DMS. - **Schema Modification**: Rather than rejecting restricted queries outright, the proxy rewrites query boundaries. Forbidden metadata arrays are mathematically nullified (`\x00`), and enforced identities are permanently masked (`\xff`). - **Scatter-Gather Parallelization**: The proxy multicasts the validated query out to all designated SCCs concurrently and assembles their disparate success/timeout hashes into HTTP 207 Multi-Status payloads.

3.2.3 Message Bus Ingestor `hd_message_bus_ingester`

The engine role that turns a source into batches of fixed-width records and streams them to an SCC. - **Sources**: Kafka, RabbitMQ and MQTT as a broker consumer; a local directory or an S3-compatible object store one file at a time, reading JSON, JSON Lines, CSV, Parquet, ORC, Avro, Protocol Buffers and XML by extension. - **Zero-Allocation Parsing**: Every input decodes to one JSON record per row and is mapped onto the dataset schema without dynamic allocation; RowBinary input skips the parse entirely. - **Batching**: The batch id comes from the schema — a time field rounded to the time index from the Unix epoch, any field's value as it is, or the ingester's clock — and a closed batch is never reopened, so a file's records are ordered by that field before they are sent. Addresses arrive as text or integers and are stored as the `uint32` / `uint128` they are.

The full account — sources, formats, how batches are formed — is on the [ingestion page](#).

3.2.4 Storage Cluster Controller (SCC) `hd_storage_controller`

The high-availability gateway directly preceding hardware disks. - **Load Balancing**: The SCC acts as the definitive controller for the Storage Nodes within its Storage Cluster (SC). It prevents N-to-N connection pooling crises by accepting ingest/query operations and logically load-balancing them across the available SN infrastructure.

3.2.5 Storage Node (`hd_storage_node`)

The lowest-level physical footprint (C++). - **DPDK / SPDK**: Uses DPDK to poll the NIC and SPDK to stream byte buffers immediately to NVMe SSD queues. No Linux Interrupts or context switching. - **Heterogeneous Storage & OCF**: Implements advanced topologies like Open CAS Framework (OCF) caching, combining fast cache devices (e.g., Optane/NVMe) with slower core devices. - **SYCL & Crypto Offload**: Offloads hashing to SmartNICs/FPGAs (Bluefield, Alveo) and compiles complex filtering requests into SYCL instructions to evaluate across clustered GPUs (or CPUs) natively over local RAM.

3.2.6 BloomRF Filter Node (`hd_bloomrf`)

An optional engine role, at most one per Storage Cluster, that prunes batches out of a query before any Storage Node reads them. - **An SN that writes an index instead of records**: it joins the SCC's ingest broadcast, computes and verifies the same chain hash, and folds each batch into a per-column zone map and BloomRF held in memory. Nothing persists; the store rebuilds from the stream. - **One question at query time**: the SCC resolves batches from its own index as always, then asks the filter node which of them cannot match. The reply is always a subset, every uncertainty falls on the side of reading the batch, and the SDK checks every answer against the records that arrive. - **Three modes**, pushed by the DMS to every SCC at once: `off`, `passive` (consulted, recorded, not applied) and `active`.

The [filter node page](#) covers modes, provisioning, sizing and the metrics.

3.2.7 Query services (`hd_sql`, `hd_graphql`, `hd_ipflow_graphql`, `hd_mcp`, Data Explore)

Front doors onto the Proxy that speak protocols people already have, each carrying the SDK and evaluating the local part of a query with its engine. - **hd_sql**: a read-only SQL server answering PostgreSQL wire, Arrow Flight SQL and HTTP from one engine — one dataset per statement, DuckDB's dialect, predicates pushed to the Storage Nodes where they can be. - **hd_graphql**: a GraphQL schema generated from every dataset every Proxy serves, plus raw SQL, graph and vector mutations over the fetched records. - **hd_ipflow_graphql**: GraphQL shaped for the network-flow dataset — time series, top-N, scalar aggregates, paging. - **hd_mcp**: the Model Context Protocol for AI assistants, with a tool that fetches a bounded window and a second that re-queries the fetched records; **Data Explore** is the browser front end over it. - **Addresses convert at these**

boundaries and nowhere else: an address literal becomes the `uint32 / uint128` it is before the query leaves the service, and an address column is rendered on the way out. The SDK sees only integers.

The [querying page](#) maps the doors and the rules they share.

3.2.8 Software Development Kit (hd_sdk)

- **Local evaluation on the returned table:** once records land in the client as an Arrow table, the SDK can run further work over them locally instead of round-tripping to the cluster — SQL via an embedded DuckDB (`apply_sql / execute_to_sql`) and vector search via a locally-built USearch index (`execute_vector_search` , cosine distance, top-`k`). Graph analysis (`execute_graph`) is declared in the API but **not implemented** — it logs which graph engine it would use and returns nothing.
- **The table is the developer's:** every execute call ends in an Arrow table — the raw record table, the projected one, or the record batches as they land — and every language binding hands it over as-is, as an Arrow IPC stream where the language needs one. Nothing in it is translated.
- **The filter verdict is checked here:** the SDK is the only component that sees both the filter node's exclusions and the records that arrive, so it reports a false negative the moment a query completes. A client that dispatches its own query hands the Proxy's verdict to the SDK for that check to run.

3.3 Host Requirements for Storage Nodes with SPDK/DPDK

Storage Nodes running in bare-metal acceleration mode (SPDK NVMe + DPDK transport) require specific host-level kernel configuration that **cannot** be satisfied at container start time. These must be applied once per physical host before deploying any SN with PCI passthrough storage.

3.3.1 IOMMU

IOMMU must be enabled in the kernel command line **and** in the system firmware (BIOS/UEFI).

CPU Vendor	GRUB parameter
AMD	<code>amd_iommu=on iommu=pt</code>
Intel	<code>intel_iommu=on iommu=pt</code>

`iommu=pt` enables passthrough mode, which eliminates IOMMU translation overhead for VFIO-assigned devices compared to full-isolation mode.

Verify after reboot:

```
dmesg | grep -i iommu
ls /sys/class/iommu/
```

3.3.2 Hugepages — Both 2 MB and 1 GB Required

DPDK's EAL memory allocator requires a **2 MB hugetlbfs** mount. The 1 GB mount present by default is insufficient on its own — DPDK will log:

```
EAL: 1023 hugepages of size 2097152 reserved, but no mounted hugetlbfs found for that size
```

and fall back to a degraded allocation path, significantly reducing NVMe throughput.

Page size	Mount point	Used by	Reserve
2 MB	<code>/dev/hugepages</code>	DPDK EAL — HD engine transport (networking)	1024 pages = 2 GiB
1 GB	<code>/dev/hugepages1G</code>	SPDK — storage DMA and its ring buffers	4-8 pages

Recommended GRUB additions (merge with existing `GRUB_CMDLINE_LINUX_DEFAULT`):

```
default_hugepagesz=2M hugepagesz=2M hugepages=1024 hugepagesz=1G hugepages=4
```

3.3.3 CPU Isolation

For consistent, low-jitter benchmarking and production workloads, OS scheduler noise must be excluded from the cores assigned to the SN via `MASTER_CPUSSET`.

Recommended GRUB additions (adjust CPU range to match your topology):

```
isolcpus=2-15 rcu_nocbs=2-15 nohz_full=2-15
```

This reserves CPUs 0–1 (and their SMT siblings) for the OS, Docker daemon, and SCC. The SN occupies CPUs 2–15 via `MASTER_CPUSSET=2-15`.

Container cpuset Split Requirement

All HD components are always deployed as containers — even on dedicated bare-metal hardware — to align with modern data-centre management and orchestration systems (Docker Compose, Kubernetes, etc.).

`isolcpus` alone is not sufficient to protect DPDK poll-mode threads when running inside containers. A cgroup cpuset assignment that **explicitly lists** a CPU overrides `isolcpus` for every process in that cgroup, including the container runtime shim (`containerd-shim`, CRI-O shim). The shim runs in the host scheduler but inherits the container's cpuset; if DPDK cores appear in the container `cpuset`, the shim is legally scheduled on those cores.

The required pattern is a strict two-group split:

Group	Specification	Used by
Overflow cores	<code>cpuset</code> : in Docker Compose / K8s pod spec	Container runtime shim, control-plane threads, non-DPDK processes
DPDK cores	<code>MASTER_CPUSSET</code> environment variable	DPDK/SPDK poll threads, self-pinned via <code>pthread_setaffinity_np</code> at SN startup

The DPDK cores must **never** appear in the container `cpuset` field. The SN process self-pins its DPDK/SPDK threads to the `MASTER_CPUSSET` cores at startup, escaping the cgroup cpuset restriction. The container shim, which never calls `pthread_setaffinity_np`, stays confined to the overflow cores.

```
# Correct Docker Compose pattern
services:
  sn-1:
    cpuset: "7,15"           # Overflow cores only – shim stays here
    environment:
      MASTER_CPUSSET: "0-1,8-9" # DPDK cores – excluded from cpuset above
```

```
# ✗ Incorrect – DPDK cores appear in cpuset; shim can preempt poll threads
services:
  sn-1:
    cpuset: "0-1,7,8-9,15"
    environment:
      MASTER_CPUSSET: "0-1,8-9"
```

For full implementation details, profiling evidence, and Kubernetes equivalents, see `hd_sn/docs/user_guide.md` — Step 3a.

Per-Role Minimum CPU Core Requirements

The table below is per-component — DPDK/active cores against overflow cores. The installer works in whole instances instead, and its floors (SCC 6, SN 9, Ingestor 4) are the totals `fermi-hdi-configure` allocates and `validate-cluster` checks. The two measure different things; where you are sizing a host, use the installer's.

The table below documents the minimum logical CPU (thread) count required per service role and the recommended configuration for production deployments. These figures are derived from profiling data collected during HD benchmark runs.

Role	DPDK / Active Cores (MASTER_CPUSSET)	Overflow Cores (cpuset)	Notes
Storage Node (SN)	4	2 (separate physical core)	Overflow core must be on a different physical core than any DPDK core to prevent SMT sibling interference
Storage Cluster Controller (SCC)	2	1 (SMT sibling acceptable on constrained hosts)	SCC does not run a continuous DPDK PMD poll loop; idle evictions from <code>swapper</code> on the DPDK core are benign. On hosts with no free physical core, <code>cpuset == MASTER_CPUSSET</code> is acceptable.
Query Proxy (HD Proxy)	1-2	1 (shared with SN overflow acceptable)	Proxy is not DPDK-intensive; no strict overflow isolation required

Minimum host configuration for a single Storage Cluster (1 SCC + 3 SNs):

Scenario	Logical CPUs Required
Sandbox / development (as in <code>tests/sandbox_dms</code>)	16 (8 physical cores, SMT×2)
Production minimum (clean per-role overflow isolation)	20+ (10+ physical cores)
Production recommended (with per-SN isolated NUMA nodes)	32+ (2× NUMA, 16 cores each)

Note on SCC `cpuset == MASTER_CPUSSET`: On a 16-logical-CPU host where all cores are allocated, the SCC has no spare core for a separate overflow. Profiling (RC12) shows the resulting evictions are exclusively `swapper` idle-thread handoffs — not active preemption of DPDK work. This is safe because the SCC reader thread is not a spin-polling PMD; it sleeps between batches. If the host has ≥ 18 logical CPUs, assign 1 dedicated overflow core to the SCC to eliminate this edge case entirely.

3.3.4 Automated Host Setup

The supported path is `install/`, which records what a host should run in `cluster.yaml` and applies it: hugepage pools per NUMA node, `hugetlbfs` mounts, `vfiopci` bindings and the kernel command line. It reports before it changes anything, is idempotent, and never reboots.

```
install/fermi-hdi-discover      # inventory this host
install/fermi-hdi-configure    # decide what it runs
install/validate-cluster cluster.yaml # check that decision
sudo install/fermi-hdi-prepare cluster.yaml --apply # apply it
```

Core counts, hugepage sizing and the `cpuset` split described below are what `validate-cluster` enforces; `install/ADMIN_GUIDE.md` gives the whole model with the numbers.

`scripts/host_setup.sh` remains the by-hand tool, for working out what a machine can do or fixing one thing at a time:

```
# Show current state
sudo bash scripts/host_setup.sh status

# Configure hugepages + persistent /etc/fstab mount (safe, no reboot needed)
sudo bash scripts/host_setup.sh hugepages

# Bind an NVMe device to vfio-pci for SPDK passthrough
sudo bash scripts/host_setup.sh bind 0000:01:00.0

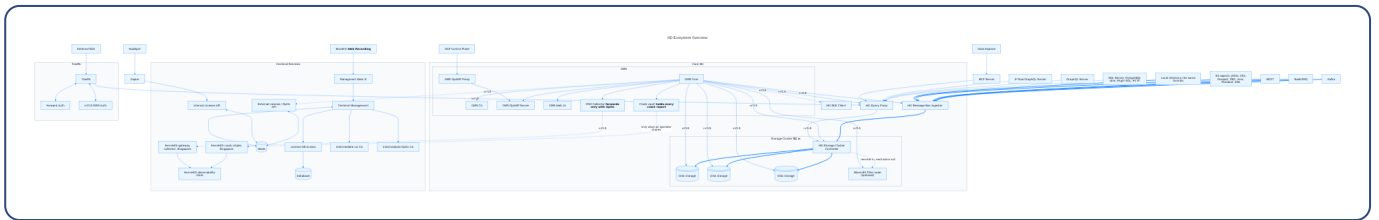
# Print the full recommended GRUB command line for this host
sudo bash scripts/host_setup.sh grub-advice
```

After making GRUB changes: `sudo update-grub && sudo reboot`

3.4 Further Reading

Topic	Document
FHDWP MTU, Jumbo Frame support, Docker/K8s CNI compatibility	docs/fhdwp_network_guide.md
The query front doors and the rules they share	docs/querying.md
Ingestion: sources, file formats, how batches are formed	docs/ingestion.md
The BloomRF filter node	docs/filter_node.md
SN configuration and storage provisioning	<code>hd_sn/docs/configuration_guide.md</code>
SN user guide and hardware requirements	<code>hd_sn/docs/user_guide.md</code>
End-to-end integration test guide	docs/end_to_end_test.md
Performance benchmark strategy	<code>BENCHMARKING.md</code>
Host setup script	<code>scripts/host_setup.sh</code>

4. HD Ecosystem Overview



5. Installation

5.1 HD — Administrator's Guide

Everything needed to stand up an HD instance and keep it standing: what the system is, what it is made of, what it needs from your hardware and network, and every option the installer offers.

This documents **HD v2.0.5**. Every image it names is `cr.fermihdi.io/hd/<service>:2.0.5`.

It assumes you administer Linux hosts and containers. It assumes nothing about FermiHDI. If you only want the commands, [README.md](#) beside this file is the short version; everything here is the reasoning behind it.

5.1.1 What HD is

HD is a record store built for one thing: taking in enormous volumes of small, uniform records and answering questions about them without the kernel getting in the way. Network packets bypass the kernel through DPDK; disk writes bypass the filesystem through SPDK, straight to NVMe queues. The work runs on cores nothing else is allowed to touch.

That single design choice is behind almost every requirement in this guide. Poll mode threads spin on dedicated cores and must not be preempted. Buffers are allocated up front in hugepages and must not be swapped. Devices are taken from the kernel and handed to the process. None of it can be arranged after a container starts, which is why installation is two distinct acts — **preparing a host**, then **running containers on it** — and why the first one changes the kernel command line and usually needs a reboot.

One instance is a deployment: one DMS, one or more Storage Clusters, and the edge and access services around them. Everything in it trusts one certificate authority and registers with one DMS.

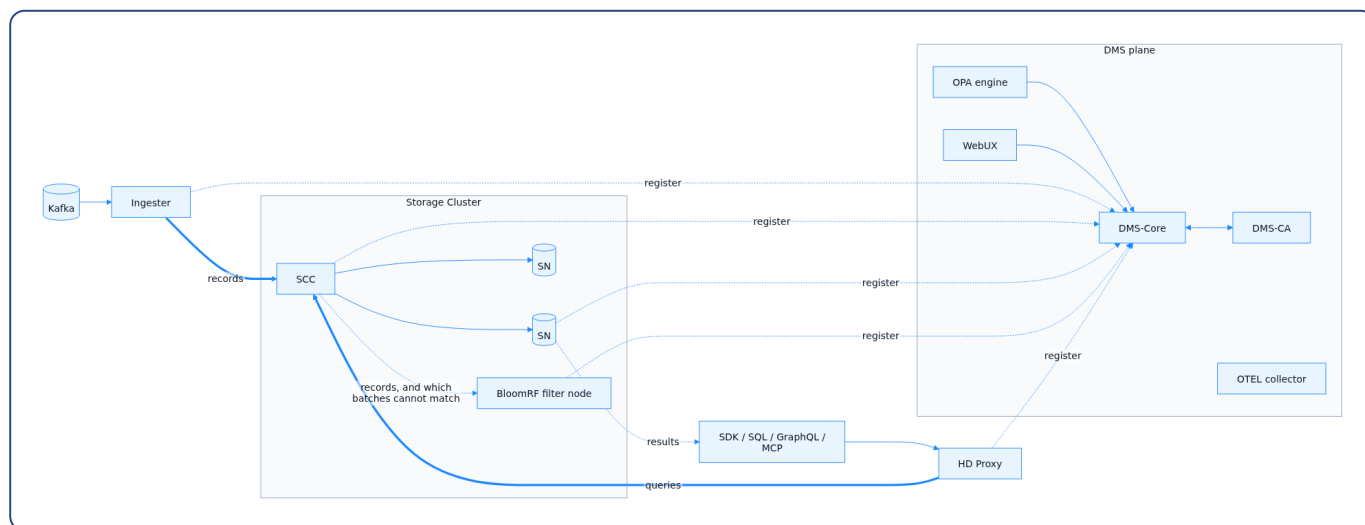
5.1.2 What it is made of

Component	Language	Does
DMS-Core	Go	The authority. Registers nodes, issues their configuration, holds the topology, the schema and the license
DMS-CA	Go	The certificate authority. Signs every identity in the deployment
DMS-OPA-Engine	Go	Compiles and serves the policy bundles the Proxy enforces
DMS-OPA-Adapter	Go	Optional. Polls an external OPA control plane and hands bundles to the engine
DMS-WebUX	Astro + Node	The admin UI, and the only thing that can apply a license interactively
SCC	C++ / HD engine	Storage Cluster Controller. The gateway in front of a set of Storage Nodes: fans writes out to them, load-balances queries across them
Storage Node (SN)	C++ / HD engine	Where records live. DPDK on the wire, SPDK to the NVMe
BloomRF filter node	C++ / HD engine	Optional, at most one per storage cluster. Takes the SCC's ingest broadcast exactly as an SN does and keeps a small filter per batch instead of the records, so at query time the SCC can leave unread the batches that cannot match. Holds no device and persists nothing: its store is rebuilt from the stream
Ingestor	C++ / HD engine	Reads a message bus, parses records without allocating, and streams them to an SCC. Kafka is what the installer wires; the ingester itself also reads RabbitMQ and object stores
HD Proxy	Go	The query front door. Enforces policy, rewrites queries it must restrict, scatters them to the SCCs and gathers the results
hd_graphql / hd_mcp	Go + the SDK	GraphQL and MCP front ends onto the Proxy
hd_sql	Go + the SDK	A read-only SQL front end onto the Proxy: PostgreSQL wire, Arrow Flight SQL and HTTP from one engine. Run from its published image beside the Proxy; the cluster file has no role for it
Data explorer	—	Optional web UI over the Proxy
OTEL collector	Go + OpenTelemetry Collector	Where every service sends traces and metrics. Managed by DMS-Core, which renders its exporters from the WebUX and adds the opt-in pipeline to FermiHDI only when an administrator turns sharing on
Crash vault	Go	Beside DMS-Core. Receives crash reports from the engine roles, DMS-Core, the Proxy and the WebUX, symbolicates them, and holds them on the DMS host; nothing leaves unless an operator shares it

Two shapes of service, and the difference decides everything about placement:

- **Engine roles** — SCC, SN, the filter node, Ingestor, and the GraphQL and MCP front ends. Pinned cores, hugepages, one reactor per core. These are the ones with hard requirements.
- **Go and Node services** — the DMS plane and the Proxy. Ordinary processes with ordinary needs.

5.1.3 How it fits together



Three planes, and a node works out its own address on each by matching its interfaces against the CIDRs you declare:

Plane	Carries	Typical
Control	Registration, configuration, cluster coordination	10.147.0.0/24
Data	Records in, results out — the FHDWP wire protocol	10.150.0.0/24
Observability	Traces, metrics, health	10.146.0.0/24

They can share one interface on a small deployment. On a multi-homed host they must not be left unset, or each node picks an interface by itself and half of them will pick differently.

The three flows:

- Ingest.** Kafka → Ingester (parses, slices into fixed-size records) → SCC over FHDWP → fanned out to every SN in the cluster. Each SN holds a full replica of the cluster's dataset. A cluster with a filter node fans the same stream out to it too; it verifies each batch's hash with the SCC as an SN does, then keeps a filter per batch and discards the records.
- Query.** Client → a front door that authenticates the caller and sets `X-Forwarded-*` headers → Proxy, which evaluates policy against those headers and rewrites the query rather than refusing it — nullifying forbidden fields, masking enforced identities — then scatters it to every SCC in parallel. Each SCC fans out to its SNs. **The records do not come back through the Proxy:** the SNs stream them to the client over the data plane, and the Proxy answers with a manifest of which batches were found and which SCC, if any, timed out (`HTTP 207`). A query that returns 207 is a partial answer, not a failure. An SCC with a filter node first asks it which of the batches in the window cannot match the query's predicates, and leaves those unread; the Proxy's manifest lists them as excluded. The filter is allowed no false negatives, and every client checks: a batch the filter excluded that delivers a record anyway is logged as an error by the SDK.
- Control.** Every node presents a bootstrap token to DMS-Core at startup, gets a signed certificate and its runtime configuration back — record size, which SCC to talk to, which devices to use — and reports health afterwards. **A node holds no configuration of its own.** What you set in the environment is identity: who am I, where is the DMS, what token proves it.

5.1.4 Identity, trust and licensing

Every control-plane connection in an HD deployment uses mutual TLS. The data plane (FHDWP) is not encrypted; see [Networking](#). Every certificate in the deployment descends from one chain:

```

FermiHDI Root CA
├── Intermediate CA
│   └── DMS-CA ICA
        (issued to you)
        (this deployment's signing certificate)
  
```

```
└─ dms-core, dms-webux, the OPA engine
└─ every SCC, SN, filter node, Ingester and Proxy that registers
```

- The **trust bundle** (`FERMIHDI_CA_CERT_PEM`) is those first three concatenated. Every service verifies against it. It is not secret.
- The **DMS-CA's ICA certificate and key** are what the CA presents and signs with. The key is the most sensitive thing in the deployment.
- **Bootstrap tokens** are how a node proves, once, that it is allowed to ask for a certificate. One per node is the convention; a single shared token works and is weaker.
- The **license** is encrypted to your identity key and gates how many nodes DMS-Core will admit. Without one it starts and admits nobody. With one it cannot parse, it exits at startup — that failure is louder on purpose. Each node type draws on its own element of the licence and on the `HD_ANY` pool otherwise; a filter node is the `HD_BLOOMRF` element.

There are two ways to get the certificates in place, and choosing between them is the first real decision of an install:

Pre-provisioned (what this installer does). You already hold the ICA certificate and key, DMS-Core's certificate, the WebUX's certificates and the license. The installer puts them in place and everything comes up ready.

Day-0 bootstrap. The DMS-CA generates its own key and a CSR, you send that CSR to FermiHDI, and the license that comes back carries the ICA certificate issued against it. Applying that license through the WebUX installs the ICA and DMS-Core's identity out of it. A license supplied as an environment variable does **not** do this — it is verified and stored, and nothing else. If all you have is a license, install without the certificates and apply it through the WebUX afterwards.

Renewal. A licence carries a term, the DMS-CA's ICA certificate carries an expiry, and a node certificate the DMS-CA signs is good for 30 days, never past the ICA's own expiry. All three renew without a rebuild and without a restart:

- **The licence** is renewed or replaced on the WebUX's License page, online or offline. A renewed licence is issued against the DMS-CA's existing key, so the renewed ICA certificate keeps every certificate already signed valid; applying it installs the ICA on the DMS-CA, re-enrols the DMS's own services, and carries the new ICA to every node in its next configuration push. A licence that changes the node types or counts takes effect for the next provisioning; running nodes are never evicted.
- **Node certificates** renew themselves. Within ten days of expiry (`FERMIHDI_CERT_RENEW_DAYS`), at startup and once a day, a node presents its current certificate and receives a new one; it needs no token and no administrator. A node whose certificate has already expired cannot do that: it registers again, with a token or an approval, exactly as a new node. A node kept off for longer than a certificate's lifetime is in that position.
- **A running process takes renewed material in place.** Its listeners swap certificate and trust without stopping, its outgoing connections use the new material from the next one opened, and connections already open keep what they were opened with until they reconnect. Nothing is restarted.

One requirement catches people either way: **DMS-Core's certificate must carry, in its SAN, the address the nodes are given.** Nodes verify what they connect to. A certificate naming only `dms-core` works inside the DMS's own network and fails every registration from outside it.

5.1.5 Ports

Port	Service	Spoken by	Leaves the host?
6986	DMS-Core southbound, mTLS	every node registering, and telemetry heartbeats	yes
6986	SCC and SN control plane	SCC ↔ SN, Proxy → SCC	yes
6987	Data plane (FHDWP)	Ingester → SCC, SCC → SN, SN → client	yes
6988/udp	NORM multicast	SCC	data plane
6997	Filter node, batch pruning (gRPC)	SCC → filter node	control plane
5432 / 31337 / 8081	hd_sql: PostgreSQL wire, Arrow Flight SQL, HTTP	SQL clients, through the front door	yes , when it is run
6985	DMS-Core internal management	WebUX and DMS-CA only	no
6985	DMS-CA signing API	DMS-Core only	no
6985	OPA renewal listener	DMS-CA → OPA engine	no
6984	DMS-Core Day-0 bootstrap	DMS-CA, during bootstrap only	no
8080	Health and metrics, plain HTTP	your orchestrator	host-local
8181	OPA engine REST	internal	no
3000	WebUX, HTTPS	the DMS Traefik in front of it	no
8081 / 8082	WebUX and Traefik dashboard, as published	operators	loopback by default
4317 / 4318	OTEL collector, gRPC and HTTP	every service in the deployment	yes

Two ports called 6985 in the same deployment is deliberate — DMS-Core and DMS-CA each own one, and they never share a network namespace. It is also why the DMS runs on its own container networks rather than the host's.

5.1.6 What you need before you start

Hosts

Per engine instance, the floor is a hard one: below it the resource controller refuses to start and the process exits.

Role	Cores the installer allocates	Refused below	2 MB pages / instance	1 GB pages
SCC	6	6	300 (600 MB)	2
Storage Node	9	9	4140 (8.1 GB)	4 per NVMe device
Filter node	2	2	300	—
Ingestor	4	2	300	—
GraphQL	2	1	300	—
MCP	2	1	300	—
Proxy	1	1	—	—
DMS	1	1	—	—

The 2 MB figure is the role's engine arena plus 88 MB of mbuf pools and EAL overhead. The Storage Node's is large because its arena scales with transmit shard count. The 1 GB pool exists only for SPDK's NVMe DMA — an SN whose storage is all kernel-path needs none of it.

Also reserve **5 GB per host** for the kernel, page cache and everything outside the engine arenas. `fermiHDI-configure` and `validate-cluster` both compute against that figure, and the validator refuses a host whose pools would not leave it.

Two placement rules the validator enforces, and the reasons they exist:

- **An instance's cores must come from one NUMA node.** A cpuset straddling two nodes makes every buffer access a cross-socket trip, and a poll-mode thread pays that on every iteration.
- **container_cpuset must not overlap cpuset.** The poll cores go in `cpuset` and nowhere else; the container's cgroup gets the overflow cores only. The container runtime shim inherits that cgroup, and if a poll core is in it the shim will be scheduled there — profiling measured over 300,000 shim evictions in a 90-second window that way. `isolcpus` does not prevent it, because an explicit cgroup cpuset overrides `isolcpus`.

Firmware and kernel

- **IOMMU** enabled in BIOS/UEFI *and* on the kernel command line, for any host binding a device to VFIO. `fermiHDI-prepare` writes the kernel side; the firmware side is yours.
- **1 GB hugepages, `isolcpus` and IOMMU are boot-time only.** Expect one reboot per host during preparation. Nothing reboots a host for you.

Network

- The network itself is yours: node connectivity, a CNI on Kubernetes, and any encryption of the data plane. [Networking](#) sets out what to provide.
- The three plane CIDRs, or one CIDR used for all three.
- A DNS name or address for the DMS that resolves from every host — and that appears in DMS-Core's certificate SAN.
- Jumbo frames are worth having on the data plane; FHDWP sizes to a 1200-byte MTU by default and 8500 with jumbo.
- Clients must reach the Storage Nodes on the data plane. Query *results* are streamed to the client directly by the SNs, not relayed by the Proxy — a network that only lets clients reach the Proxy will answer every query with an empty result and no error.

Two front doors

Nothing in HD authenticates a person. The Proxy, the query services, Data Explore and the DMS WebUX trust the `X-Forwarded-User`, `-Email`, `-Groups` and `-Role` headers on the requests that reach them, and refuse or admit on those alone. Two reverse proxies set them: the **DMS front door**, through which administrators reach the WebUX, and the **query front door**, through which clients and the query services reach the Proxy. Both are yours to provide: the lab and the reference deployment use Traefik with a forward-auth step, but any ingress that authenticates against your identity provider, sets the headers and strips client-supplied ones will do, with or without a forward-auth mechanism. The query front door also presents a client certificate to the Proxy, whose CA the Proxy is given as `FERMIHDI_NORTHBOUND_CA_PEM`; the DMS front door is never given a DMS-issued certificate. **Neither the Ansible layer nor the chart deploys either front door**; `docs/examples/ansible_example` is the reference for both. Until they exist, keep the WebUX and the Proxy on a loopback or a management network, which is where the installer publishes them by default. The contract in full, service by service, is on the [front doors page](#).

Software on each host

- Docker, and the Python Docker bindings if you use the Ansible layer.
- `python3` with PyYAML — the installer's own scripts need it.

On the machine you install *from*

- `python3` + PyYAML, and Ansible ≥ 2.14 with `community.docker` for a multi-host install, or `helm` for Kubernetes.
- SSH with `sudo` to every host.

Credentials and material

What	Needed for	Where it goes
Registry username and password	pulling images	<code>FERMIHDI_REGISTRY_PASSWORD</code>
Bootstrap tokens	every node's first handshake	<code>FERMIHDI_BOOTSTRAP_TOKEN_<NODE></code>
Trust bundle	verifying the DMS	<code>FERMIHDI_CA_CERT_PEM</code>
DMS-CA ICA certificate + key	signing identities	file, or <code>FERMIHDI_DMS_CA_CERT_PEM / _KEY_PEM</code>
DMS-Core certificate + key	its TLS identity	<code>FERMIHDI_DMS_CORE_CERT_PEM / _KEY_PEM</code>
WebUX server and client certificates	the admin UI	<code>FERMIHDI_DMS_WEBUX_*</code>
Encryption and identity keys	DMS-Core's stored state and the license	<code>FERMIHDI_DMS_ENCRYPTION_KEY / _IDENTITY_KEY</code>
License	admitting nodes	file, or <code>FERMIHDI_DMS_LICENSE_PAYLOAD</code>
Dataset schema	what a record looks like	file, named in <code>cluster.yaml</code>

5.1.7 The cluster file

One file describes the whole installation. Every tool here reads it: the validator, the host preparer, the Ansible inventory, the Kubernetes generator. It holds **no secrets** — only the names of the variables and files they come from.

```
version: 1
cluster_name: hd-prod-1

registry:
  url: cr.fermihdi.io
  username: deploy
  password_env: FERMIHDI_REGISTRY_PASSWORD # read at run time, never stored

dms:
```

```

url: https://10.147.0.5:6986 # every node is handed this verbatim
bootstrap_token_env: FERMIHDI_BOOTSTRAP_TOKEN
license_file: certs/license.json # optional, all four of these
ica_cert_file: certs/dms-ca.crt
ica_key_file: certs/dms-ca.key
schema_file: certs/dataset_schema.json

images:
  tag: "2.0.5"
  scc: cr.fermihdi.io/hd/scc # per role; the tag above applies
  dms-core: cr.fermihdi.io/hd/dms-core

networks:
  control_plane: 10.147.0.0/24
  data_plane: 10.150.0.0/24
  observability: 10.146.0.0/24

hosts:
  - name: sc-host-1
    address: 10.147.0.10 # what Ansible connects to
    ssh_user: root
    ssh_key: ~/.ssh/fermihdi_deploy
    roles:
      - type: scc # scc sn bloomrf ingester mcp graphql proxy dms
        count: 1
        dpdk: true # optional; POSIX networking otherwise
        instances:
          - name: sc-1-scc
            cpuset: "2-7" # MASTER_CPUSET: the poll cores
            container_cpuset: "0-1" # overflow only, must not overlap
            numa_node: 0
            nic: { pci: "0000:31:00.0", driver: vfio, kind: nic, iommu_group: 12 }
      - type: sn
        count: 1
        dpdk: true
        instances:
          - name: sc-1-sn-1
            cpuset: "8-18"
            container_cpuset: "0-1"
            storage:
              - { pci: "0000:61:00.0", driver: vfio, kind: nvme, iommu_group: 40 }
    host_prep: # what fermihdi-prepare applies
      hugepages_2m: 4440
      hugepages_1g: 6
      numa: [{ node: 0, hugepages_2m: 4440, hugepages_1g: 6, memory_mb: 262144 }]
      memory_total_mb: 262144
      iommu: true
      isolcpus: "2-18" # also applied as rcu_nocbs and nohz_full
      vfio_bind: ["0000:31:00.0", "0000:61:00.0"]
      grub: { apply: true } # backs up, validates, never reboots

```

Three things to know about the file's shape:

- **PCI addresses must be quoted.** YAML 1.1 reads `0000:31:00.0` as a sexagesimal number — but only when every segment is below 60, so it silently works for bus 61 and breaks for bus 31. The validator catches it first.
- **Devices carry their driver.** `vfio` means the kernel hands the device to the process; `kernel` means an ordinary block device or a network target. Two or more VFIO devices on one SN form a RAID-0 set.
- **A filter node is named after its cluster.** `sc-1-bloomrf` serves `sc-1-scc`. The file cannot say which storage cluster a node belongs to — that is set where the node is provisioned in the DMS — so the validator checks the name instead, and warns about a second filter node for the same cluster, which the SCC would never consult.

Full field reference: [schema/cluster.schema.json](#), which is what the validator checks against.

5.1.8 A deployment you can read: the sandbox

Before building one, it is worth running one. [tests/sandbox_dms](#) is a complete HD instance on a single host — the whole DMS plane, a Storage Cluster, the edge and access services, a Kafka bus with traffic generators, and Grafana over the top. It is what FermiHDI ships for evaluation and benchmarking, and it is the deployment this installer was built from: where a document and the sandbox disagree about how a service is configured, the sandbox is right, because it is the one that runs every day.

```

cd tests/sandbox_dms
sudo bash sandbox_host_setup.sh # hugepages, vfio binding, GRUB advice
# reboot if it asks, then
bash start_sandbox.sh # or --num-sns 2, --standalone, -t 2.0.5

```

It picks the image variant from the host by default — the bare tag is built for x86-64-v4 and faults on a CPU without AVX-512 — so `--type` is only needed to override that: `ubuntu` for the thin development images, `metrics` for the SCC and SN builds with metering compiled in.

It has its own equivalents of the four steps in this guide, and reading them next to each other is the fastest way to understand what the installer does:

Sandbox	This installer	Both do
<code>configure_sandbox.py</code>	<code>fermihdi-discover</code> + <code>fermihdi-configure</code>	NUMA-aware core allocation, storage device mapping, GRUB advice
<code>sandbox_host_setup.sh</code>	<code>fermihdi-prepare</code>	hugepages, hugetlbfs mounts, <code>vfiopci</code> binding
<code>docker-compose.override.yml</code>	<code>cluster.yml</code>	the placement decisions, written down
<code>compose.yml</code>	<code>ansible/</code> and <code>helm/fermihdi</code>	actually running the containers
<code>generate_certs_and_license.sh</code>	your PKI, or FermiHDI's	the certificate chain and a license

Three things in it are worth reading directly:

- `compose.yml` — every service with its full environment. This is the reference the Ansible layer and the Helm chart are checked against.
- `generate_certs_and_license.sh` — builds the whole chain from a root CA down: the DMS-CA's ICA, DMS-Core's certificate with its SANs, the WebUX's server and client identities, and a signed, encrypted 30-day license. It is the clearest statement anywhere of exactly what material an install needs and how the pieces relate.
- `cpu_allocation_matrix.md` and the resource recommendations beside it — the sizing model with real numbers behind it.

Where the sandbox is deliberately not production

Copying from it wholesale is the one mistake it invites:

The sandbox	A real deployment
One host, Docker bridge networks with fixed IPs	Real hosts, real interfaces, the plane CIDRs you declare
<code>mock-forward-auth</code> accepts everything and returns <code>admin</code>	Your SSO in front of the Proxy and the WebUX
A self-signed chain and a 30-day license generated locally	Your PKI, and a license issued to you
Storage nodes on tmpfs or a file, in the default profile	NVMe bound to VFIO
Every service on one machine, sharing cores	Placement computed per host, poll cores isolated

The full guide to running it — sizing, benchmarking, profiling, every option of `start_sandbox.sh` — is [tests/sandbox_dms/sandbox_guide.md](#).

5.1.9 Installing

Four steps, in the same order however many hosts you have:

```
fermihdi-discover → fermihdi-configure → validate-cluster → fermihdi-prepare
  what is here      what it should run    is that coherent    make it so
```

Only the last one changes anything, and only when told to twice — a bare run is a dry run. Then the containers go on top.

On Talos Linux, the first and last of those four have API-driven twins (`fermihdi-talos-discover`, `fermihdi-talos-prepare`) because a Talos machine has no shell to run anything on, and the middle two are used unchanged. The order and the reasoning are identical; what differs is that the kernel-argument half of `host_prep` becomes a boot image rather than a GRUB edit, so applying it reboots the machine. [TALOS.md](#) is the whole of that path.

Path A — one host, by hand

```
install/fermihdi-discover           # read-only inventory
install/fermihdi-configure          # interactive; writes ./cluster.yaml
install/validate-cluster cluster.yaml # refuses what will not work
sudo install/fermihdi-prepare cluster.yaml # dry run: prints what would change
sudo install/fermihdi-prepare cluster.yaml --apply
# reboot if it says to, then:
install/fermihdi-prepare cluster.yaml --verify
```

Then deploy the containers with the Ansible layer pointed at that one host, or by hand from the environment table in [ansible/README.md](#).

Path B — several hosts, with Ansible

Run `fermihdi-configure` once per host, appending each to one file:

```
install/fermihdi-configure -o cluster.yaml # first host
install/fermihdi-configure --append cluster.yaml # each one after
```

Then, from the machine that can reach them all:

```
cd install/ansible
export FERMIHDI_REGISTRY_PASSWORD=... FERMIHDI_CA_CERT_PEM="$(cat bundle.pem)" ...
ansible-playbook site.yml # prepare every host, the DMS, then the nodes
```

There is no inventory file — it is generated from `cluster.yaml`, so there is nothing to keep in step. One group per role, so `--limit fermihdi_sn` means exactly what it says.

Playbook	Does
<code>prepare.yml</code>	validates the file, then <code>fermihdi-prepare --apply</code> on every host
<code>verify.yml</code>	<code>fermihdi-prepare --verify</code> everywhere; changes nothing
<code>dms.yml</code>	the DMS plane, on the host whose block declares a <code>dms</code> role
<code>deploy.yml</code>	the node containers each host's block calls for
<code>site.yml</code>	all three, in the order the nodes need

`--check` on `prepare.yml` is a real dry run. `-e fermihdi_reboot=true` lets it reboot, one host at a time — a cluster that reboots all of its Storage Nodes at once is a cluster that has lost quorum.

Everything a deployment might change lives in `ansible/group_vars/all.yml`: the message bus the Ingestor reads, the forward-auth service in front of the WebUX, the collector, per-role core splits, image namespace, data directories. It is meant to be read.

Path C — Kubernetes

```
install/fermihdi-k8s cluster.yaml -o values.yaml
helm install fermihdi install/helm/fermihdi -f values.yaml
```

The generator maps roles to workloads, instance counts to replicas, cpuset widths to CPU requests, and the engine model to memory and hugepage requests. What cannot cross — DPDK, device passthrough, the cpuset split, the 1 GB pool — is written into the generated file's header rather than dropped quietly.

Option	Does
<code>-o PATH</code>	write the values file
<code>--secrets PATH</code>	also write a 0600 values file holding keys and tokens
<code>--manifests</code>	render the chart and print the manifests instead
<code>--chart PATH</code>	render a chart other than the bundled one
<code>--domain</code> , <code>--storage-class</code>	ingress domain, storage class for every PVC

A cluster file with no `dms` role is read as having an external DMS: the host from `dms.url` goes into the reference Services the nodes resolve DMS-Core through, and the DMS components switch off. See [helm/fermihdi/README.md](#) for what differs from bare metal.

The filter node

A storage cluster runs with or without one, and the common case is without. Adding one is three steps beyond the role in the cluster file, and their order matters:

1. **Provision it in the DMS**, as a *BloomRF Filter Node* in the SCC's storage cluster, before it starts: on the WebUX's Nodes page, *Provision New Node*, exactly as for a Storage Node, with *Bootstrap Token* as the security mode. Export the token the dialog shows as the node's own variable, `FERMIHDI_BOOTSTRAP_TOKEN_SC_1_BLOOMRF`. (A script can do the same against the DMS management API: `POST /v1/nodes/provision` with `node_type` `bloomrf`, the `instance_id`, the `sc_id` and `generate_token`.)
2. **Deploy it before, or together with, the SCC**. The SCC takes its relay list once, when its data plane starts, so the DMS holds that list back until every provisioned Storage Node *and* filter node in the cluster is ready. Provision only what you will run: a provisioned filter node that never starts holds the whole cluster's ingest with it.
3. **Choose the mode**, or leave it. Once a filter node is provisioned the DMS pushes `active` to the cluster's SCC and its exclusions are applied. Name `passive` on the deployment record to have the SCC consult the filter and record what it would exclude without acting on it, or `off` to stop consulting it: the *Query filter* setting on the deployment's card on the WebUX's Deployments page, which pushes the choice to every SCC of the instance at once. The Proxy reports the mode in force in every query's manifest.

To add one to a cluster that is already running, do the three steps and then restart the SCC; to remove one, decommission it in the DMS and restart the SCC. What the node does, how big to make it and how to read its metrics are in the [filter node page of the manual](#).

Every script, every option

Command	Options
<code>fermihdi-discover</code>	<code>-o FILE</code> , <code>--format=tsv</code> , <code>--check-deps</code>
<code>fermihdi-configure</code>	<code>-o FILE</code> , <code>--append FILE</code> , <code>--facts FILE</code>
<code>validate-cluster</code>	<code>[cluster.yaml]</code>
<code>fermihdi-prepare</code>	<code>--apply</code> , <code>--verify</code> , <code>--host NAME</code> , <code>--no-grub</code>
<code>fermihdi-k8s</code>	<code>-o</code> , <code>--secrets</code> , <code>--manifests</code> , <code>--chart</code> , <code>--domain</code> , <code>--storage-class</code>

`fermihdi-prepare` exits **0** when done or when there was nothing to do, **1** on error, and **2** when it applied something that needs a reboot — which is what makes it safe to drive from a playbook.

Agent Skills

Every tool on this page also has an Agent Skill: an instruction set an AI coding assistant (Claude Code and compatible tools) loads to drive it conversationally instead of by hand. Published individually, unpinned and under each version, at downloads.fermihdi.io/skills/<name>/ — no source checkout, and no separate download beyond what a skill itself fetches.

Skill	Covers
<code>fermihdi-hd-install</code>	Provision a deployment in any topology — Kubernetes/Helm, bare-metal Ansible, bare-Docker, or the sandbox — from the published installer and registry images
<code>fermihdi-hd-sandbox</code>	Install and run the single-host evaluation sandbox, end to end, including its own test and benchmark suite
<code>fermihdi-hd-troubleshoot</code>	Symptom → root cause → fix for a running deployment: 0-row queries, slow ingest, an SN that won't start, rock/pebble startup problems
<code>fermihdi-hd-plan</code>	Probe one or more hosts and recommend a topology, storage-node count, hugepage sizing and CPU/NUMA placement, before installing anything
<code>fermihdi-hd-schema-build</code>	Build an HD dataset schema from an existing database's DDL or a corpus of files, and validate it
<code>fermihdi-hd-dry-run</code>	Verify a data source and a dataset schema are actually compatible with the ingester's <code>--dry-run</code> mode — no deployment, no licence needed
<code>fermihdi-hd-ingest</code>	Configure an ongoing ingestion pipeline through any supported source — Kafka, RabbitMQ, MQTT, a file directory, or an S3-compatible bucket
<code>fermihdi-hd-migrate</code>	A one-time bulk load or database migration into HD
<code>fermihdi-hd-mcp</code>	Set up and query a running HD instance through the Model Context Protocol

The environment

Read on the machine running the install, never written to a host:

```
# always
export FERMIHDI_REGISTRY_PASSWORD=...
export FERMIHDI_CA_CERT_PEM="$(cat dms-ca-bundle.pem)"
export FERMIHDI_BOOTSTRAP_TOKEN_SC_1_SN_1=... # per node: name, upper-cased,
export FERMIHDI_BOOTSTRAP_TOKEN=...          # non-alphanumerics to _
                                              # ... or one shared token

# deploying the DMS as well
export FERMIHDI_DMS_CA_CERT_PEM=... FERMIHDI_DMS_CA_KEY_PEM=...
export FERMIHDI_DMS_CORE_CERT_PEM=... FERMIHDI_DMS_CORE_KEY_PEM=...
export FERMIHDI_DMS_WEBUX_CERT_PEM=... FERMIHDI_DMS_WEBUX_KEY_PEM=...
export FERMIHDI_DMS_WEBUX_FE_CERT_PEM=... FERMIHDI_DMS_WEBUX_FE_KEY_PEM=...
export FERMIHDI_DMS_ENCRYPTION_KEY=... FERMIHDI_DMS_IDENTITY_KEY=...
export FERMIHDI_DMS_FE_TOKEN=... FERMIHDI_BOOTSTRAP_TOKEN_DMS_OPA_ENGINE=...
export FERMIHDI_DMS_LICENSE_PAYLOAD=... # or dms.license_file
export FERMIHDI_DMS_ICA_CHAIN_PEM=...   # optional
```

The DMS-CA's certificate and key, and the license, can equally be named as files in `cluster.yaml` — `fermihdi-configure` asks for them on a DMS host. A file named there wins over the environment, and the run says which it used.

`ansible-playbook dms.yaml` refuses to start with any required one missing, and names every one it could not find.

5.1.10 After the install

Check it came up. `ansible-playbook verify.yaml` re-checks every host against its definition — the pass to run after a reboot, and the first one to run when a node misbehaves for reasons nobody can place. A host that lost its hugepages to a kernel upgrade looks fine until the reactors fail to reserve their arenas.

Apply a license, if you installed without one: through the WebUX. That path also installs the DMS-CA's ICA and DMS-Core's identity out of the license, which the environment variable does not.

Add a node. Run `fermihdi-configure --append cluster.yaml` on the new host, validate, `prepare.yaml --limit new-host`, then `deploy.yaml --limit new-host`. Nothing else needs to know: the node registers itself.

Add a filter node to a running cluster: provision it in the DMS, deploy it, wait for it to report ready, then restart the SCC — an SCC takes its relay list once, when its data plane starts.

Upgrade. Change `images.tag` in `cluster.yaml` and re-run `deploy.yaml`. The containers are recreated with the new image; host preparation is unaffected. Do the DMS first if the release notes say the wire protocol moved.

What is stateful, and therefore what to back up:

Path	Holds
<code>/opt/fermihdi/dms/dms-core/storage</code>	the node registry, deployments, the license
<code>/opt/fermihdi/dms/dms-ca/storage</code>	issued-certificate state
an SN's storage devices	the records themselves — every SN in a cluster holds a full replica

Reboots stay yours. The tooling never takes a host down on its own; when a change needs one it says so and exits 2.

5.1.11 When something is wrong

Symptom	Usually
A node registers, then nothing reaches it	Plane CIDRs unset or wrong; it reported an address on the wrong interface
Every registration fails TLS	DMS-Core's certificate SAN does not carry the address in <code>dms.url</code>
Nodes register and are never admitted	No license, or one that admits fewer nodes than you have
DMS-Core exits at startup	An unparseable licence payload, or no <code>FERMIHDI_DMS_ENCRYPTION_KEY</code>
A reactor fails to reserve memory	Hugepages missing — check <code>verify.yaml</code> , then the kernel command line after any upgrade
Throughput collapses under load, no errors	A poll core inside <code>container_cpuset</code> : the runtime shim is preempting it
SPDK fails at attach, looking like a driver fault	The container was handed the wrong <code>/dev/vfio</code> node — IOMMU groups move under a kernel upgrade
Services healthy, no traces anywhere	<code>OTEL_EXPORTER_OTLP_ENDPOINT</code> unset. <code>FERMIHDI_OTEL_URL</code> is a routing probe and configures no exporter
The WebUX answers 500 on every request	A forward-auth middleware pointed at an authenticator that is not there
The engine dies immediately in Kubernetes	No <code>IPC_LOCK</code> : a container's default 64 KB memlock stops it pinning its arena
Ingest never starts after a filter node was provisioned	The DMS holds the SCC's targets until every provisioned SN and filter node is ready. Start the filter node, or decommission it
The filter node is up and excludes nothing; queries report <code>bloomrf_mode: off</code>	The SCC's data plane started before the node was ready, or the deployment record names <code>off</code> . Restart the SCC once the node reports ready
A query hangs to its timeout on a batch the manifest lists as excluded	The client waited for a batch nothing will stream. Clients on the current SDK drop excluded batches from their wait; upgrade the client

`docker logs <container>` is worth reading first — the services log startup and errors to stdout, and everything else to the collector.

5.1.12 Further reading

Where	What
README.md	this directory, in brief
ansible/README.md	the multi-host layer, and every secret it needs
helm/fermihdi/README.md	Kubernetes, and what differs there
TALOS.md	installing onto Talos Linux, which has no shell and no SSH
NETWORKING.md	the network you provide: front doors, node connectivity, encryption
schema/cluster.schema.json	every field of the cluster file
../docs/architecture.md	the system in depth
../docs/system_operators_manual.md	lifecycles and recovery
../docs/filter_node.md	the BloomRF filter node: modes, sizing, verification
../docs/webux_guide.md	the DMS WebUX, page by page: licence, deployments, nodes, policy
../docs/front_door.md	the two front doors: what they authenticate, the identity headers, what you provide
../docs/ingestion.md	the ingester: sources, file formats, how batches are formed
../docs/querying.md	the query services: SQL, GraphQL, MCP, and the SDK
../docs/fhdwp_network_guide.md	the wire protocol
../tests/sandbox_dms/	a complete working deployment to read

5.2 install

Everything needed to turn a set of bare machines into a FermiHDI cluster.

New to HD, or installing one for the first time? [ADMIN_GUIDE.md](#) is the long form: what the system is, what it is made of, what it needs from your hardware and network, and every option these tools take. This file is the map of the directory.

Installing onto Talos Linux? [TALOS.md](#) is the map of that path. The file below and every tool in it still apply — Talos changes only the last mile, and one thing about it is worth knowing before you start: kernel arguments are not machine configuration, so `isolcpus`, the IOMMU and the 1 GB hugepage pool arrive through a boot image rather than a config field.

One file — `cluster.yaml` — describes the whole deployment, and every tool here either writes it, checks it or applies it. That is the point of the split: the questions get asked once, the answers get reviewed as a file, and applying them is a mechanical step that can be repeated, run from Ansible, and diffed when something looks wrong six months later.

5.2.1 The pipeline

```
fermiydi-discover → fermiydi-configure → validate-cluster → fermiydi-prepare
what the host      what it should      is that      make it so
actually has      run, and where    coherent?
```

Tool	Runs on	Reads	Writes	Touches the host
<code>fermiydi-discover</code>	each host	<code>/sys</code> , <code>/proc</code>	facts, YAML or TSV	no
<code>fermiydi-configure</code>	each host	facts	a <code>cluster.yaml</code> host block	no
<code>validate-cluster</code>	anywhere	<code>cluster.yaml</code>	nothing	no
<code>fermiydi-prepare</code>	each host	<code>cluster.yaml</code>	the host itself	only with <code>--apply</code>
<code>fermiydi-k8s</code>	anywhere	<code>cluster.yaml</code>	Helm values, or manifests	no
<code>fermiydi-docker</code>	anywhere	<code>cluster.yaml</code>	per-host <code>docker run</code> scripts	no

On Talos, four of those five have a twin that talks to the machine over the API instead of running on it, because there is nothing on a Talos machine to run on — no shell, no SSH, no package manager:

Tool	Runs on	Replaces
<code>fermiydi-talos-discover</code>	anywhere	<code>fermiydi-discover</code>
<code>fermiydi-talos-prepare</code>	anywhere	<code>fermiydi-prepare</code> , and the GRUB half of it becomes an Image Factory schematic
<code>fermiydi-talos-k8s</code>	anywhere	<code>fermiydi-k8s</code>
<code>helm/fermiydi-talos</code>	—	<code>helm/fermiydi</code> , plus a VFIO device plugin

`fermiydi-configure` and `validate-cluster` are used unchanged: neither ever touched a host. See [TALOS.md](#).

And two layers that deploy what the four steps prepare for:

Layer	Deploys
<code>ansible/</code>	the DMS and every node container, across every host in the file
<code>fermiydi-docker</code>	the same, as plain <code>docker run</code> scripts per host — no Ansible, no orchestrator
<code>helm/fermiydi</code>	the same deployment on Kubernetes, from values <code>fermiydi-k8s</code> renders

Of the four, `fermihdi-prepare` is the only one that writes to a host, and it refuses to unless asked twice: a bare invocation reports what it would do.

`ansible/` runs all four across every host at once, and deploys the DMS the cluster registers with. It builds its inventory from `cluster.yaml` rather than from a second file that would have to agree with it, so there is nothing to keep in step:

```
cd install/ansible
ansible-playbook site.yml      # prepare, the DMS, then the containers
```

One host, start to finish

```
# 1. Look at the machine. Read-only, no root needed.
install/fermihdi-discover -o facts.yaml

# 2. Decide what it runs. Interactive, and still changes nothing.
install/fermihdi-configure -o cluster.yaml

# 3. Check the answer. Step 2 does this for you at the end.
install/validate-cluster cluster.yaml

# 4. Apply it. Dry run first: this is the step that is hard to undo.
install/fermihdi-prepare cluster.yaml      # reports, changes nothing
sudo install/fermihdi-prepare cluster.yaml --apply

# 5. Reboot if step 4 said to, then confirm the kernel came up as asked.
install/fermihdi-prepare cluster.yaml --verify
```

Each tool takes explicit paths above on purpose. `fermihdi-configure` defaults to `./cluster.yaml` in the working directory, while `validate-cluster` and `fermihdi-prepare` default to `install/cluster.yaml` beside themselves — name the file and the difference stops mattering.

For more than one host, use `ansible/` rather than running the above by hand on each machine.

Kubernetes

`fermihdi-k8s` renders Helm values for `helm/fermihdi` from the same file, so a Kubernetes install is described by the same `cluster.yaml` as a bare-metal one:

```
install/fermihdi-k8s cluster.yaml -o values.yaml
helm install fermihdi install/helm/fermihdi -f values.yaml
```

Roles, counts, images, plane CIDRs and sizing carry across. Devices, DPDK and the cpuset split do not — a Pod has no IOMMU group — and the generator says so per cluster file rather than dropping them quietly. `--manifests` renders the chart instead of the values, for a cluster where helm is not what applies them.

Bare Docker, no orchestrator

For several hosts run with plain Docker and neither Kubernetes nor Ansible, `fermihdi-docker` renders the deployment as one shell script per host — the exact `docker run` commands that host's block calls for — from the same `cluster.yaml`:

```
install/fermihdi-docker cluster.yaml -o docker-deploy
# then, from a machine that can SSH to every host in the file:
FERMIHDI_REGISTRY_PASSWORD=... FERMIHDI_CA_CERT_PEM="$(cat ca-bundle.pem)" \
  docker-deploy/deploy-all.sh
```

Nodes run with `--network host`; a plane is a port, not a network, so the generator refuses to emit a host whose instances would collide on the SCC command family or a graphql/mcp listener. The DMS keeps its own bridge networks (its CA and Core both bind 6985). Secrets are read from the environment on each host, never written into a script. `docker-deploy/README.md` lists the port map, the variables to export, and the run order (DMS first). Every `docker run` is the same one `ansible/` issues — this is that layer without the orchestrator, for operators who want to read and run the commands themselves.

More than one host

`fermihdi-configure` does one host per run, because the questions it asks can only be answered by looking at that machine's own CPU topology and devices. Build the cluster file by appending:

```
# on each host in turn, against a file they share
install/fermihdi-configure --append cluster.yaml
```

`--facts` lets it work from a saved `fermihdi-discover --format=tsv` file rather than probing, which is what makes the whole interview runnable from a control node instead of over SSH one machine at a time.

5.2.2 cluster.yaml

- `cluster.example.yaml` — a worked four-host example: a box sharing an SCC with two SNs, an SN on kernel-path storage, an edge host, and a plain proxy host.
- `schema/cluster.schema.json` — the schema, and the place where each field's meaning is written down.

Two things about it are worth knowing before editing one by hand:

PCI addresses must be quoted. YAML 1.1 reads an unquoted `0000:31:00.0` as a sexagesimal number, and only when every segment is below 60 — so it silently works for bus 61 and breaks for bus 31. `validate-cluster` catches the unquoted form before the schema sees it, because the schema's own complaint ("1860.0 is not of type string") tells nobody what to do about it.

Secrets are never stored in it. The registry password and the DMS bootstrap token are named, not held: `password_env` and `bootstrap_token_env` say which environment variable to read at run time.

```
export FERMIHDI_REGISTRY_PASSWORD=...
export FERMIHDI_BOOTSTRAP_TOKEN=...
```

The same rule covers what the DMS is installed with. On a host that runs the DMS, `fermihdi-configure` asks whether you have a license, the DMS-CA's ICA certificate and key, and a dataset schema — and records the *paths*, under `dms:`. The files stay where they are and are read by whatever runs the deploy; relative paths resolve against the cluster file. Each is optional and each has another way in, so answering nothing is a decision rather than an omission: the license and the ICA can arrive through the environment instead, or be applied through the WebUX afterwards. See [ansible/README.md](#) for what each becomes.

5.2.3 What fermihdi-prepare changes

It applies one host's `host_prep` block and nothing else. Four areas:

Area	What it does	Needs a reboot
hugepages	reserves the 2 MB pool per NUMA node through sysfs	no
mounts	hugetlbfs at <code>/dev/hugepages</code> (2 MB) and <code>/dev/hugepages1G</code> (1 GB)	no
vfiio	binds the listed PCI devices to <code>vfiio-pci</code>	only if the IOMMU is off
grub	IOMMU, <code>isolcpus/rcu_nocbs/nohz_full</code> , and the 1 GB pool	yes

HD uses both pools, and each needs its own mount point, because a hugetlbfs mount carries one page size: `/dev/hugepages` is the 2 MB pool DPDK uses for networking, and `/dev/hugepages1G` is the 1 GB pool SPDK uses for storage and its ring buffers. The storage engine finds the 1 GB pool by looking for the `pagesize=1G` mount (or `FERMIHDI_SPDK_HUGEDIR`), so the two can never be confused.

The kernel command line reserves the 1 GB pool with `default_hugepagesz=2M hugepagesz=1G hugepages=N`. Only the 1 GB pages need the command line; the 2 MB pages are reserved per NUMA node through sysfs. `default_hugepagesz` does not choose a pool: it sets the page size of a hugetlbfs mount that names none, which is how systemd mounts `/dev/hugepages`. Keeping it at 2M keeps that mount the 2 MB pool; set to 1G, `/dev/hugepages` would come back from the next reboot as a second 1 GB mount.

It **never reboots**. 1 GB pages, IOMMU and `isolcpus` all need one, and when a machine goes down is the operator's call. It says which are outstanding and exits 2; `--verify` is the pass to run afterwards.

To survive that reboot it installs, and regenerates on every `--apply`:

Path	Why
<code>/usr/local/sbin/fermihdi-hugepages + its .service</code>	2 MB pages are not reserved on the kernel command line, so something has to claim them on the way up, per NUMA node. <code>sysctl</code> cannot: <code>vm.nr_hugepages</code> has no per-node form.
<code>/usr/local/sbin/fermihdi-vfio-bind + its .service</code>	<code>vfio-pci</code> binding lives in <code>sysfs</code> and does not survive a reboot.
<code>/etc/modules-load.d/fermihdi-vfio.conf</code>	loads <code>vfio-pci</code> before the bind unit runs.
<code>/etc/fstab</code> entries for <code>/dev/hugepages</code> (2 MB) and <code>/dev/hugepages1G</code> (1 GB, <code>nofail</code>)	both mounts come back at boot; nothing else creates <code>/dev/hugepages1G</code> .

Undoing it: `/etc/default/grub` is backed up with a timestamp before every edit and the path is printed, and `scripts/host_setup.sh unbind <PCI>` returns a device to the kernel driver.

5.2.4 Relationship to `scripts/`

`scripts/` holds the interactive, per-machine tools — the ones you reach for when you are working out what a machine can do, or fixing one by hand:

Script	Does
<code>host_setup.sh</code>	Bare-metal host preparation, one command at a time: <code>status</code> , <code>hugepages</code> , <code>bind <PCI></code> , <code>unbind <PCI></code> , <code>grub-advice</code>
<code>setup_scc_host.sh</code>	SCC-specific host preparation
<code>setup_sn_host.sh</code>	SN-specific host preparation, including NVMe/VFIO binding
<code>recommend_topology.sh</code>	Inspects <code>lscpu</code> and emits CPU pinning configuration
<code>prebuild_libs.sh</code>	Builds the <code>libfermihdi_*</code> libraries
<code>sync_*_submodules.sh</code>	Submodule maintenance

The split is not "which one touches the host" — `fermihdi-prepare` binds drivers and edits GRUB, and it is meant to. It is **where the decision was made**. `scripts/host_setup.sh bind 0000:61:00.0` is a decision made at a prompt, once, by whoever is at the keyboard. `fermihdi-prepare` makes no decisions at all: it applies ones already written down, reviewed and validated, the same way on every host, and reports what it would do before doing it. Anything here that starts asking the operator what to do belongs in `scripts/` instead — or in `fermihdi-configure`, which is where the asking is supposed to happen.

5.2.5 Requirements

- `python3` with **PyYAML**, for `validate-cluster`, `fermihdi-prepare` and `fermihdi-k8s`. `jsonschema` is optional: without it the schema check is skipped and the semantic checks still run.
- `bash`, `util-linux` (`lsblk`, `lscpu`, `findmnt`) and `iproute2` (`ip`) for `fermihdi-discover` and `fermihdi-configure`. Both are pure shell so they run on a host with nothing installed on it yet; `fermihdi-discover --check-deps` reports what is missing.
- `root`, for `fermihdi-prepare --apply` and nothing else. Nothing in the Talos path needs root at all: it changes machines through the API, not the machine it runs on.
- `talosctl`, for the Talos path, and network reach to an Image Factory unless every machine already boots the right kernel arguments.
- On the machine you install *from*, and only for the layer you use: Ansible ≥ 2.14 with `community.docker`, or `helm` ≥ 3 .

5.2.6 Getting it onto a host

The installer is published as an archive, with a script that fetches it, verifies it against its checksum, unpacks it and runs the read-only host inventory:

```
curl -fsSL https://downloads.fermihdi.io/install.sh | sh
```

The `hd/` folder the objects live in is applied by the route, so it does not appear in the URL. It changes nothing and needs no root — the only step that changes a host is `fermihdi-prepare --apply`, which it does not run. Pin a version with `https://downloads.fermihdi.io/2.0.5/install.sh`, or fetch `fermihdi-hd-install-2.0.5.tar.gz` and its `.sha256` directly.

`scripts/package_install.sh` builds that archive and publishes it.

The manual is not inside it. It is published under the version on the docs site, in the three forms it is read in — and so can be corrected without reissuing the installer:

https://docs.fermihdi.io/hd/2.0.5/web/	the site
https://docs.fermihdi.io/hd/2.0.5/pdf/hd-administrator-manual.pdf	one printable file
https://docs.fermihdi.io/hd/2.0.5/md/	the Markdown both are built from

`scripts/publish_docs.sh` builds and publishes all three. `ADMIN_GUIDE.md` still ships in the archive: it is what the installer needs, and a host that cannot reach the internet still has it.

5.2.7 What is here

<code>ADMIN_GUIDE.md</code>	the guide: architecture, requirements, every option
<code>README.md</code>	this file
<code>cluster.example.yaml</code>	a worked cluster definition
<code>schema/</code>	the JSON Schema validate-cluster checks against
<code>fermihdi-discover</code>	inventory a host
<code>fermihdi-configure</code>	interview, and write its block
<code>validate-cluster</code>	check a definition against the schema and against reality
<code>fermihdi-prepare</code>	apply one host's <code>host_prep</code>
<code>fermihdi-k8s</code>	render Helm values from a definition
<code>ansible/</code>	deploy across every host in the definition
<code>helm/fermihdi/</code>	the Kubernetes chart
<code>TALOS.md</code>	installing onto Talos Linux: the map of that path
<code>cluster.talos.example.yaml</code>	a worked Talos cluster definition
<code>fermihdi-talos-discover</code>	inventory a Talos machine over the API
<code>fermihdi-talos-prepare</code>	machine config, schematic, apply, upgrade, verify
<code>fermihdi-talos-k8s</code>	render Helm values for the Talos chart
<code>helm/fermihdi-talos/</code>	the Talos chart: device plugin, PSA, both hugepage pools

5.2.8 The optional filter node

A `bloomrf` role gives a storage cluster a BloomRF filter node: an engine role that takes the SCC's ingest broadcast like a Storage Node and keeps a filter per batch instead of the records, so the SCC can leave unread the batches a query cannot match. Every tool here knows the role — the interview offers it, the validator sizes it (two cores, 300 × 2 MB pages, no 1 GB pages, no devices) and checks its name against its SCC, Ansible runs it, both charts deploy it. What none of them can do is provision it: that happens in the DMS, in the storage cluster whose SCC it serves, and has to happen before the SCC takes its relay list. `ADMIN_GUIDE.md` has the three steps.

5.2.9 Not written yet

- `lib/` — shared shell helpers, if the scripts here ever grow enough to need them.
- The front doors. Neither layer deploys the two reverse proxies that sit in front of the WebUX and of the Proxy and the query services, authenticate the caller and set the `X-Forwarded-*` headers everything behind them relies on. They are the customer's, Traefik or otherwise; `docs/front_door.md` is the contract and `docs/examples/ansible_example` deploys a reference pair. Until you have them, the WebUX and the Proxy are published on the loopback address.

5.2.10 Related material worth reading before writing anything here

- [docs/examples/](#) — working Ansible, Chef, Puppet, Kubernetes and Nomad deployments. These are DMS-managed: the environment carries identity only and the DMS supplies `record_size`, `scc_ip`, `storage_devices` and the hash election while each node waits at its boot gate. Anything written here should assume the same unless it is explicitly a pre-provisioned installer.
- The per-component configuration guides, which mark every setting that is read but currently has no effect: [SCC](#) · [SN](#) · [filter node](#) · [SQL server](#) · [Proxy](#) · [DMS](#)
- [cpu_requirements_sn_scc.md](#) and [IMAGE_BUILD.md](#).

5.3 Networking

HD runs on a network you provide and operate. The installer places HD's containers on your hosts or in your Kubernetes cluster, but it does not authenticate your users, connect your nodes to each other or encrypt the traffic between them. Plan those three things before you install, and test them before you load data.

You provide	Why HD needs it	Where it is described
The two front doors and the identity and authorization system behind them	Nothing in HD authenticates a person or decides who may use it	The front doors , below
Connectivity between every node, and from clients to the Storage Nodes	Nodes register, take instructions and move records over your network	Connecting the nodes , below
Encryption for traffic HD does not encrypt, above all the data plane	FHDWP, the protocol that carries records, is not encrypted	Encryption on the wire , below

5.3.1 The front doors and user authorization

HD acts on the identity headers that reach it. Establishing who a caller is, and whether they should get in at all, is the job of your systems in front of it. You are responsible for:

- **Both front doors.** The DMS front door is the reverse proxy through which administrators reach the WebUX. The query front door is the one through which people, applications and the query services reach the Proxy, the SQL server, the GraphQL services, the MCP server and Data Explore. Any ingress you already run will do, with or without a forward-authentication step.
- **Authentication and authorization.** The identity provider your front doors authenticate against, and the policy that decides which users and groups are let through, are yours. The front doors set the `x-Forwarded-*` identity headers and must strip any a client sends.
- **Keeping HD off open networks until they exist.** Neither the Ansible layer nor the Kubernetes chart deploys a front door. Until yours are in place, keep the WebUX and the Proxy on a loopback or management network, where the installer publishes them by default.

The Traefik instances in the sandbox are stand-ins and are not a production configuration: its authenticator lets everyone in. The identity contract, service by service, is on [the front doors page](#), and [Two front doors](#) covers what to settle before installing.

5.3.2 Connecting the nodes

Every HD node must be able to reach the others on the ports in the [port table](#): DMS-Core at the address in its certificate's SAN, the SCC and Storage Nodes on the control and data planes, and the filter node from the SCC. Query clients also need the Storage Nodes themselves, on the data plane, because query results stream to the client directly from the SNs rather than through the Proxy. A network that lets clients reach only the Proxy returns empty results without an error.

How you provide that connectivity depends on where HD runs:

- **Bare metal and Docker hosts.** Nodes use host networking, and a plane is a port rather than a separate network. Your routing, VLANs and firewalls must let each host reach the others' plane addresses on those ports. The DMS keeps its own bridge networks on its host.
- **Kubernetes, including Talos Linux.** You provide the virtual networking: a CNI that gives Pods reachability across nodes, NetworkPolicy that allows the HD ports, and a way for query clients outside the cluster to reach the Storage Node Pods on the data plane. Check your CNI's MTU and encapsulation against the [FHDWP network guide](#). A DPDK data path in Kubernetes also needs SR-IOV or host-network Pods, which the chart does not render ([what differs on Kubernetes](#)).
- **Across sites or clouds.** Links between locations, VPN or interconnect included, are yours to provision and size. Jumbo frames often do not survive them.

5.3.3 Encryption on the wire

HD's control plane uses mutual TLS. Node registration, configuration and the connections that dispatch and verify work carry certificates issued by the deployment's DMS-CA, as described in [Identity, trust and licensing](#).

The data plane is not encrypted. FHDWP, the FermiHDI HD Wire Protocol on port 6987, carries records unencrypted between the Ingestor, the SCC and the Storage Nodes, and from the Storage Nodes to SDK clients receiving query results. Anyone who can capture that traffic can read the records in it. The hash chain that verifies each ingested batch detects records that are lost or altered in transit, but it does not keep them confidential. The health and metrics endpoints on port 8080 are plain HTTP as well.

If your records must be encrypted in transit, add that encryption in the network:

- **Isolate the data plane.** A physically separate or dedicated VLAN data network, reachable only by HD nodes and approved clients, is the simplest control and works with every data path.
- **Encrypt between hosts.** WireGuard or IPsec tunnels between hosts, or MACsec on the links, protect FHDWP without changing HD.
- **Encrypt in the CNI.** On Kubernetes, use your CNI's transparent encryption (for example WireGuard or IPsec in Cilium, or WireGuard in Calico).
- **Mind DPDK.** A DPDK data path sends packets from the NIC without passing through the host kernel, so WireGuard, IPsec or CNI encryption on the host does not see that traffic. With DPDK, encrypt beneath the host (MACsec on the switches or NICs, or encrypting links between sites) or keep the data plane physically isolated.
- **Leave room in the MTU.** Encryption adds overhead: about 60 to 80 bytes for WireGuard and 60 to 70 for IPsec ESP. FHDWP's default 1200-byte packets leave room for either. Jumbo-frame mode sends 8500-byte packets, so the encrypted path must carry 8500 bytes plus that overhead without fragmenting; see the [FHDWP network guide](#) before enabling it.

5.3.4 Before you install

- Both front doors are chosen, and your identity provider and authorization policy are ready for them, or the WebUX and the Proxy will stay on a management network until they are.
- Every host or Pod can reach DMS-Core at the name in its certificate, and the others on the ports in the port table.
- Query clients can reach the Storage Nodes on port 6987.
- On Kubernetes, the CNI, its MTU and NetworkPolicy are in place.
- You have decided how the data plane is protected, whether by isolation, encryption or both, and whether that decision holds for a DPDK data path.

5.4 install/ansible

The same four steps `install/README.md` describes, run across every host at once instead of one at a time — and the DMS those hosts register with.

There is no inventory file. `fermihdi-inventory` reads `cluster.yaml` and answers Ansible's questions from it, because an inventory file would be a second copy of facts the cluster file already holds — and the two would stop agreeing on the first host anyone added.

```
cd install/ansible
export FERMIHDI_CLUSTER=../cluster.yaml      # optional; this is the default

ansible-playbook site.yml                    # prepare, the DMS, then the nodes
```

5.4.1 The playbooks

Playbook	Does	Changes hosts
<code>prepare.yml</code>	validates the cluster file, then runs <code>fermihdi-prepare --apply</code> on every host	yes
<code>verify.yml</code>	runs <code>fermihdi-prepare --verify</code> everywhere and reports	no
<code>dms.yml</code>	runs the DMS plane on the host whose block declares a <code>dms</code> role	yes
<code>deploy.yml</code>	runs the node containers each host's block calls for	yes
<code>site.yml</code>	<code>prepare.yml</code> , then <code>dms.yml</code> , then <code>deploy.yml</code>	yes

```
ansible-playbook prepare.yml --check          # dry run on every host
ansible-playbook prepare.yml -e fermihdi_reboot=true
ansible-playbook dms.yml                     # the DMS plane on its own
ansible-playbook deploy.yml --limit fermihdi_sn # just the Storage Nodes
ansible-playbook verify.yml                  # after a reboot, or when a node misbehaves
```

The DMS goes up before the nodes, which is why `site.yml` orders them that way: every node registers with it, and one that starts first spends its early life retrying a registration that cannot succeed yet. Running `deploy.yml` on its own against a cluster whose DMS is not up yet is not harmful, only slow to settle.

`--check` on `prepare.yml` is a real dry run: `fermihdi-prepare` without `--apply` reports and changes nothing, so the check run tells you what every host would do. The tooling itself is copied even under `--check` — shipping a read-only script to `/opt/fermihdi/install` is not a change to the host, and without it there would be nothing to run and nothing to report.

Nothing reboots unless you pass `fermihdi_reboot=true`, and when you do, hosts reboot **one at a time** — a cluster that reboots all of its Storage Nodes at once is a cluster that has lost quorum.

5.4.2 What the inventory produces

```
./fermihdi-inventory --list | jq .          # everything
./fermihdi-inventory --host sc-host-1      # one host
ansible-inventory --graph                  # the groups
```

One group per role, so plays and `--limit` can target exactly the hosts running a thing:

```
@fermihdi:          every host in the file
@fermihdi_scc:       hosts running one or more SCCs
@fermihdi_sn:        ... Storage Nodes
@fermihdi_bloomrf:   ... the optional filter node
@fermihdi_proxy:     ... and so on
```

Each host carries `fermihdi_instances` — every instance on it with the role type, `dpdfk` flag, `cpusets` and devices flattened onto it — plus `fermihdi_host_prep` and the raw `fermihdi_roles`. Cluster-wide settings arrive as one `fermihdi_cluster` variable, and `group_vars/all.yml` derives the individual settings from it. That is the file to read and edit: every default and fallback the plays use is in it, in one place, rather than spread through a Python script and half a dozen roles.

5.4.3 Secrets

`cluster.yaml` holds none. Everything is read from the environment of the machine running `ansible-playbook`, which is what the file's `*_env` fields promise:

```
export FERMIHDI_REGISTRY_PASSWORD=...
export FERMIHDI_CA_CERT_PEM="$(cat dms-ca-bundle.pem)"

# Bootstrap tokens are per node: the instance name, upper-cased, with anything
# that is not a letter or digit turned into an underscore.
export FERMIHDI_BOOTSTRAP_TOKEN_SC_1_SN_1=...
export FERMIHDI_BOOTSTRAP_TOKEN_SC_1_SCC=...
export FERMIHDI_BOOTSTRAP_TOKEN_SC_1_BLOOMRF=... # the filter node, if the cluster has one

# ... or one shared token, if that is how the cluster issues them. The variable
# cluster.yaml names in dms.bootstrap_token_env is the fallback for every node
# with no per-node token set.
export FERMIHDI_BOOTSTRAP_TOKEN=...
```

Deploying the DMS needs the rest of them. `dms.yaml` refuses to start with any of the required ones missing and names every one it could not find, rather than deploying a stack that comes up and then fails every handshake made to it.

Three of them can come from a file instead of the environment — the license, and the DMS-CA's certificate and key. `fermihdi-configure` asks for those on the host that runs the DMS and records the paths in `cluster.yaml`'s `dms:` block, where a named file wins over the environment; the run prints which source each one came from. Paths only: the files stay where they are, are read on the machine running the playbook, and relative ones resolve against the cluster file.

```
dms:
  license_file: certs/license.json      # -> DMS_LICENSE_PAYLOAD
  ica_cert_file: certs/dms-ca.crt       # -> DMS_ICA_CERT_PEM / DMS_TLS_CERT_PEM
  ica_key_file: certs/dms-ca.key        # -> DMS_ICA_KEY_PEM / DMS_TLS_KEY_PEM
  schema_file: certs/dataset_schema.json
```

Variable	Is
<code>FERMIHDI_CA_CERT_PEM</code>	the trust bundle — root, intermediate and the DMS-CA's own certificate concatenated. The same one the nodes are given, and what every DMS service verifies against
<code>FERMIHDI_DMS_CA_CERT_PEM</code> / <code>_KEY_PEM</code>	the DMS-CA's ICA certificate and key. It presents this on the wire <i>and</i> signs every CSR in the deployment with it
<code>FERMIHDI_DMS_CORE_CERT_PEM</code> / <code>_KEY_PEM</code>	DMS-Core's TLS identity. The certificate is the leaf and its chain
<code>FERMIHDI_DMS_ICA_CHAIN_PEM</code>	the chain between the root and the DMS-CA, so a node verifying a certificate this deployment issued can build a path
<code>FERMIHDI_DMS_WEBUX_CERT_PEM</code> / <code>_KEY_PEM</code>	the WebUX's own server certificate
<code>FERMIHDI_DMS_WEBUX_FE_CERT_PEM</code> / <code>_KEY_PEM</code>	the client certificate the WebUX's back end presents to DMS-Core. A different identity, signed for client auth, and not interchangeable with the one above
<code>FERMIHDI_DMS_ENCRYPTION_KEY</code>	what DMS-Core encrypts its stored state with. It exits at startup without one
<code>FERMIHDI_DMS_IDENTITY_KEY</code>	what the license payload is encrypted to. Has to be the one the license was issued against
<code>FERMIHDI_DMS_LICENSE_PAYLOAD</code>	the encrypted license. Optional: DMS-Core starts without it and admits no node until one is applied, through the WebUX or by setting this and running again
<code>FERMIHDI_DMS_FE_TOKEN</code>	shared between DMS-Core and the WebUX — what the WebUX's back end presents on the internal management API
<code>FERMIHDI_BOOTSTRAP_TOKEN_DMS_OPA_ENGINE</code>	the OPA engine registers with DMS-Core the way a node does, so its token follows the same per-node convention, with <code>FERMIHDI_BOOTSTRAP_TOKEN</code> as the fallback

`tests/sandbox_dms/generate_certs_and_license.sh` produces a complete set of these for a sandbox, and is the clearest description of what each one is and how they are signed.

Use `ansible-vault` or a secrets manager to get them into the environment. None of them are ever written to a host: the DMS services read PEM content straight from their environment and write it inside their own containers.

5.4.4 What `deploy.yml` works out for each container

From <code>cluster.yml</code>	Becomes
<code>instances[].cpuset</code>	<code>MASTER_CPUSSET</code> — the poll cores the process pins itself to
<code>instances[].container_cpuset</code>	the container's <code>cpuset_cpus</code> — overflow cores only
<code>roles[].dpdk</code>	absent means <code>FERMIHDI_FORCE_POSIX=1</code>
<code>instances[].storage / .nic with driver: vfio</code>	<code>/dev/vfio/<group></code> device passthrough
<code>instances[].storage</code> with a <code>/dev/...</code> path	that block device, passed through
<code>images / registry.url</code>	the image reference
<code>networks</code>	<code>FERMIHDI_*_SUBNET</code>

The `cpuset` split is the invariant worth understanding before changing any of this: the container's cgroup must contain **only** the overflow cores. The runtime shim runs in the host scheduler but inherits that cgroup, and if a DPDK core is in it the shim will be scheduled there — profiling measured over 300,000 shim evictions in a 90-second window that way. `isolcpus` does not prevent it, because an explicit cgroup `cpuset` overrides `isolcpus`. So a DPDK instance with no `container_cpuset` is refused rather than deployed with the default of "every CPU on the box".

IOMMU group numbers are read from the host at deploy time rather than taken from `cluster.yml`. The file records what discovery saw, which is the right thing for the validator to reason about, but group numbering can move under a kernel upgrade — and a container handed the wrong `/dev/vfio` node fails at SPDK attach looking like a driver fault. When the two disagree the run says so.

Both hugepage mounts go into every engine container: `/dev/hugepages` is the 2 MB pool DPDK's EAL needs for networking, and `/dev/hugepages1G` is the 1 GB pool SPDK uses for storage and its ring buffers. One mountpoint carries one page size, so the two pools need two of them.

This pair used to be the other way round, because the storage engine hardcoded SPDK's EAL `hugedir` to `/dev/hugepages` — which forced that path to be the 1 GB mount, and forced `default_hugepagesz=1G` and a reboot with it, since `systemd` mounts `/dev/hugepages` at the default page size. The engine now discovers the 1 GB mount from `/proc/mounts` (and honours `FERMIHDI_SPDK_HUGEDIR`), so the layout is free to be the one that needs no kernel argument to hold. It also matches `docs/examples/ansible_example`, which already used it.

5.4.5 The DMS

`dms.yml` deploys it on the host whose `cluster.yml` block declares a `dms` role — one host, because the file names one `dms.url` and every node is pointed at it.

Container	Is	Reached
<code>dms-ca</code>	signs every identity in the deployment	by DMS-Core, and by nothing else
<code>dms-core</code>	registration, node configuration, the control plane	by every node, on 6986
<code>dms-opa-engine</code>	the policy engine. Registers with DMS-Core the way a node does	internally
<code>dms-opa-adapter</code>	polls an OPA control plane that cannot reach in, and writes what it fetches into the directory the engine reads. Off by default	outbound only
<code>dms-webux</code>	the web UI and its back end	through <code>dms-traefik</code>
<code>dms-traefik</code>	routes to the WebUX and applies the forward-auth middleware	on 8081, loopback by default
<code>otel-collector</code>	where every node in the cluster sends traces and metrics	on 4317 and 4318
<code>dms-crash-vault</code>	holds every crash report a service posts, symbolicates it, and forwards it only as an operator decides on the Telemetry page	port 3001 on the DMS host, for the nodes

This is the one place the layer does not use host networking. DMS-CA and DMS-Core both listen on 6985 for their internal API and both serve observability on 8080; on a host network that is a collision, and on their own bridges it is the isolation the design asks for. So the services find each other by container name across three networks — `fermihdi-dms-ca` holds the CA and DMS-Core alone, `fermihdi-dms-internal` holds everything the UI path touches, `fermihdi-observability` holds everything that reports — and only what the outside has to reach is published to the host.

Three things are worth getting right before the first run:

DMS-Core's certificate has to carry the address the nodes use. `dms.url` is handed to every node verbatim, and the nodes verify what they connect to. A certificate whose SAN says `dms-core` and nothing else works inside the deployment's own network and fails every registration from outside it. `dms.yml` warns when `dms.url` does not name this host; it cannot see inside the certificate.

The WebUX has no authentication of its own. It trusts the `X-Forwarded-*` headers a forward-auth service in front of it sets. Until `fermihdi_dms_forward_auth_url` names one, `dms-traefik` routes to it with no middleware, which is why 8081 is published on the loopback address and not on every interface — reach it through an SSH tunnel, or put your SSO's forward-auth on the `fermihdi-dms-internal` network, name it, and then widen the bind address.

The OPA adapter registers separately. It is a node in the DMS's eyes, with its own name and its own bootstrap token (`FERMIHDI_BOOTSTRAP_TOKEN_DMS_OPA_ADAPTER`, falling back to the cluster-wide one), and it is the one service here that reads its trust bundle only as a file — there is no PEM environment variable for it, so the run writes the bundle to the host and mounts it. Give it `fermihdi_dms_styra_api_url` and `FERMIHDI_STYRA_TOKEN` or it registers and then idles, having nothing to poll.

Nothing is regenerated. Every key and certificate comes from the environment or from the files `cluster.yml` names, on each run — so a certificate the DMS-CA renewed at run time is replaced by whatever those hold the next time this runs. Keep them the source of truth, or expect the run to undo the renewal.

The license and the certificates are not the same decision

A license supplied as `DMS_LICENSE_PAYLOAD` — which is what both `license_file` and the environment variable become — is decrypted, verified and stored, and that is all. Applying the same license through the WebUX does more: DMS-Core pulls the DMS-CA's ICA certificate and its own identity certificate out of it (`handleLicense`, `cmd/dms-core/main.go`), which is what the Day-0 bootstrap uses — the CA generates a key, posts a CSR, and waits for the ICA the license carries.

This layer deploys the pre-provisioned arrangement instead: the certificates are in place before the first container starts, which is why they are asked for separately from the license. If all you have is a license, deploy without the ICA and apply the license through the WebUX afterwards; `fermihdi-configure` says so when it is given one and not the other.

A license the DMS cannot parse is worse than no license: DMS-Core exits at startup rather than coming up unlicensed. `validate-cluster` reads the file and says so before it gets that far.

The dataset schema

`schema_file` is copied to the DMS host and mounted read-only over DMS-Core's active schema at `/app/schemas/dataset_schema.json` — the file `/v1/dms/schema/active` serves to the OPA engine, and which the image does not otherwise ship under that name.

It is not the per-deployment schema. The one Proxy and Ingester nodes are handed at registration comes from the deployment record, which the WebUX writes; this installs the deployment-wide one the policy engine reads.

5.4.6 The filter node

A `bloomrf` instance runs like an SN without a device: it registers with its own token, gets both hugepage mounts and an unlimited memlock, and is never privileged. Two things about it are not in the file. It has to be **provisioned in the DMS first**, as `HD_BLOOMRF` in the storage cluster its name says (`sc-1-bloomrf` for `sc-1-scc`), and the SCC of that cluster only learns of it at the moment its own data plane starts — so on a cluster that is already running, deploy the filter node, wait for it to report ready, then `deploy.yml --limit <the SCC's host>` to recreate the SCC. `ADMIN_GUIDE.md` has the provisioning call and the modes.

5.4.7 What this does not deploy

The public front door: the FE Traefik that terminates TLS for the Proxy and the WebUX, the forward-auth service behind it, and the data explorer. `docs/examples/ansible_example` is the reference for those. It also deploys the DMS, from an older arrangement than this one — where it and this layer disagree, `tests/sandbox_dms/compose.yml` is what the DMS is actually run as, and is what `dms.yml` is built from.

5.4.8 What `cluster.yml` does not carry

`fermihdi-configure` writes identity, placement and host preparation. The rest of what the containers need is not in the schema, and comes from `group_vars/all.yml` (or `-e`) instead:

Setting	Why it is not in the file
<code>fermihdi_kafka_brokers / _topics</code>	the ingester's message bus. No schema field; the run warns rather than starting an ingester with nothing to read
<code>fermihdi_graphql_proxies_path</code>	scaffolded on first deploy and never overwritten, so an operator's proxy list survives the next run
<code>fermihdi_extra_env</code>	per-role core splits (<code>TRANSPORT_CORES</code> , <code>STORAGE_CORES</code> , <code>HASH_CORES</code> , <code>KV_CORES</code> , <code>FILTER_CORES</code>) and anything else. Merged last, so it wins
<code>fermihdi_dms_forward_auth_url</code>	what authenticates the WebUX. Nothing in the file says who your SSO is
<code>fermihdi_crash_vault_keep_days</code> , <code>fermihdi_crash_vault_keep_mb</code>	the crash vault's retention: the oldest crash goes past either limit (30 days, 2048 MB)
<code>fermihdi_telemetry_region</code>	the FermiHDI region the deployment shares into when its administrator opts in (<code>sg</code>)
<code>fermihdi_dms_enable_analysis</code>	whether opt-in telemetry may leave the deployment. Off unless said otherwise, and a decision rather than a fact about the cluster

`fermihdi_otel_url` and `fermihdi_otel_exporter_endpoint` are not in the file either, but neither has to be set: both are derived from the host running the DMS, because that is where the collector goes. Set them by hand for a collector that lives somewhere else.

They are two different things and are easy to confuse — the first is a routing probe the C++ nodes use to work out which of their own interfaces is on the observability plane, and configures no exporter; the second is where the SDK actually sends traces and metrics.

`networks:` and `images:` **are** in the schema, but `fermihdi-configure` does not write them yet. Without `networks:` the plane CIDRs are left unset and each node falls back to its own interface selection — which is fine on a single-homed host and wrong on any other, so `deploy.yml` says so once per run. Without `images:` the reference is built from `registry.url`, the namespace in `group_vars/all.yml`, and the published image name.

5.4.9 Requirements

- **Ansible $\geq$ 2.14** on the machine you run this from, with `community.docker`:

```
ansible-galaxy collection install community.docker
```

- **python3 with PyYAML** on the control machine (the inventory) and on every target (`fermihdi-prepare`). `prepare.yml` installs it on the targets.
- **SSH with sudo** to every host. Connection details come from each host's `address`, `ssh_user` and `ssh_key` in `cluster.yml`.
- Docker on the targets. `deploy.yml` and `dms.yml` install `docker.io` and `python3-docker`; override `fermihdi_docker_packages` for a host that gets Docker from `docker-ce`.

5.5 install/helm/fermihdi

The chart the installer deploys to Kubernetes. `install/fermihdi-k8s` renders its values from the same `cluster.yaml` everything else in `install/` reads, so a Kubernetes install and a bare-metal one are described by one file:

```
install/fermihdi-k8s cluster.yaml -o values.yaml
helm install fermihdi install/helm/fermihdi -f values.yaml
```

It began as [docs/examples/k8s_example](#), which is still there and still a reference. This copy is the one that is kept current; where the two disagree, this is the one to trust.

5.5.1 What it deploys

Namespace	Runs
<code>fermihdi-dms</code>	DMS-CA, DMS-Core, the OPA engine, the WebUX, the OTEL collector
<code>fermihdi-sc</code>	the SCC, the Storage Nodes as a StatefulSet, and the optional BloomRF filter node (<code>sc.bloomrf.enabled</code>)
<code>fermihdi-edge</code>	the Ingester and the HD Proxy
<code>fermihdi-extra</code>	<code>hd_mcp</code> , <code>hd_graphql</code> , the data explorer

Each group is reached from the others through `ExternalName` Services rather than by fully-qualified names spread through the templates — which is what makes a DMS outside the cluster a value rather than an edit:

```
dms:
  externalHost: dms.corp.example.com # already installed, or on bare metal
  core: { enabled: false }          # ... and nothing here deploys one
```

5.5.2 What differs from the bare-metal install

A Pod is not a host, and four things do not survive the trip. The generator says so per cluster file rather than leaving it to be discovered:

- **DPDK.** Every node here runs with `FERMIHDI_FORCE_POSIX=1`. A DPDK data path needs SR-IOV or a host-network Pod with the NIC bound to vfio, neither of which this renders.
- **Devices.** Storage Nodes write to a PVC-backed file, sized by `sc.sn.storageFileSize`. Nothing passes an NVMe through, so the `storage: entries` in `cluster.yaml` and their IOMMU groups have nowhere to go.
- **Cpusets.** `cluster.yaml` splits poll cores from overflow cores to keep the container runtime off the reactors. The equivalent here is the static CPU manager policy with integer CPU limits, which is what the generated resources ask for — the node pool has to be configured for it or the request is just a number.
- **1 GB hugepages.** Requested on bare metal for SPDK's NVMe DMA. Nothing here DMAs, so only the 2 MB pool is asked for.

`IPC_LOCK` is granted to the SCC, the SNs, the filter node and the Ingester: the HD engine and DPDK pin their arenas, and a container's default `RLIMIT_MEMLOCK` of 64 KB stops them at startup. `hd_mcp` and `hd_graphql` get a `medium: Memory` `emptyDir` at `/dev/shm` instead — they allocate from shared memory rather than hugepages, and the 64 MiB a container gets by default is not enough to start with.

5.5.3 Secrets

The chart reads everything from three `Secret` objects — `fermihdi-dms-secrets`, `fermihdi-sc-secrets`, `fermihdi-edge-secrets` — so anything that creates them with the right keys will do: External Secrets, Sealed Secrets, or your own pipeline.

`secrets.create=true` renders them from values instead, which is what `fermihdi-k8s --secrets` writes and which is for development. The SC Secret carries `BLOOMRF_TOKEN` beside the SCC's and the SNs' tokens; it is read only when `sc.bloomrf.enabled` is true.

The filter node is the one workload here with a prerequisite outside the chart: it must be provisioned in the DMS, as `HD_BLOOMRF` in the storage cluster `sc.clusterName` names, before it starts, and the SCC takes its relay list once when its data plane starts. On a running release, enable the filter node, wait for it to report ready, then restart the SCC's Pod.

5.5.4 Keeping it honest

The services' environments are checked against `tests/sandbox_dms/compose.yml`, which is the deployment as it is actually run. When something moves there, it moves here. The last sweep (2026-08-20) closed these:

- `OTEL_EXPORTER_OTLP_ENDPOINT` and `_PROTOCOL` were absent everywhere. They are what the SDK actually exports over; `FERMIHDI_OTEL_URL` is a routing probe the C++ nodes use to find their own observability address and configures no exporter. Every service looked configured and exported nothing.
- `FERMIHDI_LOG_STDOUT` was absent everywhere, so `kubectl logs` showed startup and errors only.
- `LOG_LEVEL` was absent on all four DMS services.
- DMS-CA was published on **6989** — its container port, its Service, the NetworkPolicy and `dmsCAURL` all agreed with each other and disagreed with the binary, which listens on 6985. Nothing could reach the CA. It is now 6985, and `CA_LISTEN_ADDR` is set explicitly so the four cannot drift apart again.
- `global.dataPlaneClIDR` — misspelled in the values and read by that misspelling in every template, so setting the obvious name did nothing.
- The SCC's `FANOUT_MAX_INFLIGHT_BATCHES` and the SN's mode, write-path and engine-memory settings had no values at all.

To check it yourself:

```
helm lint install/helm/fermihdi
helm template hd install/helm/fermihdi -f values.yaml | kubectl apply --dry-run=server -f -
```

5.5.5 The OPA adapter is a sidecar

`dms.opaAdapter.enabled=true` adds it to the OPA engine's Pod rather than deploying it separately, because the handover between the two is a directory: the adapter polls an OPA control plane that cannot reach into the deployment and writes bundles into `dms.opaEngine.bundleDir`; the engine, which reaches nothing outward, validates and serves what appears there. Two Pods would need a `ReadWriteMany` volume for what an `emptyDir` does inside one.

Three things follow from sharing a Pod, and each is a value rather than a surprise:

- Containers in a Pod share one network namespace, and both binaries default their certificate-renewal listener to `0.0.0.0:6985` and their observability server to `8080`. The engine keeps the defaults; the adapter takes `renewPort` and `obsPort`.
- The adapter registers with DMS-Core in its own right, so it presents its own bootstrap token — `OPA_ADAPTER_TOKEN` in the DMS Secret, falling back to the engine's.
- It is the one service in the chart that reads its trust bundle only as a file. There is no PEM environment variable for it, so the CA bundle is projected out of the Secret to `/etc/fermihdi/certs/fermihdi_root_ca.pem`.

Without `styraApiUrl` and a `STYRA_TOKEN` it registers and then idles: there is nothing for it to poll.

5.5.6 Not deployed here

The two front doors — the reverse proxies that terminate TLS for the WebUX and for the Proxy and the query services, authenticate the caller and set the identity headers (`docs/front_door.md` is the contract; Traefik is what the reference deployment uses, and yours may be anything that keeps it) — and `hd_ipflow_graphql`. `docs/examples/k8s_example` is no further ahead on either; they are genuinely not written yet.

5.6 Installing HD on Talos Linux

Talos has no shell, no SSH, no package manager and no `/etc`. A machine is configured by sending it a document over an API and rebooted by asking it to reboot. Nothing in `README.md`'s pipeline survives that unchanged, and yet almost all of it survives in shape: the same `cluster.yaml`, the same `fermihdi-configure` interview, the same `validate-cluster`, the same sizing model. What changes is the last mile.

This file is the map of that last mile. `ADMIN_GUIDE.md` is still the long form for everything that is not Talos-specific.

5.6.1 The pipeline

```
fermihdi-talos-discover → fermihdi-configure → validate-cluster → fermihdi-talos-prepare → fermihdi-talos-k8s → helm
what the machine has    what it should run    is that                machine config,      values for the        deploy
(over the API)          coherent?              schematic, upgrade    Talos chart
```

Two of those five are the bare-metal tools unchanged. `fermihdi-configure` and `validate-cluster` never touched a host in the first place — they read facts and write a file — so they work here as they are.

Tool	Bare metal	Talos
inventory	<code>fermihdi-discover</code> , on the host	<code>fermihdi-talos-discover</code> , over the API
interview	<code>fermihdi-configure</code>	the same
check	<code>validate-cluster</code>	the same, plus <code>check_talos</code>
apply	<code>fermihdi-prepare</code> , on the host	<code>fermihdi-talos-prepare</code> , over the API
render	<code>fermihdi-k8s</code>	<code>fermihdi-talos-k8s</code>
chart	<code>helm/fermihdi</code>	<code>helm/fermihdi-talos</code>
deploy	<code>ansible/</code> or <code>helm</code>	<code>helm</code> only — there is no Docker and nothing to SSH into

`cluster.talos.example.yaml` is a worked four-machine example. It differs from `cluster.example.yaml` in exactly two places: a top-level `talos:` block, and a `talos:` block per host.

5.6.2 Start to finish

```
# 0. Boot the machines from a Talos ISO. They come up in maintenance mode with
#    an API and no configuration. Nothing else is needed on them.

# 1. Look at one. Read-only. --insecure because it has no certificates yet.
install/fermihdi-talos-discover -n 10.147.0.20 --insecure

# 2. Once it is configured, look again — this pass has the topology.
install/fermihdi-talos-discover -n 10.147.0.20 --format=tsv > facts.tsv
install/fermihdi-configure --facts facts.tsv --append cluster.yaml

# 3. Check the answer.
install/validate-cluster cluster.yaml

# 4. See what would happen. Changes nothing, uploads nothing.
install/fermihdi-talos-prepare cluster.yaml
install/fermihdi-talos-prepare cluster.yaml --render-only # offline, files only

# 5. Make it so. This reboots machines, one at a time.
install/fermihdi-talos-prepare cluster.yaml --apply

# 6. Confirm the machines came up as asked.
install/fermihdi-talos-prepare cluster.yaml --verify

# 7. Deploy.
install/fermihdi-talos-k8s cluster.yaml -o values.yaml
helm install fermihdi install/helm/fermihdi-talos -f values.yaml \
  --kubeconfig talos/kubeconfig
```

Step 1 and step 2 are the same command twice, and that is not an accident — see below.

5.6.3 Why a Talos install is two passes

Talos answers two different sets of questions depending on whether the machine has been configured:

	maintenance mode (<code>--insecure</code>)	configured
<code>talosctl get disks, pcidevices, cpu, systeminformation</code>	yes	yes
<code>talosctl read /proc/cpuinfo, /sys/...</code>	no	yes
<code>talosctl list</code>	no	yes

Everything the sizing model needs — per-NUMA-node cpulists, per-node hugepage counts, SMT sibling pairs, IOMMU groups — lives in `/sys` and `/proc`, and none of it is readable until the machine has a configuration. What is readable is enough to choose an install disk and identify the devices, which is enough to write a first configuration.

So: inventory coarsely, configure, inventory properly, configure again. `fermihdi-talos-discover` says so rather than quietly emitting zeroes — a `tsv` produced in maintenance mode exits 2 and tells you not to feed it to `fermihdi-configure`.

5.6.4 What happens to `host_prep`

The `host_prep` block is unchanged from the bare-metal file. What changes is where each half of it ends up, and the line runs somewhere no other distribution puts one.

<code>host_prep</code>	Bare metal	Talos
<code>hugepages_2m, numa[].hugepages_2m</code>	<code>sysfs write + fermihdi-hugepages.service</code> to survive reboot	<code>machine.sysfs</code> , reapplied on every boot — no unit, no script
<code>vfio_bind</code>	<code>unbind / driver_override / bind + fermihdi-vfio-bind.service</code>	one <code>PCIDriverRebindConfig</code> document per device
module loading	<code>/etc/modules-load.d/fermiHdi-vfio.conf</code>	<code>machine.kernel.modules</code>
hugetlbfs mounts	<code>/etc/fstab + systemd</code>	none — Pods get <code>emptyDir: {medium: HugePages-*</code>
<code>iommu</code>	<code>/etc/default/grub</code>	Image Factory schematic
<code>isolcpus</code>	<code>/etc/default/grub</code>	Image Factory schematic
<code>hugepages_1g</code>	<code>/etc/default/grub</code>	Image Factory schematic
<code>grub.apply: false</code>	leave GRUB alone	put no kernel arguments in the schematic

Three of the four files `fermiHdi-prepare` installs to survive a reboot have no counterpart here, because they have no reason to exist. Talos reapplies `machine.sysfs` and `machine.kernel.modules` itself.

The kernel argument list is not written twice. `fermiHdi-talos-prepare` imports `grub_tokens()` from `fermiHdi-prepare`, so the same `host_prep` produces the same arguments whether they land in GRUB or in a UKI.

5.6.5 The trap: `extraKernelArgs` does nothing

This is the one thing worth knowing before anything else.

Since Talos 1.10, a UEFI machine boots `systemd-boot` and a Unified Kernel Image, and the kernel command line is inside the UKI. From the Talos [boot loader documentation](#):

With `systemd-boot`, the `.machine.install.extraKernelArgs` field in the machine configuration is ignored, as kernel arguments are embedded in the UKI and cannot be modified without upgrading the UKI.

So `machine.install.extraKernelArgs` is not the answer, and neither is any other config field. `isolcpus`, `intel_iommu=on` and the 1 GB hugepage pool go into an **Image Factory schematic**, which is uploaded, returns a content-addressed ID, and names a custom installer image:

```
# what fermihdi-talos-prepare writes as talos-sc-1.schematic.yaml
customization:
  extraKernelArgs:
    - intel_iommu=on
    - iommu=pt
    - default_hugepagesz=2M
    - hugepagesz=1G
    - hugepages=28
    - isolcpus=2-29
    - rcu_nocbs=2-29
    - nohz_full=2-29
  systemExtensions:
    officialExtensions:
      - siderolabs/intel-ucode

curl -X POST --data-binary @schematic.yaml https://factory.talos.dev/schematics
# {"id":"a94d87ed..."}
talosctl upgrade --image factory.talos.dev/metal-installer/a94d87ed...:v1.11.2
```

The schematic is the new GRUB, and it is per host, because `isolcpus` and the 1 GB pool differ per host. Talos 1.13 adds `grubUseUKICmdline` so that legacy-BIOS machines converge on the same mechanism.

Two consequences worth planning around:

- **Changing `isolcpus` on a running machine is an upgrade and a reboot.** On bare metal it is a GRUB edit and a reboot; here the boot image itself is replaced. `--apply` does it one machine at a time, and never in parallel.
- **A machine installed fresh needs no upgrade.** The configuration it is given in maintenance mode already carries `machine.install.image`, so the installer that writes the disk is the schematic's own and the machine boots with the right command line the first time. `fermihdi-talos-prepare` detects this and says so rather than rebooting a machine that has just booted.

Air-gapped

`talos.factory_url` points at the Image Factory. Set it to a self-hosted one and nothing reaches the internet. Schematic IDs are cached under `<out_dir>/schematics/` by content hash, so a re-run of a dry run needs no network at all, and `--render-only` contacts nothing except — if you have not said which — the machine itself, to ask whether it is Intel or AMD. Put `intel_iommu=on` or `amd_iommu=on` in `host_prep.grub.extra_args` and even that goes away.

5.6.6 VFIO takes three things, not one

On bare metal, binding a device to `vfio-pci` is one step and the container gets it through a bind mount. Here it is three, and missing any one of them fails differently:

1. **The rebind.** A `PCIDriverRebindConfig` document per PCI address. A udev rule with `ATTR{driver_override}` does *not* work: the kernel has already bound `nvme` or `mlx5_core` by the time udev sees the device. Talos does the unbind, the override and the rebind itself.
2. **The modules.** `vfio-pci` and `vfio_iommu_type1`, from `machine.kernel.modules`. The second one is the expensive lesson. SPDK's setup script tries to `modprobe` it from inside the container and cannot:

```
failed to bind NVMe disk 0000:03:00.0: ... modprobe: ERROR: could not insert
'vfio_iommu_type1': Operation not permitted
```

and what follows is `EAL: No supported IOMMU extensions found! EAL: VFIO support could not be initialized`. A container has no business loading kernel modules; the machine configuration loads it and SPDK finds it already there. (Seen in the wild as [longhorn#12709](#), which is the same SPDK-on-Talos case as ours.)

1. **The device plugin.** A rebind creates `/dev/vfio/<group>` on the host and nothing follows from that: a container does not get a host device by asking, the kubelet does not know it exists, and the scheduler has no reason to place the SN on the machine that has it. `helm/fermihdi-talos` runs a VFIO device plugin that advertises each pool as an extended resource, and the Pods request them by name.

`fermihdi-talos-k8s` builds the pools from the devices `cluster.yaml` already assigns, one per role and device kind (a `bloomrf` role holds a NIC at most, and only when it opts into DPDK):

```
sc:
  sn:
    vfioResources:
      fermihdi.io/sn_nvme: 2
```

The plugin selects on `drivers: ["vfio-pci"]` as well as the address list, which is not redundant: it is what stops a device whose rebind silently failed from being advertised, and a Pod being scheduled onto a machine where the NVMe is still on the kernel driver.

The easy path

An SN on **kernel-path storage** — SATA, SAS, FC, iSCSI, NVMe-oF — needs none of this. No rebind, no IOMMU, no device plugin, and no 1 GB pages either, because those devices reach the disk through `bdev_aid` and never DMA. It is the configuration that asks the least of Talos and the one to start with on a new machine. `talos-sc-2` in the example is exactly that.

5.6.7 Pod Security Admission

Talos ships the API server enforcing the `baseline` profile with only `kube-system` exempt. `baseline` allows neither `IPC_LOCK` nor a privileged container, so the SCC and the SNs are **rejected at admission** — before scheduling, with a message about the namespace rather than about the Pod.

The chart labels the `sc` and `edge` namespaces `privileged` and leaves the DMS and extra namespaces on `baseline`, because nothing in those needs more. Set `talos.privilegedNamespaces: false` only if something else already labels them, or if the API server was given an `admissionControl` exemption instead.

5.6.8 The two hugepage pools

HD uses both pools: 2 MB pages for DPDK networking and 1 GB pages for SPDK storage and its ring buffers. A `hugetlbfs` mount carries one page size, so a component that uses both gets two mounts. On Talos they are Pod volumes, backed by `emptyDir` with `medium: HugePages-2Mi` and `medium: HugePages-1Gi`:

mount	pool
<code>/dev/hugepages</code>	<code>HugePages-1Gi</code>
<code>/dev/hugepages2M</code>	<code>HugePages-2Mi</code>

A component with no 1 GB pages has one pool, `HugePages-2Mi` on `/dev/hugepages`.

The names differ from bare metal, where `fermihdi-prepare` mounts `/dev/hugepages` at 2 MB and `/dev/hugepages1G` at 1 GB, and neither layout depends on the name. DPDK uses every `hugetlbfs` mount it finds, and the storage engine takes the one whose page size is 1 GB (or `FERMIHDI_SPDK_HUGEDIR`), so SPDK cannot draw its DMA buffers from the 2 MB pool the network data path was sized for. On the node itself, the kernel reserves the 1 GB pool from the schematic's `hugepagesz=1G hugepages=N`, and `machine.sysfs` reserves the 2 MB pool per NUMA node.

5.6.9 CPU pinning

`isolcpus` takes cores off the scheduler. It does **not** take them out of the kubelet's allocatable set — without more, the kubelet will hand an isolated core to whatever asks first and the HD instance pinned there shares it.

`fermihdi-talos-prepare` writes the kubelet's static CPU manager into the machine configuration:

```
machine:
  kubelet:
    extraConfig:
      cpuManagerPolicy: static
      reservedSystemCPUs: 0-1           # the union of container_cpuset
      topologyManagerPolicy: single-numa-node
      topologyManagerScope: pod
```

`reservedSystemCPUs` is the union of the `container_cpuset` values `cluster.yaml` already declares — the housekeeping cores — and is not invented when they are absent: a wrong reservation is worse than none, so the tool warns and leaves the policy off instead. `topologyManagerPolicy` is only set when every instance on the host declares `numa_node`.

The Pods must then be Guaranteed to get whole cores, which means integer CPU requests equal to limits. The chart does that from the `cpuset` widths.

Changing `cpuManagerPolicy` on a running node needs the old state removed. The kubelet refuses to start when the policy in its config disagrees with `/var/lib/kubelet/cpu_manager_state`. This is a Kubernetes behaviour, not a Talos one, and nothing here deletes that file for you.

single-numa-node rejects rather than degrades. Once it is on, a Pod whose CPUs, hugepages and devices cannot all be satisfied from one NUMA node is refused with `TopologyAffinityError` and never runs. That is the intended behaviour — an SN reaching across the interconnect on every I/O is the thing `numa_node` exists to prevent — but it means a device assigned to the wrong node in `cluster.yaml` shows up as an unschedulable Pod rather than as slow throughput. `validate-cluster` warns about the mismatch first; this is what happens if the warning is ignored.

5.6.10 Verifying

`--verify` reads the running machines and compares them to the definition. It changes nothing:

```
install/fermiHDI-talos-prepare cluster.yaml --verify
```

It checks four things, and each failure names its cause:

Check	Read from
kernel arguments present	<code>talosctl read /proc/cmdline</code>
2 MB pages per NUMA node	<code>talosctl read /sys/devices/system/node/nodeN/...</code>
every device on <code>vfio-pci</code>	<code>talosctl get pcidriverrebindstatus</code>
<code>vfio_pci</code> and <code>vfio_iommu_type1</code> loaded	<code>talosctl get loadedkernelmodules</code>

Run it before `helm install`. A Pod claiming a device pool with no members stays `Pending` with no other symptom, and the reason is upstream of Kubernetes entirely.

5.6.11 Support and debugging

There is no shell to gather evidence from, so gather it through the API:

```
install/fermiHDI-talos-discover -n 10.147.0.20 --dump-dir support/ # every raw response
install/fermiHDI-talos-discover --replay support/                # read it back later
talosctl -n 10.147.0.20 support -O support.zip                  # Talos's own bundle
talosctl -n 10.147.0.20 dmesg
talosctl -n 10.147.0.20 get pcidriverrebindstatus
```

`--replay` reads a dump instead of a machine and probes nothing, which makes an inventory reproducible after the fact and reviewable by someone who cannot reach the cluster.

5.6.12 Known limits

- **Neither the DPDK NIC path nor the device plugin has been exercised on real hardware from this tooling.** The mechanisms are the documented ones and the generated configuration is checked against the schema and rendered end to end, but "renders correctly" is not "works": the `accelerator` device class, the PF-rather-than-VF NIC pool, and SPDK attaching through a plugin-provided `/dev/vfio` node are all worth confirming on a real machine before anyone depends on them.
- The front door is still not deployed — same gap as the general-purpose chart. See [README.md](#).
- `helm/fermiHDI-talos` is a fork of `helm/fermiHDI` rather than a layer on top of it, so a fix to the DMS templates has to be made in both. That is deliberate for now: it keeps the working chart untouched while this one is new.

- `ssh_user` and `ssh_key` in `cluster.yaml` are ignored on Talos hosts. `validate-cluster` says so rather than letting them look meaningful.

5.6.13 Sources

- [Boot loader](#) — why `extraKernelArgs` is ignored under `systemd-boot`
- [Boot assets and the Image Factory](#) — schematics and `metal-installer` images
- [Configuration patches](#) — multi-document strategic merge, and the default PSA configuration
- [Editing machine configuration](#) — which fields apply without a reboot
- [PCIDriverRebindConfig](#)
- [v1alpha1 machine config reference](#)
- [Talos for Linux admins](#) — the `talosctl` equivalent of each familiar command
- [The `--insecure` flag](#) — what maintenance mode does and does not offer
- [longhorn#12709](#) — SPDK, `vfio_iommu_type1` and Talos

6. Using HD

6.1 Querying HD

Every query into an HD instance is the same shape underneath: a set of batches to read, a list of predicates the Storage Nodes evaluate before sending anything, and an address the records stream back to. That is the right interface for a store that moves records at line rate and the wrong one for the tools people already have, so HD offers the same query through several front doors. This page is the map: which door to use, what they have in common, and the three rules every one of them obeys.

6.1.1 The front doors

Door	Speaks	For	Page
SQL server (<code>hd_sql</code>)	PostgreSQL wire (5432), Arrow Flight SQL (31337), HTTP (8081)	BI tools, notebooks, JDBC and ODBC pipelines, anything with a database connector	SQL
GraphQL (<code>hd_graphql</code>)	GraphQL over HTTP	applications, with a schema generated from every dataset the Proxy serves	GraphQL
IP-flow GraphQL (<code>hd_ipflow_graphql</code>)	GraphQL over HTTP	dashboards and automation over the network-flow dataset: time series, top-N, aggregates	GraphQL
MCP server (<code>hd_mcp</code>)	the Model Context Protocol, over Server-Sent Events	AI assistants and agents that query on a person's behalf	MCP
Data Explore	a browser, on port 4321	analysts, with a visual query builder and charts over the MCP server	—
The SDK	C++, Go, Python, JavaScript, Rust, Julia, R, Scala, Mojo	your own service, when none of the above fits	SDK overview

All of them run **outside** the storage cluster and reach it through the HD Proxy, which authenticates the caller, applies policy and scatters the query to every Storage Cluster Controller. The records do not come back through the Proxy: the Storage Nodes stream them straight to the front door, which evaluates the rest of the query locally with the analytical engine embedded in the SDK. The SQL, GraphQL and MCP services are therefore ordinary services to place — behind the same authenticating front door as the Proxy, on a host that the Storage Nodes can reach on the data plane — and the [Administrator's Guide](#) has their ports.

The SDK alone has a second way in. A client built on it can skip the Proxy and register with the DMS as an `HD_CLIENT` node: it presents a registration token an administrator issued, receives a certificate from the DMS-CA, the dataset schema and the list of Storage Cluster Controllers, and submits its queries to those controllers itself. Such a client is inside the trust boundary — the Proxy's policy does not apply to it, the DMS's admission does — and it draws on the `HD_CLIENT` element of the licence, or the `HD_ANY` pool. This is the mode for a service that lives inside the deployment, and for a custom ingester built on the SDK, which joins a cluster the same way. It is available from C++, Python and (for querying) JavaScript; the bindings built on the C ABI work in proxy mode only. The [SDK overview](#) puts the two modes side by side. What that front door is, what it authenticates and which identity each door carries to the Proxy is on the [front doors page](#).

6.1.2 What every door has in common

The time window decides the cost. Records are stored in batches indexed by time, and a query names the batches it wants. A predicate on the dataset's batch-id field — usually the timestamp — becomes the window; a query that gives no window is either refused or answered over a default window that the service tells you about. Bound both ends.

Predicates the Storage Nodes can evaluate keep records off the network. Equality, not-equal, greater and less, on a declared field, against a literal. Everything else — `>=`, arithmetic, `NOT`, functions — is evaluated at the front door after the

records arrive. The answer is the same either way; only the volume transferred differs, because the front door re-applies the whole condition to what it receives. A partial pushdown makes a query slower, never wrong.

Sixteen branches. A Storage Node evaluates at most sixteen alternative predicate sets per query. An `OR` is one branch per side and an `IN` list one per value; past sixteen the door pushes nothing down and reads the whole window, and says so. A range is two predicates where a list would be twenty.

6.1.3 A query lasts as long as its connection

Records come back out of band: the Storage Nodes stream them straight to the front door, and nothing on that path tells the cluster whether anyone is still listening. The caller's control connection is what carries that signal, so a query lives exactly as long as that connection stays open. Two consequences follow, and the bundled front doors and the SDK already honour both:

- **Hold the connection open until every batch has arrived.** Closing it as soon as the query is accepted tells the cluster the caller is done, and the Storage Nodes stop before they have sent anything. A query is not a fire-and-forget submission followed by a separate collection; the same connection that starts it must stay open to receive it.
- **To stop early, cancel first.** A caller that abandons a running query sends a cancel and then closes. A connection that simply drops is treated as the caller having gone, and the query is stopped on every Storage Node it reached. A clean close with no cancel means the caller received everything it wanted.

A query is never ended on elapsed time alone: a legitimate query may run for a long time, and the cluster distinguishes a large query from an abandoned one by the connection, not by a clock.

6.1.4 Addresses

An `ipv4` field is a 32-bit unsigned integer and an `ipv6` field a 128-bit one; that is what the ingester stores, what the Storage Nodes compare, and what the SDK's engine sees. Every front door converts at its own boundary and nowhere else: a literal written as an address — `'10.0.0.1'`, `'2001:db8::1'` — becomes the integer before the query leaves the service, and a result column the schema calls an address is rendered as one on the way out. An IPv4 address against a sixteen-byte field is IPv4-mapped. The same conversion applies to a plain integer column that holds addresses, provided it is wide enough — four bytes for an IPv4, sixteen for an IPv6.

Ordering and comparison happen on the integer, so a range on an address column is numeric — `9.0.0.1` sorts before `10.0.0.1` — and only the presented value is text. **The SDK translates nothing:** a service or application built on it directly passes the integer and renders the result itself.

6.1.5 Enumerations

A dataset may declare an `enums` table: named lists of labels, each assigned an unsigned integer code, with one label as the default. A `uint8`, `uint16`, `uint32` or `uint64` field that names one stores the code, and only the code; the labels exist in the schema alone. Unlike an address, the code means nothing without that table, so here the SDK does translate, on the strength of what the schema declares: a result column comes back as its labels, text, with a code the table does not define rendered as the default label, and a filter on the field takes a label (or a defined code) and pushes the code to the Storage Nodes, which see only the integer.

Two consequences follow. `WHERE proto = 'tcp'` and `GROUP BY proto` read naturally in every front door, and ordering an enumerated column is by label, not by code. And a label the table does not define is not pushed at all: the default is for data, not predicates, so a mistyped label over-fetches and the engine then matches nothing, rather than quietly selecting the "unknown" rows. An application built on the SDK that wants the codes themselves turns label decoding off on its client; the filter behaviour is unchanged by that.

6.1.6 Time fields and intra-record time compression

An integer field may declare `time` with a unit, nanoseconds through days: its value is a count of that unit since the Unix epoch. A field carrying `time_reference` instead stores a signed delta, in a unit of its own, from another time field of the same record, so a handful of bytes hold what an absolute time needs eight for. The schema declares the compression, so here too the SDK translates: a result column comes back as the absolute time, a 64-bit integer in the field's own `time` unit or, when it declares

none, the source field's. On an array of integers every element expands against the same source. An application built on the SDK that wants the stored deltas turns expansion off on its client.

A filter on such a field is the one thing that is not translated: its value is compared with the stored delta, because the absolute time it stands for differs from record to record and there is no single byte pattern for the Storage Nodes to match. Filter on the source time field for a time range, and on the delta field for "within so many units of it". The SQL server is the exception: a `WHERE` on such a column is evaluated there on the presented time, never pushed down (see [SQL server](#)).

6.1.7 Chess notation

A `chess_move` field is stored as a 16-bit code that stands for one Standard Algebraic Notation move, a `chess_fen` field as a Forsyth-Edwards position packed into 32 bytes, and a `chess_move_cc` field as chess.com's own real per-ply move code -- two raw bytes, exactly as chess.com's own Parquet export stores them (see [Ingestion](#) for the layout). Unlike `chess_move`, `chess_move_cc` has no formula: its code is a reverse-engineered lookup table, not something computed from the move's parts, so a raw value only means something through that table, and a code not yet in it decodes to nothing -- the field still stores it losslessly.

`chess_move` and `chess_fen` render as their notation or as their raw stored form according to the field's own `render` property in the schema ("raw" or "decoded" , default "decoded") -- not a setting an application turns on or off on its client. Decoded, a result column of moves is the SAN text (`Nf3` , `exd5+` , `0-0`), an array of moves a list of them, and a position the full FEN string. Raw, a move column is the stored code (a plain integer) and a position column the 32 packed bytes -- the form `fen_occupied / fen_turn` filter pushdown already operates on directly, so a dataset declaring `chess_fen` as `raw` is a real performance choice, not just an alternate rendering.

`chess_move_cc` is different: every query service always hands it back raw -- chess.com's own 2-character code, exactly as stored, never reinterpreted as a number -- regardless of `render` (a field may still declare it, for forward compatibility, but it has no effect). Decoding it to long algebraic notation (`e2e4` , `g1f3` , `e7e8q` -- not SAN; check and mate are not part of chess.com's encoding) is an explicit capability a consumer calls itself against the codebook: `hd_sdk's chess_cc::decode_chess_cc()` in C++, or the equivalent client-side decode Data Explore's chess board graph does against `/api/chess-cc-codebook` . No query service does this decode on a caller's behalf.

A filter takes the notation too: an `EQUALITY` on a `chess_move` field with `Nf3` matches the records whose stored code is that move's, one on a `chess_fen` field with a FEN string matches the records holding exactly that position, and one on a `chess_move_cc` field matches its own raw 2-character code -- there is no way to filter it by long algebraic notation, since encoding LAN back into chess.com's code would need the reverse of a lookup table that only decodes today.

Beyond a whole position, the packed layout lets a few questions about a position be asked of the Storage Nodes directly, as a masked comparison on the bytes: whether a square is occupied (one bit of the occupancy mask) and which side is to move (one bit of the game state). The SQL server pushes `fen_occupied(position, 'e4')` and `fen_turn(position) = 'b'` that way, and evaluates the rest of the `fen_*` functions itself: `fen_piece(position, 'e4')` (the piece letter, empty for an empty square), `fen_castling`, `fen_en_passant`, `fen_halfmove`, `fen_fullmove` and `fen_piece_count`, plus `san_code` and `san_text` to move between a SAN string and its code. The GraphQL and MCP servers reach the same functions through their SQL paths. `chess_move_cc` has no equivalent function set today -- an equality filter on the raw code (above) is the whole of what the query path itself offers; decoding is a client-side capability, not a query-side one.

`chess_move_cc`'s decode table is reverse-engineered from chess.com's own real data, not a published chess.com specification, and covers standard chess only (no Chess960, bughouse, crazyhouse, three-check, king-of-the-hill or odds chess). It has no guarantee of staying unchanged in a future chess.com export.

Data Explore draws a game. Its Chess Board graph takes a position column as the starting board and a moves column as the game to replay from it, either one alone working too: a position on its own is shown as stored, moves on their own replay from the standard start. Pick a row, step through the plies or let them play, and the board follows, with the last move highlighted and the position's FEN beneath it. A move that is not legal where it stands ends the replay and is named.

6.1.8 Arrays

A `bytes` field may declare an array: an element type, how many elements the field holds at most, and for string or numeric elements the width of each. The field then holds a count followed by the elements, each stored exactly as a field of that type

would be, with the room past the count left zero, so a record's bytes are still a function of its value alone. The field's type stays `bytes`, so a consumer that never reads the declaration still gets the bytes it was promised.

On the query path the SDK hands the field back as a list of the element type, and every front door follows: a DuckDB `LIST` in the SQL server (`integer[]` in `information_schema`, a PostgreSQL array on the wire), a list in GraphQL, a JSON array from the MCP server and in Data Explore. An `enum` on the field labels each element.

A predicate on an array field asks whether *any* element satisfies it, and it is pushed to the Storage Nodes as such. The SDK turns `tags = 8` into an element group the storage node walks over the stored elements, matching a record when some element equals 8; `!=` matches when no element equals the value; `>` and `<` when some element does. In SQL that is `list_contains(tags, 8)` and `NOT list_contains(tags, 8)`, in GraphQL and the MCP server the same `EQUALITY`, `NOT_EQUALITY`, `GREATER` and `LESS` operators as on a scalar field, with a label where the elements carry an `enum`. A filter node indexes an array by its elements, so a contains predicate prunes batches the way a scalar equality does. An element wider than eight bytes (an `ipv6` or a wide string) is the one case the storage node cannot walk, and a predicate on such a field is evaluated after the fetch, with a notice where the front door has one.

6.1.9 Empty windows and excluded batches

A window can hold nothing — no batch was stored for it, or, on a cluster with a [filter node](#), every batch was proven unable to match. An aggregate over such a window still returns its row: a count is zero, a sum is null. A plain selection returns no rows.

The Proxy's answer to every query is a manifest rather than the records: which batches were found, which were missing, which a filter node excluded, and whether any Storage Cluster Controller timed out. The SQL, GraphQL and MCP services read it for you. A caller on the SDK reads it itself, and should hand the filter's verdict to the SDK so its false-negative check runs — a batch the filter excluded that delivers a record anyway is a defect the client is the only one positioned to see.

6.1.10 Choosing

Use the **SQL server** whenever the consumer already speaks SQL: it is the widest door and the one most tools reach without code. Use the **IP-flow GraphQL** service for the flow dataset specifically — its time series and top-N shapes are what a dashboard wants and would be several statements in SQL. Use the generic **GraphQL** service for an application that wants a typed schema over any dataset. Put an assistant on the **MCP server** rather than on raw SQL: its tools bound the time window and acknowledge the cost of a full scan on the caller's behalf. Reach for the **SDK** when you need the raw Arrow table, streaming record batches as they land, vector search or graph analysis over the result, or a language none of the services speak.

6.2 The SQL Server

`hd_sql` is a read-only SQL front door onto an HD instance. It accepts SQL over three client protocols at once and answers all three from one engine: a client sends a statement, the service works out which batches and which predicates the Storage Nodes should apply, has them stream only those records, and lets the analytical engine embedded in the SDK evaluate the rest. Nothing in a BI tool, a notebook or a JDBC pipeline has to know what an HD query is.

The rules shared by every query door — the time window, what pushes down, the sixteen-branch limit, addresses, empty windows — are on the [querying page](#) and are not repeated here.

6.2.1 Connecting

Port	Protocol	Clients
5432	PostgreSQL wire	<code>psql</code> , JDBC and ODBC drivers, DBeaver, Grafana's PostgreSQL source, Tableau
31337	Arrow Flight SQL over gRPC	columnar clients: ADBC, the Arrow Flight SQL JDBC driver, pandas and polars through ADBC
8081	HTTP	<code>POST /v1/query</code> with <code>{"sql": "..."};</code> <code>GET /v1/datasets</code> and <code>/v1/datasets/{id}</code> for the catalog
8080	HTTP	<code>/health</code> and Prometheus <code>/metrics</code>
6989	FHDWP	where the Storage Nodes stream results to; the service's own listener

```
psql "host=hd-sql port=5432 dbname=hd user=analyst sslmode=disable" \
-c "SELECT ip_src, sum(bytes) FROM ipflow WHERE timestamp > 1788660000 AND timestamp < 1788663600 GROUP BY 1 ORDER BY 2 DESC LIMIT 10"

curl -s http://hd-sql:8081/v1/query \
-d '{"sql": "SELECT count(*) FROM ipflow WHERE timestamp > 1788660000 AND timestamp < 1788663600"}'
```

```
from adbc_driver_flightsql import dbapi
with dbapi.connect("grpc://hd-sql:31337") as c:
    table = c.cursor().execute("SELECT * FROM ipflow LIMIT 5").fetch_arrow_table()
```

The service reports itself as PostgreSQL so that drivers work, and the PostgreSQL listener takes an optional password (`HD_SQL_PGWARE_PASSWORD_FILE`). It has no access control of its own beyond that: it is meant to run behind the same authenticating front door as the Proxy, which supplies the identity headers the Proxy's policy is evaluated against. A listener reachable from outside that front door gives every caller the access the service itself has, and the service says so in its start-up log when no password is set. The identity the Proxy evaluates for a query through this service is the service's own, not the SQL user's; see the [front doors page](#).

6.2.2 What is there

```
SHOW TABLES;
SELECT * FROM information_schema.columns WHERE table_name = 'ipflow';
```

Each dataset is one table, named by the dataset's id; its human-readable name works too. HD's types map to the nearest SQL type that can hold them, widened rather than reinterpreted so that a `uint32` above 2^{31} does not arrive as a negative number:

HD type	SQL type	Notes
<code>uint8, int8, int16</code>	<code>smallint</code>	
<code>uint16, int32</code>	<code>integer</code>	
<code>uint32, int64, timestamp, ipv4</code>	<code>bigint</code>	a timestamp is Unix seconds, not a SQL timestamp
<code>uint64, uint128, int128, ipv6</code>	<code>numeric</code>	no signed type is wide enough
<code>uint256, int256</code>	<code>numeric</code>	exact for storing, comparing, sorting and min/max; arithmetic over one is double precision
<code>numeric</code>	<code>numeric(p,s)</code>	
<code>float / double</code>	<code>real / double precision</code>	
<code>boolean, bytes, string</code>	<code>boolean, bytea, text</code>	a blob column holds the object key, not the object

An `ipv4` or `ipv6` column is presented as `inet` text and accepts an address literal in a predicate; comparison is on the integer underneath.

An enumerated column, an unsigned integer whose schema entry names one of the dataset's `enums`, is presented as `text`: its labels. A predicate takes a label, `WHERE proto = 'tcp'`, and the storage nodes filter on the code it stands for. A code the table does not define reads as the enumeration's default label, and `ORDER BY` on the column orders labels, not codes.

A delta-compressed time column, an integer whose schema entry carries `time_reference`, is presented as `bigint`: the absolute time the stored delta stands for, in the column's own `time` unit or the source field's. A predicate on such a column is evaluated by the SQL server on the presented time, after the storage nodes have answered, and is never pushed down: the absolute time differs from record to record and has no byte pattern a node could match. It narrows the result, not the read, so a time range still belongs on the source field.

A chess column is presented as text: a `chess_move` as its Standard Algebraic Notation (Nf3) and a `chess_fen` as its Forsyth-Edwards string, and a predicate on either takes the notation, `WHERE opening_move = 'e4'` or `WHERE position = '<FEN>'`, pushed to the storage nodes as the code or the packed position. Two position predicates are pushed as masked comparisons on the packed bytes: `fen_occupied(position, 'e4')` (also `NOT, = FALSE`) and `fen_turn(position) = 'b'`. The other position functions -- `fen_piece(position, 'e4')`, `fen_castling`, `fen_en_passant`, `fen_halfmove`, `fen_fullmove`, `fen_piece_count`, and `san_code / san_text` for moves -- are evaluated by the server over the rows the nodes return.

An array column, a `bytes` field whose schema entry declares an `array`, is presented as its element type followed by `[]` -- `integer[]`, `text[]` -- a DuckDB `LIST` that the PostgreSQL wire carries as an array literal (`{1,2}`) with the matching array type, and the HTTP API as a JSON array. DuckDB's list functions apply: `list_contains(tags, 'vip')`, `len(tags)`, `unnest(tags)`. `list_contains(col, literal)` (or `array_contains`), and `NOT list_contains`, are pushed to the storage nodes as a walk over the stored elements, so they narrow the fetch like a scalar equality; the literal may be a label where the elements carry an enumeration. Any other predicate on an array column (a comparison against the whole column, `len`, a nested expression) is evaluated by DuckDB over the fetched rows.

6.2.3 What SQL is accepted

The statement is evaluated by DuckDB, so the dialect is DuckDB's: aggregates, window functions, `CASE`, `GROUP BY`, `ORDER BY`, `LIMIT`, the full expression language. What the service constrains is the shape:

- **One table per statement.** No joins, no subqueries in `FROM`, no `UNION` across tables.
- **Read-only.** `INSERT`, `UPDATE`, `DELETE` and DDL are refused. Data enters through [ingestion](#).
- `SET`, `BEGIN`, `COMMIT` and `SHOW` are accepted and acknowledged. There are no transactions to be inside of.

A tool that assumes a full PostgreSQL will find gaps: no `pg_catalog` beyond the `information_schema` views above, no cursors, no `COPY`, no server-side prepared statements with binary parameters, and no out-of-band cancellation — closing the connection cancels the query.

6.2.4 Notices

The service tells you when it answered something other than exactly what you asked, as a PostgreSQL `NOTICE`, in the `notices` array over HTTP, and in its log for Flight SQL:

- **No time bound was given**, so the default window was read (the last hour, unless `HD_SQL_DEFAULT_LOOKBACK_SEC` says otherwise).
A one-sided bound counts as none: `timestamp > x` alone cannot be turned into a list of batches. A statement with no bound at all reads the entire dataset and says so.
- **A predicate could not be pushed to the Storage Nodes** and was applied after the transfer. `>=` and `<=` are rewritten as the strict comparison beside them, which is exact on whole-number columns and left local on `float` and `double`; `NOT` anywhere disables pushdown; one `OR` branch the planner cannot read leaves the whole clause unconstrained.
- **The clause needed more than sixteen branches**, so none were pushed.

6.2.5 Errors

What happened	SQLSTATE	gRPC	HTTP
Statement is not supported	0A000	Unimplemented	400 unsupported_statement
Unknown table	42P01	NotFound	404 unknown_dataset
Could not parse the statement	42601	InvalidArgument	400 invalid_statement
Timed out waiting for the Storage Nodes	57014	DeadlineExceeded	504 query_timeout
Query Proxy unreachable	08006	Unavailable	502 upstream_unavailable

A timeout means the Storage Nodes did not deliver every batch the Proxy said existed within `HD_SQL_QUERY_TIMEOUT_MS` (five minutes by default). The query is cancelled at the Storage Cluster Controllers when that happens.

6.2.6 Running it

Setting	Default	
HD_PROXY_ENDPOINT	—	the HD Proxy, e.g. <code>https://proxy.hd.internal:6986</code>
HD_DMS_ENDPOINT , HD_DMS_CA_FILE , HD_DMS_CLIENT_CERT_FILE , HD_DMS_CLIENT_KEY_FILE	—	where the dataset schema comes from, over mTLS. Without them the service takes the Proxy's copy
FERMIHDI_DATA_PLANE_SUBNET	10.150.0.0/24	the CIDR whose local address is advertised to the Storage Nodes; <code>FERMIHDI_DATA_RETURN_HOST</code> overrides discovery outright
FERMIHDI_DATA_RETURN_PORT	6989	where results arrive
HD_TIME_INDEX_INTERVAL	60	the batch width in seconds — must match the dataset
HD_SQL_DEFAULT_LOOKBACK_SEC	3600	the window a query with no time bound reads
HD_SQL_QUERY_TIMEOUT_MS	300000	the whole pipeline: dispatch, the wait for every Storage Node, the local evaluation
HD_SQL_MAX_ROWS	0	a cap on rows returned; 0 is unlimited
HD_SQL_QUERY_QOS	1	priority weight 1-255, sent to the Storage Nodes
HD_SQL_PGWIRE_ADDRESS , HD_SQL_FLIGHTSQL_ADDRESS , HD_SQL_HTTP_ADDRESS	:5432 , : 31337 , :8081	each listener can be moved or disabled (<code>HD_SQL_*_ENABLED</code>)

The service reaches the Storage Nodes' results on the data plane, so it is placed like any query client: on a host the Storage Nodes can reach, with the data-plane subnet declared. The address it advertises is written into every query; if it is wrong the query is accepted and the results go nowhere, with no error on either side, which is the first thing to check when every query times out. `/health` reports the Proxy connection and the schema source.

6.3 The GraphQL Services

Two GraphQL front doors sit on the HD Proxy. One is generic: its schema is generated from every dataset every Proxy it knows serves. The other is purpose-built for the network-flow dataset and speaks in the shapes a dashboard wants — time series, top-N, aggregates. Both translate a GraphQL query into the HD query underneath, receive the records directly from the Storage Nodes, and evaluate the rest locally with the SDK's engine. The rules they share with every other door are on the [querying page](#).

6.3.1 The generic service: `hd_graphql`

The service reads a proxy list and builds its schema from what each Proxy's `/v1/hd/schema` returns, watching the file for changes:

```
# proxies.yaml
proxies:
  prod:
    url: https://proxy.hd.internal:6986
```

Every dataset of every Proxy becomes a query field, `<proxy>.<dataset>`, with an object type carrying the dataset's fields (integers as `Int`, floats as `Float`, booleans, everything else as `String`) and an enum of field names for filters. The argument is one `queryElements` object:

Argument	
<code>startTime</code> , <code>endTime</code>	the window, in Unix seconds. Given either, the batches for that window are named; given neither, the whole dataset is read — the service acknowledges the full scan on the caller's behalf, because it has no interface of its own to ask through
<code>filters</code>	groups of <code>{fieldName, op, value}</code> , AND within a group and OR between groups, <code>op</code> one of <code>EQUALITY</code> , <code>NOT_EQUALITY</code> , <code>GREATER</code> , <code>LESS</code> . These are what the Storage Nodes evaluate, except a condition on a time stored as a delta from its batch, which is applied to the fetched records
<code>queryQos</code>	priority weight 1–255

```
query {
  prod {
    ipflow(queryElements: {
      startTime: 1788660000, endTime: 1788663600,
      filters: [{ andFilters: [{ fieldName: ip_src, op: EQUALITY, value: "10.0.0.1" }] }]
    }) { timestamp ip_src ip_dst port_dst bytes }
  }
}
```

Three mutations are escape hatches for a client that wants the SDK's engines directly on the records already fetched:

`executeRawSQL`, `executeRawGraph` and `executeRawVector`, each taking a `proxy` and a `query` string and returning the engine's answer as text.

An address literal in a filter is converted to the integer it is before the Storage Nodes see it, and an address column is rendered as an address in the result. `healthCheck` is the liveness field. The service listens on `GRAPHQL_LISTEN_PORT` (8083 by default) and reads `PROXY_CONFIG_PATH`; a GraphiQL page can be served beside it. It receives results on `FERMIHDI_DATA_RETURN_PORT` (6987) at the address it discovers inside `FERMIHDI_DATA_PLANE_SUBNET`, or at `FERMIHDI_DATA_RETURN_HOST` when discovery cannot pick the right interface.

6.3.2 The IP-flow service: `hd_ipflow_graphql`

One dataset, three query fields, at `POST /query` on `HD_SERVER_PORT` (8080):

Field	Returns
<code>getFlowsTS(queryElements, paging)</code>	the matching flows, or their aggregates, as a time series
<code>getFlowsTopXTS(queryElements, topXElements, paging)</code>	the top N by the ordering given, as a time series — for a "top talkers" panel
<code>getScalers(queryElements, dataPrepFilters)</code>	single figures over the selection: the maximum, the total

`queryElements` carries the whole question:

Part	
<code>when</code>	<code>fromDateTime</code> and <code>toDateTime</code> , each as an ISO string, Unix seconds or milliseconds. Both ends
<code>filters</code>	groups of <code>andFilters</code> , OR-ed together; each <code>{fieldName, type, value}</code> with type <code>EQUALITY</code> , <code>GRATER</code> or <code>LESS</code> . A <code>value</code> written as an address is converted for you
<code>aggregates</code>	<code>{fieldName, function, returnType}</code> — <code>SUM</code> , and the rest of the aggregate set — computed after the records arrive
<code>groupBy</code>	field names to group the aggregates by
<code>rollUp</code>	<code>{intervalMs}</code> : the time-series bucket width
<code>timeoutMs</code> , <code>queryQos</code>	the wait, and the priority weight

`paging` is `{first, skip}` and is honoured: the page asked for is the page returned. Address fields — `ipSrc`, `ipDst`, `peerIpSrc` — come back as addresses, an IPv4-mapped value as its dotted IPv4.

```
query TopTalkers($from: Float!, $to: Float!, $interval: Float!) {
  getFlowsTS(queryElements: {
    when: { fromDateTime: { fromDateTimeMs: $from }, toDateTime: { toDateTimeMs: $to } },
    filters: [{ andFilters: [{ fieldName: asSrc, type: GRATER, value: "0" } ] }],
    aggregates: [{ fieldName: bytes, function: SUM, returnType: uint64 },
                  { fieldName: packets, function: SUM, returnType: uint64 }],
    groupBy: [asSrc, asDst, ipSrc, ipDst],
    rollUp: { intervalMs: $interval }
  }) { timestampMS asSrc asDst ipSrc ipDst bytes packets bitsPerSecond packetsPerSecond }
}
```

The service is configured by `HD_`-prefixed environment variables or a JSON file: `HD_FERMIHDI_HD_PROXY_ENDPOINT` is the Proxy, and `HD_SERVER_AUTH_ENABLED` with `HD_SERVER_API_TOKEN_SECRET` turns on bearer-token checking for a deployment that does not put it behind the authenticating front door. It shuts down cleanly on `POST /v1/shutdown`, draining in-flight queries first. Its data-return settings are the same three as the generic service's.

6.3.3 Placing either

Both are engine roles in the installer's terms — they carry the SDK, so they want 2 MB hugepages and pinned cores like the other engine roles, and the [Administrator's Guide](#) sizes them. Both advertise a data-return address to the Storage Nodes and both fail silently when it is wrong: the query is accepted, the results go to an address nobody is listening on, and the caller waits. The [operators manual](#) walks through choosing it.

6.4 The MCP Server

`hd_mcp` puts an HD instance in front of an AI assistant. It speaks the Model Context Protocol over Server-Sent Events, so any agent framework that can call MCP tools can query the instance without being taught what an HD query is — and without being handed a raw database connection. The service runs outside the storage cluster, reaches it through the HD Proxy like every other door, and evaluates the SQL part locally with the SDK's engine. The rules it shares with the other doors are on the [querying page](#).

6.4.1 The tools

Two tools, and the split between them is the point:

`execute_hd_query` fetches live records and answers a question about them.

Argument	
<code>proxy_name</code>	which Proxy, by the name given in the proxy list
<code>sql_query</code>	the SQL to evaluate over the records that arrive, in DuckDB's dialect
<code>start_time, end_time</code>	the window, in Unix seconds. The end is clamped to now
<code>query_qos</code>	priority weight 1-255

The time window is what keeps an assistant's question affordable. With one, the tool names exactly the batches of that window; without one it reads the whole dataset, and the tool acknowledges that full scan on the agent's behalf, because an agent has no way to be asked. Prompt the assistant to always give the window. The interactive alternative is the stream endpoint on the second port, where an application such as Data Explore relays the *person's* answer to the full-scan question instead.

`execute_raw_sql_query` runs SQL over the records the previous call already fetched, with no round trip to the storage cluster. It exists so an assistant can ask several questions of one fetch — group it another way, filter it further — without paying for the transfer again. It takes `proxy_name` and `sql_query`.

An address literal in either tool's SQL — `'10.0.0.1'` against a column the schema types as an address, or against an integer column wide enough to hold one — becomes the integer before the engine sees it, and an address column in the result is rendered as an address.

6.4.2 Running it

```
# proxies.yaml
proxies:
  prod:
    url: https://proxy.hd.internal:6986
```

Setting	Default	
<code>PROXY_CONFIG_PATH</code>	<code>proxies.yaml</code>	the Proxy list, watched for changes; each Proxy's schema is fetched from its <code>/v1/hd/schema</code>
<code>MCP_LISTEN_PORT</code>	8082	the MCP endpoint, over Server-Sent Events
<code>MCP_STREAM_PORT</code>	8083	the interactive query stream Data Explore uses
<code>FERMIHDI_DATA_RETURN_PORT</code>	6987	where the Storage Nodes stream results
<code>FERMIHDI_DATA_PLANE_SUBNET /</code> <code>FERMIHDI_DATA_RETURN_HOST</code>	<code>10.150.0.0/24 /</code> <code>discovery</code>	the address advertised to the Storage Nodes — wrong, and every query waits for results that went elsewhere

An assistant's client configuration points at the MCP endpoint, for example `http://hd-mcp:8082`. The service has no authentication of its own: place it behind the authenticating front door, and give the assistant the identity the Proxy's policy should evaluate —

the [front doors page](#) says how identity reaches the Proxy from here. It is an engine role for placement — it carries the SDK — and the [Administrator's Guide](#) sizes it.

6.4.3 Data Explore

Data Explore is the browser front end over this service: a visual query builder, charts, and the interactive full-scan prompt. It reaches the MCP server on both ports (`FERMIHDI_MCP_ENDPOINT` , `FERMIHDI_MCP_STREAM_ENDPOINT`), serves on port 4321, and relies entirely on the identity headers the front door sets, so it too belongs behind it.

6.5 Ingestion

Records enter an HD instance through the Message Bus Ingestor: one engine role that reads a source, maps each record onto the dataset schema, groups records into batches and streams them to a Storage Cluster Controller over the FermiHDI HD Wire Protocol (FHDWP). Everything downstream — the fan-out to every Storage Node, the hash chain, the filter node — starts from what the ingester sends. This page is what an operator and a data owner need to know about it: where it reads from, what it accepts, and how it decides where one batch ends and the next begins.

6.5.1 Sources

One ingester reads one source, chosen by `ingression_type`:

Source	Setting	Reads
Kafka	<code>kafka.brokers</code> , <code>kafka.topic</code> , <code>kafka.group_id</code>	a topic, as a consumer group member
RabbitMQ	<code>rabbitmq.url</code> , <code>rabbitmq.queue</code>	a queue
MQTT	<code>mqtt.address</code> , <code>mqtt.topic</code>	a topic
Local directory	<code>file.directory</code> , <code>file.sort_by</code>	every file in the directory, one at a time, deleting each once its records are acknowledged
S3 or a compatible object store	<code>s3.bucket</code> , <code>s3.prefix</code> , <code>s3.region</code> , <code>s3.endpoint</code>	every object under the prefix, one at a time

Kafka is what the installer wires. The others are configured through the ingester's own settings, which reach it as environment variables, a JSON file or command-line flags in that order of precedence; the full table is in the [configuration reference of the ingester](#).

A message from a broker is one record, in JSON or in the RowBinary form (`input_format`), or a CSV row. A file or object is many records, and its extension says how they are encoded.

6.5.2 File and object formats

Every format decodes to one JSON record per row and then takes the same path a broker message takes, so the schema mapping, address parsing, batch rotation and time compression exist once. The extension is the format:

Extension	Format	Notes
<code>.json</code>	one JSON object, or an array of them	
<code>.jsonl</code> , <code>.ndjson</code>	one JSON object per line	
<code>.csv</code>	a header row, then one record per row	cells that read as numbers become numbers
<code>.parquet</code> , <code>.pq</code>	Apache Parquet	a list column is an array (what an array field takes), a dictionary-encoded column is its value; a timestamp or date column is converted to the unit the schema declares for the field (<code>time.unit</code> , or the batch field's <code>time_index_interval_unit</code>), so a PostgreSQL export in microseconds lands in a dataset keyed on seconds as seconds
<code>.orc</code>	Apache ORC	
<code>.avro</code>	Avro object container file	the writer's schema travels in the file; a union is its branch, so a nullable <code>long</code> is 5 or null
<code>.pb</code> , <code>.protobuf</code>	Protocol Buffers	needs a descriptor set (<code>formats.protobuf_descriptor_file</code> , written by <code>protoc --descriptor_set_out --include_imports</code>) and the message type (<code>formats.protobuf_message_type</code>). Messages are length-delimited unless <code>formats.protobuf_delimited</code> is false. 64-bit integers stay numbers
<code>.xml</code>	XML	each child of the root is a record, or every element named <code>formats.xml_record_element</code> wherever it sits; attributes become <code>@name</code> , repeated children become arrays, text that reads as a number or boolean becomes one

Files are taken in `sort_by` order — by name, or by modification time — and a file with any other extension is left alone. A file is removed only after the storage cluster has acknowledged its records.

6.5.3 Mapping records onto the schema

The dataset schema names every field, its type and its byte position in the fixed-width record. A JSON key that matches a field name is written there; keys the schema does not name are ignored, and fields no key supplies are zero. Four things deserve a word:

- **Addresses.** A field the schema types `ipv4` is a 32-bit unsigned integer and one typed `ipv6` a 128-bit one, both big-endian on the wire. The ingester accepts either the integer or the address written as text — `10.0.0.1`, `2001:db8::1` — and stores the integer. An IPv4 address given for an `ipv6` field is stored IPv4-mapped (`::ffff:10.0.0.1`). Every query service renders the integer back as an address; the SDK does neither and sees only the integer.
- **Time.** A timestamp field is the integer it is, in the unit the dataset declares (`time_index_interval_unit`: seconds by default, or milliseconds, nanoseconds, minutes). It is not parsed from a date string.
- **Text.** A `string` field is stored UTF-32, four bytes per character, whatever `string_encoding` names as the source, so that a byte-offset filter is always character aligned. The exception is a field whose `string_encoding` is `ascii`: it is stored one byte per character, since every ASCII character already is one, and holds four times as many characters in the same bytes. A character outside ASCII arriving for such a field is stored as `?`, and the ingester says so.
- **Enumerations.** A dataset may declare an `enums` table: named lists of labels, each assigned an unsigned integer code, with one label as the default. A `uint8`, `uint16`, `uint32` or `uint64` field that names one in its `enum` property stores the code, four bytes at most for a `uint32` rather than the sixty-four a sixteen-character string would occupy. The ingester accepts the label, matched exactly and case-sensitively, or the decimal text of a defined code. Anything else stores the default's code and is counted, because a producer that spells a label differently from the schema maps every record to the default, and that should be visible, not silent. Every query service renders the code back as its label.
- **Time fields and intra-record time compression.** An integer field may declare `time` with a unit (nanoseconds through days): its value is a count of that unit since the Unix epoch, and the unit is what lets another field be stored as a delta from it. A field carrying `time_reference` names such a source field and a delta unit, and stores not a time of its own but the signed difference from the source's value in the same record: an event a few seconds after the record's own timestamp fits an `int16` of seconds or an `int32` of milliseconds rather than eight bytes. The value arrives absolute, in the field's own `time` unit or, when it declares none, the source's; the ingester computes and stores the delta. A delta the field cannot hold is clamped to its nearest bound and counted, and a record that carries no value for the source stores zero and is counted, since either is a schema-sizing or producer problem someone should hear about. On an array of integers both declarations apply to each element, measured from the same source. Every query service hands the field back expanded, as the absolute time; a filter on it, however, compares with the stored delta.
- **Chess notation.** A `chess_move` field is a 16-bit code that stands for one Standard Algebraic Notation move (`Nf3`, `exd5`, `e8=Q`, `0-0`, with a `+` or `#`): the value arrives as the SAN text and the code is computed from it, with no table to ship, so every service reads the same move. Text the code cannot carry -- a move disambiguated by both file and rank, or not a move at all -- stores zero, "no move", and is counted. An array of `chess_move` may be given a whole PGN movetext (`1. e4 e5 2. Nf3 ...`) instead of a list: move numbers, comments, variations and the result are dropped and the moves are stored in order. A `chess_fen` field is a Forsyth-Edwards position packed into 32 bytes: an 8-byte occupancy bitmask (a1 is bit 0), 20 bytes of 5-bit piece tokens in occupancy order, and 4 bytes of game state (side to move, castling rights, en passant file, the halfmove clock in 8 bits, up to 255, and the fullmove number in 15 bits, up to 32,767; a larger value is stored as the largest). The value arrives as the FEN text; a string that is not a position stores zero, "no position", and is counted. A `chess_move_cc` field is chess.com's own real per-ply move code: two raw bytes, exactly as chess.com's own Parquet export stores them. Unlike `chess_move` there is no formula to compute the code from -- the value arrives as that same 2-character code and is stored verbatim, unchanged, so it never fails to represent a move the way `chess_move` can. An array of `chess_move_cc` may be given the source string directly instead of a list: it is split into 2-character chunks in order, not tokenized as PGN movetext the way `chess_move` is, since chess.com's own export has no PGN structure to drop. Whether a `chess_move` or `chess_fen` field is handed back on the query path as its notation or as its raw stored form is the field's own `render` property in the schema (`"raw"` or `"decoded"`, default `"decoded"`) -- not a setting the ingester has any part in. `chess_move_cc` is different: every query service always hands it back raw, its own 2-character code, whatever `render` says (or whether the field declares one at all) -- decoding it to long algebraic notation is an explicit capability a consumer calls itself, not something rendered on the query path automatically. See `querying.md`.
- **Arrays.** A `bytes` field may declare an array: an element type, the most elements the field holds, and for string or numeric elements the width of each. The value arrives as a JSON array (`"tags": ["vip", "eu"]`), and the ingester writes the count followed by the elements, each exactly as a field of that type would be written, with the rest of the field zero. A scalar is taken as a one-element list. Elements past the field's capacity are dropped from the end and counted, because a producer whose lists outgrow the field is a schema-sizing problem someone should hear about, and a dropped record would be worse than a shortened list. An `enum` on the field applies to each element.

6.5.4 How batches are formed

A batch is the unit the storage cluster stores, hashes, indexes and serves. It carries a 64-byte id, and the ingester decides the id of every record it sends. When the id changes between one record and the next the open batch is closed with an end-of-batch marker and the next one opened — and **a closed batch is never reopened**. There are three ways the id is chosen, and the dataset schema picks among them.

A time field, rounded to the time index. The schema's `batch_id_field` names a time field and sets `round_batch_id_field`; every record's time is rounded down to `time_index_interval` from the Unix epoch. With an interval of 60 seconds every batch is the top of a minute, whichever second of the minute the first record arrived in, and the batch rotates each time the minute changes. This is how a time-series dataset is stored, and it is what lets a query name its batches from a time window alone. The time field and the interval are in the same unit. The schema's flag is enough on its own; the ingester's own `round_batch_id_field` setting only forces rounding on.

Any field's value, as it is. Without `round_batch_id_field` the field's value is the batch id verbatim: a string field is the id text itself, up to 64 bytes; an integer field is its decimal text; a `uint64` field is written as eight big-endian bytes. Tenant, customer, region and device ids all work this way. Two things follow. Records have to arrive grouped by that key, or the ingester closes and reopens the same batch over and over. And the query side has to be able to name the ids — a time range enumerates itself, a key set does not, so a client passes each key it wants through the SDK's string or number batch-id generator.

The ingester's clock. With no `batch_id_field`, or for a record that lacks the field, the id is the ingest time rounded to the interval, as a `uint64`. The ingester's `batch_interval_seconds` overrides the interval. This is what a Kafka ingester does by default.

Record size and the data-plane frame

A record travels whole inside one FHDWP frame. The standard 1200-byte frame carries 1133 bytes of records, and a jumbo frame (`FERMIHDI_JUMBO=1` on the SCC, every Storage Node and the ingester) carries 8433. A record wider than the payload travels alone in a frame of its own, so a wide dataset works in both modes; jumbo frames only pack more records into each frame. The widest record a dataset may declare is 8433 bytes, and the ingester refuses to start with a wider one.

Closing the last batch

A batch closes when the first record of a later batch arrives. When records stop, the ingester closes the open batch after a timeout instead (`batch_close_timeout_mode`):

Mode	Closes the open batch after
<code>dynamic</code> (default)	the longest time any one file or object has spanned so far, measured on its batch field, and never less than one batch interval; <code>batch_close_timeout_sec</code> until a file has been read
<code>manual</code>	<code>batch_close_timeout_sec</code> (default 300) with no record
<code>off</code>	never; only a later batch closes it

A closed batch is never reopened. Records that arrive for one, from a file delivered out of time order or the second half of a batch split across files that arrived further apart than the timeout, are refused by the storage cluster, and the ingester logs them as an error.

Files that span several batches

A file's rows are rarely in time order, and a closed batch is never reopened, so whenever the batch id is a rounded time field a file or S3 source automatically orders each file's records by it before handing them over — no setting to turn on. A Parquet file holding five minutes of records yields five batches, one per minute, in order, however its rows were written. The ordering applies to numeric keys under time rounding; a file keyed by an unrounded string or integer arrives in file order and should be written grouped by that key — the ingester does not reorder it for you.

6.5.5 A custom ingester

The ingester is one program built on the SDK's ingest client, and a customer can build another: `FermiHDIIngestClient` registers with the DMS as an `HD_CLIENT` node (a registration token, a certificate from the DMS-CA, the SCC targets in return), opens the FHDWP data plane on the interface it is given, and sends its batches straight to the Storage Cluster Controllers — the same path the ingester uses, without any of its sources. It draws on the `HD_CLIENT` element of the licence, or the `HD_ANY` pool. The ingest client is available from C++ and Python; the [SDK overview](#) has the modes and the bindings that support each.

6.5.6 Verifying a source before it feeds a deployment

`--dry-run` (or `FERMIHDI_DRY_RUN=1`) runs the same published ingester image with the same source and format handling described above, but registers with no DMS, connects to no SCC and needs no licence. It reads, decodes and maps records exactly as a real ingest would — the same field-name matching, the same batching, the same per-field warnings — and reports a summary instead of sending anything anywhere:

```
docker run --rm \
-v /path/to/corpus:/data:ro -v /path/to/schema.json:/schema.json:ro \
cr.fermihdi.io/hd/ingester:<tag> \
--dry-run --ingression-type file --file-dir /data --schema /schema.json
```

Every source and format flag documented above works the same way under `--dry-run`. Two flags are specific to it: `--dry-run-idle-exit-sec` (default 5; stop after this many seconds with no new record, including when none ever arrived, so a misconfigured source fails fast rather than waiting forever — `0` runs until Ctrl-C, for watching a live stream as long as wanted) and `--dry-run-max-records` (default unlimited; stop after this many records, for a quick check against a large corpus).

The run ends with a summary — records read, batches formed, and the same mapping-warning counters [What to watch](#) describes below (enum defaults, array truncation, clamped time deltas, unmapped chess or uuid values) — and exits non-zero if nothing was read at all. A clean run, with records read and every warning counter at zero, is the strongest evidence this pipeline has that a source and a dataset schema are actually compatible, without touching a real deployment.

6.5.7 What to watch

The ingester registers with the DMS like every node and serves `/healthz` and `/metrics` on its health port (8081 by default). The figures that matter are records in, batches closed, and the acknowledgement lag from the storage cluster; a file source that stops consuming files has usually stopped receiving acknowledgements, and the SCC's log says why. The engine sizing — two cores at the floor, four allocated, 300 × 2 MB hugepages — is in the [Administrator's Guide](#).

7. Operations

7.1 HD: Operators & Admins Manual

This manual is about running an instance. Installing one is [install/ADMIN_GUIDE.md](#), which covers host requirements, sizing, the cluster definition, and the Ansible and Kubernetes paths. The invariants below — the cpuset split, the core counts — are the ones the installer enforces, so a deployment it produced already satisfies them.

HD is a high-performance, strictly segmented Data Management System built for massive-scale analytics and isolated tenancy. It heavily leverages C++ native processing (DuckDB, DPDK, NORM multicast) paired with Go-based control planes (DMS-Core, Proxy) for orchestration.

7.1.1 High-Level Architecture

The system uses three distinct planes isolated at the network level:

1. **Control Plane (Port 6986):** Over-the-wire HTTPS mTLS APIs for node registration, policy distribution, and query telemetry orchestration. Records never pass through this network.
2. **Data Plane (Port 6987):** Unidirectional, ultra-fast binary streaming using the FermiHDI HD Wire Protocol (FHDWP). Data flows natively from Storage Nodes back to the requestor over TCP.
3. **Internal DMS / CA Plane (Port 6985 / 6989):** Secure channels exclusively connecting the Datastore Management System components to bootstrap new nodes and provision PKI identities.

7.1.2 Core Component Lifecycles

Datastore Management System (DMS)

The DMS is the single source of truth for the HD Instance. * **DMS-Core:** The orchestration engine. It distributes dataset schemas and routing topologies to proxies. * **DMS-CA:** Issues cryptographic certificates for new nodes joining the cluster. * **DMS-WebUX:** Provides the operator UI for dataset administration. * **OTEL Collector:** Every service in the instance reports traces, metrics and logs to it. It runs under DMS-Core's management: DMS-Core renders its configuration (the exporters an administrator adds on the OTEL Collector page and, only while the administrator has opted in to sharing and the deployment is registered, the pipeline to FermiHDI) and the collector restarts on it. Ports 4317 (gRPC) and 4318 (HTTP) on the DMS host. * **Crash vault:** Every crash report a service posts (the engine nodes through Crashpad, the DMS and Proxy through the Go runtime, the WebUX through Node) is held on the DMS host, symbolicated against the symbol sets shipped beside the engine images (`<image>-symbols:<tag>`), and forwarded to FermiHDI only as an operator decides on the WebUX Telemetry page. The nodes reach it on port 3001 of the DMS host; the WebUX on 3002.

HD Proxy (Query Gateways)

Acts as the ingress gateway for the Control Plane. * Evaluates Attribute/Policy-Based Access Control (PBAC) using Open Policy Agent (OPA). * Dynamically rewrites queries with binary bitmask filters. * Forwards authorized payloads to Storage Cluster Controllers.

Storage Cluster Controllers (SCC)

SCCs orchestrate queries across their domain. * They operate as Fan-Out nodes, broadcasting queries to their underlying Storage Nodes (SNs). * SCCs ingest raw records, hashing and distributing them identically across their SNs over the Data Plane (TCP, port 6987). * NORM (NACK-Oriented Reliable Multicast, UDP port 6988, group `239.255.0.1`) is implemented in `libfermihdi_transport` and enabled by default there, but the SCC **deliberately disables it** (`norm_enabled = false`) and fans out over TCP instead. Treat 6988 as reserved: nothing in 2.0.x sends on it, so a firewall rule permitting it is not required for a working deployment.

Storage Nodes (SN)

The raw engine layer, executing the native C++ retrieval pipelines. * They map raw `FHDWP` payloads directly to DuckDB Apache Arrow buffers. * Once data is found, they connect directly to the Client/Target Data Port to stream the response, bypassing the Proxy completely.

BloomRF Filter Node (optional)

An SN-shaped engine role that keeps a filter per batch instead of the records, so the SCC can leave batches unread. * It registers as node type `HD_BLOOMRF` inside one Storage Cluster, receives the SCC's ingest broadcast and verifies every batch's hash with the SCC before its filter becomes usable. * The SCC learns of it once, when its data plane starts, and the DMS holds the SCC's targets until every provisioned SN and filter node is ready. Adding or removing one on a running cluster therefore ends with an SCC restart. * Its filters persist to the node's local storage and are reloaded at start, so a restart keeps every batch it held; the store is bounded by a retention count. Losing a filter costs selectivity, never answers. * It needs no licence slot of its own. The `HD_BLOOMRF` element is taken by each SCC that runs the filter in `active` mode and returned when it leaves it; an SCC refused a slot runs the filter `passive` and logs why. * The mode — `off`, `passive`, `active` — is a property of the HD instance, set on the deployment's card in the WebUX and pushed to every SCC together, and reported on every query. The [filter node page](#) has the modes, sizing and metrics; the [WebUX walk-through](#) has the pages.

Message Bus Ingester

Reads one source — a Kafka, RabbitMQ or MQTT broker, or files from a directory or an S3-compatible store — and streams batches to an SCC. * It registers with the DMS like every node and is handed the record size, the SCC's address and the hash election. * A file source removes each file only once the SCC has acknowledged its records, so a directory that stops draining points at the storage cluster, not the ingester. The [ingestion page](#) covers formats and batching.

Query services

`hd_sql`, `hd_graphql`, `hd_ipflow_graphql` and `hd_mcp` are stateless services outside the storage cluster. * Each reaches the instance through a Proxy it is configured with, fetches the dataset schema from the DMS or the Proxy at start-up, and rebuilds its view when the schema changes. * Each listens for query results itself on the data plane, so every one of them needs the data-plane subnet declared and a host the Storage Nodes can reach — the section below is about exactly that. * None has access control of its own beyond a password on the SQL server's PostgreSQL listener; all belong behind the authenticating front door, like the Proxy. The [front doors page](#) is the contract for that door and for the DMS's.

7.1.3 Query Result Return Path

Query results do not travel back the way the query came. The SN opens a **new** outbound FHDWP connection to a target the *caller* nominated, so that target must be an address the SN can reach and a port something is listening on. Getting this wrong fails silently: the query is accepted, the SN streams to an address nobody is listening on, and the caller simply waits.

How the target is chosen

Every query carries a required `target` array (`schemas/hd_query_schema.json`), and each entry pairs an address with a **required** `port_number`. There is no default in the schema — the caller must state one.

The Proxy fills the gap for callers that omit it, in this order:

1. the caller's own `target_dps[0].port_number`, validated 1-65535;
2. the `X-Data-Return-Port` request header;
3. `6987`, the FHDWP data-plane port.

The SN uses `target_dps[0]` only. The schema models `target` as an array, which reads like fan-out to several destinations; the SN streams to the first entry and ignores the rest.

Configuring it on the query services

`hd_mcp`, `hd_graphql` and `hd_ipflow_graphql` each issue queries and listen for the results themselves, so they advertise their own address:

Variable	Default	Purpose
<code>FERMIHDI_DATA_RETURN_HOST</code>	<i>(discovery, then hostname)</i>	Address advertised to SNs. Set it when discovery cannot pick the right interface.
<code>FERMIHDI_DATA_RETURN_PORT</code>	<code>6987</code>	Port advertised to SNs, and the port the service listens on. Out-of-range or unparseable values fall back to the default silently.
<code>FERMIHDI_DATA_PLANE_SUBNET</code>	<code>10.150.0.0/24</code>	CIDR used to pick the data-plane interface when the host is not set explicitly.

Resolution order for the host is: `FERMIHDI_DATA_RETURN_HOST` if set, otherwise the first local interface address inside `FERMIHDI_DATA_PLANE_SUBNET`, otherwise the container hostname. Discovery exists because DNS is unreliable across multiple container networks — a service name can resolve to the front-end network rather than the data plane — but an explicit host always wins over it.

Version note. Before 2.0.3, `hd_mcp` and `hd_graphql` ran subnet discovery *unconditionally* and overwrote `FERMIHDI_DATA_RETURN_HOST`, so on any host with an address in the data-plane subnet the variable silently did nothing. Only `hd_ipflow_graphql` honoured it. If you are on an earlier build and the setting appears to be ignored, that is why; steer the choice with `FERMIHDI_DATA_PLANE_SUBNET` instead, or upgrade.

7.1.4 Operating Principles

- **Immutable Deployments:** The system is intended to be run completely inside container orchestration systems (Docker Compose or Kubernetes). Do not mutate host networking.
- **Certificate Lifecycle:** every certificate in an HD instance renews itself without a restart; the rules are in *Certificate lifecycle and rules* below.
- **Observability:** All nodes emit OpenTelemetry metrics and traces to a shared OTEL Collector. Node-level health and Prometheus HTTP endpoints are exposed on port 8080 by default. See [observability.md](#) for exactly what each node type emits — it is not uniform across the fleet.

7.1.5 Certificate lifecycle and rules

Every process in an HD instance holds a certificate the DMS-CA signed, and every one of them renews it without a restart. The rules:

- **A node certificate lives 30 days** (`CA_LEAF_VALIDITY_DAYS` on the DMS-CA), never past the ICA's own expiry.
- **A node renews its own certificate inside the last 10 days** (`FERMIHDI_CERT_RENEW_DAYS` on the node): when it starts with an existing identity, and once a day after. It presents its current certificate to the DMS over mTLS and receives a new one for a new key. No token, no administrator.
- **A renewed certificate is taken in place.** The node's command listener swaps its certificate and trust without stopping; its outgoing connections use the new material from the next one opened; the engine's gRPC servers and channels watch the identity files and follow them; the Proxy's transports resolve the certificate at every handshake. Connections already open keep the material they were opened with until they reconnect.
- **An expired certificate cannot renew itself.** A node kept off for longer than a certificate's lifetime registers again, with a token or an approval, exactly as a new node.
- **The DMS's own services** (DMS-Core, the OPA engine, the WebUX) hold a 365-day certificate (`CA_INTERNAL_VALIDITY_DAYS`) and are re-enrolled by the DMS-CA whenever the ICA changes.
- **The ICA renews against the same key.** *Renew or Replace License* on the WebUX submits a CSR for the key the DMS-CA already holds, so every certificate it signed stays valid; the renewed ICA reaches every node in the next configuration push and is pinned in place. A licence that changes the key goes through the same page and replaces the ICA outright, after which every node's certificate is renewed against the new ICA.
- **Trust is pinned to the issuing ICA only.** A node verifies the DMS, and the DMS verifies a node, against the ICA that signed them, never the FermiHDI root on its own; the identity files a node keeps are its certificate, its key and that ICA.
- **The WebUX shows each node's expiry**, amber inside the renewal window and red once passed.

7.1.6 Container Deployment Model

All HD components (Storage Node, Storage Cluster Controller, HD Proxy, Message Bus Ingester, and DMS components) **must always be deployed as containers**, even when running as the only workload on a dedicated bare-metal server. This is a deliberate design principle to align with modern data-centre management practices and orchestration systems (Docker Compose, Kubernetes, Canonical Juju, etc.).

DPDK / SPDK Components: `cpuset` Split Invariant

Components that use kernel-bypass hardware (Storage Nodes and Storage Cluster Controllers with DPDK networking or SPDK storage) have an additional operational invariant that **must be followed on every deployment**:

The container `cpuset` and the `MASTER_CPUSSET` environment variable must reference **disjoint sets of CPUs**. DPDK-pinned cores must never appear in the container `cpuset`.

Why this matters: The container runtime shim (`containerd-shim` on Docker, CRI-O shim on Kubernetes) runs in the host scheduler but inherits the container's `cpuset` cgroup. If a DPDK core appears in the container `cpuset`, the shim will be scheduled on that core, evicting DPDK poll-mode threads. This was measured at over **300,000 shim evictions per 90-second window** in HD profiling — sufficient to cause measurable write-path latency spikes and query P99 regressions. The `isolcpus` kernel parameter alone does not prevent this, because an explicit cgroup `cpuset` assignment overrides `isolcpus`.

Correct split:

Field	Must contain
<code>cpuset</code> (Docker Compose / K8s pod spec)	Overflow / management core(s) only
<code>MASTER_CPUSSET</code> (env var)	All DPDK/SPDK-pinned cores — excluded from <code>cpuset</code>

See `hd_sn/docs/user_guide.md – Step 3a` for full examples, profiling evidence, and Kubernetes guidance.

Deployment Planning: CPU Core Count Requirements

Use the following table when sizing a host for a HD Storage Cluster. All figures are **logical CPU (thread) counts**, based on profiling data from RC12 benchmark runs. These figures assume SMT (Hyper-Threading) is enabled.

Role	DPDK Cores (<code>MASTER_CPUSSET</code>)	Overflow Cores (<code>cpuset</code>)	Isolation requirement
Storage Node (SN)	10 minimum (8 with <code>SHARE_KV_WITH_STORAGE</code>)	2	Overflow must be on a different physical core from DPDK cores
Storage Cluster Controller (SCC)	6 minimum (<code>7 + N_SNs</code> recommended)	1	SMT sibling acceptable on space-constrained hosts (see note below)
Query Proxy (HD Proxy)	1-2	1	No strict isolation — proxy is not DPDK-intensive

The SN figure is a hard floor, not a recommendation. `MASTER_CPUSSET` is divided between five buckets — transport, storage, filter, hash and kv — each with a floor of 2 logical CPUs. Below 10 (or 8 when `SHARE_KV_WITH_STORAGE=true` lets kv alias storage) `FermiController::Init()` **throws and the Storage Node does not start**. Surplus above the floor is weighted, with the filter taking the largest share: at a `MASTER_CPUSSET` of 28 the split is transport 5, storage 5, filter 10, hash 4, kv 4. See `tests/sandbox_dms/sn_resource_recommendations.md`, which is the source of truth for this.

Production sizing formula. Once above the floor, size from the workload rather than the minimum: `6 + (3 × active_drives)` cores per SN and `7 + N_SNs` cores for the SCC. `tests/sandbox_dms/fermi_hdi_sizing.py` computes it. The sandbox's own auto-allocation is deliberately more generous again and is not a production guide.

Minimum host sizes for a single Storage Cluster (1 SCC + 3 SNs):

Scenario	Logical CPUs
Absolute floor, <code>SHARE_KV_WITH_STORAGE=true</code>	36 — OS 2 + SCC 10 + 3 × 8
Absolute floor, kv on its own cores	42 — OS 2 + SCC 10 + 3 × 10
Production, sized from the formula above	46+
Production recommended (per-SN NUMA isolation)	46+ across two NUMA sockets

These are **floors, not targets**: a host sized exactly to one runs every bucket at its 2-core minimum with no surplus to weight. An earlier version of this table gave 16 logical CPUs for 1 SCC + 3 SNs, which cannot start — three SNs alone need 30 at the floor.

Fewer Storage Nodes fit a smaller host. The floor is per-SN, so 1 SCC + 1 SN needs 20 logical CPUs (2 + 8 + 10), and 1 SCC + 2 SNs needs 30. A host that cannot meet the per-service minimum falls back to a shared pool, where every service's `cpuset` is the whole available set and the SN's `auto_distribute` picks its threads from it.

SCC on fully-allocated hosts: When no spare physical core is available for a dedicated SCC overflow (as in the 16-CPU sandbox), the SCC may safely operate with `cpuset == MASTER_CPUSSET`. The SCC reader thread is not a DPDK PMD spin-poller; it blocks between batches. Profiling shows the resulting scheduler events on SCC DPDK cores are `swapper` idle handoffs, not active preemption. This is benign. On hosts with ≥ 18 logical CPUs, allocate a dedicated overflow core to the SCC.

7.1.7 Disaster Recovery & Troubleshooting

Node Quarantine

If an SN begins throwing `Bad Batch Notice (BBN)` Control Messages, quarantine the node via the WebUX. The SCC will dynamically re-route queries to healthy replicas within the Storage Cluster.

A filter node that excludes nothing

Check the SCC's `bloomrf_lookup` log line for the query. `mode: off` means the SCC's data plane started before the filter node was ready or the deployment record names `off`: restart the SCC once the node reports ready. `no_filter` equal to `candidates` means the node holds no verified filter for the window — it started after those batches, evicted them, or its verification stream to the SCC is down (`fermihdi_bloomrf_filters_pending` climbing). A workload whose predicates fall on the batch-id column, use `!=`, or name values present in every batch is not a fault.

Ingest that never starts on a cluster with a filter node provisioned

The DMS holds the SCC's relay list until every provisioned Storage Node and filter node in the cluster reports ready. Start the filter node, or decommission it in the DMS; do not leave a provisioned node unrun.

Re-establishing Data Plane Sync

If an SN falls behind due to network partition on the NORM interface (UDP 6988), reboot the SN. The SCC will trigger a cold-boot sync event over standard TCP fallback on the Control Plane to ensure data integrity before marking the node `Ready`.

7.2 Observability: metrics, logs and traces

This page documents what each node type emits — Prometheus metrics, structured logs, and OpenTelemetry traces — and how the DMS's OTEL Collector routes them to a destination you choose. For *configuring* the collector's exporters, see the [DMS WebUX guide's OTEL Collector section](#).

Every node in an HD instance does two independent things with its telemetry:

1. **Serves its own** `/metrics` and `/healthz//livez//readyz` directly, on its own observability port (`8080` by default, `OBS_PORT` where the service honours the override — see the per-node notes below, since not every service does). A tool that scrapes nodes directly reads this.
2. **Pushes** metrics, logs and traces to the DMS's OTEL Collector over OTLP (gRPC `4317` or HTTP `4318`, the OpenTelemetry standard ports). The collector then re-exports what an administrator has configured — see *The OTEL Collector* below, including its own new pull-based Prometheus endpoint.

These are two views of the *same* instruments on the C++ nodes (SCC, SN, BloomRF) — one render function serves both `/metrics` and the OTLP push, so they cannot disagree. On the Go services, the two paths are typically implemented separately (a `promhttp` handler for `/metrics`, a distinct OTLP trace exporter for traces), so the two can and sometimes do carry different information — noted per service below.

7.2.1 The OTEL Collector

The collector is one process per instance, run under DMS-Core's management (config pushed over OpAMP). It always receives; what it does with what it receives is entirely admin-configured.

Always on, every deployment:

Port	Purpose
<code>4317</code>	OTLP/gRPC receiver — every node's default push target
<code>4318</code>	OTLP/HTTP receiver — the opt-in alternative, and what the WebUX browser client's telemetry relay uses
<code>13133</code>	<code>health_check</code> extension (<code>GET /health</code>)

Admin-configured exporters (DMS → Telemetry page, or the `/v1/otel/config` API), each a distinct destination:

Exporter	Direction	Notes
OTLP/HTTP, OTLP/gRPC	push	any OpenTelemetry-compatible backend
Syslog (RFC 5424)	push	logs only, TCP or UDP, TLS on 6514 by default
Prometheus (remote write)	push	admin supplies the destination's remote-write ingest URL
Prometheus (scrape)	pull	opens a fixed <code>:9464/metrics</code> on the collector itself for an external Prometheus to scrape. At most one may be enabled — see below
Kafka	push	OTLP-proto or OTLP-JSON encoded, per-signal topic

With no exporters configured, the collector still receives everything (every node reports to it regardless) but forwards nothing — a `debug` sink at basic verbosity, so telemetry doesn't back up.

Prometheus (scrape) exporter

Every other exporter is a push: the collector dials out to a URL you supply. The scrape exporter is the one inbound exception, and it works differently because of that:

- **Port is fixed at 9464**, not admin-chosen. A pull target needs the collector's own port opened in its deployment manifest (the Helm chart's container/Service ports, or the bare-Docker installer's `-p` flags) — that happens once, at install time, the same way 4317 / 4318 / 13133 already are. DMS-Core cannot reopen a container port at runtime, so enabling the exporter on the Telemetry page is the only step left to you.
- **At most one enabled at a time** — a second would try to bind the same port and fail the collector's own startup. The config is rejected before it reaches the collector.
- Reachability depends on your deployment path: the Helm chart's `Service` exposes it cluster-wide (a `ServiceMonitor` from your own kube-prometheus-stack can scrape it with no further change); the bare-Docker installer publishes it to the host; the self-contained sandbox does **not** publish it to the host (it follows the sandbox's existing convention of publishing only the front doors and a couple of lab-only tools, nothing else).

7.2.2 Logging format

Every node in an HD instance — Message Bus Ingestor, SCC, SN, BloomRF, the DMS suite, the Query Proxy, the SDK (and therefore any client built on it), `hd_mcp`, `hd_graphql`, `hd_sql`, and Data Explore — emits the same JSON log envelope. This is a real, enforced standard (`.agents/rules/observability-standard.md`), not just a shared convention that happened to converge: two independent components (the process writing a line, and the DMS Collector's exporter reading it back out of the ring on the C++ nodes) depend on it staying exact.

```
{
  "logging_format_version": "2.0.0",
  "timestamp": 1786656385,
  "ts_ms": 1786656385433,
  "ts_us": 1786656385433025,
  "level": "info",
  "service": { "name": "hd_storage_node", "version": "1.0.0", "environment": "production", "node_name": "sn-3" },
  "event": { "action": "sn_rx_first", "category": "ingest_ops", "outcome": "success" },
  "opt_in": false,
  "tracing": { "trace_id": "0af7651916cd43dd8448eb211c80319c", "span_id": "b7ad6b7169203331" },
  "message": "[INGEST] sn received first record of batch",
  "data": { "batch_id": "0000..6a7e4d9c", "records": "60000" }
}
```

Every field is always present except `tracing`, which appears only when the line has real trace context to attach — never fabricated locally, never emitted with empty strings. `opt_in` is likewise always present, always at the top level: never nested under a `fermihdi` key, never omitted when `false`.

Facts worth knowing before treating this as a wire contract:

- **`ts_us` is the field to join on**, not `timestamp` / `ts_ms` — an SN's own ingest window is single-digit milliseconds, which `ts_ms` doesn't resolve.
- **message is never machine-parsed**. It routinely quotes JSON fragments back at the reader; anything a consumer needs lives in `data` or `event`.
- **data.batch_id is always the full 128 hex characters**, never a truncated/local form — two different truncation conventions once made a cross-node join silently return zero rows instead of erroring.
- Only query-path records carry a `tracing` block; ingest records carry none — tracing is control-plane query-path only on the storage cluster, covered per node below.
- On SCC/SN/BloomRF/the Ingestor, routing is: `error`-and-above plus startup lines go to stdout; everything else goes to the in-process ring buffer and out via the collector. `FERMIHDI_LOG_STDOUT=all` puts every level on stdout in addition to the normal export path.
- The exporter counts what it *couldn't* read: `fermihdi.log.malformed` (not valid JSON) and `fermihdi.log.dropped` (lost to ring wraparound before being drained), both labelled by the emitting role — a non-zero value here means an emitter and the schema have diverged.

How tracing gets populated differs by service, and that's expected. SCC/SN/BloomRF/the Ingester and the DMS suite (`dms-core` , `dms-ca` , `dms-opa-engine`) generate real trace context locally — the DMS suite by creating a matching span per log call (the log message becomes the span name, purely to attach `trace_id` / `span_id` ; there's no separate tracing step), the storage cluster by propagating the query's root span. `hd_proxy` does the same from its own genuine per-operation spans (`ServeHTTP` , `PBAC_Evaluation` , `SCC_Fanout` — see its own section below). `hd_sql` , `hd_graphql` , `hd_mcp` , and the SDK carry no OpenTelemetry SDK at all, so none of them generate trace context — a `tracing` block only appears in their logs when a caller explicitly hands one in (e.g. from an inbound request header, or `OTEL_TRACE_ID` / `OTEL_SPAN_ID` for the SDK), never generated locally.

Whether a log line reaches the DMS Collector is a separate question from its shape, and doesn't change with this schema. Two groups:

- **Always deployed, always pushed automatically:** SCC, SN, the optional BloomRF filter node, the Message Bus Ingester, and the DMS suite push structured JSON logs to the DMS's OTEL Collector on their own (see *The OTEL Collector* above) — no configuration needed. This is the population the DMS Collector's log parsing actually depends on.
- **Optional to deploy, local logs only unless you relay them yourself:** `hd_proxy` , `hd_sql` , `hd_graphql` , `hd_mcp` , and any SDK-embedding client write this same JSON to their own stdout only — point your own log collection (a sidecar, a log-shipping agent, your container platform's stdout capture) at each one if you want them in your pipeline. `hd_proxy` is a partial exception: its *logs* stay local like the rest of this group, but its *traces* genuinely do reach the DMS Collector (see its own section below) — logs and traces are separate signals here. Data Explore is a further exception, and different from what you might expect: its server-side logs stay local like the rest of this group (and its server-side errors still go to Sentry only, unchanged), but its *browser-side* usage traces and structured logs reach the DMS Collector through a relay when the deployment has opted in, and its *browser-side* crash reports always reach the deployment's crash vault regardless of opt-in — see its own section below.

7.2.3 Metrics, logs and traces by node

Storage Cluster Controller (SCC)

Metrics (`/metrics` , port `8080` , hardcoded — SCC's HTTP listener does not honour the `OBS_PORT` override):

Metric	Type	What it counts
<code>fermihdi_ingest_system_*</code> , <code>fermihdi_ingest_scc_*</code> , <code>fermihdi_query_scc_fanout_*</code>	count/sum/min/max (4 series each)	Periodic snapshot rollups of the ingest and fanout pipelines
<code>fermihdi_scc_records_processed</code>	counter	Records processed
<code>fermihdi_scc_packets_ingested</code>	counter	Packets ingested
<code>fermihdi_scc_wire_bytes</code>	counter	Bytes received over the wire — the <code>bytes_received</code> mandatory metric
<code>fermihdi_scc_record_bytes</code>	counter	Record payload bytes, wire overhead excluded
<code>fermihdi_scc_fhdwp_overhead_bytes</code>	counter	FHDWP protocol overhead bytes

`queries_processed` , `records_scanned` / `records_selected` and `cpu_utilization` / `disk_utilization` have no SCC-side equivalent — the SCC fans out and hashes; it does not itself scan or select records (that's the SN's job) and does not publish host-resource gauges.

A large additional set of dev-only metrics (queue depths, reactor busy/idle/steal time, NIC hardware counters, latency percentiles, memory-bandwidth/PMU samples) exists under `FERMIHDI_DEV_METRICS` / `FERMIHDI_DEV_METRICS_HOTPATH` compile flags.

Both default OFF in Release/distroless/rock builds — a production SCC image ships none of these; they exist for development-time bottleneck diagnosis only.

Logs: `event.category` values actually used include `ingest_ops` (`scc_rx_first` , `scc_hash_closed` , `scc_rx_last`) and `query_ops` (`scc_q_trace` , carrying a `tracing` block).

Traces: creates the query's **root span** (`scc.query.dispatch`) when a query is dispatched, with attributes for batch-ID count, QoS, and the SN sessions fanned out to. The trace ID is built from the issuing client's `query_id` . The ingest/data path creates no spans by design.

Health: `/healthz` , `/livez` , `/readyz` alongside `/metrics` , same port, served by a minimal raw-socket HTTP server in the OP (supervisor) process — the same process that also holds the metrics registry, so `/metrics` and the OTLP push read one shared set of instruments.

Storage Node (SN)

Metrics (`/metrics` , port `8080` , hardcoded, same caveat as SCC):

Metric	Type	What it counts
<code>fermihdi_ingest_sn_*</code> , <code>fermihdi_query_sn_*</code>	count/sum/min/ max	Periodic ingest/query rollups
<code>fermihdi_query_records_read</code>	counter	Records read during query execution — <code>records_scanned</code>
<code>fermihdi_query_records_selected</code>	counter	Records matching a query's filter — <code>records_selected</code>
<code>fermihdi_sn_records_stored</code>	counter	Records committed to storage — <code>records_stored</code>
<code>fermihdi_sn_bytes_received</code>	counter	<code>bytes_received</code>
<code>fermihdi_sn_bytes_sent</code>	counter	Query result bytes streamed back — <code>bytes_sent</code>
<code>fermihdi_kv_puts / gets / deletes / cleanups / persistence_writes / backpressure_waits</code>	counters	KV-store operation counts
<code>fermihdi_kv_memory_bytes</code>	gauge	<code>memory_usage</code>
<code>fermihdi_kv_disk_bytes</code>	gauge	<code>disk_utilization</code> (byte count, not a percentage)

Of the eight mandatory metrics in the observability rule, the SN is the one node that actually carries all of them (under the names above); `cpu_utilization` only exists as a dev-only PMU sample (`FERMIHDI_DEV_METRICS`), not in a production build. The same large dev-only metric set described for SCC exists here too (queue depths per pipeline stage, NIC counters, storage-device usage, latency percentiles, saturation counts) — same OFF-by-default gating.

Logs: `ingest_ops` (`sn_validate_start` / `sn_validate_done` , carrying `data.batch_id`) and `query_ops` (`sn_q_trace` , carrying a `tracing` block plus `data.query_id` , plus a family of `sn_read` -style actions on the read path).

Traces: continues the SCC's root span as a **child span** (`sn.query.dispatch`) via the inbound W3C `traceparent` , with `query_id` / batch-ID-count/QoS attributes. No spans on the ingest path, consistent with SCC.

Health: same shape as SCC — `/healthz` / `/livez` / `/readyz` / `/metrics` on `8080` from the OP process; unhealthy means a child process died or missed a health-ping deadline (`FERMIHDI_HEALTH_PONG_DEADLINE_MS` , default 15s) — log volume/severity does **not** by itself affect health status.

BloomRF filter node

Metrics (`/metrics` , port `8080` , hardcoded):

Metric	Reality
<code>fermihdi_bloomrf_filters_{total,verified,pending,rejected,evicted,bytes}</code>	Filter-store state: how many filters held, verified against the SCC, awaiting a verdict, hash-mismatched, evicted by retention, and their memory footprint
<code>fermihdi_bloomrf_batches_closed</code> , <code>fermihdi_bloomrf_records_seen</code>	Data-plane intake counts
<code>fermihdi_bloomrf_verify_{verified,mismatch,not_found}</code>	Verdicts from the SCC's batch-verification stream
<code>fermihdi_bloomrf_lookups</code> , <code>_lookup_candidates</code> , <code>_lookup_evaluated</code> , <code>_lookup_excluded</code> , <code>_lookup_excluded_zone_map</code> , <code>_lookup_excluded_bloomrf</code> , <code>_lookup_no_filter</code>	Per-query <code>FilterBatches</code> RPC breakdown: how many candidate batches were asked about, how many a filter existed for, and how many were excluded (split by whether a cheap zone-map check or the BloomRF itself did the excluding)

All of the above are declared `# TYPE ... gauge` in the exporter, even the ones that only ever increase (`lookups` , `batches_closed` , the `verify_*` and `lookup_*` counts). If you graph these, treat them as you would a counter (use them under `rate()` / `increase()` - style reasoning conceptually) rather than expecting a `_total` suffix or a PromQL `counter` type — the Prometheus type metadata itself says `gauge` .

BloomRF has no direct equivalent of `queries_processed` (it pre-filters, it doesn't execute queries), `records_stored` / `records_scanned` (it folds records into filters rather than storing or scanning them — closest analog is `records_seen`), or `bytes_received` / `bytes_sent` / `cpu_utilization` / `disk_utilization` (none tracked; filter memory footprint is the closest via `filters_bytes`).

Logs: rich event set across `startup` (config received, filter server bound, filters reloaded from persistence, DMS registration), `config` (schema applied/changed — a schema-generation change discards existing filters), `dataplane` (channel state), `ingest` (`filter_corrupt` on a bad batch filter), `verify` (verification stream lifecycle and verdicts), and `query_ops` (`bloomrf_filter` , a per-RPC measurement event carrying candidate/evaluated/excluded counts and elapsed microseconds).

Traces: BloomRF sits in the query path between the SCC and its own `FilterBatches` RPC, and receives the SCC's trace context over gRPC metadata, but does not itself call `StartChildSpan` to attach a span object — it carries trace correlation only through the `tracing` block on its `query_ops` log lines, not as a span in the trace view itself.

Persistence: filter reload/discard state (`filters_reloaded` vs `filters_not_persistent`) is logged, not metriced — check `/healthz` or the startup log for filter-store health, not a Prometheus series.

Message Bus Ingestor

Logs reach the DMS Collector automatically, the same as SCC/SN/BloomRF — it uses the same flat `2.0.0` shape and the same always-on logging path (see *Logging format* above).

Metrics: **no** `/metrics` **endpoint by default**. The observability server this service could run is disabled by default. Internally it does keep `records_processed_total` , `packets_sent_total` , and `send_errors_total` as in-memory counters, but these are only ever written into structured log lines (`event.category: "application"`), never exposed for scraping unless an operator changes that default.

Health: a `--healthcheck` CLI flag (not an always-running HTTP listener) connects to `127.0.0.1:<healthcheck_port>` (default `8080`) and expects a JSON `{"status":"ok","timestamp":...}` — this is what a container orchestrator's health probe should invoke, per the CLI-healthcheck convention documented in `.agents/rules/observability-standard.md` .

Traces: none. The ingester is a pure ingest-path component; per the control-plane-only tracing design, it creates and propagates no spans.

Logs: `event.category: "application"` throughout, `actions log`, `telemetry`, `metric` (a periodic uptime line, not a real metric), and `batch` (batch-ID rollover). `config.opt_in` (env `FERMIHDI_OPT_IN` or `--opt-in`) marks records for FermiHDI sharing the same way every other service's top-level `opt_in` field does.

DMS suite

The three main DMS Go binaries — `dms-core`, `dms-ca`, `dms-opa-engine` — share the same observability implementation, so they behave identically: `/healthz`, `/livez`, `/readyz` and `/metrics` on `OBS_PORT` (default `8080`, and this one genuinely honours the env override). Two more DMS binaries do not:

- `dms-opa-adapter` installs only crash reporting — no health or metrics endpoint at all.
- `dms-crash-vault` is entirely outside this framework: plain-text `/healthz` (not JSON) on its intake port (`3001`) and management port (`3002`), and plain `log.Printf` instead of structured JSON logging.

Metrics: `dms-core` (and only `dms-core`) declares six Prometheus series matching the mandatory-metric names — `queries_processed_total`, `records_stored_total`, `records_scanned_total`, `bytes_received_total`, `bytes_sent_total`, `memory_usage_bytes` — but none are incremented or set; they exist on `/metrics` and read zero. DMS sits on the control plane, not the data path, so most of those names don't apply to it in the first place. For DMS-specific activity — node registrations, licence checks/issuance, OTEL config pushes, OpAMP connection count — there is no dedicated Prometheus series: go to the logs (below) for that activity, or the WebUX's Telemetry page, which reads the OpAMP supervisor's live connection count directly rather than through a Prometheus series.

Logs (`logging_format_version: "2.0.0"` — see *Logging format* above): representative `event.category/event.action` pairs actually logged include `security/renew_cert`, `security/crl_check`, `security/revocation`, `lifecycle/node_decommission`, `lifecycle/node_provision`, `storage_cluster/capacity_degraded`, `topology/graceful_offline`, `topology/shutdown_push`, `licensing/apply`, `config/schema_update` (`dms-core`), and `security/sign`, `security/license_apply`, `security/ica_install` (`dms-ca`).

Traces: `dms-core`, `dms-ca` and `dms-opa-engine` all export traces, always-sampled, to the endpoint configured via `OTEL_EXPORTER_OTLP_ENDPOINT`. There are no explicit spans around specific operations (licence issuance, node registration, config push); a span is instead created implicitly, once, per log call — the log message becomes the span name, and its purpose is purely to attach `trace_id/span_id` to that one log record, not to model an operation's duration. `dms-opa-adapter` and `dms-crash-vault` create no traces.

DMS-WebUX (browser-side): opt-in, two-layer gate — the deployment's own opt-in decision (DMS-Core's `/v1/telemetry/optin`, cached 15s) and a per-operator `localStorage` override. When on, `@opentelemetry/sdk-trace-web` instruments page loads and `fetch()` calls and posts spans to a same-origin relay (`/api/telemetry/v1/traces`, 1 MiB/batch cap, a scrubbing step strips user-agent/query-strings/headers first), which forwards them server-side to the deployment's own collector. The browser never talks to the collector or to FermiHDI directly, and the relay enforces the opt-in decision — a client cannot bypass it by editing browser storage.

Query Proxy (`hd_proxy`)

Optional: deploy it when you want a single mTLS/PBAC front door between clients and your Storage Clusters — see the [front door page](#).

Metrics (`/metrics`, port `8080`): a minimal set — `fermihdi_queries_processed_total`, `fermihdi_bytes_received_total`, `fermihdi_bytes_sent_total` (all counters). No records-scanned/selected, no PBAC-decision counter, no resource gauges.

Traces: real, operation-scoped spans, exported over OTLP to the DMS Collector — the inbound request handler (`ServeHTTP`), the OPA policy decision (`PBAC_Evaluation`), and the broadcast to Storage Cluster Controllers (`SCC_Fanout`) each get their own span. This is genuine per-operation instrumentation, unlike the incidental per-log-line span the DMS suite creates.

Logs: `2.0.0`-shaped JSON, stdout only — like `hd_sql`/`hd_graphql`/`hd_mcp`, these do not reach the DMS Collector. Unlike those three, `hd_proxy` now attaches a real `tracing` block on log lines emitted from inside its `ServeHTTP`/`PBAC_Evaluation`/`SCC_Fanout` spans — a genuine improvement: its own log lines can now be correlated to the traces those same spans send to the DMS Collector, even though the log lines themselves stay local.

Query services

These sit behind the Proxy on the client-facing side, each exposing a different wire protocol onto the same query engine. First, the naming: `hd_sql`, `hd_graphql`, `hd_mcp` and `hd_data_explore` are four distinct services, not overlapping ones — an empty `hd_graphql/` subdirectory inside the `hd_sql` repo is a reserved/unused path and unrelated to the top-level `hd_graphql` repo. All four are optional: deploy the ones your clients need.

`hd_sql`, `hd_graphql`, and `hd_mcp` write their logs locally only — none of the three are wired into the DMS Collector (see *Logging format* above). None of them carry an OpenTelemetry SDK, so they never generate real trace spans; the `tracing` block in their logs only ever appears when a caller hands them a trace/span ID it already has.

Service	/metrics?	Mandatory metrics	Logging	Tracing
<code>hd_sql</code>	Yes, port 8080	Full set: <code>hd_sql_queries_processed_total</code> , <code>hd_sql_records_scanned_total</code> , <code>hd_sql_records_selected_total</code> , <code>hd_sql_records_stored_total</code> (defined but never incremented — read-only server), <code>hd_sql_bytes_received_total</code> , <code>hd_sql_bytes_sent_total</code> , plus <code>hd_sql_connections</code> (gauge, by protocol), <code>hd_sql_query_duration_seconds</code> (histogram), <code>hd_sql_pushdown_filters_total</code> , <code>hd_sql_batches_requested_total</code> / <code>_confirmed_total</code>	2.0.0, caller-supplied <code>tracing</code> only	None — no spans generated anywhere in the service
<code>hd_graphql</code>	Yes, port 8083	None defined — only whatever the Prometheus client library's default process collectors provide	2.0.0, caller-supplied <code>tracing</code> only	None
<code>hd_mcp</code>	Yes, port 8080	Same as <code>hd_graphql</code> : process collectors only, no explicit metrics	2.0.0, same shape and behaviour as <code>hd_graphql</code>	None
<code>hd_data_explore</code>	No	N/A — frontend only	Server-side: Sentry only, 2.0.0-shaped local logs are not yet wired here. Browser-side: 2.0.0-shaped structured logs to the console, relayed to the DMS Collector when opted in	Browser-side real OpenTelemetry spans (page loads, <code>fetch()</code> calls), relayed to the DMS Collector when opted in — see below. Server-side: Sentry performance tracing (100% sample rate), a separate pipeline

If you're building a dashboard across "all query services," only `hd_sql` has real per-query metrics; treat the others' `/metrics` output as process-health only (or, for `hd_data_explore`, not present at all).

Data Explore (browser-side): opt-in, following the same shape as DMS-WebUX's telemetry relay but with a simpler gate — `FERMIHDI_OPT_IN` in this service's own server environment, not a DMS-Core API call (Data Explore has no mTLS client identity of

its own to call DMS-Core with). When on, `@opentelemetry/sdk-trace-web` instruments page loads and `fetch()` calls and posts spans to a same-origin relay (`/api/telemetry/v1/traces`, 1 MiB/batch cap, the same scrubbing convention as DMS-WebUX's relay: no user-agent, query strings, or headers), which forwards them server-side to the deployment's own collector — the browser never talks to the collector directly. A `structuredLog()` helper emits the same `2.0.0` JSON envelope to the browser console.

Data Explore crash reporting: `window.onerror / onunhandledrejection` build a Sentry-format envelope (the crash vault's intake wire format, distinct from the `logging_format_version` JSON schema by design — see `hd_dms/docs/crash_vault_openapi.yaml`) and POST it to a second relay (`/api/crash-report`), which forwards it to the deployment's crash vault. Unlike the traces relay, this one is **not** gated by opt-in: the vault's intake always accepts and holds a report; opt-in only controls whether the vault later forwards a stack-only tier to FermiHDI on its own, a decision the vault makes, not this relay. This is genuinely new: no browser- originated crash-reporting path existed anywhere in this product before it.

7.2.4 Mandatory-metric coverage, by service

Before wiring up an alert against the "mandatory metrics" list in `.agents/rules/observability-standard.md`, check whether the service you're targeting actually publishes that metric — coverage varies by design (a control-plane service like DMS has no natural `records_scanned`, for example), not by omission everywhere:

- **queries_processed**: SN (`fermihdi_query_sn_*`), SCC (fanout snapshots), `hd_sql`, and `hd_proxy` publish this. `hd_graphql`, `hd_mcp`, DMS, BloomRF, and the Message Bus Ingestor do not.
- **records_stored / records_scanned / records_selected**: SN publishes all three. `hd_sql` defines all three but only increments two (`records_stored_total` stays zero — the SQL server is read-only). No other service publishes these.
- **bytes_received / bytes_sent**: SCC, SN, `hd_proxy`, and `hd_sql` publish these. The Message Bus Ingestor tracks them in-memory but does not expose them for scraping. BloomRF, DMS, `hd_graphql`, and `hd_mcp` do not publish these.
- **cpu_utilization / memory_usage / disk_utilization**: SN publishes `memory_usage` and `disk_utilization` as byte gauges (not percentages) — `fermihdi_kv_memory_bytes / fermihdi_kv_disk_bytes`. `cpu_utilization` is a dev-build-only PMU sample, compiled out of production builds. DMS defines a `memory_usage_bytes` series that is never set. No service publishes host disk or CPU utilization as a percentage.
- **Traces**: by design, the storage cluster (SCC/SN) only creates spans on the control-plane query path, never on ingest. Among the Go services, only `hd_proxy` creates real operation-scoped spans; DMS's spans are a side effect of its logger rather than independent instrumentation; `hd_sql`, `hd_graphql`, and `hd_mcp` create no spans.

7.3 The DMS WebUX

The WebUX is the administrator's front end to the Data Management System (DMS): the one place an HD instance is defined, licensed, given its dataset schema and its nodes, and watched. Every page is a view onto DMS-Core's management API, so anything done here can also be done by a script against that API — and nothing done here reaches a node until DMS-Core pushes it, which the pages say when they do it.

This is a walk-through in the order a new instance is brought up, followed by the pages an operator returns to. It assumes the DMS plane is running and the WebUX is reachable through its front door; how the plane is deployed and who may reach it are the installer's subject.

7.3.1 Signing in and finding your way

The WebUX holds no accounts and applies no authorization of its own: whoever reaches it is an administrator. What stands between it and the network is the **DMS front door** — a reverse proxy with an authentication step that terminates TLS, authenticates the administrator against your identity provider, and sets `X-Forwarded-User`, `X-Forwarded-Email` and `X-Forwarded-Role` on the requests it forwards, stripping any the browser sent. The reference deployment and the lab use Traefik with a forward-auth service for this; it is yours to provide and may be any ingress that keeps that contract, with or without a forward-auth mechanism. It is separate from the query front door in front of the Proxy, and it is never given a certificate by the DMS-CA. The lab's authenticator answers "admin" for everyone and must not be copied into a deployment. The navigation bar carries seven pages:

Page	Path	What it is for
Dashboard	<code>/</code>	one instance at a time: health, utilisation, query history, the topology
Deployments	<code>/deployments</code>	the instances themselves: create, dataset schema, the query filter mode
Nodes	<code>/nodes</code>	every node of an instance: provision, approve, configure, revoke
Rego Policies	<code>/policies</code>	the access-control policy the Proxy enforces
Telemetry & Logs	<code>/telemetry</code>	opt-in sharing, DMS-Core's live log, exception tracking
OTEL Collector	<code>/otel-collector</code>	where the instance's traces and metrics are exported
License	<code>/license</code>	activation, and what the licence allows

The Nodes page is reached from the navigation bar's Dashboard and Deployments pages ("View Nodes") as well as directly.

7.3.2 First: the licence

DMS-Core admits no node until a licence is installed, so the License page is where a new instance starts. A licence is issued to your organisation, is encrypted to this DMS's identity, and says how many nodes of each type it may admit.

Online activation. Enter the client id and licence id you were given and activate. The DMS fetches the licence, verifies it, and installs it.

Offline activation, for an instance with no route out. *Download Request* produces a request file carrying this DMS's identity; send it to FermiHDI, and *Upload License* installs the licence that comes back.

Either way, applying a licence through this page does more than store it: DMS-Core extracts the DMS-CA's intermediate certificate and its own identity certificate from the licence, which is the Day-0 bootstrap. A licence supplied to DMS-Core through its environment is verified and stored and nothing else.

Once installed, the page shows the licence's client, validity and **utilisation**: for each node type, how many are registered against how many the licence allows, with a type at its limit highlighted. A node type with no limit shows as unlimited.

Renew or Replace License runs the same activation again for a renewal, a change of term, or a change of the authorised node types and counts. It is issued against the DMS-CA's existing key, so every certificate already signed stays valid; applying it installs the renewed ICA on the DMS-CA, re-enrols the DMS's own services, and carries the new ICA to every node in its next configuration push. New counts apply to the next provisioning; running nodes are never evicted, and a type at or over its new limit shows red in the utilisation panel.

The types are the ones the nodes register as: `HD_SCC`, `HD_SN`, `HD_INGEST` and `HD_PROXY`, plus `HD_CLIENT` for an application built on the SDK that registers directly rather than querying through the Proxy (a custom ingester is one), and the `HD_ANY` pool a type draws on when it has no element of its own. `HD_BLOOMRF` is the exception: it is taken not by the filter node, which is always authorised, but by each SCC that puts its filter service into `active` mode, and given back when the SCC leaves it. An SCC refused the slot runs the filter `passive` and says so in its log.

7.3.3 Second: the deployment

A deployment is one HD instance: a name, an instance id, and everything the nodes of that instance share. On the Deployments page, *Register New Deployment* asks for the three; the instance id is what every node is told to register into, so choose it before any node is configured.

Registration fixes two things that can never change afterwards: the hash election (the algorithm and chain block size every SCC, Storage Node and filter node computes over each batch). They lock the moment the first storage cluster is added.

The dataset schema

Each deployment card shows whether it has a dataset schema and, when it does, the record length and the schema generation. **A deployment with no schema is not cosmetic:** DMS-Core falls back to a test layout built into the image, and the whole fleet runs on it. *Dataset Schema* → opens the editor: upload a file, paste the JSON, or paste a schema generator share link -- the link the generator's *Share* button gives carries the document in its `#s=` fragment, which never reaches a server, and *Import* fills the box with it -- then *Upload Schema*.

The schema decides how every node reads a record. Uploading one that keeps the record layout is a routine change: the nodes are updated in place and the card reports how many. Uploading one that **changes the record length** is not, and the page refuses it first, with the numbers: the current and requested record length, and how many SCC and Storage Nodes are running. Every record already stored was written at the old layout and becomes unreadable, so the change requires the acknowledgement offered below the warning, and then a rebuild of the data plane:

- SCC and Storage Nodes: reset their hashes and key-value stores and re-register; they draw new licences and certificates. Storage Nodes also reset their write pointers so old records are overwritten.
- Ingesters, Proxies and SDK clients: restart only.
- A filter node: restart; its store rebuilds from the stream.

The deployment's node registrations are forgotten and its licence utilisation resets to zero, so the rebuilt nodes are not refused as over capacity. The licence, the DMS-CA's certificate and the deployment itself are kept. The schema generation increments, which is how a node that was down during the change learns of it when it returns.

The query filter mode

Each deployment card carries the **Query filter (BloomRF)** setting: how far every SCC of the instance trusts the cluster's filter node. It is set per instance rather than per SCC because passive and active are only meaningful when every controller agrees; the Proxy reports disagreeing controllers as `mixed`.

Setting	The SCC
Default	<code>active</code> wherever a filter node is provisioned; an SCC with no filter node is <code>off</code> whatever it is told
Active	consults the filter node and leaves the excluded batches unread
Passive	consults it, records what it would have excluded, and reads every batch anyway — for measuring a new filter node against real queries
Off	never consults it

Changing the setting pushes the new mode to every SCC of the instance at once and the card says how many were reached. An SCC learns of the filter node itself only when its data plane starts, so the mode takes effect on an SCC that already knows its filter node, and a cluster whose filter node was added after the SCC started needs the SCC restarted first — the Nodes page shows which is the case.

7.3.4 Third: the storage clusters

On the Nodes page, choose the instance in the selector and `+ New Cluster`. A storage cluster is one SCC and the Storage Nodes it fans records out to, and optionally one filter node. The dialog asks for the cluster's name and which Ingest and Proxy nodes should route to it; the hash algorithm shown is the deployment's election. Creating the first cluster locks that election.

Name clusters plainly — `sc-1`, `sc-2` — because node names are derived from them: `sc-1-scc`, `sc-1-sn-1`, `sc-1-bloomrf`. The Nodes page groups nodes under their cluster, and the Dashboard's topology diagram derives the cluster from the SCC's name.

7.3.5 Fourth: the nodes

Provision New Node declares a node before it exists, so that when the process starts and registers, DMS-Core knows what it is, which instance and cluster it belongs to, and how to trust it. The dialog asks for:

- **Node name**, which the process must present as its `FERMIHDI_NODE_NAME`.
- **Node type**. Storage Node, Storage Cluster Controller, BloomRF Filter Node, Ingest Node, Proxy Node.
- **HD instance**, and for an SCC, a Storage Node or a filter node, the **storage cluster** it joins.
- For a Storage Node, optionally an existing Storage Node to **copy the configuration from**: the storage strategy, the devices and the alert thresholds.
- The **registration security mode**, which decides how the node proves itself when it first connects:

Mode	Then
Manual approval	the node registers and waits in <i>Awaiting Approval</i> until an administrator approves it here
Bootstrap token	the dialog shows a one-time token to give the node as <code>FERMIHDI_BOOTSTRAP_TOKEN</code> ; it is admitted the moment it presents the token
Pre-generate credentials	the dialog produces the node's key and certificate now, for an air-gapped host where the process cannot register itself
K8s identity	the node presents its service-account token and is admitted on that

A node is admitted only if the licence has room for its type; otherwise the dialog says which limit was reached.

The filter node

A BloomRF filter node is provisioned exactly like a Storage Node: type *BloomRF Filter Node*, the instance, and the cluster whose SCC it will serve. One per cluster, named after it. The dialog explains the one rule that matters: the node must be ready before the SCC's data plane starts, because the SCC takes its relay list once. DMS-Core holds that list until every provisioned Storage Node *and* filter node of the cluster reports ready — so a provisioned filter node that never starts holds the whole cluster's ingest with it. Provision one only when it is about to run; to add one to a running cluster, provision it, start it, wait for *FullyReady*, then restart the SCC.

Once it runs, the SCC's detail pane shows the **Query Filter** card: the mode in force, the filter node the SCC was given, and its lookup budget. `off (no filter node)` on a cluster that has one means the SCC started first.

Approving, watching and changing a node

The topology pane groups nodes: anything *Awaiting Approval* first, then Ingest nodes, Proxy nodes, each storage cluster with its SCC on top, and the DMS's own services in their panel. A banner counts the approvals waiting. Selecting a node shows its status, type, host, last heartbeat, when its certificate expires, traffic, resource utilisation and — for a Storage Node — its devices. Node certificates are signed for 30 days and every node renews its own within the last ten days, at startup and once a day, taking the new one in place without a restart; the expiry turns amber inside that window and red once passed, when the node has to register again. The pane offers:

- **Approve Registration**, for a node in manual-approval mode.
- **Configure**, which opens the node's settings. What is offered depends on the type: a Storage Node's storage strategy, devices and write path; an SCC's ports, interfaces, record size, batch timeout, fan-out pipeline, cores and filter lookup budget; an Ingest node's source and targets; a Proxy's routing and policy settings; a filter node's ports and log level. A setting the node's own environment has locked is marked and cannot be changed here. Saving pushes the configuration to the node.
- **Graceful Shutdown**, which asks the node to drain and stop.
- **Revoke**, which withdraws the node's certificate. A revoked node cannot reconnect until it is provisioned again.

A whole cluster can be taken **Offline** (maintenance) and brought back with **Resume** from its group header. Ingest and Proxy nodes are told at once and route around it; the SCC, its Storage Nodes and its filter node are marked accordingly.

7.3.6 The Dashboard

The Dashboard shows one instance at a time, chosen in the selector at the top. The summary row is the instance's active and healthy node counts, storage used, average CPU and total queries served. Below it: the instance's details, utilisation, a chart of query history, and the topology — Ingest nodes feeding storage clusters feeding Proxies. Clicking a storage cluster in the diagram drills into it: the SCC, the Storage Nodes it syncs, and the filter node when the cluster has one. *Refresh* re-reads everything.

7.3.7 Rego Policies

The Proxy evaluates every query against a policy written in Rego, compiled and served by the DMS's OPA engine. The page is an editor and a compiler: write or paste the policy, and *Deploy* compiles it and hands the bundle to the engine, from which every Proxy fetches it. A policy that does not compile is refused with the compiler's message and nothing changes.

7.3.8 Telemetry & Logs

Cloud sharing is the opt-in for sending anonymised metrics and exceptions to FermiHDI. It is off unless turned on here, and the decision is held by DMS-Core, not by the browser: the switch shows what DMS-Core holds, and every console that opens this page sees the same answer. The page also shows whether the deployment is registered with FermiHDI, the region it shares into, and the deployment identity it shares under.

Turning the switch on is refused while the DMS was installed with telemetry sharing off (the installer's `fermihdi_dms_enable_analysis` setting, or `DMS_ENABLE_ANALYSIS` on DMS-Core); the page says so and names the setting.

What the switch changes is one thing: what the deployment's OpenTelemetry collector forwards. With it on, DMS-Core has the collector forward the anonymised, shareable telemetry to FermiHDI once the deployment is registered; with it off, the collector forwards nothing. No node or service is told, and none changes what it reports to the deployment's own collector.

The switch also governs this console's own usage telemetry. Off, the WebUX collects nothing. On -- or when the WebUX runs with the `FERMIHDI_OPT_IN` environment flag -- the WebUX server reports its request timings and the browser reports page loads and the console's own API calls to the deployment's OpenTelemetry collector, the same one every node reports to, over whichever protocol the collector is configured for. Nothing that identifies a person is included: no addresses, no user agents, no identity headers, and URLs reduced to their paths.

What is shared, and what never is

Three kinds of data, and each has its own rule.

Tier	What it is	Where it goes	Consent
Never leaves	Schemas, dataset and node names, addresses, user identities, query text, record contents, and anything a console shows.	Nowhere. The collector removes it before anything is forwarded, and FermiHDI's gateway drops any key that is not on the published list below.	None needed: it is not collected.
Operational metadata	The licence ID, product and build commit, node and cluster counts, feature-usage counters, latency histograms, error types, and crash stacks without memory.	FermiHDI's own service in the region the deployment chose (Singapore), on hardware FermiHDI controls.	The switch on this page.
Crash reports	A crash's memory regions, which can hold record bytes, key material or a query in flight.	Held in this deployment's crash vault. Forwarded only per incident, only when an operator presses <i>Share full report</i> .	Per incident, every time.

Only these attribute keys can leave with the operational metadata; anything else is removed by the collector before it is sent:

```
service.name, service.version, service.namespace, deployment.environment, fermihdi.build.commit, fermihdi.product,
fermihdi.license_id, fermihdi.region, host.arch, os.type, fermihdi.opt_in, http.request.method, http.response.status_code,
http.method, http.status_code, http.route, rpc.method, rpc.service, error.type, fermihdi.feature, fermihdi.node_role,
log.level, event.category, event.action, gen_ai.operation.name, gen_ai.tool.name.
```

FermiHDI runs its service on DigitalOcean in Singapore. DigitalOcean is the only infrastructure provider involved and has no access to the data at the product level; no other software-as-a-service provider processes any tier.

Crash reports

Every crash a service in this deployment reports lands in the deployment's crash vault, on the DMS host, and stays there. The **Crash reports** panel lists them: when, which service and version, what happened, the innermost frame, and what has been done with each. Open one to see its stack, symbolicated on the box against the symbol set shipped with the release, and a preview of exactly what each of two sizes would send:

- **Stack only** -- the symbolicated frames, the release and the tags. No memory, no attachments.
- **Full report** -- the report as received, memory regions included.

With cloud sharing on, the stack-only report is sent on its own as soon as a crash arrives, because the same crash on six deployments after a release is something FermiHDI should know without six clicks. The full report goes only when you press *Share full report*, with an optional support ticket to tag it with, and only after a confirmation. *Export to file* writes the whole crash as a tar for a support conversation on an air-gapped deployment. Every share and export is written to an audit log the deployment keeps, with who asked and when, and the panel shows it under the crash.

The vault keeps crashes for 30 days or 2 GB, whichever comes first; the oldest goes when either limit is reached. Delete one sooner from the panel.

The **DMS-Core activity log** streams DMS-Core's own log into the page as it happens: registrations, config pushes, licence decisions, the storage-cluster completeness gate. It can be paused, resumed and cleared. It is the first place to look when a node registers and nothing follows.

7.3.9 OTEL Collector

Every service in the instance sends traces and metrics to the DMS's OpenTelemetry collector; this page decides where the collector sends them on. Add an exporter of one of the supported kinds — OTLP over HTTP or gRPC, syslog, Prometheus remote write, Prometheus scrape, Kafka — with its endpoint and credentials, edit or remove exporters, and *Deploy* to apply the set. Until deployed, changes are marked as pending and the collector runs on the previous set.

Every exporter except one is a push: the collector forwards data out to a destination you name. **Prometheus scrape** is the exception — it turns the collector itself into something an external Prometheus (or any scraper) pulls from, at `<collector-host>:9464/metrics`. That port is fixed rather than chosen here, because — unlike a push destination, which is just a URL the collector dials — a scrape target needs the collector's own inbound port opened in its deployment manifest, which happens once at install time, not from this page. Only one Prometheus scrape exporter can be enabled at a time; a second is rejected before it can reach the collector, since both would try to bind the same port.

For what each node type actually sends the collector — the real metric names, log fields and trace spans, not just the pipeline shape — see [docs/observability.md](#).

A syslog exporter sends RFC 5424 messages over TCP or UDP. Its *Transport* setting chooses syslog over TLS or plain syslog; left at *By port*, it uses TLS on port 6514, the syslog-over-TLS port, and plain syslog on any other port. The *Facility* is applied to every message together with each record's severity.

7.3.10 Where this leaves the nodes

Nothing on these pages configures a node's host. What a node's environment carries is identity — its name, where the DMS is, the token or certificate that proves it — and the DMS supplies the rest at registration and on every push. When something on a node looks wrong, the order to look is: the node's status and last heartbeat on the Nodes page, then the activity log on the Telemetry page, then the node's own log.

7.4 The Front Doors: Reverse Proxies, Authentication and Identity

Nothing in an HD instance authenticates a person. The Proxy, the query services, Data Explore and the DMS WebUX all trust a set of identity headers on the requests that reach them, and something in front of them has to establish who the caller is and set those headers. That something is a reverse proxy with an authentication step, and there are two of them, with different audiences. The lab and the reference deployment use Traefik for both; **nothing in HD depends on Traefik**, and in production both front doors are yours: replace them with the ingress you already run, or configure them, so long as they keep the contract on this page.

7.4.1 Two front doors, two audiences

	The DMS front door	The query front door
Who comes through it	administrators, in a browser	query clients: SDK applications in proxy mode, SQL tools, dashboards, assistants, analysts, and the query services themselves. An SDK client in direct mode, and a custom ingester, do not come through it: they register with the DMS as <code>HD_CLIENT</code> nodes and talk to the storage cluster directly (querying)
What is behind it	the DMS WebUX (port 3000)	the HD Proxy (6986, or 443 as the alternate listener), the SQL server, the GraphQL services, the MCP server, Data Explore, and any GraphQL or Grafana pages you publish
Identity comes from	your administrators' SSO	your users' SSO for people; mTLS client certificates for services and machine clients
Trust relationship	independent of the DMS-CA: it is never given a DMS-issued certificate	presents a client certificate to the Proxy, signed by a CA the Proxy is told to trust
In the lab	<code>dms-traefik</code> on port 8081, one route to the WebUX, a stand-in authenticator that answers "admin" for everyone	<code>traefik</code> on port 80, one path prefix per service, <code>x509-to-http-headers</code> for certificate identities, and a stand-in that injects a fixed identity for the query services

They are separate on purpose. The DMS front door guards the control plane of the instance and sees only administrators; the query front door sees every user of the data. Sharing one proxy between them is a configuration choice you can make, but the two should never share an authentication policy.

7.4.2 The identity contract

Every request a front door forwards carries these headers, set by the front door from the identity it established, and never taken from the client:

Header	Carries	Who requires it
X-Forwarded-User	the caller's unique identifier	the Proxy and Data Explore refuse a request without it (401)
X-Forwarded-Email	the caller's email	policy input
X-Forwarded-Preferred-Username	a display name	policy input
X-Forwarded-Groups	group memberships, comma-separated	policy input
X-Forwarded-Role	the caller's role	policy input
X-Forwarded-For, -Proto, -Host, -Uri, -Method	the original request	logging and audit

What each service does with them:

- **The HD Proxy** feeds the five identity headers, with the query itself, into the policy the DMS distributes, and rewrites the query accordingly. It authenticates nothing itself. It also requires the connection from the front door to be mutual TLS: the front door presents a client certificate, and the Proxy trusts the CA that signed it through `FERMIHDI_NORTHBOUND_CA_PEM`.
- **Data Explore** refuses a request without `X-Forwarded-User` and uses the identity for its own session and audit; the query it then issues goes to the Proxy with that identity.
- **The DMS WebUX** applies no authorization of its own. Whoever the DMS front door lets through is an administrator. Its authentication policy is the whole of the WebUX's access control.
- **The SQL server, the GraphQL services and the MCP server** apply no authorization of their own either, beyond the optional password on the SQL listener and the optional bearer token on the IP-flow service. They call the Proxy as a **service**, through the query front door, with a service identity: the person using them is not propagated to the Proxy, so the policy that governs a query through one of these services is the policy for that service's identity. Where per-user policy matters, the caller uses the SDK or the Proxy directly through the front door.

Strip what the client sends. Because the services trust these headers unconditionally, a front door that forwards a client-supplied `X-Forwarded-User` lets the client choose its identity. Every front door must delete incoming `X-Forwarded-*` identity headers before setting its own.

7.4.3 How the headers get set

Traefik's ForwardAuth is one way to set them: the proxy makes a sub-request to an authenticator, which answers 200 with the identity headers, and the proxy copies them onto the forwarded request. The reference deployment does exactly that, with an OIDC forward-auth service for people and `x509-to-http-headers` for certificate identities, which maps a client certificate's Common Name to the headers through a small mapping file.

None of that is required. NGINX's `auth_request`, Envoy's external authorization, Kong's or an API gateway's plugins, an identity-aware proxy such as oauth2-proxy, or a cloud load balancer with an identity provider all do the same job. What the services need is the outcome: an authenticated caller, an authorization decision, and the headers on the forwarded request.

7.4.4 What you provide, and what you replace

A production deployment is expected to bring its own front doors. Neither the installer's Ansible layer nor the Helm charts deploy one; the lab's are stand-ins and the reference deployment under `docs/examples` is a starting point. For each front door you provide:

1. **TLS** for browsers and clients, from your own CA or a public one.
2. **Authentication** against your identity provider: OIDC or SAML for people, mutual TLS for services and machine clients.
3. **Authorization** of who may reach what: which groups may open the WebUX at all, which may reach the SQL server, which paths are published.
4. **The headers**, set from the established identity and stripped from the incoming request.
5. **For the query front door, a client certificate** presented to the Proxy, whose CA is given to the Proxy as `FERMIHDI_NORTHBOUND_CA_PEM`; and, when the services behind it call the Proxy, a service identity for each.
6. **For the DMS front door, independence from the DMS-CA**. It is the administrator's Traefik, or whatever replaces it, with its own certificates. The DMS never issues it one.

The internal ports the front door forwards to: the WebUX on 3000, the Proxy on 6986 (or its alternate 443), the SQL server's HTTP listener on 8081, the generic GraphQL service on 8083, the IP-flow GraphQL service on 8080, the MCP server on 8082 and its stream on 8083, Data Explore on 4321. The SQL server's PostgreSQL wire (5432) and Arrow Flight SQL (31337) listeners are not HTTP and pass through a TCP-level proxy or are published directly on a network your SQL clients reach.

7.4.5 The lab's stand-ins are not for production

The sandbox exists to exercise the data path, and it fakes the front doors to do it. None of the following may survive into a real deployment:

- `mock-forward-auth`, which answers every request with the same administrator identity, on both front doors.
- `mock-auth-proxy`, which stands in for the query front door in front of the Proxy and injects one fixed service identity for every query service.
- Traefik's dashboard published without authentication, and its `insecureSkipVerify` towards the services.
- Plain HTTP on the entry points.

A deployment that copied the lab's compose file would let anyone who can reach port 8081 administer the DMS. The installer publishes the WebUX on the loopback address by default for this reason, until a front door exists.

7.4.6 Checking a front door

- A request to the Proxy or Data Explore with no identity headers is refused with 401. If it is not, the request did not come through the front door.
- A request that arrives with a client-supplied `X-Forwarded-User` reaches the service with the front door's value, not the client's.
- The Proxy's log shows the identity it evaluated on each query; the WebUX front door's access log shows who opened the console.

7.5 The BloomRF Filter Node

A storage cluster reads every batch a query's time window names, on every Storage Node, whatever the query's predicates say. The BloomRF filter node is the optional role that changes that: it keeps a small filter per batch and, before the Storage Cluster Controller (SCC) fans a query out, answers one question about it — *of these batches, which cannot possibly match?* Those batches are never read. Most deployments run without one, and a cluster with no filter node is the normal case rather than a degraded one.

7.5.1 What it is

A Storage Node that writes an index instead of records. The filter node joins the SCC's ingest broadcast exactly as a Storage Node does, computes the same chain hash over the same records, and verifies that hash with the SCC over the same stream. Where a Storage Node's second path writes the records to a device, this one folds them into a per-batch filter and discards them. Its store lives in memory, is marked usable only once the SCC has confirmed the batch's hash, and is bounded: the oldest batches are evicted first, and an evicted batch simply reads as "no filter".

Two structures are built in the same single pass over each batch:

Structure	Answers	Cost
Zone map — the minimum and maximum of every column	equality and ranges on a column whose values are clustered in time, such as a timestamp or a counter	a few bytes per column per batch
BloomRF — a presence filter per column, shaped by the column's width	equality on any column, and ranges on columns of up to eight bytes; a column of one or two bytes is answered exactly	by default 32 bits per record for an eight-byte column, 8 KiB per batch flat for a two-byte one

The zone map is asked first, because it is a handful of comparisons. Only a sub-query the zone map cannot settle reaches the BloomRF. A filter is built for every column unless the node is told which ones, and the zone map covers every column regardless.

7.5.2 What it never does

- **It never decides what exists.** The SCC resolves batches from its own index, as it always has, and then asks the filter node about that list. The reply is always a subset of what was asked, so the filter can never add a batch, resurrect one, or disagree with the SCC about what is held.
- **It persists its filters, and only its filters.** They are written to the node's local storage as batches close (`FERMIHDI_BLOOMRF_DATA_DIR`, default `/var/lib/fermihdi/bloomrf`; give it a volume) and reloaded at start, so a restart keeps the node's selectivity for the batches it retains. Losing that directory costs selectivity, never answers: a batch it holds no filter for is reported found. There is nothing else to migrate.
- **It is never allowed to be wrong in the harmful direction.** A false positive costs one batch read. A false negative would silently drop data, so every decision falls on the side of "read it": a missing filter, an unverified batch, a batch built under an older schema, a predicate it cannot decode soundly, an unreachable node or a lookup past its time budget — each means the batch is reported found. A Bloom filter has false positives and no false negatives, which is the right way round.

Every client checks. The SDK is the only component that sees both the filter's verdict and the records that arrive, so it compares them on every query: a batch the filter said could not match that nonetheless delivers a record is logged as an error the moment the query completes. A range filter is allowed no false negatives at all — one is a defect to report, not a rate to tolerate.

7.5.3 The three modes

The SCC's trust in the filter is a setting of the whole HD instance, pushed by the DMS to every SCC at once, so a cluster moves between modes in one action:

Mode	The SCC	Use it
<code>off</code>	never consults the filter	to take a filter node out of the query path without stopping it
<code>passive</code>	consults the filter, records what it <i>would</i> exclude, and reads every batch anyway	to measure a new filter node against real queries before trusting it
<code>active</code>	consults the filter and leaves the excluded batches unread	in operation

Once a filter node is provisioned for a cluster, the DMS pushes `active` unless the deployment names a mode; `passive` and `off` are what an operator chooses deliberately, on the Deployments page of the WebUX, which pushes the choice to every SCC of the instance at once. In every mode a lookup that fails or exceeds its time budget (10 ms by default) excludes nothing and the query proceeds unfiltered.

`active` is what the licence meters. When an SCC's mode becomes `active` with a filter node to consult, it takes one `HD_BLOOMRF` slot from the licence (or one from the `HD_ANY` pool when the element is not granted), and gives it back when it leaves `active`. An SCC that is refused a slot runs the filter `passive` instead: the filter is consulted and measured, every batch is still read, and the SCC's log carries `bloomrf_license_refused` with the reason. It asks again every five minutes, so a slot freed elsewhere or a larger licence is picked up without a restart. The filter node itself needs no slot.

The mode in force is visible on every query. The Proxy's manifest carries three batch lists beside the found and missing ones:

Field	Meaning
<code>excluded_batches</code>	batches that exist and were proven unable to match; nothing will be streamed for them
<code>would_exclude_batches</code>	in <code>passive</code> mode, what the filter would have excluded; the batches are read anyway
<code>bloomrf_mode</code>	<code>off</code> , <code>passive</code> , <code>active</code> — or <code>mixed</code> while a mode change is still rolling across the SCCs

A client on the current SDK drops excluded batches from its wait. An older client that does not know the field waits for a batch nothing will stream and hangs to its timeout, which is the one reason to upgrade clients before switching a cluster to `active`.

7.5.4 Provisioning and placement

The filter node is provisioned in the DMS WebUX like a Storage Node — *Provision New Node*, type *BloomRF Filter Node*, the instance, and the storage cluster whose SCC it serves. It is always authorised: the licence is drawn on by the SCC that puts it into `active` mode, not by the node (see the modes above). It registers with its own bootstrap token, like every node, and receives its runtime configuration from the DMS: record size, the hash election, the SCC's address and the dataset schema. A schema change discards every filter and they rebuild from the stream.

Two rules follow from how the SCC learns about it:

- **One per storage cluster.** The DMS hands the filter node's address to that cluster's SCC alone, and the SCC consults the first filter target it is given. Name it after the cluster — `sc-1-bloomrf` serves `sc-1-scc` — which is what the installer's validator checks.
- **It must be up before the SCC's data plane starts.** An SCC takes its relay list once, at that moment. The DMS therefore holds the SCC's targets back until every provisioned Storage Node *and* filter node in the cluster reports ready. Provision only what you will run: a provisioned filter node that never starts holds the whole cluster's ingest with it. To add one to a running cluster, provision it, start it, wait for it to report ready, then restart the SCC; to remove one, decommission it in the DMS and restart the SCC.

The installer knows the role as `bloomrf`: the interview offers it, the validator sizes it, and Ansible and both Helm charts run it. The steps around it — provisioning, the token, the mode — are in the [Administrator's Guide](#), and the WebUX pages that do them in the [WebUX walk-through](#).

7.5.5 Sizing

The filter node is an engine role: two cores (one for the ingest broadcast, one for the control plane), 300×2 MB hugepages for a 512 MB engine arena, no 1 GB pages and no devices. POSIX networking is enough for it; DPDK is optional as on every engine role.

Memory is the real cost, and it follows the data. A filter is built at a fixed size and folded at batch close to $\text{records} \times \text{bits per key} \times \text{levels}$ bits, so it never needs the batch size in advance and the stored size follows the batch. Budget the store as **retained batches $\times$ bytes per batch**, where the node's metrics report the bytes held and its data-plane log records `filter_bytes` per batch. At the defaults an eight-byte column costs 32 bits per record, a two-byte column 8 KiB per batch flat, and a sixteen-byte column 8 bits per record. The store keeps 512 batches by default.

Variable	Default	
<code>FERMIHDI_DMS_URL</code> , <code>FERMIHDI_NODE_NAME</code> , <code>FERMIHDI_BOOTSTRAP_TOKEN</code>	required	as every node
<code>FERMIHDI_PRODUCTION_PORT</code>	6987	the ingest broadcast; the SCC's question is answered on this port plus ten, 6997
<code>FERMIHDI_OBSERVABILITY_PORT</code>	8080	<code>/healthz</code> , <code>/readyz</code> , <code>/metrics</code>
<code>FERMIHDI_SEASTAR_MEMORY</code>	512M	the engine arena
<code>FERMIHDI_BLOOMRF_RETAIN_BATCHES</code>	512	the store's bound; oldest evicted first
<code>FERMIHDI_BLOOMRF_COLUMNS</code>	all	comma-separated column names to build a BloomRF for; the zone map covers every column regardless
<code>FERMIHDI_BLOOMRF_BITS_PER_KEY</code>	8	Bloom bits per record per level, after folding
<code>FERMIHDI_BLOOMRF_LEVELS</code>	4	prefix levels for columns of up to eight bytes; 1 answers equality only
<code>FERMIHDI_BLOOMRF_MAX_PROBES</code>	256	the budget for decomposing a range; a range past it gets "no opinion"
<code>FERMIHDI_BLOOMRF_MAX_KIB_PER_COLUMN</code>	1024	build-time allocation per column, folded down at close

On the SCC, `FERMIHDI_BLOOMRF_MODE` is the bootstrap value the DMS overrides, and `FERMIHDI_BLOOMRF_TIMEOUT_MS` (10) is the lookup budget.

7.5.6 What to watch

The node serves Prometheus metrics on its observability port. The ones that say whether it is doing its job:

Metric	Means
<code>fermihdi_bloomrf_filters_verified</code> / <code>_pending</code> / <code>_rejected</code>	filters the SCC has confirmed, is still to confirm, or refused. Pending that never drains means the verification stream to the SCC is not up
<code>fermihdi_bloomrf_filters_bytes</code>	the store's size — the figure to budget the memory limit from
<code>fermihdi_bloomrf_filters_evicted</code>	batches dropped for space. A steady climb means the retained-batch bound is smaller than the windows queries ask about
<code>fermihdi_bloomrf_lookups</code> , <code>_lookup_candidates</code> , <code>_lookup_evaluated</code> , <code>_lookup_excluded</code>	how often the SCC asks, how many batches it asks about, how many had a usable filter, how many were excluded — selectivity is excluded over candidates
<code>fermihdi_bloomrf_lookup_excluded_zone_map</code> / <code>_excluded_bloomrf</code>	which structure decided each exclusion
<code>fermihdi_bloomrf_lookup_no_filter</code>	candidates with no usable filter — evicted, unverified, or from before the node started
<code>fermihdi_bloomrf_verify_mismatch</code>	a hash the SCC disagreed with. Should stay at zero

The SCC logs one `bloomrf_lookup` line per query with the same figures on its side — `mode`, `candidates`, `evaluated`, `no_filter`, `would_exclude`, `protocol_error`, `lookup_ms` — which is where to look first when a filter seems to do nothing: `no_filter` equal to `candidates` means the node has no verified filters for the window, and `mode: off` means the SCC was never told about the node.

A filter that excludes nothing on a workload is not necessarily broken. Predicates on the batch-id column become the time window rather than a filter, `!=` proves nothing, a range wider than the probe budget gets no opinion, and a value that is genuinely present in every batch is genuinely present. The filter earns its keep on selective predicates — an address, a port, a small value on a wide column — over windows of many batches.

7.6 Sandbox Data Management System (DMS) Operators & Admins Manual

The `./tests/sandbox_dms` directory provides a self-contained, containerized FermiHDI Data Management System (DMS) tailored for local development, testing, and benchmarking. It deploys the entire ecosystem securely, mimicking a production-like Data Plane and Control Plane without requiring external infrastructure.

The sandbox is also the reference deployment the installer is built from and checked against: `compose.yml` is every service with its full environment, and `generate_certs_and_license.sh` is the clearest statement of the PKI an install needs. [A deployment you can read: the sandbox](#) reads the two side by side, including where the sandbox is deliberately not production.

7.6.1 Architecture

The sandbox provisions the following components via Docker Compose: * **DMS-Core**: The central control plane component responsible for orchestrating HD instances. * **DMS-CA**: The internal Certificate Authority responsible for provisioning mutual TLS (mTLS) identities. * **DMS-WebUX**: The user interface for interacting with the cluster. * **HD Proxy**: The high-performance entry point evaluating PBAC policies via OPA. * **Storage Cluster Controllers (SCC)**: Nodes managing horizontal data synchronization across Storage Nodes. * **Storage Nodes (SN)**: The data layer managing NVMe/file-backed Data Plane retrieval.

Networking Rules

The sandbox strictly enforces FermiHDI network segmentation: * **6985**: Internal DMS Plane (DMS-Core <-> DMS-CA, WebUX) * **6986**: Control Plane (HTTPS mTLS) for queries and orchestration. * **6987**: Data Plane (FHDWP) for unidirectional, high-speed query response. * **6988**: Auxiliary Network (NORM Multicast) * **6989**: DNS CA Network * **8080**: Observability (Prometheus Metrics & Health)

All containers are completely isolated on an internal bridge network (`isolated_network`), avoiding host port conflicts unless explicitly mapped.

7.6.2 Operating the Sandbox

Bootstrapping

To start the Sandbox in an automated manner:

```
cd ./tests/sandbox_dms
./start_sandbox.sh
```

The script performs the following actions: 1. Cleans up any existing sandbox state to prevent cross-contamination. 2. Regenerates cryptographic key material (Root CA, Intermediate CA). 3. Evaluates CPU architecture and builds the relevant Docker Compose topology. 4. Executes `docker compose up -d`.

Storage Node count

`--num-sns N` (1, 2 or 3) selects how many Storage Nodes run. When you do **not** pass it, the count recorded in `sandbox_hw_config.json` — what the host was actually provisioned for by `configure_sandbox.py` — is used, and the script warns that it differs from the nominal default of 3. An explicit `--num-sns` always wins, and `--standalone` forces 1.

This matters because three separate consumers must agree on the number: `compose` starts that many SN containers, `configure_sandbox.py` writes matching registration records into `nodes.jsonl`, and `rebalance_sandbox.py` trims the rest. If they disagree, the extra SNs come up with no provisioning record and loop every 30 seconds on:

```
DMS is in Setup Mode (503 Service Unavailable). Awaiting License Context.
```

That is how DMS-Core answers `/v1/register` for a node it has never heard of, and it never resolves. The symptom is easy to miss: those containers still report their control plane healthy, and the script still prints "**Sandbox is fully launched!**" — only the storage table at the end, listing fewer drives than SNs, gives it away.

Version note. Before 2.0.3 the script's own default of 3 silently beat the provisioned value, so a host configured for one SN would start three and wedge two of them. If you are on an earlier build, pass `--num-sns` explicitly to match `sandbox_hw_config.json`, or re-run `configure_sandbox.py`.

Customizing Configuration

Two variables exist for watching a certificate renewal happen in a running lab rather than waiting for one:

`FERMIHDI_CERT_RENEW_DAYS` widens the renewal window (a value larger than the 30-day certificate lifetime renews at the first check) and `FERMIHDI_CERT_RENEW_INTERVAL_SECONDS` shortens the daily check. Export both before the launcher; every node and the Proxy read them, and each renewal is logged with the certificate's new expiry. Neither belongs in a production deployment.

Administrators can modify the deployment via environment overrides. * **CPU Set Pinning:** Edit the `docker-compose.yml` to define explicit CPU affinities for high-throughput testing:

```
cpuset: "0-3" # Pin SCC to first 4 logical cores
```

* **Dataset Schema Injection:** The proxy schema can be pre-loaded by modifying the environment variable `FERMIHDI_DATASET_SCHEMA_FILE`.

7.6.3 Diagnostics & Troubleshooting

• Validating Logs:

```
docker compose -f ./tests/sandbox_dms/docker-compose.yml logs -f dms-core
```

- **Checking Data Plane Connectivity:** Ensure port 6987 is accessible natively if querying from outside the isolated bridge network.
- **Verifying Observability:** Navigate to `http://localhost:8080/metrics` on any given node container.

7.6.4 Shutdown & Cleanup

Always ensure proper teardown to release isolated networks:

```
cd ./tests/sandbox_dms
docker compose down -v --remove-orphans
```

Note: Never issue blanket commands like `docker rm -f $(docker ps -aq)`.

7.7 FermiHDI HD Wire Protocol — Network & MTU Guide

The FermiHDI HD Wire Protocol (FHDWP) is a Layer 5 binary format used exclusively on the **Data Plane (port 6987)** for the unidirectional, high-speed transfer of record batches between HD nodes. This guide covers FHDWP's MTU constraints, the rationale behind the default packet size, and the conditions under which Jumbo Frame mode may be safely enabled.

7.7.1 MTU Constraints

FHDWP enforces a **strict Maximum Transmit Unit** to prevent IP fragmentation across all supported network topologies, including heavily encapsulated environments such as MPLS and Metro Carrier Ethernet.

Mode	MTU (bytes)	Condition
Standard	1200	Default for all HD instances
Jumbo Frame	8500	Admin-elected, instance-wide, network must be verified

The 1200-byte standard MTU is intentionally conservative relative to the Ethernet standard of 1500 bytes. This headroom absorbs encapsulation overhead from:

- **VXLAN** (Docker overlay / Kubernetes Flannel/Calico/Cilium): +50 bytes
- **IPIP** (Calico in IPIP mode): +20 bytes
- **MPLS** labels: +4-12 bytes per label
- **IPsec ESP** (encrypted data planes): +60-70 bytes

Starting from 1200 bytes ensures that even triple-encapsulated paths remain below the Ethernet 1500-byte boundary, eliminating fragmentation entirely.

[!IMPORTANT] The FHDWP MTU is an **instance-wide setting**. All nodes in an HD instance must use the same MTU. Mixing standard and jumbo MTU nodes within a single instance will cause packet loss and data corruption.

7.7.2 Why Larger Packets Are Not the Answer

It is tempting to raise the FHDWP MTU to increase throughput. However, fragmentation has a disproportionate impact on FHDWP:

- **FHDWP payloads consist only of whole records.** A record may not span two packets (except in the defined multipart case). A fragmented packet that loses one IP fragment therefore causes the loss of an entire record — or forces a multipart reassembly path that defeats zero-copy delivery.
- **TCP retransmission at the L4 layer** will eventually recover fragmented packets, but the retransmit timeout adds milliseconds of jitter to what is otherwise a sub-millisecond ingest pipeline.
- **Fragmentation in the network** is silent: routers fragment without notifying the sender, and the FHDWP receiver cannot distinguish a fragmented reassembly from normal delivery.

The performance gain from larger packets (fewer syscalls, less header overhead) does not justify the reliability risk in production environments where every intermediate hop is not under your administrative control.

7.7.3 Jumbo Frame Support: Docker & Kubernetes

Jumbo Frame mode (FHDWP MTU = 8500 bytes) requires an **end-to-end MTU of at least 8550 bytes** (8500 bytes payload + FHDWP headers + IP/TCP headers + any encapsulation). Every hop from sender to receiver must be configured consistently. The table below reflects the real-world support landscape as of 2025.

Environment	Jumbo Frame Support	Notes
Docker bridge (single host)	✅ Configurable	Set "mtu": 9000 in <code>/etc/docker/daemon.json</code> . Host NIC must also be set: <code>ip link set <nic> mtu 9000</code> . Works reliably on bare metal.
Docker overlay (Swarm mode)	⚠️ Possible, complex	VXLAN adds 50 bytes of encapsulation overhead. Physical NICs must be ≥ 9050 bytes MTU. A single misconfigured intermediate switch silently drops oversized frames.
Kubernetes + Flannel (VXLAN)	⚠️ Possible, complex	Same VXLAN 50-byte overhead as Docker overlay. Physical MTU must be ≥ 9050. Flannel does not propagate MTU changes automatically to running pods; restart required.
Kubernetes + Calico (IPIP mode)	✅ Best CNI option	Calico has explicit jumbo frame support. IPIP adds only 20 bytes. Configure <code>mtu</code> in the <code>FelixConfiguration</code> CRD. All nodes must agree.
Kubernetes + Calico (eBPF / no encap)	✅ Excellent	No encapsulation overhead at all if BGP peering is configured. Physical MTU = pod MTU. Highest throughput option.
Kubernetes + Cilium (eBPF)	✅ Good	Cilium honours the kernel NIC MTU and propagates it to pods automatically via CNI config. Jumbo frames work if the physical infrastructure supports them.
Kubernetes + Weave	⚠️ Possible, complex	VXLAN-based with 50-byte encapsulation overhead. Same caveats as Flannel.
AWS (any CNI)	⚠️ Limited	Enhanced Networking (ENA) supports up to 9001-byte MTU, but only between instances in the same placement group on the same host . Cross-AZ traffic is capped at 1500 bytes.
GCP (any CNI)	❌ Not supported	GCP's VPC enforces a 1460-byte MTU ceiling on all VM network interfaces. Jumbo frames are not available.
Azure (any CNI)	❌ Not supported	Azure's virtual network stack caps MTU at 1500 bytes for all VM SKUs. Jumbo frames are not available.
Bare-metal (physical Ethernet)	✅ Full support	Configure switch ports to jumbo mode and set NIC MTU to 9000 on all hosts. No encapsulation overhead. Recommended environment for Jumbo Frame mode.

[!WARNING] Cloud environments (AWS, GCP, Azure) almost never provide reliable end-to-end jumbo frame support. Even where the hypervisor NIC supports larger frames, cloud load balancers, VPN gateways, and inter-AZ routers typically silently re-fragment or drop oversized packets. **Do not enable Jumbo Frame mode on cloud-hosted HD instances without explicit network-level verification.**

7.7.4 Enabling Jumbo Frame Mode

Jumbo Frame mode is an **admin-elected, instance-wide** setting. It must be consistent across all nodes in the HD instance before any FHDWP traffic flows.

Pre-flight Checklist

Before enabling, verify the entire data path:

```
# 1. Verify the host NIC MTU on each SN / SCC host:
ip link show <interface>
# Expected: mtu 9000

# 2. Verify end-to-end path MTU from SN to SCC (no fragmentation):
ping -M do -s 8472 <scc_ip>
# 8472 = 8500 FHDWP MTU - 28 bytes (IP + ICMP headers)
# If this succeeds without fragmentation, the path supports jumbo frames.

# 3. For Docker bridge deployments, verify daemon MTU:
docker network inspect <network_name> | grep -i mtu

# 4. For Kubernetes, verify pod MTU matches node MTU:
kubectl exec -it <sn_pod> -- ip link show eth0
```

Configuration

Set the following in the HD instance configuration before bringing up any nodes:

```
{
  "data_plane": {
    "mtu_mode": "jumbo",
    "mtu_bytes": 8500
  }
}
```

[!CAUTION] Changing the MTU mode on a running instance requires a **full cluster restart**. In-flight FHDWP packets use the old MTU and will be misinterpreted by nodes running the new MTU. There is no rolling upgrade path for MTU changes.

7.7.5 Summary

Decision	Recommendation
Default deployment (cloud, Docker, K8s)	Use standard 1200-byte MTU — no action required
Bare-metal cluster with controlled switch infrastructure	Jumbo Frame mode may be enabled after full pre-flight verification
Mixed environment (some nodes bare-metal, some cloud)	Not supported — instance MTU must be uniform
Performance optimisation via larger MTU	Fragmentation risk outweighs throughput gains; pursue hash-chain or pipeline optimisations instead

See also: [Architecture Overview](#) | [SN Configuration Guide](#)

7.8 Global User Guide

This page is a signpost rather than a guide; the material is kept where it belongs.

To	Read
Query the data — SQL, GraphQL, MCP, the SDK	querying.md , then sql_server.md , graphql_services.md , mcp_server.md
Get data in — sources, file formats, batches	ingestion.md
Build on the SDK	sdk/overview.md
Install and run an instance	install/ADMIN_GUIDE.md
Add the optional filter node	filter_node.md
Drive the DMS: licence, deployments, nodes, policy	webux_guide.md
Put the WebUX and the query services behind your own authentication	front_door.md
Operate one — lifecycles, recovery, day-2 tasks	system_operators_manual.md
Evaluate or benchmark on one host	../tests/sandbox_dms/sandbox_guide.md

Platform reference — networking, storage, PKI, troubleshooting — is in [index.md](#).

8. Development

8.1 HD SDK — Developer Guide

How to write an application that puts records into an HD instance and gets answers back, in any of the eight languages the SDK binds.

This is the guide to `hd_sdk`. It assumes you can build software and have an HD instance to talk to; it assumes nothing about FermiHDI. If you are standing the instance up rather than writing against it, that is [install/ADMIN_GUIDE.md](#).

8.1.1 One engine, eight wrappers

There is one implementation. `libfermihdi_client` is a C++ engine that speaks the FermiHDI wire protocol, holds the mTLS identity, and hands records to your process without copying them. Every language binding is a thin shim onto that engine — they are not eight ports of the same idea, and they do not drift, because there is nothing to drift.

Language	Binding	Package
C++	the engine itself	<code>FermiHDISDK.hpp</code> , <code>libfermihdi_client.so</code>
Python	pybind11	<code>fermihdi_sdk</code>
Go	cgo	<code>fermihdi_sdk</code>
Rust	<code>cxx::bridge</code>	<code>fermihdi-sdk</code>
Node.js	N-API	<code>fermihdi-sdk</code>
R	Rcpp	<code>FermiHDISDK</code>
Julia	CxxWrap	<code>FermiHDI.jl</code>
Scala / Java	JNA	<code>fermihdi-sdk</code>
Mojo	C FFI	<code>FermiHDISDK</code>

What follows describes the engine's model once. The per-language sections then show the same programme in each, and say only what differs.

8.1.2 The model

Two clients, because there are two jobs. `IngestClient` puts records in; `QueryClient` asks questions. They register separately, hold separate identities, and are used from different parts of an application. Nothing stops one process holding both.

Two networks, and the distinction is not cosmetic. The *control plane* carries registration, service discovery and query dispatch. The *data plane* carries records — in on ingest, out on query — over FHDWP, FermiHDI's wire protocol. Both clients take an interface name for each:

```
FermiHDIQueryClient client("https://proxy:6986", "eth1", "eth0");
//                          proxy (control)  data  control
```

Pass an empty data-plane interface and it binds every interface, which is fine on a single-homed host and wrong on any other.

The SDK reads no environment variables — every address, interface and URL is passed in by the caller. That is deliberate: an application's networking should not change because a shell did.

Query results do not come back from the query call. The Proxy answers with a manifest — which batches were found, which were missing, how each Storage Cluster fared — and the Storage Nodes stream the records themselves, directly to the address

your client is listening on. If your client can reach the Proxy but the Storage Nodes cannot reach your client, every query succeeds and returns nothing.

Identity is issued, not configured. `register_with_dms()` generates a key, submits a CSR, and comes back with a certificate signed by the deployment's CA. From then on the client's connections are mutually authenticated. You need a bootstrap token to do it once; you do not need to manage certificates afterwards.

Records are Arrow. Ingest builds `arrow::RecordBatch` and sends it without serialising through an intermediate form; query returns `arrow::Table`, which DuckDB will read in place. That is what "zero-copy" means here — not that no memory is ever touched, but that your records are not translated between representations on the way through.

8.1.3 The two clients

IngestClient

```
register_with_dms(dms_url, node_name, override_key_pem = "") -> token/identity
discover_scc_targets(proxy_url) -> bool
set_scc_targets([SccAddress]) explicit alternative
add_control_header(key, value) routing hints for the Proxy
load_dataset_schema(schema_json) what a record looks like

append_record_to_arrow({field: value}) -> bool
flush_arrow_batch() -> RecordBatch

stream_insert(batch_id, record | records | RecordBatch, broadcast = false)
send_eob(batch_id, broadcast = false) end of batch
send_bbn(batch_id, broadcast = false) batch boundary notice

get_health_status() / get_metrics()
stop()
```

The shape of an ingest loop: register, discover, load the schema, then append records until you have a batch worth sending, flush it to Arrow, `stream_insert` it, and `send_eob` to close it. `broadcast` sends to every SCC rather than the one the batch id routes to.

QueryClient

```
register_with_dms(dms_url, node_name, override_key_pem = "") -> RegistrationResult
set_scc_targets([SccAddress]) direct mode, no Proxy
load_dataset_schema(schema_json)
add_control_header(key, value)

generate_batch_ids_time(start_time, end_time, interval = 0) -> [BatchID]
generate_hd_query(HDQuery) -> query JSON

execute_stream(sql_or_json, callback) per-batch, as they arrive
execute_dataset(sql_or_json) -> arrow::Table
execute_to_sql(fermi_hdi_sql, ...) -> arrow::Table via DuckDB
set_local_aggregations(agg_config) fold as results land

get_found_batch_ids() / get_missing_batch_ids()
had_errors() / get_batch_errors()
stop()
```

Two ways to consume a result, and the choice matters at scale. `execute_dataset` gives you a table when everything has arrived. `execute_stream` calls you per batch as the Storage Nodes deliver, which is what you want when the answer is larger than the memory you would like to spend on it.

Always check `get_missing_batch_ids()`. A query that returns fewer batches than it asked for is a normal, reportable outcome — an SCC timed out, a node was rebalancing — and it is not an error. `had_errors()` and `get_batch_errors()` tell you which, and why.

8.1.4 The same programme, in each language

Register, query, read the answer as a table.

C++

```
#include <FermiHDISDK.hpp>
```

```
FermiHDIQueryClient client("https://proxy:6986", "eth1", "eth0");
client.register_with_dms("https://dms-core:6986", "analytics-1");
client.load_dataset_schema(schema_json);

auto table = client.execute_dataset(query_json);
if (client.had_errors()) { /* get_batch_errors() */ }
```

Link against `libfermihdi_client`. Headers are `include/FermiHDISDK.hpp`; the build wants the prebuilt library tree for your CPU variant (see *Building*).

Python — `fermihdi_sdk`

```
import fermihdi_sdk

client = fermihdi_sdk.QueryClient("https://proxy:6986", "eth1", "eth0")
client.register_with_dms("https://dms-core:6986", "analytics-1")
client.load_dataset_schema(schema_json)

table = client.execute_dataset(query_json) # pyarrow.Table, no copy
df = table.to_pandas()
```

`pybind11`, so the Arrow table crosses as a `pyarrow.Table` sharing the same buffers. `get_shared_ingest_client()` hands back a process-wide ingest client when you would otherwise build one per worker.

Go — `cgo`

```
client := sdk.NewQueryClient("https://proxy:6986", "eth1", "eth0")
defer client.Stop()
client.RegisterWithDMS("https://dms-core:6986", "analytics-1")
table, err := client.ExecuteDataset(queryJSON)
```

`cgo` owns the engine's memory; the wrapper keeps it out of Go's heap so the garbage collector never moves a buffer the engine is writing into. `Stop()` matters here — it releases what the finalizer will not.

Rust — `cxx`

```
let mut client = fermihdi_sdk::QueryClient::new("https://proxy:6986", "eth1", "eth0"?);
client.register_with_dms("https://dms-core:6986", "analytics-1"?);
let table = client.execute_dataset(&query_json)?;
```

`cxx::bridge` gives a checked boundary rather than raw FFI: lifetimes are enforced at compile time, and the engine's errors arrive as `Result`.

Node.js — N-API

```
const { QueryClient } = require('fermihdi-sdk');

const client = new QueryClient('https://proxy:6986', 'eth1', 'eth0');
await client.registerWithDms('https://dms-core:6986', 'analytics-1');
const table = await client.executeDataset(queryJson);
```

V8 buffers map onto the engine's directly. Calls that wait on the network are async and do not occupy the event loop.

R — Rcpp

```
library(FermiHDISDK)

client <- QueryClient("https://proxy:6986", "eth1", "eth0")
register_with_dms(client, "https://dms-core:6986", "analytics-1")
df <- execute_dataset(client, query_json) # via the Arrow C data interface
```

Julia — CxxWrap

```
using FermiHDI

client = QueryClient("https://proxy:6986", "eth1", "eth0")
register_with_dms(client, "https://dms-core:6986", "analytics-1")
table = execute_dataset(client, query_json)
```

Scala / Java — JNA

```
val client = new FermiHDISDK.QueryClient("https://proxy:6986", "eth1", "eth0")
client.registerWithDms("https://dms-core:6986", "analytics-1")
val table = client.executeDataset(queryJson)
```

JNA resolves `libfermihdi_client.so` at run time, so it must be on the library path of the JVM process.

Mojo — C FFI

```
from FermiHDISDK import QueryClient

var client = QueryClient("https://proxy:6986", "eth1", "eth0")
client.register_with_dms("https://dms-core:6986", "analytics-1")
var table = client.execute_dataset(query_json)
```

8.1.5 Beyond fetching rows

The engine carries two analytical libraries so that work can happen where the records already are, rather than after moving them somewhere else.

- **DuckDB.** `execute_to_sql()` runs SQL over the returned Arrow table in process. Joins, aggregates and window functions without a round trip.
 - **USearch.** HNSW vector similarity over the same buffers, for nearest-neighbour work against retrieved records.
 - **Local aggregation.** `set_local_aggregations()` folds results as they land instead of accumulating them, which is what makes a query larger than memory finish.
-

8.1.6 Building

The wrappers all need the C++ engine, and the engine needs its prebuilt library tree for the CPU variant you are targeting — `avx512`, `avx2` or `arm64`:

```
cd hd_sdk
./prebuild_libs.sh          # builds the prebuilt/<variant> tree
cmake -B build -DCMAKE_BUILD_TYPE=Release .
cmake --build build --parallel
```

`release_build.sh` does the same and then packages every wrapper, which is what CI runs. Per-language build notes are in each wrapper's own README — `hd_sdk/python/README.md` and so on — and each has a `tests/` directory worth reading as worked examples.

The variant has to match the host you will run on: an AVX-512 build faults on its first vectorised instruction on a host without it.

8.1.7 When it does not work

Symptom	Usually
Registration fails at the handshake	Wrong DMS URL, or the bootstrap token is spent — they are single-use per node
Registration succeeds, queries return nothing	The Storage Nodes cannot reach your client on the data plane. Check the interface you passed and what your firewall does with FHDWP
Every query returns 503 from the Proxy	The Proxy has not compiled its policy matrix yet. It is a startup race; retry
Queries return 401	Your caller has no <code>X-Forwarded-User</code> — the Proxy authenticates nobody itself, and something in front of it must set the identity headers
Fewer batches than expected, no error	Normal and reportable. <code>get_missing_batch_ids()</code> says which; an SCC timed out or a node was rebalancing
SIGILL at first use	Wrong CPU variant of the prebuilt libraries
Records go in and cannot be found	The dataset schema loaded by the ingest client and the query client disagree

8.1.8 Where else to look

Where	What
Getting started	connecting, wiring up and initialising
API reference	every public type and method
Overview	the SDK's own introduction and quick start
Architecture	how the engine is put together
Configuration	every configuration value
Interface binding	choosing the data and control interfaces
Proxy API	the Proxy API the SDK speaks, as OpenAPI
fhdpw_network_guide.md	the wire protocol and its MTU
../install/ADMIN_GUIDE.md	standing up the instance to write against

8.2 SDK

8.2.1 HD SDK

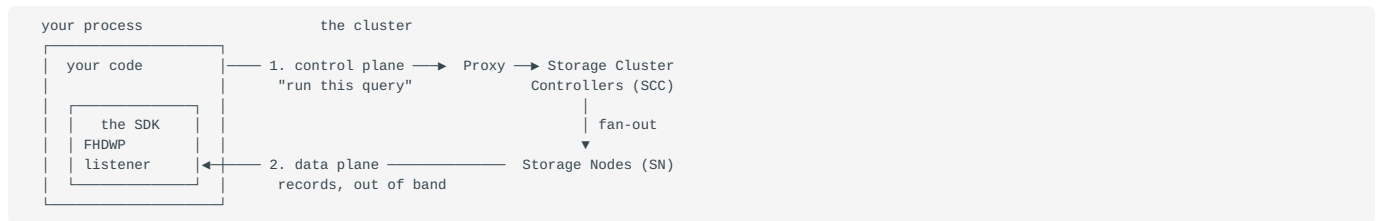
The client library for querying and ingesting against an HD storage cluster. C++ at its core, with bindings for eight languages.

All language wrappers are BETA. They build and link against a real SDK — that is verified on every change by `./build_wrappers.sh` — but they have not been exercised against a live cluster in the way the C++ core and the Go binding have. Expect rough edges in the less-travelled ones, and read [Wrapper maturity](#) before choosing one for production.

The one thing to understand first

The SDK owns the data plane. It does not own the control plane.

A query is two separate conversations with two different pieces of the system, and almost every mistake people make with this SDK comes from thinking it is one.



1. **Control plane.** A query is *submitted* over HTTPS to a Query Proxy, or over gRPC directly to storage cluster controllers. This is a small request that answers one question: *which batches actually exist for what you asked?*
2. **Data plane.** The Storage Nodes then stream the records **straight to your process**, to an address and port you advertised in the query. They do not travel back through the proxy. The SDK's job is to be listening.

Consequences that surprise people:

- **Nothing arrives on the connection you submitted the query on.** If you are waiting for records to come back from the HTTP response, you will wait forever.
- **The address you advertise must be reachable from the storage nodes.** Get it wrong and the query completes with zero records and no error — the nodes streamed perfectly, to nowhere. See [data_plane_advertised_endpoint](#).
- **The listener must be armed before you submit.** Nodes can begin streaming the moment the controllers fan out, which is *before* your submission call returns. Records that arrive at an unarmed listener are discarded. This is what `prepare_query()` exists for, and why `submit_query_to_proxy()` calls it for you.

TWO WAYS IN: THROUGH THE PROXY, OR DIRECT

The control-plane conversation has two forms, and which one you are in decides what the SDK needs before the first query.

Proxy mode is the default. The client submits to the Query Proxy through the customer's query front door; the front door authenticates the caller and the Proxy's policy applies to the query. The SDK discovers the SCC targets and the dataset schema from the Proxy. Nothing registers anywhere.

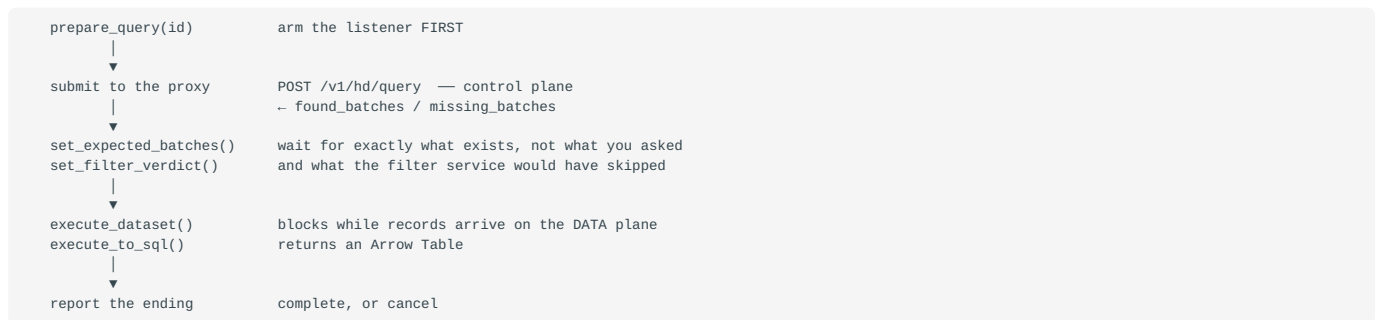
Direct mode is selected by constructing the query client with an empty `proxy_url`. The client then registers itself with the DMS as an `HD_CLIENT` node — `register_with_dms()` sends a certificate signing request and a registration token, and returns the certificate the DMS-CA signed, the private key, the dataset schema and the SCC targets — and submits queries over gRPC to those SCCs itself. An `HD_CLIENT` registration draws on the `HD_CLIENT` element of the licence, or on the `HD_ANY` pool when the licence has none, exactly as a deployed node does, and the DMS admits it under the same policy. The Proxy's policy does not apply: a direct client is inside the trust boundary, which is why it needs a registration token an administrator issued. A custom ingester built on `FermiHDIIngestClient` joins a cluster the same way: it registers as `HD_CLIENT`, receives the SCC targets, and sends its batches straight to them.

Both modes receive records the same way, over the FHDWP data plane on the interface the client names. Direct mode is for services that live inside the deployment, custom ingesters, and the cases where the Proxy's hop is not wanted; proxy mode is for everything a person or an external application does.

Binding	Proxy mode	Direct mode	Ingest
C++	yes	yes	yes
Python	yes	yes	yes
JavaScript	yes	yes	no
Go, Rust, Scala, Julia, R, Mojo (C ABI)	yes	targets only	no

The C ABI carries `fhd_query_client_set_scc_target` but no registration call and no ingest client, so the six bindings built on it can be handed SCC targets but cannot register as `HD_CLIENT` or ingest. Until the C ABI gains both, direct mode and custom ingesters are C++, Python and, for querying, JavaScript.

The lifecycle of a query



THE TABLE IS YOURS

Every execute call ends in an Arrow table, and nothing here hides it. In C++ `execute_dataset()` returns the raw record table (one `FixedSizeBinary` column of the dataset's record width, untouched), `execute_to_sql()` the projected one, `execute_stream()` the record batches as they land, and `apply_sql()` runs SQL over a table you already hold. Across the C ABI, Go and JS the same table travels as an Arrow IPC stream: `execute_dataset_to_ipc` (raw), `execute_to_ipc` (projected), `apply_sql_to_ipc` (yours in, result out). In Python the table is an `ArrowTable` with `to_ipc()`, which `pyarrow.ipc.open_stream()` reads, and `table_from_ipc()` brings one back for `apply_sql()`. Decode it, filter it, or hand it to your own engine: the SDK does not translate anything in it.

WHY `set_expected_batches` MATTERS

Your query names a *window*. The cluster holds whatever it holds. Ask for a day and it may have eleven hours of it.

The control plane answers with `found_batches` — the batches that exist — and `missing_batches`. If you wait for the batches you *asked* for, you wait for ones that were never stored, and the query hangs until its timeout with a partial result.

`set_expected_batches(found)` is what makes a query finish promptly.

ENDINGS, AND WHY THEY ARE NOT OPTIONAL

A query costs storage-node work for as long as it runs. Because the data plane is one-way, **no storage node can tell whether anyone is still listening**. A client that vanishes leaves nodes reading NVMe and streaming into a void.

So a query ends in exactly one of two ways:

ending	meaning
complete	every expected batch arrived. Stop tracking it.
cancel	stopping early, for <i>any</i> reason — an error, a timeout, a user interrupt, a shutdown.

The rule: a client that closes early MUST send a cancel first.

That rule is what lets a clean disconnect be read as "done" rather than as abandonment. It is written up in full in `.agent/rules/query-lifecycle-standard.md`.

THE HELD STREAM

When you submit through a proxy, the SDK **keeps that connection open for the entire life of the query** and sends the ending on it. That connection is the only thing telling the cluster you are still there.

`submit_query_to_proxy()` does all of this — arms the listener, submits, feeds back the batch list, holds the connection, and reports the ending from the same place the completion and cancellation handlers fire. If you use it, endings are handled and you do not need to register a handler at all.

If you submit the query yourself (direct SCC mode, or your own HTTP client), **you own the ending**. Register `set_cancel_handler()` and `set_complete_handler()` and relay them.

Queries

BATCH IDS

A batch is the unit of storage and of fan-out. A batch ID is 64 raw bytes, usually written as 128 hex characters. For time-partitioned datasets the ID encodes the bucket timestamp in its last 8 bytes, which is why the SDK can generate the IDs for a time range without asking anyone:

```
generate_batch_ids_time(from_epoch_sec, to_epoch_sec)
```

THE FULL-DATASET SCAN GUARD

An empty `batch_ids` list is an open query: every batch on every storage node. It is legitimate, and it is the most expensive thing the platform can do. The cost is invisible in the statement that produced it — a `SELECT` with no `WHERE` looks exactly like one with a narrow `WHERE`.

So it cannot be reached by omission:

```
generate_hd_query() refuses a query with no batch_ids
unless acknowledge_full_dataset_scan is set, and returns an empty string.
```

- A caller **with a user interface** should set it from an explicit acknowledgement — a checkbox, a confirmation — and never by default.
- A caller **without one** (a service, a batch job) sets it in code, having decided that a full scan is what it means.

If `generate_hd_query()` hands you an empty string, this is why.

SUB-QUERIES AND FILTERS

Predicates are pushed down to the storage nodes. You express them as filters on named schema fields; the SDK compiles them into the mask/match byte strings the nodes evaluate.

```
Filter { field_name, value, op }      op: EQUALITY | NOT_EQUAL | GREATER | LESS
SubQuery { filters[] }              filters within a sub-query are ANDed
HDQuery { sub_queries[] }           sub-queries are ORed
```

Limit: 16 sub-queries. Beyond that the storage node silently drops the excess. Restructure rather than exceed it.

ADDRESSING

Two different addresses, and confusing them is the single most common cause of "the query returned nothing and said nothing":

	meaning
<code>data_plane_return_port</code>	the port this client binds
<code>data_plane_advertised_endpoint</code>	the address storage nodes should send to

They are the same on a flat network and different behind any NAT — a client in a container binds 6987 inside it while the nodes must be told the host's address and published port. Whatever you advertise must appear in the query's `targets`.

Two clients on one host cannot both bind the default 6987, so the port is a setting, not a constant. It takes effect only before the listener starts.

Health

`get_health_status()` answers two different questions, and they are kept apart deliberately:

```
{"status": "ok", "ready": false, "timestamp": 1788589254.13}
```

field	question	probe
<code>status</code>	is this process alive and answering?	liveness
<code>ready</code>	can it serve a query yet?	readiness

A client that has been constructed but whose engine has not booted — no schema yet, because the proxy was unreachable at init — is "ok" **and not ready**. It will boot on the first query. Restarting it would not have made the proxy reachable, so reporting it as unhealthy would be the wrong answer to the question a liveness probe is asking.

Building against the SDK

GETTING IT

The package image carries everything at `/opt/fermihdi-sdk`:

```
/opt/fermihdi-sdk/
├── include/      headers, including fermihdi_c_api.h
├── lib/          libfermihdi_client.so, libfermihdi_client_static.a
├── cgo-env.sh    the link line for Go, ready to source
├── go/ python/ rust/ js/ julia/ r/ scala/ mojo/
└── PACKAGE.txt  version, variant, provenance
```

```
COPY --from=cr.fermihdi.io/hd/sdk:<version> /opt/fermihdi-sdk /opt/fermihdi-sdk
```

THE LINK LINE — READ THIS BEFORE DEBUGGING A LINK ERROR

`libfermihdi_client.so` embeds the HD engine's runtime through a **static archive**, and a shared object linked against a static archive **does not re-export that archive's transitive dependencies**. Linking `-lfermihdi_client` alone fails with undefined references to `fmt`, `hwloc`, `NORM`, `boost`, `c-ares` and `yaml-cpp`.

Every consumer must name them:

```
-lfermihdi_client -larrow -lnorm -lfmt -lhwloc -lyaml-cpp \
-lboost_program_options -lboost_thread -lcargs -lcurl $(pkg-config --libs libdpdk)
```

For Go this is written for you: `./opt/fermihdi-sdk/cgo-env.sh`.

Use `-Wl,--no-as-needed` ahead of that list if your own code does not reference those symbols directly — with the default `--as-needed` the linker records no `DT_NEEDED` entry, the link succeeds, and the failure moves to runtime as an undefined symbol.

DYNAMICALLY LOADED CONSUMERS NEED STATIC TLS SURPLUS

This affects Python, R and Node — anything the host process `dlopen`s.

The HD engine's runtime uses initial-exec thread-local storage. A library with initial-exec TLS cannot be `dlopen`ed into an already-running process unless the loader reserved surplus static TLS, and you get:

```
cannot allocate memory in static TLS block
```

Set the tunable before starting the host process:

```
GLIBC_TUNABLES=glibc.rtld.optional_static_tls=1048576 R -f script.R
GLIBC_TUNABLES=glibc.rtld.optional_static_tls=1048576 python3 app.py
```

Compiled consumers that link at build time (Go, Rust, Mojo, C++) are unaffected.

STATIC LINKING

`libfermihdi_client_static.a` is built beside the shared library (`-DFERMIHDI_BUILD_STATIC=ON` , the default). Link it instead of the `.so` when you want a binary you can copy to a host on its own, with no `rpath` to get right.

The C ABI

`include/fermihdi_c_api.h` exports a flat C surface from the shared library, so any language with FFI can use the SDK without a C++ toolchain. It is what the Mojo binding is built on, and it is the recommended base for a new binding.

```
FhdQueryClient* fhd_query_client_new(proxy_url, dp_iface, cp_iface, return_port);
void            fhd_query_client_free(client);
bool           fhd_query_client_load_schema(client, schema_json);
char*          fhd_query_client_generate_query(client, hdquery_json);
char*          fhd_query_client_submit_to_proxy(client, query_json, query_id);
const uint8_t* fhd_query_client_execute_to_ipc(client, query_json, sql, &size);
void           fhd_query_client_cancel_query(client, query_id);
```

Conventions:

- **Nothrow.** A C++ exception crossing an FFI boundary is undefined behaviour, so every entry point catches and reports through its return value.
- **`char*` returns are caller-owned** — release with `fhd_string_free`; byte buffers with `fhd_bytes_free`.
- **`*_into` variants** fill a buffer you own and return the length needed, which avoids ownership transfer entirely. Prefer these in languages where a nullable pointer is awkward to model.
- **A NULL client is tolerated everywhere**, so code that failed to construct one cannot crash by carrying on.

Wrapper maturity

All wrappers are **beta**. Every one is built and linked against a real SDK by `./build_wrappers.sh`, which runs in CI, so none of them is broken. What separates them is how much real use they have had.

language	status	notes
C++	stable	the SDK itself
Go	beta — most exercised	used by four production services
Python	beta	pybind11; needs the TLS tunable
Rust	beta	cxx bridge; narrow surface
JavaScript	beta	N-API; needs the TLS tunable
Julia	beta	ccall; narrow surface
R	beta	Rcpp; needs the TLS tunable
Scala	beta	JNA; JDK 17 only
Mojo	beta	on the C ABI; explicit <code>close()</code>

Beta means: it builds, it links, its API is documented and exercised by a smoke build. It does not mean every call has been run against a live cluster. If you are choosing for production today, Go has by far the most mileage.

Verifying a change

Building the core and every wrapper, and the tests that guard them, are the [developer guide](#)'s subject. A consumer of the published package never builds any of it.

Where to go next

- **Language guides** — the per-language READMEs linked in §7, each with a full API reference, a worked example and use cases.
- **Query lifecycle rules** — `.agent/rules/query-lifecycle-standard.md`.
- **The C ABI** — `include/fermihdi_c_api.h`.
- **The C++ API** — `include/FermiHDISDK.hpp`, documented inline.

8.2.2 HD SDK — Connecting, Wiring Up and Initialising

The HD SDK is a **client library**. You link it into your process; it does not run as a service, does not listen for HTTP, and exposes no REST API of its own. It gives you two classes:

Class	Direction	What it does
<code>fermihdi::sdk::FermiHDIIngestClient</code>	write	Encodes records and streams them to Storage Cluster Controllers over FHDWP.
<code>fermihdi::sdk::FermiHDIQueryClient</code>	read	Dispatches queries and receives result batches back over FHDWP.

Everything the SDK talks to is documented elsewhere, because it belongs to the service that serves it: registration in the DMS southbound API, client topology and dataset schema in the Proxy API, and the wire protocol in the FHDWP network guide.

Build and link

One public header, one shared library target:

```
#include <FermiHDISDK.hpp>

using namespace fermihdi::sdk;
```

```
target_link_libraries(your_app PRIVATE fermihdi_client)
```

`fermihdi_client` pulls in Apache Arrow, so anything appearing in the SDK's signatures as `arrow::RecordBatch` or `arrow::Table` is a normal Arrow object in your process — no copying or conversion at the boundary.

What construction actually does

Both clients boot the HD engine **in their constructor**. There is no separate `init()` or `start()` call; by the time the constructor returns, the engine thread is running and the client is ready to use.

Constructing a client will:

1. Probe for a GPU runtime and log which one it found, falling back to CPU routines when there is none.
2. Start the HD engine on its own thread, reserving **one core and 1 GB of memory** for it.
3. Bind the data plane to the interface you named.
4. **Block until the engine reports ready**. Expect the constructor to take a moment, and do not construct one per record — construct once and keep it.

Because a client owns a core and a gigabyte, treat it as a long-lived object. One per process is the normal shape.

INTERFACES ARE YOURS TO SUPPLY

```
FermiHDIIngestClient ingest("eth1", "eth0");
//                      ^data  ^control

FermiHDIQueryClient  query("https://proxy.internal:8443", "eth1", "eth0");
//                      ^proxy URL    ^data  ^control
```

The SDK does not read environment variables to discover its interfaces. If you pass an empty string for the data plane interface it binds to all interfaces (0.0.0.0) rather than consulting the environment. On a host with separate data and control networks, always name them explicitly.

Two environment variables *are* read, and only to select the network stack:

Variable	Effect
<code>FERMIHDI_NETWORK_STACK</code>	Names the stack the engine should use. Defaults to <code>posix</code> .
<code>FERMIHDI_FORCE_POSIX</code>	<code>1</code> or <code>true</code> selects the POSIX stack.

Passing an empty `proxy_url` to `FermiHDIQueryClient` selects **direct SCC mode**: the client registers with the DMS as an `HD_CLIENT` node through `register_with_dms()`, which returns its certificate, the schema and the SCC targets, and it submits to those SCCs itself instead of to a Proxy. The [SDK overview](#) says when to use which; direct mode needs a registration token from an administrator and an `HD_CLIENT` licence element (or the `HD_ANY` pool).

What has to be reachable

Direction	Peer	Purpose
Outbound HTTPS	DMS	<code>POST /v1/register</code> — certificates and topology
Outbound HTTPS	Proxy	<code>GET /v1/client/info</code> — SCC target discovery
Outbound HTTPS	Proxy	<code>GET /v1/hd/schema</code> — dataset schema
Outbound HTTPS	SCC	<code>POST /v1/query/stop</code> — query cancellation (retired; answers 410)
Outbound FHDWP	SCC targets	record and control submission
Inbound FHDWP, port 6987	Storage Nodes	query results return here

That last row is the one that catches people. `FermiHDIQueryClient` opens a listener on **port 6987** to receive result batches directly from Storage Nodes. If your firewall or security group blocks inbound 6987, queries will dispatch successfully and then hang until they time out, because the results have nowhere to land. Port 6987 is the FHDWP data plane port described in the FHDWP network guide.

Control-plane calls use mutual TLS with the certificate obtained during registration.

Wiring up the ingest path

The order matters. Each step depends on the one before it.

```
#include <FermiHDISDK.hpp>
using namespace fermihdi::sdk;

// 1. Construct. The engine boots here.
FermiHDIIngestClient client("eth1", "eth0");

// 2. Register with the DMS. Returns the client key in PEM form and stores
//     the signed certificate for later control-plane calls.
std::string key_pem = client.register_with_dms(
    "https://dms.internal:8443", "edge-collector-01");

// 3. Find the Storage Cluster Controllers, either by asking the proxy...
if (!client.discover_scc_targets("https://proxy.internal:8443")) {
    // ...or by naming them directly.
    client.set_scc_targets({{"10.0.4.11", 6987}, {"10.0.4.12", 6987}});
}

// 4. Load the dataset schema. Without it, record encoding produces nothing.
client.load_dataset_schema(schema_json);

// 5. Choose a batch ID. Everything in one batch shares it.
BatchID batch = client.generate_batch_id_time(start_epoch, end_epoch);

// 6. Encode and send.
for (const auto& row : rows) {
    const std::vector<uint8_t>& record = client.convert_record_to_binary(row);
    if (record.empty()) continue; // no schema, or nothing matched
    client.stream_insert(batch, record);
}
```

```
// 7. Close the batch. Until the EOB lands, the storage layer holds it open.
client.send_eob(batch);
```

Steps 3 and 4 are both mandatory. With no SCC targets set, `stream_insert()`, `send_eob()` and `send_bbn()` all return `false` without sending anything. With no schema loaded, `convert_record_to_binary()` returns an empty buffer.

If you discover mid-batch that the data was bad, send a Bad Batch Notice instead of an EOB:

```
client.send_bbn(batch); // tells the storage layer to discard the batch
```

Wiring up the query path

```
FermiHDIQueryClient client("https://proxy.internal:8443", "eth1", "eth0");

// Registration is only needed when querying SCCs directly (no proxy).
// It returns the full topology and the mTLS material in one object.
RegistrationResult reg = client.register_with_dms(
    "https://dms.internal:8443", "analytics-01");
if (reg.success) {
    client.set_scc_targets(reg.scc_targets);
    client.load_dataset_schema(reg.schema_json);
}

// Guard against a query that never completes.
client.set_query_timeout_ms(30'000);

std::shared_ptr<arrow::Table> table = client.execute_dataset(query_json);
// execute_stream() is available too, but note it completes the whole query
// before invoking your callback – it is not an incremental stream.

// A query can succeed and still be incomplete. Always check.
if (client.had_errors()) {
    for (const auto& err : client.get_batch_errors()) {
        // this batch was not delivered; the table is missing its records
    }
}
```

TWO THINGS WORTH GETTING RIGHT

Set a timeout. `set_query_timeout_ms(0)` is the default and means *no timeout*: if a Storage Node never returns its end-of-batch marker, the call blocks forever. When the timeout does fire, the SDK stops waiting and returns an **empty table** — not an error, and not a partial result.

It also sends `POST /v1/query/stop` to every known SCC target, but **the SCC retired that route on 2026-08-09 and answers 410**, so nothing downstream hears about it. The timeout and `cancel_query()` free your thread; the Storage Nodes carry on reading until the query finishes by itself. Size timeouts with that in mind — an aggressive one abandons work that still runs.

A successful query can be missing data. If a Storage Node sends a Bad Batch Notice instead of the records, the query completes normally and the Arrow Table silently omits those rows. `had_errors()` is the only thing that tells you, and `get_batch_errors()` says which batches went missing — one entry per failed batch, not one per notice received.

REMOVING THE DISPATCH RACE

If you dispatch the query yourself and only then hand the SDK the expected batch list, end-of-batch markers can arrive *before* the SDK knows to expect them. Pre-arm the listener to avoid it:

```
client.prepare_query(query_id); // BEFORE dispatching to the proxy
// ... dispatch, wait for the proxy to return found_batches ...
client.set_expected_batches(found_batches); // AFTER
client.set_filter_verdict(would_exclude_batches, bloomrf_mode); // from the same event
auto table = client.execute_dataset(query_json);
```

`set_filter_verdict()` matters on a cluster with a filter service: the SDK checks, when the query completes, that no batch the filter said could not match delivered a record. It only knows that list when it placed the query itself, so a self-dispatching caller passes it on. Skip it and the check silently never runs.

`prepare_query()` resets the query state and arms the listener, so markers arriving during the wait are kept instead of discarded. `set_expected_batches()` then completes the query immediately if everything already arrived. Skipping both still works — the SDK falls back to counting batches out of the query payload — but it reintroduces the race.

To stop an in-flight query from another thread:

```
client.cancel_query(query_id); // safe to call from any thread
```

Health and metrics are strings, not endpoints

```
std::string health = client.get_health_status(); // JSON
std::string metrics = client.get_metrics();      // Prometheus text format
```

The SDK **does not serve these**. It hands you the strings and your process decides what to do with them — typically publishing them on whatever health and metrics endpoints your application already exposes. If you want `/healthz` and `/metrics` for a process embedding the SDK, you serve them; the SDK has no listener to attach them to.

Shutting down

```
client.stop(); // stops the engine thread
```

The destructor also stops the client, so an object going out of scope is a clean shutdown. Call `stop()` explicitly when you need the core and the memory released at a point you control.

Send an EOB for any batch still open before stopping. A batch left open is a batch the storage layer is still waiting on.

When something returns false or null

Symptom	Cause
<code>stream_insert</code> / <code>send_eob</code> / <code>send_bbn</code> return false	No SCC targets. Call <code>discover_scc_targets()</code> or <code>set_scc_targets()</code> .
<code>stream_insert</code> returns false with targets set	Record was empty.
<code>convert_record_to_binary</code> returns an empty buffer	No schema loaded, or the schema declares no record length.
<code>discover_scc_targets</code> returns false	Proxy unreachable, response unparseable, or it named no targets.
<code>execute_dataset</code> returns nullptr	The schema could not be loaded, so the engine never booted.
<code>execute_dataset</code> returns an empty table	The query timeout fired. Results were abandoned and a stop was sent.
Query dispatches then hangs	Inbound port 6987 is blocked, so results cannot return.
Records silently missing from a result	Batches were BBN'd. Check <code>had_errors()</code> .
Fields silently missing from a record	The key is not in the schema, or its byte range is out of bounds. Both are skipped without error.

Methods that are not implemented yet

Four public methods currently do nothing. They are declared in the header and they compile, but there is no behaviour behind them:

Method	Actual behaviour
<code>FermiHDIIngestClient::append_record_to_arrow()</code>	Logs and returns <code>true</code> . Nothing is appended.
<code>FermiHDIIngestClient::flush_arrow_batch()</code>	Always returns <code>nullptr</code> .
<code>FermiHDIQueryClient::set_local_aggregations()</code>	Logs the configuration and ignores it.
<code>FermiHDIQueryClient::execute_graph()</code>	Runs the query, then logs which graph path it would take. Produces no result and has no way to return one.

Do not build on these. To send Arrow data today, use the `stream_insert(BatchID, std::shared_ptr<arrow::RecordBatch>, bool)` overload, which is implemented, and construct the `RecordBatch` with the Arrow API directly.

8.2.3 HD SDK — API Reference

Every public type and method in `<FermiHDSDK.hpp>`, in namespace `fermihdi::sdk`. For how to put them together, see the getting started guide.

Both clients use the pimpl idiom, are non-copyable, and boot the HD engine in their constructor. See [Not implemented](#) for the four methods that currently do nothing.

Types

BatchID

```
using BatchID = std::array<uint8_t, 64>;
```

A fixed 64-byte batch identifier. Every record, end-of-batch marker and bad batch notice carries one. Build them with the `generate_batch_id_*` methods rather than by hand — the byte layout is part of the wire contract.

FilterOp

```
enum class FilterOp { EQUALITY, NOT_EQUALITY, GREATER, LESS };
```

Filter

```
struct Filter {
    std::string field_name;
    std::string value;
    FilterOp    op;
};
```

One predicate against one field. `value` is a string regardless of the field's declared type; it is interpreted according to the schema.

SubQuery

```
struct SubQuery { std::vector<Filter> filters; };
```

A group of filters evaluated together.

SccAddress

```
struct SccAddress { std::string ip; uint16_t port; };
```

One Storage Cluster Controller on the data plane.

HDQuery

```
struct HDQuery {
    uint32_t          query_id;
    std::vector<BatchID> batch_ids;
    std::vector<SubQuery> sub_queries;
    std::vector<SccAddress> targets;
    bool              verify_hash = false;
    unsigned int      query_qos   = 1;
};
```

Field	Meaning
<code>verify_hash</code>	When true, Storage Nodes recompute the block-chain hash over every record and verify it against the closing hash stored at ingest. Costs throughput; use it for audit and compliance queries, not routine ones.
<code>query_qos</code>	Priority weight, 1–255, feeding read-queue weighting and admission ordering on the Storage Node. 1 is lowest. Values outside the range are clamped on serialisation; absent or invalid is treated as 1.

BatchError

```
struct BatchError { BatchID batch_id; std::string error; };
```

One batch that failed delivery during a query — typically because the Storage Node sent a bad batch notice instead of the data. See `had_errors()`.

RegistrationResult

```
struct RegistrationResult {
    bool success;
    std::string client_key_pem;
    std::string signed_cert_pem;
    std::string schema_json;
    std::vector<SccAddress> scc_targets;
    std::string error_message;
};
```

Returned by `FermiHDIQueryClient::register_with_dms()`. One round trip yields the mTLS material, the dataset schema and the cluster topology, so a registered query client can be fully configured without further discovery calls.

FermiHDIIngestClient

The write path: encode records, stream them to SCCs, close the batch.

CONSTRUCTION AND TEARDOWN

```
explicit FermiHDIIngestClient(const std::string& data_plane_interface = "",
                             const std::string& control_plane_interface = "");
~FermiHDIIngestClient();
void stop();
```

Boots the HD engine on its own thread, reserving one core and 1 GB, and blocks until it reports ready. Name your interfaces — an empty data plane interface binds all interfaces rather than reading the environment. Construct once and keep it; this is not a per-record object.

`stop()` stops the engine thread. The destructor does the same, so scope exit is a clean shutdown.

register_with_dms

```
std::string register_with_dms(const std::string& dms_url,
                             const std::string& node_name,
                             const std::string& override_key_pem = "");
```

Registers against the DMS, storing the signed certificate and client key for subsequent control-plane calls. **Returns the client key in PEM form**; an empty string means registration failed. Pass `override_key_pem` to register with a key you already hold instead of a freshly generated one.

discover_scc_targets

```
bool discover_scc_targets(const std::string& proxy_url);
```

Asks the proxy for the cluster topology and populates the SCC target list from the primary cluster it reports. Returns `false` if the proxy is unreachable, the response cannot be parsed, or it names no targets — check the return value, as a failure leaves you with no targets and every send will then fail.

add_control_header

```
void add_control_header(const std::string& key, const std::string& value);
```

Adds an HTTP header to every subsequent control-plane request, for proxy routing or tenancy. Headers accumulate; there is no method to clear them.

set_scc_targets

```
void set_scc_targets(const std::vector<SccAddress>& targets);
```

Sets the targets directly, replacing any discovered set. **The first target is the primary** and receives everything sent without `broadcast`, so ordering is significant.

load_dataset_schema

```
void load_dataset_schema(const std::string& schema_json);
```

Loads the schema describing the fixed-width record layout — field names, byte ranges, types, and the time index interval used by `generate_batch_id_time()`. Required before any record encoding.

convert_record_to_binary

```
const std::vector<uint8_t>& convert_record_to_binary(
    const std::map<std::string, std::string>& kv_pairs);
```

Encodes a key/value map into one fixed-width binary record, writing each value into the byte range the schema assigns its field.

Two things to know:

- **The returned reference points at a thread-local buffer reused by the next call on the same thread.** Send it or copy it before calling again. Never hold it across iterations.
- **Unmatched input is dropped silently.** A key absent from the schema, or a field whose byte range falls outside the record, is skipped without error. Returns an empty buffer when no schema is loaded.

append_record_to_arrow / flush_arrow_batch

```
bool append_record_to_arrow(const std::map<std::string, std::string>& kv_pairs);
std::shared_ptr<arrow::RecordBatch> flush_arrow_batch();
```

Not implemented. See [Not implemented](#).

BATCH ID GENERATION

```
BatchID generate_batch_id_string(const std::string& str);
BatchID generate_batch_id_number(uint64_t number);
BatchID generate_batch_id_time(int64_t start_time, int64_t end_time,
    int interval_override = 0);
```

All three produce a 64-byte ID, zero-filled and right-aligned so IDs sort and compare consistently.

Method	Layout
<code>generate_batch_id_string</code>	Up to 64 bytes of the string, right-aligned with zero padding on the left. Longer strings keep their first 64 bytes.
<code>generate_batch_id_number</code>	The value big-endian in the final 8 bytes. Deterministic — the same number always yields the same ID.
<code>generate_batch_id_time</code>	A time window: the start floored to the interval in bytes 48-55, the end rounded up to the next interval boundary in bytes 56-63.

`generate_batch_id_time` takes its interval from the schema's time index interval unless you pass `interval_override`; an interval of zero is treated as one second. Because the start floors and the end ceilings, the window covers at least the range you asked for and aligns to index boundaries.

SENDING

```
bool stream_insert(const BatchID&, const std::vector<uint8_t>& record,
    bool broadcast = false);
bool stream_insert(const BatchID&, const std::vector<std::vector<uint8_t>>& records,
    bool broadcast = false);
bool stream_insert(const BatchID&, std::shared_ptr<arrow::RecordBatch> batch,
    bool broadcast = false);
bool send_eob(const BatchID& batch_id, bool broadcast = false);
bool send_bbn(const BatchID& batch_id, bool broadcast = false);
```

`stream_insert` submits records to the data plane under a batch ID. The three overloads take a single encoded record, a vector of them, or an Arrow `RecordBatch` — the Arrow overload is the implemented path for columnar data.

`send_eob` closes a batch. Until the end-of-batch marker arrives, the storage layer treats the batch as still open. `send_bbn` sends a Bad Batch Notice instead, telling the storage layer the batch is corrupt or incomplete and should be discarded.

All five return `false` immediately if no SCC targets are set, and `stream_insert` also returns `false` for an empty record.

`broadcast` **changes the destination**. Left `false`, everything goes to the first target only. Set `true`, it goes to every target.

Metrics caveat: `records_ingested` and `bytes_ingested` are incremented only on the non-broadcast path. Records sent with `broadcast = true` are delivered but not counted, so `get_metrics()` will under-report if you broadcast.

OBSERVABILITY

```
std::string get_health_status() const; // JSON
std::string get_metrics() const;     // Prometheus text format
```

`get_health_status()` returns `{"status": "ok"|"initializing", "timestamp": ...}`, where `initializing` means the engine has not finished booting. `get_metrics()` returns `records_ingested` and `bytes_ingested` as Prometheus counters.

Both return strings. The SDK serves nothing. Publish them through whatever endpoints your own process already exposes.

FermiHDIQueryClient

The read path: dispatch queries, receive result batches over FHDWP, hand back Arrow.

CONSTRUCTION AND TEARDOWN

```
explicit FermiHDIQueryClient(const std::string& proxy_url = "",
                             const std::string& data_plane_interface = "",
                             const std::string& control_plane_interface = "");
~FermiHDIQueryClient();
void stop();
```

An empty `proxy_url` selects **direct SCC mode**, where you supply targets yourself. The client opens a listener on **port 6987** for returning result batches; inbound traffic on that port must reach this host or queries will hang.

register_with_dms

```
RegistrationResult register_with_dms(const std::string& dms_url,
                                     const std::string& node_name,
                                     const std::string& override_key_pem = "");
```

Only required when querying SCCs directly. Unlike the ingest client's version, this returns a full `RegistrationResult` — certificates, dataset schema and cluster topology together — so one call configures the client.

add_control_header / set_scc_targets / load_dataset_schema

As on `FermiHDIIngestClient`.

generate_batch_ids_time

```
std::vector<BatchID> generate_batch_ids_time(int64_t start_time, int64_t end_time,
                                             int interval_override = 0);
```

Expands a time range into one batch ID per interval — the plural counterpart to the ingest client's single-window generator. Use it to name the batches a query should read.

The range is generous at both ends: the start floors to an interval boundary and the end rounds up, then the loop runs inclusively to that rounded end. The list therefore covers at least the range you asked for and typically extends one interval past it. Filter afterwards if you need an exact window.

generate_hd_query

```
std::string generate_hd_query(const HDQuery& query);
```

Serialises an `HDQuery` into the JSON wire form the query methods accept. Building the struct and calling this is preferable to assembling JSON by hand, since it applies the QoS clamping for you.

BATCH ACCOUNTING

```
std::vector<BatchID> get_found_batch_ids() const;
std::vector<BatchID> get_missing_batch_ids() const;
bool had_errors() const;
std::vector<BatchError> get_batch_errors() const;
```

`had_errors()` is **the only signal that a successful result is incomplete**. A query whose batches were bad-batch-noticed still returns a table; it just silently lacks those records. Check it after every query. `get_batch_errors()` gives one entry per failed batch, not one per notice. Both reset at the start of the next query, or at `prepare_query()`.

EXECUTING

```
void execute_stream(const std::string& sql_or_json,
                   std::function<void(std::shared_ptr<arrow::RecordBatch>)> on_batch);
std::shared_ptr<arrow::Table> execute_dataset(const std::string& sql_or_json);
```

`execute_dataset` blocks until every batch has arrived and returns one Arrow Table.

`execute_stream` **is not incremental despite its name**. It dispatches the query, blocks until the result is complete, and only then walks the accumulated batches and calls `on_batch` once per batch. The whole result is held in memory before your first callback fires, so it saves no memory over `execute_dataset()` — it only changes the shape of the hand-off, from one Table to a sequence of `RecordBatch`es. It also dispatches to the first SCC target only.

`execute_dataset` returns `nullptr` when the schema could not be loaded and the engine therefore never booted, and an **empty table** when the query timeout fired.

ANALYTICS

```
std::shared_ptr<arrow::Table> execute_to_sql(const std::string& fermihdi_sql,
                                           const std::string& duckdb_sql,
                                           const std::string& native_args = "{}");
std::shared_ptr<arrow::Table> execute_vector_search(const std::string& sql,
                                                  const std::string& vector_column,
                                                  const std::vector<float>& target,
                                                  size_t top_k,
                                                  const std::string& native_args = "{}");
void execute_graph(const std::string& sql, const std::string& src_col,
                  const std::string& dst_col, const std::string& native_args = "{}");
void set_local_aggregations(const std::string& agg_config);
```

`execute_to_sql` runs the HD query, decodes the fixed-width records into a typed columnar table, then applies `duckdb_sql` to it locally. Passing an empty `duckdb_sql` returns the raw dataset unchanged, and if decoding fails it falls back to returning the raw data rather than failing the call. Returns `nullptr` only when the underlying query returned nothing.

`apply_sql` is the second half of `execute_to_sql` on its own: the SQL over a raw table already in hand. `execute_sql_on_empty` runs the SQL over a dataset of zero records typed by the loaded schema -- the row an aggregate produces over nothing -- for a caller whose dispatch came back with no batches to read and who therefore never ran the data plane. Both decode an empty table to the schema's typed columns first, so `SELECT count(*)` answers `0` rather than nothing.

The SDK does not translate values: a filter's value must already be in the column's own form. An `ipv4` column takes the `uint32` the address is and an `ipv6` column the `uint128`; turning `'10.0.0.1'` into `167772161` is the presentation layer's job, and the storage nodes and DuckDB only ever see the integer. A filter whose value cannot be encoded is not pushed, and a sub-query left with no encodable filter means **no pushdown for the whole query** -- never an empty sub-query, which the storage node refuses and answers with no records and no error.

When the SQL itself fails -- a text literal against an integer column, say -- `execute_to_sql` returns `nullptr` and logs the DuckDB error; it does not fall back to the raw records, which are not an answer to the statement asked.

The one translation the SDK does make is the one the schema itself declares. A dataset may carry an `enums` table -- named lists of labels, each with an unsigned integer code and one label as the default -- and a `uint8`, `uint16`, `uint32` or `uint64` field may name one in its `enum` property. Such a field stores the code and nothing else. On the query path the typed table and the DuckDB view hold the labels as text, with a code the table does not define rendered as the default label; `set_decode_enums(false)` hands back the integer instead. A filter on the field takes a label, or the decimal text of a defined code, and pushes the code; an unknown label is not pushed at all, never the default, so a typo over-fetches rather than selecting the "unknown" rows. On the ingest path `convert_record_to_binary` takes a label or a defined code and stores the default's code for anything else, counting each such fall back in `enum_default_count()`, since a producer whose labels differ from the schema's maps every record to the default and should find out.

The other translation the schema declares is an array. A `bytes` field may carry an `array` object -- `element_type`, `max_elements`, and for string or numeric elements an `element_width` -- and then holds a big-endian element count followed by room for `max_elements` elements, each stored exactly as a field of the element's type would be, with zeros past the count. The field's type stays `bytes`, so a consumer that never reads the declaration still gets the bytes it was promised. On the query path the typed table holds an Arrow list of the element type (`element[]` in DuckDB), an `enum` on the field labelling each element. A filter on an array field is not pushed to the storage nodes -- a list has no single byte pattern to match -- so the records arrive and DuckDB evaluates it (`list_contains`). On the ingest path `convert_record_to_binary` takes a JSON array as the value (`"[1, 2, 3]"`, `"[\tcp\", \udp\"]"`); a scalar is one element and an empty value is none. Elements past `max_elements` are dropped from the end and counted in `array_truncation_count()`, since a producer whose lists outgrow the field is a schema-sizing problem someone should hear about.

The third translation is intra-record time compression. An integer field may carry `time` with a unit, nanoseconds through days: its value is a count of that unit since the Unix epoch. A field carrying `time_reference` names such a source field and a delta unit, and stores the signed difference from the source's value in the same record rather than a time of its own -- an `int16` of seconds or an `int32` of milliseconds where an absolute time needs eight bytes. On the query path the typed table and the DuckDB view hold the absolute time, as a 64-bit integer (`BIGINT`) in the field's own `time` unit or, when it declares none, the source's; `set_expand_times(false)` hands back the stored delta in the field's own type instead. A filter on such a field is not translated either way: its value is compared with the stored delta. On the ingest path `convert_record_to_binary` takes the absolute value and stores the delta; one the field cannot hold is clamped to its nearest bound and counted in `time_delta_clamp_count()`, and a record carrying no value for the source stores zero and is counted in `time_source_missing_count()`. On an array of integers both declarations apply to each element, measured from the same source.

The fourth translation is chess notation. A `chess_move` field is a 16-bit code that stands for one Standard Algebraic Notation move, computed from the move's parts (`src/Chess.hpp` fixes the enumeration; code 0 is no move), and a `chess_fen` field a Forsyth-Edwards position packed into 32 bytes: an 8-byte occupancy bitmask, 20 bytes of 5-bit piece tokens in occupancy order and 4 bytes of game state. On the ingest path `convert_record_to_binary` takes the SAN text and the FEN text (an array of moves takes a JSON list or a whole PGN movetext); what it cannot represent stores zero and is counted in `chess_unmapped_count()`. On the query path the typed table and the DuckDB view hold the SAN and FEN text (`VARCHAR`) while `set_decode_chess` is on, the default; off, they hold the code (`USMALLINT`) and the packed bytes (`BLOB`). A filter takes the notation either way, encoded to the stored form; `Filter::mask`, hex exactly the field wide, narrows an `EQUALITY` to the bits it sets, which is how a query service asks the storage nodes for one occupancy bit of a position. The DuckDB the SDK opens for `execute_to_sql` knows `fen_piece(fen, sq)`, `fen_occupied(fen, sq)`, `fen_turn`, `fen_castling`, `fen_en_passant`, `fen_halfmove`, `fen_fullmove`, `fen_piece_count`, `san_code(text)` and `san_text(code)`.

`decode_records` is the decoding half of `execute_to_sql` on its own: the typed columns of a raw record table already in hand, with no statement run over them.

Text is stored UTF-32 for every `string_encoding` but `ascii`, which is stored one byte per character. The ingest encoder, the decoder and the filter encoder all take the form from the field's encoding, so an `ascii` field holds width characters rather than width/4, its filter pattern is the bytes themselves, and a literal it cannot hold is not pushed.

`execute_vector_search` runs the query, builds a dense vector index over the result with cosine distance, and returns the `top_k` nearest rows to `target`. Returns `nullptr` if the underlying query produced no table.

`execute_graph` and `set_local_aggregations` are **not implemented**.

QUERY LIFECYCLE CONTROL

```
void prepare_query(const std::string& query_id);
void set_expected_batches(const std::vector<std::string>& batch_ids);
void set_filter_verdict(const std::vector<std::string>& would_exclude_batches,
                       const std::string& bloomrf_mode);
bool set_filter_verdict_json(const std::string& verdict_json);
void set_query_timeout_ms(uint64_t ms);
void cancel_query(uint32_t query_id);
```

`prepare_query()` resets query state and arms the listener **before** you dispatch, so end-of-batch markers arriving during dispatch are captured rather than discarded. `set_expected_batches()` supplies the batch list **after** the dispatch returns it, and completes the query immediately if everything already arrived. Together they remove the race between dispatch and the first marker; skip them and the SDK falls back to counting batches from the query payload.

`set_filter_verdict()` belongs to the same self-dispatch path. The SDK records the proxy's `would_exclude_batches` and `bloomrf_mode` when `submit_query_to_proxy()` places the query, and compares that list against the batches that actually deliver records when the query completes. A caller that dispatches by its own route has to hand the verdict over here, or the comparison has nothing to work with and never runs. `set_filter_verdict_json()` takes the `result` event's JSON as received. `prepare_query()` clears the previous query's verdict.

`set_query_timeout_ms()` caps how long a query blocks waiting for markers. **The default of 0 means no timeout at all** — a Storage Node that never reports will block the call indefinitely. On expiry the SDK stops waiting, sends `POST /v1/query/stop` to every known SCC target, and returns an empty table.

`cancel_query()` aborts an in-flight query by ID and is **safe to call from any thread**, which makes it the way to interrupt a blocking `execute_dataset()`. It releases the blocked call, then sends the same stop.

The stop no longer reaches the cluster. The SCC retired `POST /v1/query/stop` on 2026-08-09 and answers `410 Gone`, naming the gRPC `QueryDispatchService/CancelQuery` RPC as its replacement. Both the timeout and `cancel_query()` still do the half that is local — your call returns, and returns promptly — but the Storage Nodes are never told, so they keep reading for that query until it finishes on its own. Cancelling frees your thread, not the cluster.

OBSERVABILITY

```
std::string get_health_status() const;
std::string get_metrics() const;
```

As on `FermiHDIIngestClient`: strings for your process to publish, not endpoints the SDK serves.

Not implemented

Declared in the public header, compile and link, but have no behaviour:

Method	Actual behaviour
<code>FermiHDIIngestClient::append_record_to_arrow()</code>	Logs and returns <code>true</code> unconditionally. Nothing is appended, and nothing accumulates for <code>flush_arrow_batch()</code> to return.
<code>FermiHDIIngestClient::flush_arrow_batch()</code>	Returns <code>nullptr</code> unconditionally.
<code>FermiHDIQueryClient::set_local_aggregations()</code>	Logs the configuration string and discards it. No aggregation is applied.
<code>FermiHDIQueryClient::execute_graph()</code>	Executes the query, then logs which graph implementation it would use. Returns <code>void</code> and has no output parameter, so no result can reach the caller.

For columnar ingest today, build an `arrow::RecordBatch` with the Arrow API and pass it to the `stream_insert` overload that accepts one — that path is implemented.

8.2.4 HD SDK — Architecture

The SDK is one C++ core, `fermihdi_client`, with bindings for eight other languages built on top of it. See the [SDK overview](#) for the part that matters most — the control plane / data plane split — and the [API reference](#) for every method. This page is the shape underneath both.

Why one core, not eight implementations

A query's data plane is latency-sensitive: Storage Nodes stream records straight into the client's own listener, out of band from the request that asked for them. Reimplementing that wire protocol and the HD engine's memory handling separately in Go, Python and JavaScript would mean three copies to keep correct, and three places for the wire format to drift. Instead, every language wraps the same compiled core:

```

FHDWP transport + the HD engine (C++)
    |
    v
fermihdi_client — the public C++ API (FermiHDISDK.hpp)
    |
    v
per-language bindings — pybind11, cgo, N-API, cxx, JNA, Rcpp, ccall, or the C ABI directly

```

Both client classes (`FermiHDIQueryClient`, `FermiHDIIngestClient`) use the pimpl idiom and are non-copyable — construct one and keep it; see [What construction actually does](#).

The C ABI

`include/fermihdi_c_api.h` exports a flat, nothrow C surface from the shared library, so any language with FFI can reach the SDK without a C++ toolchain. It is narrower than the full C++ API today: it carries a target-setting call (`fhd_query_client_set_scc_target`) but no `register_with_dms` and no ingest client, which is why six of the eight bindings (Go, Rust, Scala, Julia, R, Mojo) can be handed SCC targets but cannot register as an `HD_CLIENT` node or ingest. C++, Python, and — for querying only — JavaScript go through the fuller C++ surface instead. The [wrapper maturity table](#) has the exact support matrix per language.

What runs where

Construction does real work: it probes for a GPU runtime (falling back to CPU when none is found), starts the HD engine on its own thread, reserves one core and 1 GB of memory, and blocks until the engine reports ready — see [What construction actually does](#).

Two embedded engines process query results locally, inside the client process, after the records arrive:

- **DuckDB** — `execute_to_sql()` decodes the fixed-width record table into typed Arrow columns and runs the given SQL over it locally.
- **USearch** — `execute_vector_search()` builds a dense vector index over the result with cosine distance and returns the nearest rows.

`execute_graph()` and `set_local_aggregations()` are declared in the public API but **not implemented** — see [Not implemented](#) for the full list and what each one actually does today. Nothing in the SDK currently offloads work to a GPU beyond the CPU/GPU runtime probe at construction.

Where to go next

- [SDK overview](#) — the control plane / data plane split, and proxy vs. direct mode.
- [API reference](#) — every type and method.
- [Getting started](#) — building, linking, and wiring up both clients.

8.2.5 HD SDK — Configuration

The SDK is a client library, not a service, and it has very little configuration of its own. Almost everything it needs — the proxy or DMS URL, the network interfaces, the dataset schema — is passed explicitly by the application that links it, as constructor arguments or method calls. See [Getting started](#) for the full list of what you pass and when.

What the SDK reads from the environment

Exactly two variables, and only to select the network stack:

Variable	Effect
<code>FERMIHDI_NETWORK_STACK</code>	Names the stack the engine should use. Defaults to <code>posix</code> .
<code>FERMIHDI_FORCE_POSIX</code>	<code>1</code> or <code>true</code> selects the POSIX stack even where kernel-bypass is available.

The SDK does not read environment variables for anything else — not the proxy URL, not network interfaces, not the dataset schema, not a bootstrap token. If you pass an empty string for a network interface it binds to all interfaces (`0.0.0.0`) rather than consulting the environment; see [Interfaces are yours to supply](#).

What is not the SDK's concern

A bootstrap token, a DMS registration token, and a client's own certificate material are arguments to `register_with_dms()` (or, for a token, whatever your application already holds) — the SDK stores what registration returns and uses it for later control-plane calls, but it has no configuration file or schema of its own to hold them in. If your application reads its own configuration from environment variables, a file, or a service like the DMS southbound API, that is your application's design, not something the SDK imposes or participates in.

Where to go next

- [Getting started](#) — what to pass the constructor, and what has to be reachable.
- [Interface binding](#) — the network interfaces, in detail.

8.2.6 HD SDK — Interface Binding

Both SDK clients take their network interfaces as explicit constructor arguments. **The SDK never reads an environment variable to discover an interface** — see [Interfaces are yours to supply](#) for the full explanation, including the two environment variables the SDK does read (for the network stack, not for interfaces).

Constructors

```
// Ingest client – data_plane_interface carries the FHDWP transport,
// control_plane_interface the registration and discovery calls.
// An empty string binds to all interfaces (0.0.0.0).
explicit FermiHDIIngestClient(const std::string& data_plane_interface = "",
                             const std::string& control_plane_interface = "");

// Query client – same semantics, plus the proxy URL.
explicit FermiHDIQueryClient(const std::string& proxy_url = "",
                             const std::string& data_plane_interface = "",
                             const std::string& control_plane_interface = "");
```

```
FermiHDIIngestClient ingest("eth1", "eth0"); // data, control
FermiHDIQueryClient query("https://proxy.internal:8443", "eth1", "eth0");
```

Every other binding takes the same two interfaces the same way — as arguments to its own constructor, not as an environment lookup. The [Go binding](#), for example, exposes `NewQueryClientWithInterfaces(proxyURL, dataPlaneIface, ctrlPlaneIface string)` alongside a convenience constructor that binds to all interfaces.

Naming the right interface

HD separates traffic across three planes: control, data, and observability. An SDK client only opens the first two — it serves no health or metrics endpoint of its own (see [Health and metrics are strings, not endpoints](#)), so it has no observability interface to name.

On a host or container with separate networks for each plane, name the data and control interfaces explicitly rather than leaving them empty. Binding to all interfaces works on a flat, single-homed network, but on a segmented one it means the client may send or listen on the wrong network — which for the data plane in particular can mean a Storage Node cannot reach the address the client advertises, and a query returns zero rows with no error. See [Addressing](#) for that failure mode in full.

Where to go next

- [Getting started](#) — the full explanation, with what happens when an interface is left empty.
- [Configuration](#) — the two environment variables the SDK does read.

8.2.7 HD SDK — Client Sizing

What a client costs to run, and how many of them to run.

What one client reserves

Constructing either `FermiHDIIngestClient` or `FermiHDIQueryClient` boots the HD engine on its own thread and reserves **one core and 1 GB of memory** for it — see [What construction actually does](#). That reservation is made once, in the constructor, and held for the life of the object.

Because of that reservation, a client is a long-lived object, not a per-request one:

- **Construct once per process, and keep it.** The constructor blocks until the engine reports ready, so constructing one per record or per request costs both the core/memory reservation and that startup latency, repeatedly.
- **One client per process is the normal shape.** A service that needs both read and write access holds one `FermiHDIQueryClient` and one `FermiHDIIngestClient`, not a pool of either.

GPU probing at construction

Constructing a client probes for a GPU runtime and logs which one it found, falling back to CPU routines when there is none. This is a detection step, not a promise of GPU-accelerated execution for every call — see [Architecture](#) for exactly which embedded engines exist today (`execute_to_sql()` via DuckDB, `execute_vector_search()` via USearch) and which declared methods (`execute_graph()`, `set_local_aggregations()`) are not implemented.

Sizing a host that runs several clients

Each client instance is one core and 1 GB, reserved for as long as the object lives. A host running several client processes — several query services, for example — needs that budget multiplied by however many long-lived clients it hosts, on top of whatever else those processes need. This is separate from, and much smaller than, the per-role core and hugepage budget for the engine roles (SCC, SN, the filter node) themselves, which the Administrator's Guide covers.

Languages that `dlopen` the SDK

Python, R and Node.js load the SDK's shared library into an already-running process rather than linking it at build time, which needs a static-TLS tunable set before the host process starts — see [Dynamically loaded consumers need static TLS surplus](#). This is a startup-time requirement, not an ongoing resource cost, but it has to be set correctly or the import/require fails outright.

Where to go next

- [Getting started](#) — the full construction sequence.
- [Architecture](#) — what runs inside the client process.

8.2.8 Languages

FermiHDI SDK — Python

BETA. Builds and links against a real SDK on every change, and the module imports. It has not been exercised against a live cluster as heavily as the Go binding.

pybind11 bindings over the C++ SDK. Read the [SDK overview](#) first — especially [§1, the control plane / data plane split](#).

INSTALL

The module is not on PyPI: it is a compiled extension that links the SDK, and it ships in the SDK package image alongside the library it needs. Copy the package into your image and put its Python directory on `PYTHONPATH`:

```
COPY --from=cr.fermihdi.io/hd/sdk:<version> /opt/fermihdi-sdk /opt/fermihdi-sdk
ENV PYTHONPATH=/opt/fermihdi-sdk/python LD_LIBRARY_PATH=/opt/fermihdi-sdk/lib
```

The directory holds `fermihdi_sdk.cpython-*.so`, built for the package's CPU variant and Python version (`PACKAGE.txt` records both). Match the variant to the host the client runs on: an `avx512` module raises "Illegal instruction" on a machine without AVX-512.

You will need the TLS tunable

Python `dlopen`s the extension, and the SDK carries the HD engine runtime's initial-exec thread-local storage. Without surplus static TLS the import fails:

```
ImportError: ... cannot allocate memory in static TLS block
```

```
GLIBC_TUNABLES=glibc.rtld.optional_static_tls=1048576 python3 your_app.py
```

This is a property of the SDK, not of this binding — see [§5.3](#).

API

FermiHDIQueryClient

```
FermiHDIQueryClient(proxy_url="", data_plane_iface="", ctrl_plane_iface="")

stop()
register_with_dms(dms_url, node_name, override_key_pem="") -> RegistrationResult
add_control_header(header)
set_scc_targets(targets) # [SccAddress]
load_dataset_schema(schema_json)
generate_batch_ids_time(from_sec, to_sec) -> [str]
generate_hd_query(hd_query) -> str
get_found_batch_ids() -> [str]
get_missing_batch_ids() -> [str]
prepare_query(query_id) # arm the listener BEFORE you dispatch
set_expected_batches(batch_ids) # the proxy's found_batches, AFTER
set_filter_verdict(would_exclude_batches, bloomrf_mode)
set_filter_verdict_json(verdict_json) # the proxy's result event, verbatim
get_filter_false_negatives() -> [str] # must be empty
get_bloomrf_mode() -> str
set_local_aggregations(agg_config)
execute_dataset(sql_or_json) -> ArrowTable # the raw record table, untouched
execute_dataset_count(sql_or_json) -> int
execute_to_sql(fermihdi_json, duckdb_sql) -> ArrowTable
execute_sql_on_empty(duckdb_sql) -> ArrowTable # the SQL over zero records
apply_sql(table, duckdb_sql) -> ArrowTable # SQL over a table you hold

ArrowTable # what every execute_* returns
num_rows() num_columns() schema()
to_ipc() -> bytes # pyarrow.ipc.open_stream(b).read_all()
table_from_ipc(bytes) -> ArrowTable # e.g. from pyarrow: sink.getvalue().to_pybytes()
execute_vector_search(sql, vector_column, target, top_k) -> pyarrow.Table
execute_graph(sql, src_col, dst_col)
```

Supports the context-manager protocol (`__enter__` / `__exit__`), which is the recommended way to use it — `stop()` is then guaranteed.

FermiHDIIngestClient

```
FermiHDIIngestClient(data_plane_iface="", ctrl_plane_iface="")
```

```

register_with_dms(...)          discover_scc_targets(proxy_url) -> bool
load_dataset_schema(json)      set_scc_targets(targets)
generate_batch_id_string(s)     generate_batch_id_number(n)
generate_batch_id_time(sec)     convert_record_to_binary(record)
append_record_to_arrow(record)  flush_arrow_batch(batch_id)
stream_insert(batch_id, batch, broadcast=False) -> bool
send_eob(batch_id)             send_bbn(batch_id)

```

Types

```

Filter(field_name, value, op)    # op: FilterOp.EQUALITY | NOT_EQUAL | GREATER | LESS
SubQuery(filters=[...])
SccAddress(ip, port)
HDQuery(query_id, batch_ids, sub_queries, targets,
         query_qos=1, acknowledge_full_dataset_scan=False)

```

WORKED EXAMPLE

```

import os, json, secrets
import fermihdi_sdk as fsdk

schema = json.dumps({"datasets": [{"record_len": 89, "time_index_interval": 60}]})

with fsdk.FermiHDIQueryClient("https://proxy.internal:443") as client:
    client.load_dataset_schema(schema)

    batch_ids = client.generate_batch_ids_time(1788500000, 1788503600)

    hq = fsdk.HDQuery()
    hq.query_id = secrets.randbelow(2**32)    # distinct per query
    hq.batch_ids = batch_ids
    hq.targets = [fsdk.SccAddress("10.20.0.14", 6987)]

    sq = fsdk.SubQuery()
    # An address is its integer: 10.0.0.1 is 167772161. Convert before the SDK sees it.
    sq.filters = [fsdk.Filter("ip_src", "167772161", fsdk.FilterOp.EQUALITY)]
    sq.sub_queries = [sq]

    body = client.generate_hd_query(hq)
    if not body:
        raise RuntimeError("refused: open query without acknowledge_full_dataset_scan")

    table = client.execute_to_sql(body, "SELECT ip_src, bytes FROM ipflows LIMIT 100")
    print(table.num_rows, "rows")

```

USE CASES

Notebook analysis. The context manager plus `execute_to_sql` returning a `pyarrow.Table` drops straight into pandas:

```
df = table.to_pandas()
```

Vector search over stored embeddings.

```
hits = client.execute_vector_search(body, "embedding", target_vector, top_k=20)
```

A counting query, when you want cardinality without materialising rows:

```
n = client.execute_dataset_count(body)
```

Ingest from a Python producer. Build records, append to an Arrow batch, flush under a batch ID, then mark the batch complete:

```

ing = fsdk.FermiHDIIngestClient()
ing.load_dataset_schema(schema)
ing.discover_scc_targets("https://proxy.internal:443")
bid = ing.generate_batch_id_time(1788500000)
for rec in records:
    ing.append_record_to_arrow(rec)
ing.flush_arrow_batch(bid)
ing.send_eob(bid)

```

GOTCHAS

- **generate_hd_query returning ""** is the [full-dataset-scan guard](#), not a bug. Set `acknowledge_full_dataset_scan` deliberately or narrow the query.
- **Advertise a reachable address.** `SccAddress` in `targets` is where storage nodes send records. Wrong address, zero rows, no error.

- **Two clients cannot both bind 6987** on one host.
- **This binding does not submit the query for you.** If you dispatch it yourself you own the ending — relay `complete` or `cancel` — and you own the filter verdict: pass the result event's `would_exclude_batches` and `bloomrf_mode` through `set_filter_verdict`, or the false-negative check never runs.

FermiHDI SDK — Go

BETA. The most exercised of the wrappers: four production services query through it. Still beta, because its API may change with the SDK's.

cgo bindings over the C++ SDK. Read the [SDK overview](#) first — particularly [§1, the control plane / data plane split](#), because it explains why this API looks the way it does.

MODES

This binding is built on the C ABI, which today carries no registration call and no ingest client. A Go program therefore works in **proxy mode** (submit through the Query Proxy, records over the data plane), and can be handed SCC targets with `SetSccTarget`, but it cannot register as an `HD_CLIENT` node or ingest. Direct mode and custom ingesters are C++ and Python for now; the [SDK overview](#) has the two modes side by side.

INSTALL

The package has no `go.mod` in this repository: it is vendored by consumers, and the packaged SDK writes the module line when it assembles the payload. From the package image:

```
COPY --from=cr.fermihdi.io/hd/sdk:<version> /opt/fermihdi-sdk /opt/fermihdi-sdk
RUN . /opt/fermihdi-sdk/cgo-env.sh && go build ./...
```

`cgo-env.sh` sets `CGO_ENABLED`, `CGO_CXXFLAGS`, `CGO_LDFLAGS` and `LD_LIBRARY_PATH` — including the [full library list](#) that `lfermihdi_client` alone does not pull in. Source it rather than hand-maintaining those flags.

API

```
func NewQueryClient(proxyURL string) *QueryClient
func NewQueryClientWithInterfaces(proxyURL, dataPlaneIface, ctrlPlaneIface string) *QueryClient

func (q *QueryClient) Close()
func (q *QueryClient) LoadDatasetSchema(schemaJSON string)
func (q *QueryClient) SetSccTarget(ip string, port uint16)
func (q *QueryClient) GenerateHDQuery(query HDQuery) string
func (q *QueryClient) PrepareQuery(queryID string)
func (q *QueryClient) SetExpectedBatches(batchIDs []string)
func (q *QueryClient) SetFilterVerdict(wouldExclude []string, bloomrfMode string)
func (q *QueryClient) FilterFalseNegatives() (mode string, falseNegatives []string)
func (q *QueryClient) ExecuteToIPC(fermihdiJSON, duckdbSQL string) []byte
func (q *QueryClient) ExecuteEmptyToIPC(duckdbSQL string) []byte // the SQL over zero records
func (q *QueryClient) ExecuteDatasetToIPC(fermihdiJSON string) []byte // the raw record table, untouched
func (q *QueryClient) ApplySQLToIPC(ipc []byte, duckdbSQL string) []byte // SQL over a table you hold
func (q *QueryClient) SetQueryTimeoutMs(ms uint64)
func (q *QueryClient) CancelQuery(queryID uint32)
```

Types

```
type HDQuery struct {
    QueryID          uint32
    BatchIDs         []string
    SubQueries       []SubQuery
    Targets          []SccAddress
    QueryQoS         uint32 // 1-255, priority at the storage node
    AcknowledgeFullDatasetScan bool   // see below
}

type SubQuery struct { Filters []Filter }
type Filter struct   { FieldName, Value string; Op FilterOp }
type SccAddress struct { IP string; Port uint16 }
```

`FilterOp`: Equality, NotEqual, Greater, Less.

THE ORDERING THAT MATTERS

`PrepareQuery` **before** you dispatch. Storage nodes can start streaming the moment the controllers fan out, which is before your dispatch call returns, and records arriving at an unarmed listener are discarded.

```
client := fermihdi.NewQueryClient("https://proxy.internal:443")
defer client.Close()

client.LoadDatasetSchema(schemaJSON)

hq := fermihdi.HDQuery{
    QueryID: nextQueryID(), // distinct per query — see below
    BatchIDs: batchIDs,
```

```

Targets: []fermihdi.SccAddress{{IP: advertisedIP, Port: 6987}},
SubQueries: []fermihdi.SubQuery{{
    Filters: []fermihdi.Filter{
        // An address is its integer: 10.0.0.1 is 167772161. Convert before the SDK sees it.
        {FieldName: "ip_src", Value: "167772161", Op: fermihdi.Equality},
    },
}},
}
queryJSON := client.GenerateHDQuery(hq)
if queryJSON == "" {
    return errors.New("query refused: no batch_ids and no full-scan acknowledgement")
}

// 1. arm the listener
client.PrepareQuery(strconv.FormatUint(uint64(hq.QueryID), 10))

// 2. control plane: POST to the proxy yourself, read found_batches
found, err := dispatchToProxy(proxyURL, queryJSON)
if err != nil {
    reportQueryEnd(proxyURL, hq.QueryID, "cancel") // you own the ending
    return err
}

// 3. wait for exactly what exists, and say what the filter would have skipped
client.SetExpectedBatches(found)
client.SetFilterVerdict(wouldExclude, bloomrfMode) // from the same result event

// 4. data plane: blocks until every expected batch has arrived
ipc := client.ExecuteToIPC(queryJSON, "SELECT ip_src, bytes FROM ipflows LIMIT 10")

// 5. tell the cluster it is over
reportQueryEnd(proxyURL, hq.QueryID, "complete")

```

Query IDs must be distinct

The ID ties together the listener, the returning end-of-batch markers, and the ending you report. Two queries sharing an ID are indistinguishable at every one of those points — a completion for one untracks the other.

Seed a counter **at random**, not from the clock: services that start in the same second otherwise issue identical first IDs.

```

var queryIDSeq atomic.Uint32

func init() {
    var b [4]byte
    if _, err := rand.Read(b[:]); err == nil {
        queryIDSeq.Store(binary.BigEndian.Uint32(b[:]))
    } else {
        queryIDSeq.Store(uint32(time.Now().UnixNano()) ^ uint32(os.Getpid())<<16)
    }
}

func nextQueryID() uint32 { return queryIDSeq.Add(1) }

```

You own the ending

This binding does not submit the query, so it cannot report the ending either. Whatever route you dispatched by, relay `complete` or `cancel` back the same way — on success, on error, on timeout, and on your own shutdown.

Omitting `cancel` is the costly one: the storage nodes keep reading for a query nobody is waiting for.

READING RESULTS

`ExecuteToIPC` returns Arrow IPC bytes, or an empty slice when there is nothing. When the dispatch itself came back with no batches to read -- an empty window, or a filter service that excluded every candidate -- there is nothing to wait for and `ExecuteToIPC` is never called; `ExecuteEmptyToIPC(sql)` then gives the row an aggregate produces over nothing (count 0, sum NULL), so the answer keeps its shape.

```

if len(ipc) == 0 {
    return nil, nil // no rows; not an error
}
reader, err := ipc.NewReader(bytes.NewReader(ipcBytes))

```

USE CASES

A query service. Hold one `QueryClient` for the process lifetime, generate a distinct ID per request, and relay endings. This is what `hd_sql`, `hd_mcp`, `hd_graphql` and `hd_ipflow_graphql` do.

A batch job over a wide window. Set `AcknowledgeFullDatasetScan` in code — you have decided a full scan is what you mean — and raise `SetQueryTimeoutMs`.

Behind NAT. Bind locally, advertise the routable address, and make sure the advertised host and port appear in `Targets`. Records sent where nothing is listening are lost with no error on either side.

GOTCHAS

- `GenerateHDQuery` returning `""` is the [full-dataset-scan guard](#).
- Waiting on batches that were never stored is how a query hangs to its timeout. Always `SetExpectedBatches(found)`.
- The SDK only learns what the filter service would have excluded when it places the query itself. A Go service reads the proxy's stream, so it must pass `would_exclude_batches` and `bloomrf_mode` on with `SetFilterVerdict`, or the false-negative check never runs and passive mode measures nothing.
- `close()` releases the C++ client. It does not cancel a running query — send the cancel first.

FermiHDI SDK — JavaScript / Node.js

BETA. Builds and links on every change. Not yet exercised against a live cluster as heavily as the Go binding.

An N-API addon over the C++ SDK. Read the [SDK overview](#) first, especially [§1, the control plane / data plane split](#) — records never come back through the connection you submit on.

BUILD

```
export FERMIHDI_SDK_ROOT=/opt/fermihdi-sdk
npm install
npm run node-gyp rebuild
```

`binding.gyp` resolves headers and libraries through `FERMIHDI_SDK_ROOT` (default `/opt/fermihdi-sdk`) and sets two flags that are not optional:

- `-std=c++20` — Arrow's headers use `std::span` and `std::bit_width`. node-gyp's default standard fails inside `arrow/buffer.h`.
- `-frtti` — `arrow/util/checked_cast.h` uses `dynamic_cast`, and node-gyp disables RTTI by default.

It also names the [full library list](#), because `-lfermihdi_client` alone does not pull in the HD engine runtime's dependencies.

You will need the TLS tunable

Node `dlopen`s the addon, and the SDK carries the HD engine runtime's initial-exec TLS:

```
GLIBC_TUNABLES=glibc.rtld.optional_static_tls=1048576 node app.js
```

Otherwise `require()` fails with *cannot allocate memory in static TLS block* — see [§5.3](#).

API

```
const { QueryClient } = require('./build/Release/fermihdi_sdk');

const client = new QueryClient(proxyUrl);

client.register_with_dms(dmsUrl, nodeName) // -> { success, schema }
client.load_dataset_schema(schemaJson)
client.generate_batch_ids(from, to) // -> [hex string]
client.generate_batch_ids_time(fromSec, toSec)
client.generate_hd_query(hdQueryObject) // -> query body JSON, "" if refused
client.prepare_query(queryId) // arm the listener FIRST
client.set_expected_batches(batchIds) // after the control plane answers
client.set_filter_verdict(wouldExclude, mode) // the proxy's would_exclude_batches, bloomrf_mode
client.filter_false_negatives() // -> { bloomrf_mode, false_negatives }
client.execute_to_ipc(queryJson, duckdbSql) // -> Buffer of Arrow IPC
client.execute_empty_to_ipc(duckdbSql) // the SQL over zero records, when nothing was dispatched
client.execute_dataset_to_ipc(queryJson) // the raw record table, untouched, as Arrow IPC
client.apply_sql_to_ipc(ipcBuffer, duckdbSql) // SQL over a table you hold, in and out as Arrow IPC
client.stop()
```

WORKED EXAMPLE

```
const { QueryClient } = require('./build/Release/fermihdi_sdk');
const crypto = require('crypto');

const client = new QueryClient('https://proxy.internal:443');
client.load_dataset_schema(schemaJson);

const batchIds = client.generate_batch_ids_time(1788500000, 1788503600);
const queryId = crypto.randomInt(0, 2 ** 32); // distinct per query

const body = client.generate_hd_query({
  query_id: queryId,
  batch_ids: batchIds,
  targets: [{ ip: advertisedIp, port: 6987 }],
  // An address is its integer: 10.0.0.1 is 167772161. Convert before the SDK sees it.
  sub_queries: [{ filters: [{ field_name: 'ip_src', value: '167772161', op: 0 } ]}],
});
if (!body) throw new Error('open query refused without acknowledgement');

// 1. arm the listener before anything can answer
client.prepare_query(String(queryId));

// 2. control plane — your own HTTP call to the proxy
const found = await dispatchToProxy(proxyUrl, body);

// 3. wait for exactly what exists, and say what the filter would have skipped
client.set_expected_batches(found);
client.set_filter_verdict(wouldExclude, bloomrfMode); // from the same result event
```

```
// 4. data plane – blocks until every expected batch has arrived
const ipc = client.execute_to_ipc(body, 'SELECT ip_src, bytes FROM ipflows LIMIT 10');

// 5. report the ending
await reportQueryEnd(proxyUrl, queryId, 'complete');

client.stop();
```

`execute_to_ipc` returns a `Buffer` of Arrow IPC bytes; decode it with `apache-arrow`:

```
const { tableFromIPC } = require('apache-arrow');
const table = tableFromIPC(ipc);
```

USE CASES

A Node API in front of HD. Hold one client for the process lifetime, mint a distinct query ID per request, and relay endings. Note that `execute_to_ipc` **blocks the event loop** — run it in a worker thread if the process serves other traffic.

A CLI or ETL step. Blocking is fine; keep it simple.

GOTCHAS

- `generate_hd_query` returning `""` is the [full-dataset-scan guard](#).
- `execute_to_ipc` **is synchronous and blocking**. Use a worker thread in a server.
- **You own the ending** — this addon does not submit the query.
- **Advertise a reachable address**, or you get zero rows and no error.

FermiHDI SDK — Rust

BETA. Builds on every change (`cargo build`), but the exposed surface is narrow — a bridge, not yet a full binding. Expect to extend it.

A **cxx** bridge: Rust calls the C++ SDK directly through generated shims, so the ownership and ordering rules of the C++ API apply unchanged. The [SDK overview](#) explains those rules, starting with [§1, the control plane / data plane split](#).

MODES

The bridge exposes the C ABI's query client and nothing else: `set_scc_target` is there, `register_with_dms` and an ingest client are not, because the C ABI does not have them yet. A Rust program runs in **proxy mode** through the Query Proxy, or can be handed SCC targets by something that registered on its behalf, but cannot itself register as an `HD_CLIENT` node or ingest. The [SDK overview](#) compares the modes and says which bindings have each.

ADD THE CRATE

The crate ships in the SDK package image at `/opt/fermihdi-sdk/rust`. Copy the package into your build image and reference the crate by path:

```
[dependencies]
fermihdi-sdk = { path = "/opt/fermihdi-sdk/rust" }
```

Linking against the real library needs the [full library list](#) — `-lfermihdi_client` alone leaves `fmt`, `hwloc`, `NORM`, `boost`, `c-ares` and `yaml-cpp` undefined, because the shared object embeds the HD engine's runtime through a static archive and does not re-export its dependencies.

```
// build.rs
println!("cargo:rustc-link-search=native={}/lib", sdk_root);
for lib in ["fermihdi_client", "arrow", "norm", "fmt", "hwloc", "yaml-cpp",
            "boost_program_options", "boost_thread", "cares", "curl"] {
    println!("cargo:rustc-link-lib=dllib={lib}");
}
```

API

The bridge currently exposes:

```
fn new_ingest_client() -> UniquePtr<FermiHDIIngestClient>;
fn load_dataset_schema(self: Pin<&mut FermiHDIIngestClient>, schema: &CxxString);

fn new_query_client(proxy_url: &CxxString) -> UniquePtr<FermiHDIQueryClient>;
fn generate_hd_query(self: Pin<&mut FermiHDIQueryClient>, query: &HDQuery)
    -> UniquePtr<CxxString>;
fn execute_to_sql(self: Pin<&mut FermiHDIQueryClient>,
    hd_sql: &CxxString, duck_sql: &CxxString) -> bool;
fn prepare_query(self: Pin<&mut FermiHDIQueryClient>, query_id: &CxxString);
fn execute_sql_on_empty(self: Pin<&mut FermiHDIQueryClient>, duck_sql: &CxxString) -> bool;
fn set_filter_verdict_json(self: Pin<&mut FermiHDIQueryClient>,
    verdict_json: &CxxString) -> bool;
```

`QueryClient` wraps the last two as `prepare_query(&str)` and `set_filter_verdict_json(&str)`. The verdict is for a query you dispatched yourself: the proxy's `result` event JSON, verbatim, so the SDK knows what the filter service would have excluded. Without it the false-negative check at the end of the query never runs.

Everything is `UniquePtr` / `Pin<&mut>` because these are C++ objects with C++ lifetimes; `cxx` will not let you forget it.

EXAMPLE

```
use cxx::let_cxx_string;

let_cxx_string!(proxy = "https://proxy.internal:443");
let mut client = ffi::new_query_client(&proxy);

let_cxx_string!(schema = SCHEMA_JSON);
// ... populate an HDQuery, including a distinct query_id and reachable targets

let body = client.pin_mut().generate_hd_query(&query);
if body.is_empty() {
    // the full-dataset-scan guard refused an open query
    return Err("open query not acknowledged".into());
}
```

```
let_cxx_string!(sql = "SELECT ip_src, bytes FROM ipflows LIMIT 10");
let ok = client.pin_mut().execute_to_sql(&body, &sql);
```

USE CASES

A Rust ingest or query sidecar that must not carry a Python or JVM runtime. Link the [static archive](#) and ship one binary.

Extending the bridge. The C++ API is much wider than what is exposed here. Add the method to the `#[cxx::bridge]` block and it becomes available; the compiler will tell you immediately if the signature is wrong, which is the main reason to prefer cxx over hand-written FFI for this.

GOTCHAS

- **An empty `generate_hd_query` result is the [full-dataset-scan guard](#).**
- **You own the ending.** This bridge does not submit the query, so relay `complete` or `cancel` by whatever route you dispatched.
- **Query IDs must be distinct** across every client talking to the cluster — seed at random, not from the clock.

FermiHDI SDK — Scala

BETA. Compiles on every change. **JDK 17 only** — see below. Narrow surface; expect to extend it.

Scala binds the C ABI of the shared library through JNA, which means the JVM loads `libfermihdi_client.so` at runtime and the SDK's threads run beside the JVM's own. The [SDK overview](#) sets out the model those threads implement; read [§1](#), [the control plane / data plane split](#) before anything here.

MODES

JNA reaches the C ABI's query client. There is no registration call and no ingest client behind it yet, so from Scala the SDK is used in **proxy mode**, submitting through the Query Proxy with records arriving over the data plane; SCC targets can be set explicitly, but the JVM process cannot register as an `HD_CLIENT` node or ingest. See the [SDK overview](#) for the two modes and the bindings that support each.

BUILD

```
JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64 sbt compile
```

JDK 17 is a hard requirement. `scalaVersion` is pinned to 3.3.0, which cannot read the class files of JDK 21 or newer — it fails while loading `java.base` itself:

```
error while loading AccessFlag, class file .../AccessFlag.class is broken
bad constant pool index: 0
```

That is the compiler, not this binding. Raise `scalaVersion` if you need a newer JDK.

The JVM `dlopen`s the library, so you also need the [static TLS surplus](#):

```
GLIBC_TUNABLES=glibc.rtld.optional_static_tls=1048576 sbt run
```

API

```
trait SDKCore {
  def discoverSCCTargets(proxyUrl: String): Boolean
  def executeDataset(sql: String): Boolean
}

class NativeQueryClient(proxyUrl: String) { // over the C ABI
  def prepareQuery(queryId: String): Unit // arm the listener FIRST
  def setFilterVerdict(verdictJson: String): Boolean
  def filterFalseNegatives(): String // JSON: bloomrf_mode, false_negatives
  def close(): Unit
}
```

`setFilterVerdict` is for a query you dispatched yourself: pass the proxy's `result` event JSON verbatim so the SDK knows what the filter service would have excluded. Without it the false-negative check at the end of the query never runs.

`filterFalseNegatives` must report an empty list.

New calls are JNA declarations against [the C ABI](#) — flat, nothrow and stable, and a far better target than the C++ symbols.

EXAMPLE

```
import com.sun.jna.Native

val sdk = Native.load("fermihdi_client", classOf[SDKCore])
if (!sdk.discoverSCCTargets("https://proxy.internal:443"))
  sys.error("no storage cluster controllers discovered")
```

USE CASES

Spark or Flink jobs that need HD as a source without a Python hop. Pull a window into Arrow and hand it to the JVM pipeline.

Extending the binding. Map `fhd_*` from `fermihdi_c_api.h`; the buffer-filling `*_into` variants avoid JNA memory-ownership questions entirely.

GOTCHAS

- **JDK 17.** Anything newer fails to compile Scala 3.3.0.

- **You own the ending** — relay `complete` or `cancel`.
- **Query IDs must be distinct** across all clients; seed at random.

FermiHDI SDK — Julia

BETA. Loads and is exercised on every change, but the surface is narrow.

Julia reaches the SDK through `ccall` into the C ABI of the shared library; there is no Julia-side runtime, so every handle you receive is owned by the SDK and freed by the calls described below. Start with the [SDK overview](#), in particular §1, the [control plane / data plane split](#).

LOAD

```
include("FermiHDISDK.jl")
using .FermiHDISDK
```

The library is found through `FERMIHDI_SDK_ROOT/lib`, falling back to the loader's search path. Julia `dlopen`s it, so you need the [static TLS surplus](#):

```
GLIBC_TUNABLES=glibc.rtld.optional_static_tls=1048576 julia script.jl
```

MODES

The `ccall` surface is the C ABI's query client, which has a target-setting call but no `register_with_dms` and no ingest client. Julia therefore queries in **proxy mode**, or against SCC targets it was given, and neither registers as an `HD_CLIENT` node nor ingests; those need C++ or Python today. The [SDK overview](#) lays out both modes.

API

```
QueryClient(proxy_url)           # construct
load_dataset_schema(client, schema_json)
generate_hd_query(client, query) -> String
set_filter_verdict(client, verdict_json) -> Bool # for a query you dispatched yourself
filter_false_negatives(client) -> String         # JSON: bloomrf_mode, false_negatives
stop(client)

FilterOp                         # EQUALITY, NOT_EQUAL, GREATER, LESS
```

EXAMPLE

```
include("FermiHDISDK.jl")
using .FermiHDISDK

client = QueryClient("https://proxy.internal:443")
load_dataset_schema(client, read("schema.json", String))

body = generate_hd_query(client, query)
if isempty(body)
    error("open query refused without acknowledge_full_dataset_scan")
end

stop(client)
```

USE CASES

Analysis alongside Julia's numerics. Pull a window into Arrow and work on it with the DataFrames and statistics ecosystem, without moving data through another runtime.

Extending the binding. New calls are `ccall`s against the [C ABI](#) (`include/fermihdi_c_api.h`), which is flat, nothrow and stable — a better target than the C++ symbols.

GOTCHAS

- **You own the ending.** This binding does not submit queries; relay `complete` or `cancel` yourself.
- **An empty query body is the [full-dataset-scan guard](#).**
- **Advertise an address the storage nodes can reach**, or the query returns nothing and reports no error.

FermiHDI SDK — R

BETA. Builds, installs and loads on every change. Newly repaired: this package could not compile in any checkout before 2026-09-05, because it called a method the SDK has never had.

An Rcpp package: R functions wrap the C++ SDK classes one to one, so an R session drives the same query and ingest clients the other bindings do. The [SDK overview](#) describes those clients; its [§1, the control plane / data plane split](#) is the part an R user most needs.

MODES

The Rcpp package wraps the C++ query client through the C ABI, so it has what the C ABI has: querying in **proxy mode**, and explicit SCC targets, but no `register_with_dms` and no ingest client. An R session cannot register as an `HD_CLIENT` node or ingest until the C ABI gains those calls. The [SDK overview](#) shows the two modes side by side.

INSTALL

```
export FERMIHDI_SDK_ROOT=/opt/fermihdi-sdk
GLIBC_TUNABLES=glibc.rtld.optional_static_tls=1048576 R CMD INSTALL r
```

Both settings are required:

- `FERMIHDI_SDK_ROOT` tells `src/Makevars` where the headers and library are. It also sets `CXX_STD = CXX20`, because R defaults to `gnu++17` and the SDK headers will not parse under it, and passes the [full library list](#) with `-Wl,--no-as-needed` — without which the package links cleanly and then fails at `dyn.load()` on an undefined boost symbol.
- `GLIBC_TUNABLES` because R `dlopen`s the package and the SDK carries the HD engine runtime's initial-exec TLS. Without it: `cannot allocate memory in static TLS block`. See [§5.3](#).

If you are rebuilding after editing, remove stale objects first — R reuses any `.o` it finds, so a changed `Makevars` can appear to do nothing:

```
rm -f r/src/*.o r/src/*.so
```

API

```
sdk_version()                # SDK version string
sdk_discover_targets(proxy_url) # logical
sdk_health(proxy_url)         # JSON: status (liveness), ready
sdk_load_schema(proxy_url, schema_json) # logical
sdk_submit_query(proxy_url, query_json, query_id)
# list(ok, empty, found_batches, excluded_batches,
#       would_exclude_batches, bloomrf_mode, error)
sdk_filter_verdict(proxy_url, verdict_json) # JSON: bloomrf_mode, false_negatives
```

`excluded_batches` are batches a filter service proved cannot match: not in `found_batches`, nothing will arrive for them, and not an error. `would_exclude_batches` is what an enforcing filter *would* have excluded in `passive` mode; those were dispatched anyway. `bloomrf_mode` says which of `off`, `passive`, `active` or `mixed` the cluster is in.

`sdk_submit_query()` places the query through the proxy **and holds the connection for the life of the query**, reporting the ending itself — see [§2.3](#). You do not need to relay endings when you use it.

EXAMPLE

```
library(FermiHDISDK)

cat(sdk_version(), "\n")

health <- jsonlite::fromJSON(sdk_health("https://proxy.internal:443"))
if (!isTRUE(health$ready)) message("client is alive but not ready yet")

schema <- readLines("schema.json", warn = FALSE)
sdk_load_schema("https://proxy.internal:443", paste(schema, collapse = ""))

res <- sdk_submit_query(
  "https://proxy.internal:443",
  query_json,
  query_id = sample.int(.Machine$integer.max, 1) # distinct per query
)

if (!res$ok) stop("dispatch failed: ", res$error)
```

```
if (res$empty) message("no batches in that window")
length(res$found_batches)
```

USE CASES

Statistical analysis over a time window. Resolve the batches through `sdk_submit_query`, then work in R's native ecosystem.

Health checks from an R service. `sdk_health()` distinguishes liveness from readiness — wire `status` to a liveness probe and `ready` to a readiness probe; they are deliberately different answers.

GOTCHAS

- `status: "ok"` **with** `ready: false` **is normal** for a client whose engine has not booted. It is alive; restarting it will not make the proxy reachable.
- **Stale .o files** silently defeat `Makevars` changes.
- **Query IDs must be distinct** across all clients talking to the cluster.

FermiHDI SDK — Mojo

BETA. Written from scratch in September 2026 and built on every change. Newest of the wrappers; expect the least mileage.

A real binding to the FermiHDI SDK, built on the C ABI in `include/fermihdi_c_api.h`. Every function in `FermiHDISDK.mojo` reaches a symbol in `libfermihdi_client.so`.

HISTORY, BECAUSE IT MATTERS

Two files used to sit here and were deleted on 2026-08-27 because neither worked and both reported success while doing nothing -- one had its single FFI call commented out. They had a reason to cheat: **the SDK exported no C symbols at all**. Go worked only because its C shim (`go/sdk_bridge.cpp`) is compiled into the cgo package, which is no help to a language with plain FFI.

The `fhd_*` surface now lives in the shared library, so there is something real to bind to.

BUILDING

```
mojo build your_program.mojo \
-Xlinker -L$FERMIHDI_SDK_ROOT/lib \
-Xlinker -lfermihdi_client \
-Xlinker -rpath -Xlinker $FERMIHDI_SDK_ROOT/lib
```

Symbols resolve at link time rather than through `dlopen`, because `dlsym` in this Mojo returns a data pointer rather than a callable and there is no supported way to call through it.

For a static binary, link the archive instead and drop the `rpath`:

```
mojo build your_program.mojo \
-Xlinker $FERMIHDI_SDK_ROOT/lib/libfermihdi_client_static.a \
-Xlinker -lstc++
```

MODES

Mojo calls the C ABI's query client directly. That ABI carries a target-setting call and no registration or ingest call, which leaves Mojo in **proxy mode** (or querying SCC targets it was given) and unable to register as an `HD_CLIENT` node or to ingest. The [SDK overview](#) has the modes and their per-binding support.

API

```
sdk_version() -> String           # read from the library, not hardcoded

struct QueryClient:
  QueryClient(proxy_url, data_plane_iface="", ctrl_plane_iface="",
              return_port=6987)
  is_open()   -> Bool
  close()     # explicit; see the note below
  stop()
  load_schema(schema_json) -> Bool
  set_scc_target(ip, port)
  health()     -> String # JSON: status (liveness), ready
  generate_query(hdquery_json) -> String # "" if the full-scan guard refuses
  prepare_query(query_id) # arm the listener FIRST
  submit_to_proxy(query_json, query_id) -> String
  # JSON: {"ok":..,"empty":..,"found_batches":[..],"error":..}
  set_timeout_ms(ms)
  set_filter_verdict(verdict_json) -> Bool # for a query you dispatched yourself
  false_negatives() -> String # JSON: bloomrf_mode, false_negatives
  cancel_query(query_id)
```

`set_filter_verdict()` is for a query placed by your own route rather than `submit_to_proxy()`: pass the proxy's `result` event JSON verbatim so the SDK knows what the filter service would have excluded. Without it the false-negative check at the end of the query never runs. `false_negatives()` must come back with an empty list; one entry is a filter defect.

`submit_to_proxy()` places the query **and holds the connection for the life of the query**, reporting the ending itself — see [§2.3](#). Nothing else to relay.

USE CASES

A Mojo service querying HD directly, with no Python or JVM runtime in the image. Link the [static archive](#) for a single binary.

Numeric work on returned records, close to where they land, without an intermediate copy through another runtime.

USAGE

```

from FermiHDISDK import QueryClient, sdk_version

def main():
    print(sdk_version())
    var client = QueryClient(String("https://proxy.example:443"))
    if not client.is_open():
        return
    _ = client.load_schema(schema_json)
    print(client.health())
    client.close()

```

GOTCHAS

- **close() is explicit.** An `external_call` inside `__deinit__` crashes the current Mojo compiler outright -- `kgen.call_param op` invalid use of parameter `callee`, reproducible with a three-line struct -- so the destructor cannot be the thing that frees.
- **The full-dataset-scan guard** returns `""` from `generate_query` — see §3.2.
- **Query IDs must be distinct** across every client talking to the cluster; seed at random rather than from the clock.
- **Advertise an address the storage nodes can reach**, or the query returns nothing and reports no error.

8.3 Global Developer Guide

Welcome to the FermiHDI ecosystem. Build and contribution guidance lives with the component it applies to — each submodule carries its own developer guide for its C++ or Go source, its tests and its build.

To	Read
Build the images	IMAGE_BUILD.md
Work on a component	that submodule's <code>docs/developer_guide.md</code>
Understand the system first	Architecture
Run one locally	tests/sandbox_dms

Changes to the installer — the tooling under `install/` — are described in [The installer](#), which also says where a change belongs: anything that asks the operator a question goes in `fermihdi-configure` or in `scripts/`, and anything that applies an already-recorded decision goes in `fermihdi-prepare`.

8.4 Global Test Guide

All PRs enforce GitHub Actions evaluating coverage across `codecov` integrations natively. Each submodule holds the unit tests for its own source; what runs across components lives in `tests/`.

Harness	Exercises
<code>tests/integration_test_full_pipeline</code>	Kafka → Ingester → SCC → SN → query via Proxy → test head
<code>tests/integration_test_dms_pipeline</code>	registration, provisioning and the DMS side of that path
<code>tests/integration_test_sc_cluster</code>	a Storage Cluster on its own
<code>tests/sandbox_dms</code>	the full deployment, and the benchmark suite

`tests/run_tests.sh` is the entry point. [End-to-end test](#) describes the pipeline and the chain-of-custody checks in detail.

8.5 FermiHDI End-to-End Test Documentation

This describes a retired harness. `run_e2e_tests.py` and `data_generator.py` are no longer in the repository, and the image names and environment contract below predate the 2.x images — the DMS is two images rather than one, and nodes take their configuration from the DMS registration response rather than from a mounted config file. It is kept for the pipeline and chain-of-custody description, which still holds. The harnesses that run today are:

Test	Exercises
<code>tests/integration_test_full_pipeline</code>	Kafka → Ingester → SCC → SN → query via Proxy → test head
<code>tests/integration_test_dms_pipeline</code>	registration, provisioning and the DMS side of the same path
<code>tests/integration_test_sc_cluster</code>	a Storage Cluster on its own
<code>tests/sandbox_dms</code>	the full deployment, and the benchmark suite

The end-to-end integration test (`run_e2e_tests.py`) validates the entirety of the FermiHDI Data Management stack, proving that configuration mappings, mTLS certificate bootstrapping, parsing ingress pipelines, and underlying storage deployments work flawlessly within a containerized environment.

This document describes exactly how the test is structured, how data is processed, and how the various containers are interconnected.

8.5.1 Test Overview

- Environment Setup:** Tears down any lingering state and creates a dedicated Docker network (`hd_e2e_net`) on the `172.25.0.0/16` subnet.
- Data Generation:** Executes `data_generator.py` locally to build an expected schema (`test_dataset_schema.json`), generate millions of rows grouped into CSV batches, and export baseline DuckDB validated target query answers.
- Container Orchestration:** Instead of relying heavily on `-v` volume mounts which can present synchronization bugs or directory assumption bugs across platforms, the script extensively utilizes `docker create`, `docker cp`, and `docker start` commands to inject files immutably before booting a service.
- Cryptographic Verification:** The `data_generator.py` now supports multiple hashing algorithms (defaulting to Hybrid XXH64). It calculates a continuous chain hash across all batches. Each batch's `opening_hash` and `closing_hash` are recorded in `batch_mapping.csv`, allowing the system to verify the "Chain of Custody" during ingestion and storage audits.
- Ingestion Triggering:** After the PKI cluster binds securely, the orchestration script moves the newly generated `.csv` files into the `hd_message_bus_ingester`'s input engine directories.
- Validation Run:** Queries are forwarded directly against the Data Plane proxy to ensure they properly route payloads downstream spanning Storage Nodes (SN), and the Query Verifier assesses accuracy.

8.5.2 Container Topology and Configuration

The test environment provisions eight distinct Docker containers interconnected within the statically assigned `172.25.0.0/16` IP range.

DMS (Data Management Server)

- **Image:** `cr.fermihdi.io/hd/dms-core:2.0.3`
- **IP:** `172.25.0.10`
- **Ports:** `6986:6986` (Southbound API), `3000:3000` (Web UI)
- **Execution:** `sh -c "dms-ca & dms-core"`

Observability Stack (Telemetry)

- **Image:** `otel/opentelemetry-collector-contrib:latest` and `getsentry/relay:latest`
- **IPs:** `172.25.0.11` (OTel), `172.25.0.12` (Sentry Relay)
- **Ports:** `4317` (OTLP), `3000` (Relay)
- **Role:** Managed dynamically by the DMS OpAMP supervisor on port `4320`. The collector's configuration is rendered by DMS-Core; the reference copy is `hd_dms/config/otel-collector.yaml`.
- **Note:** Overrides the endpoint to run the Certificate Authority sidecar concurrently to allow zero-touch provisioning.
- **File Injections:**
 - `dms-core.crt` -> `dms:/etc/fermihdi/x509/dms-core.crt`
 - `dms-core.key` -> `dms:/etc/fermihdi/x509/dms-core.key`

FermiHDI Proxy

- **Image:** `cr.fermihdi.io/hd/proxy:2.0.3`
- **IP:** `172.25.0.20`
- **Ports:** `6987:6987` (Data Plane)
- **Environment Variables:**
 - `HD_PROXY_CONFIG=/tmp/config.json`
 - `HD_DATASET_SCHEMA=/tmp/test_dataset_schema.json`
- **File Injections:**
 - `proxy_config.json` -> `proxy:/tmp/config.json`
 - `test_dataset_schema.json` -> `proxy:/tmp/test_dataset_schema.json`
 - `fermihdi_root_ca.pem` -> `proxy:/tmp/dms_ca.crt` (The matched Root CA to verify `dms-core`'s certificate)

Message Bus Ingestor

- **Image:** `cr.fermihdi.io/hd/ingester:2.0.3`
- **IP:** `172.25.0.30`
- **Startup Flags:** `-c /test_data/ingester_config.json`
- **File Injections:**
 - Host directory `/test_data/` -> `ingester:/test_data`
 - Host directory `/empty_dir/` -> `ingester:/input`

SCC (Storage Cluster Controller)

- **Image:** `cr.fermihdi.io/hd/scc:2.0.3`
- **IP:** `172.25.0.50`
- **Environment Variables:**
 - `DMS_URL=https://172.25.0.10:6986`

Storage Nodes (SN1, SN2, SN3)

- **Image:** `cr.fermihdi.io/hd/sn:2.0.3`
- **IPs:** `172.25.0.101`, `172.25.0.102`, `172.25.0.103`

- **Environment Variables:**

- `DMS_URL=https://172.25.0.10:6986`

- **File Injections:**

- Host directory `snX_drives/` -> `snX:/drives` (Contains an `emulated.dat` sparse file acting as a 2GB raw block device for DPDK testing)

Query Target (Verifier)

- **Image:** `cr.fermihdi.io/hd/test-head:2.0.3`

- **IP:** `172.25.0.200`

- **File Injections:**

- Host directory `/test_data/` -> `query_target:/app/test_data`

8.5.3 Network Flow & mTLS Execution Phase

When the containers spin up, they coordinate over the following strict security lifecycle:

1. **PKI Bootstrapping:** Proxy, Ingester, SCC, and the SN instances natively utilize the `DMS_URL=https://172.25.0.10:6986` configuration property. They will securely handshake using normal TLS validation verifying `dms-core.crt` against the root CAs provided to them.
2. **Registration and Singing:** Nodes send an authenticated CSR to the DMS. The DMS uses its local `dms-ca` process internally to blindly sign those specific requests (in testing/demonstration) and hand back the signed End-Entity verification artifacts.
3. **Data Flow:** The script actively executes `docker cp` to transport `.csv` files into `ingester:/input/{f}`. The Ingester polling loop instantly catches the file and converts the contents into the binary high-speed Data Plane formats and transmits it across the `172.25.0.0` subspace to the respective SCC and SN infrastructure reliably without test script intervention.

8.5.4 Hash-Chain Auditing

To ensure zero-tampering from generation to query, the `batch_mapping.csv` file contains two cryptographic linkage fields: * `opening_hash` : The state of the cryptographic chain before the first record of the batch was processed. * `closing_hash` : The state of the cryptographic chain after the last record was processed.

The `XXH64_HYBRID` scheme uses an 8-byte chain value with a 64-bit seed rotation function derived from SHA512 at the end of each batch. For long-term archival, the `SHA512` algorithm can be selected using the `--algo sha512` flag on the generator.

9. Procurement and security

9.1 Security and development practices

This page is for security, risk and procurement reviews. It describes how HD is written, checked, built and delivered, and what you can verify yourself. The data HD shares with FermiHDI, which is nothing unless you opt in, is covered in [Data handling and opt-in](#). The open-source components and their licences are in [Third-party software and licences](#).

9.1.1 Where the code is written and kept

HD's source code is written and housed one step removed from its developers, in a secure enclave. Developers work against the enclave rather than holding the code on their own machines, so a lost or compromised laptop does not carry the product's source with it. Every change reaches the product through a reviewed pull request.

9.1.2 What checks every change

Four independent systems examine the code and its dependencies before and after a change merges:

System	What it checks
Macroscopic	Reviews each pull request for correctness defects: logic errors, concurrency and resource-lifetime faults, broken edge cases. Findings are answered in the pull request and resolved before it merges.
Aikido	Scans the source, dependencies, container images, infrastructure definitions and CI workflows for vulnerabilities, leaked secrets, insecure code patterns and end-of-life software.
Renovate	Keeps dependencies, base images and CI actions current by opening update requests, so security fixes upstream reach HD promptly.
Trivy	Scans the container images for known vulnerabilities in the operating-system packages and libraries they contain.

9.1.3 Release 2.0.5 security work

The 2.0.5 release closed the findings those systems raised against the code that ships:

- **Dependencies.** Components with published advisories were updated, including the gRPC, WebSocket and JWT libraries in the Go services, and the web framework, sanitiser and image libraries in the web applications. The few that no update yet resolves, and that HD cannot reach, are covered by the exploitability statements below.
- **Web applications.** Diagrams render with the strictest setting of the diagram library and are sanitised before display. Data Explore accepts only plain column names, a fixed set of operators and whole-number bucket widths when it builds a query.
- **Services.** Every HTTP server bounds how long a client may take to send its request headers. Keys and certificates are written only to private, uniquely named locations. The crash vault's management interface refuses every request until its access token is configured.
- **CI and supply chain.** Every third-party build action is pinned to an exact commit, build jobs do not keep repository credentials they do not need, and release workflows treat their inputs as data rather than script.
- **Kubernetes defaults.** The management and query services run without privilege escalation, without raw-socket access and under the runtime's default system-call filter. The engine roles keep only the elevated access they need to drive network cards and storage devices directly.

Findings that do not apply to HD as delivered, such as a test fixture or a vendor sample program that is never built, are recorded in the scanners with the reason, not left open. Findings in operating-system packages of the images are addressed by rebuilding on current base images, and are described per release in the vulnerability exploitability statements below.

9.1.4 Signed images

Every HD release image in `cr.fermihdi.io/hd`, from 2.0.5 on, is signed with FermiHDI's cosign key, and signing is part of publishing a release. The signature is stored beside the image in the registry, and it is deliberately not written to the public Sigstore transparency log, which would publish your image names and tags. The public key is at <https://keys.fermihdi.io/cosign.pub>.

Verify an image before you deploy it:

```
curl -fsSLO https://keys.fermihdi.io/cosign.pub
cosign verify --key cosign.pub --insecure-ignore-tlog=true \
  cr.fermihdi.io/hd/dms-core:2.0.5-ub2604
```

The `--insecure-ignore-tlog` flag tells cosign not to look for a transparency-log entry. It does not weaken the check against the key.

9.1.5 Provenance in every image

Every image carries three labels naming exactly what it was built from:

Label	Value
<code>org.opencontainers.image.source</code>	The source repository
<code>org.opencontainers.image.revision</code>	The commit
<code>com.fermihdi.git.branch</code>	The branch that carries the commit (<code>detached</code> on 2.0.5 images, which were built from pinned submodule checkouts; the commit identifies the source either way)

Read them with `docker inspect --format '{{json .Config.Labels}}' <image>`. Every repository commit that makes up a release is tagged with the release version, so a label resolves to one reviewed commit.

9.1.6 Vulnerability exploitability (VEX)

A scanner lists every known vulnerability in every package an image contains, whether or not HD can reach it. For each release, FermiHDI publishes OpenVEX statements for the findings that do not affect HD, for example a vulnerable command-line tool in the base image that HD never runs, with the reason. They are attached to each image in the registry as signed attestations and published beside the release downloads. Scanners that read OpenVEX, Trivy among them, use them to show only the findings that matter:

```
cosign verify-attestation --key cosign.pub --insecure-ignore-tlog=true \
  --type openvex cr.fermihdi.io/hd/dms-core:2.0.5-ub2604
```

9.1.7 Security built into the product

- **Mutual TLS everywhere.** Every node and service proves its identity with a certificate issued by the deployment's own certificate authority, chained to FermiHDI's root. Node certificates renew themselves in place before they expire.
- **Three separate planes.** Management traffic, record traffic and observability traffic use separate ports, and records never cross the management plane. See the [operators manual](#) and the [FHDWP network guide](#).
- **Your identity provider, your front doors.** HD does not hold user passwords. People and service accounts are authenticated by the reverse proxies you run, and HD authorises on the identity they pass on. See [The front doors](#).
- **Licences bound to the deployment.** A licence is encrypted and bound to the DMS it was issued for, and capacity is enforced.
- **Nothing leaves by default.** No telemetry or crash data leaves your deployment unless you turn sharing on. See [Data handling and opt-in](#).

9.2 Data handling and opt-in

HD runs entirely inside your environment. Your records, schemas, queries and user identities stay there. The only data that can leave is operational telemetry and crash information, and only when an administrator opts in. This page sets out what that data is, how you control it, where it goes and who can see it.

9.2.1 The short version

- **Off by default.** A new deployment shares nothing with FermiHDI.
- **One decision, held centrally.** The opt-in is a single switch in the DMS WebUX, held by DMS-Core for the whole deployment. It is not a per-browser or per-node setting.
- **Enforced at two points.** Only two components can send anything to FermiHDI. The deployment's OpenTelemetry collector forwards operational telemetry, and the crash vault forwards crash stacks, both only while the switch is on. A full crash report goes only when an operator shares it.
- **Crash memory never leaves on its own.** A crash report's memory contents are sent only when an operator shares that one report.
- **One destination.** Shared data goes to FermiHDI's own service in DigitalOcean's Singapore region (SGP1). No other company or software-as-a-service provider receives it.
- **A closed network.** That service runs in a private network with no public access. It accepts only mutual TLS connections from registered, opted-in HD deployments, and FermiHDI staff reach it only through recorded Tailscale or Border0 sessions.

9.2.2 Three kinds of data

Kind	Examples	Leaves your deployment?
Your data	Records, dataset schemas and names, node names, IP addresses, user identities, query text, anything shown in a console	Never. It is not part of what the collector forwards, and FermiHDI's intake discards anything outside the permitted fields listed below.
Operational telemetry	Licence ID, product version and build, node and cluster counts, feature usage counts, latency measurements, error types, crash stacks without memory	Only while the opt-in is on.
Full crash reports	The memory regions captured at a crash, which can contain record bytes, key material or a query in flight	Only when an operator presses <i>Share full report</i> for that report.

Permitted telemetry fields

Operational telemetry leaves with these attributes and no others. Everything else is removed by your collector before sending, and discarded again at FermiHDI's intake:

```
service.name, service.version, service.namespace, deployment.environment, fermihdi.build.commit, fermihdi.product,
fermihdi.license_id, fermihdi.region, host.arch, os.type, fermihdi.opt_in, http.request.method, http.response.status_code,
http.method, http.status_code, http.route, rpc.method, rpc.service, error.type, fermihdi.feature, fermihdi.node_role,
log.level, event.category, event.action, gen_ai.operation.name, gen_ai.tool.name.
```

9.2.3 How opt-in works

1. **Installation.** The installer can disable sharing outright (`fermihdi_dms_enable_analysis: false`). A deployment installed that way cannot be switched on from the console.

2. **The switch.** An administrator turns on *Cloud sharing* on the WebUX *Telemetry & Logs* page. DMS-Core records the decision and it survives restarts.
3. **Registration.** DMS-Core registers the deployment with FermiHDI's opt-in service over mutual TLS, using a certificate chained to FermiHDI's root. The registration carries the licence ID, the region, a certificate request for the collector's identity, and the crash vault's relay identity and public key. None of these is a credential or customer data.
4. **The collector.** DMS-Core renders the collector's configuration. The FermiHDI export pipeline exists in that configuration only while sharing is on and the registration is valid. It removes every attribute not on the permitted list, then sends over mutual TLS.
5. **Turning it off.** Turning the switch off revokes the registration and removes the export pipeline. The collector stops forwarding immediately.

No engine node or service is told about the decision, and none of them changes what it reports. They always report to your own collector and crash vault, and those two decide what, if anything, goes further. For network controls, the collector reaches FermiHDI's telemetry endpoint and the crash vault its crash endpoint, both in the Singapore region.

9.2.4 Filters and destinations you control

- **The opt-in switch** decides whether the FermiHDI pipeline exists at all.
- **The permitted-field filter** in your collector is fixed. Nothing outside the list above can be added to what FermiHDI receives.
- **Your own destinations.** The WebUX *OTEL Collector* page sends the deployment's full telemetry to systems you choose (OTLP over HTTP or gRPC, syslog, Prometheus remote write, Kafka), independently of the opt-in. That data goes only where you point it.
- **The console's own usage telemetry** follows the same switch, or the `FERMIHDI_OPT_IN` environment variable on the WebUX container, which turns it on regardless of the switch. When on, the WebUX reports page loads and request timings to your own collector, with URLs reduced to their paths, and no addresses, user agents or identity headers. That variable does not make the collector forward anything to FermiHDI; only the switch does.

9.2.5 The crash vault

Every crash in the deployment is captured on your own infrastructure first. The engine roles report through Crashpad, DMS-Core and the Proxy through the Go runtime, and the WebUX through Node.js. Reports go to the crash vault, a service on the DMS host.

- **Held locally.** The vault keeps each report, symbolicated on the box against the symbol files shipped with the release, for 30 days or until the vault reaches 2 GB, whichever comes first.
- **Stack only with the opt-in.** With sharing on, the vault sends the stack of a new crash: symbolicated frames, release and tags, with no memory and no attachments. With sharing off, it sends nothing.
- **Full reports per incident.** Memory contents go only when an operator shares a specific report, optionally tagged with a support ticket, after a confirmation.
- **Air-gapped support.** *Export to file* writes a report to a tar file for support conversations on air-gapped deployments.
- **Audited.** Every share and export is recorded in the deployment's audit log with who did it and when.
- **Access controlled.** The vault's management interface serves only the WebUX, with a shared access token. Without the token configured it refuses every request.

9.2.6 Where shared data goes

Shared telemetry and crash data go to FermiHDI's opt-in service in DigitalOcean's Singapore region (SGP1), on infrastructure FermiHDI controls. DigitalOcean's data centres and platform, including SGP1, are covered by DigitalOcean's SOC 2 Type II report and ISO/IEC 27001 certification. DigitalOcean provides the infrastructure only and has no product-level access to the data.

Access to the opt-in service

The opt-in service runs in a private network (a DigitalOcean VPC) with no public access. The only connections it accepts from outside are these:

- **Opted-in HD deployments, over mutual TLS.** Registration uses a certificate chained to FermiHDI's root, as described in [How opt-in works](#). Telemetry and crash data are accepted only from deployments that have opted in and registered, using the identity issued at registration. Any other connection is refused.
- **FermiHDI staff.** Staff reach the service only through Tailscale or Border0, and those sessions are recorded.

FermiHDI sends none of this data to any other external system: no third-party analytics, advertising, error-tracking or other software-as-a-service provider receives it.

9.2.7 If you never opt in

Nothing described above leaves your network. HD operates fully without the opt-in, and every capability, including crash capture, symbolication and export, works locally.

9.3 Third-party software and licences

HD is proprietary software from FermiHDI Limited. It is built on open-source components, and this page lists the ones that ship to you: what each is, its licence, and where HD uses it. It is written for procurement, legal and security reviews, so it names the components exactly, including the asynchronous framework the HD engine is built on.

9.3.1 What the licences require of you

Nothing beyond keeping the notices. Every component HD links into its own binaries is under a permissive licence (Apache-2.0, MIT, BSD, ISC, Zlib, BSL-1.0 or curl), which asks for attribution and, for Apache-2.0, the NOTICE text; none of them requires HD's own source to be published. The copyleft components are kept at arm's length:

- **libnuma** and **libusb** (LGPL-2.1) are linked dynamically, so they can be replaced independently of HD, as the LGPL requires.
- **Pebble** (GPL-3.0) is the service manager inside each Rock image: a separate program HD runs services under, not code HD is built from.
- **BusyBox** (GPL-2.0) is in the `-symbols` images, which carry symbol files for crash symbolication, and the Kubernetes chart runs the public `busybox` image as the storage node's `init-storage` init container, which prepares the storage file before the node starts. In both it is an unmodified, separate program; its source is available from the BusyBox project.
- **Components offered under a choice of licences** (jitterentropy, liburing, zstd, Eclipse Paho) are used under the permissive option.

The source of every GPL and LGPL component is available from its upstream project and, for packages taken from Ubuntu, from the Ubuntu archive; FermiHDI supplies it on request.

Each image also carries the licence texts of the operating-system packages it contains, under `/usr/share/doc`.

9.3.2 Native components (C and C++)

The engine roles (SCC, storage node, ingester and filter node), the SDK and the crash vault are built from these. The last column says how the licence was confirmed: from the licence file or the Debian/Ubuntu package metadata in the image the component is built in, or from the component's published upstream licence where it is built from source.

Component	Licence	Used in	Confirmed from
The HD engine's asynchronous I/O framework (Seastar)	Apache-2.0	SCC, storage node, ingester, filter node, SDK	upstream licence
DPDK (Data Plane Development Kit)	BSD-3-Clause	SCC, storage node, ingester, filter node, SDK	upstream licence (user-space libraries only)
SPDK (Storage Performance Development Kit)	BSD-3-Clause	Storage node	upstream licence
Intel ISA-L	BSD-3-Clause	SCC, storage node	upstream licence
Apache Arrow and Parquet	Apache-2.0	Every engine role, SDK, ingester	licence file in the build image
DuckDB	MIT	SDK and the query services built on it	upstream licence
gRPC	Apache-2.0	Every engine role, SDK	upstream licence
Protocol Buffers	BSD-3-Clause	Every engine role, SDK	package metadata
Abseil	Apache-2.0	Every engine role, SDK	package metadata
OpenTelemetry C++	Apache-2.0	Every engine role	upstream licence
sentry-native	MIT	SCC, storage node, ingester, filter node (crash reporting)	upstream licence
Crashpad	Apache-2.0	SCC, storage node, ingester (crash reporting)	upstream licence
OpenSSL 3	Apache-2.0	Every component	package metadata
Boost (program_options, thread)	BSL-1.0	Every engine role	package metadata
{fmt}	MIT	Every engine role	upstream licence
yaml-cpp	MIT	Every engine role	package metadata
hwloc	BSD-3-Clause	Every engine role	package metadata
c-ares	MIT	Every engine role	upstream licence
libcurl	curl	Every engine role	package metadata
NORM (NACK-Oriented Reliable Multicast)	BSD-3-Clause	Every engine role	package metadata
jitterentropy	BSD-3-Clause (dual-licensed; used under BSD)	Every engine role (through OpenSSL)	upstream licence
liburing	MIT (dual-licensed; used under MIT)	SCC, storage node, SDK	upstream licence
libnuma (numactl)	LGPL-2.1-only	SCC, storage node (dynamically linked)	upstream licence
libusb 1.0	LGPL-2.1-or-later	SCC and filter node hash offload (dynamically linked, optional)	package metadata
RE2	BSD-3-Clause	Every engine role	upstream licence

Component	Licence	Used in	Confirmed from
simdjson	Apache-2.0	SCC, storage node, filter node	upstream licence
glaze	MIT	Ingester	upstream licence
nlohmann/json	MIT	Every engine role, ingester, SDK	package metadata
xxHash	BSD-2-Clause	SCC, storage node, filter node	package metadata
USearch	Apache-2.0	SDK	upstream licence
AdaptiveCpp	BSD-2-Clause	SCC, storage node, filter node, SDK (GPU filtering, optional)	upstream licence
LLVM (opt, llc, lld)	Apache-2.0 WITH LLVM-exception	SCC, storage node (filter kernel compilation)	upstream licence
Apache Avro C	Apache-2.0	Ingester	upstream licence
librdkafka	BSD-2-Clause	Ingester	upstream licence
rabbitmq-c	MIT	Ingester	upstream licence
Eclipse Paho MQTT C	EPL-2.0 OR BSD-3-Clause (used under BSD)	Ingester	upstream licence
libxml2	MIT	Ingester (XML files)	package metadata
zlib	Zlib	Every component	package metadata
zstd	BSD-3-Clause (dual-licensed; used under BSD)	Arrow and Parquet codecs	package metadata
LZ4	BSD-2-Clause	Arrow and Parquet codecs	package metadata
Snappy	BSD-3-Clause	Arrow and Parquet codecs	package metadata
Brotli	MIT	Arrow and Parquet codecs	package metadata
utf8proc	MIT	Arrow	package metadata
Apache Thrift	Apache-2.0	Parquet, SDK	upstream licence
pybind11	BSD-3-Clause	SDK (Python binding)	upstream licence
minidump-stackwalk (rust-minidump)	MIT	Crash vault	upstream licence

9.3.3 Runtimes and base images

Component	Licence	Used in	Confirmed from
Ubuntu 26.04 LTS base	Various; each package's licence ships in the image under /usr/share/doc	Every <code>-ub2604</code> image and every Rock	distribution
Pebble	GPL-3.0-only	Every Rock image (the service manager, a separate program)	upstream licence
Go runtime and standard library	BSD-3-Clause	DMS, Proxy, SQL, GraphQL, MCP services, front door	upstream licence
Node.js	MIT	DMS WebUX, Data Explore	upstream licence
OpenTelemetry Collector Contrib	Apache-2.0	The DMS-managed OpenTelemetry collector image	upstream licence
BusyBox	GPL-2.0-only	The <code>-symbols</code> images, and the storage node's <code>init-storage</code> init container in the Kubernetes chart	upstream licence

9.3.4 Go modules

The Go services (DMS, Proxy, SQL server, GraphQL services, MCP server and the service-account front door) compile in 129 third-party modules. Each licence below was read from the module's own licence file: 59 MIT, 45 Apache-2.0, 21 BSD-3-Clause, 3 BSD-2-Clause, 1 ISC.

Module	Licence	Used in
github.com/99designs/gqlgen	MIT	IP Flow GraphQL service
github.com/agnivade/levenshtein	MIT	DMS, IP Flow GraphQL service, Proxy
github.com/andybalholm/brotli	MIT	GraphQL service, MCP server
github.com/apache/arrow/go/v12	Apache-2.0	GraphQL service, MCP server
github.com/apache/arrow/go/v14	Apache-2.0	IP Flow GraphQL service, SQL server
github.com/apache/thrift	Apache-2.0	GraphQL service, MCP server
github.com/bahlo/generic-list-go	BSD-3-Clause	MCP server
github.com/beorn7/perks	MIT	DMS, GraphQL service, MCP server, Proxy, SQL server
github.com/buger/jsonparser	MIT	MCP server
github.com/bytedance/sonic	Apache-2.0	MCP server
github.com/cenkalti/backoff/v5	MIT	DMS, IP Flow GraphQL service, Proxy
github.com/cespare/xxhash/v2	MIT	DMS, GraphQL service, IP Flow GraphQL service, MCP server, Proxy, SQL server
github.com/chenzhuoyu/base64x	Apache-2.0	MCP server
github.com/decred/dcrd/dcrec/secp256k1/v4	ISC	DMS, Proxy
github.com/felixge/httpsnoop	MIT	Proxy
github.com/fsnotify/fsnotify	BSD-3-Clause	GraphQL service, IP Flow GraphQL service, MCP server
github.com/gabriel-vasile/mimetype	MIT	MCP server
github.com/gin-contrib/sse	MIT	MCP server
github.com/gin-gonic/gin	MIT	MCP server
github.com/go-jose/go-jose/v4	Apache-2.0	DMS
github.com/go-logr/logr	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
github.com/go-logr/stdr	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
github.com/go-playground/locales	MIT	MCP server
github.com/go-playground/universal-translator	MIT	MCP server
github.com/go-playground/validator/v10	MIT	MCP server
github.com/go-viper/mapstructure/v2	MIT	IP Flow GraphQL service
github.com/gobwas/glob	MIT	DMS, Proxy
github.com/goccy/go-json	MIT	DMS, GraphQL service, IP Flow GraphQL service, MCP server, Proxy, SQL server
github.com/goccy/go-yaml	MIT	IP Flow GraphQL service
github.com/golang/snappy	BSD-3-Clause	GraphQL service, MCP server
github.com/google/flatbuffers	Apache-2.0	GraphQL service, IP Flow GraphQL service, MCP server, SQL server

Module	Licence	Used in
github.com/google/uuid	BSD-3-Clause	DMS, IP Flow GraphQL service, Proxy
github.com/gorilla/websocket	BSD-3-Clause	DMS, IP Flow GraphQL service
github.com/graphql-go/graphql	MIT	GraphQL service
github.com/grpc-ecosystem/grpc-gateway/v2	BSD-3-Clause	DMS, IP Flow GraphQL service, Proxy
github.com/hashicorp/golang-lru/v2	MIT	IP Flow GraphQL service
github.com/inpopop/jsonschema	MIT	MCP server
github.com/jackc/pgpassfile	MIT	SQL server
github.com/jackc/pgservicefile	MIT	SQL server
github.com/jackc/pgx/v5	MIT	SQL server
github.com/json-iterator/go	MIT	MCP server
github.com/klauspost/asmfmt	MIT	GraphQL service, MCP server
github.com/klauspost/compress	Apache-2.0	GraphQL service, IP Flow GraphQL service, MCP server, SQL server
github.com/klauspost/cpuid/v2	MIT	GraphQL service, IP Flow GraphQL service, MCP server, SQL server
github.com/kr/text	MIT	GraphQL service, MCP server
github.com/leodido/go-urn	MIT	MCP server
github.com/lestrrat-go/blackmagic	MIT	DMS, Proxy
github.com/lestrrat-go/dsig	MIT	DMS, Proxy
github.com/lestrrat-go/dsig-secp256k1	MIT	DMS, Proxy
github.com/lestrrat-go/httpcc	MIT	DMS, Proxy
github.com/lestrrat-go/httpc/v3	MIT	DMS, Proxy
github.com/lestrrat-go/jwx/v3	MIT	DMS, Proxy
github.com/lestrrat-go/option/v2	MIT	DMS, Proxy
github.com/mailru/easyjson	MIT	MCP server
github.com/mattn/go-isatty	MIT	MCP server
github.com/metoro-io/mcp-golang	MIT	MCP server
github.com/minio/asm2plan9s	Apache-2.0	GraphQL service, MCP server
github.com/minio/c2goasm	Apache-2.0	GraphQL service, MCP server
github.com/modern-go/concurrent	Apache-2.0	MCP server
github.com/modern-go/reflect2	Apache-2.0	MCP server
github.com/munnerz/goautoneg	BSD-3-Clause	DMS, GraphQL service, MCP server, Proxy, SQL server
github.com/open-policy-agent/opa	Apache-2.0	DMS, Proxy
github.com/open-telemetry/opamp-go	Apache-2.0	DMS
github.com/pelletier/go-toml/v2	MIT	IP Flow GraphQL service, MCP server

Module	Licence	Used in
github.com/pierrec/lz4/v4	BSD-3-Clause	GraphQL service, IP Flow GraphQL service, MCP server, SQL server
github.com/pkg/errors	BSD-2-Clause	MCP server
github.com/prometheus/client_golang	Apache-2.0	DMS, GraphQL service, MCP server, Proxy, SQL server
github.com/prometheus/client_model	Apache-2.0	DMS, GraphQL service, MCP server, Proxy, SQL server
github.com/prometheus/common	Apache-2.0	DMS, GraphQL service, MCP server, Proxy, SQL server
github.com/prometheus/procfs	Apache-2.0	DMS, GraphQL service, MCP server, Proxy, SQL server
github.com/ravilushqa/otelgqlgen	Apache-2.0	IP Flow GraphQL service
github.com/rcrowley/go-metrics	BSD-2-Clause	DMS, Proxy
github.com/sagikazarmark/locafero	MIT	IP Flow GraphQL service
github.com/santhosh-tekuri/jsonschema/v5	Apache-2.0	IP Flow GraphQL service, SQL server
github.com/segmentio/asm	MIT	DMS, Proxy
github.com/sirupsen/logrus	MIT	DMS, Proxy
github.com/sosodev/duration	MIT	IP Flow GraphQL service
github.com/sourcegraph/conc	MIT	IP Flow GraphQL service
github.com/spf13/afero	Apache-2.0	IP Flow GraphQL service
github.com/spf13/cast	MIT	IP Flow GraphQL service
github.com/spf13/pflag	BSD-3-Clause	IP Flow GraphQL service
github.com/spf13/viper	MIT	IP Flow GraphQL service
github.com/subosito/gotenv	MIT	IP Flow GraphQL service
github.com/tchap/go-patricia/v2	MIT	DMS, Proxy
github.com/tidwall/gjson	MIT	MCP server
github.com/tidwall/match	MIT	MCP server
github.com/tidwall/pretty	MIT	MCP server
github.com/tidwall/sjson	MIT	MCP server
github.com/twitchyliquid64/golang-asm	BSD-3-Clause	MCP server
github.com/ugorji/go/codec	MIT	MCP server
github.com/urfave/cli/v3	MIT	IP Flow GraphQL service
github.com/valyala/fastjson	MIT	DMS, Proxy
github.com/vektah/gqlparser/v2	MIT	DMS, IP Flow GraphQL service, Proxy
github.com/wk8/go-ordered-map/v2	Apache-2.0	MCP server
github.com/xeipuuv/gojsonpointer	Apache-2.0	DMS, Proxy

Module	Licence	Used in
github.com/xeipuuv/gojsonreference	Apache-2.0	DMS, Proxy
github.com/xeipuuv/gojsonschema	Apache-2.0	DMS, Proxy
github.com/yashtewari/glob-intersection	Apache-2.0	DMS, Proxy
github.com/zeebo/xxh3	BSD-2-Clause	GraphQL service, IP Flow GraphQL service, MCP server, SQL server
go.opentelemetry.io/auto/sdk	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.opentelemetry.io/contrib	Apache-2.0	IP Flow GraphQL service
go.opentelemetry.io/contrib/instrumentation/net/http/otelhttp	Apache-2.0	Proxy
go.opentelemetry.io/otel	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.opentelemetry.io/otel/exporters/otlp/otlpmetric/otlpmetrichttp	Apache-2.0	IP Flow GraphQL service
go.opentelemetry.io/otel/exporters/otlp/otlptrace	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.opentelemetry.io/otel/exporters/otlp/otlptrace/otlptracehttp	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.opentelemetry.io/otel/metric	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.opentelemetry.io/otel/sdk	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.opentelemetry.io/otel/sdk/metric	Apache-2.0	IP Flow GraphQL service
go.opentelemetry.io/otel/trace	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.opentelemetry.io/proto/otlp	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
go.yaml.in/yaml/v2	MIT	DMS, GraphQL service, MCP server, Proxy
go.yaml.in/yaml/v3	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
golang.org/x/arch	BSD-3-Clause	MCP server
golang.org/x/crypto	BSD-3-Clause	DMS, MCP server, Proxy
golang.org/x/mod	BSD-3-Clause	GraphQL service, IP Flow GraphQL service, MCP server, SQL server
golang.org/x/net	BSD-3-Clause	DMS, IP Flow GraphQL service, MCP server, Proxy, SQL server
golang.org/x/sync	BSD-3-Clause	DMS, IP Flow GraphQL service, MCP server, Proxy, SQL server
golang.org/x/sys	BSD-3-Clause	DMS, GraphQL service, IP Flow GraphQL service, MCP server, Proxy, SQL server
golang.org/x/telemetry	BSD-3-Clause	IP Flow GraphQL service, MCP server, SQL server
golang.org/x/text	BSD-3-Clause	DMS, IP Flow GraphQL service, MCP server, Proxy, SQL server
golang.org/x/tools	BSD-3-Clause	GraphQL service, IP Flow GraphQL service, MCP server, SQL server
golang.org/x/xerrors	BSD-3-Clause	

Module	Licence	Used in GraphQL service, IP Flow GraphQL service, MCP server, SQL server
google.golang.org/genproto/googleapis/api	Apache-2.0	DMS, IP Flow GraphQL service, Proxy
google.golang.org/genproto/googleapis/rpc	Apache-2.0	DMS, IP Flow GraphQL service, Proxy, SQL server
google.golang.org/grpc	Apache-2.0	DMS, IP Flow GraphQL service, Proxy, SQL server
google.golang.org/protobuf	BSD-3-Clause	DMS, GraphQL service, IP Flow GraphQL service, MCP server, Proxy, SQL server
gopkg.in/yaml.v3	Apache-2.0	GraphQL service, MCP server
sigs.k8s.io/yaml	MIT	DMS, Proxy

9.3.5 JavaScript packages

The web applications (DMS WebUX and Data Explore) and the SDK's JavaScript binding use these packages directly; their own dependencies are under the same family of permissive licences. Licences were read from each package's lock-file entry.

Package	Licence	Used in
@astrojs/node	MIT	DMS WebUX, Data Explore
@astrojs/react	MIT	Data Explore
@astrojs/vue	MIT	DMS WebUX, Data Explore
@fontawesome/fontawesome-svg-core	MIT	Data Explore
@fontawesome/free-solid-svg-icons	CC-BY-4.0 AND MIT	Data Explore
@fontawesome/vue-fontawesome	MIT	Data Explore
@nanostores/react	MIT	Data Explore
@nanostores/vue	MIT	Data Explore
@opentelemetry/api	Apache-2.0	DMS WebUX
@opentelemetry/auto-instrumentations-node	Apache-2.0	DMS WebUX
@opentelemetry/context-zone	Apache-2.0	DMS WebUX
@opentelemetry/exporter-metrics-otlp-grpc	Apache-2.0	DMS WebUX
@opentelemetry/exporter-metrics-otlp-http	Apache-2.0	DMS WebUX
@opentelemetry/exporter-trace-otlp-grpc	Apache-2.0	DMS WebUX
@opentelemetry/exporter-trace-otlp-http	Apache-2.0	DMS WebUX
@opentelemetry/instrumentation	Apache-2.0	DMS WebUX
@opentelemetry/instrumentation-document-load	Apache-2.0	DMS WebUX
@opentelemetry/instrumentation-fetch	Apache-2.0	DMS WebUX
@opentelemetry/resources	Apache-2.0	DMS WebUX
@opentelemetry/sdk-metrics	Apache-2.0	DMS WebUX
@opentelemetry/sdk-node	Apache-2.0	DMS WebUX
@opentelemetry/sdk-trace-base	Apache-2.0	DMS WebUX
@opentelemetry/sdk-trace-web	Apache-2.0	DMS WebUX
@opentelemetry/semantic-conventions	Apache-2.0	DMS WebUX
@sentry/astro	MIT	Data Explore
@sentry/node	MIT	Data Explore
apache-arrow	Apache-2.0	Data Explore
astro	MIT	DMS WebUX, Data Explore
chart.js	MIT	DMS WebUX
chess.js	BSD-2-Clause	Data Explore
dom-to-image-more	MIT	Data Explore
echarts	Apache-2.0	Data Explore
html2canvas	MIT	Data Explore
jspdf	MIT	Data Explore

Package	Licence	Used in
mermaid	MIT	DMS WebUX, Data Explore
nanostores	MIT	Data Explore
node-addon-api	MIT	Data Explore, SDK (JavaScript binding)
react	MIT	Data Explore
react-dom	MIT	Data Explore
react-querybuilder	MIT	Data Explore
vue	MIT	DMS WebUX, Data Explore
vue-chartjs	MIT	DMS WebUX

Font Awesome's free icons are CC-BY-4.0 (the icons) and MIT (the code); the attribution is kept in the Data Explore bundle.

9.3.6 Python packages

The DMS command-line tool and the SDK's Python binding use these.

Package	Licence
httpx	BSD-3-Clause
httpx-sse	MIT
rich	MIT
typer	MIT
pyarrow	Apache-2.0

9.3.7 Keeping this list current

The list is regenerated for every release from the dependency manifests of the code that ships (Go module vendor trees, npm lock files, the native build definitions and the images' package databases). Dependency updates arrive through automated update requests and are scanned for known vulnerabilities and licence changes before they merge; see [Security and development practices](#).