HD Administrator Manual

Hyperscale Dataset Store — Deployment and Operations Guide — v2.0.3

© *FermiHDI Limited*

Table of Contents

1. HD Administrator and Deployment Manual 3
   - 1.1 System Overview & Architecture 3
   - 1.2 Deploying an Instance 4
   - 1.3 Bare-Metal Host Preparation for Accelerated Storage Nodes 6
   - 1.4 Networking & Storage Configurations 7
   - 1.5 Security & TLS/PKI Requirements 8
   - 1.6 Troubleshooting & Operational Tasks 8
2. HD System Architecture 10
   - 2.1 Holistic System Topology 10
   - 2.2 Component Architecture Breakdown 10
   - 2.3 Host Requirements for Storage Nodes with SPDK/DPDK 11
   - 2.4 Further Reading 14
3. HD Ecosystem Overview 15
4. Installation 16
   - 4.1 HD — Administrator's Guide 16
   - 4.2 install 28
   - 4.3 install/ansible 33
   - 4.4 install/helm/fermihdi 39
   - 4.5 Installing HD on Talos Linux 41
5. Operations 47
   - 5.1 HD: Operators & Admins Manual 47
   - 5.2 Sandbox Data Management System (DMS) Operators & Admins Manual 51
   - 5.3 FermiHDI HD Wire Protocol — Network & MTU Guide 53
   - 5.4 Global User Guide 56
6. Development 57
   - 6.1 HD SDK — Developer Guide 57
   - 6.2 SDK 62
   - 6.3 Global Developer Guide 79
   - 6.4 Global Test Guide 80
   - 6.5 FermiHDI End-to-End Test Documentation 81
7. FermiHDI 3rd Party Licensing Documentation 84
   - 7.1 Core C++ Dependencies (Data Plane & Operations) 84
   - 7.2 Go Dependencies (Control Plane) 85
   - 7.3 Web UI Dependencies (Astro/Javascript) 85
   - 7.4 Summary Conclusion 86

© FermiHDI Limited

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.

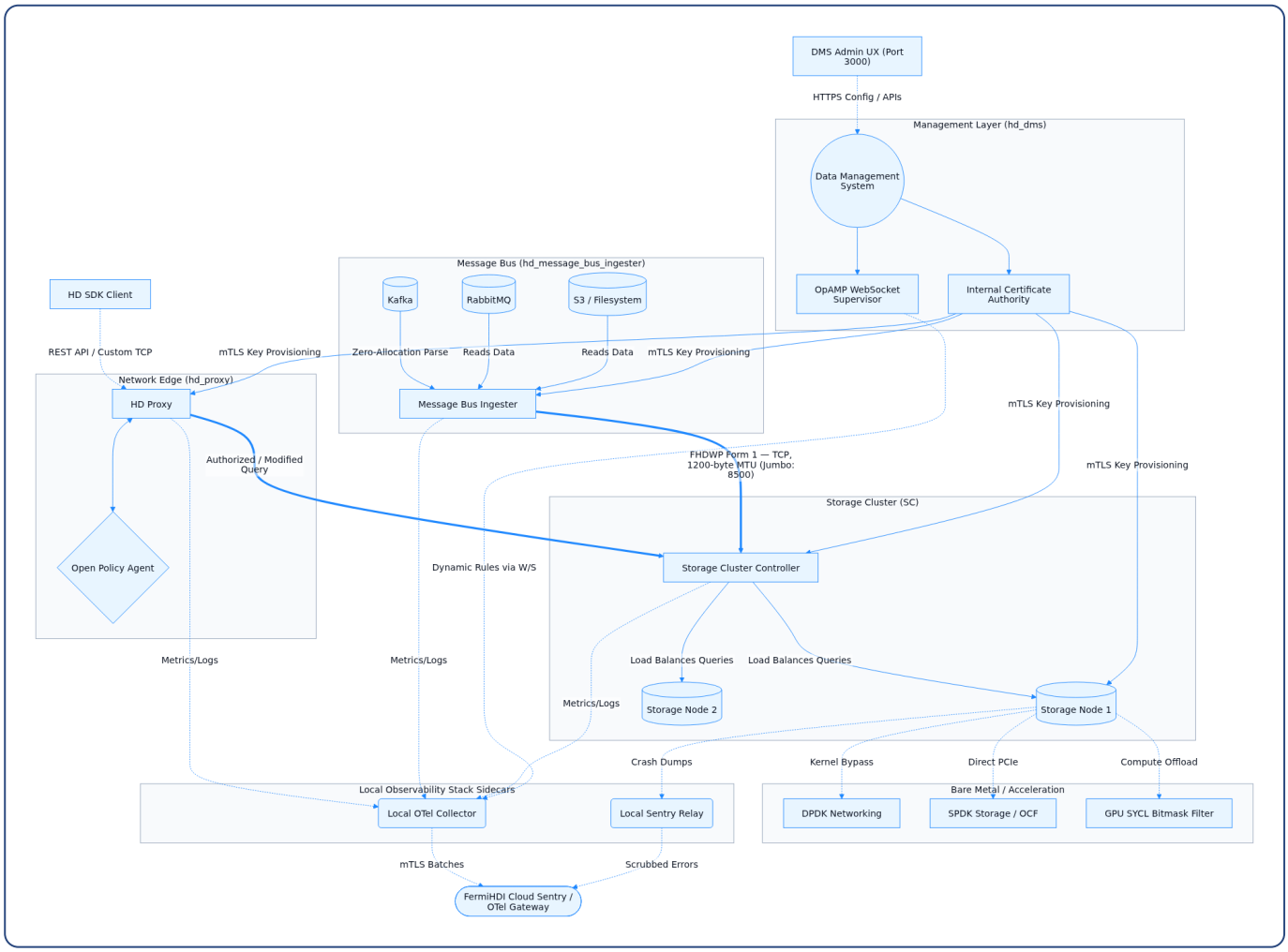
1.1.2 Core Components

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.
- **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 Ingester, 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.
- **Message Bus Ingester:** A high-performance C++ service that connects to data sources like Kafka, RabbitMQ, or S3, performing zero-copy parsing and converting data into the binary FHDWP format for ingestion.
- **HD SDK:** Client-side libraries (Python, Go, Rust, etc.) 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.

© FermiHDI Limited

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
`install/` scripts	preparing hosts: hugepages, VFIO, isolcpus, IOMMU	`install/installer.md`
`install/ansible/`	deploying the DMS and every node container across hosts	`install/ansible.md`
`install/helm/fermihdi`	the same deployment on Kubernetes	`install/helm.md`
`install/fermihdi-talos-*`	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
```


© FermiHDI Limited

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_CPUSET=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_CPUSET` 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
```


© FermiHDI Limited

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	`amd_iommu=on iommu=pt`
Intel	`intel_iommu=on iommu=pt`

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/fermihdi-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/fermihdi-prepare cluster.yaml            # dry run
sudo install/fermihdi-prepare cluster.yaml --apply
install/fermihdi-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
```


© FermiHDI Limited

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
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).

[!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",
```


© FermiHDI Limited

```
  "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 with a JWT to an ingress server (e.g., Traefik).

b. The ingress appends user identity headers (`X-Forwarded-Role`, `X-Forwarded-Email`).

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

© FermiHDI Limited

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

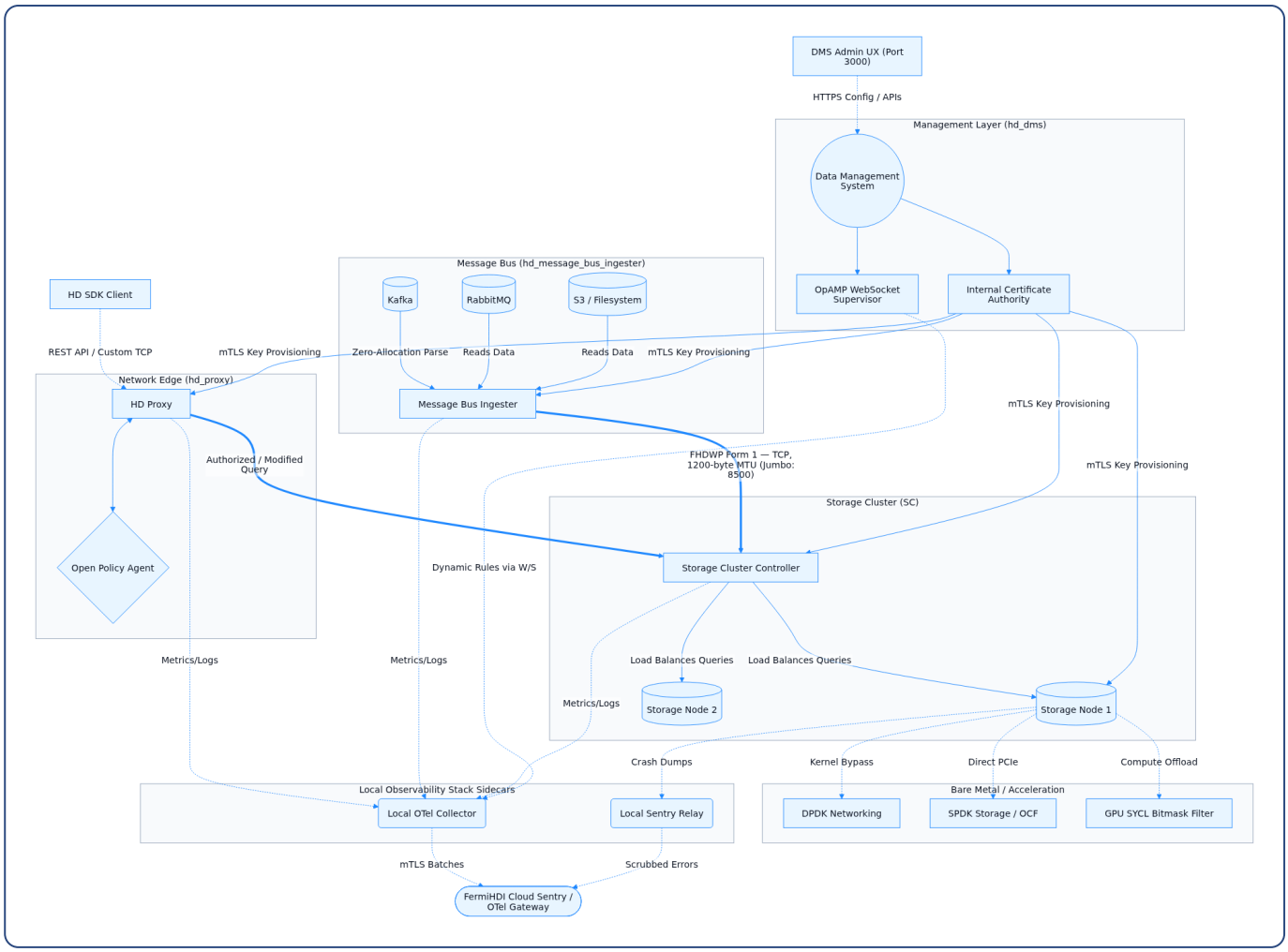
- **Observability:** All nodes expose Prometheus metrics and health endpoints on port **8080**. The system is designed to integrate with a central OpenTelemetry (OTel) Collector and Sentry Relay for logs, traces, and metrics.
- **SDK Resource Management:** The HD SDK uses asynchronous C++ reactors. To 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.

© FermiHDI Limited

2. 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.

2.1 Holistic System Topology



2.2 Component Architecture Breakdown

2.2.1 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 cluster data paths. - **Config & Schema Engine**: Synchronizes JSON schemas, Open Policy Agent (OPA) governance models, and dynamic telemetry instructions.

2.2.2 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**: Direct target for `hd_sdk` connections requesting data. - **OPA Governance (PBAC)**: The proxy evaluates

© FermiHDI Limited

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.

2.2.3 Message Bus Ingester `hd_message_bus_ingester`

A lightning-fast C++20 edge microservice performing ETL/Ingestion zero-copy translations. - **Zero-Allocation Parsing**: Uses `Glaze` C++ to interpret massive JSON/CSV arrays without dynamic memory mappings. - **Event Slicing**: Chops continuous streams from Kafka/RabbitMQ into strict deterministic 1200-byte pieces matching SPDK/DPDK requirements. - **Binary Conversion**: Translates verbose String formats into dense internal `RowBinary` arrays.

2.2.4 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.

2.2.5 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.

2.2.6 Software Development Kit (hd_sdk)

- **Local Application Calculus**: The SDK allows data engineers to run complex logic (SQL via embedded DuckDB, Vector evaluations via USearch, Graph analysis via cuGraph) directly inside the receiving local environment without requesting the centralized Storage cluster to parse string semantics. Instead, `fermihdi_client` maps natively zero-copied DPDK buffer results securely.

2.3 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.

2.3.1 IOMMU

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

CPU Vendor	GRUB parameter
AMD	`amd_iommu=on iommu=pt`
Intel	`intel_iommu=on iommu=pt`

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

© FermiHDI Limited

Verify after reboot:

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

2.3.2 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	`/dev/hugepages`	DPDK EAL — HD engine transport (networking)	1024 pages = 2 GiB
1 GB	`/dev/hugepages1G`	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
```

2.3.3 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_CPUSET`.

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_CPUSET=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	`cpuset:` in Docker Compose / K8s pod spec	Container runtime shim, control-plane threads, non-DPDK processes
DPDK cores	`MASTER_CPUSET` environment variable	DPDK/SPDK poll threads, self-pinned via `pthread_setaffinity_np` at SN startup

© FermiHDI Limited

The DPDK cores must **never** appear in the container `cpuset` field. The SN process self-pins its DPDK/SPDK threads to the `MASTER_CPUSET` 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_CPUSET: "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_CPUSET: "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, Ingester 4) are the totals `fermihdi-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_CPUSET`)	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 `swapper` on the DPDK core are benign. On hosts with no free physical core, `cpuset == MASTER_CPUSET` 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 `tests/sandbox_dms`)	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_CPUSET`: 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.

© FermiHDI Limited

2.3.4 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, `vfio-pci` bindings and the kernel command line. It reports before it changes anything, is idempotent, and never reboots.

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

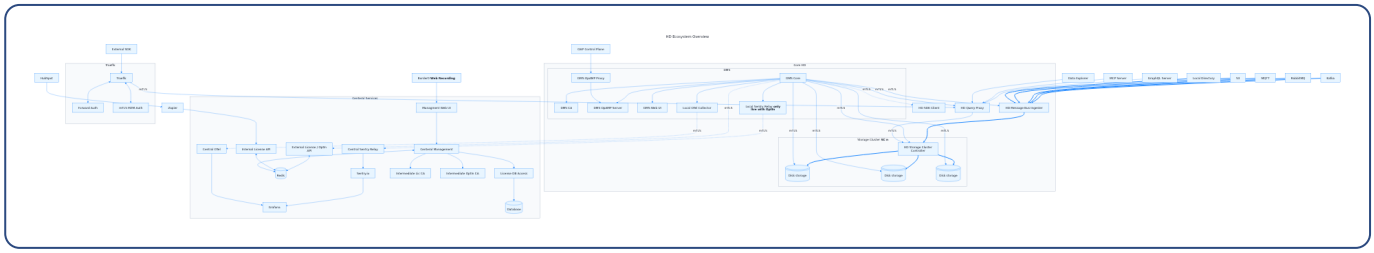
After making GRUB changes: `sudo update-grub && sudo reboot`

2.4 Further Reading

Topic	Document
FHDWP MTU, Jumbo Frame support, Docker/K8s CNI compatibility	docs/fhdwp_network_guide.md
SN configuration and storage provisioning	`hd_sn/docs/configuration_guide.md`
SN user guide and hardware requirements	`hd_sn/docs/user_guide.md`
End-to-end integration test guide	docs/end_to_end_test.md
Performance benchmark strategy	`BENCHMARKING.md`
Host setup script	`scripts/host_setup.sh`

© FermiHDI Limited

3. HD Ecosystem Overview



© FermiHDI Limited

4. Installation

4.1 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 v2.0.3**. Every image it names is `cr.fermihdi.io/hd/<service>:2.0.3`.

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.

4.1.1 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.

© FermiHDI Limited

4.1.2 What it is made of

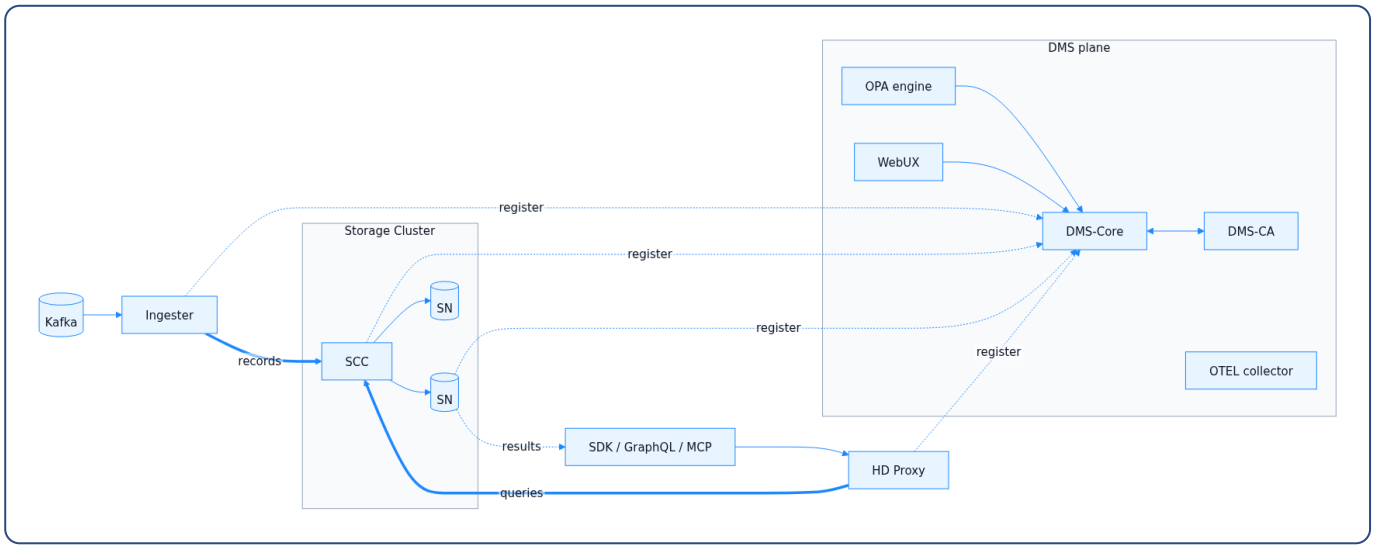
Component	Language	Does
DMS-Core	Go	The authority. Registers nodes, issues their configuration, holds the topology, the schema and the license
DMS-CA	Go	The certificate authority. Signs every identity in the deployment
DMS-OPA-Engine	Go	Compiles and serves the policy bundles the Proxy enforces
DMS-OPA-Adapter	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
Ingester	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	C++	GraphQL and MCP front ends onto the Proxy
Data explorer	—	Optional web UI over the Proxy
OTEL collector	—	Where every service sends traces and metrics

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

- **Engine roles** — SCC, SN, Ingester, 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.

© FermiHDI Limited

4.1.3 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:

1. **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.
2. **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.
3. **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.

4.1.4 Identity, trust and licensing

Nothing in an HD deployment talks to anything else without mutual TLS, and every certificate in it descends from one chain:

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


© FermiHDI Limited

```
            ├── dms-core, dms-webux, the OPA engine
            └── every SCC, SN, 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.

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.

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.

4.1.5 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
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.

© FermiHDI Limited

4.1.6 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
Ingester	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 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.

© FermiHDI Limited

A front door

The Proxy trusts `X-Forwarded-User`, `-Email`, `-Role` and `-Groups`, and does not authenticate callers itself: policy is evaluated against those headers. Something in front of it has to set them — your SSO through a reverse proxy's forward-auth, or `x509-to-http-headers` for mTLS service accounts. The same is true of the DMS WebUX. **Neither the Ansible layer nor the chart deploys that front door**; `docs/examples/ansible_example` is the reference for one. Until it exists, keep both on a loopback or a management network, which is where the installer publishes them by default.

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

4.1.7 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.

```yaml
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:
```


© FermiHDI Limited

```yaml
  tag: "2.0.3"
  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 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
```

Two 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.

Full field reference: `schema/cluster.schema.json`, which is what the validator checks against.

4.1.8 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.

```bash
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 2.0.3
```

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.

© FermiHDI Limited

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
`configure_sandbox.py`	`fermihdi-discover` + `fermihdi-configure`	NUMA-aware core allocation, storage device mapping, GRUB advice
`sandbox_host_setup.sh`	`fermihdi-prepare`	hugepages, hugetlbfs mounts, `vfio-pci` binding
`docker-compose.override.yml`	`cluster.yaml`	the placement decisions, written down
`compose.yml`	`ansible/` and `helm/fermihdi`	actually running the containers
`generate_certs_and_license.sh`	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
`mock-forward-auth` accepts everything and returns `admin`	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`.

4.1.9 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.

© FermiHDI Limited

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
`prepare.yml`	validates the file, then `fermihdi-prepare --apply` on every host
`verify.yml`	`fermihdi-prepare --verify` everywhere; changes nothing
`dms.yml`	the DMS plane, on the host whose block declares a `dms` role
`deploy.yml`	the node containers each host's block calls for
`site.yml`	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 Ingester 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
```


© FermiHDI Limited

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
`-o PATH`	write the values file
`--secrets PATH`	also write a 0600 values file holding keys and tokens
`--manifests`	render the chart and print the manifests instead
`--chart PATH`	render a chart other than the bundled one
`--domain`, `--storage-class`	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.

Every script, every option

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

`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.

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.yml` refuses to start with any required one missing, and names every one it could not find.

© FermiHDI Limited

4.1.10 After the install

Check it came up. `ansible-playbook verify.yml` 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.yml --limit new-host`, then `deploy.yml --limit new-host`. Nothing else needs to know: the node registers itself.

Upgrade. Change `images.tag` in `cluster.yaml` and re-run `deploy.yml`. 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
`/opt/fermihdi/dms/dms-core/storage`	the node registry, deployments, the license
`/opt/fermihdi/dms/dms-ca/storage`	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.

4.1.11 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 `dms.url`
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 `FERMIHDI_DMS_ENCRYPTION_KEY`
A reactor fails to reserve memory	Hugepages missing — check `verify.yml`, then the kernel command line after any upgrade
Throughput collapses under load, no errors	A poll core inside `container_cpuset`: the runtime shim is preempting it
SPDK fails at attach, looking like a driver fault	The container was handed the wrong `/dev/vfio` node — IOMMU groups move under a kernel upgrade
Services healthy, no traces anywhere	`OTEL_EXPORTER_OTLP_ENDPOINT` unset. `FERMIHDI_OTEL_URL` 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 `IPC_LOCK`: a container's default 64 KB memlock stops it pinning its arena

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

© FermiHDI Limited

4.1.12 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
`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/fhdwp_network_guide.md`	the wire protocol
`../tests/sandbox_dms/`	a complete working deployment to read

© FermiHDI Limited

4.2 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.

4.2.1 The pipeline

```
fermihdi-discover  →  fermihdi-configure  →  validate-cluster  →  fermihdi-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
`fermihdi-discover`	each host	`/sys`, `/proc`	facts, YAML or TSV	no
`fermihdi-configure`	each host	facts	a `cluster.yaml` host block	no
`validate-cluster`	anywhere	`cluster.yaml`	nothing	no
`fermihdi-prepare`	each host	`cluster.yaml`	the host itself	**only with `--apply`**
`fermihdi-k8s`	anywhere	`cluster.yaml`	Helm values, or manifests	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
`fermihdi-talos-discover`	anywhere	`fermihdi-discover`
`fermihdi-talos-prepare`	anywhere	`fermihdi-prepare`, and the GRUB half of it becomes an Image Factory schematic
`fermihdi-talos-k8s`	anywhere	`fermihdi-k8s`
`helm/fermihdi-talos`	—	`helm/fermihdi`, plus a VFIO device plugin

`fermihdi-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
`ansible/`	the DMS and every node container, across every host in the file
`helm/fermihdi`	the same deployment on Kubernetes, from values `fermihdi-k8s` renders

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

© FermiHDI Limited

`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.

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.

4.2.2 `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.

© FermiHDI Limited

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.

4.2.3 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 `/dev/hugepages` (2 MB) and `/dev/hugepages1G` (1 GB)	no
vfio	binds the listed PCI devices to `vfio-pci`	only if the IOMMU is off
grub	IOMMU, `isolcpus` / `rcu_nocbs` / `nohz_full`, and the 1 GB pool	yes

The two mount points are not interchangeable. `StorageEngine.cpp` pins SPDK's EAL `hugedir` to `/dev/hugepages` whenever it finds 1 GB pages, so that EAL never eats the 2 MB networking pool — which means `/dev/hugepages` has to *be* the 1 GB mount. That is also why `default_hugepagesz=1G` goes on the kernel command line whenever the 1 GB pool is in use: systemd mounts `/dev/hugepages` at whatever the default size is.

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
`/usr/local/sbin/fermihdi-hugepages` + its `.service`	2 MB pages are not reserved in GRUB, so something has to claim them on the way up. `sysctl` cannot: `vm.nr_hugepages` means the *default* page size, which is 1G here.
`/usr/local/sbin/fermihdi-vfio-bind` + its `.service`	`vfio-pci` binding lives in sysfs and does not survive a reboot.
`/etc/modules-load.d/fermihdi-vfio.conf`	loads `vfio-pci` before the bind unit runs.
`/etc/fstab` entry for `/dev/hugepages1G`	the 1 GB mount, which nothing else creates.

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.

© FermiHDI Limited

4.2.4 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
`host_setup.sh`	Bare-metal host preparation, one command at a time: `status`, `hugepages`, `bind <PCI>`, `unbind <PCI>`, `grub-advice`
`setup_scc_host.sh`	SCC-specific host preparation
`setup_sn_host.sh`	SN-specific host preparation, including NVMe/VFIO binding
`recommend_topology.sh`	Inspects `lscpu` and emits CPU pinning configuration
`prebuild_libs.sh`	Builds the `libfermihdi_*` libraries
`sync_*_submodules.sh`	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.

4.2.5 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.

4.2.6 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/2.0.3/install.sh`, or fetch `fermihdi-hd-install-2.0.3.tar.gz` and its `.sha256` directly.

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

© FermiHDI Limited

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/2.0.3/web/`	the site
`https://docs.fermihdi.io/hd/2.0.3/pdf/hd-administrator-manual.pdf`	one printable file
`https://docs.fermihdi.io/hd/2.0.3/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.

4.2.7 What is here

```
ADMIN_GUIDE.md        the guide: architecture, requirements, every option
README.md             this file
cluster.example.yaml  a worked cluster definition
schema/               the JSON Schema validate-cluster checks against
fermihdi-discover     inventory a host
fermihdi-configure    interview, and write its block
validate-cluster      check a definition against the schema and against reality
fermihdi-prepare      apply one host's host_prep
fermihdi-k8s          render Helm values from a definition
ansible/              deploy across every host in the definition
helm/fermihdi/        the Kubernetes chart

TALOS.md              installing onto Talos Linux: the map of that path
cluster.talos.example.yaml   a worked Talos cluster definition
fermihdi-talos-discover      inventory a Talos machine over the API
fermihdi-talos-prepare       machine config, schematic, apply, upgrade, verify
fermihdi-talos-k8s           render Helm values for the Talos chart
helm/fermihdi-talos/         the Talos chart: device plugin, PSA, both hugepage pools
```

4.2.8 Not written yet

- `lib/` — shared shell helpers, if the scripts here ever grow enough to need them.
- A front door. Neither layer deploys the reverse proxy and forward-auth service that sit in front of the Proxy and the WebUX and set the `X-Forwarded-*` headers both rely on. `docs/examples/ansible_example` deploys one; until you have it, both are published on the loopback address.

4.2.9 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 · Proxy · DMS
- `cpu_requirements_sn_scc.md` and `IMAGE_BUILD.md`.

© FermiHDI Limited

4.3 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
```

4.3.1 The playbooks

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

4.3.2 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_proxy:      ... and so on
```

Each host carries `fermihdi_instances` — every instance on it with the role type, `dpdk` 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.

© FermiHDI Limited

4.3.3 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=...

# ... 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.yml` 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
`FERMIHDI_CA_CERT_PEM`	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
`FERMIHDI_DMS_CA_CERT_PEM` / `_KEY_PEM`	the DMS-CA's ICA certificate and key. It presents this on the wire *and* signs every CSR in the deployment with it
`FERMIHDI_DMS_CORE_CERT_PEM` / `_KEY_PEM`	DMS-Core's TLS identity. The certificate is the leaf and its chain
`FERMIHDI_DMS_ICA_CHAIN_PEM`	the chain between the root and the DMS-CA, so a node verifying a certificate this deployment issued can build a path
`FERMIHDI_DMS_WEBUX_CERT_PEM` / `_KEY_PEM`	the WebUX's own server certificate
`FERMIHDI_DMS_WEBUX_FE_CERT_PEM` / `_KEY_PEM`	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
`FERMIHDI_DMS_ENCRYPTION_KEY`	what DMS-Core encrypts its stored state with. It exits at startup without one
`FERMIHDI_DMS_IDENTITY_KEY`	what the license payload is encrypted to. Has to be the one the license was issued against
`FERMIHDI_DMS_LICENSE_PAYLOAD`	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
`FERMIHDI_DMS_FE_TOKEN`	shared between DMS-Core and the WebUX — what the WebUX's back end presents on the internal management API
`FERMIHDI_BOOTSTRAP_TOKEN_DMS_OPA_ENGINE`	the OPA engine registers with DMS-Core the way a node does, so its token follows the same per-node convention, with `FERMIHDI_BOOTSTRAP_TOKEN` 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.

© FermiHDI Limited

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.

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

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

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.yaml`. 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.

© FermiHDI Limited

4.3.5 The DMS

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

Container	Is	Reached
`dms-ca`	signs every identity in the deployment	by DMS-Core, and by nothing else
`dms-core`	registration, node configuration, the control plane	by every node, on 6986
`dms-opa-engine`	the policy engine. Registers with DMS-Core the way a node does	internally
`dms-opa-adapter`	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
`dms-webux`	the web UI and its back end	through `dms-traefik`
`dms-traefik`	routes to the WebUX and applies the forward-auth middleware	on 8081, loopback by default
`otel-collector`	where every node in the cluster sends traces and metrics	on 4317 and 4318
`sentry-relay`	scrubs events before they leave the deployment. Only when a DSN is set	internally

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.yaml` 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.

© FermiHDI Limited

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.

4.3.6 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.

4.3.7 What `cluster.yaml` 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
`fermihdi_kafka_brokers` / `_topics`	the ingester's message bus. No schema field; the run warns rather than starting an ingester with nothing to read
`fermihdi_graphql_proxies_path`	scaffolded on first deploy and never overwritten, so an operator's proxy list survives the next run
`fermihdi_extra_env`	per-role core splits (`TRANSPORT_CORES`, `STORAGE_CORES`, `HASH_CORES`, `KV_CORES`, `FILTER_CORES`) and anything else. Merged last, so it wins
`fermihdi_dms_forward_auth_url`	what authenticates the WebUX. Nothing in the file says who your SSO is
`fermihdi_sentry_dsn`	the upstream a Sentry relay would forward to. Set it and a relay is deployed; leave it and none is
`fermihdi_dms_enable_analysis`	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.

© FermiHDI Limited

4.3.8 Requirements

- **Ansible ≥ 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.yaml`.
- 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`.

© FermiHDI Limited

4.4 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.

4.4.1 What it deploys

Namespace	Runs
`fermihdi-dms`	DMS-CA, DMS-Core, the OPA engine, the WebUX, the OTEL collector
`fermihdi-sc`	the SCC, and the Storage Nodes as a StatefulSet
`fermihdi-edge`	the Ingester and the HD Proxy
`fermihdi-extra`	hd_mcp, hd_graphql, 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:

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

4.4.2 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 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.

4.4.3 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.

© FermiHDI Limited

4.4.4 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 -
```

4.4.5 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.

4.4.6 Not deployed here

The public front door — the Traefik that terminates TLS for the Proxy and the WebUX, and the forward-auth service behind it — and `hd_ipflow_graphql`. `docs/examples/k8s_example` is no further ahead on either; they are genuinely not written yet.

© FermiHDI Limited

4.5 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.

4.5.1 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	`fermihdi-discover`, on the host	`fermihdi-talos-discover`, over the API
interview	`fermihdi-configure`	the same
check	`validate-cluster`	the same, plus `check_talos`
apply	`fermihdi-prepare`, on the host	`fermihdi-talos-prepare`, over the API
render	`fermihdi-k8s`	`fermihdi-talos-k8s`
chart	`helm/fermihdi`	`helm/fermihdi-talos`
deploy	`ansible/` or helm	helm 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.

4.5.2 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.

© FermiHDI Limited

4.5.3 Why a Talos install is two passes

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

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

4.5.4 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.

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

4.5.5 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.

© FermiHDI Limited

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=1G
    - 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.

4.5.6 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 Limited

`fermihdi-talos-k8s` builds the pools from the devices `cluster.yaml` already assigns, one per role and device kind:

```yaml
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_aio` 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.

4.5.7 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.

4.5.8 The two hugepage pools

Which pool is mounted where is not a style choice. `StorageEngine.cpp` scans for 1 GB pages and, when it finds any, pins SPDK's EAL `hugedir`:

```
SPDK: detected N x 1GB hugepages — EAL hugedir pinned to /dev/hugepages (1GB
pool). 2MB hugepages (networking pool) are excluded from SPDK.
```

So a component that uses 1 GB pages gets:

mount	pool
`/dev/hugepages`	`HugePages-1Gi`
`/dev/hugepages2M`	`HugePages-2Mi`

and a component with no 1 GB pages has one pool, on `/dev/hugepages`, which is where DPDK looks by default. These are the same two paths `fermihdi-prepare` creates on bare metal, for the same reason.

Getting it backwards does not fail loudly. SPDK quietly allocates its NVMe DMA buffers from the pool the network data path was sized for, and the symptom is a transmit stall under load a long way from the cause.

One arithmetic trap carries over from GRUB: `hugepages=N` on the kernel command line is a **per-node** figure that the kernel multiplies across NUMA nodes. A two-socket machine whose busy node needs 14 must reserve 28. `validate-cluster` enforces this.

4.5.9 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 Limited

`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.

4.5.10 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	`talosctl read /proc/cmdline`
2 MB pages per NUMA node	`talosctl read /sys/devices/system/node/nodeN/...`
every device on `vfio-pci`	`talosctl get pcidriverrebindstatus`
`vfio_pci` and `vfio_iommu_type1` loaded	`talosctl get loadedkernelmodules`

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.

4.5.11 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.

© FermiHDI Limited

4.5.12 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.

4.5.13 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

© FermiHDI Limited

5. Operations

5.1 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.

5.1.1 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.

5.1.2 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.

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.

© FermiHDI Limited

5.1.3 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
`FERMIHDI_DATA_RETURN_HOST`	*(discovery, then hostname)*	Address advertised to SNs. Set it when discovery cannot pick the right interface.
`FERMIHDI_DATA_RETURN_PORT`	`6987`	Port advertised to SNs, and the port the service listens on. Out-of-range or unparseable values fall back to the default silently.
`FERMIHDI_DATA_PLANE_SUBNET`	`10.150.0.0/24`	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.

5.1.4 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 Rotation**: The system utilizes short-lived mTLS tokens. Loss of the `DMS-CA` means the entire cluster must be rebooted to regenerate the trust chain.
- **Observability**: All nodes emit OpenTelemetry metrics and traces to a shared OTEL Collector. Node-level health and Prom HTTP endpoints are exposed on port 8080 by default.

© FermiHDI Limited

5.1.5 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_CPUSET` 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
`cpuset` (Docker Compose / K8s pod spec)	Overflow / management core(s) only
`MASTER_CPUSET` (env var)	All DPDK/SPDK-pinned cores — excluded from `cpuset`

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 (`MASTER_CPUSET`)	Overflow Cores (`cpuset`)	Isolation requirement
Storage Node (SN)	4	2	Overflow **must** be on a different physical core from DPDK cores
Storage Cluster Controller (SCC)	2	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

© FermiHDI Limited

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

Scenario	Logical CPUs
Sandbox / development	16 (8 physical cores, SMT×2)
Production minimum (clean overflow isolation)	20+
Production recommended (per-SN NUMA isolation)	32+ (dual NUMA socket)

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_CPUSET`. 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.

5.1.6 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.

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`.

© FermiHDI Limited

5.2 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.

5.2.1 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.

5.2.2 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.
```


© FermiHDI Limited

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

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`.

5.2.3 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.

5.2.4 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)`.

© FermiHDI Limited

5.3 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.

5.3.1 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.

5.3.2 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.

© FermiHDI Limited

5.3.3 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 `/etc/docker/daemon.json`. Host NIC must also be set: `ip link set <nic> mtu 9000`. 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 `mtu` in the `FelixConfiguration` 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.**

5.3.4 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.

© FermiHDI Limited

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.

5.3.5 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`

© FermiHDI Limited

5.4 Global User Guide

This page is a signpost rather than a guide; the material moved and is kept in three places.

To	Read
Install and run an instance	`install/ADMIN_GUIDE.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`.

© FermiHDI Limited

6. Development

6.1 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`.

6.1.1 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	`FermiHDISDK.hpp`, `libfermihdi_client.so`
Python	pybind11	`fermihdi_sdk`
Go	cgo	`fermihdi_sdk`
Rust	`cxx::bridge`	`fermihdi-sdk`
Node.js	N-API	`fermihdi-sdk`
R	Rcpp	`FermiHDISDK`
Julia	CxxWrap	`FermiHDI.jl`
Scala / Java	JNA	`fermihdi-sdk`
Mojo	C FFI	`FermiHDISDK`

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

6.1.2 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

© FermiHDI Limited

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.

6.1.3 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(fermihdi_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.

6.1.4 The same programme, in each language

Register, query, read the answer as a table.

C++

```
#include <FermiHDISDK.hpp>
```


© FermiHDI Limited

```
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)
```


© FermiHDI Limited

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)
```

6.1.5 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.

6.1.6 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.

© FermiHDI Limited

6.1.7 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 `X-Forwarded-User` — 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. `get_missing_batch_ids()` 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

6.1.8 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
`fhdwp_network_guide.md`	the wire protocol and its MTU
`../install/ADMIN_GUIDE.md`	standing up the instance to write against

© FermiHDI Limited

6.2 SDK

6.2.1 FermiHDI Software Development Kit (SDK)

The `hd_sdk` repository is the unified, high-performance gateway to HD. It abstracts the HD engine's kernel-bypass networking boundaries and the exact Form 1 & Form 2 TCP Protocol sequences into a polished set of developer primitives.

This single repository acts as a true Multi-Language Monorepo. We maintain a singular Zero-Copy C++ Engine natively wrapped in Python (`pybind11`), Go (`cgo`), Rust (`cxx`), Node.js (`napi`), R (`Rcpp`), Julia (`CxxWrap`), Scala/Java (`JNA`), and Mojo (`C-FFI`).

Key Features

1. **Zero-Copy Memory**: Inherently relies on Apache Arrow IPC strings and DuckDB relations, removing memory copying bottlenecks inside your application loops.
2. **Dynamic Service Discovery**: Built to implicitly navigate FermiHDI network topologies via the `DMS` and `Proxy` endpoints, securing X.509 sessions dynamically via Day-0 mTLS bootstrapping.
3. **Analytical Bridge**: Native integration with **DuckDB** for complex SQL post-processing and **USearch** for high-density HNSW vector similarity searches.
4. **Cloud-Native Observability**: Standardized OpenTelemetry JSON logging (v1.1.0), Prometheus metrics, and high-precision health reporting.
5. **12-Factor Configuration**: Externalized configuration with strict precedence and secure secrets injection (`_FILE` pattern).

Quick Start (Python)

```
pip install fermihdi-sdk
```

```
import fermihdi_sdk
import pyarrow as pa

# 1. Instantiate the Form 1 Ingester
ingester = fermihdi_sdk.IngestClient()
ingester.discover_scc_targets("https://proxy.fermihdi.io:6988")

# 2. Fire Arrow bytes purely over TCP (Zero TLS Overhead via mTLS) to the SCC Cluster
df = pa.Table.from_arrays([[1...10000000]], names=["event_id"])
ingester.stream_insert(batch=df, broadcast=True)

# 3. Bad Batch Notice (BBN) for error signaling
# ingester.send_bbn(batch_id)
```

Available Sub-Engines

1. **IngestClient**: Exclusively designed for pushing data into the cluster (Write Path) instantly using absolute Zero-Copy bytes over `libfermihdi_transport` . Supports EOB (End of Batch) and BBN (Bad Batch Notice) signaling.
2. **QueryClient**: Combines standard SQL Dataframes inherently linked to dynamic embedded **DuckDB** engines, Hardware-Accelerated **Graph processing (cuGraph/OpenMP)**, and **USearch HNSW Vectors (cuVS)**!

Which bindings the build produces

The top-level `CMakeLists.txt` wires up `python/` **only**, and behind `BUILD_PYTHON_BINDINGS` , which defaults to `OFF` . Every other binding (`go/` , `rust/` , `js/` , `r/` , `julia/` , `scala/` , `mojo/`) is source in this repository that the main build does not compile and CI does not exercise -- each has to be built from its own directory with its own toolchain.

Treat an unbuilt binding as unverified against the current C++ engine.

Documentation

- Getting Started — connecting, wiring up and initialising

© FermiHDI Limited

- API Reference — every public type and method
- Architectural Guides
- Configuration Guide (12-Factor)
- Interface Binding Contract
- User Manuals
- Test & Coverage Guide

The SDK is a client library: it calls endpoints and moves data over FHDWP, and serves none of its own. There is no OpenAPI specification for it, and the `docs/openapi.yaml` that used to be listed here described endpoints that did not exist. What the SDK *calls* is documented by the services that answer it — the DMS southbound and Proxy APIs.

Hardware Compatibility: AVX-512 and Windows Hyper-V

To address known `SIGILL` (Illegal Instruction) crashes on AMD Zen 4 and modern processors running within Windows 11 Hyper-V / Docker Desktop architectures:

The Issue: Microsoft Hyper-V has a long-standing kernel virtualization bug where it notoriously fails to context-switch AVX-512 (ZMM) registers correctly on certain modern processors. However, the hypervisor **still falsely reports AVX-512 support via its CPUID broadcast** to the guest containers. Standard C++ compilers, pre-compiled wheels, and dynamic analytical dispatching engines (like OpenMP, SYCL, DuckDB, PyArrow) will trust this CPUID flag, immediately attempt to vectorize high-speed matrices, and flawlessly crash their own containers via illegal instruction traps (Exit Code: -4).

The Solution: Use the built-in `DISABLE_AVX512` toggle to structurally override vectorization:

```
export DISABLE_AVX512=1
```

Activation safely guarantees stability on Docker Desktop while retaining full AVX2 functionality.

© FermiHDI Limited

6.2.2 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
`fermihdi::sdk::FermiHDIIngestClient`	write	Encodes records and streams them to Storage Cluster Controllers over FHDWP.
`fermihdi::sdk::FermiHDIQueryClient`	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.

© FermiHDI Limited

Two environment variables *are* read, and only to select the network stack:

Variable	Effect
`FERMIHDI_NETWORK_STACK`	Names the stack the engine should use. Defaults to `posix`.
`FERMIHDI_FORCE_POSIX`	`1` or `true` selects the POSIX stack.

Passing an empty `proxy_url` to `FermiHDIQueryClient` selects **direct SCC mode**, where you supply targets yourself instead of discovering them.

What has to be reachable

Direction	Peer	Purpose
Outbound HTTPS	DMS	`POST /v1/register` — certificates and topology
Outbound HTTPS	Proxy	`GET /v1/client/info` — SCC target discovery
Outbound HTTPS	Proxy	`GET /v1/hd/schema` — dataset schema
Outbound HTTPS	SCC	`POST /v1/query/stop` — 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);
```


© FermiHDI Limited

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
auto table = client.execute_dataset(query_json);
```

`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
```


© FermiHDI Limited

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
`stream_insert` / `send_eob` / `send_bbn` return `false`	No SCC targets. Call `discover_scc_targets()` or `set_scc_targets()`.
`stream_insert` returns `false` with targets set	Record was empty.
`convert_record_to_binary` returns an empty buffer	No schema loaded, or the schema declares no record length.
`discover_scc_targets` returns `false`	Proxy unreachable, response unparseable, or it named no targets.
`execute_dataset` returns `nullptr`	The schema could not be loaded, so the engine never booted.
`execute_dataset` 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 `had_errors()`.
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
`FermiHDIIngestClient::append_record_to_arrow()`	Logs and returns `true`. Nothing is appended.
`FermiHDIIngestClient::flush_arrow_batch()`	Always returns `nullptr`.
`FermiHDIQueryClient::set_local_aggregations()`	Logs the configuration and ignores it.
`FermiHDIQueryClient::execute_graph()`	Runs the query, then logs which graph path it would take. Produces no result and has no way to return one.

© FermiHDI Limited

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.

© FermiHDI Limited

6.2.3 HD SDK — API Reference

Every public type and method in `<FermiHDISDK.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
`verify_hash`	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.
`query_qos`	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.

© FermiHDI Limited

`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);
```


© FermiHDI Limited

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
`generate_batch_id_string`	Up to 64 bytes of the string, right-aligned with zero padding on the left. Longer strings keep their first 64 bytes.
`generate_batch_id_number`	The value big-endian in the final 8 bytes. Deterministic — the same number always yields the same ID.
`generate_batch_id_time`	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.

© FermiHDI Limited

`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.

© FermiHDI Limited

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.

`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_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_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.

© FermiHDI Limited

`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
`FermiHDIIngestClient::append_record_to_arrow()`	Logs and returns `true` unconditionally. Nothing is appended, and nothing accumulates for `flush_arrow_batch()` to return.
`FermiHDIIngestClient::flush_arrow_batch()`	Returns `nullptr` unconditionally.
`FermiHDIQueryClient::set_local_aggregations()`	Logs the configuration string and discards it. No aggregation is applied.
`FermiHDIQueryClient::execute_graph()`	Executes the query, then logs which graph implementation it would use. Returns `void` 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.

© FermiHDI Limited

6.2.4 FermiHDI SDK Architecture Map

Architectural Design

THE CORE MONOREPO CONCEPT

FermiHDI targets ultra-low latencies. If every single language wrapper were to natively implement the Form 1 wire protocol locally in Go, Python, and Javascript, it would heavily limit the system's ability to maintain a strictly controlled engine memory pool. It would cause exponential tech debt over separate repositories whenever a single Wire format changes.

Instead, the `hd_sdk` is an inverted dependency graph: 1. The HD engine's transport layer: handles the networking bytes, the TCP sequences, and the event loops that map onto kernel-bypass interfaces. 2. `fermihdi_client` (C++): Maps network functions into explicit `execute_stream`, `execute_graph`, and `execute_vector_search` boundaries, wrapping types transparently and safely. 3. `Foreign Language Wrappers`: Pass native pointers and strings smoothly through explicit FFI interfaces directly into `fermihdi_client`.

THE NETWORKING FLOW

When building applications: * **The Control Network (TLS/mTLS)**: Is contacted *once* leveraging `register_with_dms` to efficiently retrieve the `.SCRoutingTable`, identifying which specific Database Node IPs the client should target automatically. * **The Production Network (TCP/NORM)**: The actual heavy lifting operations (pulling billions of `RecordBatches` sequentially) bypass TLS encryption, explicitly mapping native Arrow frames dynamically over the engine's kernel-bypass channels.

Using the C++ Backend natively

If you are developing a C++ application that requires maximum native integration:

```cpp
#include <fermihdi/sdk/FermiHDISDK.hpp>
#include <arrow/api.h>

using namespace fermihdi::sdk;

int main() {
    FermiHDIQueryClient db("https://proxy.fermihdi.io");

    // Explicitly Register against Control Plane natively
    db.register_with_dms("https://dms.fermihdi.io", "my_node_1");

    // Evaluate Data using USearch Native AVX-512 integration directly
    std::vector<float> embed = {1.0, 0.5, 0.4};
    auto similar = db.execute_vector_search("SELECT text FROM embeddings", "vector", embed, 5);

    return 0;
}
```

Hardware Acceleration & Embedded Engines

The `fermihdi_client` natively encapsulates advanced analytics directly into the returning Arrow IPC buffers: - **DuckDB**: Instead of looping across arrays via generic strings, the Dataframe explicitly wraps an embedded `duckdb::DuckDB db` zero-copy memory pointer executing SQL aggregations instantaneously. - **Graph Mathematics (OpenMP / cuGraph)**: Bypasses NetworkX completely. It instantiates Native PageRank/SSSP directly alongside `arrow::Int64Array` relationships across Nvidia CUDA cores or Multi-Core Intel SIMD architectures. - **USearch (HNSW Vectors)**: Binds the raw vectors directly into `usearch/index_dense.hpp` executing Cosine distances via AVX-512 instantly on return.

Language Matrix Integration Status

- **Python**: Available via `pybind11`. Used for PyArrow/Pandas seamless zero-copy integration.
- **Go**: Available via `cgo`. Best for Kubernetes operators/microservices and concurrent Data Engineering.
- **Rust**: Bound strictly using safe `cxx::bridge`. Ideal for building hyper-secure data endpoints.
- **Node.js**: Available via `napi`. Directly pipes Web Gateway events (Express/Bun) to the SCC natively.
- **R (Rcpp) & Scala (JNA)**: Evaluates statistics directly bypassing JVM limits or R memory boundaries completely.
- **Mojo & Julia**: Hooks Form 1 strings natively, allowing state-of-the-art Generative AI processing limits.

© FermiHDI Limited

6.2.5 FermiHDI SDK Configuration Guide

The FermiHDI SDK follows the **Cloud-Native Configuration Standard (12-Factor App)**. Configuration is derived from multiple sources with a strict precedence order.

Configuration Precedence

1. **Direct API calls**: Manual overrides in C++, Python, or Go code.
2. **Environment Variables**: The primary method for runtime configuration.
3. **DMS (Data Management System)**: Dynamic settings retrieved after mTLS enrollment.
4. **Local Defaults**: Hardcoded safe defaults within the SDK.

Core Configuration Options

The SDK supports the **_FILE** pattern for all secret-sensitive variables. For any variable `X`, providing `X_FILE` with a valid path will read the secret from that file.

Option	Environment Variable	Secret Companion	Default Value	Description
Network Stack	`FERMIHDI_NETWORK_STACK`	N/A	`posix`	`posix` or `native` (kernel-bypass).
Force POSIX	`FERMIHDI_FORCE_POSIX`	N/A	`false`	If `1`, forces POSIX even where kernel-bypass is available.
Service Name	`SERVICE_NAME`	N/A	`hd_sdk_native`	Name reported in JSON logs.
Environment	`ENV`	N/A	`production`	Deployment environment (prod/staging).
Bootstrap Token	`FERMIHDI_BOOTSTRAP_TOKEN`	`_FILE`	(None)	Used for initial Day-0 PKI enrollment.
K8s Token	`K8S_SERVICEACCOUNT_TOKEN`	`_FILE`	(None)	K8s SA token for auto-enrollment.
Log Level	`LOG_LEVEL`	N/A	`info`	Minimum severity of logs.
OTEL Trace ID	`OTEL_TRACE_ID`	N/A	(None)	Injected into JSON log objects.

JSON Schema Validation

The SDK utilizes an internal JSON schema for validating dynamic configuration updates received from the Control Plane via the Proxy or DMS. The formal schema can be found in config_schema.json.

© FermiHDI Limited

6.2.6 HD SDK — Interface Binding Contract

This document defines the mandatory interface binding requirements for the HD SDK (`hd_sdk`). All consumers **must** supply explicit network interface names when initializing the SDK. The SDK itself **never reads environment variables** for interface names.

Background

HD separates network traffic across three dedicated planes:

Plane	Default Port	Protocol	Docker Compose Interface	K8s Interface Order
Data Plane	6987	FHDWP	`eth1` (hd_dataplane)	1st listed network
Control Plane	6986	HTTPS/mTLS	`eth0` (hd_controlplane)	2nd listed network
Observability	8080	HTTP	`eth2` (hd_observability)	3rd listed network

DOCKER COMPOSE INTERFACE ORDER

In Docker Compose, multiple networks attached to a container are assigned kernel interface names in **alphabetical order** by network name:

```
hd_controlplane  → eth0
hd_dataplane     → eth1
hd_observability → eth2
```

KUBERNETES INTERFACE ORDER

In Kubernetes with Multus (or similar CNI), interfaces are assigned in the order the networks are listed in the pod annotation. Follow the same data/control/observability order as above to guarantee `eth0` = control, `eth1` = data, `eth2` = observability.

C++ SDK Constructors

```cpp
// Ingest client — binds the FHDWP data plane transport to data_plane_interface,
// and the control plane to control_plane_interface.
// Pass empty strings to bind 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 as above.
explicit FermiHDIQueryClient(const std::string& proxy_url = "",
                             const std::string& data_plane_interface = "",
                             const std::string& control_plane_interface = "");
```

EXAMPLE (C++, DOCKER COMPOSE SANDBOX)

```cpp
// Docker Compose: hd_controlplane=eth0, hd_dataplane=eth1
auto ingest_client = std::make_unique<fermihdi::sdk::FermiHDIIngestClient>("eth1", "eth0");
auto query_client  = std::make_unique<fermihdi::sdk::FermiHDIQueryClient>("", "eth1", "eth0");
```

Go SDK

The Go CGO wrapper exposes two constructors:

```go
// Convenience wrapper that binds to all interfaces (0.0.0.0).
func NewQueryClient(proxyURL string) *QueryClient

// Full constructor with explicit interface names.
// dataPlaneIface: kernel interface name for FHDWP data receive port (e.g. "eth1").
// ctrlPlaneIface: kernel interface name for mTLS control plane (e.g. "eth0").
func NewQueryClientWithInterfaces(proxyURL, dataPlaneIface, ctrlPlaneIface string) *QueryClient
```

EXAMPLE (GO, USING ENV VARS SET BY DOCKER COMPOSE OR K8S)

```go
// The consumer service reads interface names from env vars (set by compose.yml / K8s manifests)
// and forwards them to the SDK. The SDK itself does not read env vars.
dataPlane := os.Getenv("FERMIHDI_DATA_PLANE_INTERFACE")    // e.g. "eth1"
```


© FermiHDI Limited

```
ctrlPlane := os.Getenv("FERMIHDI_CONTROL_PLANE_INTERFACE") // e.g. "eth0"

client := fermihdi.NewQueryClientWithInterfaces(proxyURL, dataPlane, ctrlPlane)
```

Environment Variables (Consumer Services Only)

Consumer services (`hd_graphql` , `hd_mcp` , `hd_data_explore` , `hd_ipflow_graphql`) **must** set the following environment variables in their Docker Compose `environment:` blocks or Kubernetes manifests:

Variable	Purpose	Example Value
`FERMIHDI_DATA_PLANE_INTERFACE`	Interface name for FHDWP data plane traffic	`eth1`
`FERMIHDI_CONTROL_PLANE_INTERFACE`	Interface name for mTLS control plane traffic	`eth0`
`FERMIHDI_OBSERVABILITY_INTERFACE`	Interface whose address is bound into the client certificate's SANs. The SDK serves no health or metrics endpoint of its own — see the API reference.	`eth2`

[!IMPORTANT] These env vars are consumed by the **application layer** (Go/Python wrappers), not by the C++ SDK directly. The application layer must read these values and pass them to the SDK constructor. Do **not** rely on the SDK to read env vars automatically.

[!WARNING] Failing to supply interface names results in the SDK binding to `0.0.0.0` (all interfaces). In a segmented network deployment, this causes the HD engine's reactor to receive its own internal `SIGUSR1` / `SIGUSR2` signals on the HTTP library's thread pool, leading to `SIGSEGV` . Always provide explicit interface names in production.

© FermiHDI Limited

6.3 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 `docs/developer_guide.md`
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`.

© FermiHDI Limited

6.4 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
`tests/integration_test_full_pipeline`	Kafka → Ingester → SCC → SN → query via Proxy → test head
`tests/integration_test_dms_pipeline`	registration, provisioning and the DMS side of that path
`tests/integration_test_sc_cluster`	a Storage Cluster on its own
`tests/sandbox_dms`	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.

© FermiHDI Limited

6.5 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
`tests/integration_test_full_pipeline`	Kafka → Ingester → SCC → SN → query via Proxy → test head
`tests/integration_test_dms_pipeline`	registration, provisioning and the DMS side of the same path
`tests/integration_test_sc_cluster`	a Storage Cluster on its own
`tests/sandbox_dms`	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.

6.5.1 Test Overview

1. **Environment Setup**: Tears down any lingering state and creates a dedicated Docker network (`hd_e2e_net`) on the `172.25.0.0/16` subnet.
2. **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.
3. **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.
4. **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.
5. **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.
6. **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.

6.5.2 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"`

© FermiHDI Limited

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`.
- *Note: Overrides the entrypoint 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 Ingester

- **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`

© FermiHDI Limited

- **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`

6.5.3 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.

6.5.4 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.

© FermiHDI Limited

7. FermiHDI 3rd Party Licensing Documentation

This document outlines the third-party software, applications, libraries, and source codes integrated into the FermiHDI environment. It details their license types, usage, criticality, restrictions, and explicit conditions concerning code distribution or open-sourcing.

7.1 Core C++ Dependencies (Data Plane & Operations)

7.1.1 DPDK (Data Plane Development Kit)

- **License**: BSD-3-Clause
- **Usage**: Utilized extensively in `hd_storage_node`, `hd_storage_controller`, `hd_message_bus_ingester`, and `libfermihdi_transport` for kernel bypass networking and zero-copy User Space memory isolation.
- **Criticality**: **High**. Crucial for the extreme throughput required by the FermiHDI interconnect.
- **Restrictions**: Requires distribution of the copyright notice and license text.
- **Distribution/Attribution**: Attribution is strictly required in all binaries and documentation.
- **Impact on Codebase**: Does **not** require FermiHDI code to be open-sourced. Fully compatible with closed-source commercialization.

7.1.2 SPDK (Storage Performance Development Kit)

- **License**: BSD-3-Clause
- **Usage**: Used directly in `libfermihdi_storage` to communicate directly with NVMe drives bypassing kernel interrupts.
- **Criticality**: **High**. Essential for millions of IOPS disk access on the Storage Nodes.
- **Restrictions**: None beyond standard BSD redistribution notices.
- **Distribution/Attribution**: Attribution required.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.1.3 Apache Arrow

- **License**: Apache License 2.0
- **Usage**: In-memory columnar data formatting during Storage Node filtering and query aggregation phases.
- **Criticality**: **High**. Critical for the analytic math throughput.
- **Restrictions**: Requires including the Apache 2.0 license, a NOTICE file (if they provide one), and stating any significant modifications made to Arrow code.
- **Distribution/Attribution**: Attribution and Apache 2.0 license presentation required.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.1.4 OpenSSL (v3.0+)

- **License**: Apache License 2.0
- **Usage**: Core telemetry encryption, mTLS bindings in the proxy, and internal cluster identity management.
- **Criticality**: **High**. FermiHDI cannot operate securely without it.
- **Restrictions**: Standard Apache 2.0 restrictions apply.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

© FermiHDI Limited

7.1.5 nlohmann/json

- **License**: MIT License
- **Usage**: Parsing Control Plane JSON schemas and configurations in C++ nodes.
- **Criticality**: **Low/Medium**. Replaceable, but extensively used.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.1.6 Sentry Native (`sentry-native`)

- **License**: MIT License
- **Usage**: Capturing C++ crash dumps and propagating OTel Trace IDs dynamically.
- **Criticality**: **Medium**.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.2 Go Dependencies (Control Plane)

7.2.1 OpenTelemetry Go (`go.opentelemetry.io/otel`) & OpAMP (`opamp-go`)

- **License**: Apache License 2.0
- **Usage**: Distributed tracing, metrics generation, and WebSocket dynamic configuration pushed from the DMS.
- **Criticality**: **Medium/High**. Required to fulfill enterprise telemetry compliance.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.2.2 Sentry Go (`getsentry/sentry-go`)

- **License**: MIT License
- **Usage**: Capturing Go panics in the Proxy and DMS instances.
- **Criticality**: **Medium**.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.2.3 JSONSchema Go (`github.com/xeipuuv/gojsonschema`)

- **License**: Apache License 2.0
- **Usage**: Validation of the strictly defined Node configurations mapped by the DMS.
- **Criticality**: **Medium**.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.3 Web UI Dependencies (Astro/Javascript)

7.3.1 Astro & Vue.js

- **License**: MIT License
- **Usage**: Powers the frontend routing, DOM generation, and reactivity of the Deployment Management System WebUI.
- **Criticality**: **Medium**. Strictly visual representations of the DMS API.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

© FermiHDI Limited

7.3.2 Chart.js & Vue-ChartJS

- **License**: MIT License
- **Usage**: Analytics dashboards inside the DMS UI.
- **Impact on Codebase**: Does **not** force FermiHDI to be open source.

7.3.3 Playwright

- **License**: Apache License 2.0
- **Usage**: Executed only in CI environments and testing suites simulating E2E telemetry visually.
- **Criticality**: **Low**. Dev/Test dependency only.

7.4 Summary Conclusion

There are **zero "copyleft" style licenses (like GPL, AGPL, or CDDL)** compiled into the primary distribution paths of HD. All third-party libraries have been vetted as commercially permissive (MIT, BSD, Apache 2). The system natively allows you to commercialize, distribute, and protect the entirety of the closed-source codebase safely provided you include an `ATTRIBUTIONS.txt` file releasing the standard copyright notices required by the libraries above.

© FermiHDI Limited