Reality-Native Measurements with a Single-Anchor SI Bridge

A Parameter-Free Measurement Framework for Recognition Science

Recognition Science Research Institute

December 2025

Abstract

Claims that a theory has no free parameters are only as strong as the measurement and reporting layer connecting the theory to the world. In practice, scientific frameworks often leave critical choices implicit: windowing, coarse-graining, protocol assumptions, uncertainty semantics, and unit conversion. These choices can function as hidden degrees of freedom and make third-party audit difficult.

We introduce a measurement framework for Recognition Science (RS). The framework represents measurement provenance as explicit data: each reported value carries a protocol record (including status, assumptions, and falsifiers), an optional time window, and explicit uncertainty semantics. SI reporting is separated from RS-native units by an explicit calibration seam so that external inputs are localized and auditable.

As a concrete protocol, we implement a single-anchor SI reporting scheme. The only empirical scalar is the duration of one RS tick in SI seconds. All remaining SI conversion factors are derived from SI-definitional constants together with RS-native unit conventions. The anchor protocol is tagged as hypothesis-level because it depends on an external laboratory procedure; the conversion layer and its internal consistency checks are proved from the definitions.

One-sentence summary

We built a measurement system where protocols and calibrations are explicit objects, so that reporting-layer claims can be audited end-to-end rather than asserted informally.

Contents

1 Introduction

1.1 Why measurement hygiene matters

Modern scientific results depend on complex computational pipelines. Even when the theoretical core is clean, practical results are vulnerable to hidden choices: selection filters, window sizes, resampling rules, approximations, and unit conversions. These choices can shift outcomes and can be hard to detect after the fact. The problem is not malice; it is that our tools rarely force the researcher to write down all the choices in a way that is both human-readable and machine-checkable.

Recent reproducibility audits in high-energy physics and cosmology have highlighted how implicit calibration choices can shift published results beyond stated uncertainties. Large experimental analyses often involve long chains of corrections and calibrations; when these choices are represented only informally, it becomes difficult to audit what was assumed and what was measured. A formal approach makes it practical to represent protocols and calibration seams as explicit objects and to audit them mechanically.

1.2 The Recognition Science context

Recognition Science (RS) is a discrete-ledger framework. RS expresses core quantities in RS-native units (tick, voxel, coherence quantum, action quantum) and explicitly separates SI numerals behind a calibration seam. This creates a higher bar for measurement hygiene: if a project aims to minimize adjustable parameters, the measurement and reporting layer must not reintroduce parameter-like degrees of freedom through implicit conventions.

1.3 Design goals

The design goals for this work are:

- Measurements must be represented as structured records that include the full extraction history.
- Protocol choices must be explicit and auditable, including assumptions and falsifiers.
- Uncertainty semantics must be explicit and composable.
- Unit conversion to SI must be quarantined behind an explicit calibration object (a *calibration seam*).
- A project must be able to state, truthfully and precisely, what empirical inputs were used and where.

A **calibration seam** is a clearly marked boundary in the codebase where external conventions or empirical inputs enter the system. All data on one side of the seam is internal and dimensionless; all data on the other side carries SI units and depends on the anchor.

1.4 Related work and positioning

This work is aligned with a broad push toward reproducible computational science and explicit provenance. The FAIR principles emphasize that scientific artifacts should be findable, accessible, interoperable, and reusable. W3C PROV provides a general model for representing provenance across systems.

Our contribution is complementary: we focus on making measurement protocols, uncertainty semantics, and calibration boundaries explicit objects inside the same formal system as the theory. In practice, this means provenance is not a separate metadata file that can drift from the computation; it is carried by the measurement record itself.

2 The RS-Native Unit System

This section presents the RS-native unit system that serves as the foundation for all measurements.

2.1 Design philosophy

The RS-native unit system treats the ledger primitives as base standards. All physics can be expressed in these units without reference to SI or any external anchoring. Key properties:

- The speed of light $c = 1$ (in voxel/tick units)
- All dimensionless ratios are fixed by the golden ratio φ alone
- SI conversion is explicit and optional (via `ExternalCalibration`)

2.2 Fundamental base units

The two fundamental base units are:

- **tick** (τ_0): one discrete ledger posting interval (atomic time quantum)
- **voxel** (ℓ_0): one causal spatial step (distance light traverses in one tick)

```
-- Base unit type aliases (Real-valued counts)
abbrev Time := Real      -- measured in ticks
abbrev Length := Real    -- measured in voxels
abbrev Velocity := Real  -- measured in voxels per tick
abbrev Energy := Real    -- measured in coherence quanta (coh)
abbrev Action := Real    -- measured in action quanta (act = hbar)
abbrev Mass := Real      -- measured in coh/c^2
abbrev Frequency := Real -- measured in 1/tick
abbrev Momentum := Real  -- measured in coh/c
abbrev Charge := Real    -- measured in recognition units (
   dimensionless)
```

```
-- Canonical RS-native base units
def tick : Time := 1     -- the fundamental time quantum
def voxel : Length := 1  -- the fundamental length quantum

-- Speed of light: c = ell_0/tau_0 = 1 voxel/tick
def c : Velocity := 1
```

The canonical `RSUnits` pack bundles these for use in bridge/certificate code:

```
-- RS-native gauge: tau0 = 1 tick, ell0 = 1 voxel, c = 1
def U : RSUnits :=
  { tau0 := tick
    ell0 := voxel
    c := c
    c_ell0_tau0 := by simp [tick, voxel, c] }

-- Verify the definitions
lemma U_tau0 : U.tau0 = 1 := rfl
lemma U_ell0 : U.ell0 = 1 := rfl
lemma U_c : U.c = 1 := rfl
```

2.3 Derived quanta

From the base units, we derive energy and action quanta:

- **coh** (coherence quantum): $E_{\text{coh}} = \varphi^{-5}$ (fundamental energy quantum)
- **act** (action quantum): $\hbar = E_{\text{coh}} \cdot \tau_0$ (Planck constant equivalent)

```
-- Coherence energy quantum: phi^(-5) approx 0.0902
noncomputable def cohQuantum : Real := E_coh

-- Action quantum: hbar = E_coh * tau_0 = E_coh in RS-native units
noncomputable def hbarQuantum : Real := cohQuantum * tick

-- Since tick = 1, we have:
lemma hbarQuantum_eq_Ecoh : hbarQuantum = E_coh := by
  simp [hbarQuantum, cohQuantum, tick]

-- Mass quantum: 1 coh/c^2 = 1 coh (since c = 1)
noncomputable def massQuantum : Real := cohQuantum

-- Convert mass count to raw RS scale
def mass_raw (m : Mass) : Real := m * massQuantum
```

Additional derived quanta for frequency and momentum:

```
-- Frequency quantum: 1/tick (inverse of fundamental time)
noncomputable def freqQuantum : Frequency := 1 / tick
```

```
-- Momentum quantum: E_coh/c = E_coh (since c = 1)
noncomputable def momentumQuantum : Real := cohQuantum / c

-- Since c = 1:
lemma momentumQuantum_eq_cohQuantum : momentumQuantum = cohQuantum
    := by
  simp [momentumQuantum, c]
```

2.4 The φ-ladder

All RS quantities are organized on a φ-ladder. The ladder provides natural scaling for masses, energies, times, and lengths:

- Mass rungs: $m_n = m_0 \cdot \varphi^n$
- Time rungs: $\tau_n = \tau_0 \cdot \varphi^n$
- Length rungs: $\ell_n = \ell_0 \cdot \varphi^n$
- Energy rungs: $E_n = E_0 \cdot \varphi^n$

```
-- phi-ladder scaling: compute phi^n for integer rung
noncomputable def phiRung (n : Int) : Real := phi ^ n

-- Scale any quantity by n rungs on the phi-ladder
noncomputable def scaleByPhi (x : Real) (n : Int) : Real := x *
   phiRung n

lemma phiRung_pos (n : Int) : 0 < phiRung n := zpow_pos phi_pos n
lemma phiRung_zero : phiRung 0 = 1 := by simp [phiRung]
lemma phiRung_one : phiRung 1 = phi := by simp [phiRung]

lemma phiRung_add (m n : Int) : phiRung (m + n) = phiRung m *
   phiRung n := by
  exact zpow_add phi_ne_zero m n

lemma phiRung_neg (n : Int) : phiRung (-n) = 1 / phiRung n := by
  simp only [phiRung, one_div]
  rw [zpow_neg]

lemma phiRung_neg_one : phiRung (-1) = 1 / phi := by
  simp only [phiRung, zpow_neg_one, one_div]
```

2.5 The 8-tick cycle

The fundamental ledger cycle has period $8 = 2^3$ (forced by $D = 3$ spatial dimensions). This defines the octave structure of RS:

```
-- The octave period: 8 ticks
def octavePeriod : Time := 8 * tick

-- The breath cycle: 1024 ticks (8 * 128 = 8 * 2^7)
def breathCycle : Time := 1024 * tick

-- Convert tick count to octave count (integer division)
def ticksToOctaves (t : Nat) : Nat := t / 8

-- Phase within an octave (0..7)
def octavePhase (t : Nat) : Fin 8 := { val := t % 8, isLt := Nat.
   mod_lt t 8 }
```

2.6 Gap-45 synchronization

The gap-45 rung (φ^{45}) provides critical phase synchronization:

```
-- The gap-45 rung: phi^45
noncomputable def gap45 : Real := phiRung 45

-- The synchronization period: lcm(8, 45) = 360
def syncPeriod : Nat := 360

lemma syncPeriod_eq_lcm : syncPeriod = Nat.lcm 8 45 := by
   native_decide
```

This synchronization forces $D = 3$ as the unique spatial dimension with this property.

2.7 K-gate derived displays

The K-gate provides display quantities relating RS-native units to observable scales:

```
-- Recognition time display: tau_rec = (2*pi)/(8 * ln(phi)) * tau0
noncomputable def tau_rec : Time :=
  RSUnits.tau_rec_display U

-- Kinematic wavelength display: lambda_kin = (2*pi)/(8 * ln(phi))
    * ell0
noncomputable def lambda_kin : Length :=
  RSUnits.lambda_kin_display U

-- Both equal the K-gate ratio
theorem tau_rec_eq_K_gate_ratio :
    tau_rec = RSUnits.K_gate_ratio := by
```

```
  unfold tau_rec
  have hlog : Real.log phi != 0 := ne_of_gt (Real.log_pos
     one_lt_phi)
  simp [RSUnits.tau_rec_display, RSUnits.K_gate_ratio, U, tick]
  field_simp [hlog]
  ring

theorem lambda_kin_eq_K_gate_ratio :
    lambda_kin = RSUnits.K_gate_ratio := by
  unfold lambda_kin
  have hlog : Real.log phi != 0 := ne_of_gt (Real.log_pos
     one_lt_phi)
  simp [RSUnits.lambda_kin_display, RSUnits.K_gate_ratio, U, voxel
     ]
  field_simp [hlog]
  ring
```

3 The Core Measurement Framework

This section presents the formal definitions that constitute the measurement framework.

3.1 Protocol status

Every protocol carries a status tag indicating its epistemic standing:

```
inductive Status
  | spec       -- specification (definitional)
  | derived    -- derived from other protocols
  | hypothesis -- depends on external/empirical input
  | scaffold   -- placeholder, not yet complete
  deriving DecidableEq, Repr
```

3.2 The Protocol structure

A `Protocol` describes how an observable is extracted from a state or trace. It includes explicit assumptions and falsifiers:

```
structure Protocol where
  -- Short protocol identifier (stable, machine-friendly)
  name : String
  -- Human-readable summary
  summary : String := ""
  -- Claim hygiene for the extraction step
  status : Status := .spec
  -- Explicit assumptions (kept small and testable)
  assumptions : List String := []
```

```
  -- Falsifiers: what would prove this protocol wrong
  falsifiers : List String := []
```

3.3 Protocol hygiene predicate

The hygiene rule requires that hypothesis-level and scaffold-level protocols must declare non-empty assumptions and falsifiers. This prevents hidden choices from entering the measurement layer without explicit documentation.

```
def Protocol.hygienic (p : Protocol) : Prop :=
  p.name != "" /\
    match p.status with
    | .hypothesis | .scaffold =>
        p.assumptions != [] /\ p.falsifiers != []
    | _ => True

-- Executable boolean version for audits
def Protocol.hygienicBool (p : Protocol) : Bool :=
  (!p.name.isEmpty) &&
    match p.status with
    | .hypothesis | .scaffold =>
        (!p.assumptions.isEmpty) && (!p.falsifiers.isEmpty)
    | _ => true
```

3.4 Time windows

Measurements can be time-bounded. A `Window` specifies a start tick and a length (in ticks):

```
structure Window where
  t0 : Nat    -- Start tick
  len : Nat   -- Window length in ticks (0 = instantaneous)
  deriving DecidableEq

def Window.instant (t : Nat) : Window := { t0 := t, len := 0 }
def Window.stop (w : Window) : Nat := w.t0 + w.len
```

3.5 Uncertainty semantics

The framework provides three uncertainty representations. This is intentionally lightweight—it is a reporting layer, not a full probability theory.

```
inductive Uncertainty where
  -- Symmetric 1-sigma uncertainty (standard deviation)
  | sigma (s : Real) (hs : 0 <= s)
  -- Interval uncertainty: true value lies in [lo, hi]
  | interval (lo hi : Real) (hlohi : lo <= hi)
```

```
  -- Finite discrete distribution scaffold (value, weight) pairs
  | discrete (support : List (Real * Real))
```

The `discrete` form is a scaffold: it does not enforce normalization or non-negativity constraints in the core type. Protocols must state hygiene requirements for discrete distributions.

Helper extractors provide typed access to uncertainty data:

```
namespace Uncertainty

def sigmaVal : Uncertainty -> Option Real
  | sigma s _ => some s
  | _ => none

def intervalBounds : Uncertainty -> Option (Real * Real)
  | interval lo hi _ => some (lo, hi)
  | _ => none

end Uncertainty
```

3.6 The Measurement record

The central object is a `Measurement` record that packages a value with its full provenance:

```
structure Measurement (a : Type) where
  value : a
  window : Option Window := none
  protocol : Protocol
  uncertainty : Option Uncertainty := none
  notes : List String := []
```

Measurements can be mapped through derived computations while preserving protocol and uncertainty metadata:

```
namespace Measurement

-- Map the value of a measurement, preserving window/protocol/
   uncertainty/notes
def map (f : a -> b) (m : Measurement a) : Measurement b :=
  { value := f m.value
    window := m.window
    protocol := m.protocol
    uncertainty := m.uncertainty
    notes := m.notes }

-- Map the value and replace protocol (when deriving a new
   observable)
def mapWithProtocol (f : a -> b) (p : Protocol)
    (m : Measurement a) : Measurement b :=
```

```
  { value := f m.value
    window := m.window
    protocol := p
    uncertainty := m.uncertainty
    notes := m.notes }

-- Transform the uncertainty record (when present)
def mapUncertainty (uMap : Uncertainty -> Uncertainty)
    (m : Measurement a) : Measurement a :=
  { value := m.value
    window := m.window
    protocol := m.protocol
    uncertainty := m.uncertainty.map uMap
    notes := m.notes }

def addNote (note : String) (m : Measurement a) : Measurement a :=
  { m with notes := m.notes ++ [note] }

def addNotes (notes : List String) (m : Measurement a) :
   Measurement a :=
  { m with notes := m.notes ++ notes }

end Measurement
```

3.7 Observables

An `Observable` is a function that extracts a `Measurement` from some state type:

```
abbrev Observable (S a : Type) : Type := S -> Measurement a
```

This allows catalog observables to be defined uniformly and composed while preserving provenance.

3.8 Type-tagged quantities

To prevent unit confusion, the framework provides type-tagged real-valued quantities. Each unit tag is an empty inductive type; the `Quantity` wrapper carries the tag at the type level:

```
-- A real-valued quantity tagged with a unit/semantic label
structure Quantity (U : Type) where
  val : Real

-- Coercion to Real for arithmetic
instance (U : Type) : CoeTC (Quantity U) Real := { coe := Quantity
   .val }

-- Arithmetic operations within the same unit type
namespace Quantity
```

```
instance (U : Type) : Zero (Quantity U) := { zero := { val := 0 }
   }
instance (U : Type) : Add (Quantity U) := { add := fun a b => {
   val := a.val + b.val } }
instance (U : Type) : Sub (Quantity U) := { sub := fun a b => {
   val := a.val - b.val } }
instance (U : Type) : Neg (Quantity U) := { neg := fun a => { val
   := -a.val } }
instance (U : Type) : SMul Real (Quantity U) := { smul := fun r a
   => { val := r * a.val } }

-- Simp lemmas for reduction
theorem val_zero (U : Type) : (0 : Quantity U).val = 0 := rfl
theorem val_add (a b : Quantity U) : (a + b).val = a.val + b.val
   := rfl
theorem val_sub (a b : Quantity U) : (a - b).val = a.val - b.val
   := rfl
theorem val_neg (a : Quantity U) : (-a).val = -a.val := rfl
theorem val_smul (r : Real) (a : Quantity U) : (r * a).val = r * a
   .val := rfl
end Quantity
```

Unit tags are empty inductive types:

```
-- Unit/semantic tags (empty types)
inductive TickUnit : Type
inductive VoxelUnit : Type
inductive CohUnit : Type
inductive ActUnit : Type
inductive CostUnit : Type
inductive SkewUnit : Type
inductive MeaningUnit : Type
inductive QualiaUnit : Type
inductive ZUnit : Type

-- Type aliases for RS-native quantities
abbrev Tick := Quantity TickUnit
abbrev Voxel := Quantity VoxelUnit
abbrev Coh := Quantity CohUnit      -- coherence quantum (energy)
abbrev Act := Quantity ActUnit      -- action quantum (hbar)
abbrev Cost := Quantity CostUnit    -- J-cost
abbrev Skew := Quantity SkewUnit    -- reciprocity skew
abbrev Meaning := Quantity MeaningUnit
abbrev Qualia := Quantity QualiaUnit
abbrev ZCharge := Quantity ZUnit    -- Z-invariant
```

These tags prevent accidental addition of mismatched quantities unless the tag is explicitly discarded via the coercion to `Real`.

4 Observable Catalogs

The framework includes catalogs of concrete RS-native observables. This section presents the core catalogs and adapters that connect (i) ledger states, (ii) meaning carriers, and (iii) ethics/qualia operators into protocol-carrying measurements.

4.1 Ledger observables

These observables extract measurements from `LedgerState` (the core Recognition Science state type).

First, helper constructors wrap raw values in type-tagged quantities:

```
-- Helper constructors for type-tagged quantities
def asCost (x : Real) : Cost := { val := x }
def asSkew (x : Real) : Skew := { val := x }
```

Protocol definitions include explicit falsifiers for derived observables:

```
def protocol_recognitionCost : Protocol :=
  { name := "ledger.recognition_cost"
    summary := "Sum of J-costs over active bonds in a single
       LedgerState."
    status := .derived
    assumptions := []
    falsifiers := [] }

def protocol_netSkew : Protocol :=
  { name := "ledger.net_skew"
    summary := "Net reciprocity skew s = sum(log(x_e)) over active
        bonds."
    status := .derived
    assumptions := []
    falsifiers := ["Find an admissible state with net_skew != 0."]
        }

def protocol_totalZ : Protocol :=
  { name := "ledger.total_Z"
    summary := "Total Z-invariant (integer) from Z_patterns.sum."
    status := .derived
    assumptions := []
    falsifiers := ["Find evolution step where total_Z changes."] }

def protocol_pathAction : Protocol :=
  { name := "ledger.path_action"
    summary := "Discrete path action C[gamma] = sum(
       RecognitionCost(s_t)) over trace."
    status := .derived
    assumptions := ["Trace is ordered in time."]
    falsifiers := [] }
```

```
def protocol_reciprocitySkewAbs : Protocol :=
  { name := "ledger.reciprocity_skew_abs"
    summary := "Total absolute reciprocity skew sum(|log(x_e)|)
       over active bonds."
    status := .derived
    assumptions := []
    falsifiers := [] }
```

A helper function computes the time window for a trace:

```
private def windowOfTrace (gamma : List LedgerState) : Option
   Window :=
  match gamma with
  | [] => none
  | s :: ss =>
    let t0 := s.time
    let tLast := (List.getLastD (s :: ss) s).time
    some { t0 := t0, len := tLast - t0 }
```

The observables themselves extract measurements with full provenance:

```
-- Observable on a single ledger state
noncomputable def recognitionCost : Observable LedgerState Cost :=
  fun s =>
  { value := asCost (RecognitionCost s)
    window := some (Window.instant s.time)
    protocol := protocol_recognitionCost }

noncomputable def netSkew : Observable LedgerState Skew :=
  fun s =>
  { value := asSkew (net_skew s)
    window := some (Window.instant s.time)
    protocol := protocol_netSkew }

noncomputable def reciprocitySkewAbs : Observable LedgerState Skew
    :=
  fun s =>
  { value := asSkew (reciprocity_skew_abs s)
    window := some (Window.instant s.time)
    protocol := protocol_reciprocitySkewAbs }

def totalZ : Observable LedgerState Int :=
  fun s =>
  { value := total_Z s
    window := some (Window.instant s.time)
    protocol := protocol_totalZ }

-- Observable on a trace (list of states)
```

```
noncomputable def pathAction : Observable (List LedgerState) Cost
   :=
  fun gamma =>
  { value := asCost (PathAction gamma)
    window := windowOfTrace gamma
    protocol := protocol_pathAction }
```

4.2 Voxel meaning observables

These observables extract DFT-based meaning primitives from `MeaningfulVoxel`:

4.2.1 Meaning primitives (the semantic carrier)

In RS, "meaning" begins as a structured carrier, not a post-hoc label. The carrier is a `MeaningfulVoxel`: eight phase slots, each containing a `Photon`. Meaning is then read from the voxel's DFT spectrum.

```
-- A photon carries amplitude and within-slot phase.
structure Photon where
  amplitude : Real
  phase_offset : Real
  amp_nonneg : 0 <= amplitude

namespace Photon
def vacuum : Photon := { amplitude := 0, phase_offset := 0,
   amp_nonneg := by simp }
def unit : Photon := { amplitude := 1, phase_offset := 0,
   amp_nonneg := by norm_num }

-- Complex representation for DFT processing.
def toComplex (p : Photon) : Complex :=
  p.amplitude * Complex.exp (Complex.I * p.phase_offset)
end Photon

-- A meaningful voxel: 8 photons at 8 phase positions.
structure MeaningfulVoxel where
  photon : Fin 8 -> Photon

namespace MeaningfulVoxel
def totalEnergy (v : MeaningfulVoxel) : Real :=
  Finset.univ.sum (fun p : Fin 8 => (v.photon p).amplitude ^ 2)

def toComplexSignal (v : MeaningfulVoxel) : Fin 8 -> Complex :=
  fun p => (Photon.toComplex (v.photon p))

-- THE KEY OPERATION: DFT decomposition into frequency modes.
def frequencySpectrum (v : MeaningfulVoxel) : Fin 8 -> Complex :=
```

```
  dft8 v.toComplexSignal

def modeAmplitude (v : MeaningfulVoxel) (k : Fin 8) : Real :=
  amplitude (v.frequencySpectrum k)
end MeaningfulVoxel

-- Neutrality is the DC constraint: the 0-mode must vanish.
def isNeutral (v : MeaningfulVoxel) : Prop :=
  v.frequencySpectrum 0 = 0

theorem neutral_iff_zero_sum (v : MeaningfulVoxel) :
    isNeutral v <-> (Finset.univ.sum (fun p : Fin 8 => v.
       toComplexSignal p) = 0) := by
  -- At k=0, the DFT kernel is identically 1, so the DC component
     equals the time-domain sum.
  unfold isNeutral MeaningfulVoxel.frequencySpectrum dft8
  constructor
  -- forward direction (isNeutral -> time-domain sum = 0)
  intro h
    -- DFT[0] = sum x_n, and DFT[0] = 0 by assumption.
    -- (The simplification step uses that the 0-mode exponent is
       0, so omega^(n*0)=1.)
    convert h using 1
    congr 1
    ext p
    simp
  -- reverse direction (time-domain sum = 0 -> isNeutral)
  intro h
    -- If the time-domain sum is 0, then DFT[0] is 0.
    convert h using 1
    congr 1
    ext p
    simp
```

4.2.2 Measurement observables on meaning carriers

```
-- Helper constructor for coherence quanta
def asCoh (x : Real) : Coh := { val := x }
```

Protocol and observable definitions:

```
def protocol_totalEnergy : Protocol :=
  { name := "voxel.total_energy"
    summary := "Total voxel energy: sum_p amplitude(p)^2 (RS-
       native units)."
    status := .derived
    assumptions := []
```

```
    falsifiers := [] }

def protocol_modeEnergy : Protocol :=
  { name := "voxel.mode_energy"
    summary := "Per-mode energy: (modeAmplitude k)^2 for k in Fin
       8."
    status := .derived
    assumptions := []
    falsifiers := [] }

noncomputable def totalEnergy : Observable MeaningfulVoxel Coh :=
  fun v =>
  { value := asCoh v.totalEnergy
    window := none
    protocol := protocol_totalEnergy }

noncomputable def modeEnergy (k : Fin 8) : Observable
   MeaningfulVoxel Coh :=
  fun v =>
  { value := asCoh ((v.modeAmplitude k) ^ 2)
    window := none
    protocol := protocol_modeEnergy
    notes := ["Interpretation: squared DFT-mode amplitude."] }

noncomputable def spectrum : Observable MeaningfulVoxel (Fin 8 ->
   Coh) :=
  fun v =>
  { value := fun k => asCoh ((v.modeAmplitude k) ^ 2)
    window := none
    protocol := protocol_modeEnergy }
```

4.3 Ethics observables (moral skew and virtue operators)

Skew appears in RS at two layers:

- **Ledger layer**: skew is a reciprocity imbalance computed from bonds in a `LedgerState` (see `netSkew` above).
- **Ethics layer**: skew is carried in a `MoralState` and used by virtue operators that select among candidate actions.

```
-- Helper constructor for moral skew.
def asSkew (x : Real) : Skew := { val := x }

def protocol_skew : Protocol :=
  { name := "ethics.skew"
```

```
    summary := "Agent skew (ledger reciprocity imbalance proxy)
       from MoralState."
    status := .derived
    assumptions := []
    falsifiers := [] }

def protocol_absSkew : Protocol :=
  { name := "ethics.abs_skew"
    summary := "Magnitude of skew |sigma| (risk/imbalance
       magnitude)."
    status := .derived
    assumptions := []
    falsifiers := [] }

noncomputable def skew : Observable MoralState Skew :=
  fun s => { value := asSkew s.skew, window := none, protocol :=
     protocol_skew }

noncomputable def absSkew : Observable MoralState Skew :=
  fun s => { value := asSkew (abs s.skew), window := none,
     protocol := protocol_absSkew }

def protocol_wiseChoice : Protocol :=
  { name := "ethics.wise_choice"
    summary := "WiseChoice: selects option minimizing phi-
       discounted skew."
    status := .derived
    assumptions := ["Choice list represents admissible
       alternatives."]
    falsifiers := [] }

-- NOTE: lambda is a policy parameter (risk-aversion weight), not
   a core physics parameter.
def protocol_prudentChoice : Protocol :=
  { name := "ethics.prudent_choice"
    summary := "PrudentChoice: risk-adjusted minimization over {
       status quo} union choices."
    status := .derived
    assumptions := ["lambda is an externally chosen risk-aversion
       weight (policy, not core physics)."]
    falsifiers := [] }

noncomputable def wiseChoice (s : MoralState) : Observable (List
   MoralState) MoralState :=
  fun choices => { value := WiseChoice s choices, window := none,
     protocol := protocol_wiseChoice }
```

```
noncomputable def prudentChoice (s : MoralState) (lambda : Real)
    : Observable (List MoralState) MoralState :=
  fun choices =>
  { value := PrudentChoice s choices lambda
    window := none
    protocol := protocol_prudentChoice
    notes := ["If lambda=0, Prudence reduces to min(|skew|) over {
       s} union choices."] }
```

4.4 Qualia observables (ULQ primitives)

ULQ introduces qualia primitives (mode, level/intensity, valence, timing). The measurement catalog wraps those primitives in protocol-carrying observables.

```
def asQualia (x : Real) : Qualia := { val := x }

def protocol_qualiaModeOfWToken : Protocol :=
  { name := "qualia.mode_of_wtoken"
    summary := "Qualia mode derived from a WToken's non-DC
       dominant DFT mode."
    status := .derived
    assumptions := ["Token is neutral (DC excluded) as in ULQ/ULL
       constraints."]
    falsifiers := [] }

def protocol_qualiaEnergy : Protocol :=
  { name := "qualia.energy"
    summary := "QualiaEnergy(q) := phi^(level) * (1 + |valence|)."
    status := .derived
    assumptions := []
    falsifiers := [] }

noncomputable def qualiaModeOfWToken : Observable WToken
   QualiaMode :=
  fun w => { value := qualiaModeOfWToken w, window := none,
     protocol := protocol_qualiaModeOfWToken }

noncomputable def qualiaEnergy : Observable QualiaSpace Qualia :=
  fun q => { value := asQualia (qualiaEnergy q), window := none,
     protocol := protocol_qualiaEnergy }
```

4.5 Meaning adapters (voxel → token) (scaffold)

Meaning also has a protocol seam: turning a `MeaningfulVoxel` into a token candidate for downstream symbolic/semantic layers. This adapter is explicitly marked `.scaffold`: it

performs a mathematically well-defined projection (neutrality + normalization) but makes no claim that the resulting token is the correct semantic category.

```
-- Mean of an 8-phase complex signal.
def mean8 (x : Fin 8 -> Complex) : Complex :=
  ((1 / 8 : Real) : Complex) * (Finset.univ.sum (fun i : Fin 8 =>
     x i))

-- Mean-free projection (neutrality enforcement).
def centered (x : Fin 8 -> Complex) : Fin 8 -> Complex :=
  fun i => x i - mean8 x

-- Energy of an 8-phase signal.
def energy8 (x : Fin 8 -> Complex) : Real :=
  Finset.univ.sum (fun i : Fin 8 => Complex.normSq (x i))

-- Normalize by L2 energy (requires 0 < energy8 x).
def normalize8 (x : Fin 8 -> Complex) : Fin 8 -> Complex :=
  fun i => x i / (Real.sqrt (energy8 x) : Complex)

-- Protocol: voxel -> WToken projection (placeholder, falsifiable)
   .
def protocol_voxel_to_wtoken : Protocol :=
  { name := "adapter.voxel_to_wtoken"
    summary := "Scaffold: project an 8-phase voxel signal to a
       token candidate by neutrality and L2 normalization."
    status := .scaffold
    assumptions :=
      [ "A MeaningfulVoxel's complex 8-signal is an appropriate
         raw carrier for the token basis."
      , "Neutrality is enforced by subtracting the mean (DC
         component)."
      , "Unit norm is enforced by dividing by sqrt(energy) after
         neutrality projection."
      , "Semantic classification is out of scope here."
      ]
    falsifiers :=
      [ "If the correct semantic basis is in frequency-domain
         rather than time-domain, replace this projection."
      , "If neutrality must be enforced by a different constraint
         than mean subtraction, replace centered()."
      , "If zero-energy projected signals are meaningful, replace
         the Option-returning seam."
      ] }

-- Observable: voxel -> optional legal token (none if neutral-
   projected energy is zero).
noncomputable def legalWToken : Observable MeaningfulVoxel (Option
```

```
    LegalWToken) :=
  fun v =>
    { value := toLegalWToken v
      window := none
      protocol := protocol_voxel_to_wtoken
      uncertainty := none
      notes := ["Returns none when the neutral-projected signal
         has zero energy."] }
```

5 Cross-Agent Alignment

Comparing measurements across agents is a common source of hidden assumptions. Even if two agents report numbers with the same unit label, they may be using different extraction conventions or incompatible semantics. The RS-native measurement framework includes an explicit alignment seam.

5.1 Alignment protocol

An `AlignmentProtocol` extends `Protocol` with explicit invariants for cross-agent comparability:

```
structure AlignmentProtocol where
  protocol : Protocol
  -- Invariants that must be preserved under alignment
  invariants : List String := []

def AlignmentProtocol.name (A : AlignmentProtocol) : String := A.
   protocol.name
def AlignmentProtocol.status (A : AlignmentProtocol) : Status := A
   .protocol.status
```

5.2 Alignment structure

An `Alignment` packages a map with its protocol:

```
-- An alignment map from one agent's coordinate system to another'
   s
abbrev AlignmentMap (a b : Type) : Type := a -> b

-- A packaged alignment: map + protocol hygiene
structure Alignment (a b : Type) where
  map : AlignmentMap a b
  protocol : AlignmentProtocol
```

5.3 Applying alignments

Applying an alignment transforms the measurement value while preserving the original window and uncertainty metadata, and appends an audit note:

```
def Alignment.apply (A : Alignment a b) (m : Measurement a) :
   Measurement b :=
  { value := A.map m.value
    window := m.window
    protocol := A.protocol.protocol
    uncertainty := m.uncertainty
    notes := m.notes ++ ["Aligned via " ++ A.protocol.name] }
```

This layer is intentionally marked as scaffold: it does not solve cross-agent comparability, but it forces alignment choices to be explicit and auditable.

6 The SI Calibration Seam

This section presents the explicit boundary where RS-native quantities can be reported in SI.

6.1 The ExternalCalibration structure

If you want "seconds" and "meters," you must provide an explicit calibration mapping RS primitives to SI. This keeps the empirical seam explicit and auditable:

```
structure ExternalCalibration where
  -- Seconds per tick (tau_0 in seconds)
  seconds_per_tick : Real
  -- Meters per voxel (ell_0 in meters)
  meters_per_voxel : Real
  -- Joules per coherence quantum
  joules_per_coh : Real
  -- All conversion factors are positive
  seconds_pos : 0 < seconds_per_tick
  meters_pos : 0 < meters_per_voxel
  joules_pos : 0 < joules_per_coh
  -- Consistency: c = ell_0/tau_0 must equal 299792458 m/s
  speed_consistent : meters_per_voxel / seconds_per_tick =
     299792458
```

The `speed_consistent` field enforces that any valid calibration respects the SI-definitional speed of light.

6.2 Conversion functions

Given an `ExternalCalibration`, the following functions convert RS-native quantities to SI:

```
-- Time: ticks to seconds
def to_seconds (cal : ExternalCalibration) (t : Tick) : Real :=
  (t : Real) * cal.seconds_per_tick

-- Length: voxels to meters
def to_meters (cal : ExternalCalibration) (L : Voxel) : Real :=
  (L : Real) * cal.meters_per_voxel

-- Velocity: voxels/tick to m/s
def to_m_per_s (cal : ExternalCalibration) (v : Velocity) : Real
   :=
  v * (cal.meters_per_voxel / cal.seconds_per_tick)

-- Energy: coh to Joules
def to_joules (cal : ExternalCalibration) (E : Coh) : Real :=
  (E : Real) * cal.joules_per_coh

-- Mass: coh/c^2 to kg (using E=mc^2)
def to_kg (cal : ExternalCalibration) (m : Mass) : Real :=
  m * cal.joules_per_coh / (299792458 ^ 2)

-- Frequency: 1/tick to Hz
def to_hertz (cal : ExternalCalibration) (f : Frequency) : Real :=
  f / cal.seconds_per_tick

-- Action: act to Joule-seconds
-- In RS-native units: 1 act = 1 coh * 1 tick
-- So SI conversion is: joules_per_coh * seconds_per_tick
def to_joule_seconds (cal : ExternalCalibration) (A : Act) : Real
   :=
  (A : Real) * (cal.joules_per_coh * cal.seconds_per_tick)
```

6.3 Speed consistency theorem

The framework proves that any valid calibration reports the correct SI speed of light:

```
theorem c_in_si (cal : ExternalCalibration) :
    to_m_per_s cal c = 299792458 := by
  simp [to_m_per_s, c, cal.speed_consistent]
```

6.4 Measurement-level SI conversion

The SI adapter module provides measurement-level conversion that preserves provenance and scales uncertainty. The key insight is that when we scale a value by a conversion factor, we must also scale its uncertainty by the same factor.

```
-- Scale uncertainty by a positive conversion factor
private def scaleUncertainty (c : Real) (hc : 0 <= c)
    (u : Uncertainty) : Uncertainty :=
  match u with
  | .sigma s hs => .sigma (s * c) (mul_nonneg hs hc)
  | .interval lo hi hlohi =>
      .interval (lo * c) (hi * c) (mul_le_mul_of_nonneg_right
         hlohi hc)
  | .discrete support =>
      .discrete (support.map (fun vw => (vw.1 * c, vw.2)))

-- Ticks to SI seconds (preserving provenance)
def measure_to_seconds (cal : ExternalCalibration)
    (m : Measurement Tick) : Measurement Real :=
  Measurement.map (to_seconds cal)
    (Measurement.mapUncertainty
      (fun u => scaleUncertainty cal.seconds_per_tick
                  (le_of_lt cal.seconds_pos) u) m)

-- Voxels to SI meters (preserving provenance)
def measure_to_meters (cal : ExternalCalibration)
    (m : Measurement Voxel) : Measurement Real :=
  Measurement.map (to_meters cal)
    (Measurement.mapUncertainty
      (fun u => scaleUncertainty cal.meters_per_voxel
                  (le_of_lt cal.meters_pos) u) m)

-- Coh to SI Joules (preserving provenance)
def measure_to_joules (cal : ExternalCalibration)
    (m : Measurement Coh) : Measurement Real :=
  Measurement.map (to_joules cal)
    (Measurement.mapUncertainty
      (fun u => scaleUncertainty cal.joules_per_coh
                  (le_of_lt cal.joules_pos) u) m)

-- Act to SI Joule-seconds (preserving provenance)
def measure_to_joule_seconds (cal : ExternalCalibration)
    (m : Measurement Act) : Measurement Real :=
  Measurement.map (to_joule_seconds cal)
    (Measurement.mapUncertainty
      (fun u => scaleUncertainty (cal.joules_per_coh * cal.
         seconds_per_tick)
                  (mul_nonneg (le_of_lt cal.joules_pos)
                              (le_of_lt cal.seconds_pos)) u) m)
```

This ensures that when a measurement is converted to SI, its uncertainty is scaled consistently with its value. The uncertainty scaling carries the proof that the scaling factor is

non-negative, which is required for the interval bounds to remain valid.

7 Single-Anchor Calibration

This section presents the concrete single-anchor calibration protocol.

7.1 SI definitional constants

The SI 2019 revision defines the speed of light and Planck's constant as exact values. We encode these as constants (not fit parameters):

```
-- SI-definitional speed of light in meters per second (exact)
def c_SI : Real := 299792458

-- SI-definitional Planck constant h (exact, 2019 SI):
-- 6.62607015 * 10^(-34) J*s
-- Written exactly as a rational: 662607015 / 10^42
def h_SI : Real := (662607015 : Real) / ((10 : Real) ^ (42 : Nat))

-- Reduced Planck constant hbar = h/(2*pi)
def hbar_SI : Real := h_SI / (2 * Real.pi)

-- Positivity lemmas
lemma c_SI_pos : 0 < c_SI := by norm_num [c_SI]
lemma h_SI_pos : 0 < h_SI := by unfold h_SI; exact div_pos ... ...
lemma hbar_SI_pos : 0 < hbar_SI := by unfold hbar_SI; exact
   div_pos h_SI_pos ...
```

7.2 The single-anchor protocol

The protocol is explicitly tagged as `.hypothesis` because it depends on an external laboratory procedure. It includes explicit assumptions and falsifiers:

```
def tau0_seconds_protocol : Protocol :=
  { name := "calibration.single_anchor.tau0_seconds"
    summary :=
      "Single-anchor SI calibration. Supply tau0 (seconds per tick
         ) " ++
      "as the only empirical scalar. Derive meters_per_voxel via "
          ++
      "SI-defined c=299792458 and derive joules_per_coh via " ++
      "SI-defined h (hbar=h/(2*pi)) together with RS identity " ++
      "1 act = 1 coh * 1 tick. No mass-data fitting; " ++
      "this is a reporting seam only."
    status := .hypothesis
    assumptions :=
```

```
      [ "A1: tau0 (one RS tick) corresponds to a stable physical "
         ++
        "duration that can be measured in SI seconds."
      , "A2: SI definitional constants (c and h) are treated as "
         ++
        "exact conventions; they introduce no fit freedom."
      ]
    falsifiers :=
      [ "F1: Two independent anchors for tau0 (e.g. time-first vs
         " ++
        "length-first) disagree beyond stated uncertainties."
      , "F2: Under the derived calibration, the SI speed
         consistency " ++
        "check fails beyond uncertainty."
      ]
  }

-- The protocol satisfies the hygiene predicate
theorem tau0_seconds_protocol_hygienic :
    Protocol.hygienic tau0_seconds_protocol := by
  simp [Protocol.hygienic, tau0_seconds_protocol]
```

7.3 Deriving the full calibration from one scalar

Given only τ_0 in seconds, we derive all other SI conversion factors:

```
def externalCalibration_of_tau0_seconds (tau0_s : Real)
    (htau : 0 < tau0_s) : ExternalCalibration :=
  { seconds_per_tick := tau0_s
    meters_per_voxel := c_SI * tau0_s       -- derived from SI c
    joules_per_coh := hbar_SI / tau0_s      -- derived from SI
       hbar
    seconds_pos := htau
    meters_pos := mul_pos c_SI_pos htau
    joules_pos := div_pos hbar_SI_pos htau
    speed_consistent := by
      have htau_ne : tau0_s != 0 := ne_of_gt htau
      simp [c_SI, htau_ne]
  }
```

The key derivations are:

- `meters_per_voxel` $= c_{\mathrm{SI}} \times \tau_0$ (from the SI definition of the speed of light)
- `joules_per_coh` $= \hbar_{\mathrm{SI}}/\tau_0$ (from the RS identity: one action quantum = one coherence quantum $\times$ one tick)

7.4 The CalibrationCert structure

A calibration certificate bundles the single-anchor protocol, the measured scalar, and proves the required properties:

```
structure CalibrationCert where
  -- The single empirical scalar: tau0 in seconds
  tau0_seconds : Measurement Real
  -- Protocol is the canonical single-anchor protocol
  protocol_ok : tau0_seconds.protocol = tau0_seconds_protocol
  -- Positivity: tau0 must be positive
  tau0_pos : 0 < tau0_seconds.value

-- The derived ExternalCalibration associated to a certificate
def calibration (cert : CalibrationCert) : ExternalCalibration :=
  externalCalibration_of_tau0_seconds cert.tau0_seconds.value cert
     .tau0_pos

-- The protocol hygiene is inherited
theorem calibration_protocol_hygienic (cert : CalibrationCert) :
    Protocol.hygienic cert.tau0_seconds.protocol := by
  simpa [cert.protocol_ok] using tau0_seconds_protocol_hygienic
```

7.5 Internal consistency theorems

Two theorems verify that the derived calibration is internally consistent:

```
-- Under the derived calibration, 1 voxel/tick reports exactly
   c_SI
theorem c_reports_exact (cert : CalibrationCert) :
    to_m_per_s (calibration cert) c = c_SI := by
  -- This follows from the speed_consistent field plus c_in_si
  -- Note: RSNativeUnits.c = 1 (voxel/tick)
  have := c_in_si (calibration cert)
  simpa [c_SI] using this

-- Under the derived calibration, 1 act reports hbar in J*s
theorem one_act_reports_hbar (cert : CalibrationCert) :
    to_joule_seconds (calibration cert) ({ val := 1 } : Act) =
       hbar_SI := by
  -- Expand definitions and cancel tau0_s
  have htau_ne : cert.tau0_seconds.value != 0 := ne_of_gt cert.
     tau0_pos
  -- to_joule_seconds uses: (A:Real) * (joules_per_coh *
     seconds_per_tick)
  simp [to_joule_seconds, calibration,
        externalCalibration_of_tau0_seconds, htau_ne, hbar_SI]
```

These theorems ensure that the SI bridge does not accidentally introduce inconsistent scale factors.

7.6 Convenience constructor

For downstream modules and tests, a helper function builds a certificate from a chosen τ_0:

```
-- Build a certificate from a chosen tau0 (seconds per tick)
-- Real usage should supply a Protocol'd measurement record
-- coming from a declared empirical procedure
def mkCert (tau0_s : Real) (htau : 0 < tau0_s) : CalibrationCert
   :=
  { tau0_seconds :=
      { value := tau0_s
        protocol := tau0_seconds_protocol
        notes := ["Units: SI seconds per RS tick (single-anchor)
           ."] }
    protocol_ok := rfl
    tau0_pos := htau
  }
```

This is a helper for tests and examples; production usage should supply a fully documented `Measurement` record coming from a declared empirical procedure.

8 Worked Examples

The framework includes concrete worked examples that demonstrate end-to-end usage.

8.1 Example 1: Voxel energy reported in SI

This example shows the calibration seam explicitly:

1. Compute RS-native energy (`Coh`) from a `MeaningfulVoxel`
2. Report the same measurement in SI Joules via an externally supplied `ExternalCalibration`

```
-- A toy voxel: one unit photon in phase 0, vacuum elsewhere
def spikeVoxel : MeaningfulVoxel :=
  { photon := fun p => if p = 0 then Photon.unit else Photon.
     vacuum }

-- Compute RS-native energy (Coh units)
noncomputable def energyRS : Measurement Coh :=
  Catalog.VoxelMeaning.totalEnergy spikeVoxel

-- Calibration is *external*; we keep it as a parameter
noncomputable def energyJ (cal : ExternalCalibration) :
   Measurement Real :=
```

```
  Calibration.SI.measure_to_joules cal energyRS
```

The key point: `energyRS` is computed entirely in RS-native units. SI reporting happens only when an `ExternalCalibration` is supplied.

8.2 Example 2: Stream to ledger trace action (no SI)

This example shows end-to-end RS-native measurement plumbing without SI:

1. An observed Boolean stream
2. A scaffold instrument adapter producing `LedgerState` evidence
3. A derived catalog observable on a trace (`pathAction`)

```
-- A simple 8-bit window: first 4 bits are true, last 4 bits are
   false
def window4On4Off : Pattern 8 :=
  fun i => decide (i.val < 4)

-- Periodic stream repeating the 8-bit window
def stream : Stream := extendPeriodic8 window4On4Off

-- Sample starting at tick 0
def sample0 : Adapters.StreamToLedger.StreamSample8 :=
  { stream := stream, t0 := 0 }

-- Z-count and centered Z-charge are computable
#eval Adapters.StreamToLedger.zWindow sample0   -- evaluates to 4
#eval Adapters.StreamToLedger.zCharge sample0   -- evaluates to 0

-- The rest of the pipeline is noncomputable (uses phi and Real)
noncomputable def ledgerEvidence0 : Measurement LedgerState :=
  Adapters.StreamToLedger.ledgerEvidence sample0

noncomputable def cost0 : Measurement Cost :=
  Catalog.Ledger.recognitionCost ledgerEvidence0.value

noncomputable def trace3 : Measurement (List LedgerState) :=
  Adapters.StreamToLedger.traceOctaves 3 sample0

noncomputable def action3 : Measurement Cost :=
  Catalog.Ledger.pathAction trace3.value
```

This shows that measurement records can be carried through a multi-step pipeline while preserving protocol and window metadata, and while keeping the conversion and adapter steps explicit.

9 Conceptual Architecture

Layer 1: RS-native unit system
Base units: tick (τ_0), voxel (ℓ_0), with $c = 1$ voxel/tick
Derived quanta: coh $= \varphi^{-5}$, act $=$ coh $\times$ tick
All quantities organized on φ-ladder; 8-tick octave cycle

Layer 2: Measurement records
`Measurement` = value + window + protocol (assumptions, falsifiers) + uncertainty
Type-tagged quantities: `Tick`, `Voxel`, `Coh`, `Act`, `Cost`, etc.
Observable catalogs: Ledger, VoxelMeaning, Ethics, Qualia

Layer 3: Optional SI calibration seam
Provide `ExternalCalibration` (or derive from `CalibrationCert`)
Only `seconds_per_tick` is empirical; everything else is derived

Layer 4: SI report
Convert to seconds, meters, joules, joule-seconds while preserving provenance
Uncertainty scaled consistently with value

Figure 1: Four-layer architecture of the RS-native measurement framework.

10 Module Summary

The framework consists of several integrated components:

- **Core**: Protocol, Window, Measurement, Observable, Uncertainty
- **Tagged quantities**: tick, voxel, coh, act, cost, skew, meaning, qualia, Z
- **Observable catalogs**: Ledger, VoxelMeaning, Ethics, Qualia
- **Cross-agent alignment seam**: AlignmentProtocol, Alignment
- **SI calibration**: ExternalCalibration, SingleAnchor protocol
- **Adapters** (scaffold): Stream→Ledger, Voxel→WToken

11 Implementation Map and Proof Status

Table 1 maps the key manuscript claims to formal objects and their status.

12 Reproducibility and Audit Workflow

A practical audit of any RS-derived result proceeds in four steps:

1. Inspect the protocol record attached to each reported measurement.
2. Check protocol hygiene using the executable hygiene check (`Protocol.hygienicBool`) for hypothesis/scaffold protocols.
3. If SI conversion is used, identify the calibration certificate providing seconds per tick and verify that the conversion path flows only through the explicit calibration seam.
4. Confirm that any additional empirical numerals appear as explicit measurement records (calibration or validation targets), rather than being embedded implicitly in conversion logic.

This workflow scales: the same protocol discipline can be applied to particle physics, cosmology, or any other domain in the repository.

13 Limitations

Single-anchor calibration does not eliminate the need for empirical contact; it makes that contact explicit. If SI numerals are required, at least one physical identification must be made between RS-native time and external time. The contribution here is not to deny this, but to represent it as a protocol-level seam rather than a hidden parameter.

Additionally, the current implementation does not guard against non-hygienic protocol construction at compile time. Enforcement relies on audit tooling that checks the hygiene predicate (and, in practice, the executable hygiene check). Future work could add a compile-time check using a "hygienic" subtype.

14 Next Steps

The next steps are straightforward and test-driven:

- Add a two-route cross-check protocol (time-first versus length-first) that compares independent anchors and flags inconsistency.
- Expand the measurement catalogs with additional RS-native observables and standardized protocols.
- Build automated audit tooling that lists the full protocol dependency chain for any reported quantity.

15 Conclusion

We introduced a measurement framework designed for audit-friendly reporting in programs that aim to minimize adjustable parameters: protocols are explicit, falsifiers are required for hypothesis-level protocols, uncertainties are explicit, and SI reporting is separated by an auditable calibration seam. The single-anchor calibration protocol provides a concrete and practical way to connect RS-native quantities to SI reporting using one empirical scalar, while keeping the reporting seam distinct from model validation and making any additional calibration targets explicit.

The key innovation is that the measurement layer is not separate from the formal system—it is part of the same codebase, subject to the same type-checking and proof requirements. This means that claims about parameter-freedom can be audited mechanically, and any empirical input is localized and named.

Data availability

No datasets are introduced in this manuscript.

Code availability

The full implementation is contained in the Recognition Science codebase, available from the Recognition Science Research Institute upon reasonable request.

Competing interests

The authors declare no competing interests.

Acknowledgments

We thank collaborators and reviewers who emphasized the importance of protocol hygiene and explicit calibration seams for auditability.

Author contributions

Conceptualization, software, and writing: Recognition Science Research Institute.

References

1. BIPM (2019). The International System of Units (SI), 9th edition (SI Brochure).

2. Wilkinson, M. D. et al. (2016). The FAIR Guiding Principles for scientific data management and stewardship. Scientific Data 3:160018.

3. Moreau, L. and Missier, P. (eds.) (2013). PROV-DM: The PROV Data Model. W3C Recommendation.

4. Peng, R. D. (2011). Reproducible Research in Computational Science. Science 334(6060):1226–1227.

5. ATLAS Collaboration (2008). The ATLAS Experiment at the CERN Large Hadron Collider. Journal of Instrumentation 3:S08003.

6. CMS Collaboration (2008). The CMS Experiment at the CERN LHC. Journal of Instrumentation 3:S08004.

Manuscript claim	Formal object(s)	Status
RS-Native Unit System		
RS-native base units (tick, voxel, c=1)	`RSNativeUnits.tick`, `voxel`, `c`	DEFINED
Canonical RSUnits pack (U)	`RSNativeUnits.U`, `U_tau0`, `U_ell0`	DEFINED
Coherence/action quanta	`cohQuantum`, `hbarQuantum`, `massQuantum`	DEFINED
φ-ladder scaling	`phiRung`, `scaleByPhi`	DEFINED
φ-ladder algebra	`phiRung_add`, `phiRung_neg`	PROVED
8-tick octave cycle	`octavePeriod`, `octavePhase`	DEFINED
Gap-45 synchronization	`gap45`, `syncPeriod_eq_lcm`	PROVED
K-gate displays	`tau_rec`, `lambda_kin`	DEFINED
Measurement Framework Core		
Protocol status enum	`Status` (spec, derived, hypothesis, scaffold)	DEFINED
Protocol record with assumptions/falsifiers	`Protocol`	DEFINED
Hygiene predicate (Prop)	`Protocol.hygienic`	DEFINED
Hygiene check (Bool)	`Protocol.hygienicBool`	DEFINED
Time window	`Window`, `Window.instant`, `Window.stop`	DEFINED
Uncertainty semantics	`Uncertainty` (sigma, interval, discrete)	DEFINED
Uncertainty extractors	`sigmaVal`, `intervalBounds`	DEFINED
Measurement record	`Measurement`	DEFINED
Measurement transformers	`map`, `mapWithProtocol`, `mapUncertainty`	DEFINED
Observable type	`Observable`	DEFINED
Tagged quantity structure	`Quantity` with arithmetic instances	DEFINED
Unit tags (9 types)	`TickUnit`, `CohUnit`, etc.	DEFINED
Observable Catalogs		
Ledger observables (5 protocols)	`Catalog.Ledger.*`	DEFINED
VoxelMeaning observables (2 protocols)	`Catalog.VoxelMeaning.*`	DEFINED
Ethics/Qualia catalogs	`Catalog.Ethics`, `Catalog.Qualia`	DEFINED
Cross-Agent Alignment		
AlignmentProtocol with invariants	`AlignmentProtocol`	DEFINED
Alignment structure	`Alignment`, `Alignment.apply`	SCAFFOLD
SI Calibration Seam		
ExternalCalibration structure	`ExternalCalibration` (7 fields)	DEFINED
SI conversion functions (7 functions)	`to_seconds`, `to_joules`, etc.	DEFINED
Measurement-level SI conversion	`measure_to_seconds`, `measure_to_joules`, etc.	DEFINED
Speed consistency theorem	`c_in_si`	PROVED
Single-Anchor Calibration		
SI definitional constants	`c_SI`, `h_SI`, `hbar_SI`	EXTERNAL
Positivity lemmas	`c_SI_pos`, `h_SI_pos`, `hbar_SI_pos`	PROVED
Single-anchor protocol	`tau0_seconds_protocol`	HYPOTHESIS
Protocol hygiene proof	`tau0_seconds_protocol_hygienic`	PROVED
Derive calibration from τ_0	`externalCalibration_of_tau0_seconds`	DEFINED
Calibration certificate	`CalibrationCert`	DEFINED
Certificate convenience constructor	`mkCert`	DEFINED
Speed check theorem	`c_reports_exact`	PROVED
Action check theorem	`one_act_reports_hbar`	PROVED
Worked Examples		
Voxel energy to SI	`Examples.VoxelEnergyToSI`	DEFINED
Stream to ledger action	`Examples.StreamLedger`	DEFINED

Table 1: Implementation map: manuscript claims, formal definitions, and status. DEFINED = present as a definition or structure. PROVED = established by a formal theorem. HYPOTHESIS = depends on external empirical input. SCAFFOLD = interface present but not a complete solution. EXTERNAL = SI definitional constants (exact by international