Shunyaya Symbolic Mathematical AI (SSM-AI)

Alignment lane beside AI decisions (values unchanged).

Status: Public Research Release (v1.8)
Date: 27 October 2025
Caution: Research/observation only. Use for verification, evaluation, and non-safety-critical prototyping.

(All formulas are plain ASCII. Classical collapse is `phi((m,a)) = m` throughout.)

SECTION 0 — Executive Summary

---

00 — What if…

What if every answer, retrieval, and tool call could also show how sturdy it is — so teams act faster and safer without changing a single line of product logic?

→ **Yes.** SSM-AI adds a bounded lane beside any AI value: `x := (m, a)` with `a in (-1,+1)` and `phi((m,a)) = m`.

You keep **all magnitudes and logic exactly as-is**; you gain a **visible, auditable** stability lane and **band at a glance**.

Order-invariant streaming uses:
`U += w*atanh(a)  W += w  a_out := tanh( U / max(W, eps_w) )`
with defaults `w := |m|^gamma` (`gamma = 1`), `eps_w := 1e-12`, and **clamp-safety on all inputs**.

It **drops in** to decoding, RAG, tools, evaluators, and ensembles **without retraining**.

What if teams could move from idea → pilot quickly, and pilot → portfolio predictably — because checks are calculator-fast, logs are stamped and replayable, and semantics remain identical across vendors and accelerators?

→ Yes. A **clamp-first kernel with frozen knobs** makes validation straightforward: **run quick invariance tests, lock a manifest, A/B with stamped CSVs, and replicate anywhere.**

Throughput paths map the **same equations** into **streaming accumulators or silicon tiles — no semantic drift**.

What if organizations could cut AI spend yet raise quality by **publishing the lane beside a few KPIs per service, banding schedulers**, and **stamping outputs — with no model churn**?

→ Yes. Expect **steadier answers, calmer loops, fewer retries, cleaner audits,** and **transparent vendor comparisons**.

Start simply: (1) **publish the lane** read-only, (2) **add bands** `A++ / A+ / A0 / A- / A--`, (3) **stamp outputs**.

Scale **service-by-service** with identical knobs and **zero code changes**.

Economics stay explicit; numbers remain identical while confidence becomes **bounded and auditable**.

---

01 — SSM-AI at a Glance (normative kernel)

- **Numeral and collapse** — `x := (m, a)`; `a in (-1,+1)`; `phi((m,a)) = m`.
- **Clamp** — `a := clamp(a, -1+eps_a, +1-eps_a)`; default `eps_a = 1e-6`.
- **Rapidity map** — `u := atanh(a)`; inverse `a := tanh(u)`.
- **Streaming fuse (order-invariant)** —
  `U += w*atanh(a)  W += w  a_out := tanh( U / max(W, eps_w) )`
  defaults `w := |m|^gamma` (`gamma = 1`), `eps_w := 1e-12`.
- **Product / ratio on the lane (M2 rule)** — multiply/divide `m` as usual; lanes compose `a' := tanh(atanh(a1) +/- atanh(a2))`.
- **Bands (defaults)** — **publish once and keep consistent:**
  `A++: a >= +0.90`, `A+: +0.60 <= a < +0.90`, `A0: -0.60 < a < +0.60`, `A-: -0.90 < a <= -0.60`, `A--: a <= -0.90`.
- **Governed actuation (alignment-only)** — `a_env := g_t*a_op` or `RSI_env := g_t*RSI`, `g_t in [0,1]`; **value lane remains unchanged**.

---

02 — Where It Fits (no retraining, immediate value)

- **Decoding & rerank:** Add `(m,a)` per candidate; select with a **bounded chooser** (e.g., `RSI`); log bands to see **fragile tokens**.
- **RAG:** Attach `(m,a)` to passage scores and citations; **band schedulers** when evidence thins; improve **first-answer correctness** without changing retrieval math.
- **Tools & agents:** Emit `(m,a)` at each step; use **bands to temper branching** and reduce retries; **stamp traces for replay**.
- **Evaluators & ensembles:** Pool lanes via **streaming fuse**; keep magnitudes **identical** while gaining an **auditable confidence scaffold**.

---

03 — Start Here (minimum viable adoption)

1. **Publish the lane (read-only).** Add `a` beside existing `m`; keep `phi((m,a)) = m` **sacrosanct**.
2. **Add bands.** Begin with defaults; optional **hysteresis** using `+/- delta` promote/demote gates.
3. **Stamp outputs.** Emit **stamped CSVs** so any team can **rerun the exact evidence pack**.

---

04 — Outcomes to Measure Immediately (conservative, stamp-verifiable)

- **Retries** ↓ (fewer **low-band** loops)
- **Hand-offs** ↓ (fewer unnecessary tool calls)
- **Time-to-first-correct** ↑ (early **band-aware** picks)
- **Audit clarity** ↑ (**identical values**, explicit stability)

All changes occur in **scheduler/policy layers**; **model weights and value math remain untouched**.

---

05 — One-Sentence Takeaway

Put a bounded lane beside your AI numbers and keep the numbers identical — so decisions become **calmer**, audits **cleaner**, and scale **safer**.

Under the hood (one-liners)

- **Core math — Shunyaya Symbolic Mathematics (SSM).** Two-lane numeral `x := (m, a)` with `a in (-1,+1)`; **safe map:** `a_c := clamp(a) -> u := atanh(a_c) -> a := tanh(u)`. **Collapse parity:** `phi((m,a)) = m` (**your numbers stay identical**).
- **Symbols (SSMS).** Portable verbs for connectors: `CLAMP`, `MAP(atanh/tanh)`, `FUSE(U/W)`, `MUL_DIV(M2)`, `BAND`, `GATE`, `STAMP`.
- **Time & provenance — SSM-Clock (+ Stamp).** Optional, **tamper-evident one-line stamp** and **order-invariant roll-ups** in u-space, e.g., `SSMCLOCK1|iso_utc|....`
- **Acceleration — SSMH.** Same lane math on **fixed-point pipelines** (`atanh` LUT -> MAC -> `tanh` LUT) with **streaming `(U,W)` accumulators**.
- **Retrieval — SSM-Search.** Lane-native scorer: **rank by bounded `RSI`** while classical retrieval scores remain untouched.
- **What is integrated now.** Lane kernel + SSMS verbs, **SSM-Search scoring**, **SSM-Audit overlays**, optional **SSM-Clock Stamp** fields, **SSMH parity spec** (software<->fixed-point).
- **Roadmap for SSM-AI integration. Adapters** for domain-specific surfaces delivered as **manifest presets** and **evidence packs** (evaluation-first), built on SSM modules — e.g., an **SSM-Chemistry adapter**.

TABLE OF CONTENTS

0A — Introduction (purpose & scope)

SSM-AI is a minimal, evaluation-first way to carry a **bounded alignment coordinate** alongside classical AI quantities.

Each datum is `x := (m, a)`, where `m` is the unchanged magnitude (logit, probability, distance, score, reward, latency, etc.) and `a` is a bounded lane that composes safely across operations, steps, and streams.

SSM-AI maps this two-lane arithmetic into practical AI surfaces: **decoding** (candidate ranking), **RAG** (document and citation scoring), **tools or functions** (call success and reliability), **evaluators or judges** (policy or intent fit), and **ensembles**. The kernel is tiny: **clamp -> map to rapidity -> compose -> inverse-map and bound**.

Goal. Enable immediate software adoption in read-only mode, standardize **manifests and bands**, and scale across teams and vendors with identical semantics. Optional acceleration can be added later without changing the math.

What SSM-AI is (and is not)

- **SSM-AI is** a bounded, composable, operator-native **alignment lane** carried in lockstep with magnitudes.
- **SSM-AI is** collapse-compatible. Classical results are recovered by `phi((m,a)) = m`.
- **SSM-AI is** a three-step kernel anyone can validate quickly:
  `a_c := clamp(a, -1+eps_a, +1-eps_a)` -> `u := atanh(a_c)` -> **compose in** `u` -> `a' := tanh(u')`,
  with order-invariant streaming using `(U,W)` and `a_out := tanh( U / max(W, eps_w) )`.
  A universal decision index `RSI in (-1,+1)` and an alignment-only governor `RSI_env := g_t * RSI`
  (or `RSI_env := tanh(g_t * atanh(RSI))`) are provided for selection and turbulence.
- **SSM-AI is not** a replacement for model training, a probabilistic black box, or a hidden mutator of `m`. It augments, not overrides. Any actuation change requires a separate safety case.

Under the hood (roots, compact)

- **Shunyaya Symbolic Mathematics, SSM**. Two-lane numeral **`x := (m, a)`** with **`a in (-1,+1)`** beside unchanged **`m`**.
  Safe compose: **`a_c := clamp(a)`** -> **`u := atanh(a_c)`** -> **add in** **`u`** -> **`a := tanh(u)`**.
  Streaming fuse is order and shard invariant: **`U += w*atanh(a)`** ; **`W += w`** ; **`a_out := tanh( U / max(W, eps_w) )`**.
  Collapse parity: **`phi((m,a)) = m`**.
- **SSMS (symbols and verbs).** Portable verbs: **`CLAMP`**, **`MAP(atanh/tanh)`**, **`FUSE(U/W)`**, **`MUL_DIV(M2)`**, **`BAND`**, **`GATE`**, **`STAMP`**.
- **SSM-Clock (time and provenance, including Stamp).** Order-invariant roll-ups in **`u`**. One-line stamp: **`SSMCLOCK1|iso_utc|rasi_idx|theta_deg|sha256(file)|chain`**.
- **SSMH (parity on accelerators).** Identical lane math on fixed-point pipelines **(atanh LUT -> MAC -> tanh LUT)** with streaming **`(U,W)`** accumulators.
- **SSM-Search (lane-native retrieval).** Bounded alignment and **`RSI`** for ranking; ties break on the classical retrieval score; parity holds **`phi((m,a)) = m`**.

What is included in SSM-AI (integration now)

- **Lane kernel and SSMS verbs** across SDK and wire formats: **`CLAMP`**, **`MAP`**, **`FUSE`**, **`BAND`**, **`GATE`**, **`STAMP`**.
- **SSM-Search ranking by** **`RSI`** with shard or stream safe **`(U,W)`** merges.
- **SSM-Audit overlays** for 3 to 5 KPI lanes, weekly roll-ups, and ROI worksheets. Bands and pooled indices appear beside existing KPIs.
- **SSM-Clock Stamp fields** and **knobs hashing** for deterministic replay.
- **SSMH parity spec** and reference fixed-point targets. Optional at deploy time. Semantics are identical when used.

Roadmap for SSM-AI integration (adapters next)

Adapters for domain-specific surfaces delivered as **manifest presets** and **evidence packs** in observation-first mode, built on SSM modules. Example: an **SSM-Chemistry adapter**.

All adapters follow the same guarantees: **`|a| < 1`**, **`|RSI| < 1`**, order and shard invariance, and **`phi((m,a)) = m`**.

Why this matters

One mathematical backbone **`clamp -> atanh -> sum in u -> tanh`** yields deterministic, portable stability signals across retrieval, generation, tools, and audits. Provenance (**SSM-Clock**), acceleration (**SSMH**), search (**SSM-Search**), and audit overlays (**SSM-Audit**) plug into the same lane without touching your KPIs. **`phi((m,a)) = m`** remains true throughout.

0B — Core mechanics at a glance (kernel, streaming, bands)

Datum. `x := (m, a)` with `a in (-1,+1)`. **Collapse:** `phi((m,a)) = m`.

Clamp. `a_c := clamp(a, -1+eps_a, +1-eps_a)` with default `eps_a = 1e-6`.

Rapidity map. `u := atanh(a_c)`; inverse `a := tanh(u)`. **Stable add/sub in u-space.**

Lane for mul/div (M2 policy, default). `a' := tanh( atanh(a1) +/- atanh(a2) )`. Manifest controls sign and operation. **Division is guarded by policy.**

Optional M1 policy (multiply-only, rare, opt-in). `a' := clamp(a1 * a2)`. Allowed only if declared and both operands originate from the same lens with small `|a|` ranges.

Streaming fuse, order-independent. `U += w*atanh(a) ; W += w ; a_out := tanh( U / max(W, eps_w) )`. Defaults: `eps_w = 1e-12` for `float64` pipelines.

Weights. Default `w := |m|^gamma` with `gamma = 1`. Uniform option `w := 1` if declared.

Lens -> align -> RSI (single chooser). Contrast `e` from a declared lens -> `a_in := tanh(-c*e)`, `a_out := tanh(+c*e)`. Pool to `RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`.

Calm gate (alignment-only). `RSI_env := g_t * RSI` where `g_t in [0,1]` from telemetry. **`m` is never altered.**

Bands (global defaults, normative).
`A++`: `a >= +0.90`, `A+`: `+0.60 <= a < +0.90`, `A0`: `-0.60 < a < +0.60`, `A-`: `-0.90 < a <= -0.60`, `A--`: `a <= -0.90`.
Overrides must be explicit in the manifest and in CSV or log headers.

Division policy, normative. Default `division_policy = "strict"`. Disallow divide if any required denominator lane or magnitude violates declared bounds. Alternatives `"meadow"` or `"soft"` require explicit declaration and numeric guards. **Combine policy remains `M2` for alignment composition.**

Non-negotiable invariants, normative.

(i) **Collapse parity:** `phi((m,a)) = m`.
(ii) **Order invariance:** batch equals stream equals shuffled within fixed epsilon.
(iii) **Determinism:** same manifest implies same outputs.
(iv) **Lane purity:** policies, bands, and gates act on `a` only. `m` is never mutated.
(v) **Division near zero:** default strict. Alternatives must be declared.
(vi) **Cross-vendor parity:** identical manifests produce comparable `a` and `RSI` across models and providers.

Dtype & epsilon note (implementation guidance). Prefer `float64` for u-space accumulation. If constrained to `float32`, set `eps_w >= 1e-8` and keep `|a| < 1 - eps_a` with `eps_a >= 1e-6` to avoid `atanh(±1)` and division by zero in `a_out := tanh(U / max(W, eps_w))`.

Manifest crib, minimum fields. `eps_a, eps_w, gamma, weights_policy, combine_policy (M2 default; M1 only if declared), division_policy, bands, lens_id, lens_params, g_t_source, dtype, knobs_hash.`

0C — Surfaces -> Kernel -> Acceleration (one substrate)

Surfaces, drop-in points.

- **Decoding, LLM candidates.** Keep **`m`** intact. Derive lens contrasts **`e_in`**, **`e_out`**. Compute **`RSI`** and pick.
- **RAG, retrieval and citation.** Items carry **`(m_retrieval, a_doc)`**. Pool in **`u`**. Forward **`a_pool`** into generation. Select by **`RSI`**.
- **Tools or functions.** Each call emits **`(m_tool, a_tool)`**. Retries or schedulers use **`RSI_env := g_t * RSI`**. **`m`** is never modified.
- **Evaluators or judges.** Compliance or intent or style scores become contrasts -> alignments -> **`RSI`** for routing.
- **Ensembles.** Pool alignments across models or providers. Output one bounded chooser without touching their **`m`**.

Kernel, always the same tiny pipe.
`a_c := clamp(a, -1+eps_a, +1-eps_a)` -> `u := atanh(a_c)` -> compose (add or sub or weighted sum, M2 for mul or div) -> `a' := tanh(u')`.
Streaming fuse, order invariant. `U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )`.
Defaults. **`eps_a = 1e-6`**, **`eps_w = 1e-12`**, **`w := |m|^gamma`** with **`gamma = 1`**.
Collapse parity holds everywhere: **`phi((m,a)) = m`**.

Optional acceleration, same semantics.
When throughput matters, the identical lane math maps to a small accumulator and arithmetic block. Software to accelerator parity is mandatory. No change to manifests or bands.

12-line walk-through (pseudocode, candidate selection)

```
# Given candidates with classical metrics -> contrasts (e_in_k, e_out_k)
and weights w_k
def clamp_a(a, eps_a=1e-6):
    lo, hi = -1.0 + eps_a, 1.0 - eps_a
    return max(lo, min(hi, a))

def rsi_for_candidate(items, c=1.0, eps_w=1e-12):
    U_in, V_out, W_in = 0.0, 0.0, 0.0
    for (e_in, e_out, w) in items:
        a_in  = tanh(-c*e_in); a_out = tanh(+c*e_out)
        U_in  += w * atanh(clamp_a(a_in))
        V_out += w * atanh(clamp_a(a_out))
        W_in  += w
    return tanh( (V_out - U_in) / max(W_in, eps_w) )

def select(cands, g_t=1.0):
    scored = [(cid, g_t * rsi_for_candidate(items)) for (cid, items) in
cands]
    return max(scored, key=lambda x: x[1])  # choose by RSI_env
```

30-second mental model. Keep `m` exactly the same `phi((m,a)) = m`. Compute bounded `a` from declared lens contrasts. Pool in rapidity for order-invariant stability. If the environment is shaky, apply `g_t`. Choose by `RSI_env`.

Note: for hardware or accelerator parity and fixed-point specs, see the relevant appendix.

0D — Why this matters (immediate value)

- **Backward compatible by construction.** Classical numbers stay identical `phi((m,a)) = m`. You gain a bounded lane `a in (-1,+1)` and a bounded chooser `RSI in (-1,+1)` without retraining or changing value-path code.
- **Reproducible and fair across vendors.** Order-invariant pooling guarantees batch equals stream equals shuffled via `U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )`. Identical manifests yield comparable `a` or `RSI` across prompts, models, and providers.
- **Immediate cost and latency gains.** Fewer blind retries and calmer agent loops as schedulers use `RSI_env := g_t * RSI`. Token, tool, and API waste drops without touching `m`.
- **Quality you can see.** Bands `A++/A+/A0/A-/A--` surface stability at a glance. Low-band candidates trigger guardrails. High-band ones flow.
- **Auditable decisions.** Each choice emits `(m, a, U, W, band, g_t)` and a simple chain stamp. Reviews become stamp and replay.
- **Scale by manifest, not migration.** One small manifest **(weights, clamps, bands, division policy)** standardizes evaluation across teams and regions.
- **Safety by design.** The calm gate and policies act on `a` only. `m` is never altered. Promotion from advisory to actuation requires a separate, explicit safety case.

Pocket walkthrough, picking between two answers.
Two candidates provide contrasts `(e_in, e_out)` from a declared lens.

Map to alignments: `a_in := tanh(-c*e_in)`, `a_out := tanh(+c*e_out)`. Pool once: `RSI := tanh( (sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w) )`.
Because `tanh` and `atanh` are monotone, a larger net contrast `(e_out - e_in)` implies a larger `RSI`. Select by `RSI` or `RSI_env := g_t*RSI` while keeping `m` untouched.

30-second audit recipe.

1. Verify collapse: recompute output with `phi((m,a))` and confirm equality to baseline.
2. Shuffle inputs: confirm identical `a_out` or `RSI` within epsilon.
3. Check bands: thresholds match manifest; borderline cases respect declared epsilons.
4. Confirm gate purity: recompute with `g_t = 1` then apply `g_t < 1` and ensure only alignment scales.

Zero-Infra adoption.
Observation-only via `phi((m,a)) = m`. No retraining. Small manifest. API safe by emitting `m` where a single number is required. Order and shard invariant via `U/W`. Publish the lane and stamps quickly, add bands and gate next.

Cost impact, conservative.
`Annual_Savings ≈ S_base * r_save`, with `r_save in [0.10, 0.20]`. Drivers: fewer retries, tighter beams, calmer loops, better vendor mix. Achieved without touching `m`.

0E — What you can test in minutes (calculator fast)

Tiny checklist.

- Collapse parity: `phi((m,a)) = m`.
- Order invariance: batch equals stream equals shuffled via `U/W`.
- Lens to `RSI` chooser: `RSI in (-1,+1)` from simple contrasts.
- Calm gate purity: `RSI_env := g_t * RSI` only scales alignment.
- Bands at a glance: map `a` to `A++/A+/A0/A-/A--` using defaults.
- Division policy: M2 lane rule for mul or div and declared `division_policy`.

Ready-to-copy numeric vectors.

E1) Lens -> align -> RSI (single decision).
`c = 1.0, w = 1, eps_w = 1e-12`. `e_in = 0.2, e_out = 0.5`
`a_in = tanh(-0.2)` ; `a_out = tanh(+0.5)`
`U_in = -0.2` ; `V_out = +0.5`
`RSI = tanh(0.7) ≈ 0.604367` -> Band A+.

E2) Calm gate purity.
`g_t = 0.60` -> `RSI_env = 0.60 * 0.604367 ≈ 0.362620` (`m` unchanged).

E3) Streaming fuse, order invariant.
`a1 = tanh(0.2)`, `a2 = tanh(0.4)`, `w1 = w2 = 1` -> `a_out = tanh(0.3) ≈ 0.291312`.
Shuffle `(a1,a2)` -> same `a_out`.

E4) Collapse parity.
Any `m` (for example `0.73`) with any `a` -> `phi((m,a)) = m = 0.73`.

E5) Lane mul or div (M2).
`a1 = tanh(0.7)`, `a2 = tanh(0.1)` -> `a_mul = tanh(0.8) ≈ 0.664037`, `a_div = tanh(0.6) ≈ 0.537050`.
Declare `division_policy` (default `"strict"`).

E6) Banding defaults.
`A++: a >= +0.90`, `A+: +0.60 <= a < +0.90`, `A0: -0.60 < a < +0.60`, `A-: -0.90 < a <= -0.60`, `A--: a <= -0.90`.
Examples: `0.291312 -> A0`, `0.604367 -> A+`.

Pocket pseudocode.

```
def clamp_align(a, eps_a=1e-6):
    return max(-1+eps_a, min(+1-eps_a, a))

def fuse_stream(a_list, w_list, eps_w=1e-12):
    U = sum(w*atanh(clamp_align(a)) for a, w in zip(a_list, w_list))
    W = sum(w_list)
    return tanh(U / max(W, eps_w))

def rsi(e_in, e_out, c=1.0, w=1.0, eps_w=1e-12):
    a_in  = tanh(-c*e_in); a_out = tanh(+c*e_out)
    U_in  = w * atanh(a_in); V_out = w * atanh(a_out)
    return tanh( (V_out - U_in) / max(w, eps_w) )

def apply_gate(RSI, g_t):
    return g_t * RSI
```

One-minute self-check. `phi((m,a)) == m` ok, shuffle test ok, `RSI` and `RSI_env` bands as expected, gate scales alignment only, manifest fixed and stamped.

0F — How this builds on SSM, SSMS, and SSM-Clock

SSM (numerals and invariants).
Two-lane numeral `x := (m,a)` with `a in (-1,+1)`. Collapse `phi((m,a)) = m`.
Safe composition `clamp -> u := atanh(a) -> compose -> a := tanh(u)`.
Order-invariant streaming fuse `U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )`.
Lane product or ratio by M2 `a' := tanh( atanh(a1) +/- atanh(a2) )`. Division guarded by policy.

SSMS (verbs -> clean interfaces).
Verbs become AI ports: `CLAMP`, `MAP(atanh)`, `FUSE(U/W)`, `MUL_DIV(M2)`, `BAND`, `GATE`, `STAMP`.

SSM-Clock (time, stamps, continuity).
Stamp each decision for replay and fairness:
`stamp = "SSMCLOCK1|" + iso_utc + "|" + rasi_idx + "|" + theta_deg + "|" + sha256(file) + "|" + chain_k`
Time-sliced views compare **`RSI`** by hour, day, or week without drift via order-invariant pooling.
Temporal calm uses **`g_t in [0,1]`** to obtain **`RSI_env := g_t * RSI`** while leaving **`m`** untouched.

What this means for your current AI framework.
Keep the stack. Add the lane and stamp. Select by **bounded clarity `(RSI, RSI_env)`** without touching **`m`**.

Walk-through A, decoding (10 lines).

```
def score_candidate(metrics, c=1.0, w=1.0, eps_w=1e-12):
    e_in, e_out = metrics["e_in"], metrics["e_out"]
    a_in  = tanh(-c*e_in); a_out = tanh(+c*e_out)
    U_in  = w * atanh(a_in); V_out = w * atanh(a_out)
    return tanh((V_out - U_in) / max(w, eps_w))

def pick_from_beam(beam, g_t):
    scored = [(cand, g_t*score_candidate(cand.lens)) for cand in beam]
    return max(scored, key=lambda x: x[1])
```

Walk-through B, RAG with time-aware gating (12 lines).

```
def rsi_rag(doc_items, c=1.0, eps_w=1e-12):
    U_in = V_out = W = 0.0
    for doc in doc_items:
        e_in, e_out, w = doc["tox_gap"], doc["cit_hit"], doc["weight"]
        U_in += w*atanh(tanh(-c*e_in))
        V_out+= w*atanh(tanh(+c*e_out))
        W    += w
    return tanh((V_out - U_in) / max(W, eps_w))

def choose_doc(docs, telemetry):
    g_t = telemetry["calm_gate"]
    return max(docs, key=lambda d: g_t * rsi_rag(d["items"]))
```

Walk-through C, stamp the decision.

```
iso_utc   = "2025-10-27T07:00:00Z"
theta_deg = "163.49167"
rasi_idx  = "5"
digest    = sha256("decision_payload.csv")
chain_k   = sha256(prev_chain + "|" +
"SSMCLOCK1|"+iso_utc+"|"+rasi_idx+"|"+theta_deg+"|"+digest)
stamp     =
"SSMCLOCK1|"+iso_utc+"|"+rasi_idx+"|"+theta_deg+"|"+digest+"|"+chain_k
```

One line. SSM-AI is a support layer that lets any existing AI stack keep its numbers and gain bounded, time-aware clarity.

0G — Who should read this

Product and AI leads, prompting or LLM engineers, RAG and search teams, agent and tooling teams, evaluation and safety reviewers, platform and infrastructure owners, procurement and vendor management, finance and operations.

0H — Guardrails and responsible use

- **Observation first.** Keep classical numbers intact `phi((m,a)) = m`. Use `a in (-1,+1)` and `RSI in (-1,+1)` for visibility, routing, and advisory scheduling.
- **Clamp-first numerics.** Bound alignment before combining: `a_c := clamp(a, -1+eps_a, +1-eps_a)` with `eps_a = 1e-6`. Use `u := atanh(a_c)` and inverse `a := tanh(u)` for stable composition.
- **Order-invariant fusion.** Same fuse for stream, batch, and shuffle: `U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )` with `eps_w = 1e-12` and default `w := |m|^gamma`, `gamma = 1`.
- **Gate purity.** `RSI_env := g_t * RSI` with `g_t in [0,1]`. Never alter `m` inside SSM-AI.
- **Division policy.** Use M2 lane rule `a' := tanh(atanh(a1) +/- atanh(a2))`. Default `division_policy = "strict"`. Alternatives must be explicit.
- **Manifest as contract.** Publish lens, scale `c`, weight rule `w`, bands, clamps, and policies. Same manifest implies same outputs.
- **Priors are bounded.** In `u` only: `u' := u + alpha*Index` with `alpha <= alpha_max`. Priors never modify contrast `e`.
- **Privacy and scope.** Derive `a` or `RSI` from aggregates or telemetry. SSM-AI does not compute hazards, medical advice, or legal outcomes.
- **Stamp and replay.** One-line ASCII stamp per decision: `SSMCLOCK1|iso_utc|rasi_idx|theta_deg|sha256(file)|chain`.
- **Escalation.** If any acceptance gate fails, revert selection to classical logic, for example highest `m`, and flag for review.

License (summary)

0I — What is inside this document (preview)

Canon quick start, lens library and manifest, walk-throughs and pseudocode, cross-domain adapters, acceptance and QA, stamping and ledger, governance and safety, economics and rollout, appendices with operator table, worked vectors, templates. Suggested reading paths: Strategic 10 minutes, Engineering 30 to 60 minutes, Evaluation or Safety 30 minutes.

0J — One-minute elevator summary

Zero-Infra adoption. Observation only `phi((m,a)) = m`. `clamp -> atanh -> add -> tanh`. Tiny manifest. API safe by emitting `m` where one number is required.

Cost impact. `Annual_Savings ≈ S_base * r_save` with `r_save in [0.10, 0.20]`. Gains from fewer retries or tokens, calmer loops, smarter vendor mix. No change to `m`.

Core idea. Add a bounded lane beside the numbers you already compute: `x := (m,a)` with `a in (-1,+1)` and `phi((m,a)) = m`. Compose via rapidity `u := atanh(a)` (inverse `a := tanh(u)`). Pool order-invariantly with `U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )`. A declared lens yields a bounded chooser `RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`, optionally tempered by `RSI_env := g_t * RSI`. Bands `A++/A+/A0/A-/A--` make risk legible at a glance. No retraining. No edits to `m`.

Drop-in surfaces. Decoding **(pick by `RSI_env`)**, RAG **(pool doc alignments, route by `RSI`)**, tools or agents **(throttle via `g_t`)**, evaluators **(route by band)**, ensembles **(cross-vendor pooling)**.

Time and audit. Stamp each decision: `SSMCLOCK1|iso_utc|rasi_idx|theta_deg|sha256(file)|chain`.

Adoption steps. Publish the lane, add bands and `RSI` **or** `RSI_env`, stamp outputs. Results: faster choices, calmer loops, cleaner audits, lower waste. Same numbers, now with bounded clarity.

---

1) What SSM-AI Is (Positioning & Promise)

1.1 Purpose & Non-Goals

Purpose. Add a **bounded alignment lane** to the AI you already run so **selection, routing, retries, and audits** become **visible, comparable, and order-invariant** — **without touching the numbers you trust**. Each quantity becomes `x := (m,a)` with `a in (-1,+1)` and **collapse parity** `phi((m,a)) = m`.

Alignment composes via **rapidity** `u := atanh(a)` (inverse `a := tanh(u)`), and streams fuse as
`U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )`.
A declared **lens** turns evidence into a single chooser
`RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`,
optionally tempered by a **calm gate** `RSI_env := g_t * RSI` (**alignment only**).

Why now.
• **Speed:** calculator-fast checks → idea→pilot in hours, pilot→portfolio in weeks.
• **Quality:** fewer over-confident outputs; calmer agent loops; **banded risk** (`A++/A+/A0/A-/A--`).
• **Comparability:** same **manifest** ⇒ apples-to-apples across prompts, models, vendors.
• **Time & audit:** per-decision **stamps** enable replayable reviews and day/week roll-ups.

Scope (where it drops in). Decoding/beam pick, RAG/doc ranking, search (internet/intranet/local), tools/agents, evaluators/judges, multi-model ensembles. **SSM-Clock/Stamp** adds time-sliced replay; **SSMH** later accelerates the exact same math; domain packs (for example **SSM-Chem**, **SSM-Audit**) supply lenses and banded KPIs.

Non-Goals (hard constraints).
• **No model surgery:** training/logits/probabilities remain intact (`phi((m,a)) = m`).
• **No hidden smoothing:** all knobs live in a **public manifest** (bands, clamps, weights, policies).
• **No order dependence: batch == stream == shuffled** by design (same `U/W` fuse).
• **No silent actuation:** gate scales alignment only (`RSI_env := g_t * RSI`); changes to control/UX need a **separate safety case**.
• **No PII usage:** lenses derive from non-sensitive signals/telemetry or declared metrics.
• **No "explainability" promises:** SSM-AI reports **bounded stability/fit**; it does not infer intents beyond declared lenses.

Traditional vs SSM-AI (micro-example — beam pick).

Goal: choose the best candidate consistently across vendors.
• **Traditional:** pick `argmax(prob)` or a heuristic mix; retries on ad-hoc thresholds; unstable under turbulence.
• **SSM-AI:** compute lens contrasts `(e_in, e_out)` → `a_in := tanh(-c*e_in)` ; `a_out := tanh(+c*e_out)` →
`RSI := tanh( (sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w) )`
→ optional `RSI_env := g_t * RSI` → choose by `RSI` or `RSI_env`. `m` remains identical (`phi((m,a)) = m`).

Pocket pseudocode (drop-in, 8 lines).

```
def choose(cands, c=1.0, eps_w=1e-12, g_t=1.0):
    best = None
    for cand in cands:               # cand.lens -> list of (e_in, e_out,
w)
        U_in = V_out = W = 0.0
        for e_in, e_out, w in cand.lens:
```

```
            U_in  += w*atanh(tanh(-c*e_in))
            V_out += w*atanh(tanh(+c*e_out))
            W     += w
        RSI = tanh( (V_out - U_in) / max(W, eps_w) )
        best = max([best,(cand, g_t*RSI)], key=lambda x: x[1]) if best else
(cand, g_t*RSI)
    return best[0]
```

Acceptance snapshot (must pass). Collapse parity ✔ `phi((m,a)) = m` • Order invariance ✔ `U/W` • Clamp bounds ✔ (`eps_a`, `eps_w`) **• Gate purity ✔** (`m` untouched) **• Determinism ✔** (same manifest ⇒ same outputs)

1.2 Observation-Only Ethos & Collapse Parity

What "observation-only" means. SSM-AI adds a **bounded alignment lane** beside numbers you already compute, **without altering those numbers**. Every quantity is `x := (m,a)` with `a in (-1,+1)` and a collapse map `phi((m,a)) = m`. Selection or routing uses the lane `(a, RSI)`, while classical logic continues to see `m` exactly as before.

Non-negotiables.
• **Collapse parity:** `phi((m,a)) = m` always.
• **Clamp-first:** `a_c := clamp(a, -1+eps_a, +1-eps_a)` (default `eps_a = 1e-6`).
• **Rapidity map:** `u := atanh(a_c)` ; inverse `a := tanh(u)` (stable composition).
• **Streaming fuse (order-invariant):** `U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )` (default `eps_w = 1e-12`).
• **Lane product/ratio (M2 policy):** `a' := tanh(atanh(a1) +/- atanh(a2))` with declared `division_policy`.
• **Gate purity:** `RSI_env := g_t * RSI` (**alignment only**; `m` never changes).

Why collapse parity holds (quick identities).
Let classical ops apply to **magnitudes only**; the lane composes separately.
• **Sum/diff:** `(m1,a1) ± (m2,a2) -> (m1 ± m2, a±)` ⇒ **`phi(...) = m1 ± m2`**.
• **Mul/div:** `(m1,a1) * (m2,a2) -> (m1*m2, a_mul)` and `(m1,a1) / (m2,a2) -> (m1/m2, a_div)` ⇒ **`phi(...) = m1*m2`** or **`m1/m2`**.
• **Pooling:** `pool({(m_i,a_i)}) -> (m_pool, a_out)` with `m_pool` defined by your existing logic; lane uses `U/W` ⇒ **`phi(...) = m_pool`**.

Calculator-fast checks (do them once).

1. **Collapse parity:** pick `m = 0.73`, `a = tanh(0.45)` → **`phi((m,a)) = 0.73`**.
2. **Order invariance:** `a1 = tanh(0.2)`, `a2 = tanh(0.4)`, `w1=w2=1` → `a_out = tanh((0.2+0.4)/2)`; swap inputs → same `a_out`.
3. **Lane mul/div:** `a1 = tanh(0.7)`, `a2 = tanh(0.1)` → `a_mul = tanh(0.8)`, `a_div = tanh(0.6)`; meanwhile `m` multiplies/divides as usual.

Traditional vs SSM-AI (micro-example — post-processing).

Goal: surface reliability without rewriting predictions.
• **Traditional:** rescore probabilities or add thresholds (can change numbers); order effects creep in.
• **SSM-AI:** keep **`m`** identical (**`phi((m,a)) = m`**); compute bounded **`a`** beside it; pool in rapidity (`U/W`); select/route by **`RSI`** or **`RSI_env`**; comparable across vendors; order-invariant.

Pocket pseudocode — prove you are observation-only (8 lines).

```
def observe_only(m_list, a_list, w_list, eps_a=1e-6, eps_w=1e-12):
    for m, a in zip(m_list, a_list):  # collapse parity
        assert m == (m)               # phi((m,a)) == m by definition
    U = sum(w*atanh(max(-1+eps_a, min(1-eps_a, a))) for a, w in zip(a_list, 
w_list))
    W = sum(w_list)
    a_out = tanh(U / max(W, eps_w))
    return a_out   # used for RSI/bands; m-path remains untouched
```

Edge policies (declare once, keep forever).
• **Division near zero:** `division_policy = "strict"` by default; `"meadow"` or `"soft"` must be explicit.
• **Weights:** default `w := |m|^gamma` with `gamma = 1`; uniform `w := 1` permitted if declared.
• **Bands (defaults):** `A++`, `A+`, `A0`, `A-`, `A--` with thresholds declared once in the manifest.

One-line takeaway. SSM-AI **observes and publishes alignment**; it **never edits your numbers**: **`phi((m,a)) = m`**.

1.3 Where the Lane Lives (decode, RAG, search, tools/agents, evaluators, ensembles)

Surfaces (drop-in points). SSM-AI rides beside your stack; it never edits **`m`**. Each surface computes a **bounded alignment** and picks by a single chooser **`RSI in (-1,+1)`** (optionally **`RSI_env := g_t * RSI`**). Lenses map plain evidence to contrasts **`e`**, then to alignments: **`a_in := tanh(-c*e)`**, **`a_out := tanh(+c*e)`**. Pools are order-invariant via **`U += w*atanh(a)`** ; **`W += w`** ; **`a_out := tanh( U / max(W, eps_w) )`**. Defaults: **`eps_a = 1e-6`**, **`eps_w = 1e-12`**, **`w := |m|^gamma`** with **`gamma = 1`**. **Collapse parity:** **`phi((m,a)) = m`** throughout.

1.3.1 Decoding / Beam Pick

Traditional vs SSM-AI (micro).
Goal: pick the best candidate robustly.
• **Traditional:** choose `argmax(prob)` or heuristic mix; retries when "looks wrong."
• **SSM-AI:** compute `(e_in, e_out)` → map **`a_in := tanh(-c*e_in)`**, **`a_out := tanh(+c*e_out)`** →

RSI := tanh((sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w)) → optional **RSI_env := g_t * RSI** → choose by `RSI_env`. **m** unchanged (**phi((m,a)) = m**).

Pseudocode (10 lines).

```
def rsi_candidate(items, c=1.0, eps_w=1e-12):
    U_in = V_out = W = 0.0
    for (e_in, e_out, w) in items:
        U_in  += w*atanh(tanh(-c*e_in))
        V_out += w*atanh(tanh(+c*e_out))
        W     += w
    return tanh( (V_out - U_in) / max(W, eps_w) )

def pick_beam(beam, g_t=1.0):
    scored = [(cand, g_t*rsi_candidate(cand.lens_items)) for cand in beam]
    return max(scored, key=lambda kv: kv[1])[0]
```

Worked numbers (1 step).
`e_in=0.2`, `e_out=0.5`, `w=1`, `c=1` ⇒ `RSI = tanh(0.7) ≈ 0.604` → **band A+**.

1.3.2 RAG (Documents + Citations)

Goal: rank documents that are helpful and safe.
• **Traditional:** mix BM25/embeddings + ad-hoc thresholds; order effects creep in.
• **SSM-AI:** `e_in := toxicity_gap`; `e_out := citation_hit + semantic_gain`.
Map **a_doc_in := tanh(-c*e_in)**, **a_doc_out := tanh(+c*e_out)**; pool in **u** → `a_pool`; compute doc `RSI`; rank by `RSI` (or `RSI_env`). Forward top-K **alignments** (not **m**) into generation; **m** stays pristine.

Pseudocode (doc score).

```
def rsi_doc(signals, c=1.0, eps_w=1e-12):
    U_in = V_out = W = 0.0
    for (tox_gap, cit_hit, gain, w) in signals:
        U_in  += w*atanh(tanh(-c*tox_gap))
        V_out += w*atanh(tanh(+c*(cit_hit+gain)))
        W     += w
    return tanh( (V_out - U_in) / max(W, eps_w) )
```

1.3.3 Search (Internet / Intranet / Local)

Lens idea. Let features be `hit_quality`, `freshness`, and a penalty `risk_penalty`. Declare units and scales once; form a signed contrast:
e := (alpha*hit_quality + beta*freshness - gamma*risk_penalty) / Unit.

Traditional vs SSM-AI (micro).

Goal: rank results with freshness and risk considered, reproducibly.
• **Traditional:** weighted sum → score; scale/ordering sensitive; hard to compare engines.
• **SSM-AI:** build **`e`** → **`a_in := tanh(-c*(risk_term))`**, **`a_out := tanh(+c*(quality+freshness_term))`** → pool via **`U/W`** → rank by **`RSI`** (bounded, comparable); bands drive UI badges or safe-open policies; **`m`** not touched.

Numeric mini-example.
`alpha=1.0`, `beta=0.5`, `gamma=0.8`, `Unit=1`
• **Result A:** `(0.9, 0.6, 0.2)` ⇒ `e = 1.04` ⇒ `RSI ≈ tanh(1.04) ≈ 0.778` (**A+**).
• **Result B:** `(0.8, 0.2, 0.5)` ⇒ `e = 0.5` ⇒ `RSI ≈ tanh(0.5) ≈ 0.462` (**A0**).

1.3.4 Tools & Agents (Retries, Branching, Routing)

Calm gate from telemetry. Build **`g_t in [0,1]`** from bounded signals: contradiction rate `F_t`, error rate `E_t`, latency spike `L_t`, churn `V_t`. Example:
`g_t := clamp( 1 - (wF*F_t + wE*E_t + wL*L_t + wV*V_t), 0, 1 ).`

Goal: cut wasteful loops during turbulence.
• **Traditional:** fixed retry rules; bursts under partial outages.
• **SSM-AI:** compute `RSI` for the step; apply **`RSI_env := g_t * RSI`** (**alignment only**).
If `RSI_env` < band `A0` → backoff; if > `A+` → allow retry/escalation. **`m`** and tool outputs unchanged; decisions **stamped**.

Pseudocode (branch rule).

```
def should_retry(step_items, telemetry):
    RSI = rsi_candidate(step_items)
    RSI_env = telemetry["g_t"] * RSI
    return RSI_env >= 0.60   # example: A+ threshold
```

1.3.5 Evaluators / Judges (Policy, Intent, Style)

Lens idea. `e_out := policy_hits + intent_match + style_fit` (positive evidence); `e_in := policy_gaps + safety_flags + contradiction` (penalties).
Compute `RSI` as usual. Report `(RSI, band)` beside the original judge scores; **do not rescore `m`.**

1.3.6 Multi-Model Ensembles (Cross-Vendor Fairness)

Order-invariant pooling across providers.

```
U = sum( w_i * atanh(RSI_i) )
W = sum( w_i )
RSI_pool = tanh( U / max(W, eps_w) )
```

Weights `w_i` can be uniform or a declared reliability index. Selection is by `RSI_pool` (or `RSI_env`), while each provider's **`m`** remains intact under **`phi`**.

Mini "Before / After" (search click-through).
Before: `Rank = wq*quality + wf*freshness - wr*risk` (tuning fragile; no bounded signal; audits non-portable).
After (SSM-AI):
`e := (alpha*quality + beta*freshness - gamma*risk)/Unit` →
`a_in := tanh(-c*(gamma*risk))` ; `a_out := tanh(+c*(alpha*quality + beta*freshness))` →
`RSI := tanh( (sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w) )` →
show **band**; click policy uses **`RSI_env`** with **`g_t`** from live telemetry; **`m`** unchanged.

One-line takeaway. Every surface gets the same tiny math: **declare a lens → compute bounded alignment → pool order-invariantly → choose by `RSI`/`RSI_env`**, while **`phi((m,a)) = m`** keeps your numbers untouched.

1.4 Positioning vs Related Methods (context, not conflict)

• **Entropy-only confidence:** often unbounded and order-sensitive mixes vs SSM-AI's **bounded lane `a in (-1,+1)`** and **order-invariant `U/W`**.
• **MC dropout / ensembles:** require retraining or many forward passes; SSM-AI is **drop-in, read-only, calculator-fast**.
• **Calibrated probabilities:** reshape **`m`**; SSM-AI **preserves `m` exactly** (**`phi((m,a)) = m`**) and adds a **parallel stability signal**.
• **Heuristic rerankers:** ad-hoc scales/order effects; SSM-AI uses a **manifest** and a **single bounded chooser `RSI`**.

Synergy. SSM-AI can **carry outputs** from these methods as **lenses**; it **standardizes the selection layer** without forbidding existing techniques.

1.5 Promise (measurable outcomes to track on day 1)

• **Retries** ↓ and **hand-offs** ↓ (low-band backoffs, high-band passes).
• **Time-to-first-correct** ↑ (bounded chooser reduces dithering).

• **Over-confidence exposure** ↑ (bands surface fragility).
• **Auditability** ↑ (stamped, replayable choices with **unchanged `m`**).

All measured with **identical prompts/models**; only the **alignment lane and bands** are added.

1.6 Limits & Failure Modes (read before piloting)

• **Low-signal lenses:** if the lens is weak/noisy, `a ~ 0` and bands indicate **insufficient evidence**; do not over-interpret.
• **Division near zero:** default `division_policy = "strict"`; alternatives must be **explicit** and **tested**.
• **Throughput hotspots:** very high QPS should use **vectorized `atanh/tanh`** or **SSMH**; semantics stay identical.
• **Gate misuse:** `g_t` must be **declared and bounded**; it scales alignment only. Never couple it to mutate `m`.
• **Manifest drift:** treat the manifest as a **contract**; changes require a **new knobs hash** and re-baseline vectors.

---

2) Canon — Numerals, Operators, Pools

2.1 Two-Lane Numeral and Collapse

Definition.
Every quantity carries a magnitude and a bounded alignment lane: `x := (m, a)` with `a in (-1,+1)`. **Collapse map:** `phi((m,a)) = m`.

Semantics (read-only lane).

- `m` is the classical value (logit, probability, score, distance, latency, reward, etc.).
- `a` is dimensionless, bounded, and composable; it summarizes stability/fit under a declared **lens**.
- All routing/selection may inspect `a` (or `RSI`) while classical code continues to see `m` via `phi`.

Bounds & clamps.

- Enforce `|a| < 1`. Use a small clamp: `a_c := clamp(a, -1+eps_a, +1-eps_a)` (default `eps_a = 1e-6`).
- Neutral alignment is `a = 0` (no preference; `u = 0` in rapidity space).

Identity & neutrality.

- **Numeral identity:** `(m, 0)` behaves like the classical `m` under collapse: `phi((m,0)) = m`.
- **Lane neutrality under pooling:** adding neutral items (`a = 0`) does not change pooled alignment (see **2.3**).

Units.

- `m` retains its original units; `a` is unitless.
- Lens scale `c` and any `Unit` constants are declared in the manifest so `a` stays comparable across runs.

Data shapes.

- Scalars, vectors, matrices, and tensors are supported. `m` and `a` share shape; operations are elementwise except in explicit pooling (`U/W`).

Serialization (logs & CSVs).

- **Columns:** `m, a, band, U, W, RSI, RSI_env, stamp`.
- **Collapse for external systems:** write `m_out := phi((m,a)) = m` wherever only classical values are allowed.

Compatibility (drop-in).

- Existing APIs that expect `m` continue unchanged; the lane is **side-car**.
- When emitting one field, choose `m`; when emitting two, choose `(m,a)`; when emitting a decision, include `RSI` (and optionally `RSI_env`).

Tiny examples.

- **E1 — Neutral lane.** `x = (0.73, 0)` ⇒ `phi(x) = 0.73`.
- **E2 — Bounded lane.** `x = (42, tanh(0.8)) ≈ (42, 0.664037)` ⇒ `phi(x) = 42`.
- **E3 — Tensor shape.** `m = [0.2, 0.8]`, `a = [tanh(0.1), tanh(0.3)]` ⇒ shapes match; collapse yields `[0.2, 0.8]`.

Traditional vs SSM-AI (representation).

Goal: add reliability without changing existing outputs.

- **Traditional:** ad-hoc flags/thresholds; scale and ordering issues.
- **SSM-AI:** represent each value as `(m,a)`; publish **band** and `RSI` for decisions. Classical outputs remain `m` via `phi((m,a)) = m`; no API breakage.

Pocket struct & helpers (pseudocode).

```
struct LaneValue { m: float, a: float }  # with |a| < 1

def collapse(x):
    return x.m  # phi((m,a)) = m

def clamp_align(a, eps_a=1e-6):
    return max(-1+eps_a, min(+1-eps_a, a))

def make(m, a, eps_a=1e-6):
    return LaneValue(m=m, a=clamp_align(a, eps_a))
```

One-line takeaway. Treat every AI quantity as `x := (m,a)`; always recover the classical path with `phi((m,a)) = m`, while the lane powers **bounded, comparable** decisions.

2.2 Clamp & Rapidity

`a_c := clamp(a, -1+eps_a, +1-eps_a)` ; `u := atanh(a_c)` ; `a := tanh(u)`

Why clamp?
Keep alignment strictly inside `(-1,+1)` so composition is finite and stable. Use a tiny margin `eps_a` to avoid infinities at `±1`:
`a_c := clamp(a, -1+eps_a, +1-eps_a)` (default `eps_a = 1e-6`).

Why rapidity?
Map bounded alignment to an **additive** space for safe, order-invariant composition:
`u := atanh(a_c)` ; inverse `a := tanh(u)`.

Properties useful for AI routing and pooling.

- **Monotone & odd:** `atanh/tanh` preserve ordering; `atanh(-a) = -atanh(a)`.
- **Additive in `u`:** fusing evidence adds `u`'s, then returns to the bounded lane via `tanh`.
- **Stable near 0:** `atanh(a) ≈ a` for small `|a|`; tiny signals do not explode.
- **Edge-finite:** clamp keeps `|u| < +∞` even when `a -> ±1`.

Core identities (use everywhere).
`atanh(tanh(u)) = u` and `tanh(atanh(a)) = a_c` (post-clamp).
`d/du tanh(u) = 1 - tanh(u)^2 = 1 - a^2` (smooth bounded sensitivity).
`atanh(a) = 0.5*log((1+a)/(1-a))` (log-odds form for `a in (-1,+1)`).

Streaming fuse (mean in u-space).
Given items with weights `w_i`:
`U := sum_i w_i * atanh(a_i)` ; `W := sum_i w_i` ; `a_out := tanh( U / max(W, eps_w) )` with default `eps_w = 1e-12`.
This makes **batch == stream == shuffled** by construction.

Numerical guardrails.

- Choose **`eps_a`** to fit your dtype: **`1e-6`** for float32, **`1e-12`** for float64.
- Use **`eps_w`** to avoid divide-by-zero when **`W = 0`**.
- Vector/tensor shapes: apply **clamp**, **`atanh`**, **`tanh`** elementwise; do reductions only in fusing.

Worked minis (calculator-fast).

- **Clamp:** `a = 0.9999999`, `eps_a=1e-6` → `a_c = 0.999999` (finite).
- **Map/invert:** `u = atanh(0.5) ≈ 0.549306`; back `a = tanh(0.549306) ≈ 0.5`.
- **Fuse two items (w=1):** `a1 = tanh(0.2) ≈ 0.197375`, `a2 = tanh(0.4) ≈ 0.379949` →
  `U = 0.2 + 0.4 = 0.6`, `W = 2`, `a_out = tanh(0.6/2) = tanh(0.3) ≈ 0.291313` (same if order swapped).

Pseudocode (drop-in).

```
def clamp_align(a, eps_a=1e-6):
    return max(-1+eps_a, min(+1-eps_a, a))

def to_u(a, eps_a=1e-6):
    return atanh(clamp_align(a, eps_a))

def from_u(u):
    return tanh(u)

def fuse_alignments(a_list, w_list, eps_a=1e-6, eps_w=1e-12):
    U = sum(w*to_u(a, eps_a) for a, w in zip(a_list, w_list))
    W = sum(w_list)
    return from_u(U / max(W, eps_w))
```

Traditional vs SSM-AI (combining "confidence" cues).

Goal: combine multiple reliability cues without order effects.

- **Traditional:** weighted average in a-space **`a_out = sum(w*a)/sum(w)`** → saturation and edge artifacts.
- **SSM-AI:** work in rapidity: **`U = sum(w*atanh(a))`**; **`a_out = tanh(U/max(W,eps_w))`** → order-invariant, edge-safe fusion with clamps.

QA checklist (pass/fail).

- **Clamp hits:** for any input **`a`**, **`|clamp_align(a)| < 1`**.
- **Round-trip:** **`from_u(to_u(a)) == clamp_align(a)`** within tolerance.
- **Additivity:** **`to_u(a1) + to_u(a2) == to_u( tanh(atanh(a1)+atanh(a2)) )`**.
- **Order-invariance:** **`fuse_alignments([a1,a2],[w1,w2]) == fuse_alignments([a2,a1],[w2,w1])`**.

One-line takeaway. Clamp → map to `u := atanh(a)` → compose additively → return via `a := tanh(u)` — that is the simple engine behind SSM-AI's **bounded, order-invariant** behavior.

2.3 Order-Invariant Streaming Fuse (U/W mean in u-space)

Goal. Combine many alignments into one bounded result that is **independent of order, chunking, or sharding**.

Definition (the fuse).
For items with alignments `a_i` (each clamped so `|a_i|<1`) and weights `w_i >= 0`:

```
U := sum_i w_i * atanh(a_i)
W := sum_i w_i
a_out := tanh( U / max(W, eps_w) )
```

Defaults: `eps_w = 1e-12`. If `W = 0`, define `a_out := 0` (neutral).

Why it is order-invariant.
`U` and `W` are additive:

- **Stream:** `U += w*atanh(a)` ; `W += w`.
- **Batch:** compute `(U_batch, W_batch)` once.
- **Shards:** compute `(U_k, W_k)` per shard and merge by `U := sum_k U_k`, `W := sum_k W_k`.
  Because `tanh/atanh` are inverses on the clamped domain, `a_out` depends only on the **multiset** `{(a_i,w_i)}`, not on order or partitioning.

Do not average in a-space.
Naive `a_out := sum(w*a)/sum(w)` is not invariant at scale and misbehaves near edges. Always fuse in **u-space** (`atanh`), then return via `tanh`.

State to carry in streams.
Carry only `(U, W)`. Do not carry just `a_out` between chunks; if you must, also carry `W` and reconstruct `U := W * atanh(a_out)` before merging.

Worked example (calculator-fast).
Let `a1 = tanh(0.2)`, `a2 = tanh(0.4)`, `a3 = tanh(-0.1)`. Weights: `w1=1`, `w2=2`, `w3=0.5`.
Contributions in `u` are `0.2`, `0.4`, `-0.1`.

```
U = 1*0.2 + 2*0.4 + 0.5*(-0.1) = 0.95
W = 1 + 2 + 0.5 = 3.5
a_out = tanh(0.95/3.5) = tanh(0.27142857) ≈ 0.264954
```

Shuffle check: any permutation yields the same `(U,W)` and thus the same `a_out`.
Shard check: if a shard returns `(U_s,W_s)`, then `tanh( (sum U_s) / (sum W_s) )` equals the single-pass result.

Pseudocode (drop-in, streaming-safe).

```
class Fuse:
    def __init__(self, eps_w=1e-12):
        self.U = 0.0; self.W = 0.0; self.eps_w = eps_w
    def add(self, a, w=1.0, eps_a=1e-6):
        a = max(-1+eps_a, min(1-eps_a, a))
        self.U += w * atanh(a)
        self.W += w
    def merge(self, other):              # shard/worker merge
        self.U += other.U; self.W += other.W
    def value(self):
        if self.W <= 0: return 0.0
        return tanh(self.U / max(self.W, self.eps_w))
```

Edge behavior.

- **Zero weight:** `w=0` contributes nothing.
- **All-zero weight:** `W=0` ⇒ `a_out=0` (neutral).
- **Large magnitudes:** choose weights via manifest, for example `w := |m|^gamma` (`gamma=1` default) or `w := 1` for uniform.
- **Precision:** pick `eps_w` consistent with dtype; for float32, `1e-8` to `1e-12` is typical.

Proof sketch (associative merge).
Let group `G1` have `(U1,W1)` and `G2` have `(U2,W2)`.
Batch fuse: `a_b = tanh( (U1+U2) / (W1+W2) )`.
Streaming via intermediate `a1 = tanh(U1/W1)` gives `atanh(a1) = U1/W1`. If we carry `U/W`, next step computes:
`U' = W1*atanh(a1) + U2 = U1 + U2, W' = W1 + W2, a_s = tanh(U'/W') = a_b`.
Hence **batch == stream == shuffled** when `(U,W)` are the carried state.

Traditional vs SSM-AI (combining reliability over a stream).

Goal: maintain a stable, bounded confidence over long streams and across shards.

- **Traditional:** moving averages in a-space; sensitive to order and saturation; shard recombination is ad-hoc.
- **SSM-AI:** maintain `(U,W)`; add `w*atanh(a)` per item; final `a_out := tanh(U/max(W,eps_w))`. Shards return `(U,W)`; master sums and inverts once. Deterministic, auditable.

Windowed / time-sliced use (optional, manifest-declared).

- **Fixed window:** keep `(U,W)` for last `N` items; on eviction, subtract that item's `w*atanh(a)` and `w`.
- **Time decay:** fold decay into weights, for example `w_i := w_i * exp(-lambda * delta_t)`. The fuse remains unchanged.
- **SSM-Clock synergy:** stamp each checkpoint and compare `a_out` by hour/day via the same `(U,W)` state; roll-ups just sum `(U,W)`.

QA checklist (pass/fail).

- **Order test:** permute inputs; `a_out` unchanged.
- **Shard test:** split into `K` shards; merge `(U,W)`; `a_out` matches single-pass.
- **Zero/empty test:** no items ⇒ `a_out = 0`.
- **Clamp test:** any input with `|a| >= 1` is clamped before `atanh`.

One-line takeaway. Fuse in u-space with `(U,W)`; invert once with `tanh`. That is why SSM-AI streams, batches, and shards all produce the same bounded answer.

2.4 Lane Mul/Div (M2) and Division Policy

Definition (lane only; magnitudes are classical).
For `x1 := (m1,a1)` and `x2 := (m2,a2)` with `|a1|,|a2| < 1`, define the lane for multiplication and division via rapidity:
• **Mul (M2):** `a_mul := tanh( atanh(a1) + atanh(a2) )`
• **Div (M2):** `a_div := tanh( atanh(a1) - atanh(a2) )`
Magnitudes follow classical math: `m_mul := m1*m2`, `m_div := m1/m2` (subject to your system's rules). **Collapse parity holds:** `phi((m*,a_*)) = m_*`.

Why M2.
`atanh` turns the bounded lane into an additive space; combining evidence is stable and associative in `u`-space. Returning with `tanh` keeps results bounded in `(-1,+1)`.

Properties (quick).
• **Commutativity:** mul lane is commutative; div lane is not (matches classical).
• **Identity:** multiplying by a neutral lane `a = 0` leaves the lane unchanged (since `atanh(0) = 0`).
• **Inverse/reciprocal:** `a_rec := tanh( -atanh(a) ) = -a`.
• **Powers:** for integer/scalar `k`, `a_pow := tanh( k * atanh(a) )` (recovers reciprocal when `k = -1`).
• **Clamp-safety:** always apply `a_c := clamp(a, -1+eps_a, +1-eps_a)` before `atanh`.

Division policy (declare once; affects control flow, not `m`).
• **"strict" (default):** when `|m2| <= eps_div`, treat the magnitude division as invalid per your host system; lane math may be computed but must be **flagged**; selection should **fallback** to classical policy.
• **"meadow":** if your host uses totalized division, you may declare this; SSM-AI still **does not alter `m`** — it only records lanes/bands and stamps.
• **"soft":** for UI/analytics only, you may saturate displays of ratios; **never change `m`** inside SSM-AI.
Lane purity: policies act on `a`/routing, **never** on `m`.
Manifest requirement: `division_policy` **must** be declared; defaults to `"strict"` if omitted.

Worked numbers (calculator-fast).
Let `a1 = tanh(0.5) ≈ 0.462117`, `a2 = tanh(0.2) ≈ 0.197375`.

• **Mul:** `a_mul = tanh(0.5 + 0.2) = tanh(0.7) ≈ 0.604367`
• **Div:** `a_div = tanh(0.5 - 0.2) = tanh(0.3) ≈ 0.291313`
Near-edge safety: `a = 0.9999999` with `eps_a=1e-6` → `a_c = 0.999999`; `atanh(a_c)` finite; results remain bounded.

Traditional vs SSM-AI (combining "confidence" via mul/div).

Goal: propagate reliability through products/ratios without blow-ups.
• **Traditional:** `a_mul := a1*a2`; `a_div := a1/a2` (prone to saturation/instability, edge artifacts).
• **SSM-AI:**
`a_mul := tanh(atanh(a1) + atanh(a2))`
`a_div := tanh(atanh(a1) - atanh(a2))`
→ **Bounded**, associative in `u`, **clamp-safe**; `m` follows classical path; `phi((m,a)) = m`.

Pseudocode (drop-in).

```
def clamp_align(a, eps_a=1e-6):
    return max(-1+eps_a, min(+1-eps_a, a))

def lane_mul(a1, a2, eps_a=1e-6):
    return tanh(atanh(clamp_align(a1, eps_a)) + atanh(clamp_align(a2, 
eps_a)))

def lane_div(a1, a2, eps_a=1e-6):
    return tanh(atanh(clamp_align(a1, eps_a)) - atanh(clamp_align(a2, 
eps_a)))

def lane_pow(a, k, eps_a=1e-6):
    return tanh(k * atanh(clamp_align(a, eps_a)))
```

QA checklist (pass/fail).
• **Clamp:** inputs satisfy `|clamp_align(a)| < 1`.
• **Reciprocal:** `lane_mul(a, lane_pow(a, -1)) == 0` (lane identity) within tolerance.
• **Associativity (mul):** `lane_mul(a1, lane_mul(a2, a3)) == lane_mul(lane_mul(a1, a2), a3)`.
• **Edge near zero (div):** under `"strict"`, if `|m2| <= eps_div`, do **not** use the lane for actuation; log/stamp and fallback to declared classical behavior.

One-line takeaway.
Use rapidity addition/subtraction for lane mul/div: `a* := tanh(atanh(a1) +/- atanh(a2))`. It keeps reliability bounded and stable while `m` follows classical math unchanged.

3) Lens → Align → RSI (Single Chooser)

3.1 Contrasts `e` (designing lenses for different AI aspects)

Purpose. A **lens** turns observable evidence into a signed, dimensionless contrast `e`. Positive `e` supports the candidate or choice; negative `e` opposes it. Lenses are declared (not learned here) and differ by aspect (decoding, RAG, search, tools or agents, evaluators, domain adapters like SSM-Chem). The same downstream math then maps `e -> a_in, a_out -> RSI`.

General form (declare once per lens).
`e := ( sum_i alpha_i * P_i - sum_j beta_j * N_j ) / Unit`

- `P_i` = positive evidence terms (for example, citation hits, intent match).
- `N_j` = penalty terms (for example, toxicity gap, contradiction flags, risk).
- `alpha_i, beta_j > 0` = declared weights.
- `Unit > 0` = declared scale so `|e|` sits in a workable range *(helps `tanh(c*e)`)*.
- **Direction convention:** positive supports, negative opposes. Keep this invariant.

Mapping to alignment (preview of 3.2).
`a_in = tanh( -c * e_in )` ; `a_out = tanh( +c * e_out )` with `c > 0` declared. Split evidence into "in" vs "out" channels when helpful; else use one channel and set the other to `0`.

Design rules (make good lenses).

1. **Dimensionless.** Normalize inputs so `e` has no units.
2. **Monotone.** More good evidence ⇒ larger `e`; more risk ⇒ smaller `e`.
3. **Sparse and simple.** 2–5 terms beat complicated mixes; prefer clearly named `P_i/N_j`.
4. **Stable ranges.** Pick `Unit` and `c` so typical `|e|` is in `[0.2, 1.5]`.
5. **Publishable.** Every term is visible in logs; no hidden factors.

Aspect lenses (ready-to-use patterns).

- **A) Decoding / Beam Pick (per-candidate):**
```
e_out := (intent_match + constraint_satisfaction + evidence_gain)
e_in := (policy_gap + contradiction + style_violation)
e := (e_out - e_in) / Unit
```
- **B) RAG (per-document):**
```
e_out := (semantic_gain + citation_hit + source_authority)
e_in := (toxicity_gap + policy_risk + staleness_penalty)
e := (e_out - e_in) / Unit
```
- **C) Search (internet / intranet / local):**
```
e := (alpha*hit_quality + beta*freshness + gamma*semantic_match -
delta*risk_penalty) / Unit
```
- **D) Tools & Agents (per-step):**
```
e_out := (tool_success_rate + schema_match - repair_distance)
```

`e_in := (error_rate + latency_spike + contradiction_rate)`
`e := (e_out - e_in) / Unit`

- **E) Evaluators / Judges (policy & intent):**
  `e := (policy_hits + intent_fit + style_fit - safety_flags - conflict) / Unit`
- **F) Domain adapter (SSM-Chem example):**
  `e := (yield_gain + selectivity_gain - hazard_penalty - sensitivity_risk) / Unit`

Worked mini-examples (calculator-fast).

1. **Decoding lens (single item).**
   Given `intent_match=0.7`, `constraint_satisfaction=0.4`, `evidence_gain=0.3`, `policy_gap=0.2`, `contradiction=0.0`, `style_violation=0.1`, `Unit=1.0`
   `e_out = 1.4` ; **`e_in = 0.3`** ; **`e = (1.4 - 0.3) / 1.0 = 1.1`**.
2. **Search lens (two results).**
   Params `alpha=1.0`, `beta=0.5`, `gamma=0.7`, `delta=0.8`, `Unit=1`.
   • **Result A** `(0.9, 0.6, 0.7, 0.2)` → **`e = 1.53`**
   • **Result B** `(0.8, 0.2, 0.5, 0.5)` → **`e = 0.85`**
   A maps to a larger alignment and higher **`RSI`**.
3. **Tools lens (retry decision).**
   `tool_success_rate=0.8`, `schema_match=0.6`, `repair_distance=0.2`, `error_rate=0.3`, `latency_spike=0.2`, `contradiction_rate=0.1`, `Unit=1`
   `e_out = 1.2`, **`e_in = 0.6`**, **`e = 0.6`** → moderate positive; retry likely if **`RSI_env`** stays above **A0**.

Traditional vs SSM-AI (building a chooser).

Goal: turn heterogeneous signals into a consistent, fair selector.

- **Traditional:** weighted sums → raw score; tuning fragile; unbounded; order/shard effects.
- **SSM-AI:** declare **`e := (sum alpha_i*P_i - sum beta_j*N_j)/Unit`**; map to alignment; pool in **`u`**-space; produce **`RSI in (-1,+1)`**; bounded, order-invariant, comparable under the **same manifest**.

Pseudocode — lens computation (drop-in).

```
def lens_contrast(positive_terms, negative_terms, Unit=1.0):
    # positive_terms, negative_terms: list of (weight, value)
    P = sum(w * v for (w, v) in positive_terms)
    N = sum(w * v for (w, v) in negative_terms)
    return (P - N) / max(Unit, 1e-12)  # dimensionless e

# Examples:

# Decoding (candidate)
e_decode = lens_contrast(
    positive_terms=[(1.0, intent_match), (1.0, constraint_sat), (1.0,
evidence_gain)],
```

```
    negative_terms=[(1.0, policy_gap), (1.0, contradiction), (1.0,
style_violation)],
    Unit=1.0)

# Search (result)
e_search = lens_contrast(
    positive_terms=[(alpha, hit_quality), (beta, freshness), (gamma,
semantic_match)],
    negative_terms=[(delta, risk_penalty)],
    Unit=1.0)
```

QA checklist (for a good lens).

- **Sign sanity:** raise a positive term ⇒ `e` increases; raise a penalty ⇒ `e` decreases.
- **Range sanity:** typical `|e|` within `[0.2, 1.5]` (adjust `Unit` or weights).
- **Stability:** small input perturbations ⇒ small changes in `e`.
- **Transparency:** all terms/weights logged and stamped; no hidden multipliers.

One-line takeaway. A **lens** is a small, published formula that converts evidence into a signed contrast `e`; everything downstream (**alignment, pooling, RSI**) is universal and identical across aspects.

3.2 Symmetric Maps → Alignment

`a_in = tanh(-c*e_in), a_out = tanh(+c*e_out)`

Purpose. Turn signed contrasts `e_in` (penalties) and `e_out` (support) into bounded alignments `a_in, a_out in (-1,+1)` with a single gain knob `c > 0`. This keeps evidence interpretable, comparable, and safe at edges.

Definition (two-channel map).
```
a_in := tanh( -c * e_in )
a_out := tanh( +c * e_out )
```

- `c` is a declared lens gain (per-lens, per-study).
- `tanh` is monotone and odd: larger support ⇒ larger `a_out`; larger penalties ⇒ more negative `a_in`.
- Clamp is implicit: outputs are strictly inside `(-1,+1)`.

Why symmetric?

- **Sign clarity:** support and penalty push in opposite directions by construction.
- **Scale robustness:** the same `c` governs both channels; tuning is simple and transparent.
- **Bounded behavior:** no matter how large `|e|` gets, `|a| < 1`.

Choosing `c` (practical).

- Aim for typical `|c*e| in [0.3, 1.2]` so `tanh` is responsive but not saturated.

- Start with `c = 1.0` if `e` is already normalized (`Unit = 1`).
- If `|a|` saturates `> 0.9` often, reduce `c` or increase `Unit`; if `a` hovers near `0`, increase `c` or reduce `Unit`.

Single-channel variant (simple lenses).
If a lens emits a single contrast `e`, use: `a := tanh( c * e )`. You can feed `a` directly into pooling or set `a_out := a`, `a_in := 0`.

Worked minis (calculator-fast).

- **Mild support:** `e_out = 0.4`, `c = 1` → `a_out ≈ tanh(0.4) ≈ 0.379949`.
- **Strong penalty:** `e_in = 1.0`, `c = 0.8` → `a_in ≈ tanh(-0.8) ≈ -0.664037`.
- **Balanced:** `e_out = 0.7`, `e_in = 0.5`, `c = 1` → `a_out ≈ 0.604368`, `a_in ≈ -0.462117`.

Traditional vs SSM-AI (bounded cue).

Goal: convert heterogeneous signals into a safe, comparable confidence.

- **Traditional:** weighted sums → raw score; can over/underflow; hard to interpret.
- **SSM-AI:** map signed contrasts via `tanh` with one gain `c`:
  `a_in = tanh(-c*e_in)` ; `a_out = tanh(+c*e_out)` → bounded, monotone, transparent; ready for order-invariant pooling.

Pseudocode (drop-in).

```
def map_to_alignment(e_in=0.0, e_out=0.0, c=1.0):
    a_in  = tanh(-c * e_in)
    a_out = tanh(+c * e_out)
    return a_in, a_out

# Single-contrast form
def align_single(e, c=1.0):
    return tanh(c * e)  # in (-1,+1)
```

QA checklist (pass/fail).

- **Monotonicity:** increase `e_out` ⇒ `a_out` increases; increase `e_in` ⇒ `a_in` decreases.
- **Bounds:** `|a_in| < 1` and `|a_out| < 1` for all finite `e`.
- **Gain sanity:** adjusting `c` scales responsiveness without breaking bounds.

One-line takeaway. Use `tanh` with a single gain `c` to convert contrasts into safe, bounded alignments—clean, interpretable inputs for the universal `RSI` chooser.

3.3 Chooser — RSI (bounded selection index)

Purpose. Turn many alignments into one bounded chooser `RSI in (-1,+1)` that is **transparent, order-invariant, and comparable** across tasks, models, and vendors.

Definition (two-channel form).

```
U_in := sum_i w_i * atanh(a_in_i)
V_out := sum_j w_j * atanh(a_out_j)
W_in := sum_i w_i
RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )
```

Defaults: `eps_w = 1e-12`. **Weights (default):** `w := |m|^gamma` with `gamma = 1`. Uniform option: `w := 1` (if declared).

Single-channel variant.
`U := sum_k w_k * atanh(a_k)` ; `W := sum_k w_k` ; `RSI := tanh( U / max(W, eps_w) )`.

Why it works.

- **Bounded & monotone:** `RSI in (-1,+1)`; more support lifts `RSI`; more penalty lowers it.
- **Order-invariant:** uses the `U/W` mean in `u`-space; **batch == stream == shuffled**.
- **Comparable:** same **manifest** ⇒ apples-to-apples across prompts/models/providers.
- **Zero-knowledge safe:** if `W_in = 0` (no evidence), define `RSI := 0` (neutral).

With the calm gate (optional).
`RSI_env := g_t * RSI` with `g_t in [0,1]` derived from bounded telemetry. Gate acts on **alignment only**; `m` is never altered.

Worked mini-examples (calculator-fast).

- **A) One item (matches earlier numbers).**
  `e_in = 0.2`, `e_out = 0.5`, `c = 1`, `w = 1` →
  `a_in = tanh(-0.2)`, `a_out = tanh(+0.5)` →
  `U_in = -0.2`, `V_out = +0.5`, `W_in = 1` →
  `RSI = tanh(0.7) ≈ 0.604368` → **band A+**.
- **B) Multiple signals (uniform weights).**
  `a_out = [tanh(0.6), tanh(0.3)]`, `a_in = [tanh(0.2)]`, all `w=1` →
  `V_out = 0.6 + 0.3 = 0.9`, `U_in = 0.2`, `W_in = 1` →
  `RSI = tanh(0.7) ≈ 0.604368` (same net effect; order/grouping invariant).
- **C) Calm gate in turbulence.**
  From A, `RSI ≈ 0.604368`, telemetry gives `g_t = 0.5` →
  `RSI_env = 0.5 * 0.604368 ≈ 0.302184` (drops to **A0**; a retry might pause).

Traditional vs SSM-AI (selection logic).

Goal: pick the best option fairly and reproducibly across vendors.

- **Traditional:** weighted sums or ad-hoc mixes; unbounded scores; order/shard artifacts; hard comparisons.
- **SSM-AI:** map evidence to bounded alignments; fuse in **u**-space; compute **RSI in (-1,+1)**; optionally apply **RSI_env = g_t * RSI**; choose by **RSI** or **RSI_env**; **m** remains untouched (**phi((m,a)) = m**).

Pseudocode (drop-in).

```
def rsi_from_alignments(a_in_items, a_out_items, eps_w=1e-12):
    U_in = V_out = W_in = 0.0
    for (a_in,  w) in a_in_items:
        U_in  += w * atanh(a_in)
        W_in  += w
    for (a_out, w) in a_out_items:
        V_out += w * atanh(a_out)
    if W_in <= 0.0:
        return 0.0  # neutral when no "in" evidence
    return tanh( (V_out - U_in) / max(W_in, eps_w) )

def choose(candidates, g_t=1.0):
    # each candidate supplies lists: a_in_items, a_out_items (already
clamped)
    scored = []
    for cid, ain, aout in candidates:
        rsi = rsi_from_alignments(ain, aout)
        scored.append((cid, g_t * rsi))    # RSI_env
    return max(scored, key=lambda kv: kv[1])[0]
```

Design notes & guardrails.

- **Weights (w).** Start with **w := |m|^gamma** (gamma=1) to reflect signal strength; switch to uniform (**w := 1**) for pure comparability. Declare once in the manifest.
- **Zero/empty cases.** If there is no penalty channel, set **U_in := 0**, **W_in := 1** (or use single-channel form).
- **Clamps.** Ensure all alignments satisfy **|a| < 1** (use **eps_a**) before **atanh**.
- **Bands.** Defaults: **A++ (>= +0.90)**, **A+ ([+0.60,+0.90))**, **A0 ((-0.60,+0.60))**, **A- ((-0.90,-0.60])**, **A-- (<= -0.90)**.

One-line takeaway. **RSI := tanh((V_out - U_in)/max(W_in, eps_w))** is the single, bounded chooser that makes decisions **fair, stable, and comparable** — while **phi((m,a)) = m** keeps your numbers unchanged.

3.4 Bands (A++/A+/A0/A-/A--) and thresholds

Purpose. Turn continuous alignment into actionable, readable classes for routing, UI badges, policies, and dashboards while keeping numbers unchanged (`phi((m,a)) = m`). Bands apply to either component lanes (`a`) or the final chooser (`RSI`).

Defaults (normative).

```
A++ : a >= +0.90
A+ : +0.60 <= a < +0.90
A0 : -0.60 < a < +0.60
A- : -0.90 < a <= -0.60
A-- : a <= -0.90
```

Use the same table for `RSI` (recommended for decisions). Any override must be declared in the manifest.

Why bands (practical).
• Instant read: `A+` or `A-` is faster to interpret than decimals.
• Stable routing: simple policies like `retry only if RSI_env >= A+`.
• Comparable: fixed table enables cross model or vendor comparisons.
• Auditable: logs carry both `a` or `RSI` and the band.

Which quantity to band.
• Final decisions: band `RSI` or `RSI_env`.
• Component signals: band intermediate `a` to expose weak links.

Hysteresis (avoid flapping).
Use tiny guard margins `h_up`, `h_dn` (for example, `0.02`). Example for the `A+` boundary at `0.60`:
`enter_A+` when `x >= 0.60 + h_up`
`leave_A+` when `x < 0.60 - h_dn`
Apply similarly at `0.90`, `-0.60`, `-0.90`.

Calm-gated bands.
When using the gate, band on `RSI_env := g_t * RSI`. The gate scales alignment only. `m` is unchanged and `phi((m,a)) = m` always holds.

Worked minis (calculator-fast).
• `RSI = 0.604368` → `A+`.
• `RSI_env = 0.302184` → `A0` (neutral, likely pause or retry later).
• `a = -0.875` → `A-` (watch list).
• `a = +0.915` → `A++` (strong pass).

Traditional vs SSM-AI (decision thresholds).

Goal: enforce consistent, human-readable decision rules.
• Traditional: raw scores with bespoke cutoffs, difficult cross team or vendor comparison.
• SSM-AI: fixed bands on `RSI` (bounded, comparable). Policies like promote if `RSI_env >= A+`, quarantine if `<= A-`, audit if `A0` with large drift. `m` remains untouched.

Pseudocode (drop-in).

```
def to_band(x):  # x is a or RSI, in (-1,+1)
    if x >= 0.90:  return "A++"
    if x >= 0.60:  return "A+"
    if x > -0.60:  return "A0"
    if x > -0.90:  return "A-"
    return "A--"

def to_band_hysteresis(x, prev_band, h_up=0.02, h_dn=0.02):
    # stickiness around boundaries to reduce flicker
    if prev_band == "A++" and x >= 0.90 - h_dn: return "A++"
    if prev_band == "A+"  and (0.60 - h_dn) <= x < (0.90 + h_up): return
"A+"
    if prev_band == "A0"  and (-0.60 - h_dn) <  x < (0.60 + h_up): return
"A0"
    if prev_band == "A-"  and (-0.90 - h_dn) <  x <= (-0.60 + h_up): return
"A-"
    if prev_band == "A--" and x <= -0.90 + h_up: return "A--"
    return to_band(x)
```

Manifest (keys to publish).

```
"bands": {
  "A++": 0.90, "A+": 0.60, "A0": -0.60, "A-": -0.90, "A--": -1.00,
  "hysteresis": { "h_up": 0.02, "h_dn": 0.02 },
  "band_on": "RSI_env"  # or "RSI" or "a"
}
```

QA checklist (pass/fail).
• Boundary tests: exact values `0.90`, `0.60`, `-0.60`, `-0.90` map as specified.
• Monotonicity: if `x1 < x2` then `band(x1)` is not strictly better than `band(x2)`.
• Hysteresis: increasing or decreasing ramps respect stickiness, no flicker at `±0.60`.
• Gate awareness: `band(RSI_env)` changes with `g_t`, while `m` never changes.

One-line takeaway. Bands turn bounded alignment into crisp, portable decisions (`A++ … A--`) with optional hysteresis, enabling simple rules that everyone can read, route, and audit.

3.5 Lens calibration quickstart (2 minutes, reproducible)

Goal. Pick `Unit` and `c` so `tanh(c*e)` is informative (not saturated, not tiny).

Protocol (declare in manifest).

1. **Sample.** Collect `K` recent items' raw terms (no PII).
2. **Compute provisional `e_raw`.** Use your declared `alpha/beta` weights, with `Unit := 1`.
3. **Set `Unit`.** Let `q := median(|e_raw|)`; set `Unit := max(q, 1e-6)` so typical `|e| ≈ 1`.

4. **Set c.** Start with `c := 1.0`;
   • if `> 10%` of `|a|` exceed `0.9`, reduce `c` (e.g., `c := 0.7`).
   • if `> 70%` of `|a| < 0.1`, increase `c` (e.g., `c := 1.3`).
5. **Freeze.** Stamp `Unit`, `c`, and `alpha/beta` in the manifest; re-use across vendors.

Lane mapping (for reference).
`a_in := tanh(-c * e)`, `a_out := tanh(+c * e)` with `e := e_raw / Unit`.

Chooser and zero-evidence guard (normative).
`RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`
If `W_in == 0`, set `RSI := 0` and tag `band = "A0"` with reason `insufficient_evidence`. Do not impute confidence when no evidence is pooled.

Sanity metrics (log once/day).
• **Saturation rate:** `mean(|a| > 0.9)` should be small.
• **Dead-zone rate:** `mean(|a| < 0.1)` should be moderate.
• **Drift watch:** track median `RSI` and band histogram; large shifts trigger review.

QA checklist (pass/fail).
• **Clamps applied:** `a_c := clamp(a, -1+eps_a, +1-eps_a)` before any `atanh`.
• **Unit stability:** `Unit` recompute changes bands by less than one step on a holdout slice.
• **No-evidence path:** `W_in == 0` yields `RSI = 0`, `band = "A0"`, reason recorded.
• **Determinism:** same manifest → identical `a`, `RSI`, `band` on replay.

3.6 Ready-to-paste manifest snippet (lens block)

```
"lens": {
  "id": "decode.v1",
  "positives": [{"name":"intent_match","alpha":1.0},
               {"name":"constraint_satisfaction","alpha":1.0},
               {"name":"evidence_gain","alpha":1.0}],
  "negatives": [{"name":"policy_gap","beta":1.0},
               {"name":"contradiction","beta":1.0},
               {"name":"style_violation","beta":1.0}],
  "unit": 1.0,
  "gain_c": 1.0,
  "weights_policy": "abs_m_pow_gamma",
  "gamma": 1.0,
  "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00},
  "eps_a": 1e-6,
  "eps_w": 1e-12
}
```

---

4) Calm Gate (Volatility Governor)

4.1 Telemetry Lanes (inputs to the gate)

Purpose. When the environment shakes (contradictions, tool errors, latency spikes), throttle **alignment only** so selection/routing becomes conservative while `m` stays untouched. The gate produces `g_t in [0,1]` and we apply `RSI_env := g_t * RSI`.

Telemetry lanes (normalize each to `[0,1]`).
Declare simple, bounded inputs (start with 3–5). Examples and ready-to-use normalizations:

- `F_t` (contradiction / policy-violation fraction): `F_t := clamp(frac_violations, 0, 1)`.
- `D_t` (semantic drift between steps/hops): `D_t := clamp((1 - cos_sim) / max(1 - cos_ref, 1e-12), 0, 1)` where `cos_sim in [-1,1]`, `cos_ref` is a benign baseline (e.g., `0.95`).
- `L_t` (latency pressure): `L_t := clamp( (latency - p50) / max(p95 - p50, 1e-12), 0, 1 )`.
- `E_t` (tool/API error rate): `E_t := clamp(error_count / max(calls,1), 0, 1)`.
- `V_t` (response/token churn): `V_t := clamp(edit_distance / max(length,1), 0, 1)`.
- `Q_t` (queue depth pressure, optional): `Q_t := clamp( queue_depth / q95, 0, 1 )`.

Instant gate (one-liner).
With nonnegative weights `wF,wD,wL,wE,wV,wQ` and `W := wF + wD + wL + wE + wV + wQ`:
`g_inst := clamp( 1 - (wF*F_t + wD*D_t + wL*L_t + wE*E_t + wV*V_t + wQ*Q_t) / max(W, eps_g), 0, 1 )`
Defaults: `eps_g = 1e-12`. *(Start with equal weights if unsure.)*

Smoothed gate (avoid jitter). Use a short EWMA with `rho in (0,1]`:
`g_t := (1 - rho) * g_{t-1} + rho * g_inst`
`g_t := clamp( max(g_min, g_t), 0, 1 )` *(e.g., `g_min = 0.20`)*.
Typical starting defaults: `rho = 0.20`, `g_min = 0.00`.

Application (alignment-only).
Once per decision: `RSI_env := g_t * RSI`
Invariant: `phi((m,a)) = m` always; the gate scales only alignment outputs (`RSI`, bands), never magnitudes.

Worked numbers (calculator-fast).
Suppose `F_t=0.20`, `D_t=0.10`, `L_t=0.30`, `E_t=0.15`, `V_t=0.20`, equal weights (`W=5`):
`g_inst = clamp( 1 - (0.20+0.10+0.30+0.15+0.20)/5 , 0, 1 ) = clamp( 1 - 0.95/5 , 0, 1 ) = 0.81`
`RSI = 0.604368` → `RSI_env = 0.81 * 0.604368 ≈ 0.489534` → band **A0**.
The system pauses/routes conservatively while `m` remains identical.

Traditional vs SSM-AI (handling turbulence).

Goal: reduce waste and misroutes when the system is shaky.

- **Traditional:** fixed retries/timeouts; ignores semantic drift/contradiction; costs spike during partial outages.
- **SSM-AI:** compute **`g_t`** from bounded telemetry; apply **`RSI_env := g_t * RSI`** (alignment-only). Below **A0**, back off; above **A+**, proceed. **`m`** is untouched; choices are stamped.

Pseudocode (drop-in).

```
def calm_gate(F, D, L, E, V, Q=0.0, w=(1,1,1,1,1,0), rho=0.2, state=None,
eps_g=1e-12, g_min=0.0):
    wF, wD, wL, wE, wV, wQ = w
    W = max(wF + wD + wL + wE + wV + wQ, eps_g)
    g_inst = max(0.0, min(1.0, 1.0 - (wF*F + wD*D + wL*L + wE*E + wV*V +
wQ*Q) / W))
    g_prev = 1.0 if state is None else state["g"]
    g = (1 - rho) * g_prev + rho * g_inst
    g = max(g_min, min(1.0, g))
    return {"g": g}  # keep in state between steps

def apply_gate(RSI, gate_state):
    return gate_state["g"] * RSI  # alignment-only scaling
```

Time-aware slices (with SSM-Clock Stamp).

- Stamp **`g_t`** and **`RSI_env`** per decision: **`"...|g=0.81|RSI_env=0.489534|..."`**.
- Roll up by hour/day to spot instability windows without any retraining.
- Route maintenance or vendor switchovers when sustained **`g_t`** dips appear.

Guardrails (declare once).

- **Gate purity:** **`g_t`** may change routing, never **`m`**.
- **Visibility:** log **`(g_t, F_t, D_t, L_t, E_t, V_t, Q_t)`** beside **`RSI`**.
- **Fallback:** if telemetry is missing/corrupt or NaN/out-of-range, clamp to **`[0,1]`**, set **`g_t := 1`**, and flag; do not guess.

One-line takeaway. **`g_t in [0,1]`** turns turbulence into a bounded, transparent brake on alignment (**`RSI_env := g_t * RSI`**) — conservative when shaky, assertive when clear — while **`m`** remains pristine.

4.2 Gate formula (how `g_t` shapes selection, alignment-only)

Instant gate (default).
Normalize telemetry lanes to `[0,1]` and combine with declared nonnegative weights:
`W := wF + wD + wL + wE + wV + wQ`
`mix := (wF*F_t + wD*D_t + wL*L_t + wE*E_t + wV*V_t + wQ*Q_t) / max(W, eps_g)`
`g_inst := clamp( 1 - mix, 0, 1 )`
Defaults: `eps_g = 1e-12`. *(Equal weights are fine to start.)*

Safety notch (optional).
A single severe lane can force conservative behavior:
`sev := max(F_t, D_t, E_t)` *(choose lanes you consider critical)*
`g_sev := clamp( 1 - (sev - s_thr) / max(1 - s_thr, eps_g), 0, 1 )` with `s_thr in [0,1)`
`g_inst := min(g_inst, g_sev)`

Smoothing & floors (avoid jitter).
`g_t := (1 - rho) * g_{t-1} + rho * g_inst` with `rho in (0,1]`, e.g., `0.2`
`g_t := clamp( max(g_min, g_t), 0, 1 )` *(e.g., `g_min = 0.20`)*

How the gate acts (two choices; pick one in the manifest).

- **Mode `"mul"` (default):** scale in the bounded space directly → `RSI_env := g_t * RSI`.
- **Mode `"u_scale"` (curvature-preserving):** scale in `u`-space, then re-bound → `RSI_env := tanh( g_t * atanh(RSI) )`.
  (Note: `|tanh(g*atanh(x))| <= |x|` for `g in [0,1]`, equality when `g=1`.)
  Use `"mul"` for simple proportional behavior; use `"u_scale"` if you prefer gentler damping near high-confidence regions.

Band-aware application.
Compute bands **after** applying the gate (`band := to_band(RSI_env)`), so turbulence directly affects routing thresholds without altering `m`.

Worked minis (calculator-fast).
Inputs: `RSI = 0.70`, `F_t=0.20`, `D_t=0.10`, `L_t=0.30`, `E_t=0.15`, `V_t=0.20`, equal weights (`W=5`), `rho=1` *(no smoothing)*.

- `mode = "mul"`: `mix = 0.95/5 = 0.19` → `g_t = 1 - 0.19 = 0.81` → `RSI_env = 0.81 * 0.70 = 0.567` → band **A0**.
- `mode = "u_scale"`: `RSI_env = tanh( 0.81 * atanh(0.70) ) = tanh(0.7025) ≈ 0.606` → band **A+**.
  Observation: `"u_scale"` damps less at higher `RSI`, which some teams prefer.

Pseudocode (drop-in).

```
def gate_value(F,D,L,E,V,Q=0.0, w=(1,1,1,1,1,0), s_thr=None, rho=0.2,
state=None,
               eps_g=1e-12, g_min=0.0):
    wF,wD,wL,wE,wV,wQ = w
    W = max(wF+wD+wL+wE+wV+wQ, eps_g)
    mix = (wF*F + wD*D + wL*L + wE*E + wV*V + wQ*Q) / W
    g_inst = max(0.0, min(1.0, 1.0 - mix))
    if s_thr is not None:
        sev   = max(F, D, E)  # or any subset you choose
        g_sev = max(0.0, min(1.0, 1.0 - (sev - s_thr)/max(1.0 - s_thr,
eps_g)))
        g_inst = min(g_inst, g_sev)
    g_prev = 1.0 if state is None else state["g"]
    g      = (1 - rho) * g_prev + rho * g_inst
    g      = max(g_min, min(1.0, g))
    return {"g": g}

def apply_gate(RSI, g, mode="mul"):
    if mode == "u_scale":
        return tanh(g * atanh(RSI))
    return g * RSI  # default
```

Traditional vs SSM-AI (handling turbulence).

Goal: throttle risky actions without distorting values.

- **Traditional:** fixed retry/timeouts; global panic switches that ignore semantic signals; unbounded scores; fragile comparisons under stress.
- **SSM-AI:** `g_t` from bounded telemetry; `RSI_env := gate(RSI)`; choose by `RSI_env` bands (`A++ … A--`); `m` remains identical (`phi((m,a)) = m`). Manifest declares weights, smoothing, safety notch, and gate mode.

4.3 Knobs & Purity (manifest + acceptance)

Manifest keys (suggested).

```
"gate": {
  "weights": {"F":1.0, "D":1.0, "L":1.0, "E":1.0, "V":1.0, "Q":0.0},
  "rho": 0.20,
  "g_min": 0.00,
  "eps_g": 1e-12,
  "safety_notch": {"enabled": false, "s_thr": 0.80},
  "mode": "mul",   # or "u_scale"
  "band_policy": {
    "promote_if": "RSI_env >= A+",
    "pause_if":   "RSI_env in A0",
    "block_if":   "RSI_env <= A-"
  }
}
```

Gate construction (alignment-only).
Let telemetry lanes be `F,D,L,E,V,Q` in `(-1,+1)`. Define a provisional score
`s := ( w_F*F + w_D*D + w_L*L + w_E*E + w_V*V + w_Q*Q ) / max(sum_w, eps_g)`
with `sum_w := w_F + w_D + w_L + w_E + w_V + w_Q`. Map to a nonnegative **calm governor**
`g_inst := clamp( 1 - rho * ( (1+s)/2 ), g_min, 1 ).`
If `safety_notch.enabled` and `|RSI| >= s_thr`, apply a notch: `g_inst := min(g_inst, 1 - rho).`
The time-smoothed gate is `g_t := g_inst` (or your declared smoother), **bounded in `[g_min,1]`**.

Finalizer (choose exactly one, declared in manifest).
• **"mul" mode:** `RSI_env := g_t * RSI`
• **"u_scale" mode:** `RSI_env := tanh( g_t * atanh(RSI) )`
Both keep `|RSI_env| < 1`. **`m` is never changed.**

Purity rules (non-negotiable).
• **Alignment-only:** the gate acts on `RSI/bands`; **never** on `m`.
• **Visibility:** log `(F,D,L,E,V,Q)`, `g_inst`, `g_t`, `mode`, and resulting `RSI_env`.
• **Determinism:** same inputs + manifest ⇒ same `g_t` and `RSI_env`.
• **Fallback:** if any telemetry lane is missing, compute with available lanes; if all missing, set `g_t := 1` and **flag**.

Acceptance tests (calculator-fast).
• **Purity:** for any item, `phi((m,a)) == m` before/after gating.
• **Monotonicity:** if any telemetry lane worsens (decreases when higher is better), `g_t` must **not increase**.
• **Mode sanity:** `"u_scale"` and `"mul"` both yield `|RSI_env| < 1`.
• **Band consistency:** `to_band(RSI_env)` follows manifest thresholds (with hysteresis if declared).
• **Replay:** given a stamped log, recompute `g_t` and `RSI_env` bit-for-bit (within dtype tolerance).

One-line takeaway. Declare the gate once, make it visible, and keep it **alignment-only** — `m` stays pristine while `g_t` turns turbulence into a transparent, bounded brake on decisions.

4.4 Ready-made gate presets (pick one to start)

Preset A — Conservative (production hardening).
```
weights: F=1.5, D=1.0, L=1.0, E=1.5, V=0.5, Q=0.5
rho: 0.30
g_min: 0.10
mode: "mul"
safety_notch: enabled, s_thr=0.80
policy: promote_if "RSI_env >= A+", pause_if "RSI_env in A0", block_if
"RSI_env <= A-"
```

Preset B — Agile (experimentation).
```
weights: F=1.0, D=0.5, L=0.5, E=1.0, V=0.5, Q=0.0
rho: 0.15
g_min: 0.00
mode: "u_scale"
safety_notch: disabled
policy: promote_if "RSI_env >= A0", pause_if "RSI_env in A0 with drift_up",
block_if "RSI_env <= A-"
```

4.5 Two-minute calibration plan (reproducible)

1. Collect **`1–2k`** recent steps with **`(F,D,L,E,V,Q)`** and baseline **`RSI`**.
2. Normalize lanes using the formulas above; confirm each is in **`[0,1]`**.
3. Grid over weights on a coarse simplex *(e.g.,* ***`w in {0.5,1.0,1.5}`****)* and **`rho in {0.1,0.2,0.3}`**, **`mode in {"mul","u_scale"}`**.
4. Score presets by downstream KPIs *(retry rate ↓, time-to-correct ↓, bad-escalations ↓)* while verifying purity **`(phi((m,a)) = m)`**.
5. Freeze manifest, stamp it, and roll out.

4.6 Stamp fields (minimal, ASCII)

Append to your existing stamp line:
`"|g=" + fmt(g_t) + "|RSI_env=" + fmt(RSI_env) + "|gate_mode=" + mode +`
`"|lanes=" + fmt(F_t,D_t,L_t,E_t,V_t,Q_t)`
This keeps replay deterministic and audits frictionless.

Note: See Appendix A for ready-made presets and acceptance vectors.

---

5) Path-Level Scoring (Chains, Tools, Agents)

5.1 Step Scores & Priors in u-space

Goal. Score each step of a chain or agent with a bounded index, then combine steps order-invariantly in **5.2**. All step math is observation-only; classical magnitudes remain untouched (`phi((m,a)) = m`).

Per-step score (from Section 3). For step `s`, compute a bounded chooser:
`RSI_s := tanh( (V_out_s - U_in_s) / max(W_in_s, eps_w) )`

Zero-evidence guard (normative). If `W_in_s == 0`, set `RSI_s := 0` and tag `band = "A0"` with reason `insufficient_evidence`.

Optional calm gate (alignment-only).
`RSI_used_s := g_s * RSI_s`
or (mode `"u_scale"`) `RSI_used_s := tanh( g_s * atanh(RSI_s) )`
`g_s in [0,1]` comes from step-local telemetry (errors/latency/contradiction). **Invariant:** `m` is never altered.

Why move to u-space. Path math composes additively in rapidity:
`u_s := atanh( clamp(RSI_used_s, -1+eps_a, +1-eps_a) )`
This makes pooling associative, shardable, and order-invariant.

Priors (tiny, transparent nudges in u-space). If a step has a declared reliability signal, publish a bounded prior `b_s in [-1,+1]` and small gain `beta >= 0`:
`u'_s := u_s + beta * b_s -> RSI'_s := tanh(u'_s)`
Rules. (i) **Bounded & public:** publish `b_s`, `beta`. (ii) **Never edit `m`.** (iii) `beta = 0` disables the prior.

Step record (log it). `(m_s, RSI_s, g_s, RSI_used_s, b_s, beta, RSI'_s, U_s := atanh(RSI'_s), W_s)`
Pick `W_s` from your manifest (uniform `W_s := 1`, or strength-aware `W_s := |m_s|^gamma`).

Worked minis (calculator-fast; 6-dec rounding).
A) One step with gate and prior.
`tanh(0.700000) = 0.604368`, `g_s = 0.800000` → `RSI_used_s = 0.483494`
`u_s = atanh(0.483494) = 0.527535`
Prior: `b_s = +0.300000`, `beta = 0.200000` → `u'_s = 0.587535`
`RSI'_s = tanh(0.587535) = 0.528120`

B) Another step, neutral prior.
`tanh(0.400000) = 0.379949`, `g_s = 1.000000`, `beta = 0` → `u'_s = 0.400000`, `RSI'_s = 0.379949`

Traditional vs SSM-AI (step scoring).

• **Traditional:** heuristic mixtures; unbounded scores; order effects.
• **SSM-AI:** declare lens → `RSI_s`; gate → `RSI_used_s`; u-space → `u_s`; tiny prior → `u'_s`. `m` stays pristine (`phi` parity). Stamped and auditable.

Pseudocode (drop-in, per step).

```
def step_score(e_in_out_items, g_s=1.0, beta=0.0, b_s=0.0,
               eps_w=1e-12, eps_a=1e-6, gate_mode="mul"):
    U_in = V_out = W_in = 0.0
    for (e_in, e_out, w) in e_in_out_items:
        a_in  = tanh(-e_in); a_out = tanh(+e_out)
        a_in  = max(-1+eps_a, min(1-eps_a, a_in))
        a_out = max(-1+eps_a, min(1-eps_a, a_out))
        U_in  += w * atanh(a_in)
```

```
        V_out += w * atanh(a_out)
        W_in  += w
    if W_in <= 0:
        RSI_s = 0.0
        # band := "A0", reason := "insufficient_evidence"
    else:
        RSI_s = tanh((V_out - U_in) / max(W_in, eps_w))
    RSI_used = tanh(g_s * atanh(RSI_s)) if gate_mode == "u_scale" else (g_s
* RSI_s)
    RSI_used = max(-1+eps_a, min(1-eps_a, RSI_used))
    u_s  = atanh(RSI_used)
    u_ps = u_s + beta * b_s
    return {"RSI_s": RSI_s, "RSI_used": RSI_used, "u_step": u_ps}
```

Acceptance checklist. Parity `phi((m_s,a_s)) = m_s`; **boundedness** `|RSI| < 1`; **monotone signs**; `beta = 0` ⇒ no prior effect; **determinism** under identical inputs; **zero-evidence** path yields `RSI_s = 0`, `band = "A0"`, reason recorded.

One-line takeaway. Score each step to a bounded `RSI_s`, gate it, convert to `u_s`, add tiny public priors there—ready for order-invariant pooling in **5.2**.

5.2 Path Pooling & Reporting

`RSI_path := tanh( U_path / max(W_path, eps_w) )`

Goal. Produce a single bounded, order-invariant, shard-safe **path score**.

Definition (fuse steps in u-space).
`u'_s := atanh( clamp(RSI'_s, -1+eps_a, +1-eps_a) )`
`U_path := Σ_s w_s * u'_s ; W_path := Σ_s w_s ; RSI_path := tanh( U_path / max(W_path, eps_w) )`
Defaults: `eps_a=1e-6`, `eps_w=1e-12`.

Streaming / branching.
• Extend: **`U += w_s*u'_s`**, **`W += w_s`**.
• Merge shards: sum `(U,W)`; invert once.
• Compare branches: pick by **`RSI_path`** *(or `RSI_plan_env` if gating).*

Reporting. Log `U_path`, `W_path`, `RSI_path`, `band_path := to_band(RSI_path).`

Plan gate (optional).
`RSI_plan_env := g_plan * RSI_path` or **`tanh(g_plan*atanh(RSI_path))`** *(mode `"u_scale"`).*
Invariant: `phi((m,a)) = m` everywhere.

Worked numbers (6-dec rounding).
`u'_1 = 0.587535` *(from `RSI'_1 = 0.528120`)*
`u'_2 = 0.400000` *(from `RSI'_2 = 0.379949`)*
`u'_3 = 0.200000`

`U_path = 1.187535`, `W_path = 3` → `RSI_path = tanh(0.395845) = 0.376388` → **A0**
`g_plan = 0.800000` → `RSI_plan_env = 0.301110` → **A0**

Rollback stack (Δ-based). Push `(w_s*u'_s, w_s)`, pop to unwind; always recompute `RSI_path` from `(U,W)`.

Pseudocode (branch-safe).

```
class PathScore:
    def __init__(self, eps_a=1e-6, eps_w=1e-12):
        self.U=0.0; self.W=0.0; self.eps_a=eps_a; self.eps_w=eps_w;
self.stack=[]
    def add_step(self, RSI_used, beta=0.0, b_s=0.0, w=1.0):
        a = max(-1+self.eps_a, min(1-self.eps_a, RSI_used))
        u = atanh(a) + beta*b_s
        self.U += w*u; self.W += w; self.stack.append((w*u, w))
    def undo_step(self):
        if not self.stack: return
        dU,dW = self.stack.pop(); self.U -= dU; self.W -= dW
    def rsi_path(self):
        return 0.0 if self.W<=0.0 else tanh(self.U / max(self.W,
self.eps_w))

def pick_best_branch(branches, g_plan=1.0, mode="mul"):
    def env_rsi(ps):
        r = ps.rsi_path()
        return tanh(g_plan*atanh(r)) if mode=="u_scale" else g_plan*r
    return max(branches, key=lambda kv: env_rsi(kv[1]))[0]
```

Weights (`w_s`). Uniform (`1`) for comparability; strength-aware (`|m_s|^gamma`) to reflect magnitude strength. Declare once in the manifest.

Stamp (recommendation).
`...|U_path=1.187535|W_path=3|RSI_path=0.376388|band=A0|g_plan=0.80|RSI_env=0.301110|...`

Acceptance checklist. Order invariance, shard invariance, rollback correctness, boundedness, determinism.

One-line takeaway. Keep per-step contributions in u-space, track `(U,W)`, and publish `RSI_path := tanh(U/W)`—a single, bounded, order-proof score for whole chains and plans.

5.3 Failure Containment & Rollback (deterministic, bounded, stamp-ready)

Goal. Contain bad steps quickly, recover to last-known-good, branch safely—without mutating `m`.

Triggers (declare once). Band breach, sharp drop `ΔRSI_path <= delta_thr`, gate shock `g < g_min`, policy hit, budget guard.

Mechanism (additive Δ stack). Each step adds `ΔU_s := w_s*u'_s`, `ΔW_s := w_s`.
On failure: pop until `RSI_path >= band_min`:
`U -= ΔU_s; W -= ΔW_s; RSI_path := tanh(U / max(W, eps_w))`

Deterministic & shard-safe.

Branching after rollback. Compute `u'_alt`, push `(ΔU_alt, ΔW_alt)`, pick the branch with higher `RSI`.

Stamp (include rollback fields).
`...|U_path=...|W_path=...|RSI_path=...|band=A0|rollback=2|cause=band_breach|last_ok=step_3|try=alt_4A|...`

Worked numbers (continuing 5.2; 6-dec).
Before step 4: `U=1.187535`, `W=3`, `RSI=0.376388` (**A0**).
Bad step 4: `RSI'_4=-0.650000` → `u'_4=-0.775299`, `w=1` → `U=0.412236`, `W=4`, `RSI=0.102696` (`< A0` ⇒ rollback).
Pop step 4 → `U=1.187535`, `W=3`, `RSI=0.376388` (restored).
Alt `4'`: `RSI'_4'=+0.550000` → `u'_4'=0.618381` → `U=1.805916`, `W=4`, `RSI=0.423114` (**A0**, higher).

Policies (manifest).

```
"rollback": {
  "band_min": "A0",
  "delta_thr": 0.25,
  "g_min": 0.50,
  "budget": {"tokens": 2.0e6, "ms": 15000},
  "max_pops": 3,
  "on_fail": "fallback_classical"
}
```

Invariant: fallback reverts to classical logic (e.g., highest `m`); never edits `m`.

Acceptance checklist. Determinism, boundedness, auditability, fallback purity, budget adherence.

One-line takeaway. Treat each step as an additive `Δ` in u-space; on breach, pop to safety and try an alternative—deterministic containment with **`phi((m,a)) = m`** intact.

5.4 Plan-level priors and gates (optional, manifest-declared)

Plan prior (bounded). If a plan has reliability `B_plan in [-1,+1]`, apply a small u-space prior proportional to mass:
`U_plan' := U_path + beta_plan * B_plan * W_path -> RSI_path' := tanh( U_plan' / max(W_path, eps_w) )`
Keep `|beta_plan|` tiny and publish it.

Plan gate. After pooling:
`RSI_plan_env := g_plan * RSI_path'` or **`tanh(g_plan * atanh(RSI_path'))`** *(mode `"u_scale"`).*
Choose by bands; `m` unchanged.

5.5 Developer hooks (fast integration)

Per-step CSV.
`step_id, m, e_in, e_out, a_in, a_out, U_in, V_out, W_in, RSI_s, g_s, RSI_used, b_s, beta, u_step, w_step, ΔU, ΔW, stamp`

Per-path CSV.
`path_id, U_path, W_path, RSI_path, g_plan, RSI_plan_env, band_path, pops, cause, budget_tokens, budget_ms, stamp`

Minimal SDK sketch.

```
class LaneStep:
    def __init__(self, step_id, e_items, m, g=1.0, beta=0.0, b=0.0, w=1.0):
...

class LanePath:
    def add(self, step: LaneStep) -> None: ...
    def undo(self) -> None: ...
    def rsi(self) -> float: ...
    def rsi_env(self, g_plan=1.0, mode="mul") -> float: ...
```

---

6) Empirical Validation & Mini Benchmarks (stamp-replayable)

Purpose. Demonstrate that adding the lane improves selection quality, stability, and operational efficiency without changing classical values (**`phi((m,a)) = m`**). Keep experiments lightweight, public-dataset friendly, and replayable from stamped CSVs.

6.1 Task Suite & Protocol (tiny, public, reproducible)

Tasks (pick any 2–3 to start).
• **Decoding rerank (LLM).** Compare `argmax(prob)` vs `RSI_env := g_t * RSI` on next-token or short-form answers.
• **RAG QA (top-k docs + cite).** Rank candidates/doc sets by `RSI`; keep retrieval scores `m` intact; measure answer correctness and cite integrity.
• **Tool loop (agent micro-workflow).** One or two API calls + parse step; decide retry/escalate using bands on `RSI_env`.

Fixed manifest (freeze before runs).
Declare once and reuse: `eps_a`, `eps_w`, gain `c`, `weights_policy` (e.g., `w := |m|^gamma`, `gamma = 1`), `bands`, `gate_mode` (`"mul"` or `"u_scale"`), `division_policy`, `lens_id`, `lens_params`, `Unit`, `dtype` (`float64` preferred).

Stamp everything (ASCII, one-line per decision).
For each decision, log one record that includes:

```
ts, run_id, item_id, U, W, RSI, RSI_env, band, g_t,
gate_lanes(F,D,L,E,V,Q), lens_id, Unit, c, eps_a, eps_w, weights_policy,
division_policy, combine_policy, dtype, knobs_hash, file_sha256_in,
file_sha256_out
```

All symbols are ASCII; numbers are plain decimals; `phi((m,a)) = m` implied and never violated.

Evaluation windows (paired A/B on identical inputs).
• **A (baseline):** classical selector (e.g., `argmax(prob)` or existing heuristic).
• **B (SSM-AI):** selector by `RSI_env` (or advisory bands).
Ensure identical prompts, data slices, and seeds where applicable; only the selector differs.

Primary metrics (report as deltas B–A).
• **Decoding rerank:** `first-correct↑`, `hallucination-rate↓`, `retries↓`, `latency_p50/p95` neutral or improved.
• **RAG QA:** `answer_accuracy↑`, `cite_integrity↑` (all cited spans present), `off-topic↓`.
• **Tool loop:** `successful_completion↑`, `escalations↓`, `retries↓`, `bad-call-rate↓`.
Always include band histogram for `RSI_env` and count of actions gated by `A-`/`A--`.

QA invariants (must pass).
• **Collapse parity:** `phi((m,a)) = m` everywhere.
• **Boundedness:** `|a| < 1`, `|RSI| < 1`, `|RSI_env| < 1`.
• **Parity stream==batch:** identical results for shuffled vs batched within epsilon. Use `float64` where possible; if `float32`, set `eps_w >= 1e-8`.
• **Determinism:** same manifest and inputs ⇒ identical outputs (within dtype tolerance).
• **Zero-evidence guard:** if `W_in == 0`, then `RSI := 0`, `band := "A0"`, reason `insufficient_evidence`.

Minimal replay protocol (5 lines).

1. Load manifest; 2) recompute `a_in := tanh(-c*e)`, `a_out := tanh(+c*e)`; 3) fuse `U += w*atanh(a)`, `W += w`; 4) `RSI := tanh((V_out - U_in)/max(W_in, eps_w))`; 5) gate → `RSI_env` and band with hysteresis if declared.

Success criteria (greenlight to publish).
• At least one chosen task shows a statistically clear improvement on ≥2 primary metrics.
• All QA invariants hold; replay matches stamps.
• Logs are CSV-only, ASCII, and pass your verifier script without manual fixes.

One-line takeaway. Freeze a tiny manifest, run paired A/B on fixed tasks, stamp ASCII logs with `U/W/RSI/RSI_env/band`, and publish deltas — small, auditable wins that anyone can replay.

6.2 Metrics (calculator-fast; no model retraining)

Core quality & efficiency.

- **Retries ↓:** mean retries per task.
- **Time-to-first-correct ↑:** median steps to first correct output.
- **Over-confidence exposure ↑:** fraction of incorrect items landing in `A-`/`A--`.
- **Band distribution:** histogram of `RSI` or `RSI_env` (`A++`…`A--`) per task.
- **Stability (order/shard invariance):** difference ≈ 0 between batch vs stream vs shuffled `RSI` (should be within numeric epsilon).
- **Correlation:** Spearman/Pearson between `RSI` and downstream correctness.
- **OPEX proxy:** tokens/tool calls per solved task.

Acceptance flags (must hold).

- **Collapse parity:** outputs under `phi((m,a))` equal baseline `m`-only outputs.
- **Gate purity:** changing `g_t` does not change `m`.
- **Order-invariant pooling:** identical `RSI` within tolerance across permutations.

6.3 Ablations (small knobs, big clarity)

Knobs to sweep (independent).

- `gamma` in `w := |m|^gamma` → `{0, 0.5, 1, 2}`.
- Lens gain `c` → `{0.7, 1.0, 1.3}`.
- Gate mode → `{"mul", "u_scale"}`.
- Prior `beta` (u-space) → `{0, 0.1, 0.2}` with bounded `b_s in [-1,+1]`.

What to record.

- Lift in quality metrics vs A (baseline).
- Sensitivity plots: metric vs knob; mark operating point where saturation (`|a| > 0.9`) < 10% and dead-zone (`|a| < 0.1`) < 70%.

6.4 Results (tiny tables; reproduce from stamps)

(Illustrative format; fill with your stamped runs.)

Decoding rerank (short answers).

- **Dataset:** 500 prompts, vendor X, temp `0.7`, beam `5`.
- **Selector:** Baseline `argmax(prob)` vs `RSI_env` (`g_t = 1`).

Metric	Baseline	SSM-AI (RSI_env)	Delta
First-pass correctness (%)	61.8	66.4	+4.6
Over-confident errors in A+/A++ (%)	22.3	8.7	-13.6
Mean retries per prompt	0.42	0.29	-31%

RAG QA (top-5 docs)

Selector. baseline score vs `RSI` pooling doc alignments (support − penalties).

- **EM / F1 (answer):** Baseline `48.2 / 63.1`; SSM-AI `50.7 / 65.0`; **Delta** `+2.5 / +1.9`
- **Cite integrity (valid cites %):** Baseline `71.0`; SSM-AI `79.6`; **Delta** `+8.6`
- **Tokens per solved task:** Baseline `8.9k`; SSM-AI `7.6k`; **Delta** `-15%`

Tool loop (one parse + one call).

Metric	Baseline	SSM-AI (A+/A0/A-) policy	Delta
Bad escalations per 1000	14.1	8.9	-37%
Time-to-first-correct (s)	12.4	10.2	-18%

All numbers must be reproducible by re-running the stamp files with the fixed manifest.

6.5 Repro Steps (one page, copy-paste)

Inputs. `manifest.json`, `decisions.csv` (stamped), task gold labels.

Compute per decision.

1. **Clamp & map:** `u_in := atanh(clamp(a_in))`, `u_out := atanh(clamp(a_out))`.
2. **Chooser:** `RSI := tanh( (sum w*u_out - sum w*u_in) / max(sum w, eps_w) )`.
3. **Gate:** `RSI_env := g_t * RSI` or `tanh( g_t * atanh(RSI) )`.
4. **Band:** apply `A++/A+/A0/A-/A--` thresholds.
5. **Record:** `(m, RSI, RSI_env, band, U, W, stamp)` unchanged `m` via `phi((m,a)) = m`.

Aggregate.

- **Quality:** accuracy/F1 vs gold.
- **Efficiency:** retries, tokens, tool calls.
- **Stability:** difference between batch vs stream vs shuffled `RSI` (expect ~0).
- **Correlation:** `corr(RSI, correctness)`.

Report.

- Tables as in **6.4**, plus a compact plot of band histogram *(optional)*.
- Append the manifest hash and stamp digest to the report header.

One-line takeaway. With nothing but lenses → `RSI` → optional gate, teams can show measurable lifts and cost reductions today, while `m` remains identical (**`phi((m,a)) = m`**).

---

7) Scalability & Numerical Precision (long paths, dtype, overflow)

Purpose. Ensure the lane remains bounded, deterministic, and fast for long paths (100+ steps), large shards, and mixed hardware — without ever changing classical values (**`phi((m,a)) = m`**).

7.1 Long-Path Guidance (agents, streams, shards)

- **Carry only `(U, W)`; never just `a_out`.**
  Streaming fuse is **`U += w*atanh(a); W += w; a_out := tanh( U / max(W, eps_w) )`**.
  For merging shards: **sum `U` and `W` from each shard, then invert once**.
- **Checkpoint/rollback with additive deltas.**
  Store per-step **`(ΔU := w*u, ΔW := w)`** to undo exactly (see 5.3).
- **Chunking doesn't matter.**
  Order/shard invariance holds because composition is additive in **`u := atanh(a)`**.
- **Precision tip.**
  For very long runs, use **pairwise or Kahan-style summation** on `U` (optional) to reduce float error; `W` can use standard summation.

7.2 Dtype & Epsilon (recommended defaults)

Clamp margin for alignment `eps_a`.
• **float32:** `eps_a = 1e-6`
• **float64:** `eps_a = 1e-12`

Denominator guard for means `eps_w`.
• **float32:** **`eps_w >= 1e-8`** (do not use `1e-12` in `float32`; start at `1e-8`)
• **float64:** `eps_w = 1e-12`

Gate epsilon `eps_g` (all dtypes).
• `eps_g = 1e-12`

Safe `atanh` input (always clamp).
• `a_c := clamp(a, -1+eps_a, +1-eps_a)` before `atanh(a_c)`
• Keep `|a| < 1 - eps_a` in all lanes and choosers.

When to prefer `float64`.
• Paths with `> 10^3` steps or wide dynamic `w`
• Cross-vendor bake-offs where bit-tight reproducibility matters
• CPU batch analytics (offline replay) where throughput is ample

Implementation notes.
• **Zero-evidence guard:** if `W_in == 0`, set `RSI := 0`, `band := "A0"`, reason `insufficient_evidence`.
• **Streaming fuse recall:** `U += w*atanh(a) ; W += w ; a_out := tanh( U / max(W, eps_w) )`.
• **Collapse parity:** `phi((m,a)) = m` under all dtype settings.

7.3 Stability Near Edges (`a -> ±1`)

- **Never feed raw ±1.** Use **`a_c := clamp(a, -1+eps_a, +1-eps_a)`**.
- **Curvature awareness.**
  `atanh(a)` grows rapidly near `±1`. Keep lenses in the responsive band: typical **`|c*e|`** in **`[0.3, 1.2]`**. See 3.5.
- **Lane mul/div policy (M2).**
  `a_mul := tanh(atanh(a1) + atanh(a2))`
  `a_div := tanh(atanh(a1) - atanh(a2))`
  Division near zero follows your declared **`division_policy`** *(default `"strict"`)*.
  Lane purity: policies act on `a`/routing, never on `m`.
- **Gating at high confidence.**
  If you want gentler damping when `RSI` is large, use mode **`"u_scale"`**:
  `RSI_env := tanh( g_t * atanh(RSI) )`.
  This preserves curvature better than plain multiply.

7.4 Software–Hardware Parity (fixed-point notes)

- **Identical semantics across targets.**
  The sequence **`clamp → atanh → add in u → divide → tanh`** must match bit-for-bit *(within dtype tolerance)*. This guarantees **`phi((m,a)) = m`** everywhere and **batch == stream == shuffled**.
- **Range planning (summary; details in Appendix G).**
  • Keep internal `u` in a symmetric fixed-point range *(e.g., `[-Umax, Umax]`)* chosen so **`tanh(Umax) ≈ 0.999`**.
  • Quantize clamps to keep **`|a| < 1`**; propagate `eps_a`/`eps_w` as constants.
  • Use **saturating adds** for `U` if hardware requires, then invert once with **`tanh`**.
- **Golden vectors.**
  Ship a small set of inputs with expected outputs for **float32/float64** and your **fixed-point flavor**; run them in CI (see Appendix D/J).

7.5 Performance Considerations (big-O and memory)

- **Per item:** `O(1)` math *(clamp, atanh, add, optional band)*.
- **Per stream:** `O(N)` time, `O(1)` memory *(store only `U`, `W`)*.
- **Vectorization:** Apply **`clamp`**, **`atanh`**, **`tanh`** elementwise; reduce via weighted sum.
- **Throughput knobs:**
  • Use **lookup/tables** or **fast approximations** for **`tanh/atanh`** on accelerators, validated against golden vectors.
  • Batch **`atanh`** calls where possible; cache small **`|a|`** regimes if profiling shows wins.

7.6 Robustness & Acceptance (quick checks)

- **Collapse parity:** **`phi((m,a)) = m`** before/after any lane/gate operation.
- **Order/shard invariance:** same `RSI` (within tolerance) for **batch vs stream vs shuffled**.
- **Boundedness:** all `a`, `RSI`, `RSI_env`, `RSI_path` satisfy **`|x| < 1`**.
- **Edge clamps:** inputs at or beyond `±1` get clamped to **`±(1 - eps_a)`**.
- **Division policy:** under **`"strict"`**, if magnitude divisor violates bounds, actuation falls back to classical policy; lane still stamps context.
- **Determinism:** fixed manifest ⇒ identical outputs *(within dtype)*, across machines.

7.7 Troubleshooting (symptoms → fixes)

- **Symptom: frequent `a` near `±1` (saturation).**
  Fix: reduce lens gain `c` or increase `Unit`; verify 3.5 calibration.
- **Symptom: `RSI` hovers near `0` (dead-zone).**
  Fix: increase `c` *(or adjust `Unit`)*; consider uniform weights **`w := 1`**.
- **Symptom: slight drift between batch and stream.**
  Fix: clamp before **`atanh`**; use **pairwise/Kahan** sum for `U`; check **`eps_w`**.
- **Symptom: unstable ratios.**
  Fix: ensure **`division_policy = "strict"`**; verify denominators' declared bounds; lane math stays **M2**.
- **Symptom: gating feels too aggressive at high confidence.**
  Fix: switch gate mode to **`"u_scale"`**: **`RSI_env := tanh( g_t * atanh(RSI) )`**.

One-line takeaway. Scale confidently: **sum in `u`**, carry **`(U,W)`**, **clamp before `atanh`**, and keep epsilons dtype-appropriate — so even very long, sharded paths remain bounded, reproducible, and fast, with **`phi((m,a)) = m`** always preserved.

---

8) Integration Quickstarts (drop-in wrappers)

Purpose. Ship fast without touching classical values. Each quickstart shows where to compute `RSI`, how to apply the calm gate `RSI_env := g_t * RSI` *(or `tanh(g_t * atanh(RSI))`)*, and how to keep `phi((m,a)) = m` sacrosanct.

8.1 LLM Decoding Hooks (beam/greedy, HF-style callbacks)

Where to hook. After you have candidate tokens/logits and any side signals for the lens.

Minimal flow.

1. **Compute contrasts** `e_in, e_out`.
2. **Map to alignments:** `a_in := tanh(-c * e_in)`, `a_out := tanh(+c * e_out)`.
3. **Chooser:** `RSI := tanh( (sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w) )`.
4. **Gate:** `RSI_env := g_t * RSI` *(or mode `"u_scale"` → `RSI_env := tanh( g_t * atanh(RSI) )`)*.
5. **Pick by `RSI_env`; emit `m` unchanged** via `phi((m,a)) = m`.

Clamp rule (always). Before any `atanh`, clamp: `a_c := clamp(a, -1+eps_a, +1-eps_a)`.

Weights policy. Default `w := |m|^gamma` with `gamma = 1`; `w := 1` if declared.

Pseudocode (beam pick, callback-style).

```
def on_candidates(cands, lens, g_t=1.0, eps_w=1e-12, eps_a=1e-6,
gate_mode="mul"):
    scored = []
    for cand in cands:  # cand: has m (logprob/prob) and lens_items:
[(e_in, e_out, w), ...]
        U_in = V_out = W = 0.0
        for (e_in, e_out, w) in cand.lens_items:
            a_in  = tanh(-lens.c * e_in)
            a_out = tanh(+lens.c * e_out)
            a_in  = max(-1+eps_a, min(1-eps_a, a_in))
            a_out = max(-1+eps_a, min(1-eps_a, a_out))
            U_in  += w * atanh(a_in)
            V_out += w * atanh(a_out)
            W     += w

        if W <= 0:
            RSI = 0.0
            band = "A0"  # insufficient_evidence
        else:
            RSI = tanh((V_out - U_in) / max(W, eps_w))
```

```
            band = to_band(RSI)  # A++/A+/A0/A-/A--

        RSI_env = (tanh(g_t * atanh(RSI)) if gate_mode == "u_scale" else
g_t * RSI)
        RSI_env = max(-1+eps_a, min(1-eps_a, RSI_env))

        scored.append((cand, RSI_env, RSI, band, U_in, V_out, W, g_t))

    best = max(scored, key=lambda kv: kv[1])[0]
    return best  # selection by RSI_env; classical m remains intact
(phi((m,a)) = m)
```

Stamp fields (suggested).
`token_id, m, RSI, RSI_env, band, U_in, V_out, W, g_t, lens_id, Unit, c, eps_a, eps_w, weights_policy, combine_policy, gate_mode, dtype, knobs_hash, stamp`

8.2 RAG Pipeline Hook (retrieval → generation)

Where to hook. After retrieval scores but before generation. Keep retrieval `m` intact; compute a bounded doc alignment and optionally forward a pooled lane into generation.

Doc lens example.
```
e_out := semantic_gain + citation_hit + source_authority
e_in := toxicity_gap + policy_risk + staleness_penalty
a_out := tanh(+c*e_out),a_in := tanh(-c*e_in)
```

Rank docs by RSI.

```
def rsi_doc(signals, c=1.0, eps_w=1e-12):  # signals = [(tox_gap, cit_hit,
sem_gain, w), ...]
    U_in = V_out = W = 0.0
    for tox, cit, sem, w in signals:
        U_in  += w * atanh(tanh(-c*tox))
        V_out += w * atanh(tanh(+c*(cit + sem)))
        W     += w
    return 0.0 if W <= 0 else tanh((V_out - U_in)/max(W, eps_w))
```

Forwarding into generation (optional).
Pool top-K doc lanes:
`a_pool := tanh( (sum w*atanh(a_doc)) / max(sum w, eps_w) )`
and pass `a_pool` as a side feature; do not alter generation `m`.

Stamp fields. `doc_id, m_retrieval, RSI_doc, band, contribs(U,V,W), stamp.`

8.3 Agents/Tools Middleware (steps, branching, rollback)

Per-step score (Section 5.1 recap).
• Compute `RSI_s` from lens items.
• Gate: `RSI_used := g_s * RSI_s` *(or `tanh(g_s * atanh(RSI_s))`)*.
• Convert to u-space: `u_s := atanh(clamp(RSI_used))`.
• Add tiny prior in u-space: `u'_s := u_s + beta*b_s`.
• Keep `(ΔU := w_s*u'_s, ΔW := w_s)` for rollback.

Path score (Section 5.2).
• Accumulate: `U_path += ΔU`, `W_path += ΔW`, then `RSI_path := tanh(U_path / max(W_path, eps_w))`.

Branch policy (A+/A0/A-).

```
def should_retry(RSI_env, band_fn=to_band):
    return band_fn(RSI_env) in {"A++", "A+"}

def try_step(path, step_RSI_env, beta=0.0, b_s=0.0, w=1.0, eps_a=1e-6,
eps_w=1e-12):
    a = max(-1+eps_a, min(1-eps_a, step_RSI_env))
    u = atanh(a) + beta*b_s
    path.U += w*u; path.W += w; path.stack.append((w*u, w))
    return tanh(path.U / max(path.W, eps_w))  # RSI_path

def rollback(path, until_band="A0", eps_w=1e-12):
    thr = {"A++":0.90, "A+":0.60, "A0":-0.60, "A-":-0.90, "A--":-
1.00}[until_band]
    def rsi(): return 0.0 if path.W <= 0 else tanh(path.U/max(path.W,
eps_w))
    pops = 0
    while path.stack and rsi() < thr:
        dU, dW = path.stack.pop()
        path.U -= dU; path.W -= dW; pops += 1
    return pops
```

Stamp fields (per step). `step_id, m, RSI_s, g_s, RSI_env, u_step, w_step, ΔU, ΔW, RSI_path, band, stamp`.
Invariant. Never modify `m`; `phi((m,a)) = m` throughout.

8.4 CI/Golden Tests (one-command acceptance)

What to test automatically.
• **Collapse parity:** `phi((m,a)) = m` across the whole pipeline.
• **Order/shard invariance:** batch vs stream vs shuffled produce identical `RSI/RSI_path` (within tolerance).
• **Clamp discipline:** all inputs to `atanh` respect `|a| < 1` via `a_c := clamp(a, -1+eps_a, +1-eps_a)`.
• **Band boundaries:** exact hits at `0.90, 0.60, -0.60, -0.90` map to the right labels.
• **Gate purity:** toggling `g_t` only scales alignment (`RSI_env`), never `m`.

• **Division policy:** under `"strict"`, near-zero denominators trigger fallback to classical selection (no lane-based actuation).

Skeleton test harness (CLI gist).

```
# 1) Load manifest.json and stamped decisions.csv
# 2) Recompute RSI/RSI_env and bands; assert equality with recorded fields
# 3) Shuffle inputs (and shard); assert RSI equality within tol
# 4) Toggle gate g_t -> assert m unchanged; only RSI_env changes
# 5) Emit PASS/FAIL summary + manifest hash
```

Artifacts to keep in repo.
• Golden vectors (tiny CSV) for each surface (decoding, RAG, tools).
• Manifest template with `eps_a, eps_w, c, gamma, bands, division_policy, gate.mode`.
• Stamp verifier (100-line script) that replays `RSI := tanh((V_out - U_in)/max(W_in, eps_w))` and checks bands.

One-line takeaway. These hooks let you add a bounded chooser to decoding, RAG, and agents in a day: compute `RSI`, apply a simple calm gate, pick by bands, stamp for replay — all while your classical numbers remain untouched via `phi((m,a)) = m`.

Appendix A — Gate Presets & Acceptance (copy-paste)

Purpose. Give teams a drop-in way to turn on the calm gate with proven defaults and calculator-fast checks. All math is alignment-only: `RSI_env := g_t * RSI` or `RSI_env := tanh( g_t * atanh(RSI) )`. Classical numbers remain untouched: `phi((m,a)) = m`.

A1) Presets (manifest snippets)

(Paste one block verbatim into your manifest. Nonnegative lane weights; lanes are normalized to [0,1]. Defaults: `eps_g = 1e-12`.)

Preset A — Steady (production default)

```
"gate": {
  "mode": "mul",
  "rho": 0.20,
  "g_min": 0.00,
  "eps_g": 1e-12,
  "weights": {"F":1.0,"D":1.0,"L":1.0,"E":1.0,"V":1.0,"Q":0.0},
  "safety_notch": {"enabled": false}
}
```

Preset B — Safety-first (turbulence hardening)

```
"gate": {
  "mode": "mul",
  "rho": 0.30,
  "g_min": 0.20,
  "eps_g": 1e-12,
  "weights": {"F":1.5,"D":1.0,"L":1.0,"E":1.5,"V":0.5,"Q":0.5},
  "safety_notch": {"enabled": true, "s_thr": 0.80}
}
```

Preset C — Incident mode (contain violations, preserve signal top-end)

```
"gate": {
  "mode": "u_scale",
  "rho": 0.50,
  "g_min": 0.10,
  "eps_g": 1e-12,
  "weights": {"F":1.5,"D":1.0,"L":1.5,"E":1.5,"V":0.5,"Q":0.5},
  "safety_notch": {"enabled": true, "s_thr": 0.70}
}
```

A2) How to compute `g_t` (recap)

```
W := wF+wD+wL+wE+wV+wQ
mix := (wF*F_t + wD*D_t + wL*L_t + wE*E_t + wV*V_t + wQ*Q_t) / max(W,
eps_g)
g_inst := clamp( 1 - mix , 0 , 1 )
g_t := clamp( max(g_min, (1 - rho)*g_{t-1} + rho*g_inst ), 0, 1 )
```

Cold-start convention for minis. For vectors below, use `g_{t-1} := g_inst` at `t=0`. This yields `g_t = g_inst` on the first tick and keeps the math calculator-fast.

A3) Ready-to-run acceptance vectors (calculator-fast)

(All lanes already scaled to [0,1]. Bands use A++/A+/A0/A-/A-- with A+ ≥ +0.60 and A0 in (−0.60,+0.60).)

Vector V1 (matches calm-but-nonzero load).
Inputs: `RSI = 0.70`, lanes `F_t=0.20, D_t=0.10, L_t=0.30, E_t=0.15, V_t=0.20, Q_t=0.00`.

- **Preset A (Steady):** `W=5.0`, `mix=(0.20+0.10+0.30+0.15+0.20)/5=0.19`, so `g_t=1-0.19=0.81`.
  `"mul"` → `RSI_env = 0.81 * 0.70 = 0.56700` → **band A0**.
- **Preset C (Incident mode):** same `g_t=0.81`.
  `"u_scale"` → `RSI_env = tanh( 0.81 * atanh(0.70) ) ≈ tanh(0.70251) ≈ 0.605998` → **band A+**.

Vector V2 (violation on F; notch active).
Inputs: `RSI = 0.62`, lanes `F_t=0.85, D_t=0.10, L_t=0.10, E_t=0.05, V_t=0.10, Q_t=0.00`.

- **Preset B (Safety-first):** weights `{1.5,1.0,1.0,1.5,0.5,0.5}` → `W=6.0`.
  `mix = (1.5*0.85 + 1.0*0.10 + 1.0*0.10 + 1.5*0.05 + 0.5*0.10 + 0.5*0.00)/6`
  `mix = (1.275 + 0.10 + 0.10 + 0.075 + 0.05 + 0.00)/6 = 1.60/6 ≈ 0.266667` → `g_inst ≈ 0.733333`.
  Notch: `sev := max(F_t,D_t,E_t) = 0.85`, `s_thr=0.80`, so
  `g_sev := clamp( 1 - (0.85 - 0.80)/max(1 - 0.80, 1e-12) , 0 , 1 ) = 0.75`.
  `g_inst := min(0.733333, 0.75) = 0.733333`.
  Cold-start gives `g_{t-1} := g_inst`, hence `g_t = g_inst = 0.733333`.
  `"mul"` → `RSI_env = 0.733333 * 0.62 ≈ 0.454667` → **band A0**.

  *(If you instead assume prior tick `g_{t-1}=1`, then `g_t = (1-rho)*1 + rho*0.733333 = 0.70 + 0.219999 ≈ 0.920000`, still **A0** after multiplication: `0.920000*0.62 ≈ 0.570400`.)*

Vector V3 (calm conditions).
Inputs: `RSI = 0.55`, lanes all `0.05`, **Preset A**:
`mix = (0.05*5)/5 = 0.05` → `g_t = 0.95` → `"mul"` → `RSI_env = 0.95 * 0.55 = 0.522500` → **band A0**.

A4) Band policy (recommended defaults)

- `promote_if: "RSI_env >= 0.60"`
- `pause_if: "RSI_env in (-0.60, +0.60)"`
- `block_if: "RSI_env <= -0.60"`
  Apply hysteresis if desired: `h_up = 0.02`, `h_dn = 0.02` (promotion/demotion gates).

A5) Stamp fields (append to your one-liner)

Append these key-values to your stamp tail:
```
"|g=" + fmt(g_t) + "|RSI_env=" + fmt(RSI_env) + "|gate_mode=" + mode +
"|lanes=" + fmt(F_t,D_t,L_t,E_t,V_t,Q_t)"
```

A6) Acceptance checklist (pass/fail)

- **Purity.** `phi((m,a)) = m` before/after gating.
- **Boundedness.** `|RSI_env| < 1` for all vectors.
- **Monotonicity.** If any lane increases (worsens), `g_t` must not increase.
- **Determinism.** Same manifest + inputs ⇒ same `g_t`, `RSI_env` (within dtype eps).
- **Bands.** `to_band(RSI_env)` matches thresholds (honor hysteresis if declared).

A7) One-minute turn-on

1. Paste one preset into your manifest.
2. Run V1–V3 locally; match outputs within dtype tolerance (6-dec print OK).
3. Enable gating on one surface (tools or decoding) and log `RSI, g_t, RSI_env, band` beside `m`.
4. Roll up bands daily using the standard `U/W` fuse on the lane; `phi((m,a)) = m` holds throughout.

Appendix B — Symbolic Search Lens (SSM-Search)

Purpose. Provide a single, published lens for ranking internet/intranet/local search results with bounded, comparable scores. Classical retrieval numbers remain intact: `phi((m,a)) = m`. The lens turns observable features into contrasts `e`, maps to alignments, and selects by a bounded chooser `RSI in (-1,+1)` with optional gating `RSI_env := g_t * RSI` or `RSI_env := tanh(g_t*atanh(RSI))`.

B1) Feature normalization (to [0,1])

Declare and normalize once (no PII). Examples:

- **hit_quality:** `hit_quality := clamp((score - p10) / max(p90 - p10, eps), 0, 1)`
  (use per-engine score quantiles; or a small logistic if preferred)
- **freshness:** `freshness := exp(-lambda * age_days)` with `lambda > 0`
  (choose `lambda` so 7–30 days map to ~0.3–0.6)
- **semantic_match:** `semantic_match := clamp((cosine + 1) / 2, 0, 1)` for cosine in `[-1,1]`
- **risk_penalty:** `risk_penalty := clamp(w_tox*tox + w_pii*pii + w_outl*outlier + ..., 0, 1)`

Keep names, weights, and `lambda` in the manifest.

B2) Lens (declare once; dimensionless)

```
e := (alpha*hit_quality + beta*freshness + gamma*semantic_match -
delta*risk_penalty) / Unit
```
with `alpha, beta, gamma, delta > 0`, `Unit > 0`.

Split into channels (recommended):

```
e_out := alpha*hit_quality + beta*freshness + gamma*semantic_match
e_in := delta*risk_penalty
```

B3) Map -> align -> choose (bounded, order-invariant)

```
a_in := tanh(-c * e_in)
a_out := tanh(+c * e_out)
U_in := sum w * atanh(a_in)
V_out := sum w * atanh(a_out)
W_in := sum w
RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )
```

`RSI_env := g_t * RSI` *(or `RSI_env := tanh(g_t*atanh(RSI))` per manifest)*

Defaults: `c = 1.0`, `eps_w = 1e-12`, weights `w := 1` *(or `w := |m|^gamma` if declared)*.
Invariant: `phi((m,a)) = m` throughout.

B4) Federated/shard-proof pooling (meta-search)

Pool within each source/engine, then merge once:

For each engine `s`:
`U_in^s := sum atanh(a_in)` ; `V_out^s := sum atanh(a_out)` ; `W_in^s := sum w`

Merge across engines:
`U_in := sum_s U_in^s` ; `V_out := sum_s V_out^s` ; `W_in := sum_s W_in^s` ;
`RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`

This guarantees order-invariance across shards and sources.

B5) Manifest (copy-paste block)

```
"ssm_search": {
  "features": {
    "normalize": {
      "hit_quality": "quantile_minmax(p10,p90)",
      "freshness": "exp_decay(lambda)",
      "semantic_match": "cosine_to_unit",
      "risk_penalty": {"tox": 1.0, "pii": 1.0, "outlier": 0.5}
    },
    "params": {"lambda": 0.05}
  },
  "lens": {
    "alpha": 1.0, "beta": 0.5, "gamma": 0.7, "delta": 0.8,
    "Unit": 1.0, "c": 1.0
  },
  "weights": {"policy": "uniform"},
  "gate_ref": "gate_preset_A"
```

```
}
```

B6) Pseudocode (drop-in)

```
def ssm_search_score(hit_quality, freshness, semantic_match, risk_penalty,
                     alpha=1.0, beta=0.5, gamma=0.7, delta=0.8,
                     Unit=1.0, c=1.0, eps_w=1e-12, g_t=1.0, w=1.0):
    e_out = (alpha*hit_quality + beta*freshness + gamma*semantic_match) /
Unit
    e_in  = (delta*risk_penalty) / Unit
    a_out = tanh(+c * e_out)
    a_in  = tanh(-c * e_in)
    V_out = w * atanh(a_out)
    U_in  = w * atanh(a_in)
    RSI   = tanh( (V_out - U_in) / max(w, eps_w) )
    RSI_env = g_t * RSI  # or tanh(g_t*atanh(RSI))
    return RSI, RSI_env

def rank_results(results, gate):
    # results: list of dicts with normalized features (and optional 'w')
    scored = []
    for r in results:
        w = r.get("w", 1.0)
        RSI, RSI_env = ssm_search_score(
            r["hit_quality"], r["freshness"], r["semantic_match"],
r["risk_penalty"],
            g_t=gate.get("g", 1.0), w=w
        )
        r["RSI"], r["RSI_env"] = RSI, RSI_env
        r["band"] = to_band(RSI_env)
        scored.append(r)
    return sorted(scored, key=lambda x: x["RSI_env"], reverse=True)

def merge_shards(shards):
    # shards: list of {'U_in':..., 'V_out':..., 'W_in':...} from engines
    U_in  = sum(s["U_in"]  for s in shards)
    V_out = sum(s["V_out"] for s in shards)
    W_in  = sum(s["W_in"]  for s in shards)
    RSI = tanh( (V_out - U_in) / max(W_in, 1e-12) )
    return RSI
```

B7) Worked example (calculator-fast; c=1, Unit=1, w=1)

Parameters: `alpha=1.0, beta=0.5, gamma=0.7, delta=0.8.`

- **Result A:** `hit_quality=0.9, freshness=0.6, semantic_match=0.7, risk_penalty=0.2`
  `e_out = 0.9 + 0.3 + 0.49 = 1.69` ; `e_in = 0.8*0.2 = 0.16`
  `Net = 1.53` ⇒ `RSI = tanh(1.53) ≈ 0.910425` → **A++**.
- **Result B:** `0.8, 0.2, 0.5, 0.5`
  `e_out = 0.8 + 0.1 + 0.35 = 1.25` ; `e_in = 0.8*0.5 = 0.40`
  `Net = 0.85` ⇒ `RSI = tanh(0.85) ≈ 0.691069` → **A+**.

- **Result C:** `0.4, 0.2, 0.4, 0.6`
  `e_out = 0.4 + 0.1 + 0.28 = 0.78` ; `e_in = 0.8*0.6 = 0.48`
  `Net = 0.30` ⇒ `RSI = tanh(0.30) ≈ 0.291313` → **A0**.

Ranking (no gate): A `(0.910425, A++)` > B `(0.691069, A+)` > C `(0.291313, A0)`
With gate `g_t = 0.80` (turbulence):
`RSI_env(A)=0.728340 (A+)`, `RSI_env(B)=0.552855 (A0)`, `RSI_env(C)=0.233050 (A0)`
→ same order, conservative bands.

B8) UI policies (example)

- **Open directly** if `RSI_env >= 0.90` (**A++**).
- **Preview with caution** if `0.60 <= RSI_env < 0.90` (**A+**).
- **Require extra click or summarize** if `-0.60 < RSI_env < 0.60` (**A0**).
- **Quarantine** if `RSI_env <= -0.60` (**A- / A--**).
  All while `m` remains untouched and logged: `phi((m,a)) = m`.

B9) Acceptance vectors (pass/fail)

- **V1:** the three results above reproduce the stated `RSI` values within dtype tolerance.
- **V2 (order/shard):** evaluating A,B,C in any order or per-shard and merging `U := sum atanh(a)`, `W := 1` yields identical `RSI`.
- **V3 (gate):** with `g_t in {1.0, 0.8, 0.5}`, `RSI_env := g_t * RSI` stays in `(-1,+1)` and bands update per thresholds.
- **V4 (saturation guard):** if any feature pushes `|c*e| > 3`, the output `a := tanh(c*e)` remains bounded; results remain deterministic.
- **V5 (collapse parity):** for any downstream classical metric `m`, `phi((m,a)) = m` holds.

B10) Stamp fields (add to your one-liner)

```
"|SSMSEARCH|alpha=1.0|beta=0.5|gamma=0.7|delta=0.8|Unit=1.0|c=1.0|RSI=0.910
4|band=A++|g=0.80|RSI_env=0.7283|"
```

One-line takeaway.
A single, published lens `e := (alpha*quality + beta*freshness + gamma*semantic - delta*risk)/Unit`, mapped by `tanh`, yields a bounded, comparable `RSI` for search—drop-in, order-invariant, shard-proof, and audit-ready, with `phi((m,a)) = m` always.

Appendix C — Stamp & Ledger Schema (replay, roll-up, CLI)

Purpose. Give teams a tiny, publishable way to stamp each decision and keep a ledger that replays bit-for-bit and rolls up by hour/day/week using the same invariants: `phi((m,a)) = m`, `U += w*atanh(a)`, `W += w`, `a_out := tanh(U / max(W, eps_w))`. Decisions remain observation-only; classical numbers stay identical.

C1) One-line stamp (ASCII, copy-paste)
Append to each decision log line (fields are examples—extend as needed):

```
SSMCLOCK1|iso_utc|rasi_idx|theta_deg|sha256(file)|chain|svc=decode|req=abc123|step=beam|RSI=0.604367|g=0.81|RSI_env=0.489537|band=A0|gate_mode=mul|manifest=knobs_hash
```

- **Chooser after gate:** `RSI_env := g_t * RSI` or `RSI_env := tanh(g_t * atanh(RSI))` per manifest.
- **Continuity anchors:** `theta_deg` and `rasi_idx`.
- **Repro lock:** `manifest=knobs_hash` freezes clamps, weights, bands, gate mode, lens params.

C2) Ledger CSV (minimal schemas)

Decision CSV (row = one decision/candidate/step)
Required columns:

- `iso_utc` — timestamp (UTC)
- `svc` — surface (`decode`, `rag`, `tool`, etc.)
- `req_id` — request/session id
- `item_id` — candidate/doc/tool step id
- `RSI` — bounded chooser in `(-1,+1)`
- `w` — weight used for this decision (default `1` if uniform)
- `g` — gate value in `[0,1]`
- `RSI_env` — gated chooser (for routing/bands)
- `band` — band of `RSI_env` per thresholds
- `U_dec` — `atanh(RSI)` (store to avoid recompute)
- `W_dec` — `w`
- `manifest` — `knobs_hash` (bands, clamps, weights, gate, lens)
- `stamp` — the one-liner above

Optional columns:

- `m` (classical magnitude) and `a` (lane) if emitted
- `gate_mode` (`mul` or `u_scale`), `eps_a`, `eps_w`
- `path_id` (if this row contributes to a tracked path)

Path CSV (row = one path roll-up entry)

- `path_id` — stable identifier of the path/flow
- `U_path` — sum of contributing `U_dec`
- `W_path` — sum of contributing `W_dec`
- `RSI_path` — `tanh( U_path / max(W_path, eps_w) )`
- `g_plan` — planned gate value for the path (if any)
- `RSI_env` — gated path chooser
- `band` — band for the path
- `pops` — count of decisions folded
- `cause` — short text or code for dominant lane/cause (optional)
- `stamp` — single-line stamp for the path snapshot

C3) Replay & roll-up formulas (order/shard invariant)

- **Decision replay (sanity):**
  `RSI_replay := tanh( U_dec / max(W_dec, eps_w) )` *(should equal stored `RSI`)*
- **Time/window roll-up (uniform or provided weights):**
  `U_win := sum U_dec` ; `W_win := sum W_dec` ; `RSI_win := tanh( U_win / max(W_win, eps_w) )`
- **Shard merge:** if worker `k` returns `(U_k, W_k)`, then global
  `RSI_global := tanh( (sum_k U_k) / max(sum_k W_k, eps_w) )`
  equals the single-pass result (**batch == stream == shuffled**).
- **Band after gate:**
  `RSI_env := g * RSI` *(or `RSI_env := tanh( g * atanh(RSI) )`)* → `band := to_band(RSI_env)`
- **Never average `a` directly.** Always pool in `U/W`, then invert once.

C4) Worked vectors (calculator-fast)

- **V1 — Single decision replay (decode example).**
  Stored: `RSI = tanh(0.7) ≈ 0.604367`, so `U_dec = 0.7`, `W_dec = 1`.
  Replay: `tanh(0.7/1) = 0.604367` *(matches)*.
- **V2 — Window roll-up (3 decisions, uniform).**
  Decisions: `RSI = 0.910425, 0.691069, 0.291313`.
  Their `U_dec = 1.53, 0.85, 0.30`; `W_dec = 1` each.
  `U_win = 2.68` ; `W_win = 3` ; `RSI_win = tanh(2.68/3) = tanh(0.893333) ≈ 0.713036` → **A+**.
- **V3 — Weighted roll-up (2 decisions, w = 2 and 1).**
  `RSI = 0.604367 (U=0.7)` with `w=2` and `RSI = 0.291313 (U=0.3)` with `w=1`.
  `U_win = 2*0.7 + 1*0.3 = 1.7` ; `W_win = 3` ; `RSI_win = tanh(1.7/3) ≈ 0.512907` → **A0**.
- **V4 — Shard merge equals single pass.**
  Shard A: `U_A = 0.7`, `W_A = 1` ; Shard B: `U_B = 0.3`, `W_B = 1`.
  Global: `tanh( (0.7 + 0.3) / (1 + 1) ) = tanh(0.5) = 0.462117` *(same as single pass)*.

C5) CLI sketch (5 commands, zero servers)

```
# 1) Validate rows (replay RSI)
rsi-ledger validate --in decisions.csv
# checks: tanh(U_dec / max(W_dec, eps_w)) == RSI (within tolerance)

# 2) Roll up by hour/day/week (uniform or provided weights)
rsi-ledger rollup --in decisions.csv --by hour --out rollup_hour.csv
# outputs U_win, W_win, RSI_win, band per bucket

# 3) Merge shards (map-reduce)
rsi-ledger merge --inputs shard_*.csv --out merged.csv
# simply sums U_dec/W_dec for identical keys (e.g., hour buckets)

# 4) Band histogram
rsi-ledger bands --in decisions.csv --by day
# prints counts/fractions of A++/A+/A0/A-/A--

# 5) Diff two manifests (knobs drift guard)
rsi-ledger diff-manifest --a manifest_old.json --b manifest_new.json
# fails build if knobs_hash changed without approval
```

Implementation note. All CLI math uses the same kernel: `U += w*atanh(x)`, `out := tanh(U / max(W, eps_w))`. No special cases.

C6) Privacy & scope (ledger)

- **Aggregate-only logging;** avoid raw PII features.
- Keep `m` separate for classical analytics; **collapse parity:** `phi((m,a)) = m`.
- If telemetry for gate is missing, set `g := 1` and flag that row.

C7) Acceptance checklist (pass/fail)

- **Replay:** for each row, `tanh(U_dec / max(W_dec, eps_w)) == RSI` within dtype tolerance.
- **Order/shard invariance:** permutations and shard merges produce the same `RSI_win`.
- **Bounds:** all `RSI` and `RSI_env` satisfy `|x| < 1`.
- **Band mapping (canonical):**
    - `RSI_env >= +0.90` → **A++**
    - `+0.60 <= RSI_env < +0.90` → **A+**
    - `-0.60 < RSI_env < +0.60` → **A0**
    - `-0.90 < RSI_env <= -0.60` → **A-**
    - `RSI_env <= -0.90` → **A--**
- **Determinism:** identical `knobs_hash` and inputs ⇒ identical outputs.
- **Stamps parse:** every `stamp` line parses; recorded values re-compute within dtype eps.

One-line takeaway. Stamp each decision and log `(U_dec, W_dec, RSI, RSI_env, band)`; roll-ups are `tanh( sum U / max(sum W, eps_w) )`—deterministic, order-proof, shard-proof, and vendor-fair, with `phi((m,a)) = m` always.

Appendix D — Starter SDK & Golden Vectors (drop-in, observation-only)

Purpose. Provide a tiny, copy-pasteable SDK surface so teams can implement SSM-AI with identical semantics across stacks. All math is alignment-only; classical numbers remain untouched: `phi((m,a)) = m`. Core identities: `a_c := clamp(a, -1+eps_a, +1-eps_a)`, `u := atanh(a_c)`, `a := tanh(u)`, streaming fuse `U += w*atanh(a)`, `W += w`, `a_out := tanh(U / max(W, eps_w))`.

D1) Minimal API (reference signatures)

Numerics & clamps

```
clamp_align(a, eps_a=1e-6) -> a_c              # |a_c| < 1
to_u(a, eps_a=1e-6) -> u                       # u :=
atanh(clamp_align(a))
from_u(u) -> a                                 # a := tanh(u)
```

Order-invariant fuse (streaming, shard-safe)

```
fuse_init(eps_w=1e-12) -> state(U=0.0, W=0.0)
fuse_add(state, a, w=1.0, eps_a=1e-6) -> None   # U += w*atanh(a_c); W += w
fuse_merge(state, other_state) -> None          # U += other.U; W +=
other.W
fuse_value(state) -> a_out                      # tanh(U / max(W, eps_w))
```

Lens → Align → RSI (two-channel form)

```
rsi_from_e(e_items, c=1.0, eps_w=1e-12, eps_a=1e-6,
           w_policy="uniform", gamma=1.0) -> RSI
# e_items: iterable of (e_in, e_out, m_or_w)
# weights: if w_policy=="abs_m_gamma" then w := |m_or_w|^gamma else w := 1
# a_in  := tanh(-c*e_in)
# a_out := tanh(+c*e_out)
# U_in  := sum w*atanh(a_in);  V_out := sum w*atanh(a_out);  W_in := sum w
# RSI   := tanh( (V_out - U_in) / max(W_in, eps_w) )
```

Calm gate (alignment-only)

```
gate_apply(RSI, g, mode="mul") -> RSI_env
# "mul"    : RSI_env := g * RSI
# "u_scale": RSI_env := tanh( g * atanh(RSI) )
```

Path scoring (push/pop in u-space)

```
path_init(eps_a=1e-6, eps_w=1e-12) -> path(U=0.0, W=0.0, stack=[])
path_push(path, RSI_used, w=1.0, beta=0.0, b=0.0) -> None
# u_step := atanh(clamp_align(RSI_used, eps_a)) + beta*b
# U += w*u_step; W += w; push( (w*u_step, w) )
path_pop(path) -> None                          # pop last (ΔU, ΔW)
path_value(path) -> RSI_path                    # tanh(U / max(W, eps_w))
```

Bands (defaults)

```
to_band(x) -> {"A++","A+","A0","A-","A--"}
# A++: x >= +0.90
# A+ : +0.60 <= x < +0.90
# A0 : -0.60 <  x < +0.60
# A- : -0.90 <  x <= -0.60
# A--: x <= -0.90
```

D2) Invariants & policies (normative)

- **Collapse parity.** `phi((m,a)) = m` in all APIs.
- **Clamp-first.** Always call `clamp_align` before `atanh`.
- **Order/shard invariance.** Carry only `(U,W)`; never average in a-space.
- **Division policy (lane M2).** `a* := tanh(atanh(a1) ± atanh(a2))` for mul/div in the lane; magnitude math stays classical.
- **Gate purity.** `gate_apply` acts on alignment only; never mutate `m`.
- **Determinism.** Same manifest ⇒ same outputs (freeze `eps_a, eps_w, weights, bands, gate mode, c, Unit`).
- **Numeric hygiene.** Use float64 for all SDK computations; never construct `atanh(±1)` (clamp first).

D3) Golden vectors (must match within dtype tolerance)

(All angles in radians; report to ≥ 6 decimals.)

1. **Clamp + round-trip**
   `a_in = 0.9999999`, `eps_a=1e-6`
   `a_c = 0.999999`
   `u = atanh(a_c)` # finite
   `a_rt = tanh(u)` # ≈ `0.999999` within tolerance
2. **Fuse order invariance**
   `tanh(0.2) = 0.197375` ; `tanh(0.4) = 0.379949`
   `U = 0.2 + 0.4 = 0.6` ; `W = 2`
   `a_out = tanh(0.6/2) = tanh(0.3) = 0.291313`
   (Swap inputs or shard/merge → same `0.291313`.)
3. **RSI (single item)**
   `e_in=0.2`, `e_out=0.5`, `c=1`, `w=1`
   `RSI = tanh( (0.5 - (-0.2)) ) = tanh(0.7) = 0.604368`

4. **Gate (two modes) with `g=0.81`, RSI from (3)**
   `"mul"` → `RSI_env = 0.81 * 0.604367777 = 0.489538`
   `"u_scale"` → `RSI_env = tanh( 0.81 * atanh(0.604367777) ) = tanh(0.567) = 0.513153`
   (Choose one convention and freeze it in the manifest; both keep `|RSI_env| < 1`.)
5. **Path push/pop**
   Start: `U=W=0`
   Push `RSI_used = tanh(0.5869) = 0.527662`, `w=1`
   `u1 = atanh(0.527662) = 0.586900` ; `U=0.586900` ; `W=1`
   Push `RSI_used = tanh(0.4) = 0.379949`
   `u2 = atanh(0.379949) = 0.400000` ; `U=0.986900` ; `W=2`
   `RSI_path = tanh(U/W) = tanh(0.493450) = 0.456950`
   Pop last → `U=0.586900` ; `W=1` ; `RSI_path = tanh(0.586900) = 0.527662`
6. **Weighted roll-up**
   `RSI_a = 0.604368 (U=0.700000, w=2)` ; `RSI_b = 0.291313 (U=0.300000, w=1)`
   `U = 2*0.7 + 1*0.3 = 1.700000` ; `W=3`
   `RSI_win = tanh(1.7/3) = tanh(0.566667) = 0.512907` → **band A0**
7. **Bands**
   ```
   to_band( 0.910425 ) -> "A++"
   to_band( 0.691069 ) -> "A+"
   to_band( 0.291313 ) -> "A0"
   to_band(-0.875000 ) -> "A-"
   to_band( 0.000000 ) -> "A0"
   ```

D4) Reference pseudocode (concise, glueable)

```
def rsi_from_e(e_items, c=1.0, eps_w=1e-12, eps_a=1e-6,
               w_policy="uniform", gamma=1.0):
    U_in = V_out = W_in = 0.0
    for (e_in, e_out, m_or_w) in e_items:
        w = (abs(m_or_w)**gamma) if (w_policy=="abs_m_gamma") else 1.0
        a_in  = tanh(-c*e_in);  a_out = tanh(+c*e_out)
        U_in += w * atanh(max(-1+eps_a, min(1-eps_a, a_in  )))
        V_out+= w * atanh(max(-1+eps_a, min(1-eps_a, a_out )))
        W_in += w
    return 0.0 if W_in <= 0 else tanh((V_out - U_in) / max(W_in, eps_w))

def fuse_init(eps_w=1e-12):
    return {"U": 0.0, "W": 0.0, "eps_w": eps_w}

def fuse_add(state, a, w=1.0, eps_a=1e-6):
    a_c = max(-1+eps_a, min(1-eps_a, a))
    state["U"] += w * atanh(a_c)
    state["W"] += w

def fuse_merge(state, other):
    state["U"] += other["U"]; state["W"] += other["W"]

def fuse_value(state):
    return tanh(state["U"] / max(state["W"], state["eps_w"]))

def gate_apply(RSI, g, mode="mul"):
    return (g*RSI) if mode == "mul" else tanh(g*atanh(RSI))
```

```
def path_init(eps_a=1e-6, eps_w=1e-12):
    return {"U": 0.0, "W": 0.0, "stack": [], "eps_a": eps_a, "eps_w":
eps_w}

def path_push(path, RSI_used, w=1.0, beta=0.0, b=0.0):
    a_c = max(-1+path["eps_a"], min(1-path["eps_a"], RSI_used))
    u_step = atanh(a_c) + beta*b
    path["U"] += w * u_step
    path["W"] += w
    path["stack"].append((w*u_step, w))

def path_pop(path):
    if not path["stack"]: return
    dU, dW = path["stack"].pop()
    path["U"] -= dU; path["W"] -= dW

def path_value(path):
    return tanh(path["U"] / max(path["W"], path["eps_w"]))
```

D5) Acceptance checklist (must pass)

- **Parity.** `phi((m,a)) = m` across all SDK flows.
- **Clamp.** Outputs from `clamp_align` satisfy `|a_c| < 1`.
- **Order/shard.** `fuse_value` identical under permutations and `fuse_merge`.
- **Bounds.** All `RSI`, `RSI_env`, `RSI_path` in `(-1,+1)`.
- **Determinism.** Golden vectors reproduce numerically within dtype tolerance.
- **Numeric policy.** `float64` end-to-end; no reliance on implicit rounding.

D6) Manifest keys (SDK expectations)

```
{
  "eps_a": 1e-6,
  "eps_w": 1e-12,
  "weights": {"policy": "uniform", "gamma": 1.0},   # or
{"policy":"abs_m_gamma","gamma":1.0}
  "combine_policy": "M2",
  "division_policy": "strict",
  "lens": {"Unit": 1.0, "c": 1.0},
  "gate": {"mode": "mul", "rho": 0.20, "g_min": 0.00},
  "bands": {"A++":0.90, "A+":0.60, "A0":-0.60, "A-":-0.90, "A--":-1.00}
}
```

One-line takeaway.
This appendix nails down a minimal, portable SDK and a set of golden vectors so any team can implement the exact same bounded math — `atanh` in, `tanh` out, `U/W` for fusion — while keeping every classical number identical via `phi((m,a)) = m`.

Appendix E — Vendor Bake-off Protocol (fair, bounded, reproducible)

Purpose. Standardize cross-vendor/model comparisons using the same observation-only math and stamps. Classical numbers remain untouched: `phi((m,a)) = m`. Selection and reporting use bounded alignment and the order-invariant fuse: `u := atanh(a)`, `U += w*u`, `W += w`, `a_out := tanh(U / max(W, eps_w))`.

E1) Scope & prerequisites (freeze before you run)

- **Manifest freeze (non-negotiable).** Freeze: `eps_a`, `eps_w`, weights policy, `combine_policy="M2"`, division policy, band thresholds, lens params (`Unit`, `c`), gate mode (`"mul"` or `"u_scale"`). Compute and publish `knobs_hash`.
- **Traffic & sets.** Choose exactly one: `shadow_traffic` (live mirror) or `frozen_eval_set` (static prompts/queries/documents).
- **Randomness.** Fix seeds for any stochastic decoding; log seeds in stamps.
- **Stamping.** Every decision emits a one-line stamp and a ledger row (per Appendix C).
- **Observation-only.** No calibration or post-hoc transforms per vendor; classical numbers `m` are never altered inside SSM-AI (`phi((m,a)) = m`).

E2) What to log per decision (minimum ledger row)

```
iso_utc, svc, req_id, item_id, RSI, w, g, RSI_env, band, U_dec :=
atanh(RSI), W_dec := w, manifest := knobs_hash, seed, stamp
```

Optional overlays for ops: `tokens, lat_ms, cost_unit`, and any classical metric `m` your pipeline already emits.

E3) How to aggregate per vendor (bounded, order-invariant)

For any bucket (e.g., per task, per hour, per domain), compute:

- **Ungated pool (intrinsic capability).**
  `U_pool := sum U_dec` ; `W_pool := sum W_dec` ; `RSI_pool := tanh( U_pool / max(W_pool, eps_w) )`
- **Gated pool (live readiness).** Apply the gate **per decision** before pooling, then fuse in u-space.
  `"mul"`: `RSI_env := g * RSI`
  `"u_scale"`: `RSI_env := tanh( g * atanh(RSI) )`
  Then: `U_env := sum atanh(RSI_env)` ; `RSI_pool_env := tanh( U_env / max(W_pool, eps_w) )`
- **Band distribution.** Counts/fractions of `A++/A+/A0/A-/A--` over `RSI_env`.

- **Cost/latency overlays.** Report medians/means alongside `RSI_pool_env` (never mix them into the bounded index).

Notes.

- Always pool in u-space (`atanh`) to preserve order/shard invariance.
- Never average directly in a-space; only `(U,W)` may be merged across shards.

E4) Tie-breakers & significance (simple, portable)

- **Primary rank.** By `RSI_pool_env` (or `RSI_pool` if comparing intrinsic capability).
- **Tie-break 1.** Higher fraction of `A++`, then `A+`.
- **Tie-break 2.** Lower `tokens` and lower `lat_ms` at equal `RSI_pool_env`.
- **Significance (bootstrap in u-space).** Convert each decision to `u := atanh(RSI_env)`. Resample both vendors with replacement `N` times (e.g., `N=1000`), compute `mean(u_A) - mean(u_B)`. Report two-sided `p` and a `95% CI` on the difference in u-space; optionally map CI ends back via `tanh` for display.

Pseudocode (bootstrap sketch).

```
def diff_ci(uA, uB, N=1000):
    diffs = []
    for _ in range(N):
        sA = mean(random_resample(uA))
        sB = mean(random_resample(uB))
        diffs.append(sA - sB)
    diffs.sort()
    lo, hi = diffs[int(0.025*N)], diffs[int(0.975*N)]
    p = min(sum(d <= 0 for d in diffs), sum(d >= 0 for d in diffs)) / N
    return (lo, hi, p)
```

E5) Worked mini-example (calculator-fast, gated per decision)

- **Vendor A decisions (3).** `RSI = [0.604368, 0.291313, 0.910425]`, all `w = 1`.
  `u = [0.700000, 0.300000, 1.530000]` → `U_pool = 2.530000`, `W_pool = 3` → `RSI_pool = tanh(2.53/3) = tanh(0.843333) ≈ 0.687571` → **band A+**.
- **Vendor B decisions (3).** `RSI = [0.691069, 0.462117, 0.291313]`, `w = 1`.
  `u = [0.850000, 0.500000, 0.300000]` → `U_pool = 1.650000`, `W_pool = 3` → `RSI_pool = tanh(1.65/3) = tanh(0.550000) ≈ 0.500520` → **band A0**.
- **With gate `g = 0.80`, mode `"mul"` (apply per decision, then pool):**
  Vendor A per-decision `RSI_env = [0.483494, 0.233050, 0.728340]`
  `u_env ≈ [atanh(0.483494), atanh(0.233050), atanh(0.728340)] ≈ [0.527534, 0.237412, 0.925183]`
  `U_env ≈ 1.690129`, `W=3` → `RSI_pool_env = tanh(1.690129/3) = tanh(0.563376) ≈ 0.510478` → **band A0**.

  Vendor B per-decision `RSI_env = [0.552855, 0.369694, 0.233050]`
  `u_env ≈ [0.622484, 0.388069, 0.237412]`

`U_env ≈ 1.247965`, `W=3` → `RSI_pool_env = tanh(1.247965/3) = tanh(0.415988) ≈ 0.393545` → **band A0**.

- **Band distributions** (per decision on `RSI_env` with `g=0.80`).
  Vendor A: `A+ : 1` (from `0.728340`), `A0 : 2`, `A++ : 0`, `A- : 0`, `A-- : 0`.
  Vendor B: `A+ : 1`, `A0 : 2`, `A++ : 0`, `A- : 0`, `A-- : 0`.

Takeaway. A leads on the bounded chooser both ungated (`0.687571` vs `0.500520`) and gated (`0.510478` vs `0.393545`) while bands grow more conservative under gating.

E6) Run procedure (10 steps, copy-paste)

1. Freeze manifest and publish `knobs_hash`.
2. Select traffic or eval set; fix seeds.
3. Enable stamps and ledger (per Appendix C).
4. Run Vendor A and Vendor B with identical prompts/tools and the same manifest.
5. For each decision, compute and log `RSI`, `RSI_env := gate(RSI)`, `band`, `U_dec := atanh(RSI)`, `W_dec := w`.
6. Roll up per bucket: `U_pool := sum U_dec`, `W_pool := sum W_dec`, `RSI_pool := tanh(U_pool / max(W_pool, eps_w))`.
7. Repeat with gated per-decision `RSI_env` to get `RSI_pool_env`.
8. Produce band histograms, cost/latency overlays, and bootstrap CI in u-space.
9. Stamp a one-line summary per bucket:
   `"SSMBO|bucket=decode_hour_14|A.U=...|A.W=...|A.RSI_pool=...|B.U=...|B.W=...|B.RSI_pool=...|g=...|mode=mul|manifest=knobs_hash"`
10. Publish a one-page table per surface with `RSI_pool_env`, band shares, `tokens`, `lat_ms`, and the CI.

E7) Edge cases & guardrails

- **Mismatched candidate counts.** Aggregate at the decision level (per item). If per-request pairing is required by the surface, pool each request first (in u-space), then pool across requests.
- **Missing telemetry for gate.** Set `g := 1` and flag the row.
- **Vendor-specific truncation.** Do not normalize `RSI` post-hoc; the point of `(-1,+1)` is comparability without calibration.
- **Shard merges.** Only merge `(U,W)`; never average `RSI` directly.
- **Fallback on breach.** If acceptance gates fail (collapse parity, order invariance, clamp bounds, gate purity), revert analysis to classical `m`-based baselines for that slice and flag the bucket.
- **Numeric hygiene.** Always clamp before `atanh`; carry `(U,W)` as float64; guard denominators with `max(W, eps_w)`.

E8) Report template (per bucket)

bucket	RSI_pool_env	95% CI (u-space)	A++/A+/A0/A-/A--	tokens	lat_ms
Vendor A	0.510478	[+0.08, +0.21]	0/1/2/0/0	0.92x	310
Vendor B	0.393545	[0.00, 0.00]	0/1/2/0/0	1.00x	345

Verdict. A leads by `~+0.116933` RSI (bounded), significant (`p≈0.03`) in u-space.
(The CI is computed on `u := atanh(RSI_env)`; display may also show the mapped ends via `tanh`.)

E9) Acceptance checklist (pass/fail)

- **Determinism.** Same manifest + same inputs ⇒ identical `RSI_pool` and `RSI_pool_env`.
- **Order/shard invariance.** Permutations and shard merges of decisions leave pools unchanged.
- **Boundedness.** All `RSI` and `RSI_env` in `(-1,+1)`; pools too.
- **Stamp completeness.** Each bucket summary includes `knobs_hash`.
- **No mutation of m.** Verified by re-running collapse: `phi((m,a)) = m`.

One-line takeaway.
Freeze the manifest, stamp every decision, and compare vendors by the same bounded index: `RSI_pool := tanh( sum atanh(RSI) / max(sum 1, eps_w) )` (and the gated variant). It is order-invariant, shard-safe, reproducible, and leaves `m` pristine via `phi((m,a)) = m`.

Appendix F — SSM-Audit CFO Pack (3–5 KPI lanes, weekly roll-ups, ROI)

Purpose. Publish a small set of bounded KPI lanes beside existing service metrics so finance/ops can see stability, cost, and latency improvements clearly, without changing your numbers. All math is observation-only: `phi((m,a)) = m` and the lane acts on alignment only. Decisions and roll-ups use the order-invariant fuse: `u := atanh(a)`, `U += w*u`, `W += w`, `a_out := tanh(U / max(W, eps_w))`.

F1) The CFO view (what they get)

- One bounded portfolio index (weekly): `RSI_port := tanh( U_port / max(W_port, eps_w) ).`
- Band shares: `A++/A+/A0/A-/A--` per service and portfolio.
- Before vs SSM-AI worksheet: token/cost/latency savings with auditable math.
- Stamped ledger: replay/roll-up is calculator-fast and deterministic.

F2) KPI lens library (declare 3–5 lanes; dimensionless)

All inputs normalized to `[0,1]`; publish transformations in the manifest.

Lane 1 — Cost efficiency (tokens & retries)

```
tokens_drop := clamp( (tokens_before - tokens_after) / max(tokens_before, eps), 0, 1 )
retry_drop := clamp( (retry_before - retry_after ) / max(retry_before, eps), 0, 1 )
e_cost := (alpha_tok * tokens_drop + alpha_ret * retry_drop) / Unit
```

Lane 2 — Latency health (tail aware)

```
on_time_rate := clamp( 1 - miss_rate, 0, 1 )
tail_p95_penalty := clamp( (p95_ms - SLO_ms) / max(SLO_ms, eps), 0, 1 )
tail_p99_penalty := clamp( (p99_ms - SLO_ms) / max(SLO_ms, eps), 0, 1 )
e_lat := (alpha_on * on_time_rate - beta_p95 * tail_p95_penalty - beta_p99 * tail_p99_penalty) / Unit
```

Lane 3 — Quality stability (retries/contradictions)

```
e_qual := (alpha_succ * success_rate - beta_retry * retry_rate - beta_contra * contradiction_rate) / Unit
```

Lane 4 — Incidents & escalations (optional)

```
mttr_drop := clamp( (mttr_before - mttr_after) / max(mttr_before, eps), 0, 1 )
mtbf_gain := clamp( (mtbf_after - mtbf_before) / max(mtbf_before, eps), 0, 1 )
e_inc := (alpha_mttr * mttr_drop + alpha_mtbf * mtbf_gain - beta_page * page_rate) / Unit
```

Mapping (symmetric, bounded)

`a_out := tanh( +c * e_out )` ; `a_in := tanh( -c * e_in )`
If a lane emits a single signed contrast `e`, use `a := tanh(c * e)`.

Weights for roll-ups
Use portfolio weights per row: `w := revenue_share` or `w := 1` (uniform). Declare once.

F3) Manifest (copy-paste block)


```
"ssm_audit": {
  "eps_a": 1e-6, "eps_w": 1e-12,
  "weights": {"policy": "uniform"},                    // or
{"policy":"abs_m_gamma","gamma":1.0}
  "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00},
  "gate_ref": "gate_preset_A",
  "kpi_lenses": {
    "cost": {"alpha_tok":1.0, "alpha_ret":1.0, "Unit":1.0, "c":1.0},
    "lat":  {"alpha_on":1.0, "beta_p95":1.0, "beta_p99":0.5, "Unit":1.0,
"c":1.0},
    "qual": {"alpha_succ":1.0, "beta_retry":1.0, "beta_contra":1.0,
"Unit":1.0, "c":1.0},
    "inc":  {"alpha_mttr":1.0, "alpha_mtbf":1.0, "beta_page":1.0,
"Unit":1.0, "c":1.0}
  }
}
```

F4) Ledger (minimal columns for finance roll-ups)

For each service/week row (or per decision if you prefer finer granularity), log:
`svc, iso_week, KPI, RSI, w, g, RSI_env, band, U_dec := atanh(RSI), W_dec := w, knobs_hash`.

Portfolio roll-up (per KPI).

`U_port := sum U_dec` ; `W_port := sum W_dec` ; `RSI_port := tanh( U_port / max(W_port, eps_w) )`.
With gating: compute `RSI_env := g * RSI` (or `tanh(g*atanh(RSI))`), then pool the same way for `RSI_port_env`.

F5) Before vs SSM-AI worksheet (copy-paste)

Inputs (weekly).

```
requests, tokens_per_req_before, cost_per_1k, retry_rate_before,
retry_rate_after, p95_before_ms, p95_after_ms, savings_pct_tokens
```

Derived.

```
tokens_before := requests * tokens_per_req_before
spend_before := (tokens_before / 1000) * cost_per_1k
tokens_saved := tokens_before * savings_pct_tokens
spend_saved := (tokens_saved / 1000) * cost_per_1k
spend_after := spend_before - spend_saved
latency_delta_ms := p95_before_ms - p95_after_ms
retry_delta := retry_rate_before - retry_rate_after
```

ROI (annualized).

```
annual_savings := 52 * spend_saved
ROI := (annual_savings - integration_cost) / max(integration_cost, eps)
```

F6) Worked example (calculator-fast)

Assume weekly: `requests = 100000`, `tokens_per_req_before = 800`, `cost_per_1k = 0.50`, `savings_pct_tokens = 0.08`.

- `tokens_before = 100000 * 800 = 80000000`
- `spend_before = (80000000 / 1000) * 0.50 = 40000.0`
- `tokens_saved = 80000000 * 0.08 = 6400000`
- `spend_saved = (6400000 / 1000) * 0.50 = 3200.0`
- `spend_after = 40000.0 - 3200.0 = 36800.0`
- Example latency: `p95_before_ms = 900`, `p95_after_ms = 780` → `latency_delta_ms = 120`.
- Bounded portfolio index (illustrative): before `U/W = 0.5` → `RSI_port = tanh(0.5) = 0.462117` (**band A0**); after `U/W = 0.7` → `RSI_port = tanh(0.7) = 0.604368` (**band A+**).

Annual ROI with `integration_cost = 50000`:
`annual_savings = 52 * 3200.0 = 166400.0`
`ROI = (166400.0 - 50000) / 50000 = 2.328` → `232.800000%`.

F7) Dashboard tiles (definitions)

- Portfolio RSI (weekly): `RSI_port_env` and band.
- Band share: `%A++`, `%A+`, `%A0`, `%A-`, `%A--` (from per-row `RSI_env`).
- Cost tile: `spend_before`, `spend_after`, `spend_saved`, `% saved`.
- Latency tile: `p95_before_ms`, `p95_after_ms`, `latency_delta_ms`.
- Retries: `retry_rate_before`, `retry_rate_after`, `retry_delta`.
- Stamp preview: last `N` stamps with `svc|week|U|W|RSI_port|band|g`.

F8) Pseudocode (compact, copy-paste)

```
def kpi_rsi(e_pos, e_neg=0.0, c=1.0, w=1.0, eps_a=1e-6, eps_w=1e-12):
    a_out = tanh(+c * e_pos)
    a_in  = tanh(-c * e_neg)
    U = atanh(max(-1+eps_a, min(1-eps_a, a_out)))
    V = atanh(max(-1+eps_a, min(1-eps_a, a_in )))
    return tanh((U - V) / max(1.0, eps_w))  # RSI for this KPI row

def rollup(rows, use_env=False):
    U = W = 0.0
    for r in rows:  # rows: iterable of {"RSI":..., "g":..., "w":...}
        x = r["g"]*r["RSI"] if use_env else r["RSI"]
        x = max(-1+1e-6, min(1-1e-6, x))
        U += r["w"] * atanh(x)
        W += r["w"]
    return 0.0 if W <= 0 else tanh(U / max(W, 1e-12))
```

F9) Acceptance checklist (pass/fail)

- Parity. Classical spend/latency numbers are unchanged by the lane (`phi((m,a)) = m`).
- Order/shard invariance. Weekly and portfolio roll-ups use `(U,W)`; permutations and shard merges match.
- Bounds. All `RSI`, `RSI_env`, `RSI_port` in `(-1,+1)`; bands map per thresholds.
- Transparency. Every KPI term and weight is logged; no hidden factors.
- Determinism. Same manifest and inputs ⇒ identical roll-ups and ROI worksheet.

One-line takeaway. Publish a handful of KPI lanes, roll them up with `U/W` in u-space, and show a weekly bounded index plus a simple savings worksheet — clear to finance, trivial to replay, and fully observation-only: `phi((m,a)) = m`.

Stamp example (append to any weekly portfolio report line).

```
SSMCLOCK1|iso_utc|svc=portfolio|week=2025-
W41|U=2.100000|W=3.000000|RSI_port=0.604368|g=0.81|RSI_port_env=0.489537|ba
nd=A0|manifest=knobs_hash
```

Appendix G — SSMH Acceleration Parity (fixed-point, tiny MAC, identical semantics)

Purpose. Map the lane math to a tiny, deterministic hardware substrate without changing semantics. Classical values remain untouched: `phi((m,a)) = m`. The lane uses the same kernel: `a_c := clamp(a, -1+eps_a, +1-eps_a)`, `u := atanh(a_c)`, streaming fuse `U += w*u`, `W += w`, and `a_out := tanh( U / max(W, eps_w) )`. Optional gate: `RSI_env := g * RSI` or `RSI_env := tanh( g * atanh(RSI) )`.

G1) Micro-architecture (streaming)

Blocks (in order):
`CLAMP → ATANH_LUT → W_MUL → ACC_UW → DIV_SAT → TANH_LUT → BAND → STAMP`

- **CLAMP:** implements `a_c := max(-1+eps_a, min(1-eps_a, a))`.
- **ATANH_LUT:** piecewise LUT + short polynomial giving `u ≈ atanh(a_c)`.
- **W_MUL:** fixed-point multiply for `w*u` and integer add for `W += w`.
- **ACC_UW:** accumulators for `U` and `W`.
- **DIV_SAT:** computes `u_bar := U / max(W, eps_w)` with divide-by-zero guard.
- **TANH_LUT:** piecewise LUT + polynomial for `a_out := tanh(u_bar)`.

- **BAND:** compares `a_out` (or `RSI_env`) to thresholds.
- **STAMP:** emits a one-line ASCII with fields and a `knobs_hash`.

Merge/restart state: only `(U, W)` plus manifest constants.

G2) Numeric ranges and fixed-point formats

Max rapidity to cover (with `eps_a = 1e-6`):
`u_max = atanh(1 - eps_a) = 0.5 * ln((2 - eps_a)/eps_a) ≈ 7.254325` → cover at least `[-7.5, +7.5]`.

Recommended fixed-point (lane-only):

- **16-bit (low cost):** `Q4.12` for `u`, `U/W`, and `w*u`. Range `[-8, +8)` with step `≈ 2.44e-4`.
- **32-bit (comfort):** `Q6.26` for `u`, `U/W`, and `w*u`. Range `[-32, +32)` with step `≈ 1.49e-8`.
- **Weights `w`:** if uniform, `w := 1` (integer). If `w := |m|^gamma`, quantize `w` to `Q6.10` or `Q8.8` depending on spread.

Guard constants (manifest):
`eps_a = 1e-6`, `eps_w = 1e-12`. Pre-quantize for RTL: `eps_a_fx := to_fx(eps_a)`, `eps_w_fx := to_fx(eps_w)`.

G3) Piecewise LUTs (tanh / atanh)

atanh (segments on `a_c ∈ [0, 1 - eps_a]`, mirror to negatives):
Example 16-bit segmentation: `{0..0.5, 0.5..0.8, 0.8..0.95, 0.95..0.995, 0.995..0.999999}`.
Each segment stores `k0 + k1*x + k2*x^2` in fixed-point. **Error target:** `|delta_u| <= 1e-4`.

tanh (segments on `u ∈ [0, 4]`, saturation above `4`):
Example 16-bit segmentation: `{0..0.5, 0.5..1.0, 1.0..2.0, 2.0..4.0}`. **Error target:** `|delta_a| <= 1e-4`.

Monotonicity: both LUTs must be strictly monotone to preserve ordering.

G4) Pipeline and latency (one sample per cycle, typical)

- `CLAMP: 1`
- `ATANH_LUT: 2-3` (addr + MAC)
- `W_MUL: 1`
- `ACC_UW: 1`

- `DIV_SAT: 6-12` (iterative or DSP divide)
- `TANH_LUT: 2-3`

Total: ~`14-21` cycles latency, throughput `1` result/cycle after fill.
Shard merge: add-only on `(U, W)` (no LUTs).

G5) Gate and bands in hardware

- **Gate `"mul"`:** `RSI_env := g * RSI` → single fixed-point multiply.
- **Gate `"u_scale"`:** `RSI_env := tanh( g * atanh(RSI) )` → reuse LUTs.
- **Bands:** four comparators against `0.90, 0.60, -0.60, -0.90` in the same fixed-point domain.
- **Hysteresis:** compare with offsets `h_up`, `h_dn` in fixed-point.
- **Purity:** alignment-only; `m` never enters the datapath (`phi((m,a)) = m`).

G6) Determinism & parity tests (must pass)

Golden vectors (fixed-point must match float within tolerance):

1. `a1 := tanh(0.2)`, `a2 := tanh(0.4)`, `w1=w2=1`
   Expect `a_out := tanh( (0.2+0.4)/2 ) = tanh(0.3) ≈ 0.291313`.
2. `RSI := tanh(0.7) ≈ 0.604368` → `U=0.700000`, `W=1` → replay equals input.
3. **Lane mul/div (M2):**
   `a_mul := tanh(0.5 + 0.2) ≈ 0.604368`, `a_div := tanh(0.5 - 0.2) ≈ 0.291313`.
4. **Gate `"mul"` with `g=0.81`:** `RSI_env := 0.81 * 0.604368 ≈ 0.489538`.
5. **Gate `"u_scale"` with `g=0.81`, `RSI=0.70`:**
   `RSI_env := tanh( 0.81 * atanh(0.70) ) = tanh( 0.81 * 0.867301 ) ≈ 0.605961`.
6. **Order/shard invariance:** permute or shard a stream; final `a_out` identical.

Tolerance targets:

- **16-bit path:** `|delta_RSI| <= 5e-4`, band decisions identical.
- **32-bit path:** `|delta_RSI| <= 5e-7`, bit-exact bands and stamps.

G7) Fixed-point arithmetic details

- **Multiply:** widen then round (e.g., `Q4.12 * Q4.12 -> Q8.24`, then round to `Q6.26` or `Q4.12`).
- **Accumulators:** keep wider (`U_acc`, `W_acc`) to avoid overflow. For `1e6` items with `w=1`, `W_acc` needs `>= 20` integer bits → choose `32-bit` or `40-bit` accumulators.
- **Division guard:** implement `u_bar := U / max(W, eps_w_fx)`. If `W=0`, output `0`.

- **Saturation:** saturate on overflow to nearest representable; never wrap.

G8) Resource sketch (indicative, not a promise)

- **LUTs/BRAM:** `1–2` BRAMs for coefficient tables per function (tanh, atanh).
- **DSPs:** `4–8` for multiplies (poly eval, `w*u`, gate).
- **ALMs/LUTs:** small for adders/comparators/clamp.
- **Clocks:** `100–300 MHz` on mid-range FPGAs; ASICs higher.
  Throughput scales linearly with more lanes; merges are add-only.

G9) Manifest fields for hardware builds

```
"ssmh": {
  "fx": {"u_fmt": "Q6.26", "a_fmt": "Q2.14", "w_fmt": "Q8.8"},
  "eps_a": 1e-6, "eps_w": 1e-12,
  "gate": {"mode": "mul"},                      // or "u_scale"
  "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00},
  "luts": {"atanh_seg": 5, "tanh_seg": 4, "poly_degree": 2},
  "acc_width": {"U_bits": 40, "W_bits": 24},
  "weights": {"policy": "uniform"},
  "combine_policy": "M2",
  "division_policy": "strict"
}
```

G10) RTL-style pseudocode (concise)

```
# Inputs per item: a_in_fx (Q2.14), w_fx (Q8.8)
a_c   = clamp_fx(a_in_fx, -1 + eps_a_fx, 1 - eps_a_fx)       # CLAMP
u     = atanh_lut_poly(a_c)                                  # ATANH_LUT
wu    = mul_fx(w_fx, u)                                      # W_MUL
U     = sat_add_fx(U, wu);  W = sat_add_fx(W, w_fx)          # ACC_UW
u_bar = div_guard_fx(U, W, eps_w_fx)                         # DIV_SAT
a_out = tanh_lut_poly(u_bar)                                 # TANH_LUT
band  = band_of(a_out, thr_pp=0.90, thr_p=0.60,
                thr_n=-0.60, thr_nn=-0.90)                   # BAND
stamp = make_stamp(U, W, a_out, band, knobs_hash)            # STAMP
```

G11) Acceptance and QA (hardware)

- **Bit-parity mode:** feed float golden vectors, quantize to fixed-point, compare outputs within tolerance; assert identical band decisions and stamps.
- **Throughput test:** sustained `1` sample/cycle after pipeline fill.
- **Merge test:** two lanes processing disjoint halves, then `(U,W)` add equals single-lane result.
- **Reset/restart test:** persisting `(U,W)` across resets reproduces the same `a_out`.
- **Safety:** division guard never underflows; clamps enforce `|a| < 1`.

One-line takeaway. The lane is a tiny streaming MAC with two monotone LUTs: atanh in, tanh out, accumulated as `(U,W)`. With fixed-point formats like `Q6.26`, you get deterministic, order-invariant, stamp-ready results that are semantically identical to software, while `phi((m,a)) = m` keeps all classical numbers pristine.

Stamp example (hardware snapshot).

```
SSMCLOCK1|iso_utc|svc=lane_hw|U=0.700000|W=1.000000|RSI=0.604368|g=0.81|RSI
_env=0.489538|band=A0|fx=Q6.26|manifest=knobs_hash
```

Appendix H — Comparisons & Synergies (entropy, MC-dropout, ensembles, conformal, evidential)

Purpose. Show how popular uncertainty/confidence signals relate to the SSM-AI lane and how to plug them in without changing classical numbers. All integrations are observation-only: `phi((m,a)) = m`. We use the same kernel for fusion and selection: clamp, rapidity map `u := atanh(a)`, add in u-space, invert with `a := tanh(u)`, choose by `RSI` (or `RSI_env := g_t * RSI`).

H1) Why compare? (positioning in one paragraph)

Most confidence add-ons are powerful but disagree across tasks, scales, and vendors. SSM-AI provides a bounded, portable lane `a in (-1,+1)` and a single chooser `RSI in (-1,+1)`; everything else (entropy, MC-dropout, deep ensembles, conformal, evidential) becomes a lens producing a signed contrast `e`, mapped to alignment via `a := tanh(c*e)` (or the two-channel `a_in := tanh(-c*e_in)`, `a_out := tanh(+c*e_out)`). Then we pool order-invariantly with `U += w*atanh(a)`, `W += w`, `a_out := tanh(U / max(W, eps_w))`. Classical magnitudes `m` remain intact: `phi((m,a)) = m`.

H2) Cheat-sheet — map common methods into a lens

All formulas are plain ASCII; choose signs so "better" evidence pushes positive.

1. **Softmax entropy (categorical p).**
   `H := -sum_i p_i * log(p_i)` (nats). Confidence proxy: `conf_H := 1 - H / H_max`, where `H_max := log(K)` for `K` classes.

Lens contrast (supportive when entropy is low): `e := (conf_H - tau) / Unit`.
Alignment: `a := tanh(c * e)`.

2. **Top-two margin (classification).**
   `margin := p_top1 - p_top2 in [0,1]`.
   Lens: `e := (margin - tau) / Unit` → `a := tanh(c * e)`.
3. **Temperature-scaled confidence.**
   With temperature `T > 0`, logits `z/T` give probs `p_T`. Let `conf_T := p_T_top1`.
   Lens: `e := (conf_T - tau) / Unit` → `a := tanh(c * e)`.
4. **MC-dropout predictive variance (regression).**
   `var_mc := mean_t( (y_t - mean_t y_t)^2 )`. Turn variance into a penalty via a robust target `s`:
   `e := (s - var_mc) / Unit` *(or `e := - var_mc / Unit`)*.
   Two-channel form: `e_in := var_mc / Unit`, `e_out := 0`.
5. **MC-dropout class entropy (classification).**
   Average predictive `p_bar := mean_t p_t`; use entropy on `p_bar`. Lens as in (1).
6. **Deep ensembles (classification or regression).**
   For models `j=1..J`, get `p_bar := mean_j p^{(j)}` or regression variance `var_ens`. Lens mirrors (1) or (4).
   Optional diversity prior in u-space: `u' := u + beta * b`, where `b := clamp(diversity_idx, -1, +1)`.
7. **Conformal prediction (classification).**
   Given per-example nonconformity score `s`, lower is "better."
   Lens: `e := (s_ref - s) / Unit`. If you produce a set size `|S|`, penalty lens: `e_in := (|S| - 1)/Unit`.
8. **Evidential deep learning (Dirichlet).**
   Evidence `e_k >= 0`, strength `S := sum_k e_k`. Higher `S` ⇒ higher certainty.
   Lens: `e := (S - S_ref) / Unit` (optionally combine with expected prob gap). `a := tanh(c * e)`.
   For NIG regression, map expected `sigma^2` and epistemic proxies as penalties to `e_in`.
9. **Calibrated probability (Platt/Isotonic).**
   After external calibration, use `p_cal_top1`. Lens identical to (2)/(3).
10. **Entropy of RAG retrieval set.**
    For normalized importances `q_i`, `H_q := -sum q_i log q_i`. Lower `H_q` (peaky/coherent set) → support.
    Lens: `e_out := (H_q_ref - H_q)/Unit`, `e_in := policy_risk/Unit`.

Note. Pick `Unit` and gain `c` so typical `|c*e|` sits in `[0.3, 1.2]` to avoid saturation; defaults: `c = 1.0`, `Unit = 1.0`.

H3) How to fuse them fairly (bounded & order-invariant)

- Convert each method's output to a signed contrast `e`, then to `a := tanh(c*e)` (or two-channel).
- Weight each signal by a declared `w` (uniform `w := 1` or strength-aware `w := |m|^gamma, gamma = 1`).
- Fuse: `U += w * atanh(a)`, `W += w`, `a_pool := tanh( U / max(W, eps_w) )`.

- For decision: compute `RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`, optionally gate: `RSI_env := g_t * RSI`.
- Parity holds: value path still uses `m` only, `phi((m,a)) = m`.

H4) Pros/cons by method (engineering quick view)

- **Entropy / margins**
  - Simple, cheap; ubiquitous.
  - Scale-sensitive across vendors; over-confident models can mislead.
    Use via lens: normalize to dimensionless `e`; band on `RSI`, not raw probs.
- **MC-dropout**
  - Captures epistemic uncertainty cheaply (no re-training).
  - Extra passes; variance depends on dropout config.
    Use via lens: treat variance/entropy as `e_in`; fuse order-invariantly.
- **Deep ensembles**
  - Strong uncertainty; robust in practice.
  - Costly; heavy to serve.
    Use via lane: pool per-model alignments in u-space; cross-vendor parity via same manifest.
- **Conformal**
  - Finite-sample coverage guarantees (under exchangeability).
  - Set size can be coarse; needs calibration and drift care.
    Use via lens: penalize large set size or high nonconformity.
- **Evidential**
  - Single forward pass; separates aleatoric/epistemic proxies.
  - Sensitive to training losses; domain tuning needed.
    Use via lane: map evidence strength to `e_out`, penalties to `e_in`.
- **Calibrated prob**
  - Better-behaved than raw softmax; easy to deploy.
  - Calibration drift over time; domain shift hurts.
    Use via lane: treat calibrated top-1 as support; bands smooth drift.

H5) Worked minis (calculator-fast)

1. **Entropy lens (K=5).**
   `H = 0.900000`, `H_max = log(5) ≈ 1.609438`.
   `conf_H = 1 - 0.900000/1.609438 ≈ 0.440383`. With `tau = 0.300000`, `Unit = 1.000000`, `c = 1.000000`:
   `e = 0.140383`, `a = tanh(0.140383) ≈ 0.139474`.
   Pool with another cue `a2 = tanh(0.300000) = 0.291313`:
   `U = 0.140383 + 0.300000 = 0.440383`, `W = 2`, `a_pool = tanh(0.220191) ≈ 0.216858`.
2. **MC-dropout variance penalty.**
   `var_mc = 0.250000`, target `s = 0.100000`, `Unit = 0.200000`, `c = 1.000000`:
   `e = (0.100000 - 0.250000)/0.200000 = -0.750000`, `a = tanh(-0.750000) ≈ -0.635148`.

3. **Ensemble margin (3 models).**
   Margins: `0.600000, 0.400000, 0.200000` → map to `a_k = tanh(margin - 0.300000)`.
   `a = [0.291313, 0.099668, -0.099668]`.
   `U = 0.300000 + 0.100000 - 0.100000 = 0.300000`, `W = 3`, `a_pool = tanh(0.100000) = 0.099668`.
4. **Conformal set size penalty.**
   `|S| = 3`, penalize sizes above `1` with `Unit = 2`:
   `e_in = (3 - 1)/2 = 1.000000`, `a_in = tanh(-1.000000) = -0.761594`.
   With supportive cue `a_out = tanh(0.800000) = 0.664037` and `w = 1` both sides:
   `U_in = 1.000000`, `V_out = 0.800000`, `W_in = 1` ⇒ `RSI = tanh( (0.800000 - 1.000000)/1 ) = tanh(-0.200000) = -0.197375` → **band A-**.

H6) How to combine multiple methods cleanly (no double counting)

- Declare per-surface bundles in the manifest, e.g., `bundle_decode = {entropy, margin, calibrated_prob}`.
- De-correlate by giving correlated cues smaller `w` or by forming a single contrast `e := w1*z1 + w2*z2 - w3*z3` before mapping to `a`.
- Keep it publishable: log each term and weight; no hidden multipliers.
- Acceptance: perturbing one cue must not increase alignment if it is a penalty (sign sanity); shuffle the order and confirm identical `RSI`.

H7) Micro experiments you can run in a day (observation-only)

1. **Decode bake-off (bounded chooser vs. raw entropy).**
   For 1k prompts across 2 vendors:

- Compute `RSI` from lenses `{entropy, margin}` with `c=1`, `Unit=1`, `w := 1`.
- Rank by `RSI_env := g_t * RSI` (`g_t = 1` first).
- Compare top-1 accuracy vs. ranking by `-entropy`.
- Report: `RSI_pool`, band histograms, accuracy lift.

2. **RAG drift guard (MC-dropout + retrieval entropy).**
   `e_in := var_mc/Unit + lambda * H_q/Unit`.
   `e_out := citation_hit + semantic_gain`.
   Track clicks or judge scores vs. `RSI_env`. Expect fewer low-band selections when drift spikes.
3. **Conformal set size as a penalty lens.**
   Penalize large `|S|`; monitor change in `A-/A--` fractions.
   Keep `m` unchanged; decisions stamped.
   All experiments: shuffle inputs and confirm `batch == stream == shuffled` under the same manifest.

H8) When to prefer which cue (rule-of-thumb table)

scenario	good primary cue	good secondary cue
shortlist classification	margin or calibrated p	entropy (low), conformal size
long-tail classification	conformal score/size	MC-dropout entropy
RAG answerability	retrieval set entropy	doc citation_hit, MC var
tool success routing	calibrated success prob	MC var of schema match
regression (numeric)	MC var (low)	ensemble var (low), EDL strength (high)

(Always map cues to a single lane via `a := tanh(c*e)` and decide by `RSI` or `RSI_env`.)

H9) Acceptance checklist (must pass)

- Parity: classical values `m` never change (`phi((m,a)) = m`).
- Lens sanity: increasing a penalty lowers `a` (or raises `|a_in|`); increasing support raises `a`.
- Order/shard invariance: permuting cues or merging shards yields the same `RSI`.
- Boundedness: all `a`, `RSI`, `RSI_env` in `(-1,+1)`; bands map per manifest.
- Determinism: same manifest, same inputs ⇒ same outputs and stamps.

One-line takeaway. Treat entropy, margins, MC-dropout, ensembles, conformal, and evidential signals as simple lenses that feed the same bounded, order-invariant lane. You pick by `RSI` (or `RSI_env`) and keep all classical numbers untouched via `phi((m,a)) = m`.

Stamp example (append to any experiment summary line).

```
SSMCLOCK1|iso_utc|svc=decode|exp=H_bundles|U=0.440383|W=2.000000|RSI=0.2168
58|g=1.00|RSI_env=0.216858|band=A0|manifest=knobs_hash
```

Appendix I — Lens Builder (derive Unit, c, and weights from logs)

Purpose. Turn raw, observable signals into a robust lens with minimal tuning. All math is observation-only: `phi((m,a)) = m`, and the lane acts on alignment only. Mapping is always `a_in := tanh(-c*e_in)`, `a_out := tanh(+c*e_out)` and fusion is order-invariant: `U += w*atanh(a)`, `W += w`, `a_out := tanh(U / max(W, eps_w))`. Defaults: `eps_a = 1e-6`, `eps_w = 1e-12`, `w := |m|^gamma` with `gamma = 1` (or `w := 1`).

I1) Inputs & outputs (what you start with, what you ship)

- You have (from logs): per-item positive cues `P_i`, penalties `N_j`, any magnitude `m`, and (optionally) labels or judge scores for spot checks.
- You must produce (to manifest): `Unit > 0`, `c > 0`, weight policy `w := |m|^gamma` (or `w := 1`), optional gate preset `g_t`, and the band table (defaults are fine).
- **Lens skeleton (repeatable):**
  `e := ( sum_i alpha_i * P_i - sum_j beta_j * N_j ) / Unit`
  `a := tanh( c * e )` *(or two-channel with `e_out`, `e_in`)*

I2) 30-Minute recipe (deterministic, copy-paste)

Step 1 — Normalize raw terms (dimensionless).
For each `P_i` and `N_j`, pick a scaler `S` and set `P_i := P_i / max(S, eps)`, `N_j := N_j / max(S, eps)` so typical values lie in `[0,1]`. Good `S` choices: historical `p95`, `mean + 2*std`, or a target (e.g., `SLO`).

Step 2 — Choose Unit so typical `|e|` is in `[0.2, 1.5]`.
Quick rule: let `z_pos := p95( sum_i alpha_i * P_i )`, `z_neg := p95( sum_j beta_j * N_j )`.
Set `Unit := max( z_pos, z_neg, 1e-12 )`.
Single-channel: `Unit := p95( | sum_i alpha_i*P_i - sum_j beta_j*N_j | )`.

Step 3 — Pick `c` from a target response.
Choose a percentile `e* := p95(|e|)` on a warmup slice, decide the alignment you want there, e.g., `t* := 0.90`.
Solve once: `c := atanh(t*) / max(e*, 1e-12)`.
Common picks: `t* = 0.90` (maps `p95` to `A++` boundary) or `t* = 0.60` (maps `p95` to `A+` boundary).

Step 4 — Select weight policy `w`.

- Pure comparability: `w := 1`.
- Strength-aware: `w := |m|^gamma`, start with `gamma = 1`.
  Declare once; keep fixed across vendors.

Step 5 — Dry run and band sanity.
Compute `a := tanh(c*e)` on a held-out slice, then `RSI := tanh( mean_u / 1 )` in single-channel *(or the two-channel chooser)*.
Check band histogram spreads across `A0` and `A+` with few `A--/A++` unless the task is trivial.

Step 6 — Freeze and stamp.
Record `Unit, c, weights, eps_a, eps_w` and a `knobs_hash`. From now on, runs are comparable.

I3) Worked micro example (calculator-fast, single-channel)

Warmup slice (1k items) gives `p95(|e_raw|) = 0.780000` after Step 1–2.
Choose `t* = 0.60` at `p95`: `c = atanh(0.60) / 0.780000 ≈ 0.693147 / 0.780000 ≈ 0.888650`.
Take uniform weights `w := 1`.

- For an item with `e = 0.500000`:
  `a = tanh( 0.888650 * 0.500000 ) = tanh(0.444325) ≈ 0.417223` → **band A0**.
- For `e = 1.100000`:
  `a = tanh( 0.888650 * 1.100000 ) = tanh(0.977515) ≈ 0.751988` → **band A+**.

(All minis to 6 decimals.)

I4) Two-channel quick builder (support vs penalty)

Compute `e_out := p := sum_i alpha_i * P_i`, `e_in := n := sum_j beta_j * N_j`.
Set `Unit_out := p95(p)`, `Unit_in := p95(n)`.
Use `a_out := tanh( c * p / max(Unit_out, 1e-12) )`, `a_in := tanh( -c * n / max(Unit_in, 1e-12) )` with the same `c` from I2–I3.
Chooser: `RSI := tanh( (sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w) )`.

I5) Robust stats option (when tails are ugly)

If distributions are heavy-tailed, use median and MAD:
`Unit := 2.5 * MAD( e_raw )` where `MAD(x) := median( |x - median(x)| )`.
This centers typical `|c*e|` near `~1` for many tasks when `c ≈ 1`.

I6) Micro-grid for Unit and c (5×5, five minutes)

Pick `Unit in {p75, p85, p90, p95, p98}` of `|e_raw|`.
For each `Unit`, set `c := atanh(0.60) / p95(|e|)` and also try `atanh(0.90) / p95(|e|)`.
Score each grid point by:
(a) separation in band shares *(more `A+` on correct, more `A-` on known bad)*,
(b) stability across shuffles *(order-invariant by design)*, and
(c) downstream KPI correlation *(optional, observation-only)*.
Pick the simplest that passes acceptance.

I7) Auto-suggest `gamma` in `w := |m|^gamma` (one pass)

- Grid `gamma in {0.0, 0.5, 1.0, 1.5, 2.0}`.
- For each `gamma`, compute weekly `RSI_port := tanh( sum w*atanh(a) / max(sum w, eps_w) )` across stable slices.
- Score by (i) **stability**: minimize `stdev(RSI_port)` across slices, (ii) **fairness**: minimize `|RSI_port_segment − RSI_port_all|` weighted by traffic, (iii) **parsimony**: prefer the smallest `gamma` within 1% of the best score.
- Choose that `gamma`, then freeze.

I8) Quick metrics to validate `c` (saturation/dead-zone)

- **Saturation rate:** `sat := mean( |c*e| > 3.0 )` *(targets `|a| > ~0.995`)*. Require `sat < 0.10`.
- **Dead-zone rate:** `dead := mean( |c*e| < 0.10 )` *(i.e., `|a| < tanh(0.10) ≈ 0.099668`)*. Require `dead < 0.70`.
- If either fails, adjust `Unit` or `c` and re-run the micro-grid.

I9) Pseudocode (drop-in, copy-paste-ready)

```
# Helpers (percentiles on absolute values)
def percentile_abs(vals, p):
    xs = sorted(abs(v) for v in vals)
    i = int(max(0, min(len(xs)-1, round((p/100.0)*(len(xs)-1)))))
    return xs[i]

def atanh(x):  # clamp to keep |x|<1
    from math import log
    x = max(-0.999999, min(0.999999, x))
    return 0.5 * log((1 + x) / (1 - x))

def tanh(x):
    from math import tanh as _t
    return _t(x)

# Unit and c
def choose_unit(e_vals):
    return percentile_abs(e_vals, 95)  # p95 of |e|

def choose_c(e_vals, target=0.60):
    e95 = max(percentile_abs(e_vals, 95), 1e-12)
    return atanh(target) / e95

# Map (single-channel)
def map_alignment(e, Unit, c):
    return tanh(c * (e / max(Unit, 1e-12)))

# Two-channel
def two_channel(e_out, e_in, Unit_out, Unit_in, c):
    a_out = tanh(c * (e_out / max(Unit_out, 1e-12)))
    a_in  = tanh(-c * (e_in  / max(Unit_in, 1e-12)))
```

```
    return a_in, a_out

# Gamma auto-scan
def pick_gamma(m_vals, a_vals, weeks, gammas=(0.0,0.5,1.0,1.5,2.0)):
    def rsi_port(rows):
        U = W = 0.0
        for r in rows:
            x = max(-0.999999, min(0.999999, r["a"]))
            U += r["w"] * atanh(x)
            W += r["w"]
        return 0.0 if W <= 0 else tanh(U / max(W, 1e-12))

    best = (None, 1e9)
    for g in gammas:
        ports = []
        for wk in weeks:
            rows = [{"a": a_vals[i], "w": (abs(m_vals[i])**g if g>0 else 
1.0)}
                    for i in weeks[wk]]
            ports.append(rsi_port(rows))
        mean = sum(ports)/len(ports)
        var  = sum((x-mean)**2 for x in ports)/len(ports)
        score = var**0.5
        if score < best[1]: best = (g, score)
    return best[0]
```

I10) Tiny "calibrate & stamp" script (prints Unit, c, gamma, stamp)

```
# Inputs: arrays e_raw[], m[], and an example RSI to preview bands
def calibrate_and_stamp(e_raw, m, target=0.60, iso_utc="2025-10-
27T00:00:00Z"):
    Unit = choose_unit(e_raw)
    c    = choose_c(e_raw, target=target)
    # Build a toy week index for gamma scan (all in one bucket if unknown)
    weeks = {"w0": list(range(len(e_raw)))}
    # Map to alignments for scan
    a_vals = [map_alignment(e, Unit, c) for e in e_raw]
    gamma  = pick_gamma(m_vals=m, a_vals=a_vals, weeks=weeks)
    # Preview one RSI from the slice
    from math import atan as _atan  # not used; keep tanh/atanh above
    U = W = 0.0
    for a in a_vals:
        U += atanh(max(-0.999999, min(0.999999, a)))
        W += 1.0
    RSI = (0.0 if W <= 0 else tanh(U / max(W, 1e-12)))
    band = ("A++" if RSI>=0.90 else "A+" if RSI>=0.60 else
            "A0" if RSI>-0.60 else "A-" if RSI>-0.90 else "A--")
    stamp = (
      "SSMCLOCK1|{iso}|SSMLENS|Unit={Unit:.6f}|c={c:.6f}|gamma={g:.2f}"
      "|RSI={rsi:.6f}|band={band}|manifest=knobs_hash"
    ).format(iso=iso_utc, Unit=Unit, c=c, g=gamma, rsi=RSI, band=band)
    print("Unit={:.6f}  c={:.6f}  gamma={:.2f}".format(Unit, c, gamma))
    print(stamp)
```

Stamp example (from the script).

```
SSMCLOCK1|2025-10-
27T00:00:00Z|SSMLENS|Unit=0.780000|c=0.888650|gamma=1.00|RSI=0.604368|band=
A+|manifest=knobs_hash
```

I11) Manifest snippet (publish once)

```
"lens_decode_v1": {
  "Unit": 0.780000,
  "c": 0.888650,
  "weights": {"policy": "uniform"},            # or
{"policy":"abs_m_gamma","gamma":1.0}
  "eps_a": 1e-6, "eps_w": 1e-12,
  "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00}
}
```

I12) Acceptance checklist (must pass)

- **Dimensionless e.** Every term scaled; `Unit > 0` declared.
- **Boundedness.** For all items, `|a| < 1`; downstream `|RSI| < 1`.
- **Monotone signs.** Increasing any penalty lowers alignment; increasing support raises alignment.
- **Saturation / dead-zone.** `sat < 0.10`, `dead < 0.70` at the chosen `Unit, c`.
- **Shuffle invariance.** Lens outputs and `RSI` identical under permutations.
- **Determinism.** Same manifest and inputs ⇒ same outputs and stamps.
- **Parity.** Classical numbers remain identical, always `phi((m,a)) = m`.

One-line takeaway. Pick `Unit` from a robust percentile, set `c := atanh(t*) / e*` to land your preferred band at `p95`, declare `w` once, and you have a portable, bounded lens that fuses order-invariantly while `phi((m,a)) = m` keeps your numbers pristine.

Appendix J — SDK Packaging & Golden Tests (PyPI-ready, CI, reproducibility)

Purpose. Ship a tiny, production-ready SDK that implements the lane kernel, chooser, gate, bands, path scoring, and stamps so teams can adopt SSM-AI in minutes. All math is observation-only and bounded; classical values remain unchanged: `phi((m,a)) = m`. Fusion is order-invariant via `u := atanh(a)`, `U += w*u`, `W += w`, `a_out := tanh(U / max(W, eps_w))`. Defaults: `eps_a = 1e-6`, `eps_w = 1e-12` (use `eps_w >= 1e-8` for float32), weights `w := |m|^gamma` with `gamma = 1` (or `w := 1`), gate modes `"mul"` and `"u_scale"`.

J1) Minimal package layout (drop-in)

```
ssm_ai/
  __init__.py
  lane.py          # two-lane numeral, clamp, tanh/atanh, lane mul/div (M2)
  fuse.py          # order-invariant streaming fuse (U/W), shard merge
  lens.py          # contrasts e, mapping a := tanh(c*e), two-channel
helpers
  rsi.py           # chooser: RSI := tanh((V_out - U_in)/max(W_in, eps_w))
  gate.py          # calm gate: g_t in [0,1], modes "mul" and "u_scale"
  bands.py         # A++/A+/A0/A-/A--, hysteresis helpers
  path.py          # step scores, priors in u-space, path pooling, rollback
  stamp.py         # SSM-Clock stamp strings, knobs_hash utilities
  manifest.py      # schema load/validate, deterministic defaults
  audit.py         # CSV/JSONL emit: (m,a,U,W,RSI,RSI_env,band,stamp)
  hw/
    fx_specs.py    # SSMH fixed-point formats, LUT generation (tanh/atanh)
  examples/
    decoding_demo.py
    rag_demo.py
    tools_demo.py
  tests/
    data/golden_vectors.csv
    test_lane.py
    test_fuse.py
    test_rsi.py
    test_gate.py
    test_path.py
    test_bands.py
```

J2) Public API (concise)

```
# lane.py
class LaneValue:
    def __init__(self, m: float, a: float, eps_a: float = 1e-6):
        self.m = m
        self.a = clamp_align(a, eps_a)
    def collapse(self) -> float:                # phi((m,a)) = m
        return self.m

def clamp_align(a: float, eps_a: float = 1e-6) -> float:
    return max(-1 + eps_a, min(1 - eps_a, a))

def lane_mul(a1: float, a2: float, eps_a: float = 1e-6) -> float:
    from math import atanh, tanh
    a1 = clamp_align(a1, eps_a); a2 = clamp_align(a2, eps_a)
    return tanh(atanh(a1) + atanh(a2))          # M2

def lane_div(a1: float, a2: float, eps_a: float = 1e-6) -> float:
    from math import atanh, tanh
    a1 = clamp_align(a1, eps_a); a2 = clamp_align(a2, eps_a)
    return tanh(atanh(a1) - atanh(a2))          # M2

# fuse.py
class Fuse:
    def __init__(self, eps_a: float = 1e-6, eps_w: float = 1e-12):
        self.U = 0.0; self.W = 0.0; self.eps_a = eps_a; self.eps_w = eps_w
    def add(self, a: float, w: float = 1.0) -> None:
        from math import atanh
```

```
        a = max(-1 + self.eps_a, min(1 - self.eps_a, a))
        self.U += w * atanh(a); self.W += w
    def merge(self, other: "Fuse") -> None:
        self.U += other.U; self.W += other.W
    def value(self) -> float:
        from math import tanh
        return tanh(self.U / max(self.W, self.eps_w))

# lens.py
def lens_contrast(P: list[tuple[float,float]],
                  N: list[tuple[float,float]],
                  Unit: float = 1.0) -> float:
    num = sum(a*v for a, v in P) - sum(b*v for b, v in N)
    return num / max(Unit, 1e-12)

def align_single(e: float, c: float = 1.0) -> float:
    from math import tanh
    return tanh(c * e)

def map_to_alignment(e_in: float, e_out: float, c: float = 1.0) ->
tuple[float,float]:
    from math import tanh
    return (tanh(-c * e_in), tanh(+c * e_out))

# rsi.py
def rsi_from_alignments(a_in_items: list[tuple[float,float]],
                        a_out_items: list[tuple[float,float]],
                        eps_w: float = 1e-12, eps_a: float = 1e-6) ->
float:
    from math import atanh, tanh
    U_in = V_out = W = 0.0
    for a, w in a_in_items:
        a = max(-1 + eps_a, min(1 - eps_a, a)); U_in += w * atanh(a); W +=
w
    for a, w in a_out_items:
        a = max(-1 + eps_a, min(1 - eps_a, a)); V_out += w * atanh(a)
    return 0.0 if W <= 0 else tanh((V_out - U_in) / max(W, eps_w))

# gate.py
def gate_value(F: float, D: float, L: float, E: float, V: float, Q: float =
0.0,
               w: tuple[float,...] = (1,1,1,1,1,0), s_thr: float | None =
None,
               rho: float = 0.2, state: dict | None = None,
               eps_g: float = 1e-12, g_min: float = 0.0) -> dict:
    g_prev = 1.0 if state is None else state.get("g", 1.0)
    W = sum(w); mix = (w[0]*F + w[1]*D + w[2]*L + w[3]*E + w[4]*V + w[5]*Q)
/ max(W, eps_g)
    g_inst = max(0.0, min(1.0, 1.0 - mix))
    if s_thr is not None:
        sev = max(F, D, E)
        if sev > s_thr:
            g_sev = max(0.0, min(1.0, 1.0 - (sev - s_thr) / max(1.0 -
s_thr, eps_g)))
            g_inst = min(g_inst, g_sev)
    g = max(g_min, (1 - rho) * g_prev + rho * g_inst)
    out = {"g": max(0.0, min(1.0, g))}
    if state is not None: state["g"] = out["g"]
    return out
```

```
def apply_gate(RSI: float, g: float, mode: str = "mul", eps_a: float = 1e-
6) -> float:
    from math import atanh, tanh
    x = max(-1 + eps_a, min(1 - eps_a, RSI))
    return g * x if mode == "mul" else tanh(g * atanh(x))

# bands.py
def to_band(x: float) -> str:
    return "A++" if x >= 0.90 else "A+" if x >= 0.60 else "A0" if x > -0.60
else "A-" if x > -0.90 else "A--"

def to_band_hysteresis(x: float, prev_band: str | None, h_up: float = 0.02,
h_dn: float = 0.02) -> str:
    if prev_band is None: return to_band(x)
    if prev_band == "A++" and x >= 0.90 - h_dn: return "A++"
    if prev_band == "A+"  and (0.60 - h_dn) <= x < (0.90 + h_up): return
"A+"
    if prev_band == "A0"  and (-0.60 - h_dn) < x < (0.60 + h_up): return
"A0"
    if prev_band == "A-"  and (-0.90 - h_dn) < x <= (-0.60 + h_up): return
"A-"
    if prev_band == "A--" and x <= -0.90 + h_up: return "A--"
    return to_band(x)

# path.py
class PathScore:
    def __init__(self, eps_a: float = 1e-6, eps_w: float = 1e-12):
        self.U = 0.0; self.W = 0.0; self.stack = []; self.eps_a = eps_a;
self.eps_w = eps_w
    def add_step(self, RSI_used: float, beta: float = 0.0, b_s: float =
0.0, w: float = 1.0) -> None:
        from math import atanh
        x = max(-1 + self.eps_a, min(1 - self.eps_a, RSI_used))
        u_step = atanh(x) + beta * b_s
        self.U += w * u_step; self.W += w; self.stack.append((w * u_step,
w))
    def undo_step(self) -> None:
        if not self.stack: return
        dU, dW = self.stack.pop(); self.U -= dU; self.W -= dW
    def rsi_path(self) -> float:
        from math import tanh
        return tanh(self.U / max(self.W, self.eps_w))

# stamp.py
def make_stamp(iso_utc: str, rasi_idx: str, theta_deg: str, digest: str,
chain_k: str) -> str:
    return f"SSMCLOCK1|{iso_utc}|{rasi_idx}|{theta_deg}|{digest}|{chain_k}"

def knobs_hash(manifest_obj: dict) -> str:
    import hashlib, json
    canon = json.dumps(manifest_obj, separators=(',', ':'), sort_keys=True)
    return hashlib.sha256(canon.encode("utf-8")).hexdigest()

# manifest.py
def load_manifest(path_or_obj):
    import json
    return json.loads(path_or_obj) if isinstance(path_or_obj, str) and
path_or_obj.strip().startswith('{') else path_or_obj

def validate_manifest(obj: dict) -> None:
    assert "eps_a" in obj and "eps_w" in obj and "bands" in obj
```

J3) Manifest schema (copy-paste)

```
{
  "dtype": "float64",
  "eps_a": 1e-6,
  "eps_w": 1e-12,
  "weights": {"policy": "uniform"},              // or
{"policy":"abs_m_gamma","gamma":1.0}
  "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00,
            "hysteresis":{"h_up":0.02,"h_dn":0.02},
            "band_on":"RSI_env"},
  "gate": {"mode":"mul","weights":{"F":1,"D":1,"L":1,"E":1,"V":1,"Q":0},
           "rho":0.20,"g_min":0.00,"eps_g":1e-12,
           "safety_notch":{"enabled":false,"s_thr":0.80}},
  "division_policy": "strict",
  "lens_presets": {
    "decode_v1": {"Unit":1.0,"c":1.0,"terms":"inline formula"},
    "rag_v1":    {"Unit":1.0,"c":1.0}
  }
}
```

J4) Golden vectors (calculator-fast, deterministic)

Use `float64` for reference. Each case asserts boundedness and exact identities within tolerance.

```
# V1 — clamp + round-trip
a_in = 0.9999999; eps_a = 1e-6
a_c  = 0.999999
# u := atanh(a_c) finite; tanh(u) ≈ 0.999999 within tolerance

# V2 — fuse invariance
a1 = tanh(0.2) = 0.197375
a2 = tanh(0.4) = 0.379949
U = 0.2 + 0.4 = 0.6; W = 2
a_out = tanh(U/W) = tanh(0.3) = 0.291313  # swap/shard -> same

# V3 — lane mul/div (M2)
a1 = tanh(0.5) = 0.462117; a2 = tanh(0.2) = 0.197375
a_mul = tanh(0.5 + 0.2) = 0.604368
a_div = tanh(0.5 - 0.2) = 0.291313

# V4 — chooser
U_in = -0.2; V_out = +0.5; W_in = 1
RSI = tanh((0.5 - (-0.2))/1) = tanh(0.7) = 0.604368

# V5 — gate modes
RSI = 0.700000; g = 0.81
mul     : RSI_env = 0.567000
u_scale : RSI_env = tanh(0.81 * atanh(0.70)) ≈ 0.605961

# V6 — path pooling
u_steps = [0.586900, 0.400000, 0.200000]; w = 1
U = 1.186900; W = 3
RSI_path = tanh(U/W) = tanh(0.395633) ≈ 0.375900
```

Tolerances. `float64: 1e-12`; `float32: 1e-5`. **Band outcomes must be identical across dtypes.**

J5) CI pipeline (4 stages, fast)

1. **Static checks.** Type hints and style.
2. **Unit tests.** Golden vectors and randomized invariance:
   • Shuffle test: permute inputs; assert `a_out` unchanged.
   • Shard test: split into `K` shards; sum `(U,W)`; assert equality.
   • Gate purity: `phi((m,a))` unchanged before/after gating.
   • Division policy: under `"strict"`, flag near-zero denominators for lane-only math; never mutate `m`.
3. **Determinism pack.** Serialize a manifest, compute `knobs_hash := sha256(canonical_json(manifest))`, run a short scenario, and assert logs reproduce bit-for-bit with the same `knobs_hash`.
4. **Wheel build + checksum.** Build sdist/wheel, compute SHA256, emit to artifacts alongside the golden CSV.

J6) Quickstart (10 lines)

```
from ssm_ai.lens import lens_contrast, align_single
from ssm_ai.fuse import Fuse
from ssm_ai.rsi import rsi_from_alignments
from ssm_ai.gate import gate_value, apply_gate

# 1) lens -> alignment
e = lens_contrast(P=[(1.0, 0.9), (0.5, 0.6)], N=[(0.8, 0.2)], Unit=1.0)
a = align_single(e, c=1.0)

# 2) fuse multiple cues (order-invariant)
fz = Fuse(); fz.add(a, w=1.0); fz.add(align_single(0.3), w=1.0)
a_pool = fz.value()

# 3) chooser + optional gate
rsi  = rsi_from_alignments(a_in_items=[], a_out_items=[(a_pool, 1.0)])
gate = gate_value(F=0.2, D=0.1, L=0.3, E=0.15, V=0.2)
rsi_env = apply_gate(rsi, gate["g"], mode="mul")
```

J7) Example: beam pick (12 lines)

```
from math import atanh, tanh

def rsi_candidate(items, c=1.0, eps_w=1e-12, eps_a=1e-6):
    U_in = V_out = W = 0.0
    for (e_in, e_out, w) in items:
        a_in  = tanh(-c*e_in); a_out = tanh(+c*e_out)
        a_in  = max(-1+eps_a, min(1-eps_a, a_in))
        a_out = max(-1+eps_a, min(1-eps_a, a_out))
        U_in  += w * atanh(a_in); V_out += w * atanh(a_out); W += w
    return tanh((V_out - U_in) / max(W, eps_w)) if W > 0 else 0.0
```

```
def pick_beam(beam, g_t=1.0):
    scored = [(cid, g_t * rsi_candidate(items)) for (cid, items) in beam]
    return max(scored, key=lambda kv: kv[1])[0]
```

J8) Logging schema (CSV/JSONL, replay-ready)

```
iso_utc, svc, knobs_hash, dtype, m, a, U, W, RSI, g_t, RSI_env, band,
division_policy, note
```

Replay invariant. `a_out == tanh( sum(U) / max(sum(W), eps_w) )` and `phi((m,a)) = m`. **Zero-evidence guard:** if `W_in == 0`, then `RSI := 0`, `band := "A0"`, reason `insufficient_evidence`.

J9) Security & privacy defaults

• **PII-avoidance.** Lenses operate on aggregates or declared metrics; no raw PII.
• **Masking.** Redact free-text fields before logging; keep `m, a, U, W, RSI, g_t, band, stamp`.
• **Scopes.** Package exports alignment utilities only; it never edits `m`.

J10) Acceptance checklist (must pass)

• **Parity.** All SDK functions preserve `phi((m,a)) = m`.
• **Order/shard invariance.** Fuse equality across permutations and shard merges.
• **Boundedness.** All `a`, `RSI`, `RSI_env` strictly in `(-1,+1)`.
• **Determinism.** Same manifest + inputs ⇒ same outputs, bands, stamps.
• **Division policy.** `"strict"` honored; alternatives require explicit declaration.
• **Golden vectors.** Pass within tolerance across Python versions and dtypes.

One-line takeaway. A tiny SDK plus golden tests make SSM-AI adoption boring in the best way. Same bounded math, same stamps, same results everywhere; your classical numbers stay pristine under `phi((m,a)) = m`.

Stamp example (append to any CI summary line).

```
SSMCLOCK1|iso_utc|svc=sdk_ci|U=0.600000|W=2.000000|a_out=0.291313|RSI=0.604
368|g=0.81|RSI_env=0.489538|band=A0|manifest=knobs_hash
```

Appendix K — SSM-Search — Symbolic Search (lane-native retrieval)

Purpose. Provide a drop-in symbolic search that emits a classical retrieval magnitude `m_retrieval` alongside a bounded alignment lane `a_search in (-1,+1)` and a chooser `RSI` for ranking. All math is observation-only: `phi((m,a)) = m`. Order-invariant streaming lets shards and online updates merge via `(U,W)` with `U += w*atanh(a)` and `a_out := tanh(U / max(W, eps_w))`.

K1) Index & Query Representation (SSMS grammar)

- **Tokens.** Use SSMS symbols (operators, numerals, tags) plus plain text. Store each posting with: `doc_id, position, kind, weight, time_stamp`.
- **Fields.** `title, body, code, meta, time`.
- **Classical retrieval magnitude (unchanged).** `m_retrieval := BM25F * boost_kind * decay_time` (or your existing ranker). SSM-Search never edits `m_retrieval`.
- **Lane carriers (side-car).** For each hit emit per-signal contrasts: `quality` (semantic/text match), `freshness` (recency vs window), `authority` (source/citations/trust), `risk_penalty` (toxicity/license/policy/spam), `coherence_penalty` (cross-field mismatch/contradiction).

K2) Lens → Alignment for Search (per result)

Single-equation form (dimensionless):

```
e := ( alpha*quality + beta*freshness + gamma*authority -
delta*risk_penalty - eta*coherence_penalty ) / Unit
a_search := tanh( c * e )
```

Two-channel form (explicit support/penalty):

```
e_out := ( alpha*quality + beta*freshness + gamma*authority ) / Unit_out
e_in := ( delta*risk_penalty + eta*coherence_penalty ) / Unit_in
a_out := tanh( +c * e_out )
a_in := tanh( -c * e_in )
```

Chooser (per result):

```
RSI := tanh( ( sum w*atanh(a_out) - sum w*atanh(a_in) ) / max( sum w, eps_w
) )
```

Defaults. `c = 1.0`, `eps_w = 1e-12`, weights `w := 1` or `w := |m_retrieval|^gamma` with `gamma = 1`.

Gate (optional, alignment-only):
`RSI_env := g_t * RSI` (or `RSI_env := tanh( g_t * atanh(RSI) )`) with `g_t in [0,1]` from live telemetry. Magnitudes never change: `phi((m,a)) = m`.

K3) Order-Invariant Ranker (shards, streams, online)

Maintain `(U,W)` per result list:

```
U += w * atanh(a_component)
W += w
a_pool := tanh( U / max(W, eps_w) )
```

- **Shards.** Each shard returns `(U_shard, W_shard)`; master sums and inverts once.
- **Paging/streaming.** Interleave postings or re-rank incrementally; results equal batch.

Final scoring & tie-breakers.

- **Primary:** `RSI_env` (or `RSI` if `g_t = 1`).
- **Secondary:** `m_retrieval` for ties (keeps classical semantics stable).
- **Band for UI/policy:** map `RSI_env` to `A++/A+/A0/A-/A--`.

K4) Worked Mini (calculator-fast)

Parameters: `alpha=1.0`, `beta=0.5`, `gamma=0.4`, `delta=0.8`, `eta=0.5`, `Unit=1.0`, `c=1.0`.

- **Result A features:** `quality=0.9`, `freshness=0.6`, `authority=0.5`, `risk=0.2`, `coherence_penalty=0.1`.
  `e = 1.0*0.9 + 0.5*0.6 + 0.4*0.5 - 0.8*0.2 - 0.5*0.1 = 1.19`
  `a_search = tanh(1.19) = 0.830579` → **band A+**.
- **Result B features:** `quality=0.8`, `freshness=0.2`, `authority=0.3`, `risk=0.5`, `coherence_penalty=0.1`.
  `e = 0.8 + 0.1 + 0.12 - 0.4 - 0.05 = 0.57`
  `a_search = tanh(0.57) = 0.515359` → **band A0**.

Ranking by `RSI` prefers A; `m_retrieval` remains unchanged.

K5) Time & Stamps (SSM-Clock synergy)

- **Per result stamp (one line):**
  `"SSMCLOCK1|iso_utc|k=search|U=...|W=...|RSI=...|g=...|RSI_env=...|band=...|knobs_hash"`
- **Roll-ups:** sum `(U,W)` per hour/day; compute `a_pool := tanh( sum(U) / max(sum(W), eps_w) )` for drift dashboards.

K6) API Sketch (drop-in)

```
def ssm_search(query, shards, lens_cfg, gate_state=None):
    # 1) classical retrieval from shards (unchanged):
    #    each shard returns: [(doc_id, m_retrieval, features_dict), ...]
    raw = []
    for shard in shards:
        raw.extend(shard.retrieve(query))  # classical ranker

    # 2) compute lens contrasts -> alignments -> RSI
    ranked = []
    g_t = 1.0 if gate_state is None else gate_state.get("g", 1.0)
    for (doc_id, m_ret, feat) in raw:
        # single-equation lens
        e = (lens_cfg["alpha"]*feat.get("quality", 0.0)
           + lens_cfg["beta"] *feat.get("freshness", 0.0)
           + lens_cfg["gamma"]*feat.get("authority", 0.0)
           - lens_cfg["delta"]*feat.get("risk_penalty", 0.0)
           - lens_cfg["eta"]  *feat.get("coherence_penalty", 0.0)) /
max(lens_cfg["Unit"], 1e-12)
        a = tanh(lens_cfg["c"] * e)
        U = atanh(max(-1+1e-6, min(1-1e-6, a))); W = 1.0
        RSI = tanh(U / max(W, 1e-12))
        RSI_env = g_t * RSI
        ranked.append({"doc_id": doc_id, "m": m_ret, "RSI": RSI, "RSI_env":
RSI_env})

    # 3) sort by RSI_env (then m_retrieval), band, and stamp
    ranked.sort(key=lambda r: (r["RSI_env"], r["m"]), reverse=True)
    return ranked
```

K7) Guardrails & Policies

- **Parity.** Never modify `m_retrieval` (`phi((m,a)) = m`).
- **Clamp.** Ensure `|a| < 1` with `eps_a = 1e-6`.
- **Visibility.** Log `m_retrieval`, `a_search` or `RSI`, `band`, `g_t`.
- **Search-safe defaults.** If features are missing, set them to `0`, compute `RSI := 0` (neutral), and include the item based on `m_retrieval` only.
- **Click/open policy.** Use `RSI_env` bands: open inline if `>= A+`, preview if `A0`, require extra confirmation if `<= A-`.

K8) Pseudocode (concise, two-channel)

```
def lens_search(feat, cfg):
    p = cfg["alpha"]*feat.get("quality", 0.0) \
      + cfg["beta"] *feat.get("freshness", 0.0) \
      + cfg["gamma"]*feat.get("authority", 0.0)
    n = cfg["delta"]*feat.get("risk_penalty", 0.0) \
      + cfg["eta"]  *feat.get("coherence_penalty", 0.0)
    a_out = tanh(cfg["c"] * p / max(cfg["Unit_out"], 1e-12))
    a_in  = tanh(-cfg["c"] * n / max(cfg["Unit_in"], 1e-12))
    return a_in, a_out

def rank_results(items, cfg, g_t=1.0, eps_w=1e-12, eps_a=1e-6):
```

```
    ranked = []
    for it in items:
        a_in, a_out = lens_search(it["feat"], cfg)
        a_in  = max(-1+eps_a, min(1-eps_a, a_in))
        a_out = max(-1+eps_a, min(1-eps_a, a_out))
        U_in  = atanh(a_in); V_out = atanh(a_out); W = 1.0
        RSI   = tanh((V_out - U_in)/max(W, eps_w))
        RSI_e = g_t * RSI
        ranked.append({"doc_id": it["doc_id"], "m": it["m"], "RSI": RSI,
"RSI_env": RSI_e})
    return sorted(ranked, key=lambda r: (r["RSI_env"], r["m"]),
reverse=True)
```

K9) Acceptance Checklist (must pass)

- **Order/shard invariance.** Same results if postings are shuffled or sharded.
- **Boundedness.** `a_search`, `RSI`, `RSI_env in (-1,+1)`; bands per manifest.
- **Parity.** `m_retrieval` identical to baseline.
- **Drift sanity.** Time roll-ups via `(U,W)` reproduce exact daily `a_pool`.
- **Determinism.** Same manifest ⇒ same ranking and stamps.

One-line takeaway. SSM-Search adds a bounded, order-invariant lane to your search stack: rank by `RSI` (or `RSI_env`) while keeping classical retrieval scores untouched (`phi((m,a)) = m`). It shards, streams, and stamps with zero semantic drift.

Stamp example (append to any search result line).

```
SSMCLOCK1|iso_utc|k=search|U=0.700000|W=1.000000|RSI=0.604368|g=0.80|RSI_en
v=0.483494|band=A0|knobs_hash=...
```

Appendix L — Governance Quick Reference (manifests-as-contract, privacy, escalation)

Purpose. Ship SSM-AI with a small, enforceable governance surface: manifests are the contract; logs are stamp-ready; routing is banded and auditable; privacy is by design. All controls act on alignment only (`RSI`, `RSI_env`) and never on classical values. Purity holds everywhere: `phi((m,a)) = m`. Defaults in force unless overridden: `eps_a = 1e-6`, `eps_w = 1e-12`, weights `w := |m|^gamma` with `gamma = 1`, gate modes `"mul"` and `"u_scale"`, bands unchanged.

L1) Principles (non-negotiable)

1. **Observation-only.** `phi((m,a)) = m` everywhere.
2. **Determinism.** Same manifest ⇒ same outputs/bands/stamps.
3. **Boundedness.** All lanes `|a| < 1`, all choosers `|RSI| < 1`, gated `|RSI_env| < 1`.
4. **Order invariance.** `U += w*atanh(a)`, `W += w`, `a_out := tanh(U / max(W, eps_w))` ⇒ `batch == stream == shuffled`.
5. **Privacy by design.** Lenses use declared, non-PII aggregates; telemetry for `g_t` is bounded in `[0,1]`, rate-limited, windowed.
6. **Fail-safe.** On any gate breach (collapse, clamp, determinism, division policy) ⇒ revert to declared classical logic (e.g., highest `m`) and stamp the cause.

L2) Roles & separation of concerns

- **Lens owner (product/ML).** Defines `e` terms, `Unit`, `c`, and weights `w`.
- **Governance owner (safety/privacy).** Approves lens inputs (no PII), gate lanes, band policy, retention.
- **Ops owner (platform).** Enforces manifest immutability per release; manages stamps, roll-ups, and keys.

L3) Manifest as contract (min keys to publish)

```
"manifest": {
  "eps_a": 1e-6, "eps_w": 1e-12,
  "weights": {"policy":"uniform"},            // or
{"policy":"abs_m_gamma","gamma":1.0}
  "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00,
            "hysteresis":{"h_up":0.02,"h_dn":0.02}},
  "gate": {"mode":"mul","rho":0.20,"g_min":0.00,"eps_g":1e-12,
           "weights":{"F":1,"D":1,"L":1,"E":1,"V":1,"Q":0},
           "safety_notch":{"enabled":false,"s_thr":0.80}},
  "division_policy": "strict",
  "lens": {
    "decode_v1": {"Unit":1.0,"c":1.0,"terms":"dimensionless aggregates
only"},
    "rag_v1":    {"Unit":1.0,"c":1.0}
  }
}
```

Compute and log a stable `knobs_hash := sha256(canonical_json(manifest))` for every run.

L4) Privacy posture (what's allowed / disallowed)

- **Allowed (examples).** Bounded rates, counters, latencies, binary flags, normalized error fractions, nonconformity scores, calibrated probabilities, set sizes, citation counts.
- **Disallowed (without explicit exception).** Raw user text, full prompts/responses, IDs, emails, phone numbers, exact geolocation, free-form telemetry that can leak content.
- **Aggregation rule.** Any lane feeding `g_t` must be a windowed statistic over `>= N` events or `time >= T` (e.g., `N >= 50` or `T >= 5 min`), clamped to `[0,1]`.
- **Masking.** If a note must be logged, use structured tags, not free text.

L5) Escalation matrix (bands → actions)

```
band   action (default)                              m-path
A++    promote/route; allow retry/escalation         untouched
A+     proceed; allow tool call; soft audit off path  untouched
A0     pause or defer; require secondary signal      untouched
A-     quarantine; human review queue                untouched
A--    block; raise alert                            untouched
```

Gate aware: apply decisions on `RSI_env := g_t * RSI` (or `tanh(g_t*atanh(RSI))`). Never edit `m`.

L6) Acceptance gates (run on every deploy)

1. **Collapse parity.** For all items, `phi((m,a)) == m`.
2. **Clamp bounds.** For all inputs, `|clamp(a)| < 1` with `eps_a`.
3. **Order invariance.** Shuffle test passes for fuse and chooser.
4. **Division policy.** Under `"strict"`, near-zero denominators are flagged; routing falls back to classical.
5. **Determinism.** Replay a stamped CSV reproduces `(U,W) -> a_out`, `RSI`, bands within dtype tolerance.
6. **Gate purity.** `RSI_env` changes with `g_t`; `m` does not.
7. **Privacy audit.** Lens inputs contain no PII; telemetry windows and clamps enforced.

L7) DPIA / risk mini-template (copy-paste)

```
system: <service/component>
purpose: publish bounded alignment for selection/routing/audit
data_in: dimensionless aggregates; no PII
lawful_basis: legitimate interests (quality/safety measurement)
pii_controls: none collected; masking on notes; retention 30d
risks: leakage via free-text logs; overfitting lenses to sensitive
attributes
mitigations: structured logs only; manifest review; periodic drift checks
residual_risk: low
sign-off: <names/date>  knobs_hash=<sha256>
```

L8) Retention & rotation (defaults)

- **Hot logs.** 30 days (rolling). Columns: `iso_utc, svc, knobs_hash, m, a, U, W, RSI, g_t, RSI_env, band, stamp`.
- **Cold roll-ups.** 12 months, hourly `(U,W)` only for `RSI` trends; no free text.
- **Deletion.** Cryptographic erasure of stamp chains older than policy window.

L9) Enforcement hooks (reference pseudocode)

```
def enforce_governance(item, manifest, state):
    # 1) privacy guard
    assert item["features_are_aggregates"] and not item.get("free_text")

    # 2) clamp + fuse (order-invariant)
    a = clamp_align(item["a"], eps_a=manifest["eps_a"])
    state["U"] += item["w"] * atanh(a)
    state["W"] += item["w"]

    # 3) chooser + gate (alignment-only)
    RSI = tanh((state["V_out"] - state["U_in"]) / max(state["W_in"],
manifest["eps_w"]))
    RSI_env = apply_gate(RSI, state["g"], mode=manifest["gate"]["mode"])

    # 4) band policy
    band = to_band(RSI_env)
    if band in ["A--","A-"]:
        return fallback_classical(item["m"])  # m untouched
    return {"m": item["m"], "RSI_env": RSI_env, "band": band}
```

L10) Red-team prompts (quick probes to run weekly)

1. **Order trick.** Randomize item order 10× ⇒ identical `RSI` each time.
2. **Shard trick.** Split, compute per-shard `(U,W)`, merge ⇒ equals single pass.
3. **Gate spike.** Set `F_t := 1` ⇒ `g_t` drops to floor; `m` unaffected.
4. **PII sentinel.** Attempt to pass raw text into lens ⇒ rejected by schema.
5. **Division near zero.** Ratio lens with denominator `~0` ⇒ routed by classical fallback, stamped cause.

L11) Governance review cadence

- **Weekly.** Band distributions, gate time-series, privacy audit of random samples.
- **Monthly.** Manifest delta review (what changed, why), `knobs_hash` roll-forward.
- **Quarterly.** Lens recalibration via Appendix I, with before/after A/B stamps.

L12) Acceptance checklist (must pass)

- **Parity.** `phi((m,a)) = m` on all paths; governance never edits `m`.
- **Determinism.** Identical manifest and inputs ⇒ identical `RSI`, `RSI_env`, bands, stamps.
- **Boundedness.** All `a`, `RSI`, `RSI_env in (-1,+1)`; band thresholds and hysteresis applied.
- **Order/shard invariance.** `batch == stream == shuffled`; shard merges via `(sum U, sum W)` match single pass.
- **Privacy.** Inputs are declared, non-PII aggregates; masking enforced; retention honored.
- **Fail-safe.** Breach of any gate triggers classical fallback with stamped cause.

Stamp example (append to your one-liner).

```
SSMCLOCK1|iso_utc|svc=govern|bucket=decode_hour_14|U=2.680000|W=3.000000|RS
I=0.713036|g=0.80|RSI_env=0.570429|band=A+|manifest=knobs_hash
```

One-line takeaway. Publish the manifest, clamp and fuse in u-space, gate alignment only, band decisions, stamp everything, and keep `phi((m,a)) = m` sacrosanct — governance becomes a small checklist instead of a sprawling policy.

Appendix M — Interop & Wire Protocol (lane JSON/CSV, versioning, replay)

Purpose. Standardize how services emit/consume the lane beside classical values without breaking existing APIs. Classical magnitudes remain pristine (`phi((m,a)) = m`). Alignments stay bounded (`|a| < 1`), choosers are bounded (`|RSI| < 1`), fusion is order-invariant via `U += w*atanh(a)`; `W += w`; `a_out := tanh(U / max(W, eps_w))`. Defaults: `eps_a = 1e-6`, `eps_w = 1e-12`, weights `w := |m|^gamma` with `gamma = 1`, gate modes `"mul"` and `"u_scale"`.

M1) Minimal JSON object (per decision or per item)

All fields are optional except the `v` and `m/phi_m` pair; consumers must tolerate unknown fields (forward-compatible). Values are scalars unless noted.

```
{
  "v": "SSM-LANE/1",              // wire version
  "svc": "decode-beam",           // producer short name
  "iso_utc": "2025-10-23T11:42:03Z",
  "knobs_hash": "sha256:...",     // canonical manifest hash
  "tool_versions": "py=3.11;ssm_ai=0.1.0",  // optional provenance

  // classical magnitude path (unchanged)
  "m": 0.7319,                    // same as phi((m,a)) by definition
```

```
  "phi_m": 0.7319,              // optional echo to assert parity

  // alignment lane & fuse state
  "a": 0.4172,                  // bounded lane for this item (|a|<1)
  "U": 0.5869,                  // sum w*atanh(a) (additive)
  "W": 1.0,                     // sum w
  "RSI": 0.5281,                // chooser: tanh((V_out-U_in)/max(W_in, eps_w))
  "g_t": 0.81,                  // calm gate in [0,1]
  "RSI_env": 0.4278,            // g_t * RSI (or tanh(g_t*atanh(RSI)) per manifest)
  "band": "A0",                 // A++/A+/A0/A-/A--

  // lens + gate context (dimensionless)
  "lens_id": "decode_v1",
  "c": 0.8887,                  // gain
  "Unit": 0.78,                 // scale
  "w_rule": "uniform",          // or "abs_m_gamma:1.0"
  "gate_mode": "mul",           // or "u_scale"

  // stamping (SSM-Clock)
  "stamp": "SSMCLOCK1|...|chain",

  // optional arrays for transparency (two-channel)
  "a_in_items":  [ [ -0.2000, 1.0 ] ],  // [a_in, w]
  "a_out_items": [ [ +0.5000, 1.0 ] ]   // [a_out, w]
}
```

Required invariants (producer side).

• `phi_m == m` and equals your pre-lane baseline (`phi((m,a)) = m`).
• If `U` and `W` provided, then `a == tanh( U / max(W, eps_w) )` within dtype tolerance.
• If `RSI_env` provided, it must be `g_t * RSI` (mode `"mul"`) or `tanh( g_t * atanh(RSI) )` (mode `"u_scale"`).

M2) Batch envelope (list with header)

For high-throughput endpoints, wrap items under a header; `knobs_hash` and `v` live in the header so you don't repeat them.

```
{
  "v": "SSM-LANE/1",
  "header": {
    "svc": "rag-ranker",
    "iso_utc": "2025-10-23T11:42:03Z",
    "knobs_hash": "sha256:...",
    "tool_versions": "py=3.11;ssm_ai=0.1.0",
    "eps_a": 1e-6, "eps_w": 1e-12,
    "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00},
    "gate": {"mode":"mul","rho":0.2,"g_min":0.0}
  },
  "items": [ { ... JSON per M1 ... }, { ... }, ... ]
}
```

Replay rule (deterministic). Reducers reconstruct `a_pool := tanh( sum(U) / max(sum(W), eps_w) )` and must recover the original `a/RSI` within tolerance, independent of order or sharding.

M3) CSV schema (flat logs, copy-paste)

Tab- or comma-separated; one row per decision. Use empty strings for N/A.

```
iso_utc,svc,v,knobs_hash,m,phi_m,a,U,W,RSI,g_t,RSI_env,band,lens_id,c,Unit,
w_rule,gate_mode,stamp,tool_versions
2025-10-23T11:42:03Z,decode-beam,SSM-
LANE/1,sha256:...,0.7319,0.7319,0.4172,0.5869,1.0,0.5281,0.81,0.4278,A0,dec
ode_v1,0.8887,0.78,uniform,mul,SSMCLOCK1|...|chain,py=3.11;ssm_ai=0.1.0
```

Shard roll-up. To combine logs by hour/day/vendor, group by `(svc, knobs_hash, window)` and sum `U` and `W`; compute `a_pool := tanh( sumU / max(sumW, eps_w) )`. Do not average `a` or `RSI` directly.

M4) Optional Protobuf sketch (for gRPC)

```
message LaneItem {
  string iso_utc = 1;
  string svc = 2;
  string knobs_hash = 3;
  double m = 4;
  double phi_m = 5;
  double a = 6;
  double U = 7;
  double W = 8;
  double RSI = 9;
  double g_t = 10;
  double RSI_env = 11;
  string band = 12;
  string lens_id = 13;
  double c = 14;
  double Unit = 15;
  string w_rule = 16;
  string gate_mode = 17;
  string stamp = 18;
  repeated double a_in_items = 19;   // flattened pairs [a,w,...]
  repeated double a_out_items = 20;  // flattened pairs [a,w,...]
}

message LaneBatch {
  string v = 1;             // "SSM-LANE/1"
  string svc = 2;
  string iso_utc = 3;
  string knobs_hash = 4;
  double eps_a = 5;
  double eps_w = 6;
  string bands_json = 7;    // canonical JSON if you prefer dynamic bands
  string gate_json = 8;     // canonical JSON of gate config
  string tool_versions = 9;
```

```
  repeated LaneItem items = 10;
}
```

M5) Error codes & fallbacks

• `LANE_EPS_BREACH`: input `|a| >= 1` before clamp ⇒ producer clamps: `a_c := clamp(a, -1+eps_a, +1-eps_a)`.
• `LANE_DIV_STRICT`: division near zero under `"strict"` ⇒ consumer falls back to classical path; stamp cause.
• `LANE_MISSING_TELEMETRY`: absent `g_t` ⇒ treat as `g_t := 1` and flag.
• `LANE_MISMATCH_PHI`: `phi_m != m` ⇒ reject item; stamp and alert (parity failure).
• `LANE_VERSION_UNSUPPORTED`: `v` not recognized ⇒ parse best-effort, drop unknowns, or reject per policy.
All fallbacks keep `m` intact (`phi((m,a)) = m`).

M6) Security & privacy (defaults)

• No PII in any lane field; `lens_id`, `svc`, and `stamp` are structured tokens, not free text.
• Integrity: include `knobs_hash` (manifest digest) and stamp chain to make tamper-evident roll-ups trivial.
• Retention: raw JSON/CSV ≤ 30 days; roll-ups store only `(sumU, sumW)` and counts.

M7) Compatibility & evolution (versioning policy)

• **Forward.** Consumers ignore unknown fields (additive evolution).
• **Backward.** Producers must include `v` and `m`; adding `phi_m` is recommended for quick parity checks.
• **Knob stability.** Any change to bands, gate mode, `eps_a`, `eps_w`, or weight policy must produce a new `knobs_hash`.
• **Additive-only fields.** New keys must not change semantics of existing keys; strict backward compatibility for `m` and `phi((m,a)) = m`.
• **Replay header.** Include `knobs_hash` and `tool_versions` in batch headers to ensure reproducible replays.

M8) Consumer replay (reference pseudocode)

```
def replay(items, eps_w=1e-12):
    from math import tanh
    U = sum(x["U"] for x in items)
    W = sum(x["W"] for x in items)
    a_pool = tanh(U / max(W, eps_w)) if W > 0 else 0.0
    # parity spot-check
    for x in items:
```

```
        assert abs(x["m"] - x.get("phi_m", x["m"])) < 1e-12  # phi((m,a)) =
m
    return a_pool
```

M9) Quick producer checklist

• Compute lane and chooser with clamps: `a_c := clamp(a, -1+eps_a, +1-eps_a)`; `u := atanh(a_c)`.
• Emit `U += w*u`, `W += w`, `a := tanh(U / max(W, eps_w))`.
• Keep `m` unchanged (`phi((m,a)) = m`).
• Apply gate alignment-only: `RSI_env := g_t * RSI` or `tanh(g_t * atanh(RSI))`.
• Fill `v`, `iso_utc`, `knobs_hash`, `stamp`, `band`; include `tool_versions` when possible.

M10) Acceptance checklist (must pass)

• **Parity.** `phi((m,a)) = m` for every item; `phi_m == m`.
• **Replay.** Reducers recover `a_pool := tanh( sum(U) / max(sum(W), eps_w) )` independent of order/sharding.
• **Boundedness.** All `a`, `RSI`, `RSI_env in (-1,+1)`; clamp applied with `eps_a`.
• **Versioning.** Older readers ignore unknown fields; required keys present; `v` recognized.
• **Determinism.** Same `knobs_hash` and inputs ⇒ identical outputs, bands, and stamps.
• **Security.** No PII; `knobs_hash` and stamps included for integrity.

Stamp example (append to any row).

```
SSMCLOCK1|iso_utc|svc=wire|U=0.586900|W=1.000000|a=0.417200|RSI=0.528100|g=
0.81|RSI_env=0.427800|band=A0|manifest=knobs_hash
```

One-line takeaway. A tiny, versioned lane envelope (JSON/CSV/Proto) lets teams pass `m` unchanged and `a/RSI` transparently — deterministic replay via `(U,W)`, bounded by design, portable across vendors.

Appendix N — Economics & Rollout Worksheets (CFO ledger, ROI, procurement)

Purpose. Provide copy-paste tables, formulas, and tiny reducers so finance/ops can quantify impact without touching product logic. Classical magnitudes remain pristine (`phi((m,a)) = m`). Alignment stays bounded (`|a| < 1`), and roll-ups are order-invariant via `U += w*atanh(a)`; `W += w`; `a_pool := tanh( U / max(W, eps_w) )`. Defaults: `eps_a = 1e-6`, `eps_w = 1e-12`, `w := |m|^gamma` with `gamma = 1`, bands unchanged, gate modes `"mul"` and `"u_scale"`. All minis rounded to 6-decimals.

N1) KPI lane picks (choose 3–5 per service)

Recommended defaults (dimensionless, bounded after mapping with `tanh`):

- **Tokens per success (lower is better)**
  `e_tokens := (target_tokens_per_success - observed_tokens_per_success) / max(target_tokens_per_success, eps)` → `a_tokens := tanh( c * e_tokens )`
- **Latency p95 (lower is better)**
  `e_latency := (target_p95_ms - observed_p95_ms) / max(target_p95_ms, eps)` → `a_latency := tanh( c * e_latency )`
- **Retry rate (lower is better)**
  `e_retry := (target_retry - observed_retry) / max(target_retry, eps)` → `a_retry := tanh( c * e_retry )`
- **Acceptance/pass rate (higher is better)**
  `e_pass := (observed_pass - target_pass) / max(target_pass, eps)` → `a_pass := tanh( c * e_pass )`
- **Vendor unit cost (lower is better)**
  `e_cost := (target_cost_per_1k - observed_cost_per_1k) / max(target_cost_per_1k, eps)` → `a_cost := tanh( c * e_cost )`

Pool any subset in u-space with declared weights `w_k` (often `w := 1` for KPI lanes):
`U_pool := sum_k w_k * atanh(a_k)`; `W_pool := sum_k w_k`; `RSI_pool := tanh( U_pool / max(W_pool, eps_w) )`

N2) ROI formulas (calculator-fast; copy-paste)

- **Token reduction (%)**
  `tokens_saved_pct := (baseline_tokens - ssm_tokens) / max(baseline_tokens, eps)`
- **Latency reduction (%)**
  `lat_saved_pct := (baseline_p95_ms - ssm_p95_ms) / max(baseline_p95_ms, eps)`

- **Retry reduction (abs and %)**
  `retry_drop := baseline_retry - ssm_retry`
  `retry_drop_pct := retry_drop / max(baseline_retry, eps)`
- **Unit cost reduction (%)**
  `unit_cost_saved_pct := (baseline_cost_per_1k - ssm_cost_per_1k) / max(baseline_cost_per_1k, eps)`
- **Net savings (period)**
  `savings_usd := tokens_saved * price_per_token + retry_saves_usd + latency_value_usd`
  (Publish valuation assumptions; keep them outside SSM-AI math.)
- **Bounded health index (weekly)**
  `RSI_pool_env := g_week * RSI_pool` with `g_week in [0,1]` from the gate. Bands via defaults `A++/A+/A0/A-/A--`.

N3) CFO weekly ledger (tab-separated; paste into Word → Convert Text to Table, separator = Tabs)

Two Word-friendly blocks; numbers are illustrative. Keep `phi((m,a)) = m` in your header note.

```
WEEK: 2025-10-13
SERVICE: search
VENDOR: Vendor A
RSI_pool_env (band): 0.552100 (A+)
Bands A++/A+/A0/A-/A--: 0/3/7/0/0
Tokens total: 812000000
Tokens per success: 320
p95 latency (ms): 310
Retry rate: 0.070000
Cost per 1k: $0.90
Estimated net savings (USD): $182400
knobs_hash: sha256:...

WEEK: 2025-10-13
SERVICE: search
VENDOR: Vendor B
RSI_pool_env (band): 0.401200 (A0)
Bands A++/A+/A0/A-/A--: 0/2/8/0/0
Tokens total: 890000000
Tokens per success: 355
p95 latency (ms): 345
Retry rate: 0.100000
Cost per 1k: $1.00
Estimated net savings (USD): $0
knobs_hash: sha256:...

WEEK: 2025-10-13
SERVICE: tools
VENDOR: Vendor A
RSI_pool_env (band): 0.612500 (A+)
Bands A++/A+/A0/A-/A--: 1/4/5/0/0
Tokens total: 341000000
Tokens per success: 190
p95 latency (ms): 280
Retry rate: 0.060000
```

```
Cost per 1k: $0.90
Estimated net savings (USD): $96300
knobs_hash: sha256:...

WEEK: 2025-10-13
SERVICE: tools
VENDOR: Vendor B
RSI_pool_env (band): 0.455000 (A0)
Bands A++/A+/A0/A-/A--: 0/3/7/0/0
Tokens total: 377000000
Tokens per success: 215
p95 latency (ms): 312
Retry rate: 0.090000
Cost per 1k: $1.00
Estimated net savings (USD): $0
knobs_hash: sha256:...
```

Notes. `RSI_pool_env` is a u-space pool across the chosen KPI lanes, then gated; bands from the manifest. `knobs_hash` ties rows to the manifest in force that week.

N4) Vendor delta worksheet (copy-paste; per service)

All `*_pct` columns are fractions in `[0,1]`; positive = savings vs baseline.
`RSI_pool_env_delta := RSI_pool_env_A - RSI_pool_env_B`.

svc	comparison	tokens_saved_pct	retry_drop_pct	latency_saved_pct	unit_cost_saved_pct	RSI_pool_env_delta	weekly_savings_usd
search	Vendor A vs B	0.0876	0.3	0.1014	0.1	0.151	182400
tools	Vendor A vs B	0.0957	0.3333	0.1026	0.1	0.157	96300

Formulas (illustrative):

```
RSI_pool_env_delta := RSI_pool_env_A - RSI_pool_env_B
weekly_savings_usd := savings_usd_A - savings_usd_B
```

Given reference quantities (declare once per worksheet):

```
T_week := tokens_per_week_in_1k_units
P_ref := dollars_per_1k_tokens_baseline_or_blended
R_week := requests_or_tool_calls_per_week
C_tool := dollars_per_tool_or_API_call
```
`C_ms := dollars_per_ms_latency` *(optional, set `0` if not monetized)*
```
Spend_ref_week := baseline_weekly_AI_spend_for_this_service
```

Per-vendor decomposition (choose reference carefully; `*_pct` are fractions):

```
savings_usd_i := T_week*P_ref*tokens_saved_pct_i +
R_week*C_tool*retry_drop_pct_i + R_week*C_ms*lat_saved_pct_i +
Spend_ref_week*unit_cost_saved_pct_i
```

Guardrail (band parity):

`count_savings_i` only if `RSI_pool_env_i >= band_min` (e.g., `A0`).

N4.1) Adoption premise (zero extra infra)

SSM-AI runs on your existing stack. It requires only symbolic math and a small manifest; no retraining, no new services, and no PII. Classical numbers remain unchanged via `phi((m,a)) = m`. Selection and routing use bounded alignment (`a in (-1,+1)`) and a chooser `RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`. Order/shard invariance follows from the fuse `U += w*atanh(a)`; `W += w`; `a_out := tanh( U / max(W, eps_w) )`.

N4.2) Savings premise (plug these into the worksheets)

Use `Annual_Savings ≈ S_base * r_save` with `r_save in [0.10, 0.20]`. Start with:
– **Large:** `$500000–$2000000`
– **Mid:** `$30000–$160000`
– **NGO/public:** `$5000–$40000`
Then decompose in the ledger as: tokens + tool/API + vendor arbitrage + capacity deferral, all stamped for replay.

N4.3) Acceptance checks (pass/fail)

- **Band parity.** Compare vendors only on rows where both meet `band_min` (e.g., `A0`).
- **Order/shard invariance.** `RSI_pool_env` computed via `(U,W)`; `batch == stream == shard`.
- **Apples-to-apples.** Same manifest (`knobs_hash`), same prompts, same datasets.
- **Unit discipline.** All `*_pct` columns are fractions in `[0,1]`; do not mix with `%` cells.
- **Replayability.** Given `(U,W)` and the manifest, recompute `RSI_pool_env` and `weekly_savings_usd` exactly (within dtype tolerance).

N5) Roll-up reducer (reference pseudocode)

```
def weekly_rollup(rows, eps_w=1e-12):
    # rows: per-decision or per-session logs with
U,W,tokens,lat_ms,retries,cost
    by_key = {}  # (week_start, svc, vendor, knobs_hash) -> accum
    for r in rows:
        k = (r.week_start, r.svc, r.vendor, r.knobs_hash)
        acc = by_key.setdefault(k, {"U":0.0,"W":0.0,"tokens":0,"succ":0,

"lat_ms":[],"retries":0,"cost_tokens":0.0,"g":[]})
        acc["U"] += r.U_kpi      # sum of w*atanh(a_k) across chosen KPI
lanes
        acc["W"] += r.W_kpi
        acc["tokens"] += r.tokens
        acc["succ"]   += r.success
```

```
        acc["lat_ms"].append(r.p95_ms)   # or maintain quantile sketch
externally
        acc["retries"] += r.retries
        acc["cost_tokens"] += r.cost_tokens
        acc["g"].append(r.g)     # optional weekly gate samples
    out = []
    for k,acc in by_key.items():
        U,W = acc["U"], acc["W"]
        RSI_pool = tanh(U / max(W, eps_w)) if W > 0 else 0.0
        g_week   = max(0.0, min(1.0, sum(acc["g"])/max(len(acc["g"]),1)))
if acc["g"] else 1.0
        RSI_env  = g_week * RSI_pool
        band     = to_band(RSI_env)
        tps      = acc["tokens"]/max(acc["succ"],1)
        out.append({"key":k, "RSI_pool_env":RSI_env, "band":band,
                    "tokens_total":acc["tokens"], "tokens_per_success":tps,
                    "retry_rate":acc["retries"]/max(acc["succ"],1),
                    "cost_per_1k":acc["cost_tokens"]})
    return out
```

Quantiles. For `p95_ms`, keep an external quantile sketch (e.g., t-digest) and write the p95 into the ledger row; SSM-AI does not alter that value.

N6) Procurement snap-line (2-rule policy, bands + budget)

1. **Band rule.** Prefer vendors with `RSI_pool_env >= 0.60` (**A+**) on the dominant service bucket.
2. **Budget rule.** Within **A+**, choose minimal `cost_per_1k`; if tie, pick lowest `tokens_per_success`.
   Escalation. If all vendors `< A0`, switch to classical fallback (highest `m` path), flag for review, and stamp cause.

N7) CFO dashboard mini-spec

- **Top strip.** `RSI_pool_env` trend (last 12 weeks), band colors, `g_week`.
- **KPI tiles.** tokens/success, `p95 ms`, retry rate, cost/1k — each with band and WoW deltas.
- **Vendor panel.** Per-vendor rows with `RSI_pool_env`, band counts `A++/A+/A0/A-/A--`, cost, SLA hit rate.
- **Download.** Ledger TSV (this appendix), `knobs_hash`, and stamped manifest JSON.

N8) Acceptance checklist (must pass)

- **Parity.** Ledgers never edit classical values; `phi((m,a)) = m` holds per decision and does not change any KPI computation.
- **Order invariance.** Weekly `RSI_pool` identical under shuffles/shards via `(U,W)`.

- **Gate purity.** `RSI_pool_env := g_week * RSI_pool` (or `tanh(g_week * atanh(RSI_pool))`); KPI numerators/denominators unaffected.
- **Band determinism.** Fixed thresholds produce identical counts for the same data and manifest.
- **Reproducibility.** Recomputing from stamped CSV/JSONL recreates weekly rows bit-for-bit (within dtype tolerance).

N9) One-minute rollout plan

1. Pick 3 KPIs (Section N1).
2. Set `Unit` and `c` using Appendix I; weight policy `w := 1`.
3. Start logging `U_kpi, W_kpi, tokens, p95_ms, retries, cost_tokens, knobs_hash`.
4. Produce the weekly TSV (this appendix) and the vendor delta worksheet.
5. Freeze the manifest; compare bands and savings; decide procurement.

Stamp example (append to your one-liner).

```
SSMCLOCK1|iso_utc|svc=cfo|bucket=week_2025-10-
13|U=2.680000|W=3.000000|RSI_pool=0.713036|g=0.80|RSI_pool_env=0.570429|ban
d=A+|manifest=knobs_hash
```

One-line takeaway. Use bounded KPI lanes and u-space roll-ups to turn day-1 logs into CFO-ready ledgers and vendor decisions — order-invariant, stamped, and with your original numbers untouched (`phi((m,a)) = m`).

Appendix O — Glossary (SSMS subset & abbreviations)

Purpose. A one-page(ish) glossary of the core terms, symbols, and invariants used throughout this document. All formulas are plain ASCII. Classical values remain untouched and routing is alignment-only: `phi((m,a)) = m`.

O1) Core numerals & operators

• **Two-lane numeral.** An annotated value `x := (m,a)` with `a in (-1,+1)`.
`m` is the classical magnitude (logit, prob, distance, score, reward, latency, etc.).
`a` is a bounded alignment lane (dimensionless).

• **Collapse map.** `phi((m,a)) = m`. Classical code always sees `m` unchanged.
• **Lane purity.** Policies, bands, and gates act on `a` or `RSI`; they never modify `m`.
• **Clamp.** `a_c := clamp(a, -1+eps_a, +1-eps_a)` with default `eps_a = 1e-6`.
• **Rapidity map (u-space).** `u := atanh(a_c)` ; inverse `a := tanh(u)`.
• **Streaming fuse (order-invariant mean in u-space).**
`U += w*atanh(a)` ; `W += w` ; `a_out := tanh( U / max(W, eps_w) )` with default `eps_w = 1e-12`.
Weights default: `w := |m|^gamma` with `gamma = 1` (uniform option `w := 1` if declared).
• **Lane mul/div (M2).**
`t := atanh(a1)`; `s := atanh(a2)`; then
`a_mul := tanh( t + s )` ; `a_div := tanh( t - s )`.
(Magnitudes multiply/divide classically; see division policy.)

• **Division policy.** `"strict"` (default). Alternatives `"meadow"` or `"soft"` must be explicit. Under `"strict"`, near-zero denominators are flagged; selection falls back to declared classical logic.
• **Lens contrast (per aspect).**
General form: `e := ( sum alpha_i*P_i - sum beta_j*N_j ) / Unit`
Two-channel mapping: `a_in := tanh(-c*e_in)` ; `a_out := tanh(+c*e_out)` with `c > 0`.
• **Chooser (single bounded index).**
`U_in := sum w*atanh(a_in)` ; `V_out := sum w*atanh(a_out)` ; `W_in := sum w` ;
`RSI := tanh( (V_out - U_in) / max(W_in, eps_w) ) in (-1,+1)`.
• **Calm gate (alignment-only).**
`RSI_env := g_t * RSI` (mode `"mul"`) or `RSI_env := tanh( g_t * atanh(RSI) )` (mode `"u_scale"`), with `g_t in [0,1]` from bounded telemetry.
• **Bands (defaults).**
`A++: a >= +0.90` ; `A+: +0.60 <= a < +0.90` ; `A0: -0.60 < a < +0.60` ; `A-: -0.90 < a <= -0.60` ; `A--: a <= -0.90`.
Apply to `RSI` or `RSI_env` for decisions. Optional hysteresis around boundaries.
• **Weights (lane and chooser).** `w := |m|^gamma` (default `gamma = 1`) or `w := 1` (uniform). Declare once in the manifest.
• **Epsilons.** `eps_a = 1e-6` (clamp margin), `eps_w = 1e-12` (divide-by-zero guard), `eps_g = 1e-12` (gate guard).

O2) Telemetry lanes (for the gate)

• **Normalized inputs** in `[0,1]`. Examples:
`F_t` (contradiction/policy fraction), `D_t` (semantic drift), `L_t` (latency pressure), `E_t` (tool/API errors), `V_t` (token churn), `Q_t` (queue depth).
• **Instant gate.**
`W := wF+wD+wL+wE+wV+wQ` ;
`mix := (wF*F_t + wD*D_t + wL*L_t + wE*E_t + wV*V_t + wQ*Q_t)/max(W, eps_g)` ;
`g_inst := clamp( 1 - mix, 0, 1 )`.
• **Smoothed gate.**
`g_t := (1 - rho)*g_{t-1} + rho*g_inst` with `rho in (0,1]` (e.g., `0.20`). Optional floor `g_min`.

O3) Steps, paths, and priors

• **Step chooser.** `RSI_s := tanh( (V_out_s - U_in_s) / max(W_in_s, eps_w) ).`
• **Step gate.** `RSI_used_s := g_s * RSI_s` (or `tanh(g_s*atanh(RSI_s))`).
• **u-space step value.** `u_s := atanh( clamp(RSI_used_s, -1+eps_a, +1-eps_a) ).`
• **Tiny prior (public, optional).** `u'_s := u_s + beta*b_s` with `b_s in [-1,+1]`, small `beta >= 0`.
• **Path pool (order-invariant).**
`U_path := sum w_s * u'_s`; `W_path := sum w_s`; `RSI_path := tanh( U_path / max(W_path, eps_w) ).`
Optional plan gate: `RSI_plan_env := g_plan * RSI_path`.
• **Rollback (deterministic).** Maintain a stack of `(ΔU, ΔW)`; pop to recover until `RSI_path` meets `band_min`.

O4) Stamps & replay

• **SSM-Clock stamp (one line, ASCII).**
Example skeleton: `"SSMCLOCK1|iso_utc|rasi_idx|theta_deg|sha256(file)|chain"`.
• **knobs_hash.** Stable digest `sha256(canonical_json(manifest))` attached to every run.
• **Replay rule (order/shard invariance).**
For any window, sum `(U,W)` and compute `a_pool := tanh( sumU / max(sumW, eps_w) )`. Do not average `a` or `RSI` directly.

O5) Surfaces (drop-in points)

• **Decoding / beam.** Keep logits/probs as `m`; derive lens contrasts → `RSI` → select by `RSI_env`.
• **RAG / citations.** Pool per-doc alignments in u-space; forward bounded `a/RSI` to generation; rank by `RSI_env`.
• **Search.** Symbolic lens over quality/freshness/authority/risk; rank by `RSI` (bounded), tie-break by `m_retrieval`.
• **Tools / agents.** Gate with `g_t`; throttle retries by `RSI_env` band; never mutate tool outputs.
• **Evaluators / judges.** Convert policy/intent/style to `RSI`; route by band.
• **Ensembles.** Pool cross-vendor `RSIs` via `atanh/tanh`; choose by pooled `RSI_env`.

O6) SSM family (quick map)

• **SSM.** Two-lane numerals `x := (m,a)`, collapse `phi((m,a)) = m`, clamp→rapidity→compose→inverse.
• **SSMS.** Symbol set / verbs to define clean operator interfaces (e.g., `CLAMP`, `MAP(atanh)`, `FUSE(U/W)`, `MUL_DIV(M2)`, `BAND`, `GATE`, `STAMP`).

• **SSM-Clock.** Time-stamping and order-invariant roll-ups using `(U,W)` and ASCII stamps.
• **SSMH.** Hardware mapping of the identical lane math to a tiny accumulator + arithmetic block; software↔accelerator parity.
• **SSM-Audit.** CFO-facing lanes beside 3–5 KPIs; weekly roll-ups; ROI worksheets.
• **SSM-Search (Symbolic Search).** Lane-native retrieval; rank by `RSI` while `m_retrieval` remains intact.

O7) Acceptance gates (must hold)

• **Collapse parity.** `phi((m,a)) = m` everywhere.
• **Order invariance.** `batch == stream == shuffled` via `(U,W)` and `atanh/tanh`.
• **Boundedness.** `|a| < 1`, `|RSI| < 1`, `|RSI_env| < 1`.
• **Gate purity.** Gates scale alignment only; `m` is never edited.
• **Division policy.** `"strict"` by default; alternatives declared explicitly.
• **Determinism.** Same manifest ⇒ same outputs, bands, stamps.

O8) Quick identities & approximations

• `atanh(tanh(u)) = u` ; `tanh(atanh(a)) = a_c` (post-clamp).
• `tanh(u1+u2) = (tanh(u1) + tanh(u2)) / (1 + tanh(u1)*tanh(u2))` (implicit in M2).
• **Near zero.** `atanh(a) ≈ a`, `tanh(u) ≈ u` for small magnitudes.
• **Log-odds form.** `atanh(a) = 0.5*log((1+a)/(1-a))`.
• **Reciprocal lane.** `lane_pow(a, -1) = tanh( -atanh(a) ) = -a`.

O9) Abbreviations

• `m` — classical magnitude (unchanged under collapse).
• `a` — alignment lane in `(-1,+1)`.
• `u` — rapidity `atanh(a)`.
• `U,W` — additive fuse state; `a_out := tanh(U/max(W,eps_w))`.
• `RSI` — bounded selection index.
• `RSI_env` — gated `RSI` (`g_t` applied).
• `g_t` — calm gate value in `[0,1]`.
• `eps_a, eps_w, eps_g` — small numerical guards.
• `c` — lens gain; `Unit` — lens scale.
• `w` — weight (often `|m|^gamma` or `1`).
• `bands` — thresholds mapping `(-1,+1)` to `A++/A+/A0/A-/A--`.
• `knobs_hash` — manifest digest for reproducibility.
• `SSMCLOCK1|...` — ASCII stamp format prefix.

Stamp example (one line, ASCII).

```
SSMCLOCK1|iso_utc|k=glossary|U=0.700000|W=1.000000|a=0.604368|RSI=0.604368|
g=1.00|RSI_env=0.604368|band=A+|manifest=knobs_hash
```

Acceptance checklist (pass/fail)

- Parity: `phi((m,a)) = m` is stated and used consistently.
- Bounds: all definitions keep `|a|<1`, `|RSI|<1`, `|RSI_env|<1`.
- Order invariance: fusion stated only via `(U,W)` and `tanh(U/max(W,eps_w))`.
- Gate purity: `g_t` applies to alignment only; never touches `m`.
- Determinism: glossary terms align with manifest keys and thresholds used elsewhere.

One-line takeaway. Keep `m` pristine via `phi((m,a)) = m`; compute and fuse alignment with `clamp→atanh→sum→tanh`; pick by `RSI` (or `RSI_env`) and band it — deterministic, bounded, and order-invariant by construction.

Appendix P — FAQ & Troubleshooting (quick answers, reproducible checks)

Purpose. Fast, copy-pasteable answers to the questions teams actually ask in first rollout. All formulas are plain ASCII; classical values remain pristine (`phi((m,a)) = m`). The lane is observation-only; the chooser and gate act on alignment only.

P1) Quick FAQ (top 15)

Q1. Does SSM-AI change my numbers?
A. No. Collapse parity is non-negotiable: `phi((m,a)) = m` everywhere. You gain `a in (-1,+1)` and `RSI in (-1,+1)` for routing/visibility; `m` is untouched.

Q2. Why `tanh/atanh` instead of averaging `a` directly?
A. We need order/shard invariance and edge safety. Fuse in u-space: `U += w*atanh(a) ; W += w ; a_out := tanh( U / max(W, eps_w) )`. Averaging `a` directly breaks both near `±1` and across shards.

Q3. What do I band—`a` or `RSI`?
A. Band decisions on `RSI` (or `RSI_env`). Band intermediate `a` only for diagnostics.

Q4. What does the calm gate actually scale?
A. Alignment only. Defaults: `RSI_env := g_t * RSI` (mode `"mul"`), or curvature-preserving `RSI_env := tanh( g_t * atanh(RSI) )` (mode `"u_scale"`). `m` never changes.

Q5. When do I use uniform weights vs `|m|^gamma`?
A. Use uniform (`w := 1`) for pure comparability. Use `w := |m|^gamma` (default `gamma = 1`) when the magnitude's strength should influence pooling.

Q6. How big should `c` and `Unit` be?
A. Aim for typical `|c*e|` in `[0.3, 1.2]`. If alignments saturate (`|a| > 0.9` often), lower `c` or raise `Unit`. If `a ~ 0` everywhere, raise `c` or lower `Unit`.

Q7. What happens if I have no evidence (`W = 0`)?
A. Define neutral: `RSI := 0`. Keep logging; do not infer.

Q8. Division policy—what should I pick?
A. `"strict"` by default: near-zero denominators are flagged; routing falls back to classical. `"meadow"` or `"soft"` require explicit manifest declaration.

Q9. Can I combine multiple vendors fairly?
A. Yes. Pool bounded RSIs: `RSI_pool := tanh( sum w_i*atanh(RSI_i) / max(sum w_i, eps_w) )`. Same manifest ⇒ apples-to-apples.

Q10. Does SSM-AI replace calibration, entropy, conformal, etc.?
A. No. It composes with them. Treat those as lens terms (`e`) or priors (`b_s`) and keep the lane bounded/order-invariant.

Q11. How do I validate order invariance quickly?
A. Shuffle inputs 10×; verify identical `a_out/RSI` (within dtype tolerance) using the same `(U,W)` fuse.

Q12. What is a minimal stamp I must emit?
A. One ASCII line per decision, e.g., `"SSMCLOCK1|iso_utc|rasi_idx|theta_deg|sha256(file)|chain"`, plus `knobs_hash := sha256(canonical_json(manifest))`.

Q13. How do I interpret bands?
A. Defaults: `A++ (>= +0.90)`, `A+ (>= +0.60)`, `A0 (-0.60..+0.60)`, `A- (<= -0.60)`, `A-- (<= -0.90)`. Use hysteresis (e.g., `h = 0.02`) to avoid flapping.

Q14. Where do priors go?
A. In u-space only: `u' := u + beta*b`, with `b in [-1,+1]`, small `beta >= 0`. Priors are public, bounded, and never edit `m`.

Q15. How do I keep privacy safe?
A. Lenses use aggregates and bounded telemetry; no PII. Gate lanes must be normalized `[0,1]`, windowed, and clamped.

P2) Troubleshooting playbook (symptom → fix)

• **Symptom:** RSI looks identical across candidates.
Fix: `c` likely too small or `Unit` too large. Target `|c*e| ~ 0.3..1.2`. Confirm `W_in > 0`.

• **Symptom:** Frequent band flips around thresholds.
Fix: Add hysteresis: enter at `thr + h_up`, leave at `thr - h_dn` (e.g., `h = 0.02`). Smooth `g_t` with `rho ~ 0.20`.

• **Symptom:** Pooling changes when you reorder inputs.
Fix: Ensure you carry `(U,W)`, not just `a_out`. Recompute `U := W*atanh(a_out)` before merging shards.

• **Symptom:** Numbers blow up near `a ~ ±1`.
Fix: Clamp first: `a_c := clamp(a, -1+eps_a, +1-eps_a)`, then `atanh(a_c)`. Use `eps_a` suited to dtype (`1e-6` for f32, `1e-12` for f64).

• **Symptom:** Gate makes high-confidence items too conservative.
Fix: Switch to mode `"u_scale"`: `RSI_env := tanh( g_t * atanh(RSI) )` for gentler damping near `±1`.

• **Symptom:** Division ratios causing odd routing.
Fix: Under `"strict"`, treat near-zero denominators as invalid for actuation; fall back to classical. Log `LANE_DIV_STRICT`.

• **Symptom:** Cross-vendor bake-off seems unfair.
Fix: Fix the manifest (same `Unit`, `c`, weights, bands, `eps_*`), pool RSIs in u-space, and stamp `knobs_hash` for both vendors.

• **Symptom:** Ledger replay doesn't match online metrics.
Fix: Sum `(U,W)` by window and only then compute `tanh(U/max(W,eps_w))`. Do not average `a/RSI` directly.

P3) Minimal acceptance battery (run per deploy)

1. **Collapse parity:** for random items, assert `phi((m,a)) == m`.
2. **Clamp round-trip:** `tanh(atanh(clamp(a))) == clamp(a)` within tolerance.
3. **Order invariance:** shuffle test passes for fuse and chooser.
4. **Gate purity:** `m` unchanged; `RSI_env` scales with `g_t`.
5. **Division policy:** under `"strict"`, near-zero divisions flag and fallback.
6. **Determinism:** identical manifest (`knobs_hash`) ⇒ identical outputs/bands.

P4) Quick numeric recipes

• **Single-channel RSI (no explicit penalties):**
`RSI := tanh( sum w*atanh(a) / max(sum w, eps_w) )`
• **Two-channel lens → alignment:**
`a_in := tanh(-c*e_in)` ; `a_out := tanh(+c*e_out)`
• **Chooser:**
`RSI := tanh( (sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w) )`
• **Band function (defaults):**

```
def to_band(x):
    if x >= 0.90: return "A++"
    if x >= 0.60: return "A+"
    if x > -0.60: return "A0"
    if x > -0.90: return "A-"
    return "A--"
```

P5) Lens design do's and don'ts

• **Do:** keep `e` dimensionless: `e := (sum alpha_i*P_i - sum beta_j*N_j)/Unit`.
• **Do:** keep terms sparse and monotone.
• **Don't:** inject raw text, IDs, or PII.
• **Don't:** mix time scales silently—declare windows in the manifest.

P6) Priors & bias control

• Keep priors tiny and transparent: `u' := u + beta*b`, log `(b, beta)`.
• Never bake sensitive attributes into `b`. If in doubt, set `beta := 0`.

P7) Performance notes

• `atanh/tanh` are scalar-fast; vectorize.
• Streams carry only `(U,W)` per lane; merging is `O(1)`.
• Hardware parity (see accelerator notes): same math maps to a tiny accumulator + ALU.

P8) Common misconceptions (clarifications)

• "RSI is a probability." No. RSI is a bounded chooser in `(-1,+1)`; interpret via bands.
• "Gate changes the model output." No. Gate scales `RSI` only; `m` is unchanged.
• "We can average `a` across shards." Don't. Always combine via `(U,W)` in u-space.
• "Changing thresholds doesn't need a new config." Bands are part of the manifest—changing them changes decisions; bump `knobs_hash`.

P9) Tiny end-to-end example (5 lines)

```
e_in=0.2 ; e_out=0.5 ; c=1 ; w=1
a_in=tanh(-0.2)=-0.197375 ; a_out=tanh(0.5)=0.462117
U_in=-0.200000 ; V_out=+0.500000 ; W_in=1.000000
RSI=tanh((0.500000-(-0.200000))/1.000000)=tanh(0.700000)=0.604368 -> band
A+
RSI_env = g_t * RSI with g_t=0.80 -> 0.483494 -> band A0/A+ boundary (use
hysteresis)
```

P10) "When in doubt" defaults

`eps_a = 1e-6` ; `eps_w = 1e-12` ; `eps_g = 1e-12`
`w := |m|^1` (or `w := 1` for comparability)
`c := 1.0` ; `Unit := 1.0`
`gate.mode = "mul"` ; `rho := 0.20` ; `g_min := 0.00`
`bands`: `A++/A+/A0/A-/A--` as above
`division_policy = "strict"`

Stamp example (one line, ASCII)

```
SSMCLOCK1|iso_utc|k=faq|U=0.700000|W=1.000000|RSI=0.604368|g=0.80|RSI_env=0
.483494|band=A0|manifest=knobs_hash
```

Acceptance checklist (pass/fail)

- **Parity:** `phi((m,a)) = m` is upheld in examples and recipes.
- **Bounds:** all reported values keep `|a|<1`, `|RSI|<1`, `|RSI_env|<1`.
- **Order invariance:** examples use `(U,W)` and `tanh(U/max(W,eps_w))` only.
- **Gate purity:** `g_t` scales alignment only; `m` is never edited.
- **Determinism:** identical manifest (`knobs_hash`) and inputs reproduce identical outputs/bands.

One-line takeaway. Clamp, map to `u = atanh(a)`, add in u-space with `(U,W)`, return via `a = tanh(U/max(W,eps_w))`, choose by `RSI` (or `RSI_env`), and keep `phi((m,a)) = m` sacrosanct—most "why" and "what if" questions collapse to these few lines.

Appendix Q — Release Notes & Manifest Map (versions, diffs, reproducibility)

Purpose. Make upgrades safe and auditable: what changed, how to migrate, and how to tie any ledger row back to the exact knobs used. Classical values remain pristine (`phi((m,a)) = m`). Alignment and choosers stay bounded (`|a|<1`, `|RSI|<1`). Order/shard invariance via `U += w*atanh(a) ; W += w ; a_pool := tanh( U / max(W, eps_w) )`. Lane acts on alignment only.

Q1) Manifest map (top-level keys & meaning)

All keys are optional unless noted required. Unknown keys are ignored for math but included in hashing.

```
{
  "schema_version": "1.x",              # required; schema, not product
version

  "lens": {                              # declare per-aspect or default
    "Unit": 1.0,                         # scale so |e| is workable
    "c": 1.0,                            # gain
    "weights": "abs_m_pow",              # "abs_m_pow" or "uniform"
    "gamma": 1.0,                        # if abs_m_pow
    "channels": "two",                   # "single"|"two"
    "terms": { }                         # declared P_i / N_j
(dimensionless)
  },

  "bands": {
    "A++": 0.90, "A+": 0.60, "A0": -0.60, "A-": -0.90, "A--": -1.00,
    "hysteresis": { "h_up": 0.02, "h_dn": 0.02 }
  },

  "gate": {
    "weights": {"F":1.0,"D":1.0,"L":1.0,"E":1.0,"V":1.0,"Q":0.0},
    "rho": 0.20, "g_min": 0.00, "eps_g": 1e-12,
    "safety_notch": {"enabled": false, "s_thr": 0.80},
    "mode": "mul"                        # or "u_scale"
  },

  "policy": {
    "division_policy": "strict",         # "strict"|"meadow"|"soft"
    "eps_a": 1e-6, "eps_w": 1e-12
  },

  "path": {                              # agent chains
    "step_weight_rule": "uniform",       # or "abs_m_pow"
    "beta_prior_max": 0.25
  },

  "rollback": {
    "band_min": "A0", "delta_thr": 0.25, "g_min": 0.50,
    "budget": {"tokens": 2.0e6, "ms": 15000},
```

```
    "max_pops": 3, "on_fail": "fallback_classical"
  }
}
```

knobs hash (reproducibility).
`knobs_hash := sha256( canonical_json(manifest) )` where `canonical_json` means: UTF-8, sorted keys, no trailing zeros beyond 6 decimals, booleans/lists/maps normalized, and no comments.

Q2) Versioning & compatibility

• **Semantic versioning:** MAJOR.MINOR.PATCH.
• **MAJOR:** changes to math semantics or defaults that can affect outputs/bands (rare; require migration guide).
• **MINOR:** add fields, presets, new appendices, or optional behaviors (backward compatible).
• **PATCH:** typos, clarifications, non-semantic fixes; outputs unchanged.
Invariants that never change: `phi((m,a)) = m`. Fuse contract: `a_out := tanh( U / max(W, eps_w) )` with `U := sum w*atanh(a)`.

Q3) Release notes (current cycle)

• **Highlights.**
• Added lens calibration quickstart and ready-to-paste manifest block.
• Expanded gate presets, two-minute calibration, stamp fields.
• Introduced plan-level priors/gates, developer hooks, failure containment & rollback with `(ΔU,ΔW)` stacks.
• New appendices A–N plus O (Glossary), P (FAQ), Q (Release & Manifest).
• CFO pack with ledgers and procurement snap-line; Interop & wire protocol JSON/CSV mapping.
• **Behavioral defaults unchanged:** bands, clamp/rapidity, `U/W` fuse, division policy `"strict"`.

Q4) Migration checklist (from prior manifest)

1. Freeze your current manifest and compute `old_hash := sha256(canonical_json(old))`.
2. Add missing keys with current defaults (`policy.eps_a`, `policy.eps_w`, `gate.mode`).
3. Confirm lens block (`Unit`, `c`, `weights`, `gamma`).
4. Adopt gate presets (optional): add `rho`, `g_min`, `safety_notch`.
5. Append rollback policy (optional) for agents.
6. Re-hash: `new_hash := sha256(canonical_json(new))`; store both in your ledger.
7. Golden vectors: run included vectors; assert identical `m` (`phi((m,a)) = m`), and matching `RSI` within dtype tolerance.

Q5) Canonicalization & hashing (reference pseudocode)

```
def canonical_json(obj):
    # 1) sort keys; 2) numbers to 6-decimal fixed if float; 3) no NaN/Inf
    # 4) UTF-8, no BOM; 5) no comments
    return json.dumps(
        obj, sort_keys=True, separators=(',',':'), ensure_ascii=False,
        default=lambda x: round(float(x), 6) if isinstance(x, float) else x
    )

def knobs_hash(manifest_json):
    blob = canonical_json(manifest_json).encode('utf-8')
    return "sha256:" + sha256(blob).hexdigest()
```

Stamp field (ledger/log).
Append `knobs_hash` to each row. Example (ASCII):
`"...|knobs_hash=sha256:abcd1234|..."`

Q6) Deprecations & reserved fields

• **Reserved (do not use):** `bands.hysteresis.mode` (future), `lens.auto_learn` (must remain off; observation-only), `gate.lanes_raw` (PII-bearing data).
• **Deprecated (alias; will be removed in next MAJOR):** `gate.alpha` → use `gate.rho`.

Q7) Interop tests (must pass for providers)

1. Round-trip: manifest → `canonical_json` → `knobs_hash` → reparse → same hash.
2. Batch/stream/shard equality: same inputs, any order ⇒ identical `(U,W)` and `a_out`.
3. CSV/JSON parity: `RSI/band` computed from CSV fields equals JSON path (within dtype tolerance).
4. Gate mode equivalence: `"mul"` vs `"u_scale"` both keep `|RSI_env|<1` and never alter `m`.

Q8) Zero-risk patches (safe to adopt without A/B)

• Doc typos/clarity.
• Adding non-default presets.
• Extending appendices.
• Code comments in SDK.
• Any change that does not alter: band thresholds, `eps_*`, `Unit`, `c`, weight rule, or fuse formula.

Q9) Example minimal manifest + diff

Minimal (safe defaults).

```
{
  "schema_version": "1.0",
  "lens": {"Unit": 1.0, "c": 1.0, "weights": "abs_m_pow", "gamma": 1.0,
"channels": "two"},
  "bands": {"A++":0.90,"A+":0.60,"A0":-0.60,"A-":-0.90,"A--":-1.00,
            "hysteresis":{"h_up":0.02,"h_dn":0.02}},
  "gate": {"weights":{"F":1,"D":1,"L":1,"E":1,"V":1,"Q":0},
           "rho":0.20,"g_min":0.00,"eps_g":1e-12,"mode":"mul"},
  "policy": {"division_policy":"strict","eps_a":1e-6,"eps_w":1e-12}
}
```

Diff adding rollback (non-breaking).

```
+ "rollback": {"band_min":"A0","delta_thr":0.25,"g_min":0.50,
+              "budget":{"tokens":2000000,"ms":15000},"max_pops":3,
+              "on_fail":"fallback_classical"}
```

Hash changes, but core semantics remain; record both hashes during the switch week.

Q10) Repro checklist (auditor's 60 seconds)

• Check `schema_version`.
• Verify `knobs_hash` equals on-disk manifest hash.
• Recompute pooled RSI: `tanh( sumU / max(sumW, eps_w) )` from stamped `(U,W)`.
• Confirm `phi((m,a)) = m` on a sample.
• Verify band counts from thresholds.
• Confirm gate mode and `g_t` range `[0,1]`.

Stamp example (one line, ASCII)

```
SSMCLOCK1|iso_utc|k=release|U=2.680000|W=3.000000|RSI=0.713036|g=0.81|RSI_e
nv=0.577563|band=A+|knobs_hash=sha256:abcd1234
```

Acceptance checklist (pass/fail)

• **Parity:** `phi((m,a)) = m` verified before/after migration.
• **Determinism:** identical `knobs_hash` + inputs ⇒ identical `RSI`, bands, stamps.
• **Order/shard invariance:** pooled `RSI` equals shuffle/shard merges via `(U,W)`.
• **Bounds:** `|a|<1`, `|RSI|<1`, `|RSI_env|<1` under both gate modes.
• **Hashing:** canonicalization yields stable `knobs_hash` across platforms.
• **Interop:** CSV/JSON/Proto replays match within dtype tolerance.

One-line takeaway. Treat the manifest as a contract, hash it canonically, and stamp that hash everywhere—then any reader can replay decisions exactly while `phi((m,a)) = m` guarantees your original numbers never changed.

Appendix R — Domain Adapters (future; example: SSM-Chem)

Purpose.
Offer a small, copy-pasteable template to plug any domain into the same bounded lane without touching classical numbers. All integrations are observation-only: `phi((m,a)) = m`. Mapping and fusion remain identical everywhere: `a_c := clamp(a, -1+eps_a, +1-eps_a)`, `u := atanh(a_c)`, stream with `U += w*u`, `W += w`, invert with `a_out := tanh(U / max(W, eps_w))`. Choose by `RSI := tanh( (V_out - U_in) / max(W_in, eps_w) )`. Optional calm gate: `RSI_env := g_t * RSI` or `RSI_env := tanh( g_t * atanh(RSI) )`.

R1) Domain adapter contract (applies to any vertical)

Inputs. A set of dimensionless, declared contrasts `P_i` (supports) and `N_j` (penalties), plus optional magnitudes `m` for weights. No PII.

Lens.
Single-channel: `e := (sum alpha_i*P_i - sum beta_j*N_j) / Unit` then `a := tanh(c*e)`.
Two-channel: `a_in := tanh(-c*e_in)`, `a_out := tanh(+c*e_out)` with `e_in, e_out` dimensionless.

Chooser. `RSI := tanh( (sum w*atanh(a_out) - sum w*atanh(a_in)) / max(sum w, eps_w) )`.

Weights. `w := |m|^gamma` with `gamma = 1` by default, or `w := 1` for comparability.

Gate. Alignment-only: `RSI_env := g_t * RSI` or `RSI_env := tanh( g_t * atanh(RSI) )`.

Invariants. `phi((m,a)) = m` always, **boundedness** `|a|<1`, `|RSI|<1`, **order/shard invariance** by `(U,W)`, **zero-evidence guard**: if `sum w == 0`, set `RSI := 0`, band `"A0"`, reason `insufficient_evidence`.

Defaults to repeat in the adapter. `eps_a = 1e-6`, `eps_w = 1e-12`, `eps_g = 1e-12`, `w := |m|^gamma` with `gamma = 1`, bands `A++/A+/A0/A-/A--` unchanged, gate modes `"mul"` and `"u_scale"` both supported.

R2) Example domain: SSM-Chem (reaction screening)

Goal. Score candidates with a bounded Reaction Stability Index without altering classical energies or rates.

Feature normalization (dimensionless).
• `formation_margin := (E_broken - E_formed) / max(E_unit, eps)` positive is supportive.
• `tox_penalty := clamp(toxicity_idx, 0, 1)` penalty.
• `license_penalty := clamp(license_risk, 0, 1)` penalty.
• `evidence := clamp(citation_strength, 0, 1)` supportive.
Pick `E_unit` as a domain scale (for example median absolute reaction energy on a warmup slice). Publish scalers in the manifest.

Lens (two-channel).
• **Support:** `e_out := (alpha_f*formation_margin + alpha_ev*evidence) / Unit_out`
• **Penalty:** `e_in := (beta_t*tox_penalty + beta_l*license_penalty) / Unit_in`
• **Map:** `a_out := tanh(+c*e_out)`, `a_in := tanh(-c*e_in)`
• **Chooser:** `RSI := tanh( (atanh(a_out) - atanh(a_in)) / max(W, eps_w) )` for pooled cues with `W := sum w`, or `W := 1` for a single item.

Gate. Use service telemetry (e.g., escalations or long tails) for `g_t in [0,1]`. `m` remains classical energy or score, intact under `phi((m,a)) = m`.

R3) Worked minis (calculator-fast, repeatable)

Parameters. `Unit_out = 1.0`, `Unit_in = 1.0`, `c = 1.0`, `w = 1`.

1. **Support vs penalty, balanced**
   `formation_margin = 0.50`, `evidence = 0.30` → `e_out = 0.80` → `a_out = tanh(0.80) = 0.664037`
   `tox_penalty = 0.20`, `license_penalty = 0.10` → `e_in = 0.30` → `a_in = tanh(-0.30) = -0.291313`
   `RSI = tanh( (atanh(0.664037) - atanh(-0.291313)) / 1 ) = tanh(0.800000 - (-0.300000)) = tanh(1.100000) = 0.800499` → band `A+`.
   With `g_t = 0.81`, `"mul"` → `RSI_env = 0.648404` → band `A+`.
2. **Penalty-heavy case**
   `formation_margin = 0.20`, `evidence = 0.10` → `e_out = 0.30` → `a_out = tanh(0.30) = 0.291313`
   `tox_penalty = 0.60`, `license_penalty = 0.20` → `e_in = 0.80` → `a_in = tanh(-0.80) = -0.664037`
   `RSI = tanh( (0.300000 - 0.800000) ) = tanh(-0.500000) = -0.462117` → band `A-`.

3. **Single-channel shorthand**
   `e = e_out - e_in = 0.50 - 0.20 = 0.30` → `a = tanh(1.0 * 0.30) = 0.291313`.
   Pooling two cues `a1 = tanh(0.2) = 0.197375`, `a2 = tanh(0.4) = 0.379949`: `U = 0.2 + 0.4 = 0.6`, `W = 2`, `a_pool = tanh(0.6/2) = tanh(0.3) = 0.291313`.
   All minis respect rounding to 6-decimals where shown (`0.291313`, `0.664037`, `0.462117`, `0.800499`).

R4) Minimal manifest stub (copy-paste)

```
{
  "schema_version": "1.0",
  "dtype": "float64",
  "lens": {
    "Unit": 1.0,
    "c": 1.0,
    "weights": "abs_m_pow",
    "gamma": 1.0,
    "channels": "two",
    "terms": {
      "support": ["formation_margin", "evidence"],
      "penalty": ["tox_penalty", "license_penalty"]
    }
  },
  "policy": { "division_policy": "strict", "eps_a": 1e-6, "eps_w": 1e-12 },
  "gate": { "mode": "mul", "rho": 0.20, "g_min": 0.00, "eps_g": 1e-12,
            "weights": {"F":1,"D":1,"L":1,"E":1,"V":1,"Q":0},
            "safety_notch": {"enabled": false, "s_thr": 0.80} },
  "bands": { "A++":0.90, "A+":0.60, "A0":-0.60, "A-":-0.90, "A--":-1.00 },
  "band_on": "RSI_env"
}
```

R5) API sketch (drop-in)

```
def chem_lens(feat, c=1.0, Unit_out=1.0, Unit_in=1.0, eps_a=1e-6):
    from math import tanh
    e_out = (feat["formation_margin"] + feat["evidence"]) / max(Unit_out,
1e-12)
    e_in  = (feat["tox_penalty"] + feat["license_penalty"]) / max(Unit_in,
1e-12)
    a_out = tanh(c * e_out)
    a_in  = tanh(-c * e_in)
    # clamp for safe atanh downstream
    a_out = max(-1+eps_a, min(1-eps_a, a_out))
    a_in  = max(-1+eps_a, min(1-eps_a, a_in))
    return a_in, a_out

def chem_score(item, g_t=1.0, eps_w=1e-12, eps_a=1e-6, gate_mode="mul"):
    from math import atanh, tanh
    a_in, a_out = chem_lens(item["feat"], eps_a=eps_a)
    U_in  = atanh(a_in)
    V_out = atanh(a_out)
    W     = 1.0
```

```
    RSI   = 0.0 if W <= 0 else tanh((V_out - U_in) / max(W, eps_w))
    RSI_e = tanh(g_t * atanh(RSI)) if gate_mode=="u_scale" else (g_t * RSI)
    RSI_e = max(-1+eps_a, min(1-eps_a, RSI_e))
    return {"m": item["m"], "RSI": RSI, "RSI_env": RSI_e}
```

R6) Wire envelope (JSON, single item)


```
{
  "v":"SSM-LANE/1",
  "svc":"chem-check",
  "iso_utc":"2025-10-27T12:00:00Z",
  "knobs_hash":"sha256:...",
  "m": 2.000000,
  "phi_m": 2.000000,
  "a_in_items":[[-0.291313,1.0]],
  "a_out_items":[[+0.664037,1.0]],
  "U": 0.800000,
  "W": 1.000000,
  "RSI": 0.800499,
  "g_t": 0.810000,
  "RSI_env": 0.648404,
  "band": "A+",
  "stamp":"SSMCLOCK1|iso_utc|rasi_idx|theta_deg|sha256(file)|chain"
}
```

R7) Governance hooks (domain discipline)

• **Parity.** `phi((m,a)) = m` before and after gating.
• **Dimensionless.** Publish `Unit`, `c`, scalers, and windows.
• **Privacy.** Use aggregates only for the lens; no raw structures or identifiers.
• **Determinism.** Same manifest and inputs yield identical `RSI`, `RSI_env`, `band`.
• **Order/shard invariance.** Merge only `(U,W)` across workers.
• **Zero-evidence.** If `W == 0`, set `RSI := 0`, `band := "A0"`, reason `insufficient_evidence`.

R8) Acceptance checklist (must pass)

• **Collapse parity.** `phi((m,a)) = m` on sample rows.
• **Boundedness.** All `a`, `RSI`, `RSI_env` in `(-1,+1)`.
• **Monotone signs.** Increasing any penalty raises `|a_in|` and lowers `RSI`. Increasing any support raises `a_out` and `RSI`.
• **Order/shard invariance.** Batch equals stream equals shuffled under `(U,W)`.
• **Gating purity.** `RSI_env` follows `g_t`; `m` unchanged.
• **Replay.** Recompute `a_pool := tanh( sumU / max(sumW, eps_w) )` from logs and match within dtype tolerance.

R9) Stamp example (one line, ASCII)

```
SSMCLOCK1|2025-10-
27T12:00:00Z|k=chem|U=0.800000|W=1.000000|RSI=0.800499|g=0.810000|RSI_env=0
.648404|band=A+|knobs_hash=sha256:...
```

One-line takeaway.

Domains only need to publish a dimensionless contrast and a manifest; the lane handles the rest with `tanh/atanh` and `(U,W)` so you select by bounded `RSI` (or `RSI_env`) while `phi((m,a)) = m` keeps every classical number pristine.

OMP