ChronoHive Reference Manual

Version 1.0 — 2026-09-28

© 2026 Layer1Labs. All rights reserved. Confidential — prepared for evaluation purposes.

This manual is the formal product reference for ChronoHive: the system architecture, the Lingua Franca reactor model, the chronoc compiler, the Runtime kernel, the effector contract, the admission-policy model, the demo API, the evaluation API, and the local VS Code workflow. It describes verified current behavior only.

Companion documents:

- Record-trial report (GPU_AB_RECORD_RESULTS.pdf) — the measured 11.6% GPU A/B result.
- Storage-workflow integration guide (CHRONOHIVE_STORAGE_INTEGRATION_GUIDE.pdf) — where ChronoHive fits in your stack and how to adopt it.

---

1. Product overview

1.1 What ChronoHive is

ChronoHive is a **storage admission-control layer for GPU training**. A training workload declares its storage operations in the Lingua Franca coordination language; the chronoc compiler lowers that declaration to a constraint specification; and the ChronoHive Runtime kernel executes the specification, granting, deferring, or refusing each storage operation against a capacity budget. The operations themselves run through **effectors** — small, customer-owned functions that perform the real I/O against the customer's storage backend.

The contract is simple:

- **Training is never refused.** Compute-critical operations are admitted unconditionally.
- **Checkpoints are deferred, never lost.** Retry-policy operations that cannot be admitted now are carried and re-proposed until capacity frees.
- **Prefetch is shed first.** Drop-policy operations are discarded when capacity is exhausted.

On a recorded trial (RTX 5060 Ti, ResNet-18, six preregistered interleaved trials), admission control through this kernel produced a **median 11.6% end-to-end training speedup** with zero change to training correctness (loss trajectories bit-identical).

1.2 The problem it solves

AI training clusters routinely stall expensive GPUs on storage I/O: periodic model and optimizer checkpoints collide with read-ahead prefetch streams and data loading, and the resulting incast saturates storage bandwidth at exactly the moment training needs it. No storage system offers **application-level admission control** to arbitrate this contention — the application knows which I/O is critical and which is deferrable, but the storage layer cannot see that distinction. ChronoHive is that arbitration layer: the application declares what each operation needs and what may happen to it under pressure, and the kernel enforces the policy at runtime.

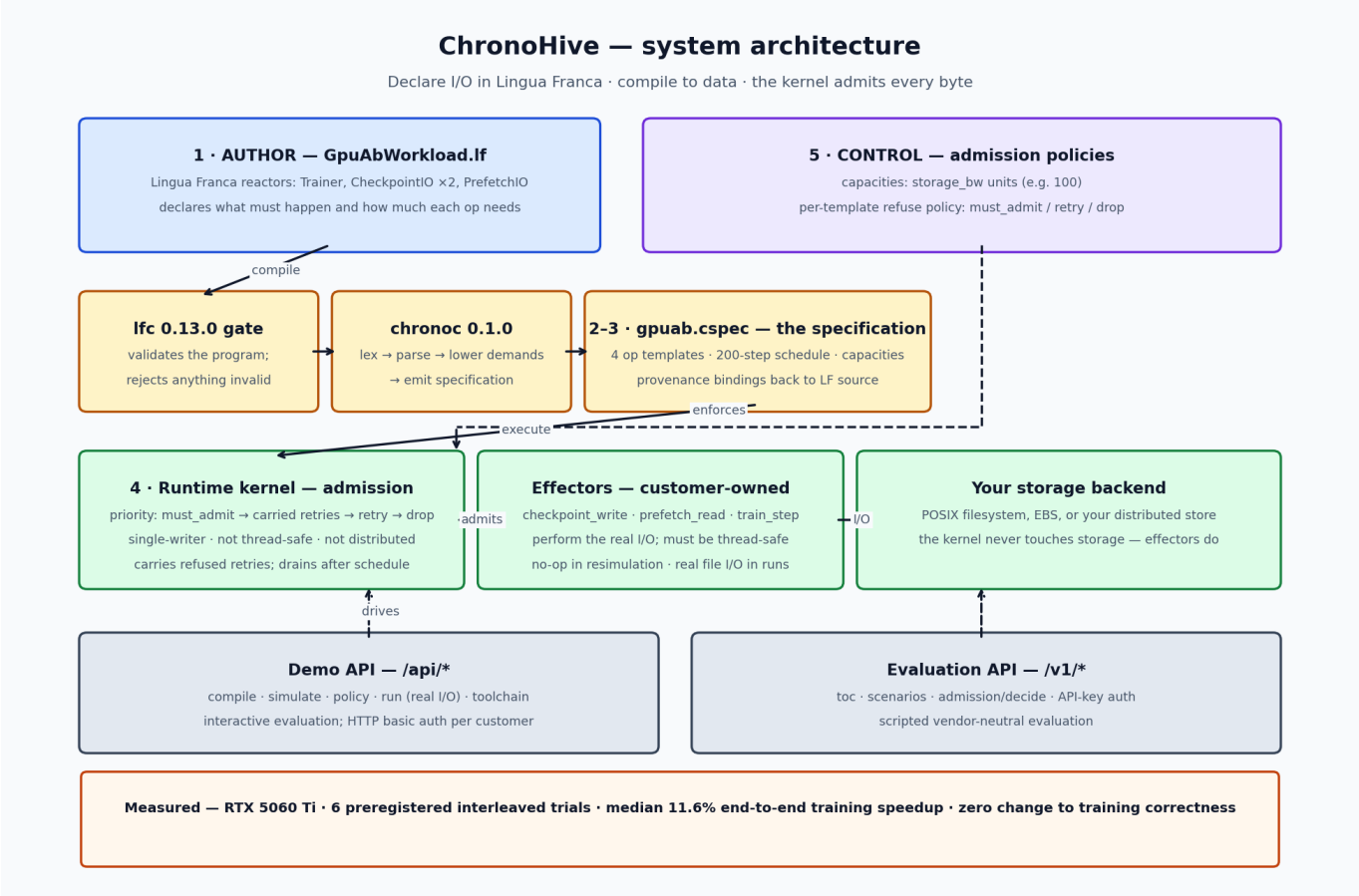
1.3 What ChronoHive is not

- ChronoHive is **not a scheduler** in the operating-system sense and **not a** distributed system**. The kernel is a single-writer admission engine; it is not thread-safe and must not be shared across threads without external sequencing.**
- ChronoHive is **not a storage backend**. It performs no I/O itself. Every byte moves through an effector the customer supplies.
- The interactive resimulation in the demo uses **no-op effectors**: the compile and admission decisions are real, but the timings are not simulated and no I/O occurs. Measured timings come only from real-I/O runs and the record GPU trial.

---

2. System architecture

2.1 The five layers



The five layers, top to bottom:

Two API planes sit alongside these layers:

- **Demo API** (/api/*) — compile Lingua Franca, resimulate admission decisions, edit rules, and run real I/O, all against the hosted demo server.
- **Evaluation API** (/v1/*) — the hosted admission-decision service: submit transfer requests, run scripted contention scenarios, retrieve results.

© 2026 Layer1Labs · Confidential

A full block diagram with every component and data flow is published with the evaluation materials (see chronohive_architecture.png).

2.2 Data flow, end to end

1. You author GpuAbWorkload.lf (§3). 2. chronoc compile GpuAbWorkload.lf -o gpuab.cspec --lfc <lfc-0.13.0> --capacity storage_bw=100 runs the lfc gate, then lowers the program to gpuab.cspec (§4). 3. Your host program loads the specification, supplies an effector per effector name, sets capacities and refuse policies, and calls execute(blob, effectors) (§6, §8). 4. The kernel admits each step's operations in priority order — must_admit first, then carried retries, then retries, then drops — and runs the admitted effectors, concurrently when asked (§6.1). 5. Refused retry-policy operations are carried and re-proposed; refused drop-policy operations are discarded (§6.3, §7.1).

2.3 Trust boundaries

- The **lfc gate** is the trust boundary for program validity: chronoc refuses to compile anything the pinned lfc rejects.
- The **blob** is the trust boundary for provenance: every template carries a binding back to its Lingua Franca reaction site (§5.3).
- The **kernel** is the trust boundary for admission: effectors never decide; they only execute what the kernel admitted.
- The **effector contract** is the trust boundary for I/O: the kernel never touches storage; the customer-owned effector does (§8.3).

3. Lingua Franca reactor specifics

This chapter is the reference for the Lingua Franca subset ChronoHive consumes. It is written for an evaluator who knows general programming but may not know Lingua Franca. The running example is the record workload, GpuAbWorkload.lf.

3.1 Why Lingua Franca

Lingua Franca (LF) is a coordination language for reactive systems. A program is a set of **reactors** — encapsulated components with ports, timers, state, and **reactions** (event handlers) — wired together by explicit **connections**. For ChronoHive this gives three properties a plain script cannot:

- **Isolation of concerns.** Training, model checkpoints, optimizer checkpoints, and prefetch are separate reactors. The compiler can see which I/O belongs to which concern.
- **Declared resource needs.** Sizes (MiB) and depths are reactor parameters, so the compiler turns them into quantified resource demands — not comments.
- **Explicit wiring.** Every storage request flows through a connection the kernel can observe. There is no hidden I/O.

ChronoHive does not execute Lingua Franca. It consumes the program's structure (reactors, timers, reactions, connections) and lowers it to data.

3.2 The target declaration

© 2026 Layer1Labs · Confidential

Every Lingua Franca program must declare a target language:

```
target Python
```

In upstream Lingua Franca, the target declaration chooses which programming language's code the compiler generates — target Python makes lfc generate a Python program, target C a C program, and so on. ChronoHive is different: the reaction bodies in ChronoHive workloads are written in Python, so the file is validated against the Python target's grammar by the real lfc compiler, but chronoc is not a target and generates no Python at all. It consumes the validated structure and emits the constraint specification (§5) — that blob, not generated code, is chronoc's output, and it is the same specification for every runtime that executes it. **No Python is generated by chronoc, and no Python runs in the ChronoHive engine.** The engine executes blob data through effectors (§8), which may be written in any language.

So that a specification stays self-describing — traceable to the exact source and toolchain that produced it, the way an lfc output is tied to its target — chronoc records the declared target in the specification's provenance metadata (the lf_target field, §5.4). The engine's loader reads it back and reports it with every run, so a specification never has to be guessed at: its source hash, the compiler versions, and the target its source declared are all inside it.

3.3 Reactors, parameters, and state

```
reactor Trainer(steps=200, ckpt_every=5) {
    ...
}
```

A reactor is a named component with **parameters** in parentheses. Parameters are compile-time constants visible to the lowering step: steps=200 becomes the schedule length, ckpt_every=5 becomes the checkpoint period. Instances are created with the new keyword and may override parameters:

```
t = new Trainer(steps=steps, ckpt_every=ckpt_every)
```

Reactor **state** is private mutable memory, declared with state:

```
state step = 0
```

State is per-instance and persists across reactions. In the record workload, Trainer keeps a step counter; each timer tick increments it.

3.4 Ports: input and output

Ports are the only way reactors communicate:

```
output step_done
output ckpt_request
...
input ckpt
```

An **output port** emits events; an **input port** receives them. Reactions trigger on ports

© 2026 Layer1Labs · Confidential

and write to ports with .set(...). Ports carry values (here a small dictionary such as {"step": step}), but ChronoHive's lowering cares about the event structure — which reactions fire, and what they request — not the payload values.

3.5 Timers

```
timer tick(0, 1 ms)
```

A timer produces periodic events: first at offset 0, then every 1 ms. The Trainer reactor's timer is the heartbeat of the workload — one tick per training step. The period is a logical tag; the ChronoHive kernel advances a synthetic clock (step_no * step_ns) rather than sleeping in real time.

3.6 Reactions

```
reaction(tick) -> step_done, ckpt_request {=
    train_step()
    step = step + 1
    step_done.set({"step": step})
    if step % ckpt_every == 0:
        ckpt_request.set({"step": step})
    if step >= steps:
        request_stop()
=}
```

A reaction declares its **triggers** in parentheses (tick), its **effects** after -> (step_done, ckpt_request), and its body between {= and =}. The body is target-language code (Python here). The calls that matter to ChronoHive are the effector invocations — train_step(), admit_checkpoint(mb=mb), prefetch(depth=depth) — which the lowering step maps to operation templates. Control flow in the body (if step % ckpt_every == 0) determines when operations are proposed: the lowering evaluates it against the parameters to produce the schedule.

Key reaction rules in the supported subset:

- A reaction lists every port it reads as a trigger and every port it writes

after ->. Undeclared port access is rejected by the lfc gate.

- request_stop() ends the schedule; the lowering uses it to bound the steps.
- Bodies may use parameters and state but not arbitrary I/O — the only

observable actions are port writes and effector calls.

3.7 Connections and the main reactor

```
main reactor GpuAbWorkload(steps=200, ckpt_every=5,
        ckpt_mb=70, opt_ckpt_mb=55, prefetch_depth=22) {
    t  = new Trainer(steps=steps, ckpt_every=ckpt_every)
    c  = new CheckpointIO(mb=ckpt_mb)       // 70 MiB model
    c2 = new CheckpointIO(mb=opt_ckpt_mb)   // 55 MiB optimizer
    p  = new PrefetchIO(depth=prefetch_depth) // depth 22
    t.ckpt_request -> c.ckpt
    t.ckpt_request -> c2.ckpt
    t.step_done -> p.tick
}
```

© 2026 Layer1Labs · Confidential

The **main reactor** is the program's top level: it instantiates reactors and wires them with **connections** (a.port -> b.port). Connections are directed and explicit. Fan-out is allowed (t.ckpt_request feeds both c.ckpt and c2.ckpt).

Naming rule. The file name must match the main reactor name: GpuAbWorkload lives in GpuAbWorkload.lf. The lfc gate enforces this; the demo server derives the file name from the main reactor declaration so edited programs keep passing.

Reading this wiring tells you the whole contention story before any code runs: every 5th step, t emits ckpt_request, which reaches both checkpoint reactors, while step_done continuously drives prefetch. The compiler turns this into the schedule; the kernel turns the schedule into admission decisions.

3.8 The preregistered Lingua Franca subset (subset v1)

chronoc lowers a precisely defined subset of Lingua Franca, called **subset v1**. It is the subset used for the preregistered record trial: any program outside it is rejected at compile time with a lowering error (surfaced as compile_failed). The rules below summarize the subset as implemented by chronoc v0.1.0 (parser + lowering); the compiler is the authority and this prose tracks it but does not replace it.

Program shape.

- S1. The program contains exactly one main reactor.
- S2. The file name matches the main reactor name (GpuAbWorkload in

GpuAbWorkload.lf); the lfc gate enforces this before lowering begins. The target must be Python — chronoc pins the Python target for the lfc validation path.

- S3. Reactor bodies may contain only: state declarations (integer

initializers, state step = 0), timer declarations, input/output ports, reaction blocks, reactor instances (x = new R(...)), and connections (a.p -> b.q). Any other member is a parse error.

- S4. Connections resolve instance paths relative to the enclosing reactor's

scope; nested instantiation is expanded recursively, so nested instance paths do resolve. In subset v1, connections are expected in the main reactor addressing top-level instance paths. Connection endpoints are validated: instances must exist, ports must be declared, sources must be outputs, destinations inputs.

- S5. Timer units are us, ms, s, or min (enforced by the parser for

every timer). Time is logical: the kernel advances a synthetic clock (step_no * step_ns, from the driver's timer period); nothing sleeps in real time.

Triggers.

- S6. Logical (untimed) actions are not supported — timers only.
- S7. A reaction may have at most one timer trigger.

The driver loop.

- S8. The program must contain exactly one driver reaction: a reaction on a

periodic timer that increments a state counter (<c> = <c> + 1) and stops the schedule with if <c> >= <param>: request_stop(). Zero or multiple driver reactions are compile errors. The stop bound must resolve (against instance parameters) to an integer in

© 2026 Layer1Labs · Confidential

1..=1_000_000; it sets the schedule length.

- S9. A timer-driven reaction without the counter increment and stop guard is

rejected; a >= stop guard without a timer trigger is rejected.

Reaction bodies. A body may contain only these statements:

- S10. **Intrinsic calls** — exactly three are recognized (see the table below).
- S11. **Counter increment** — <c> = <c> + 1 where <c> is a declared state

variable, at most once per reaction. The parser accepts other assignments, but the lowering rejects every assignment that is not the recognized increment.

- S12. **Port writes** — port.set(...).
- S13. **request_stop()** — recognized only inside a >= stop guard, where

it marks the stop condition. A bare request_stop() anywhere else is silently ignored: no lowering error, no effect.

Guards.

- S14. Only one nesting is rejected: a modulo guard nested inside another

modulo guard. A >= stop guard may appear inside a % guard. Statements inside a >= stop guard other than request_stop() are silently ignored.

- S15. Exactly two guard forms exist: the periodic form

if <counter> % <k> == 0: (the right-hand side must be exactly 0) and the stop form if <counter> >= <param>: (the latter only together with request_stop()). Any other condition is a parse error.

- S16. A guarded intrinsic call — and a guarded port set — must appear after

the counter increment in the reaction body (the guard reads the post-increment counter).

Intrinsics. These are the only effector call sites the lowering recognizes. Any other call is an "unknown intrinsic" compile error.

Intrinsic call	Effector	Refuse policy	Required argument
train_step()	train_step	must_admit	none
admit_checkpoint(mb=<n>)	checkpoint_write	retry	mb=
prefetch(depth=<n>)	prefetch_read	drop	depth=

Deliberately excluded. Actions, multi-timer reactions, nested modulo guards, arbitrary assignments, physical-time behavior, and target code generation are all outside subset v1. The lfc gate additionally rejects any program that violates Lingua Franca's own well-formedness rules (undeclared ports, undeclared effects, and so on) before the subset rules are even applied.

The subset is intentionally narrow: it admits exactly the programs whose resource demands and schedules can be derived statically and checked honestly. Widening it is a compiler change, not a configuration change.

3.9 The record workload, annotated

© 2026 Layer1Labs · Confidential

```
target Python

reactor Trainer(steps=200, ckpt_every=5) {
    output step_done
    output ckpt_request
    state step = 0
    timer tick(0, 1 ms)

    reaction(tick) -> step_done, ckpt_request {=
        train_step()                        // template t.train_step, must_admit
        step = step + 1
        step_done.set({"step": step})       // drives prefetch every step
        if step % ckpt_every == 0:
            ckpt_request.set({"step": step}) // drives both checkpoints
        if step >= steps:
            request_stop()                  // 200 steps total
    =}
}

reactor CheckpointIO(mb=70) {               // model weights
    input ckpt
    reaction(ckpt) {=
        admit_checkpoint(mb=mb)             // template c.admit_checkpoint, retry
    =}
}
// A second instance is wired with mb=55 for optimizer state.

reactor PrefetchIO(depth=22) {              // read-ahead stream
    input tick
    reaction(tick) {=
        prefetch(depth=depth)               // template p.prefetch, drop
    =}
}

main reactor GpuAbWorkload(...) { ... }    // wiring as in §3.7
```

Lowered result: 4 operation templates, a 200-step schedule. A normal step runs **train + prefetch**; every 5th step runs **train + model ckpt + optimizer ckpt + prefetch** — demand 70 + 55 + 44 = 169 against capacity 100. Each demand fits alone, so a correct kernel sheds only the deferrable and droppable work, never training.

3.10 Editing guidance

- Change **parameters** (steps, ckpt_every, mb, depth) to reshape the workload; the compiler re-derives demands and the schedule.
- Change **wiring** to change which events drive which I/O.
- Do not rename the main reactor without renaming the file.
- Keep reaction bodies to port writes, parameter/state logic, and effector calls. Anything else is invisible to the kernel.
- In the demo, press **Ctrl/Cmd+Enter** to compile; the editor highlights Lingua Franca syntax (§11 covers the VS Code setup for local editing).

© 2026 Layer1Labs · Confidential

4. The chronoc compiler

4.1 CLI reference

```
chronoc compile <program.lf> -o <out.cspec> [options]
```

Option	Meaning
-o <out.cspec>	Output blob path (required)
\--lfc <path>	Path to the pinned lfc 0.13.0 binary (required)
\--param <name>=<int>	Main-reactor parameter override (repeatable)
\--mb-bw <n>	MiB of demand per MiB of checkpoint parameter (calibration)
\--depth-bw <n>	Units of demand per unit of prefetch depth (calibration)
\--capacity <resource>=<units>	Capacity budget, e.g. storage_bw=100 (repeatable)
\--java-home <path>	Java home for the lfc gate (needs Java 17+)
\--skip-lfc	Skip the lfc gate (testing only — never for record builds)

Example:

```
chronoc compile GpuAbWorkload.lf -o gpuab.cspec \
    --mb-bw 1 --depth-bw 2 --capacity storage_bw=100 \
    --lfc /opt/lfc/bin/lfc
```

With --mb-bw 1, a 70 MiB checkpoint parameter becomes 70 demand units; with --depth-bw 2, prefetch depth 22 becomes 44 demand units. Calibration is explicit and recorded — it is part of the compile, not hidden in the kernel.

4.2 The lfc gate

Before lowering begins, chronoc invokes the pinned lfc 0.13.0 on the source. If lfc rejects the program — unknown ports, undeclared effects, a file name that does not match the main reactor, any grammar violation — compilation stops and the real lfc diagnostic is surfaced. chronoc never compiles what lfc rejects. The demo API surfaces this as HTTP 422 compile_failed with the compiler's message.

4.3 Lowering

Lowering walks the validated program and produces:

1. **Operation templates** — one per distinct effector call site (e.g. c.admit_checkpoint), each with its effector name, demand vector, and refuse policy derived from the wiring (training reactions are unconditional → must_admit; checkpoint reactions are periodic and re-drives are safe → retry; prefetch is continuous and loss-tolerant → drop). 2. **The schedule** — the ordered list of steps; each step lists the template indices proposed at that step, derived by evaluating reaction guards (step % ckpt_every == 0) against the parameters. 3. **Provenance bindings** — each template records the Lingua Franca reaction site it came from (e.g. c.CheckpointIO::reaction(ckpt)::admit_checkpoint), plus the

© 2026 Layer1Labs · Confidential

source sha256, lfc version, and chronoc version.

4.4 Errors

Condition	Surfaced as
lfc rejects the program	compile_failed (422) with the lfc diagnostic
Construct outside the supported subset	compile_failed (422) with the lowering error
Schedule exceeds the step cap	schedule_too_large (422)
Toolchain missing (no chronoc / no lfc)	toolchain_unavailable (500)
Compile exceeds 180 s	compile_timeout (500)

5. The specification (.cspec)

The specification is the compilation artifact: binary **data**, not code. The engine never parses Lingua Franca, never invokes a code generator, and never runs a language runtime. It reads the specification and executes what the specification describes.

Not the CPSC binary format. Layer1Labs maintains a separate CPSC Binary Format Specification (Deterministic State Interchange Format) for encoding CPSC degrees of freedom and reconstruction metadata (layer1labs/core, docs/ip/specs/Binary-Format-Specification.md). The .cspec file is a different artifact: a compiled workload description — operation templates, schedule, demands, provenance — for the admission kernel, not CPSC state. It follows the same header conventions (magic bytes, version, flags, little-endian integers, CRC integrity) but is specified in this section, not there.

5.1 Operation templates

One template per distinct intrinsic call site (instance + intrinsic name):

Field	Meaning	Record example
id	Template id, <instance>.<intrinsic>	c.admit_checkpoint
effector	Effector name to invoke	checkpoint_write
on_refuse	must_admit / retry / drop	retry
demands	List of (resource, units)	("storage_bw", 70)
binding	Provenance: LF reaction site	c.CheckpointIO::reaction(ckpt):: admit_checkpoint

The record blob carries 4 templates: t.train_step (must_admit, no demand), c.admit_checkpoint (retry, 70), c2.admit_checkpoint (retry, 55), p.prefetch (drop, 44).

5.2 The schedule

The schedule is the ordered list of steps. Each step lists the template indices proposed at that step, in topological order (dependencies first; Kahn's algorithm on the connection graph). The record schedule has 200 steps: a normal step proposes train + prefetch; every 5th step proposes train + both checkpoints + prefetch.

5.3 Capacities and provenance

The specification records the capacity budget as (resource, units) pairs — e.g. ("storage_bw", 100) — and, for every template, the binding back to its Lingua Franca

© 2026 Layer1Labs · Confidential

reaction site plus the source sha256, the lfc version, the chronoc version, and the Lingua Franca target declared in the source (lf_target, §3.2). Every byte the engine executes traces to a line of the authored source.

5.4 Byte layout (as written by `chronoc`)

All integers are little-endian. String indices refer to the string table.

Offset	Field
0:4	Magic "CSP1"
4:6	Format version u16 (= 1)
6:8	Flags u16 (= 0)
8:16	step_ns u64 — synthetic clock nanoseconds per step
…	String table: count u32, then per string: length u32 + UTF-8 bytes
…	Capacities: count u32, then per entry: name_idx u32 + units u32
…	Operations: count u32, then per op: id_idx u32, effector_idx u32, on_refuse u8 (0 = must_admit, 1 = retry, 2 = drop), demand count u16 then per demand res_idx u32 + units u32, dependency count u16 then per dep op_idx u32
…	Schedule: step count u32, then per step: step_no u32, op count u16, then op_idx u32 each
…	Bindings: count u32, then per binding: lf_path_idx u32 + op_idx u32
…	Provenance: lf_sha256 (32 bytes), lfc_version_idx u32, chronoc_version_idx u32, lf_target_idx u32 (the LF target declared in the source, e.g. "Python"), target_steps u32
last 4	CRC-32 (IEEE) u32 over all preceding bytes

The loader (src/chronohive/blob.py::load_blob) validates magic, version, CRC-32, string/op index bounds, demand ⊆ capacities, and dependency acyclicity before the kernel is ever constructed; any violation raises. (Spec 002 REQ-106 describes this format; the table above follows the chronoc implementation, which additionally carries step_ns and uses u16 counts for demands, dependencies, and per-step ops.)

---

6. The Runtime kernel

6.1 The admission algorithm

For each schedule step, the kernel proposes operations in this priority order:

1. The step's **must_admit** operations (training always runs). 2. **Carried retries** — overdue obligations from earlier steps. An overdue obligation outranks the current step's

© 2026 Layer1Labs · Confidential

sheddable work, so a deferred checkpoint can never be starved by current drop-policy prefetch. 3. The step's own **retry** operations. 4. The step's own **drop** operations.

Within each class, order is stable (schedule order). Each operation is proposed as a single-operation plan: the kernel checks it against the remaining capacity and either admits it (recording a hold) or refuses it. Holds accumulate across the step — nothing completes until every admission is decided — so the kernel sees the real combined demand and refuses honestly.

Admitted operations then run in program order: overdue retries first, then the step's operations in topological schedule order. Admitted retries leave the carried set; refused retries stay carried.

6.2 Capacity discipline

A plan is admitted only if every operation is pending, its dependencies are complete, the plan is within its validity interval, and the summed demand fits the remaining capacity on every resource. The check-then-commit is atomic: the kernel validates the complete proposal before recording anything. Capacity is a plain integer budget per resource (e.g. storage_bw = 100); one demand unit is one MiB of checkpoint or depth_bw units per prefetch depth unit, per the compile-time calibration (§4.1).

6.3 Retry carry and drain

A refused retry-policy operation is **carried**: it keeps its original step identity (<template>#<origin_step>) and is re-proposed at every subsequent step, ahead of that step's sheddable work, until admitted. After the schedule ends, the kernel drains: carried retries are re-proposed for up to drain_steps extra steps (default 5) so a retry deferred near the end still completes instead of being dropped at the boundary. Anything still carried after the drain is reported as dropped.

6.4 Concurrency model

The kernel is **single-writer and not thread-safe**: all kernel transitions (admission, completion observation) happen on one sequencer thread in deterministic run order, and admission decisions are unchanged by concurrency. When concurrent=True, a step's admitted effectors run on worker threads so their I/O genuinely overlaps (this is what creates real storage contention in the record trial). **The effectors themselves must be thread-safe against each other** — they share no kernel state, only backend I/O.

The kernel is never described as thread-safe or distributed. It is a local admission engine.

6.5 Run reports

Every execution returns a run report: wall seconds, steps completed, target steps, total refusals, and per-template counts (admitted / refused / retried / dropped) plus per-effector seconds. The demo API validates every report field before serializing it; a malformed report is surfaced as an error, never as plausible numbers.

---

7. Admission policies

7.1 The three refuse policies

Policy	On refusal	Use for
must_admit	The run fails rather than refuse. A refused must_admit operation is a kernel error, never a shed.	Training steps — anything correctness-critical.
retry	The operation is carried and re-proposed at later steps until admitted or the drain ends.	Checkpoints — deferrable but must eventually land.
drop	The operation is discarded for this step.	Prefetch — loss-tolerant read-ahead.

Policies come from the lowering (intrinsic table, §3.8) and can be overridden per template at run time through the API (policy.on_refuse), e.g. to ask "what if checkpoints were droppable?" The override is explicit and reported; it does not rewrite the specification.

7.2 Capacities and units

Capacities are integer budgets per resource. The demo and record trial use one resource, storage_bw, with 1 unit = 1 MiB of checkpoint demand (calibrated by --mb-bw 1) and depth_bw units per prefetch depth unit. Capacity 100 with demand 169 on checkpoint steps is the record trial's contention point. The unmanaged baseline is modeled as effectively infinite capacity: everything is admitted and the run pays the full contention cost.

7.3 Quality of service

QoS falls out of the priority order, not out of a separate mechanism: training is must_admit, overdue checkpoints outrank fresh prefetch, and fresh retries outrank fresh drops. There are no tunable weights to misconfigure — the policy is the three refuse classes plus the capacity numbers.

7.4 Editing the rules

Through the demo API (POST /api/policy to read, policy field on /api/simulate and /api/run to override):

```
policy = {
  "capacities": {"storage_bw": 60},
  "on_refuse": {"p.prefetch": "drop", "c.admit_checkpoint": "retry"}
}
```

Unknown template ids, unknown policy names, and non-positive capacities are rejected with bad_policy (422). In the demo page, the rules editor (Act 3) loads the compiled defaults and applies your edits to the resimulation and the real-I/O run.

---

8. Effectors

8.1 The standard three

Effector	Does	Record behavior
train_step	The training step's storage I/O (data loading)	1 MiB batch read per step

© 2026 Layer1Labs · Confidential

Effector	Does	Record behavior
checkpoint_write	Durable checkpoint write	ckpt_mb MiB write + fsync per admitted op
prefetch_read	Read-ahead from the data corpus	prefetch_mb MiB bulk read per admitted op

8.2 The EffectorContext contract

Every effector is called as effector(ctx) where ctx carries:

Field	Meaning	Example
op_id	Kernel instance id	c.admit_checkpoint#5
template_id	Blob template id	c.admit_checkpoint
effector	Effector name	checkpoint_write
step	Current schedule step	6
origin_step	Step the instance belongs to	5 (a carried retry)
now_ns	Synthetic kernel clock	6000000
attempt	1 + number of prior refusals	2

An effector performs its I/O and returns. It never sees the kernel, the specification, or other operations.

8.3 The thread-safety contract

Effectors run on worker threads when concurrent=True and share the customer's backend I/O path. **Concurrent effectors must themselves be thread-safe.** The kernel guarantees single-writer sequencing of admission decisions; it does not serialize your I/O. Design effectors as independent workers: no shared mutable state without your own locking, one file handle per operation or a thread-safe pool, fsync scoped to the file being written.

8.4 Writing a customer effector

A customer effector adapts the kernel's admission decision to the customer's storage backend:

```
def checkpoint_write(ctx: EffectorContext) -> None:
    # ctx.template_id   -> which checkpoint stream (model vs optimizer)
    # ctx.origin_step   -> which training step this checkpoint belongs to
    # ctx.attempt       -> 1 on first admission, higher after deferrals
    backend.durable_write(
        stream=ctx.template_id,
        tag=f"step-{ctx.origin_step}",
        sync=True,   # fsync: the checkpoint must survive a crash
    )
```

Map each blob effector name to one callable and pass the mapping to execute. The specification declares the names; you own the bytes. This is the integration seam: the same kernel and the same specification drive a POSIX filesystem today and your distributed store tomorrow, with only the effectors changing.

8.5 Demo effectors: no-op vs real I/O

The demo exposes both ends honestly:

© 2026 Layer1Labs · Confidential

- **Resimulation** (/api/simulate) uses **no-op effectors**: every admission

decision is real, but no I/O happens and timings are not simulated. Use it to study decisions.

- **Real-I/O runs** (/api/run) use **real file-I/O effectors**: checkpoint

writes are fsync'd to disk, prefetches really read, page cache is dropped between operations (best effort), and both arms run with real thread overlap. Wall-clock seconds are measured, not simulated. This is a same-host, back-to-back measurement, not a controlled experiment.

9. Demo API reference

Base URL: the demo server (e.g. https://chronohive-demo.fly.dev). All routes require HTTP basic auth (one username/password per customer, managed as server secrets). All request/response bodies are JSON.

9.1 `GET /api/toolchain`

Returns the resolved toolchain and calibration. Use it to confirm the server can compile before posting source.

```
{"ok": true,
 "tools": {"chronoc": "/path/to/chronoc", "lfc": "/path/to/lfc", "java_home": "..."},
 "calibration": {"mb_bw": "1", "depth_bw": "2", "default_capacity": 100}}
```

9.2 `POST /api/compile`

Compiles Lingua Franca source through the real lfc gate and chronoc.

Request:

```
{"source": "<lf source text>", "capacity": 100}
```

Response (200):

```
{"ok": true, "cached": false, "lfc_gate": "passed",
 "lfc_version": "0.13.0", "chronoc_version": "0.1.0",
 "lf_sha256": "178d85ce…", "blob_bytes": 3683,
 "n_ops": 4, "n_steps": 200,
 "capacities": {"storage_bw": 100},
 "ops": [{"id": "t.train_step", "effector": "train_step",
          "on_refuse": "must_admit", "demands": []}, …],
 "schedule_head": [{"step": 1, "ops": ["t.train_step", "p.prefetch"], "demand": 44}, …]}
```

schedule_head shows the first 12 steps. Results are cached by (source sha256, capacity); cached responses set "cached": true.

9.3 `POST /api/simulate`

Compiles (as above) and resimulates the kernel's admission decisions with no-op effectors. Decisions are real; timings are not simulated and no I/O occurs.

Request:

© 2026 Layer1Labs · Confidential

```
{"source": "<lf source>", "capacity": 100,
 "policy": {"capacities": {"storage_bw": 60},
            "on_refuse": {"p.prefetch": "drop"}}}
```

The policy field is optional; when omitted, the compiled defaults apply. Response (200):

```
{"ok": true, "compile": {…as above…},
 "steps_completed": 200,
 "decisions": {"t.train_step": {"effector": "train_step", "admitted": 200,
               "refused": 0, "retried": 0, "dropped": 0}, …},
 "collision": {"step": 5, "capacity": 100, "demand": 169,
               "trace": […admission trace…],
               "next_step": {"step": 6, "trace": […]}}}
```

collision identifies the first step whose demand exceeds capacity, with the per-operation admission trace and the following step (showing the carried retry being admitted).

9.4 `POST /api/policy`

Returns the default rules for a compiled workload: capacities plus the per-template refuse policies, demands, and effector names. The demo page's rules editor loads this.

Request: {"source": "<lf source>", "capacity": 100}

9.5 `POST /api/run`

Runs the workload for real: real file-I/O effectors, timed, two arms back-to-back on the host in **randomized order** (reported as arm_order):

- baseline — effectively infinite capacity (the unmanaged system: admit everything, pay full contention).
- managed — your policy capacities (admission control on).

Request:

```
{"source": "<lf source>", "capacity": 100,
 "policy": {…optional, as in /api/simulate…},
 "io": {"ckpt_mb": 1, "prefetch_mb": 1, "fsync": true}}
```

io sizes are clamped to 1–16 MiB. Response (200):

```
{"ok": true, "arm_order": ["managed", "baseline"],
 "baseline": {"wall_seconds": 4.213, "steps_completed": 200, …},
 "managed": {"wall_seconds": 4.017, "steps_completed": 200, …}}
```

Each arm carries the full validated run report (§6.5). One run at a time per server (a second concurrent request gets 429 busy). This is a same-host, back-to-back measurement, not a controlled experiment: arm order is randomized, but cache, thermal, and allocator state still differ between arms.

9.6 `GET /files/<name>`

Serves the evaluation documents: GPU_AB_RECORD_RESULTS.pdf, CHRONOHIVE_STORAGE_INTEGRATION_GUIDE.pdf, CHRONOHIVE_REFERENCE_MANUAL.pdf, and the architecture diagram chronohive_architecture.png.

© 2026 Layer1Labs · Confidential

9.7 Status and error codes

HTTP	code	Meaning
400	bad_json	Malformed or non-object JSON body
400	bad_capacity	Capacity missing, out of 1–1,000,000, or not an integer
400	empty_source	No source field
400	bad_io	io sizes outside 1–16 MiB or malformed
413	body_too_large	Request body over 1 MiB
413	source_too_large	Source over 200,000 characters
422	compile_failed	lfc gate or lowering rejected the program (message included)
422	bad_policy	Unknown template, unknown policy, or bad capacity in policy
422	schedule_too_large	Schedule over the 5,000-step cap
429	busy	A real-I/O run is already in progress
500	toolchain_unavailable	chronoc or lfc not found on the server
500	compile_timeout	Compile exceeded 180 s
500	blob_invalid	Blob read-back failed
500	kernel_failed	Kernel execution failed
500	bad_report	Run report failed validation
500	internal	Anything else

9.8 Limits

Source 200,000 characters; body 1 MiB; capacity 1–1,000,000; 5,000 compile/run steps; 20-entry compile cache; one real-I/O run at a time.

---

10. Evaluation API reference

Base URL: https://api.layer1labs.ai. The evaluation API is the hosted admission-decision service used for scripted, vendor-neutral evaluation. **Storage backend, contention model, and vendor hardware surface are simulated; every grant/refuse decision comes from the real ChronoHive Runtime kernel.** Responses are stamped with honesty markers; numbers are simulation outcomes, not hardware measurements.

10.1 Authentication and the TOC flow

1. Authenticate with your API key: Authorization: Bearer <key>. 2. GET /v1/toc returns the canonical Terms of Confidentiality and its pinned sha256. 3. Sign it: python3 tools/sign_toc.py --api-url https://api.layer1labs.ai --key-id <your key id> --submit. The signature is verified against the pinned TOC hash before the key is activated. 4. Eval

© 2026 Layer1Labs · Confidential

endpoints stay locked until a verified TOC acceptance is on file for the key.

Quotas: 50 scenarios/day and 60 decisions/minute per key (defaults; per-key configurable). Quotas and scenario state reset on server restart; keys load at startup.

10.2 `POST /v1/scenarios`

Runs a scripted contention scenario through the kernel.

```
{"jobs": 8, "ckpt_gb": 200, "prefetch_gb": 150,
 "incast_alpha": 0.2, "seeds": [7]}
```

Response: a scenario_id plus per-seed results — baseline vs admission GPU stall seconds, checkpoint misses, stall reduction, and a pass/fail verdict per seed. GET /v1/scenarios/<id> retrieves a stored scenario (last 128 kept).

10.3 `POST /v1/admission/decide`

The live kernel decision path: submit transfer requests, get grant/refuse decisions.

```
{"now_s": 12.5, "window": 4,
 "requests": [{"id": "t1", "bytes": 73400320, "qos": "ckpt"}, …]}
```

Response: per-request grant / refuse decisions with the kernel's reasoning. This is the endpoint an external scheduler calls to arbitrate its own I/O.

10.4 `GET /v1/health`

Liveness check; no authentication required.

---

11. VS Code workflow — local end to end

This chapter is written so you can follow it on your own machine, in VS Code, and reproduce the full loop: edit the Lingua Franca source, compile it, and drive the demo API. It assumes the chronohive repository checked out locally.

11.1 Prerequisites

- **Java 17+** — required by the Lingua Franca toolchain.
- **Rust toolchain** — to build chronoc (cargo build).
- **Python 3.10+** — for the demo server and execute harness.
- **The pinned lfc 0.13.0** — tools/bin/install-lf-toolchain.sh fetches it,

or point --lfc / $DEMO_LFC at an existing binary.

- **VS Code** — plus the extension below.

11.2 Install the Lingua Franca extension

The official extension is lf-lang.vscode-lingua-franca (VS Marketplace and Open VSX). In VS Code, Quick Open (Ctrl+P / Cmd+P) and enter:

```
ext install lf-lang.vscode-lingua-franca
```

or from a terminal:

© 2026 Layer1Labs · Confidential

```
code --install-extension lf-lang.vscode-lingua-franca
```

The extension provides Lingua Franca syntax highlighting, hover, find references, folding, code validation as you edit, and target-code validation on build/save. (The extension's own Build/Run commands generate target code — ChronoHive does not use that path; you compile with chronoc instead.)

11.3 Open the project

```
git clone <chronohive-repo-url> chronohive
code chronohive
```

The record workload source lives at demo/lf/GpuAbWorkload.lf. Open it: the extension highlights reactors, ports, timers, reactions, and the {= =} target-code blocks.

11.4 Get chronoc

chronoc's source of truth is the layer1labs/chronohive-toolchain repository; this repo vendors the pinned binary (v0.1.0-alpha.2) — no Rust build is needed. In the VS Code terminal (Ctrl+):

```
scripts/fetch-chronoc.sh
```

The script verifies toolchain/chronoc-linux-x86_64 against the pin record in toolchain/README.md (SHA-256 + version) and rebuilds it from the pinned release only if the vendored copy is missing or mismatched.

11.5 Compile from the terminal

```
LF=demo/lf/GpuAbWorkload.lf
OUT=/tmp/gpuab.cspec
LFC=<path-to-lfc-0.13.0>

toolchain/chronoc-linux-x86_64 compile "$LF" -o "$OUT" \
    --mb-bw 1 --depth-bw 2 --capacity storage_bw=100 \
    --lfc "$LFC"
```

A successful compile prints the specification summary (templates, steps, bytes). Now break something on purpose to see the subset rules fire — change prefetch(depth=depth) to prefetch2(depth=depth) and recompile: you get an "unknown intrinsic" error (§3.8, S10). Change if step % ckpt_every == 0: to a nested if and you get "nested guards are not supported" (S14). Revert when done.

11.6 Recommended VS Code tasks

Create .vscode/tasks.json in the repo root:

© 2026 Layer1Labs · Confidential

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "chronoc: verify pinned binary",
      "type": "shell",
      "command": "scripts/fetch-chronoc.sh",
      "group": "build"
    },
    {
      "label": "chronoc: compile GpuAbWorkload.lf",
      "type": "shell",
      "command": "toolchain/chronoc-linux-x86_64 compile demo/lf/GpuAbWorkload.lf -o /tmp/gpuab.cspec --mb-bw 1 -
      "group": "build",
      "dependsOn": "chronoc: verify pinned binary"
    },
    {
      "label": "demo: serve locally",
      "type": "shell",
      "command": "python3 docs/demo/live/demo_server.py",
      "isBackground": true,
      "problemMatcher": []
    }
  ]
}
```

Run with Ctrl+Shift+P → "Tasks: Run Task". The compile task needs DEMO_LFC set to your lfc 0.13.0 binary (add it to your shell profile or a .env file).

11.7 Serve the demo locally

```
python3 docs/demo/live/demo_server.py
```

With no DEMO_USERS set, the server runs open on http://localhost:8765 (local development only). Open it in your browser: the full interactive page — the same editor, compile, rules, and real-I/O run as the hosted demo — now drives your local toolchain. The page's Act 3 editor compiles through /api/compile on your machine; Act 4's real-I/O run executes against your disk.

To require login locally (mirroring hosted):

```
DEMO_USERS="you:<password>" python3 docs/demo/live/demo_server.py
```

11.8 Drive the API from the terminal

Compile:

```
curl -s -u you:<password> http://localhost:8765/api/toolchain
```

© 2026 Layer1Labs · Confidential

```
python3 - <<'EOF'
import json, urllib.request
src = open("demo/lf/GpuAbWorkload.lf").read()
req = urllib.request.Request(
    "http://localhost:8765/api/compile",
    data=json.dumps({"source": src, "capacity": 100}).encode(),
    headers={"Content-Type": "application/json"})
# add HTTP basic auth header as needed
print(json.load(urllib.request.urlopen(req))["n_steps"])
EOF
```

Simulate admission decisions:

```
# POST /api/simulate  {"source": src, "capacity": 100}
# -> decisions per template + the first collision step with its trace
```

Run real I/O (unmanaged vs managed, randomized arm order):

```
# POST /api/run  {"source": src, "capacity": 100,
#                 "io": {"ckpt_mb": 1, "prefetch_mb": 1, "fsync": true}}
# -> {"arm_order": [...], "baseline": {…}, "managed": {…}}
```

11.9 Editing tips

- Keep the file name and the main reactor name in sync (§3.7, S2).
- The extension validates Lingua Franca as you type; chronoc enforces subset

v1 (§3.8) at compile time. A file can be valid LF and still be rejected by chronoc — that is the subset doing its job.

- Start from the presets: heavier checkpoints (ckpt_mb=140), deeper prefetch

(prefetch_depth=44), tighter capacity (60). Each change recompiles to a new blob; the kernel's decisions are recomputed from scratch.

---

12. Honesty boundaries and glossary

12.1 What is measured, what is decided, what is not simulated

- **The compile is real.** lfc 0.13.0 validates; chronoc lowers; the specification is

the artifact.

- **The admission decisions are real.** The resimulation runs the production

Runtime kernel with no-op effectors. Grant/refuse/carry/drop outcomes are exactly what the kernel decides.

- **The resimulation timings are not simulated.** No-op effectors do no I/O;

no timing model is applied.

- **The 11.6% result is the separately measured record GPU trial**: six

preregistered interleaved trials, RTX 5060 Ti, real ResNet-18 training, real checkpoint bytes through the timed write path.

- **The hosted real-I/O comparison is a same-host, back-to-back measurement**,

© 2026 Layer1Labs · Confidential

not a controlled experiment. Arm order is randomized; cache, thermal, and allocator state may differ.

12.2 Glossary

Term	Meaning
Blob (.cspec)	The compiled binary artifact: templates, schedule, capacities, provenance
Capacity	Integer budget per resource (e.g. storage_bw = 100)
Carried retry	A refused retry-policy operation re-proposed at later steps
chronoc	The Rust compiler: LF subset → blob
Demand	Units of a resource an operation needs (1 unit = 1 MiB checkpoint)
Driver loop	The one timer-driven reaction that bounds the schedule (§3.8, S8)
Effector	A customer-owned callable that performs an admitted operation's I/O
Intrinsic	One of the three recognized effector call sites (§3.8)
lfc	The Lingua Franca compiler; the pinned 0.13.0 validates before lowering
must_admit / retry / drop	The three refuse policies (§7.1)
Operation template	A lowered intrinsic call site with effector, demands, and policy
Subset v1	The preregistered Lingua Franca subset chronoc lowers (§3.8)
Template id	<instance>.<intrinsic>, e.g. c.admit_checkpoint

End of manual. For the integration path (where ChronoHive fits in your stack, adoption paths, migration plan), see the storage-workflow integration guide. For the measured evidence, see the record-trial report.

© 2026 Layer1Labs · Confidential