Render

Inference Gateway & Audit Trail System

Design Specification v2.1

fleek.sh

January 2026

Abstract

Design specification for Render, a high-throughput inference gateway supporting generative media (images, video) and large language models. The architecture employs STM-based concurrency for queue management, circuit breakers, and rate limiting; separates hot-path operational queries (ClickHouse) from cold-path compliance storage (Parquet on R2/S3). Configuration is expressed in Dhall with System Fω typing guarantees. The system is implemented in Haskell (Warp/grapesy, GHC 9.10→9.12) with formal queueing theory underpinning capacity planning.

Implementation status: Gateway STM core is complete. Triton backends are production: idoru for diffusers (FLUX, WAN 2.2), TRT-LLM for LLMs (Qwen3-235B at NVFP4). ClickHouse metrics are deployed. Straylight CAS (content-addressed audit with Lean4 proofs) deployed at `aleph-cas.fly.dev` with R2 backend; proto wiring and GPU attestation shipping Friday.

Contents

1 System Overview

1.1 Design Goals

1. **High throughput**: 10K+ concurrent connections, sub-100ms gateway latency
2. **Model heterogeneity**: Images, video, and LLMs with unified API
3. **STM concurrency**: Composable atomicity for queues, rate limiters, circuit breakers
4. **Type safety**: Dhall configuration (System Fω), Haskell implementation
5. **Cost attribution**: GPU billing at second granularity via NVXT traces
6. **Auditability**: Immutable trail with cryptographic integrity

1.2 Repository Map

This specification references artifacts across multiple repositories. Not all are public.

Component	**Repo**	**Description**	**Status**
Container build	`fleek-sh/nix2gpu`	NixOS containers for GPU clouds	Public
Init system	`weyl-ai/nimi`	PID1 + NixOS modular services	Public
GPU drivers	`weyl-ai/nvidia-sdk`	CUDA + driver integration	Public
Verified PS	`straylight-software/verified-purescript`	Lean4 proof extraction	Public
Gateway	`fleek-sh/render`	Haskell monolith (Warp/STM)	Private
Triton configs	`fleek-sh/render`	Model repos, TRT-LLM engines	Private
NativeLink CAS	`aleph-cas.fly.dev`	Content-addressed audit storage (R2)	Deployed
Armitage	`straylight/aleph`	CAS.hs + Lean4 attestation proofs	Friday

Table 1: Repository cross-reference. Private repos require access.

Registry namespace: Container images are published to `ghcr.io/weyl-ai/` (package namespace), while source code lives in `github.com/fleek-sh/` (code host).

Flake targets (defined in `fleek-sh/render`):

- `nix build .#render-gateway` — Haskell gateway binary
- `nix build .#render-triton-diffusers` — idoru backend container
- `nix build .#render-triton-llm` — TRT-LLM backend container
- `nix run .#<name>.copy-to-github` — Push to GHCR

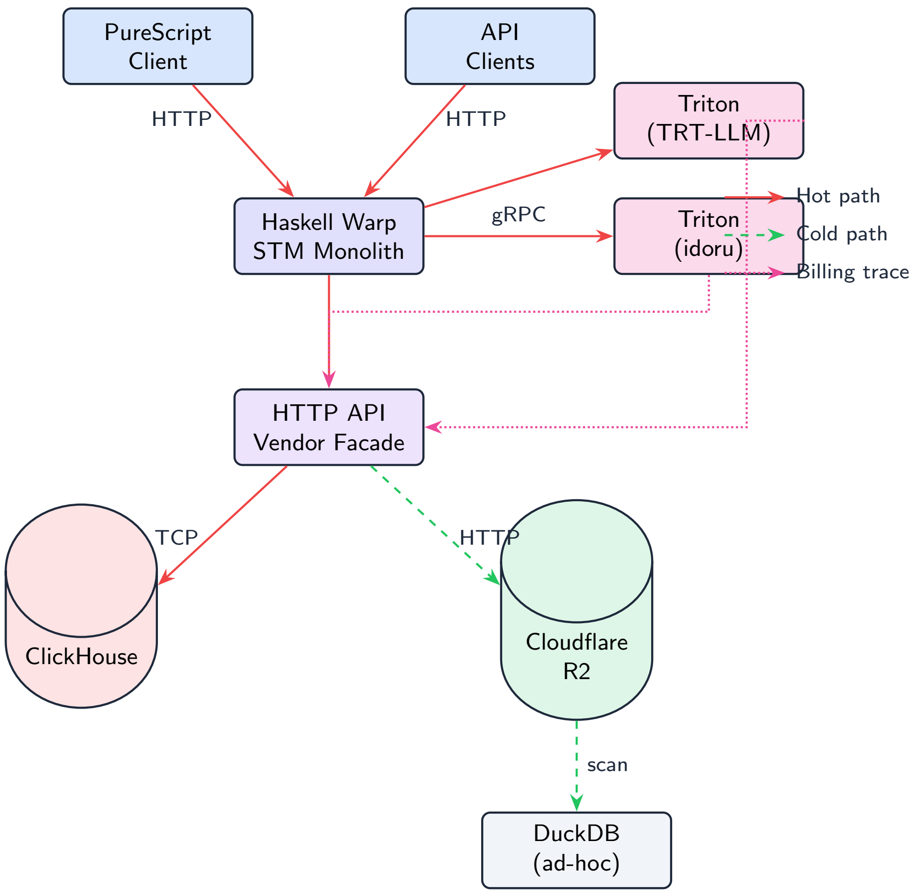


Figure 1: Current architecture: idoru + TRT-LLM backends, R2 for cold storage

1.3 Architecture Diagram

2 Supported Models

2.1 Generative Media (idoru backend, NVFP4)

idoru is an Inductor + TorchAO fork with NVFP4 quantization. **s4** (in-progress compiler stack) is roadmap.

2.2 Large Language Models (TRT-LLM at NVFP4)

All LLM inference uses **TensorRT-LLM** with NVFP4 quantization:

TRT-LLM configuration:

- In-flight batching for continuous throughput
- Paged KV cache for memory efficiency
- Tensor parallelism for large models (DeepSeek, CodeLlama-70B)
- NVFP4 quantization with $< 5\%$ quality loss on HumanEval

3 Type System & Configuration

3.1 Dhall Configuration (System Fω)

All configuration is expressed in Dhall, providing:

Family	Model	Tasks	Backend	Status
Image Models				
flux	dev2	t2i, i2i	idoru	Active
flux	dev	t2i, i2i	idoru	Active
flux	schnell	t2i, i2i	idoru	Active
zimage	turbo	t2i	idoru	Active
qwen	edit	t2i, i2i, edit	idoru	Preview
Video Models				
wan	default	i2v	idoru	Preview

Table 2: Generative media models served via Triton + idoru backend

Family	Model	Tasks	Params	Context	GPU
qwen	3-235b	complete, chat	235B MoE	32K	RTX PRO 6000
deepseek	v3	complete, chat	671B MoE	128K	H100×8 (planned)
qwen	coder-32b	complete, chat, infill	32B	32K	L4/L40S

Table 3: Large language models with TRT-LLM backend. Qwen3-235B is primary for Friday.

- Total evaluation (no runtime errors from config)
- Polymorphic types with higher-kinded type constructors
- Import system with integrity checking (sha256 hashes)

```
-- render/audit.dhall
let AuditConfig : Type =
  { storage : StorageBackend
  , retention : RetentionPolicy
  , billing : BillingConfig
  , compliance : ComplianceLevel
  }

let StorageBackend : Type =
  < R2 : { endpoint : Text, bucket : Text }          -- current
  | S3Compatible : { endpoint : Text, bucket : Text }
  | Straylight : { endpoint : Text, bucket : Text } -- roadmap
  >

let ComplianceLevel : Type =
  < Prudent    -- don't get sued
  | Auditable  -- answer to E&Y
  >
```

3.2 Vendor HTTP API Facade

The vendor facade abstracts storage operations behind a typed HTTP interface:

```
-- Dhall-derived OpenAPI spec
POST   /audit/batches                -- write batch
GET    /audit/batches/{hash}         -- retrieve by content hash
GET    /audit/manifest               -- get manifest
POST   /audit/manifest/append        -- append to manifest
GET    /audit/query                  -- ad-hoc DuckDB passthrough
```

Backend implementations:

- **Cloudflare R2**: Parquet files, content-addressed by hash (fast_slow store)
- **NativeLink CAS**: Deployed at `aleph-cas.fly.dev`, R2 backend, proto-lens interface
- **Armitage/CAS.hs**: Friday—coeffect tracking, GPU attestation, Lean4-verified proofs

4 STM Concurrency Architecture

4.1 Why STM?

The gateway manages shared mutable state across thousands of concurrent connections:

- Priority queues (high/normal/low)
- Per-customer rate limiters
- Backend circuit breakers
- Job state machines

STM provides **composable atomicity**:

```haskell
-- Compose queue, rate limiter, and backend selection atomically
processRequest :: STM (Maybe (Job, Backend))
processRequest = do
  job <- dequeueJob queue
  backend <- selectBackend backends (jobFamily job)
  case backend of
    Nothing -> requeueJob queue job >> retry  -- atomic rollback
    Just b  -> pure (Just (job, b))
```

4.2 Clock TVar Pattern

Time-dependent STM operations require a “clock TVar” updated by a background thread:

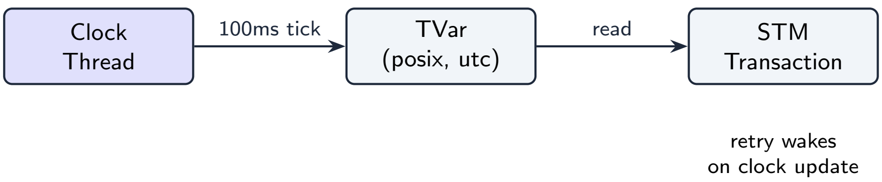


Figure 2: Clock TVar pattern enables time-dependent retry

```haskell
-- Rate limiting with clock-driven retry
acquireTokenBlocking :: RateLimiter -> Clock -> STM ()
acquireTokenBlocking rl clock = do
  (now, _) <- readClockSTM clock  -- read triggers retry wake
  acquired <- acquireToken rl now
  unless acquired retry           -- wakes on next 100ms tick
```

4.3 Priority Queue Design

Three-lane priority queue with O(1) operations:

```
data RequestQueue = RequestQueue
  { rqHigh   :: TQueue QueuedJob   -- Priority: High
  , rqNormal :: TQueue QueuedJob   -- Priority: Normal
  , rqLow    :: TQueue QueuedJob   -- Priority: Low
  , rqSize   :: TVar Int           -- Total count
  }

dequeueJob :: RequestQueue -> STM QueuedJob
dequeueJob RequestQueue{..} = do
  job <- readTQueue rqHigh
           `orElse` readTQueue rqNormal
           `orElse` readTQueue rqLow
  modifyTVar' rqSize (subtract 1)
  pure job
```

4.4 Circuit Breaker

Per-backend circuit breaker with rolling window statistics:

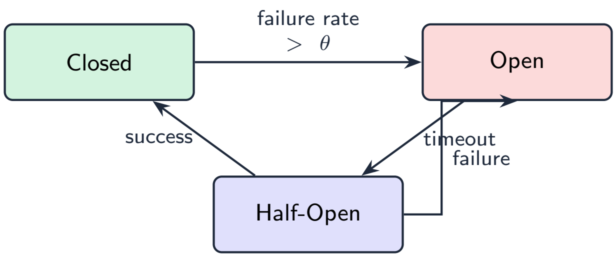


Figure 3: Circuit breaker state machine

```
selectBackend :: [Backend] -> Text -> Text -> UTCTime -> STM (Maybe
    Backend)
selectBackend backends family model now = do
  candidates <- forM (filter matchesRoute backends) $ \b -> do
    circuit <- readTVar (beCircuit b)
    load <- readTVar (beInFlight b)
    let available = case circuit of
          CircuitClosed -> load < beCapacity b
          CircuitHalfOpen -> load == 0  -- probe slot
          CircuitOpen t -> elapsed t > timeout && load == 0
    pure (b, load, available)

  case [(b, l) | (b, l, True) <- candidates] of
    [] -> pure Nothing
    xs -> do
      let (selected, _) = minimumBy (compare `on` snd) xs
      modifyTVar' (beInFlight selected) (+ 1)
      pure (Just selected)
```

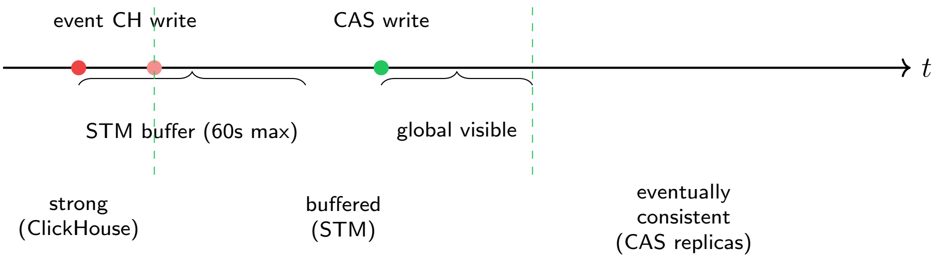


Figure 4: Consistency timeline: event to global visibility

Path	Latency	Consistency	Guarantee
ClickHouse	<10ms	Strong	Read-your-writes
STM buffer	0–60s	Session	Causally ordered
Cloudflare R2	1–5s	Eventual	Content-addressed (immutable)

Table 4: Consistency characteristics by storage tier

5 Consistency Model

5.1 Recency vs. Eventual Consistency

Key insight: Once written to R2, Parquet files are immutable and content-addressed by hash. "Eventual" only applies to replica propagation.

6 GPU Billing: Triton NVXT Plugin

6.1 Architecture

GPU billing requires sub-second attribution of compute to requests. Triton's NVTX (NVIDIA Tools Extension) markers provide the instrumentation point.

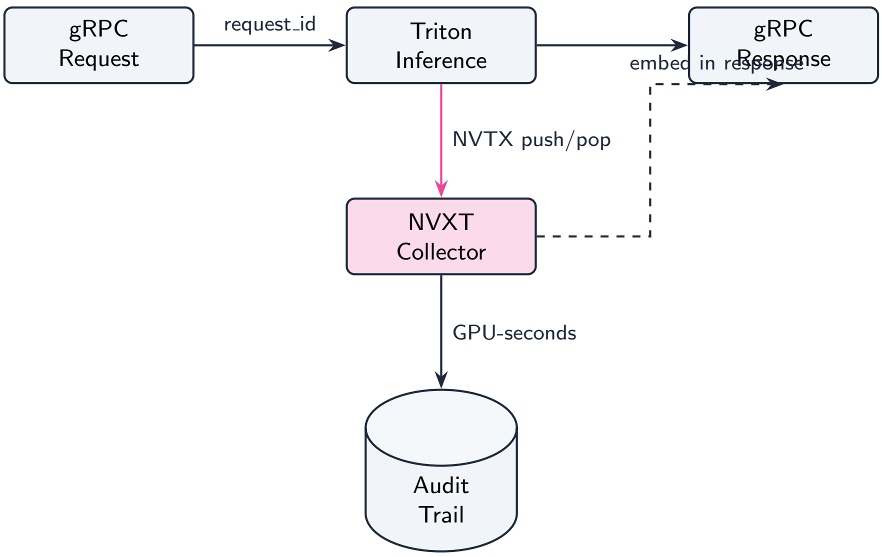


Figure 5: NVXT trace collection and response embedding

6.2 Plugin Implementation Sketch

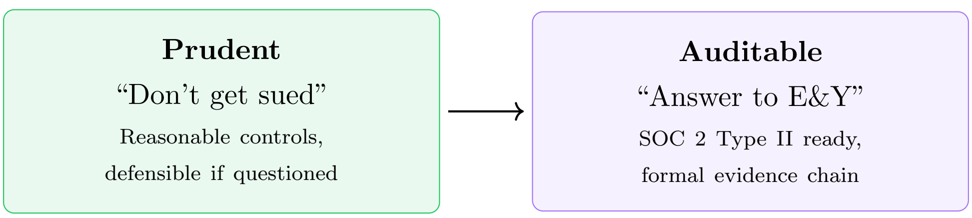
```cpp
// triton_nvxt_billing.cc
class NVXTBillingPlugin : public TritonPlugin {
public:
  void OnRequestStart(const Request& req) override {
    nvtxRangePushA(req.request_id().c_str());
    start_time_[req.request_id()] = CuptiGetTimestamp();
  }

  void OnRequestEnd(const Request& req, Response* resp) override {
    nvtxRangePop();
    auto elapsed_ns = CuptiGetTimestamp() - start_time_[req.request_id
        ()];
    auto gpu_seconds = elapsed_ns / 1e9;

    // Embed billing data in response metadata
    resp->mutable_parameters()->insert(
      {"x-gpu-seconds", std::to_string(gpu_seconds)});
    resp->mutable_parameters()->insert(
      {"x-gpu-device", GetDeviceUUID()});

    // Async flush to audit trail
    audit_queue_.Push(BillingRecord{
      .request_id = req.request_id(),
      .gpu_seconds = gpu_seconds,
      .device_uuid = GetDeviceUUID(),
      .model_name = req.model_name(),
      .timestamp = Now()
    });
  }
};
```

6.3 Billing Record Schema

```
message BillingRecord {
  required binary request_id (UUID);
  required double gpu_seconds;
  required binary device_uuid (STRING);
  required binary model_name (STRING);
  required int64 timestamp_us (TIMESTAMP(MICROS, true));
  optional binary customer_id (STRING);
  optional binary pricing_tier (STRING);
}
```

7 Compliance Gap Analysis

7.1 Two Compliance Tiers



7.2 Gap Matrix

Control	Prudent	Auditable (E&Y)	Gap
Data Integrity			
Immutability	Content-addressed storage	+ Object Lock / WORM	✓ Low
Tamper detection	BLAKE3 hash chain	+ HSM-signed manifests	• Medium
Chain of custody	Internal audit log	+ Third-party attestation	■ High
Access Control			
Write segregation	IAM roles (app/audit)	+ Hardware security keys	✓ Low
Read audit	Application logs	+ SIEM integration	• Medium
Privileged access	Role-based	+ PAM with session recording	■ High
Retention & Recovery			
Retention policy	7 years cold storage	+ Legal hold automation	✓ Low
Backup verification	Hash verification	+ Periodic restore drills	• Medium
Geographic redundancy	Single region	+ Multi-region + DR site	■ High
Monitoring & Alerting			
Anomaly detection	Threshold alerts	+ ML-based behavioral	• Medium
Incident response	Runbooks	+ Documented IR plan	✓ Low
Evidence preservation	Log retention	+ Forensic imaging SOP	• Medium
Documentation			
System documentation	README + Dhall types	+ Formal policies	✓ Low
Change management	Git history	+ CAB approval process	• Medium
Risk assessment	Informal	+ Annual formal assessment	■ High

Table 5: Compliance gap analysis: Prudent baseline vs. E&Y audit readiness

Legend: ✓ Low effort — • Medium effort — ■ High effort

7.3 Prudent Baseline (Current Architecture)

What you get out of the box with R2 + ClickHouse + Dhall config:

- **Immutable audit trail**: Content-addressed Parquet files cannot be modified
- **Cryptographic integrity**: BLAKE3 hash chain detects tampering
- **Type-safe configuration**: Dhall eliminates config drift class of errors
- **Segregated access**: Separate write/read IAM principals (R2 API tokens)
- **Queryable history**: DuckDB ad-hoc access without standing infrastructure
- **GPU billing attribution**: Request-level compute metering

Prudent posture: If sued, you can demonstrate reasonable controls, reconstruct any transaction, and show that tampering would be detectable. This is sufficient for most operational contexts and early-stage compliance.

7.4 E&Y Audit Readiness: Key Gaps

1. **Third-party attestation**: Auditors want independent verification. Options:
   - Integrate with transparency log (e.g., Sigstore Rekor)
   - Periodic attestation by external party to manifest hashes
   - Lean4 proofs in Armitage provide cryptographic attestation (advanced)
2. **HSM-backed signing**: Current Ed25519 signing uses software keys. Upgrade path:
   - AWS CloudHSM / GCP Cloud HSM for manifest signing
   - YubiHSM for smaller deployments
   - Key ceremony documentation
3. **Privileged Access Management**: Console access to production systems:
   - Session recording (e.g., Teleport, Boundary)
   - Just-in-time access with approval workflows
   - Quarterly access reviews
4. **Formal Risk Assessment**: Annual documented assessment covering:
   - Threat modeling
   - Control effectiveness evaluation
   - Residual risk acceptance (signed by leadership)

7.5 Estimated Effort

From	To	Effort
Nothing	Prudent	2–4 weeks (current architecture)
Prudent	Auditable (SOC 2 Type I)	2–3 months + $50–100k
SOC 2 Type I	SOC 2 Type II	6–12 months observation period

Table 6: Compliance progression effort estimates

8 Straylight CAS Integration

Status: Friday. NativeLink CAS is deployed at `aleph-cas.fly.dev` with Cloudflare R2 backend. Armitage/CAS.hs provides the Haskell interface via proto-lens + grapesy. GPU attestation (signed proofs to CAS) and ClickHouse↔CAS reconciliation ship Friday. Lean4 continuity proofs complete at `straylight/aleph/docs/rfc/aleph-008-continuity/`.

8.1 Coeffect Algebra

Straylight CAS provides stronger guarantees than commodity object storage through coeffect tracking. Each stored object carries:

- **Content hash**: BLAKE3 of serialized bytes
- **Coeffect annotation**: What resources were required to produce it

- **Discharge proof**: Evidence that coeffects were satisfied
- **Lean4 verification**: Machine-checkable proof of algebraic properties

```
-- Coeffect-annotated audit record
data AuditRecord (r :: Resource) where
  MkAuditRecord
    :: { content     :: ByteString
       , coeffect    :: Sing r              -- resource requirement
       , discharge   :: Discharge r         -- proof of satisfaction
       , signature   :: Ed25519Sig
       } -> AuditRecord r

-- The type system ensures:
-- if you have (AuditRecord r), the coeffect r was discharged
```

8.2 Why This Matters for Compliance

Traditional audit: "Trust me, this is the authentic record."

Straylight audit: "Here's the record, here's the coeffect declaration, here's the discharge proof, here's the Lean4 kernel verification, here's the cryptographic signature."

The attestation is machine-checkable. For E&Y, this shifts the audit from "examine the process" to "verify the proof"—a much stronger posture.

9 ClickHouse Schema Design

9.1 Design Goals

1. **Operator affordance**: Internal dashboards for capacity planning, incident response
2. **Customer dashboards**: Per-tenant metrics with sub-second refresh
3. **Best-effort real-time**: Inference metrics visible within seconds, not minutes
4. **Model heterogeneity**: Support both LLM (autoregressive, variable-length) and rectified flow (fixed-step diffusion) workloads

9.2 Core Tables

9.2.1 Inference Events (Raw)

```
CREATE TABLE inference_events (
    event_id          UUID,
    timestamp         DateTime64(6, 'UTC'),
    customer_id       LowCardinality(String),
    model_name        LowCardinality(String),
    model_type        Enum8('llm' = 1, 'rectified_flow' = 2, 'other' =
       3),

    -- Request characteristics
    request_id        UUID,
    input_tokens      UInt32,              -- LLM: prompt tokens
    output_tokens     UInt32,              -- LLM: completion tokens
    input_dims        Array(UInt32),       -- Flow: input tensor shape
    num_steps         UInt16,              -- Flow: diffusion steps
```

```sql
    -- Timing (microseconds)
    queue_time_us    UInt64,
    inference_time_us UInt64,
    total_time_us    UInt64,

    -- Resource attribution
    gpu_seconds      Float64,
    device_uuid      LowCardinality(String),
    batch_size       UInt16,

    -- Status
    status           Enum8('success' = 1, 'error' = 2, 'timeout' = 3),
    error_code       Nullable(String)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMMDD(timestamp)
ORDER BY (customer_id, model_name, timestamp)
TTL timestamp + INTERVAL 7 DAY TO VOLUME 'cold',
    timestamp + INTERVAL 90 DAY DELETE
SETTINGS index_granularity = 8192;
```

9.2.2 Metrics Rollups (Materialized)

```sql
CREATE MATERIALIZED VIEW metrics_1m
ENGINE = SummingMergeTree()
PARTITION BY toYYYYMMDD(window_start)
ORDER BY (customer_id, model_name, window_start)
AS SELECT
    customer_id,
    model_name,
    model_type,
    toStartOfMinute(timestamp) AS window_start,

    -- Counts
    count() AS request_count,
    countIf(status = 'success') AS success_count,
    countIf(status = 'error') AS error_count,

    -- Latency aggregates (for percentile estimation)
    sum(total_time_us) AS total_latency_us,
    sum(total_time_us * total_time_us) AS total_latency_sq,  -- for
        stddev
    min(total_time_us) AS min_latency_us,
    max(total_time_us) AS max_latency_us,

    -- Throughput
    sum(input_tokens) AS total_input_tokens,
    sum(output_tokens) AS total_output_tokens,
    sum(gpu_seconds) AS total_gpu_seconds,

    -- Queue health (Little's Law inputs)
    sum(queue_time_us) AS total_queue_time_us,
    max(queue_time_us) AS max_queue_time_us

FROM inference_events
GROUP BY customer_id, model_name, model_type, window_start;
```

9.2.3 Operator Aggregates (Hourly)

```sql
CREATE MATERIALIZED VIEW operator_metrics_1h
ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(window_start)
ORDER BY (device_uuid, model_name, window_start)
AS SELECT
    device_uuid,
    model_name,
    toStartOfHour(timestamp) AS window_start,

    -- Utilization
    sum(gpu_seconds) AS gpu_seconds_consumed,
    count() AS total_requests,
    uniqExact(customer_id) AS unique_customers,

    -- Capacity signals
    max(batch_size) AS max_batch_observed,
    avg(batch_size) AS avg_batch_size,

    -- Error budget
    countIf(status = 'error') AS errors,
    countIf(status = 'timeout') AS timeouts

FROM inference_events
GROUP BY device_uuid, model_name, window_start;
```

9.3 Quantile Estimation

For percentile dashboards without storing all values, use t-digest:

```sql
CREATE MATERIALIZED VIEW latency_quantiles_1m
ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMMDD(window_start)
ORDER BY (customer_id, model_name, window_start)
AS SELECT
    customer_id,
    model_name,
    toStartOfMinute(timestamp) AS window_start,
    quantilesTDigestState(0.5, 0.9, 0.95, 0.99)(total_time_us) AS
        latency_tdigest
FROM inference_events
GROUP BY customer_id, model_name, window_start;

-- Query p99:
SELECT
    quantilesTDigestMerge(0.99)(latency_tdigest) AS p99_us
FROM latency_quantiles_1m
WHERE customer_id = 'cust_123'
  AND window_start >= now() - INTERVAL 1 HOUR;
```

Option	Pros	Cons	When
Self-hosted (k8s)	Full control, no egress costs, co-located with Triton	Operational burden, tuning expertise required	High volume, cost-sensitive
ClickHouse Cloud	Managed, auto-scaling, separation of storage/compute	Egress costs, vendor dependency	Faster time-to-market
Hybrid	Hot tier self-hosted, cold tier in cloud	Complexity	Large scale with variable load

Table 7: ClickHouse hosting tradeoffs

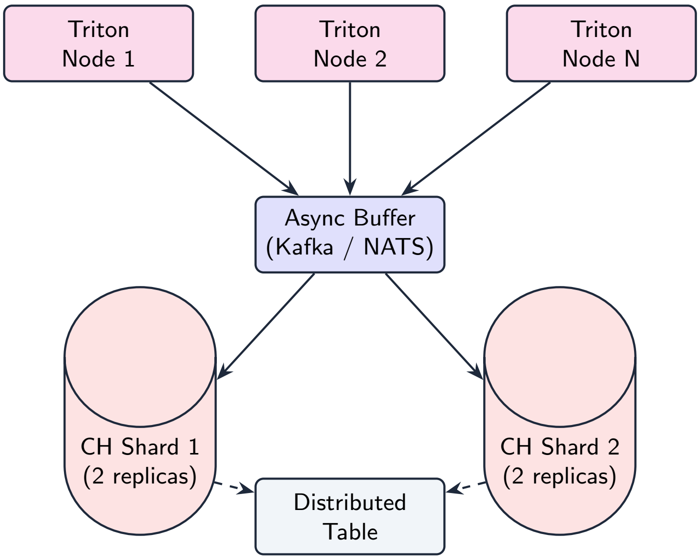


Figure 6: Sharded ClickHouse with async ingestion buffer

10 ClickHouse Hosting Strategy

10.1 Deployment Options

10.2 Recommended Architecture

Key decisions:

- **2 shards, 2 replicas each**: Balances write throughput with query fan-out. Scale shards for write volume, replicas for read throughput and HA.
- **Async buffer**: Decouples Triton latency from ClickHouse availability. Kafka or NATS JetStream. Provides backpressure without blocking inference.
- **Shard key**: `cityHash64(customer_id) % 2` for even distribution and customer-local queries.

10.3 Resource Sizing

For 10K inference requests/second baseline:

Component	Spec	Rationale
CPU per shard	16 cores	MergeTree background merges
RAM per shard	64 GB	Mark cache + query buffers
Storage per shard	2 TB NVMe	90-day hot retention
Network	10 Gbps	Replication + query traffic

Table 8: Per-shard resource baseline at 10K req/s

11 Fast Path / Slow Path Reconciliation

11.1 Dual-Write Architecture

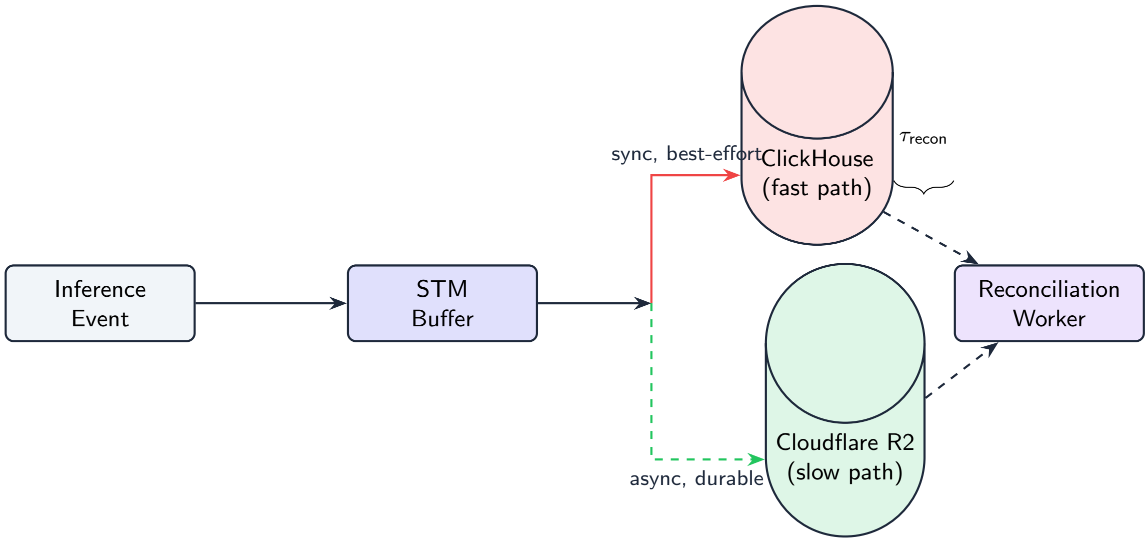


Figure 7: Dual-write with reconciliation window τ_{recon}

11.2 Path Characteristics

Property	Fast Path (ClickHouse)	Slow Path (R2)
Latency to visible	10–100ms	1–60s (batch flush)
Durability	Best-effort (async replication)	Guaranteed (content-addressed)
Query interface	SQL, real-time	DuckDB, ad-hoc
Retention	90 days hot	7+ years cold
Failure mode	Data loss on cluster failure	Delayed visibility
Use case	Dashboards, alerts	Audit, billing, compliance

Table 9: Fast vs. slow path tradeoffs

11.3 Reconciliation Window Tradeoffs

The reconciliation window τ_{recon} is the time after which the slow path becomes the authoritative source and fast path data can be validated/corrected.

Factors influencing τ_{recon}:

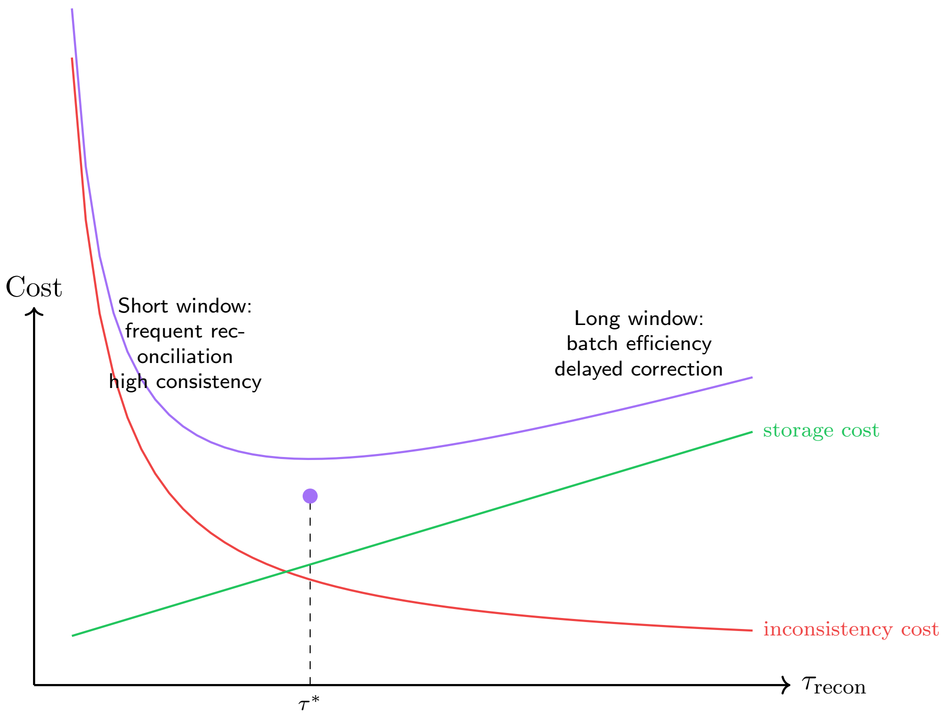


Figure 8: Reconciliation window optimization: $\tau^* \approx 5$ minutes balances consistency and efficiency

- **Billing cycles**: If customers see real-time usage, discrepancies cause support tickets. Shorter τ reduces confusion.
- **Batch efficiency**: Larger Parquet files compress better. Longer τ improves storage efficiency.
- **Incident response**: During outages, how quickly must you know ground truth? Shorter τ aids debugging.
- **Regulatory requirements**: Some compliance regimes mandate maximum delay to authoritative record.

Recommended: $\tau_{\text{recon}} = 5$ minutes for inference metrics. Reconciliation worker runs continuously, comparing ClickHouse aggregates against CAS batch manifests, flagging discrepancies $> 0.1\%$.

11.4 Reconciliation Procedure

```
reconcile :: TimeRange -> IO ReconciliationReport
reconcile range = do
    -- Aggregate from fast path
    chCounts <- queryClickHouse $
        "SELECT customer_id, model_name, count(*) as n " <>
        "FROM inference_events WHERE timestamp IN range " <>
        "GROUP BY customer_id, model_name"

    -- Aggregate from slow path (authoritative)
    casCounts <- queryDuckDB $
        "SELECT customer_id, model_name, count(*) as n " <>
        "FROM read_parquet('s3://audit/...') " <>
```

```
        "WHERE␣timestamp␣IN␣range␣GROUP␣BY␣1,␣2"

    -- Compute deltas
    let deltas = Map.differenceWith compareCounts chCounts casCounts

    -- Report discrepancies > threshold
    forM_ (Map.filter (> threshold) deltas) $ \delta ->
        alert $ DiscrepancyDetected range delta

    pure $ ReconciliationReport { range, deltas, status = ... }
```

12 Queueing Theory: Capacity Planning

12.1 Poisson Arrival Assumption

Inference requests arrive as a Poisson process with rate λ (requests/second). This assumption holds when:

- Many independent customers
- No temporal coordination between requests
- Stationary demand (within analysis window)

Validation: Plot inter-arrival times; should be exponentially distributed. For bursty workloads (batch inference jobs), use a compound Poisson or consider the superposition of multiple Poisson streams.

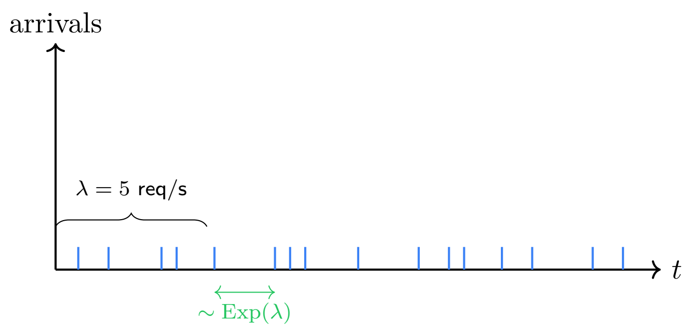


Figure 9: Poisson arrivals: inter-arrival times are exponentially distributed

12.2 Little's Law

The fundamental relationship for any stable queueing system:

$$\boxed{L = \lambda W} \tag{1}$$

Where:

- L = average number of requests in system (queue + being served)
- λ = arrival rate (requests/second)
- W = average time in system (queue time + service time)

No distributional assumptions required—holds for any stationary process.

12.3 Application to Inference Queue

12.3.1 Buffer Sizing

For the STM buffer between Triton and ClickHouse:

$$\begin{aligned} L_{\text{buffer}} &= \lambda_{\text{events}} \cdot W_{\text{flush}} && (2) \\ &= 10{,}000 \text{ req/s} \cdot 1 \text{ s (flush interval)} && (3) \\ &= 10{,}000 \text{ events} && (4) \end{aligned}$$

With safety margin (2× for bursts):

$$\text{Buffer capacity} = 20{,}000 \text{ events} \approx 20{,}000 \times 500\text{B} = 10 \text{ MB}$$

12.3.2 GPU Queue Depth

For Triton's inference queue, let μ = service rate (inferences/second per GPU):

$$\rho = \frac{\lambda}{\mu} \quad \text{(utilization)} \tag{5}$$

$$L_q = \frac{\rho^2}{1-\rho} \quad \text{(M/M/1 queue length)} \tag{6}$$

$$W_q = \frac{L_q}{\lambda} = \frac{\rho}{\mu(1-\rho)} \quad \text{(queue wait time)} \tag{7}$$

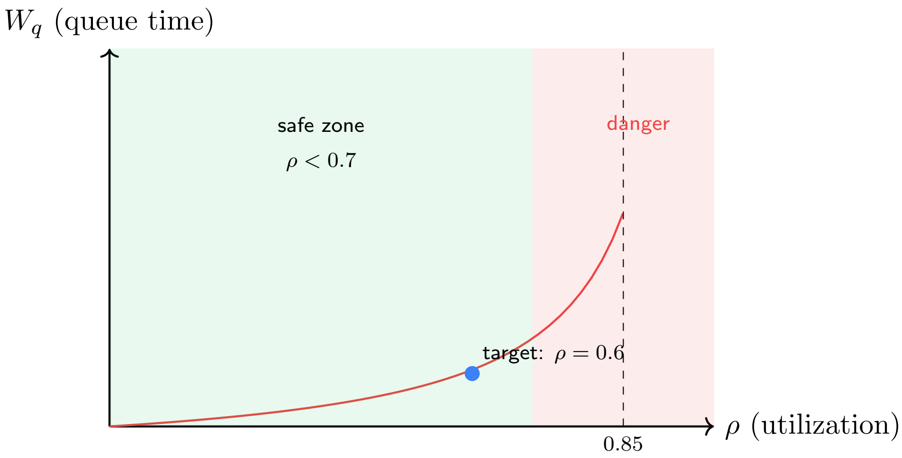


Figure 10: Queue wait time explodes as utilization approaches 1. Target $\rho < 0.7$ for predictable latency.

12.3.3 Capacity Planning Formula

Given target p99 latency W_{99} and arrival rate λ:

$$\text{Required GPUs} = \left\lceil \frac{\lambda}{\mu \cdot \rho_{\text{target}}} \right\rceil \tag{8}$$

Example: 1000 LLM requests/sec, each taking 200ms ($\mu = 5$/s/GPU), target $\rho = 0.6$:

$$\text{GPUs} = \left\lceil \frac{1000}{5 \times 0.6} \right\rceil = \lceil 333 \rceil = 334 \text{ GPUs}$$

12.4 Model-Specific Considerations

Property	LLM (Autoregressive)	Rectified Flow (Diffusion)
Service time distribution	Heavy-tailed (output length varies)	Near-deterministic (fixed steps)
Batching efficiency	Dynamic batching, in-flight batching	Static batching, predictable
Queue model	M/G/k (general service)	M/D/k (deterministic service)
Variance	High ($C_s^2 > 1$)	Low ($C_s^2 \approx 0$)
Implication	Need more headroom, $\rho_{\text{target}} \leq 0.5$	Can run hotter, $\rho_{\text{target}} \leq 0.7$

Table 10: Queueing characteristics by model type

For LLMs with high service time variance, use the Pollaczek-Khinchine formula:

$$W_q = \frac{\rho(1 + C_s^2)}{2\mu(1 - \rho)} \tag{9}$$

Where $C_s^2 = \mathrm{Var}(S)/E[S]^2$ is the squared coefficient of variation of service time. For LLMs, $C_s^2 \approx 2$–4 is common, doubling or tripling queue times versus deterministic service.

12.5 Dashboard Metrics (Derived from Theory)

```sql
-- Real-time queueing health dashboard
SELECT
    model_name,
    -- Arrival rate (requests/sec, 1-minute window)
    count() / 60.0 AS lambda,

    -- Service rate (inferences/sec/GPU)
    count() / (sum(gpu_seconds) + 0.001) AS mu,

    -- Utilization
    sum(gpu_seconds) / (60.0 * gpu_count) AS rho,

    -- Little's Law check: L = lambda * W
    avg(queue_time_us + inference_time_us) / 1e6 AS W_observed,
    count() / 60.0 * avg(queue_time_us + inference_time_us) / 1e6 AS
       L_predicted,

    -- Queue health
    quantile(0.99)(queue_time_us) / 1e6 AS queue_p99_sec,

    -- Alert threshold
    CASE
        WHEN sum(gpu_seconds) / (60.0 * gpu_count) > 0.7 THEN 'WARNING
           '
        WHEN sum(gpu_seconds) / (60.0 * gpu_count) > 0.85 THEN '
           CRITICAL'
        ELSE 'OK'
    END AS capacity_status

FROM inference_events
WHERE timestamp > now() - INTERVAL 1 MINUTE
```

```
GROUP BY model_name;
```

13 Hosting Infrastructure

13.1 Multi-Cloud Strategy

Tier	Provider	Workload	Notes
Control Plane	fly.io	Haskell gateway (Warp)	Global edge, easy deploys
Critical Infra	latitude.sh	ClickHouse Keeper	Bare-metal NixOS
GPU Primary	GCE	Triton backends	DWS Flex Start, credits
GPU Overflow	vast.ai/RunPod	Burst capacity	nix2gpu containers
CDN/Security	Cloudflare	R2, WAF, caching	Edge + storage

Table 11: Hosting tier allocation

13.2 Inference Backends

13.2.1 Diffusers (idoru / s4)

Two backend generations for image/video, both at NVFP4:

- **idoru**: Inductor + TorchAO fork, production-ready (FLUX, WAN)
- **s4**: In-progress compiler stack, next-generation throughput

13.2.2 LLMs (TRT-LLM)

All LLM inference uses **TensorRT-LLM** at NVFP4:

```
services."triton-llm" = {
  process.argv = [
    "${pkgs.tritonserver}/bin/tritonserver"
    "--model-repository=/models/trtllm"
    "--backend-config=tensorrtllm,batching_type=inflight"
  ];
};
```

13.3 GCE Dynamic Workload Scheduler

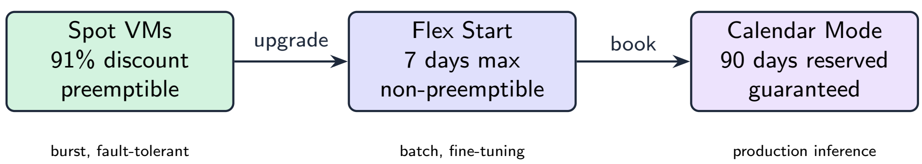


Figure 11: GCE provisioning modes for GPU workloads

Key insight: DWS Flex Start uses *preemptible quota* (usually much higher than on-demand) but provides non-preemptible guarantees once running—ideal for inference that can tolerate startup delay.

13.4 fly.io for Gateway

The Haskell monolith runs on fly.io:

- Anycast IPs with global edge routing
- `fly deploy` from CI
- Automatic TLS, managed Postgres (optional)
- Horizontal scaling: `fly scale count 4 --region iad`

13.5 latitude.sh for Critical Infrastructure

Bare-metal NixOS for stateful services:

```
-- ClickHouse Keeper (3-node quorum)
services.clickhouse-keeper = {
  enable = true;
  settings.server_id = 1;
  settings.coordination_settings = {
    operation_timeout_ms = 10000;
    session_timeout_ms = 30000;
  };
};

-- Tailscale for secure mesh
services.tailscale.enable = true;
networking.firewall.trustedInterfaces = [ "tailscale0" ];
```

13.6 Cloudflare Integration

- **R2**: Audit trail storage (Parquet), model artifacts
- **CDN**: Content-addressed assets with 1-year cache
- **WAF**: Rate limiting, bot protection
- **Zero egress**: R2 → CDN is free

14 Service Discovery & Coordination

14.1 PXE/iPXE Boot Infrastructure

Nodes boot from network via PXE, chainload iPXE for flexible boot scripting, and automatically join the Tailscale mesh.

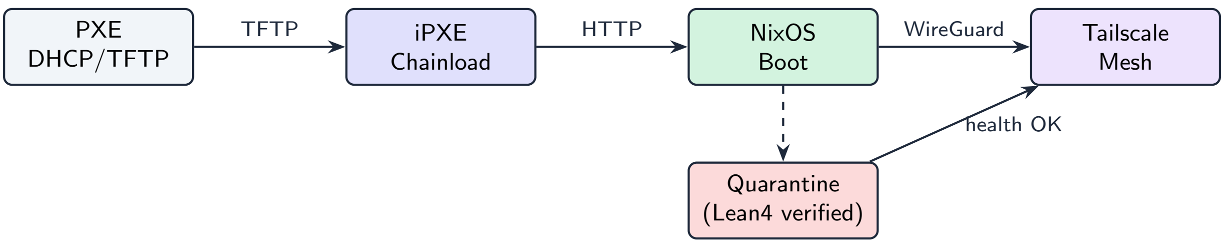


Figure 12: PXE boot flow with Lean4-verified quarantine

14.2 Tailscale Service Discovery

Lightweight service discovery via Tailscale tags (v1). No external coordinator required.

```
-- Tailscale ACLs for service mesh
{
  "tagOwners": {
    "tag:backend": ["autogroup:admin"],
    "tag:gateway": ["autogroup:admin"],
    "tag:infra":   ["autogroup:admin"]
  },
  "acls": [
    {"action": "accept", "src": ["tag:gateway"],
     "dst": ["tag:backend:8001"]},                -- gRPC to Triton
    {"action": "accept", "src": ["tag:gateway"],
     "dst": ["tag:infra:8123,9000"]},             -- ClickHouse, CAS
    {"action": "accept", "src": ["tag:infra"],
     "dst": ["tag:infra:*"]}                      -- infra mesh
  ]
}
```

Tags:

- `tag:backend` — GPU inference backends (tritonserver)
- `tag:gateway` — API gateways (openai-proxy-hs)
- `tag:infra` — Infrastructure (ClickHouse, NativeLink CAS)

14.3 Quarantine Proof (Lean4)

New nodes are isolated until health check passes. The quarantine invariant is formally verified in Lean4:

```
-- straylight/aleph/docs/rfc/aleph-008-continuity/continuity.lean
theorem quarantine_isolation (n : Node) (h : n.state = .quarantine) :
    forall m : Node, m.state = .active -> !canCommunicate n m := by
  intro m hm
  simp [canCommunicate, h, hm]
  decide
```

Friday scope: Lean4 quarantine proof complete, service registration via Tailscale tags, health propagation over mesh.

v2 (post-alpha): ClickHouse Keeper for distributed consensus and stronger coordination guarantees.

15 Summary

Render — Current State (January 2026)

Gateway	openai-proxy-hs (Warp/STM) on bizon
LLM	Qwen3-235B at NVFP4 on RTX PRO 6000 Blackwell (128GB)
Diffusers	FLUX + WAN 2.2 via idoru (Inductor + TorchAO)
Hot path	ClickHouse with X-Weyl-* headers
Cold path	NativeLink CAS at `aleph-cas.fly.dev` (R2 backend)
Networking	Tailscale mesh with PXE/iPXE boot
Containers	nix2gpu + Nimi + Isospin (<1s VM boot)

Render — Friday Ship (January 31, 2026)

DNS	api.fleek.ai → bizon
Auth	API key middleware (`flk_...`)
Rate limits	X-RateLimit-* headers
Chat UI	PureScript/Halogen with Radix Pure components
CAS wiring	Armitage/CAS.hs proto-lens → grapesy
Attestation	GPU-seconds signed and attested to CAS
Quarantine	Lean4-verified node isolation proof

Render — Post-Alpha Roadmap

s4 compiler	Next-gen diffuser backend (2× throughput target)
DeepSeek-V3	671B MoE on H100×8
Billing	GPU-seconds via Stripe integration
ClickHouse Keeper	v2 coordination (replaces Tailscale-only discovery)
WebSocket	Streaming via WebSocket (SSE fallback)

`github.com/fleek-sh/render` (private)

`github.com/fleek-sh/nix2gpu` — `github.com/weyl-ai/nimi` (public)