MOLGENIS EMX2 & HPC Job Orchestration

Contents

1 TL;DR — Why This Exists

EMX2 is evolving to support heavy compute workloads (including AI) that cannot run inside the application stack and cannot be triggered via inbound connections into the available HPC clusters. At the same time, EMX2 must remain the authoritative system of record for job state and artifact metadata, while HPC governance must retain control over scheduling and resource allocation. This creates a coordination problem across a strict trust boundary.

This design introduces an outbound-only execution bridge: the HPC head node polls EMX2 for work, claims jobs atomically, submits them to Slurm, and reports lifecycle transitions and artifacts back to EMX2. EMX2 owns state and integrity; HPC owns execution and scheduling. The result is deterministic, auditable job orchestration without breaking institutional network constraints or governance boundaries.

2 Overview

Molgenis EMX2 is a metadata-driven platform for scientific data built around FAIR principles (findability, accessibility, interoperability and reusability). As the platform evolves to incorporate AI-backed enhancements (like automated annotation, similarity search, inference pipelines) it needs the ability to offload compute-intensive workloads to GPU-enabled infrastructure that typically lives outside the application's own network.

This document proposes a protocol for bridging EMX2 with one or more HPC clusters managed by Slurm. The design addresses a specific institutional constraint: the HPC environment cannot accept inbound connections. All communication must be initiated from the HPC side.

The result is an outbound-only job execution bridge. HPC workers poll EMX2 for work, claim jobs, execute them inside Apptainer containers, and report results — all without EMX2 needing to reach into the cluster.

2.1 Scope

The protocol covers worker registration and capability advertisement, job lifecycle management, artifact management (typed, content-addressed data objects for job inputs and outputs), and authentication across the trust boundary.

It does not cover job creation by end users (that is an EMX2 application concern), Slurm cluster administration, or artifact retention policy.

2.2 Terminology

Term	Meaning
Worker	A head node controller that registers with EMX2, polls for jobs, and submits them to Slurm.
Processor	A logical identifier for a type of workload (e.g. `text-embedding:v3`).
Profile	An abstract resource tier (e.g. `gpu-medium`) mapped to Slurm parameters on the HPC side.
Artifact	A typed, content-addressed data object tracked by the artifact registry.
Transition	A recorded state change on a job, created as a sub-resource of that job.

3 System Architecture

The system is divided into two trust domains connected by outbound HTTPS from the HPC environment. This section describes the components in each domain and how they interact.

3.1 EMX2 Application Domain

EMX2 exposes three API surfaces:

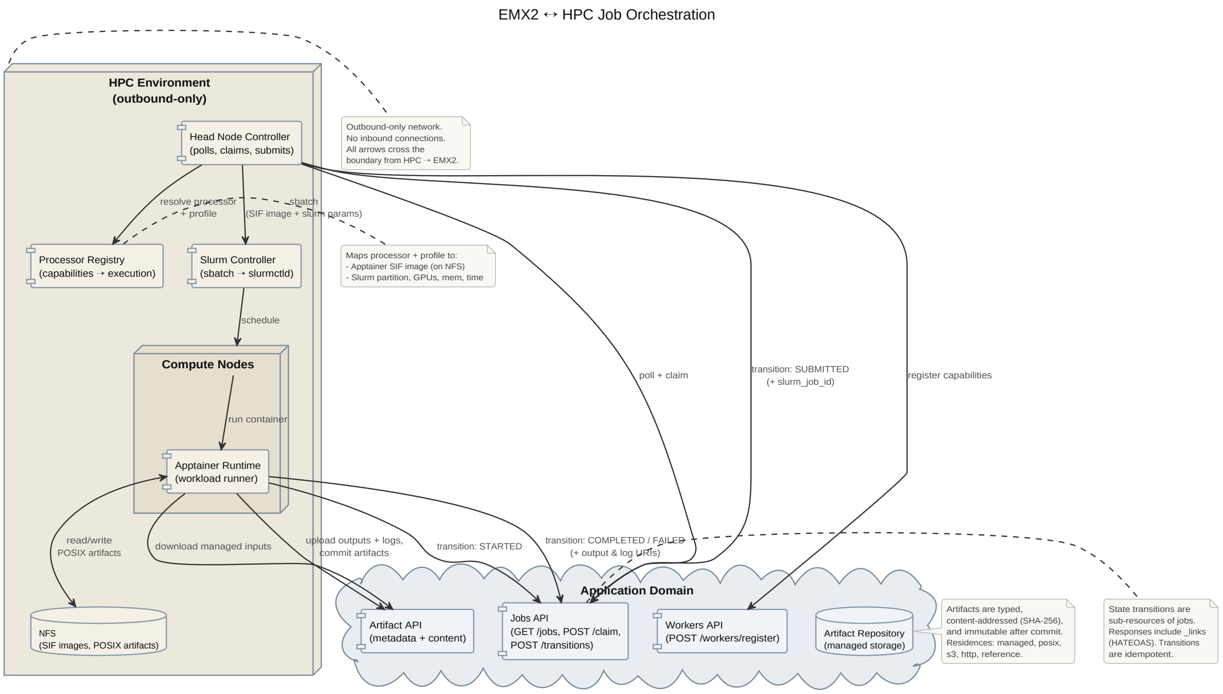


Figure 1: HPC Topology

- **Workers API** — registration of head nodes and their capabilities.
- **Jobs API** — job listing, filtering by capability, atomic claiming, state transitions, and output artifact linking.
- **Artifact API** — lifecycle management (create/upload/commit), path-based file operations (PUT/GET/HEAD/DELETE), paginated file listing, and integrity verification. Exposes an S3-minimal surface for managed artifacts.

These are backed by tables in the EMX2 `_SYSTEM_` schema (prefixed with `Hpc` to avoid collisions). The system tables hold job state (including `output_artifact_id` foreign key to artifacts), worker registrations, capability advertisements, transition audit logs, artifact metadata, and artifact file content (stored in EMX2 FILE columns for managed residence). A Vue-based HPC dashboard provides browser access to jobs, workers, and artifacts including direct file upload.

All endpoints live under `/api/hpc/*` with a shared before-handler that validates protocol headers and (when configured) HMAC authentication. The health endpoint (`/api/hpc/health`) is exempt from authentication.

3.2 HPC Environment

The HPC side consists of:

- **Head Node Controller** — a daemon that registers capabilities, polls for pending jobs, maps processor + profile to Slurm parameters, submits `sbatch`, and reports the result back to EMX2.
- **Slurm Controller** — the cluster's workload manager, unchanged from its standard role.
- **Apptainer Runtime** — executes the workload inside an Apptainer (formerly Singularity) container on a compute node. The daemon handles all communication with EMX2: staging input artifacts (symlink for posix, download for managed), monitoring Slurm job state, uploading or registering output

artifacts, and posting status transitions.

- **NFS Shared Storage —** an NFS export mounted on the head node and all compute nodes. Stores Apptainer SIF images, POSIX-resident artifacts, and shared scratch data.
- **Local Scratch —** per-node temporary storage, discarded after job completion.

3.3 Separation of Responsibilities

Concern	Owner
Job registry, lifecycle state, artifact metadata, managed file storage	EMX2
Capability registration, job claiming, Slurm submission, artifact staging	Head Node Controller
Workload execution	Apptainer Runtime
Scheduling, resource allocation, node dispatch	Slurm Controller
Managed artifact binary content (FILE columns)	EMX2 Database
Shared data between jobs, POSIX-resident artifacts	NFS

3.4 Head Node Daemon

The head node controller is implemented as a Python CLI (`emx2-hpc-daemon`) with four commands:

Command	Purpose
`run`	Start the daemon main loop (register → poll → claim → submit → monitor, repeating).
`once`	Run a single poll-claim-monitor cycle, then exit. Suitable for cron-based invocation.
`register`	Register the worker with EMX2 and exit.
`check`	Validate config, connectivity, and Slurm command availability.

Both `run` and `once` accept a `—simulate` flag that walks jobs through all lifecycle states without invoking Slurm or creating working directories. In simulate mode, each poll cycle advances tracked jobs one step (CLAIMED → SUBMITTED → STARTED → COMPLETED), completing a full lifecycle in approximately three cycles.

The daemon sends periodic heartbeats (default: every 120 seconds) to keep the worker registration alive. On startup, it recovers tracking state for non-terminal jobs from a previous run. On SIGTERM/SIGINT, it stops accepting new work and exits gracefully; Slurm jobs continue running independently and are recovered on next startup.

Configuration is via a YAML file specifying EMX2 connection details, Slurm parameters, Apptainer settings, and profile-to-resource mappings.

4 End-to-End Protocol

The happy-path sequence proceeds in four phases.

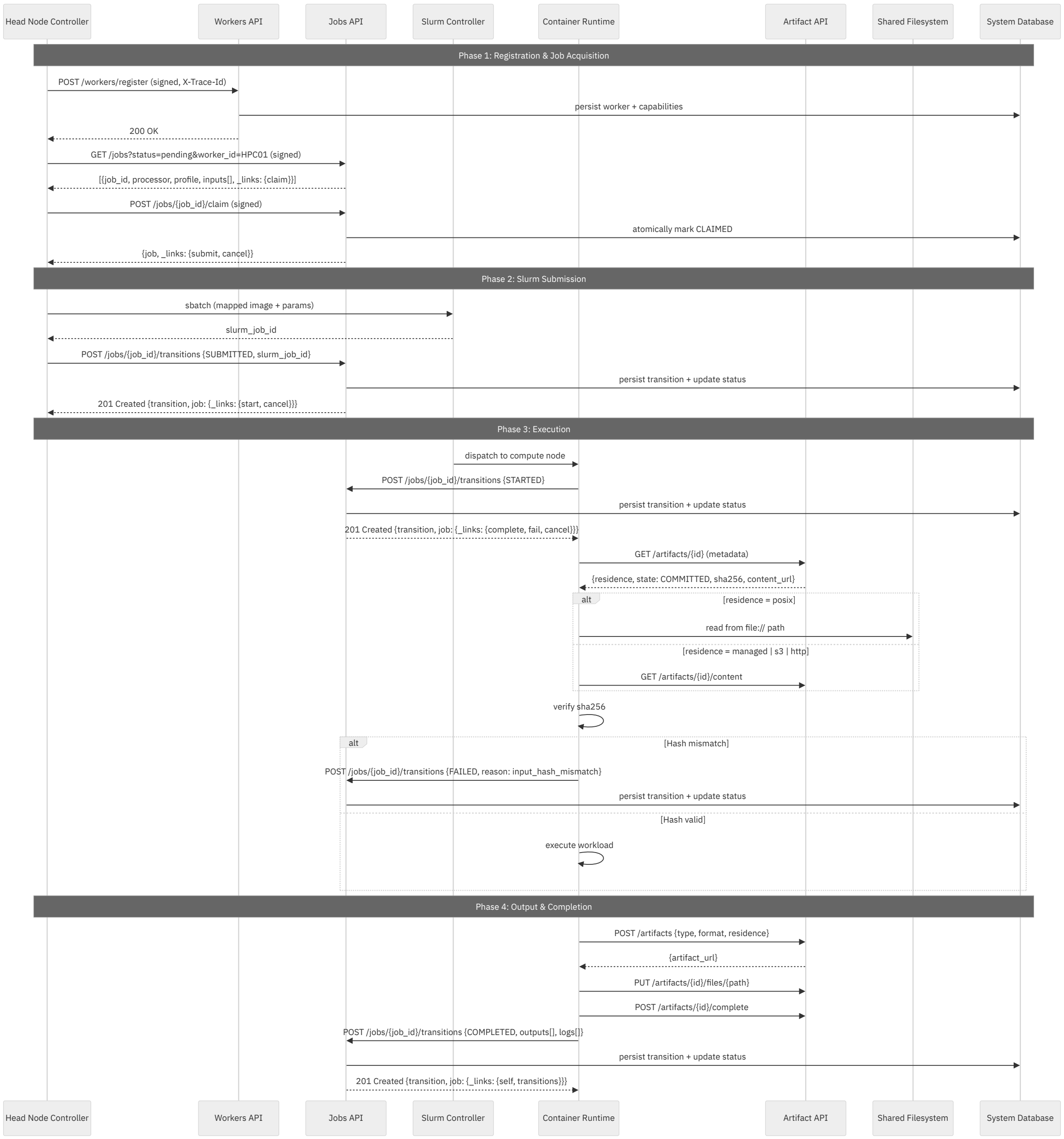

Head Node Controller
Workers API
Jobs API
Slurm Controller
Container Runtime
Artifact API
Shared Filesystem
System Database
Phase 1: Registration & Job Acquisition
POST /workers/register (signed, X-Trace-Id)
persist worker + capabilities
200 OK
GET /jobs?status=pending&worker_id=HPC01 (signed)
[{job_id, processor, profile, inputs[], _links: {claim}}]
POST /jobs/{job_id}/claim (signed)
atomically mark CLAIMED
{job, _links: {submit, cancel}}
Phase 2: Slurm Submission
sbatch (mapped image + params)
slurm_job_id
POST /jobs/{job_id}/transitions {SUBMITTED, slurm_job_id}
persist transition + update status
201 Created {transition, job: {_links: {start, cancel}}}
Phase 3: Execution
dispatch to compute node
POST /jobs/{job_id}/transitions {STARTED}
persist transition + update status
201 Created {transition, job: {_links: {complete, fail, cancel}}}
GET /artifacts/{id} (metadata)
{residence, state: COMMITTED, sha256, content_url}
alt
[residence = posix]
read from file:// path
[residence = managed | s3 | http]
GET /artifacts/{id}/content
verify sha256
alt
[Hash mismatch]
POST /jobs/{job_id}/transitions {FAILED, reason: input_hash_mismatch}
persist transition + update status
[Hash valid]
execute workload
Phase 4: Output & Completion
POST /artifacts {type, format, residence}
{artifact_url}
PUT /artifacts/{id}/files/{path}
POST /artifacts/{id}/complete
POST /jobs/{job_id}/transitions {COMPLETED, outputs[], logs[]}
persist transition + update status
201 Created {transition, job: {_links: {self, transitions}}}
Head Node Controller
Workers API
Jobs API
Slurm Controller
Container Runtime
Artifact API
Shared Filesystem
System Database

Figure 2: HPC Sequence

Phase 1 — Registration and job acquisition. The head node registers its capabilities with the Workers API, then polls the Jobs API for pending jobs that match its declared processors and profiles. When it finds one, it claims it. The claim is atomic: if two workers try to claim the same job, only one succeeds.

Phase 2 — Slurm submission. The head node maps the job's processor and profile to an Apptainer SIF image (stored on NFS) and a set of Slurm parameters, then submits via `sbatch`. It reports the Slurm job ID back to EMX2 as a SUBMITTED transition.

Phase 3 — Execution. Slurm dispatches the job to a compute node. The daemon monitors the job via `squeue`/`sacct` and posts a STARTED transition to EMX2 when execution begins. Before submission, the daemon stages input artifacts: for posix artifacts it symlinks the `file://` path into the job's input directory (zero-copy); for managed artifacts it downloads files via `GET /api/hpc/artifacts/{id}/files/{path}`. SHA-256 hashes are verified on access. If verification passes, the workload runs.

Phase 4 — Output and completion. The daemon creates an output artifact, uploads files (for managed residence) or registers the output directory path (for posix residence), and commits. It posts a COMPLETED transition with the `output_artifact_id` field linking the job to its output artifact. The artifact ID is stored as a foreign key on the job record, making outputs discoverable via GraphQL.

At every step, the client discovers what it can do next from hypermedia links in the response. If a transition is not legal in the current state, the corresponding link is absent. Failure at any point results in a FAILED transition with a reason code (see Job Lifecycle, below).

4.1 Design Principles

The architecture is deliberately minimal:

- **EMX2 is the system of record** for jobs, lifecycle state, and artifact metadata.
- **HPC is responsible for execution** via Slurm and Apptainer. EMX2 never tells the cluster how to schedule.
- **Inputs and outputs are tracked as artifacts.** Jobs reference artifacts by ID. Content is accessed via the artifact file API (managed) or directly via `file://`, `s3://`, or `https://` URIs (external). Managed artifacts store binary content in EMX2; external artifacts store only metadata.
- **Workers declare capabilities; EMX2 assigns only compatible jobs.** There is no negotiation.
- **The API is resource-oriented.** State transitions are sub-resources of jobs. Responses include hypermedia links advertising legal next actions.
- **Everything is recoverable.** Transitions are idempotent, timeouts detect stuck jobs, and the system converges to a consistent state after any single failure.

5 Processor and Execution Model

Jobs reference a logical processor identifier (e.g. `text-embedding:v3`) and an optional execution profile (e.g. `gpu-medium`). EMX2 does not encode cluster-specific scheduling parameters. Instead, the protocol uses a hybrid model that separates **application intent** from **cluster policy**.

EMX2 specifies *what* to run — a processor identifier and a profile. The head node determines *how* — which SIF image, which Slurm partition, how many GPUs, how much memory, and what wall time.

For example, given a job requesting `text-embedding:v3` with profile `gpu-medium`, the head node resolves this to:

```
text-embedding:v3 + gpu-medium
    → image: /nfs/images/text-embedding_v3.sif
    → partition: gpu
    → gpus: 1
    → cpus_per_task: 8
    → mem: 64G
    → time: 04:00:00
    → command: apptainer exec --nv /nfs/images/text-embedding_v3.sif ...
```

This mapping is maintained locally on the HPC system and may evolve independently of the protocol.

5.1 Why Hybrid Profiles

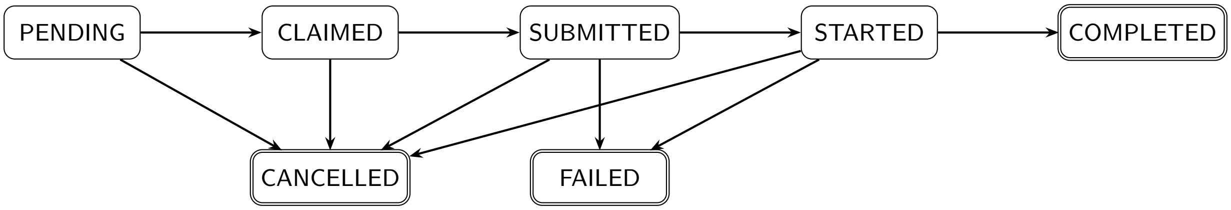
Three models were considered:

1. **Full embedding** — Slurm parameters in EMX2 payloads. Rejected: couples EMX2 to cluster configuration and violates institutional scheduling governance.
2. **Full delegation** — no hint from EMX2 at all. Rejected: EMX2 cannot express workload intent, making it impossible to distinguish a lightweight job from a GPU-heavy one.
3. **Hybrid profiles** (chosen) — EMX2 expresses intent through a logical profile; the HPC side maps it to concrete resources.

The hybrid model keeps scheduling policy within HPC governance while letting EMX2 express meaningful workload requirements. Cluster configuration can change without protocol changes.

6 Job Lifecycle

A job passes through a strict state machine. Every transition is recorded as a sub-resource and the full history is queryable as an audit log.



From	To	Initiated by	Trigger
PENDING	CLAIMED	Head node	Atomic claim
PENDING	CANCELLED	EMX2 or user	Cancel before claim
CLAIMED	SUBMITTED	Head node	After `sbatch`
CLAIMED	CANCELLED	Head node or EMX2	Cancel before submission
SUBMITTED	STARTED	Apptainer wrapper or daemon	Execution begins
SUBMITTED	FAILED	Head node or EMX2	Slurm rejection or timeout
SUBMITTED	CANCELLED	Head node or EMX2	Cancel; head node issues `scancel`
STARTED	COMPLETED	Apptainer wrapper or daemon	Outputs committed; `output_artifact_id` set

From	To	Initiated by	Trigger
STARTED	FAILED	Apptainer wrapper	Runtime error or hash mismatch
STARTED	CANCELLED	EMX2	Cancel; wrapper terminates

All other transitions are rejected with `409 Conflict`. Jobs in terminal states (COMPLETED, FAILED, CANCELLED) cannot transition further.

Jobs can be deleted via `DELETE /api/hpc/jobs/{id}`. Non-terminal jobs are automatically cancelled before deletion. The transition history is deleted with the job.

6.1 Failure Recovery

The protocol is designed to converge to a consistent state after any single failure.

Idempotent transitions. A transition request is identical when `job_id`, `status`, `worker_id`, and all payload fields match a previously accepted transition. Duplicates return `200 OK`. Non-identical submissions to the same state return `409 Conflict`. This allows safe retries on network failure.

Timeout-driven state progression. If a job stalls, EMX2 resolves it. CLAIMED with no SUBMITTED within timeout → FAILED, CANCELLED, or reset to PENDING (a deployment-time choice). STARTED with no terminal transition within timeout → FAILED. Timeouts should be generous enough for long-running GPU jobs and tunable per-processor or per-profile.

Infrastructure termination. If Slurm kills a job unexpectedly (node failure, preemption, wall-time exceeded), the Apptainer wrapper should detect this and post FAILED. If the wrapper itself is killed, the timeout mechanism applies.

Concurrency control. Workers declare `max_concurrent_jobs` during registration and are responsible for not over-claiming. EMX2 may optionally enforce an upper bound.

7 Artifact Store

Artifacts are the primary data objects in the system: job inputs, job outputs, model weights, execution logs, container images. This section describes how they are classified, where they live, how their integrity is ensured, and how their lifecycle is managed.

7.1 Classification

Every artifact is described along three dimensions.

Type describes the semantic role — what the artifact represents to consumers.

Type	Description	Typical formats
`tabular`	Structured row/column data	Parquet, CSV
`model`	Trained model weights or pipelines	GGUF, ONNX, SafeTensors
`dataset`	Multi-file scientific data	VCF + index, FASTA + .fai, Zarr

Type	Description	Typical formats
log	Execution logs from the Apptainer runtime	JSONL, plain text
report	Human-readable output	PDF, HTML, PNG
container	Apptainer image	SIF, OCI tar
blob	Opaque binary	Any

The type registry is extensible; new types can be added without protocol changes.

Format identifies the file encoding. For multi-file artifacts this describes the primary data file; ancillary files are described in the file manifest.

Format	Media type	Notes
parquet	application/vnd.apache.parquet	Columnar; supports remote range-request queries.
csv	text/csv	Row-oriented; queryable but less efficient at scale.
gguf	application/octet-stream	Quantised LLM weights for llama.cpp.
onnx	application/onnx	Open Neural Network Exchange format.
jsonl	application/jsonlines	Newline-delimited JSON; structured logs.
binary	application/octet-stream	Opaque.

Residence specifies where the content physically lives.

Residence	Content URI	Access pattern
managed	Artifact API endpoint	Upload/download via API; also S3-compatible endpoint.
posix	file:///nfs/...	Direct filesystem read from any node with the NFS mount.
s3	s3://bucket/key	Direct access with presigned URLs or credentials.
http	https://...	Direct download; supports range requests.
reference	N/A	Metadata-only; EMX2 tracks but does not store or proxy.

7.2 Residence: NFS

The cluster's NFS export is mounted on both the head node and all compute nodes. This makes it the natural location for large, frequently-reused artifacts: model weights, pre-built indices, and Apptainer SIF images. Data produced by one job is immediately available to the next.

The `posix` residence registers artifacts that live on NFS. The content URI is a `file://` path referencing the absolute mount location. The Apptainer runtime reads directly from NFS with no transfer overhead — the fastest access pattern for data already co-located with compute.

Since all nodes share the same NFS export, mount availability is not a per-node concern. Immutability is enforced by convention: operators must ensure that committed paths are not modified or deleted outside the protocol. Hash verification still applies — the runtime checks SHA-256 hashes before use, just as for any other residence.

7.3 Residence: Managed Repository

Managed artifacts are stored in EMX2's database using the FILE column type. The artifact file API exposes an S3-minimal surface — path-based `PUT` (upload), `GET` (download), `HEAD` (metadata), and `DELETE` operations on individual files within an artifact, plus paginated listing. This maps cleanly to WebDAV semantics and makes future S3-compatible gateway implementation straightforward.

The file API uses path-addressed URLs: `/api/hpc/artifacts/{id}/files/{path}`. Paths are logical names within the artifact (e.g. `data.parquet`, `model/weights.bin`) and support any depth. The server computes SHA-256 on upload and returns it in the response; clients do not need to pre-compute hashes for individual file uploads (though the overall artifact hash is provided at commit time).

For analytical tools (DuckDB, pandas), committed artifacts can be accessed via the GET endpoint with standard HTTP range requests. A future S3-compatible gateway could proxy these paths to provide native S3 connector support.

7.4 Multi-File Artifacts and Integrity

Some artifacts consist of multiple files: a model with a tokenizer sidecar, a VCF with a tabix index. Multi-file artifacts include a file manifest listing each file's path, role (`primary`, `index`, `metadata`, `ancillary`), size, and individual SHA-256 hash.

For single-file artifacts, the content hash is the SHA-256 of the file bytes. For multi-file artifacts, the top-level hash is a tree hash computed over sorted file paths and their individual hashes (see Appendix B.4). Any modification to any constituent file is detectable.

Input artifacts must be COMMITTED before a job can reference them. The Apptainer wrapper verifies hashes before execution — for managed artifacts it downloads and hashes locally; for NFS artifacts it reads from the mount. A mismatch results in a FAILED transition with reason `input_hash_mismatch`. Output artifacts are immutable after commit.

7.5 Schema Metadata

Tabular artifacts (`parquet` or `csv` format) include a `schema` field describing column names, types, nullability, row count, and (for Parquet) row group count. This is extracted automatically from the Parquet file footer at commit time, so consumers can discover data shape without downloading the file.

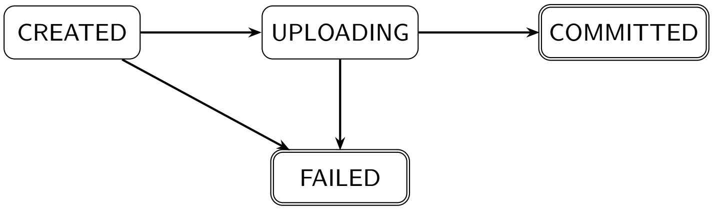
This metadata makes artifacts directly queryable. Tools like DuckDB can read a Parquet file from its NFS path or HTTP URL using range requests — fetching the footer first, then only the needed columns and rows. For NFS-resident artifacts this happens with zero network overhead. In practice, EMX2 application code or analytical scripts use this to inspect job outputs without pulling entire datasets: a SQL query against the artifact's `content_url` (or NFS path) returns results in place.

7.6 Execution Logs

The Apptainer wrapper uploads execution logs as artifacts of type `log`, governed by the same retention policy and integrity model as other artifacts. Log artifacts are referenced alongside outputs in the COMPLETED (or FAILED) transition. Structured JSONL is preferred over plain text for machine queryability.

7.7 Artifact Lifecycle

Managed artifacts pass through a state machine:



External artifacts (POSIX, S3, HTTP, reference) skip the upload phase: REGISTERED → COMMITTED or REGISTERED → FAILED.

Artifacts are immutable after COMMITTED. If an artifact stalls in CREATED or UPLOADING with no activity within a configured timeout, it transitions to FAILED and becomes eligible for garbage collection.

7.8 Job→Artifact Link

Jobs can reference output artifacts via the `output_artifact_id` field, which is a foreign key to the `HpcArtifacts` table. When the daemon completes a job and uploads (or registers) output artifacts, it passes the `output_artifact_id` in the COMPLETED transition request. EMX2 stores this link on the job record, making it queryable via GraphQL (`output_artifact_id { id type { name } status { name } }`).

Input artifacts are referenced in the job's `inputs` field (a JSON array of artifact IDs). The daemon stages input artifacts before execution: for managed artifacts it downloads files via GET; for posix artifacts it symlinks the `file:`// path into the job's input directory. This two-residence model means that large datasets on NFS incur zero transfer overhead, while smaller browser-uploaded artifacts are served from the managed store.

8 API Design

This section describes the principles governing the API. Full endpoint specifications with request and response payloads are in Appendix A.

8.1 Versioning

The API is versioned via the `X-EMX2-API-Version` request header rather than a URL path prefix. Versions are date-based strings (e.g. `2025-01`). This keeps URLs stable across versions and avoids cascading changes to hypermedia links. Missing header → `400 Bad Request`; unsupported version → `400 Bad Request`.

8.2 Request Headers and Traceability

Every request from a worker or runtime must include a standard set of headers:

Header	Required	Purpose
`X-EMX2-API-Version`	Yes	Protocol version (date-based, e.g. `2025-01`).
`X-Request-Id`	Yes	Unique per-request identifier (UUID v4).
`X-Timestamp`	Yes	Request creation time; used for HMAC verification and replay prevention.
`X-Nonce`	When HMAC enabled	Cryptographically random single-use value; replay prevention.
`X-Trace-Id`	No	Identifier spanning a logical operation, e.g. an entire job lifecycle.
`X-Worker-Id`	No	Worker identifier; used by some endpoints (e.g. cancel).
`Authorization`	When HMAC enabled	`HMAC-SHA256 <hex-signature>` (see Authentication and Trust).

EMX2 echoes `X-Request-Id` in error responses for traceability.

8.3 Resource Model and Hypermedia

The API is organised around two resources: **jobs** and **artifacts**. State transitions on jobs are a **transitions** sub-resource: each transition is a created resource (`POST` returns `201 Created`), the history is queryable, and the job representation includes `_links` advertising legal next actions. Clients follow links rather than hardcoding URL patterns (HATEOAS). All URLs in `_links` fields are opaque — clients dereference them as-is.

8.4 Error Responses

Client errors return structured JSON with `type`, `title`, `status`, and `detail` fields, following the RFC 9457 (Problem Details for HTTP APIs) structure. Whether to formally adopt the `application/problem+json` media type is an open deployment decision.

8.5 Endpoint Summary

All endpoints are under `/api/hpc`. Detailed specifications are in Appendix A.

Workers API: `POST /api/hpc/workers/register`, `POST /api/hpc/workers/{id}/heartbeat`.

Jobs API: `POST /api/hpc/jobs` (create), `GET /api/hpc/jobs` (list/filter), `GET /api/hpc/jobs/{id}`, `POST /api/hpc/jobs/{id}/claim`, `POST /api/hpc/jobs/{id}/transition`, `POST /api/hpc/jobs/{id}/cancel`, `DELETE /api/hpc/jobs/{id}`, `GET /api/hpc/jobs/{id}/transitions`.

Artifact API: `POST /api/hpc/artifacts` (create), `GET /api/hpc/artifacts/{id}` (metadata), `PUT /api/hpc/artifacts/{id}/files/` (upload file by path), `GET /api/hpc/artifacts/{id}/files/{path}` (download file), `HEAD /api/hpc/artifacts/{id}/files/{path}` (file metadata), `DELETE /api/hpc/artifacts/{id}/files/{path}` (delete file before commit), `GET /api/hpc/artifacts/{id}/files` (list files, paginated), `POST /api/hpc/artifacts/{id}/commit` (commit). Legacy: `POST /api/hpc/artifacts/{id}/files` (multipart upload, retained for backward compatibility).

Health: `GET /api/hpc/health` (exempt from authentication).

9 Authentication and Trust

The protocol operates across a trust boundary between EMX2 (public internet or institutional network) and the HPC environment (internal cluster network). Every API call crosses this boundary, so the security model must answer three questions: who is making this request, has the request been tampered with, and is this request fresh (not a replay of an earlier one)?

9.1 How Requests Are Authenticated

Every request from the HPC side to EMX2 carries a set of standard headers that together form an authentication envelope:

```
X-EMX2-API-Version: 2025-01
X-Request-Id:       <UUID v4>
X-Timestamp:        <unix epoch seconds>
X-Nonce:            <random value, used exactly once>
Authorization:      HMAC-SHA256 <hex-encoded signature>
```

The `Authorization` header contains an HMAC-SHA256 signature computed over a canonical request string: `METHOD\nPATH\nSHA256(body)\nTIMESTAMP\nNONCE`. This means:

- **Origin** — EMX2 can verify which worker sent the request, because only that worker has the secret needed to produce a valid signature.
- **Integrity** — if any part of the request (body, path, headers) is altered in transit, the signature will not match and EMX2 will reject it.
- **Freshness** — the `X-Timestamp` tells EMX2 when the request was created, and the `X-Nonce` is a random value that is never reused. EMX2 rejects requests whose timestamp is too far in the past (e.g. more than 5 minutes) and rejects any nonce it has seen before. Together these prevent an attacker from capturing a valid request and re-submitting it later.

9.2 Provisioning

Each head node is provisioned with two things before it can communicate with EMX2:

- A unique **`worker_id`** that identifies the head node across all API calls.
- A **shared secret** (for HMAC-based signing) or a **key pair** (for asymmetric signing or mTLS).

These credentials are issued by EMX2 and can be rotated without disrupting running jobs — during rotation, EMX2 accepts both the old and new credentials for a configurable grace period. Revoked credentials are rejected immediately.

9.3 Signing Mechanism

The protocol defines *what* is signed (method, path, body hash, timestamp, nonce) and *what headers carry it*, but leaves the choice of *how* the signature is computed as a deployment decision. Three options are common:

Mechanism	How it works	When to prefer it
HMAC-SHA256	Worker and EMX2 share a secret key. The worker computes a keyed hash over the canonical request string; EMX2 recomputes it and compares.	Simplest to implement; good when both sides can securely share a symmetric key. **Reference implementation available.**
JWT	Worker signs a short-lived JSON Web Token containing the request claims. EMX2 verifies the signature using the worker's public key or shared secret.	Useful when the infrastructure already has JWT tooling, or when tokens need to be inspected by intermediaries.
mTLS	Both sides present X.509 certificates during the TLS handshake. The connection itself authenticates the caller.	Strongest guarantee, but requires certificate management and TLS termination at the right point in the network.

The current implementation uses **HMAC-SHA256** as the default. The canonical request string is `METHOD\nPATH\nSHA256(body)\nTIMESTAMP\nNONCE`, transmitted as `Authorization: HMAC-SHA256 <hex>`. The shared secret is stored as a database setting (`MOLGENIS_HPC_SHARED_SECRET`, minimum 32 characters). When no secret is configured, HMAC verification is disabled (suitable for development only).

Replay protection enforces a 5-minute timestamp drift window and an LRU nonce cache.

9.4 What EMX2 Enforces

EMX2 is the sole authority for job state, lifecycle transitions, and artifact metadata. Workers cannot unilaterally change anything — they can only *request* transitions, which EMX2 validates against the state machine before accepting. This means even a compromised worker can only submit requests that are legal given the current state; it cannot, for example, mark someone else's job as completed or overwrite a committed artifact.

10 Summary, Trade-offs, and Open Questions

10.1 What This Protocol Provides

A minimal, deterministic bridge between EMX2 and HPC infrastructure with these invariants:

- **Outbound-only communication.** EMX2 never initiates connections to the cluster.
- **Apptainer-based execution.** SIF images on NFS, invoked by Slurm on compute nodes.
- **NFS as the primary shared data path.** Artifacts co-located with compute require no transfer.
- **Resource-oriented API with HATEOAS.** Clients discover actions from server responses.
- **Typed, content-addressed artifacts.** A single metadata model governs everything from queryable Parquet tables to multi-gigabyte model weights, with SHA-256 integrity verification.

- **Idempotent transitions and timeout-based recovery.** The system converges after any single failure.

10.2 Key Trade-offs

Polling vs. push. The outbound-only constraint requires the head node to poll EMX2 for new jobs, introducing latency proportional to the poll interval. A shorter interval reduces latency but increases API load. For the expected workload (long-running GPU jobs), 10–30 seconds is likely acceptable, but this should be validated under realistic load.

Hybrid profiles. EMX2 expresses workload intent; the HPC side interprets it. This preserves scheduling governance but means EMX2 cannot guarantee exact resource allocation. The head node's profile-to-Slurm mapping is an out-of-band dependency that must be kept in sync manually.

NFS immutability by convention. The protocol registers NFS paths as artifacts but cannot enforce immutability on them. If an operator modifies a file after it has been committed, the hash check will catch it at runtime — but the job will fail rather than being prevented. In environments with strict data governance this may need filesystem-level write protection (e.g. read-only snapshots or chattr).

S3-minimal file surface. The artifact file API exposes path-based GET/PUT/HEAD/DELETE operations that map to S3 semantics (`GetObject`, `PutObject`, `HeadObject`, `DeleteObject`). This is sufficient for the current use case and makes a future S3-compatible gateway straightforward to implement. Until then, analytical tools access managed artifacts via HTTP GET with range request support.

Authentication mechanism. The wire format (headers, nonce, timestamp) is defined; the signing mechanism is deliberately left open. This accommodates different institutional PKI, but means the authentication layer must be fully designed during implementation.

10.3 Resolved Design Decisions

Decision	Resolution
Error response format	RFC 9457 structure (`title`, `status`, `detail`), without formal `application/problem+json` content type.
Signing mechanism	HMAC-SHA256 as reference implementation. Protocol remains compatible with JWT and mTLS.
Implementation language for the daemon	Python (click CLI, httpx client, subprocess for Slurm).
Concurrency enforcement	Worker-side only. Workers declare `max_concurrent_jobs` during registration and self-enforce.
API version scheme	Date-based strings (e.g. `2025-01`) rather than integers.

10.4 Resolved Since Initial Design

Decision	Resolution
Managed storage access pattern	S3-minimal surface via path-based REST endpoints (`PUT/GET/HEAD/DELETE /files/{path}`). S3-compatible gateway deferred.
Job→artifact linking	`output_artifact_id` REF column on HpcJobs, set during COMPLETED transition.

Decision	Resolution
Input artifact staging	Two-path: symlink for posix, download for managed. Configured via `artifact_residence` in daemon config.
Browser upload	Direct PUT with raw binary body; browser computes SHA-256 via SubtleCrypto. No multipart required.

10.5 Open Design Decisions

Decision	Options	Considerations
Timeout values	Per-processor, per-profile, or global	Must accommodate longest GPU job; too short → false failures.
Artifact retention	TTL, reference-counted, or manual	Out of protocol scope, but the store must accommodate the chosen strategy.
S3-compatible gateway	MinIO proxy, custom gateway, or none	Current path-based API maps to S3 semantics; gateway adds DuckDB/pandas native S3 support.

Appendix A: API Reference

Full endpoint specifications. All endpoints require the standard headers from §8.2. URLs shown here are illustrative; in practice, clients follow `_links` from server responses.

A.1 Workers API

POST /api/hpc/workers/register

Registers a worker or updates its registration. Idempotent — subsequent calls update the heartbeat timestamp and replace the capability set.

```
{
  "worker_id": "hpc-headnode-01",
  "hostname": "login-node.cluster.local",
  "capabilities": [
    {
      "processor": "text-embedding:v3",
      "profile": "gpu-medium",
      "max_concurrent_jobs": 4
    }
  ]
}
```

Response: `200 OK` with worker metadata and HATEOAS links.

```
{
  "worker_id": "hpc-headnode-01",
  "hostname": "login-node.cluster.local",
  "registered_at": "2026-02-21T10:00:00",
  "last_heartbeat_at": "2026-02-21T10:00:00",
  "_links": {
    "self": { "href": "/api/hpc/workers/hpc-headnode-01", "method": "GET" },
    "heartbeat": { "href": "/api/hpc/workers/register", "method": "POST" },
    "jobs": { "href": "/api/hpc/jobs?status=PENDING", "method": "GET" }
  }
}
```

POST /api/hpc/workers/{id}/heartbeat

Lightweight heartbeat. Updates `last_heartbeat_at` without re-submitting capabilities. The daemon sends this periodically (default: every 120 seconds) between poll cycles.

Response: `200 OK` with `{"worker_id": "...", "status": "ok"}`.

A.2 Jobs API

POST /api/hpc/jobs

Creates a new job in PENDING status.

```
{
  "processor": "text-embedding:v3",
  "profile": "gpu-medium",
  "submit_user": "researcher@example.org",
  "parameters": { "model": "multilingual-e5-large", "batch_size": 256 },
  "inputs": { "dataset": "corpus-01" }
}
```

Response: `201 Created`

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "PENDING",
  "_links": {
    "self": { "href": "/api/hpc/jobs/550e8400-...", "method": "GET" },
    "claim": { "href": "/api/hpc/jobs/550e8400-.../claim", "method": "POST" },
    "cancel": { "href": "/api/hpc/jobs/550e8400-.../cancel", "method": "POST" }
  }
}
```

GET /api/hpc/jobs

Lists jobs with optional filtering and pagination. Query parameters: `status`, `processor`, `profile`, `limit` (default 100), `offset` (default 0). When `status` is omitted, defaults to `PENDING` (backwards compatible with worker polling).

Response: `200 OK` with paginated result.

```
{
  "items": [ { "id": "...", "status": "PENDING", "processor": "text-embedding:v3", ... } ],
  "count": 2,
  "total_count": 42,
  "limit": 100,
  "offset": 0,
  "_links": { "self": { "href": "/api/hpc/jobs", "method": "GET" } }
}
```

POST /api/hpc/jobs/{id}/claim

Atomically claims a job. Returns `409 Conflict` if already claimed, `404` if not found.

Request: `{ "worker_id": "hpc-headnode-01" }`

Response: `200 OK` with the job in CLAIMED state, including `_links` for `submit` and `cancel`.

```
{
  "id": "550e8400-...",
  "status": "CLAIMED",
  "worker_id": "hpc-headnode-01",
  "processor": "text-embedding:v3",
  "profile": "gpu-medium",
  "_links": {
    "self": { "href": "/api/hpc/jobs/550e8400-...", "method": "GET" },
    "transitions": { "href": "/api/hpc/jobs/550e8400-.../transitions", "method": "GET" },
    "submit": { "href": "/api/hpc/jobs/550e8400-.../transition", "method": "POST" },
    "cancel": { "href": "/api/hpc/jobs/550e8400-.../cancel", "method": "POST" }
  }
}
```

POST /api/hpc/jobs/{id}/transition

Reports a state transition. Rejects invalid transitions with `409 Conflict`. Idempotent: re-posting an identical transition returns `200 OK`. Response includes the updated job.

SUBMITTED (head node, after `sbatch`):

```
{ "status": "SUBMITTED", "worker_id": "hpc-headnode-01", "detail": "sbatch id 45678", "slurm_job_id": "45678" }
```

STARTED (Apptainer wrapper or daemon monitor):

```
{ "status": "STARTED", "worker_id": "hpc-headnode-01", "detail": "running on node-05" }
```

COMPLETED (after outputs committed):

```
{
  "status": "COMPLETED",
  "worker_id": "hpc-headnode-01",
  "detail": "exit code 0",
  "output_artifact_id": "art_abc123-..."
}
```

FAILED (head node or wrapper):

```
{ "status": "FAILED", "worker_id": "hpc-headnode-01", "detail": "input_hash_mismatch" }
```

POST /api/hpc/jobs/{id}/cancel

Convenience endpoint for cancellation. Transitions the job to CANCELLED from any non-terminal state. The head node issues `scancel` if a Slurm job ID is known.

Response: `200 OK` with updated job, `409 Conflict` if already terminal.

DELETE /api/hpc/jobs/{id}

Deletes a job and its transition history. Non-terminal jobs are automatically cancelled before deletion.

Response: `204 No Content`, `404 Not Found`.

GET /api/hpc/jobs/{id}/transitions

Ordered transition history (audit log).

```
{
  "items": [
    { "id": "tr_001", "from_status": null, "to_status": "PENDING",
      "timestamp": "2026-02-21T10:28:00", "worker_id": null, "detail": "Job created" },
    { "id": "tr_002", "from_status": "PENDING", "to_status": "CLAIMED",
      "timestamp": "2026-02-21T10:29:00", "worker_id": "hpc-headnode-01", "detail": "Claimed by worker hpc-headnode-
  ],
  "count": 2
}
```

Error response example

Follows RFC 9457 (Problem Details for HTTP APIs) structure.

```
{
  "title": "Conflict",
  "status": 409,
  "detail": "Cannot transition job 550e8400-... from PENDING to STARTED"
}
```

A.3 Artifact API

POST /api/hpc/artifacts

Creates an artifact. Managed artifacts start in CREATED; external artifacts (posix, s3, http, reference) start in REGISTERED.

Managed: `{ "type": "tabular", "format": "parquet", "residence": "managed" }`

NFS: `{ "type": "blob", "format": "mixed", "residence": "posix", "content_url": "file:///nfs/outputs/job-123" }`

S3: `{ "type": "tabular", "format": "parquet", "residence": "s3", "content_url": "s3://..." }`

Response: `201 Created`

```
{
  "id": "art_abc123-...",
  "type": "tabular",
  "status": "CREATED",
  "_links": {
    "self": { "href": "/api/hpc/artifacts/art_abc123-...", "method": "GET" },
    "upload": { "href": "/api/hpc/artifacts/art_abc123-.../files/{path}", "method": "PUT" },
    "upload_legacy": { "href": "/api/hpc/artifacts/art_abc123-.../files", "method": "POST" },
    "files": { "href": "/api/hpc/artifacts/art_abc123-.../files", "method": "GET" }
  }
}
```

GET /api/hpc/artifacts/{id}

Returns full metadata with HATEOAS links. Links vary by status: CREATED/UPLOADING include `upload` and (for UPLOADING) `commit`; COMMITTED includes `download`; all include `files`.

Managed artifact (committed):

```json
{
  "id": "art_abc",
  "type": "tabular", "format": "parquet", "residence": "managed",
  "status": "COMMITTED", "sha256": "b3a3f0...", "size_bytes": 52428800,
  "content_url": null,
  "created_at": "2026-02-21T10:00:00", "committed_at": "2026-02-21T10:05:00",
  "_links": {
    "self": { "href": "/api/hpc/artifacts/art_abc", "method": "GET" },
    "download": { "href": "/api/hpc/artifacts/art_abc/files/{path}", "method": "GET" },
    "files": { "href": "/api/hpc/artifacts/art_abc/files", "method": "GET" }
  }
}
```

NFS artifact: `"residence": "posix"`, `"content_url": "file:///nfs/data/outputs/embeddings.parquet"`

S3 artifact: `"residence": "s3"`, `"content_url": "s3://data-lake/outputs/analysis.parquet"`

PUT /api/hpc/artifacts/{id}/files/{path}

Uploads a file to an artifact by path. The `{path}` segment is the logical file name within the artifact (e.g. `data.parquet`, `model/weights.bin`). Upserts: if a file already exists at that path, it is replaced. Transitions the artifact from CREATED to UPLOADING on the first upload.

Accepts two body formats:

- **Raw binary** (preferred): `Content-Type` describes the file's media type; body is the raw file bytes. The server computes SHA-256 and stores the file.
- **Multipart**: `Content-Type: multipart/form-data` with a `file` part and optional `role`, `content_type` form params.

Request (raw binary):

```
PUT /api/hpc/artifacts/art_abc/files/data.parquet
Content-Type: application/vnd.apache.parquet
X-EMX2-API-Version: 2025-01
...

<raw file bytes>
```

Response: `201 Created`

```json
{
  "id": "file-uuid-...",
  "artifact_id": "art_abc",
  "path": "data.parquet",
  "sha256": "b3a3f0...",
  "size_bytes": 52428800
}
```

HMAC note: For non-JSON request bodies (raw binary uploads), the HMAC signature is computed over an empty string rather than the file bytes. This avoids encoding issues with large binary payloads and matches the token auth path.

GET /api/hpc/artifacts/{id}/files/{path}

Downloads file content. Returns the raw bytes with appropriate `Content-Type`, `Content-Disposition: attachment`, `Content-Length`, and `X-Content-SHA256` headers.

For managed artifacts with stored content, serves bytes directly. For posix/external artifacts where the file metadata exists but no binary is stored, returns `302 Found` redirecting to `{content_url}/{path}`.

Response headers:

```
Content-Type: application/vnd.apache.parquet
Content-Disposition: attachment; filename="data.parquet"
Content-Length: 52428800
X-Content-SHA256: b3a3f0...
```

HEAD /api/hpc/artifacts/{id}/files/{path}

Returns file metadata as headers without body content. Useful for checking existence and integrity without downloading.

Response headers: `X-Content-SHA256`, `Content-Length`, `Content-Type`. Status `200 OK` if found, `404 Not Found` otherwise.

DELETE /api/hpc/artifacts/{id}/files/{path}

Deletes a file from an artifact. Only allowed when the artifact is not yet COMMITTED.

Response: `204 No Content` on success, `409 Conflict` if artifact is committed, `404 Not Found` if file does not exist.

GET /api/hpc/artifacts/{id}/files

Lists files in an artifact with pagination and optional prefix filtering.

Query parameters: `prefix` (filter paths starting with this string), `limit` (default 100), `offset` (default 0).

Response: `200 OK`

```
{
  "items": [
    {
      "id": "file-uuid-...",
      "path": "data.parquet",
      "role": "primary",
      "sha256": "b3a3f0...",
      "size_bytes": 52428800,
      "content_type": "application/vnd.apache.parquet",
      "_links": {
        "content": {
          "href": "/api/hpc/artifacts/art_abc/files/data.parquet",
          "method": "GET"
        }
      }
    }
  ],
  "count": 1,
  "total_count": 1,
  "limit": 100,
  "offset": 0
}
```

POST /api/hpc/artifacts/{id}/files (legacy)

Uploads a file to an artifact. Retained for backward compatibility. Accepts either JSON metadata-only (`{ "path": "...", "sha256": "...", "size_bytes": ... }`) or multipart with a `file` part. New clients should prefer `PUT /files/{path}`.

POST /api/hpc/artifacts/{id}/commit

Commits the artifact with a top-level SHA-256 hash and total size. The artifact must be in UPLOADING (managed) or REGISTERED (external) status. Immutable after commit — subsequent uploads and deletes are rejected.

Request: `{ "sha256": "abc123...", "size_bytes": 1024 }`

Response: `200 OK` with full artifact metadata.

Artifact examples by residence

Multi-file model on NFS (posix):

```
{
  "id": "art_model_nfs",
  "type": "model", "format": "gguf", "residence": "posix",
  "status": "COMMITTED", "sha256": "d1e2f3...",
  "content_url": "file:///nfs/models/llama-3-8b/"
}
```

File listing for this artifact:

```
{
  "items": [
    { "path": "model.gguf", "role": "primary", "size_bytes": 4294967296, "sha256": "a1b2c3..." },
    { "path": "tokenizer.json", "role": "metadata", "size_bytes": 524288, "sha256": "d4e5f6..." }
  ]
}
```

For posix artifacts, the daemon registers file metadata without binary content. Consumers access files directly via the NFS mount at the `content_url` path. The GET file endpoint returns a `302` redirect to `{content_url}/{path}` for files without stored binary content.

Appendix B: State Machine Reference

B.1 Hypermedia Link Mapping (Jobs)

All states include `self` and `transitions` (read) links.

Current state	Mutation links
PENDING	`claim`, `cancel`
CLAIMED	`submit`, `cancel`
SUBMITTED	`start`, `cancel`
STARTED	`complete`, `fail`, `cancel`
COMPLETED	*(terminal)*
FAILED	*(terminal)*
CANCELLED	*(terminal)*

B.2 Hypermedia Link Mapping (Artifacts)

All states include `self` and `files` (list) links.

Current state	Mutation links
CREATED	`upload` (PUT), `upload_legacy` (POST)
UPLOADING	`upload` (PUT), `upload_legacy` (POST), `commit`
REGISTERED	`commit`
COMMITTED	`download` (GET template)
FAILED	*(terminal)*

B.3 Artifact Lifecycle Transitions

Managed: CREATED → UPLOADING (first `PUT /files/{path}`) → COMMITTED (`POST /commit`). CREATED or UPLOADING → FAILED (timeout).

External (POSIX, S3, HTTP, reference): REGISTERED → COMMITTED (verified) or REGISTERED → FAILED (unreachable / hash mismatch).

B.4 Tree Hash

```
sha256_tree = SHA256(concat(for each file in sorted(paths): path + ":" + sha256_hex(file_bytes)))
```

Single-file artifacts: `sha256(file_bytes)`.