Coherence-Controlled Fusion:

Tunneling as Ledger Commitment and the $T \mapsto S^2 T$ Efficiency Lever

Jonathan Washburn
Recognition Science Research
jonathan@recognitionscience.org

January 2026

Abstract

Recognition Science (RS) treats quantum "tunneling" not as a particle traversing a classically forbidden region, but as a *ledger commitment event* across an effective recognition barrier (a cost-gap between stable ledger states). In this framing, the classical Gamow suppression exponent is a coarse-grained proxy for the barrier cost; crucially, *coherence* can reduce the effective barrier, increasing commitment probability at fixed thermal conditions.

This paper formalizes an applied program for live fusion shots: define two facility-computable coherence variables—(i) φ-coherence C_φ from timing/phase alignment and interference minimization, and (ii) ledger synchronization C_σ from diagnostics→ratios→symmetry ledger—and combine them into a barrier scale

$$S = \frac{1}{1 + C_\varphi + C_\sigma} \in (0, 1].$$

Under a standard Gamow-like proxy $\eta(T) \propto 1/\sqrt{T}$, the RS-effective exponent becomes $\eta_{\mathrm{RS}}(T) = S\eta(T)$, yielding a direct temperature/drive-energy efficiency lever:

$$T_{\mathrm{needed}} = S^2\, T_{\mathrm{classical}}, \qquad T_{\mathrm{eff}} = \frac{T}{S^2}.$$

We present precise metric definitions suitable for real-time computation and certification, and a controlled run protocol: hold all non-coherence variables fixed, increase S, then step down temperature/driver energy by S^2 until yield degradation is observed. The goal is not to debate RS truth but to operationalize the coherence-control mechanism as a measurable efficiency improvement.

Contents

1 Introduction (Outline)

1.1 Problem statement

1.2 Core hypothesis (applied)

1.3 Contributions

1.4 Reading guide (how the paper is internally complete)

1.5 Summary of testable predictions and falsifiers

2 Scope and Claim Hygiene

This is an applied paper intended for a science/engineering team executing live fusion shots. We focus on *measurement, control, and calibration* of coherence variables, and on the empirical procedure to convert coherence improvements into reduced temperature/drive requirements. We do *not* provide device-construction instructions.

We separate:

- **Formal layer (Lean)**: internal logical guarantees given stated definitions/hypotheses.
- **Facility layer (applied)**: how the variables are computed from timing traces and diagnostics.
- **Empirical layer**: how we validate the bridge and quantify the efficiency gain.

Recognition Science (RS), that picture is downstream. The primitive is:

> **A transition is a ledger commitment event** selecting a stable outcome branch, and the "barrier" is an **effective recognition barrier**—a cost-gap between stable ledger states.

The practical reason we adopt this framing is that it immediately turns "make fusion easier" into a control problem: if the barrier is a cost-gap, then any mechanism that *reduces the effective cost-gap* (without destabilizing the system) increases the transition rate. The rest of the paper is about measuring and controlling precisely those cost-reducing mechanisms.

8.5 Barrier as recognition cost gap (definition-level)

Ledger states and stability. RS models physical configurations as ledger states with a well-defined recognition cost. In the fusion context, we do not attempt to model the full ledger microstate; we only require the existence of a scalar *cost* that orders states by stability. At $T_R = 0$ ("strict recognition minimization"), only zero-cost states are stable. At finite T_R, near-minima appear with nonzero probability.

Barrier as a cost-gap. Consider two stable ledger basins A and B (e.g. two recognition-consistent nuclear configurations). Any transition $A \to B$ must pass through intermediate ledger configurations; the barrier is the minimal achievable *cost elevation* above the basin floor along an admissible path. In the applied program we do not claim to know this barrier exactly; instead we work with a physically motivated *barrier-cost proxy* η such that larger η means exponentially lower commitment probability.

8.6 Rate/commitment probability: what is being modeled vs measured

What we model. We model a transition rate proxy using an exponential suppression in an effective barrier cost, in direct analogy with classical barrier-penetration formulas:

$$P(\text{commit } A \to B) \;\propto\; \exp(-\eta_{\text{eff}}).$$

What we measure. In live runs we measure yield proxies, burn histories, and diagnostics. The connection from measured quantities to η_{eff} is not assumed; it is established empirically via controlled experiments (Section "Run Protocol").

RS thermodynamic weighting. RS has a native thermodynamic layer: at finite recognition temperature T_R, costs induce Gibbs weights. The Lean formalization is explicit:

```
-- Lean excerpt: IndisputableMonolith/Thermodynamics/RecognitionThermodynamics.lean
structure RecognitionSystem where
  TR : ℝ
  TR_pos : 0 < TR

noncomputable def gibbs_weight (sys : RecognitionSystem) (x : ℝ) : ℝ :=
  exp (- Jcost x / sys.TR)
```

This is the precise meaning of the informal statement

$$p(\omega) \propto \exp\left(-\frac{\text{cost}(\omega)}{T_R}\right).$$

In other words: *cost is the ordering principle*, and probability mass is concentrated near low-cost configurations.

Commitment as a primitive operation. RS treats superposition as an uncommitted ledger entry and collapse as commitment. The measurement formalization makes the "commit" operation explicit (we will reuse the same commitment semantics for barrier transitions):

```
-- Lean excerpt: IndisputableMonolith/Quantum/Measurement/WavefunctionCollapse.lean
structure LedgerBranch (n : ℕ) where
  outcome : Fin n
  amplitude : Amplitude
  weight : ℝ
  weight_eq : weight = ‖amplitude‖^2

structure UncommittedLedger (n : ℕ) where
  branches : List (LedgerBranch n)
  normalized : (branches.map LedgerBranch.weight).sum = 1

structure CommittedLedger (n : ℕ) where
  outcome : Fin n
  amplitude : Amplitude
  unit_norm : ‖amplitude‖ = 1

noncomputable def commit {n : ℕ} (L : UncommittedLedger n) (i : Fin n)
    (_h : ∃ b ∈ L.branches, b.outcome = i) : CommittedLedger n :=
  let b := L.branches.find? (fun b => b.outcome = i)
  match b with
  | some branch =>
      if hz : branch.amplitude ≠ 0 then
        ⟨i, branch.amplitude / ‖branch.amplitude‖, norm_div_norm_eq_one branch.amplitude hz⟩
      else
        ⟨i, 1, by simp⟩
  | none => ⟨i, 1, by simp⟩
```

The conceptual move for this paper is to treat "tunneling" transitions as the same kind of commitment event, but with the branch weights governed by an effective barrier cost proxy.

8.7 Why "tunneling" is not spatial traversal in RS

Not a particle crossing a forbidden region. The spatial picture is a convenient approximation when one already assumes a potential-energy landscape and a wavefunction over position. RS inverts that: the primitive is the ledger and its admissible commitments. A "forbidden" configuration is simply one with prohibitively high recognition cost relative to available recognition temperature and coherence resources.

What survives the translation. RS keeps what is operationally real:

- exponential sensitivity of rates to an effective barrier parameter,
- interference/phase sensitivity in pre-commitment dynamics,
- and discrete commitment/collapse events.

What RS discards is the literal narrative that a localized object traverses an unphysical region.

8.8 Mapping to classical Gamow suppression (as a coarse-grained proxy)

Classical baseline proxy. In fusion and alpha decay, the classical "tunneling" rate is often summarized by a Gamow-like exponent η derived from Coulomb suppression. In this paper, we treat that classical η as a *baseline barrier-cost proxy*—a coarse-grained stand-in for the underlying recognition barrier.

RS coherence lowers the effective barrier. RS adds a second layer: coherence and ledger synchronization can reduce the effective barrier *multiplicatively*. This is exactly what we have formalized in the fusion rate model:

```
-- Lean excerpt: IndisputableMonolith/Fusion/ReactionNetworkRates.lean
structure RSCoherenceParams where
  phiCoherence : ℝ
  phiCoherence_nonneg : 0 ≤ phiCoherence
  phiCoherence_le_one : phiCoherence ≤ 1
  ledgerSync : ℝ
  ledgerSync_nonneg : 0 ≤ ledgerSync
  ledgerSync_le_one : ledgerSync ≤ 1

def rsBarrierScale (c : RSCoherenceParams) : ℝ :=
  1 / (1 + c.phiCoherence + c.ledgerSync)

theorem rsBarrierScale_le_one (c : RSCoherenceParams) : rsBarrierScale c ≤ 1 := by
  unfold rsBarrierScale
  have hden : (1 : ℝ) ≤ (1 + c.phiCoherence + c.ledgerSync) := by
    linarith [c.phiCoherence_nonneg, c.ledgerSync_nonneg]
  have h := one_div_le_one_div_of_le (by norm_num : (0 : ℝ) < 1) hden
  simpa using h

def rsGamowExponent (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) : ℝ :=
  rsBarrierScale c * gamowExponent params cfgA cfgB

theorem rsGamowExponent_le_gamowExponent (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) :
    rsGamowExponent c params cfgA cfgB ≤ gamowExponent params cfgA cfgB := by
  unfold rsGamowExponent
  have hs : rsBarrierScale c ≤ 1 := rsBarrierScale_le_one c
  have hη : 0 ≤ gamowExponent params cfgA cfgB := gamowExponent_nonneg params cfgA cfgB
  have := mul_le_mul_of_nonneg_right hs hη
  simpa using this

def rsTunnelingProbability (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) : ℝ :=
  Real.exp (-rsGamowExponent c params cfgA cfgB)

theorem rsTunnelingProbability_ge_classical (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) :
    Real.exp (-gamowExponent params cfgA cfgB) ≤ rsTunnelingProbability c params cfgA cfgB := by
  have hη : rsGamowExponent c params cfgA cfgB ≤ gamowExponent params cfgA cfgB :=
    rsGamowExponent_le_gamowExponent c params cfgA cfgB
  have hneg : (-gamowExponent params cfgA cfgB) ≤ (-rsGamowExponent c params cfgA cfgB) := by
    linarith
  have hexp : Real.exp (-gamowExponent params cfgA cfgB) ≤ Real.exp (-rsGamowExponent c params cfgA cfgB) :=
    (Real.exp_le_exp).2 hneg
  simpa [rsTunnelingProbability] using hexp
```

The key applied consequence is monotonic: increasing `phiCoherence` or `ledgerSync` cannot increase the barrier in this model; it can only reduce the effective exponent and thus increase the transition probability proxy.

In the remainder of the paper we will define facility-computable surrogates for `phiCoherence` and `ledgerSync`, and then use controlled experiments to measure how much classical temperature/drive can be traded for coherence (the $T \mapsto S^2 T$ lever).

9 Operational Coherence Variables

The RS tunneling model in the previous section introduced two abstract control levers: `phiCoherence` and `ledgerSync`. Those quantities are *not* assumed to be directly observable.

Instead, the facility needs *operational surrogates* computed from available logs and diagnostics, in real time, with a clean audit trail.

This section defines two facility-computable scalars:

$$C_\varphi \in [0,1], \qquad C_\sigma \in [0,1],$$

designed to satisfy five requirements:

1. **Monotonicity in the intended direction**: "better coherence" and "better synchronization" should strictly increase the corresponding scalar, all else equal.
2. **Boundedness and interpretability**: values lie in $[0,1]$ with clear limiting cases.
3. **Real-time computability**: the metric uses only signals available during a shot (or in immediate post-shot reduction).
4. **Graceful degradation**: missing channels or partial logs should not produce nonsense; the metric should fall back to conservative values.
5. **Auditability**: the scalar must be accompanied by intermediate components so disagreements are diagnosable (jitter vs skew vs phase alignment, etc.).

9.1 φ-Coherence C_φ (timing + phase alignment)

9.1.1 Context and RS basis

RS treats "coherence" operationally as a property of *phase-consistent ledger evolution*: phase alignment is what makes interference constructive rather than destructive, and low jitter is what makes phase alignment reproducible (not a one-off accident). In the fusion control setting, this translates to:

- **Temporal coherence**: events occur when they are supposed to occur (low timing jitter).
- **Cross-channel coherence**: channels agree on a shared phase (high phase alignment).
- **Low skew**: channel-to-channel timing offsets remain small (so phases do not shear).

We encode these as three components and multiply them, forcing all three to be high if C_φ is to be high.

9.1.2 Data sources and required signals

The required inputs per shot are:

- **Expected event times** (seconds): derived from the schedule table (what we asked the hardware to do).
- **Measured event times** (seconds): derived from timing logs (what the hardware did).
- **Channel phases** (radians): per channel, measured at the relevant window (for example: at/near peak compression or the burn-relevant sub-window).
- **Optional channel time offsets** (seconds): per-channel timing offsets relative to a reference channel (or a common trigger).
- **Two scale parameters**: `jitterScale` and `skewScale` setting the scale at which timing errors should significantly reduce coherence.

9.1.3 Definition: timing-jitter term (RMS error)

We compute the pairwise timing error between expected and measured times, then take RMS. The implementation intentionally truncates to the shorter list to remain robust to partial logs.

9.1.4 Definition: phase alignment term (mean resultant length)

For phases $\theta_1, \ldots, \theta_n$, define the mean resultant length

$$R := \frac{1}{n}\sqrt{\left(\sum_{k=1}^{n}\cos\theta_k\right)^2 + \left(\sum_{k=1}^{n}\sin\theta_k\right)^2} \in [0,1].$$

This is a standard circular-statistics coherence measure: $R = 1$ means perfect alignment, and $R \approx 0$ means phases are effectively random.

9.1.5 Definition: cross-channel skew term

We compute RMS over the optional channel offsets list. If no offsets are provided, skew defaults to 0 and the skew score defaults to 1 (i.e. "no evidence of skew" is treated as neutral).

9.1.6 Normalization, scoring, and bounds

We convert RMS errors to bounded scores using a quadratic penalty:

$$\texttt{timingScore} = \frac{1}{1 + (j_{\mathrm{rms}}/s_j)^2}, \qquad \texttt{skewScore} = \frac{1}{1 + (s_{\mathrm{rms}}/s_s)^2},$$

then combine:

$$C_\varphi := \mathrm{clamp}_{[0,1]}\big(R \cdot \texttt{timingScore} \cdot \texttt{skewScore}\big).$$

The quadratic form is deliberate: it punishes large violations strongly while not overreacting to small measurement noise; it also matches the intended "jitter robustness" scaling used elsewhere in the project.

9.1.7 Lean formalization (executable interface)

The facility-facing implementation is written as an executable Lean module (Float-level), so the same computation can be re-run deterministically for audit:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
def clamp01 (x : Float) : Float :=
  if x < 0.0 then 0.0 else if x > 1.0 then 1.0 else x

def rms (xs : List Float) : Float :=
  match xs.length with
  | 0 => 0.0
  | n =>
      let meanSq := xs.foldl (fun acc x => acc + x * x) 0.0 / n.toFloat
      Float.sqrt meanSq

def timingErrors (expected measured : List Float) : List Float :=
  match expected, measured with
  | e :: es, m :: ms => (m - e) :: timingErrors es ms
  | _, _ => []

def meanResultantLength (phases : List Float) : Float :=
```

```
  match phases.length with
  | 0 => 0.0
  | n =>
      let cosSum := phases.foldl (fun acc θ => acc + Float.cos θ) 0.0
      let sinSum := phases.foldl (fun acc θ => acc + Float.sin θ) 0.0
      clamp01 (Float.sqrt (cosSum * cosSum + sinSum * sinSum) / n.toFloat)

structure PhiCoherenceInput where
  expectedTimes : List Float
  measuredTimes : List Float
  channelPhases : List Float
  channelTimeOffsets : List Float := []
  jitterScale : Float := 1e-12
  skewScale : Float := 1e-12

structure PhiCoherenceOutput where
  jitterRMS : Float
  skewRMS : Float
  phaseAlignment : Float
  phiCoherence : Float

def computePhiCoherence (input : PhiCoherenceInput) : PhiCoherenceOutput :=
  let errs := timingErrors input.expectedTimes input.measuredTimes
  let jrms := rms errs
  let srms := rms input.channelTimeOffsets
  let phaseAlign := meanResultantLength input.channelPhases
  let timingScore :=
    if input.jitterScale > 0 then
      1.0 / (1.0 + (jrms / input.jitterScale) * (jrms / input.jitterScale))
    else 0.0
  let skewScore :=
    if input.skewScale > 0 then
      1.0 / (1.0 + (srms / input.skewScale) * (srms / input.skewScale))
    else 0.0
  let phiC := clamp01 (phaseAlign * timingScore * skewScore)
  ⟨jrms, srms, phaseAlign, phiC⟩
```

9.1.8 Uncertainty propagation and stability under missing data

We do *not* attempt to propagate full covariance structures in this paper. Instead, we require that each shot log the intermediate components `jitterRMS`, `skewRMS`, and `phaseAlignment` alongside C_φ. This makes uncertainty and failure modes explicit: if C_φ is low, we can attribute it to (i) timing jitter, (ii) skew, or (iii) phase disorder.

9.1.9 Control objectives: what it means to "increase C_φ"

Operationally, "increase C_φ" means:

- reduce `jitterRMS` at fixed schedule (*timing stabilization*),
- reduce `skewRMS` across channels (*cross-channel sync*),
- increase `phaseAlignment` at the burn-relevant window (*phase locking*).

In RS terms, this is the facility's attempt to create an uncommitted ledger evolution with minimal destructive interference prior to commitment.

9.2 Ledger Synchronization C_σ (diagnostics → ratios → ledger)

9.2.1 Context and RS basis

In RS, ledger structure is double-entry and conservation-like: stable evolution requires that imbalances are minimized or cycle-cancelled. In ICF-style fusion control, the most salient macroscopic manifestation of "ledger imbalance" is asymmetry in the implosion/burn geometry. We therefore define a ledger-like scalar from diagnostic mode ratios, and convert it into a dimensionless synchronization score $C_\sigma \in [0, 1]$.

9.2.2 Diagnostics-to-ratios calibration model

The facility observes raw diagnostic values (e.g. mode amplitudes for $P_2, P_4, \dots$) and must map them to *dimensionless ratios* where the ideal value maps to 1. The general bridge structure (with calibration versioning and uncertainty) is:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure Calibration where
  version : String
  toRatio : DiagnosticMode → ℝ → ℝ
  monotone : ∀ m x y, x ≤ y → toRatio m x ≤ toRatio m y
  ideal_maps_to_one : ∀ m, toRatio m 0 = 1
  uncertainty : ℝ
  uncertainty_pos : 0 < uncertainty
  uncertainty_small : uncertainty ≤ 0.1

def measurementToRatios (cfg : BridgeConfig) (meas : DiagnosticMeasurement) :
    DiagnosticMode → ℝ :=
  fun m => cfg.calibration.toRatio m (meas.rawValues m)

def diagnosticLedger (cfg : BridgeConfig) (meas : DiagnosticMeasurement) : ℝ :=
  cfg.modes.foldl (fun acc m =>
    acc + cfg.weights m * Cost.Jcost (measurementToRatios cfg meas m)
  ) 0
```

For live operations, we use a minimal affine calibration model (ratio $= 1 + g \cdot \texttt{raw}$) when a full calibration map is not yet available.

9.2.3 Ledger computation (weights, ratios, J)

Given weights $w_i > 0$ and ratios r_i, define the executable ledger value

$$\texttt{ledgerValue} = \sum_i w_i \, J(r_i).$$

At the executable layer, J is implemented as the Float analog of $J(x) = (x + 1/x)/2 - 1$ for $x > 0$.

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure LedgerInput where
  weights : List Float
  ratios : List Float

structure LedgerOutput where
  ledgerValue : Float
  passed : Bool
  threshold : Float

def jCostFloat (x : Float) : Float :=
  if x > 0 then (x + 1/x) / 2 - 1 else 0

def computeLedger (input : LedgerInput) (threshold : Float) : LedgerOutput :=
```

```
  let pairs := input.weights.zip input.ratios
  let ledgerVal := pairs.foldl (fun acc (w, r) => acc + w * jCostFloat r) 0.0
  ⟨ledgerVal, ledgerVal ≤ threshold, threshold⟩
```

9.2.4 Normalization to $[0, 1]$ and interpretation

We map `ledgerValue` to a dimensionless synchronization score:

$$C_\sigma := \mathrm{clamp}_{[0,1]}\left(\frac{1}{1 + \texttt{ledgerValue}/\Lambda}\right),$$

where $\Lambda > 0$ sets the scale at which ledger deviations become "significant". This map has the correct limiting behavior: `ledgerValue`$= 0$ gives $C_\sigma = 1$, and very large `ledgerValue` drives $C_\sigma \to 0$.

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure AffineRatioCalibration where
  gains : List Float

def applyAffineCalibration (cal : AffineRatioCalibration) (raw : List Float) : List Float :=
  match cal.gains, raw with
  | g :: gs, x :: xs => (1.0 + g * x) :: applyAffineCalibration ⟨gs⟩ xs
  | _, _ => []

structure LedgerSyncInput where
  weights : List Float
  ratios : List Float := []
  rawValues : List Float := []
  calibration : AffineRatioCalibration := ⟨[]⟩
  threshold : Float := 0.1
  ledgerScale : Float := 0.1

structure LedgerSyncOutput where
  ledgerValue : Float
  passed : Bool
  ledgerSync : Float

def computeLedgerSync (input : LedgerSyncInput) : LedgerSyncOutput :=
  let ratios :=
    if input.ratios.isEmpty then
      applyAffineCalibration input.calibration input.rawValues
    else
      input.ratios
  let ledgerOut := computeLedger ⟨input.weights, ratios⟩ input.threshold
  let Λ := input.ledgerScale
  let sync :=
    if Λ > 0 then clamp01 (1.0 / (1.0 + ledgerOut.ledgerValue / Λ)) else 0.0
  ⟨ledgerOut.ledgerValue, ledgerOut.passed, sync⟩
```

9.2.5 Uncertainty, drift, and calibration versioning

Unlike C_φ, `ledgerSync` depends heavily on diagnostic calibration. Therefore every shot must carry:

- a **calibration version string** (so ratios are reproducible),
- a stated **uncertainty envelope** for the calibration (fractional),
- and the raw diagnostic values used to compute the ratios.

This turns calibration into an explicit seam rather than an implicit source of irreproducibility.

9.2.6 Control objectives: what it means to "increase C_σ"

Operationally, "increase C_σ" means decreasing `ledgerValue` while staying within the facility's PASS envelope. In RS terms, we are driving the macroscopic dynamics toward a more balanced ledger configuration, which is precisely what the barrier reduction model treats as synchronization.

10 Barrier Scale and Temperature Efficiency Lever

The previous section defined two facility-level scalars C_φ and C_σ. This section defines how those scalars enter the *effective recognition barrier* and explains the central applied consequence:

> **Coherence can be traded for temperature/drive.** Increasing coherence reduces the effective barrier exponent, and under a standard $\eta(T) \propto 1/\sqrt{T}$ proxy this implies a concrete scaling $T_{\text{needed}} = S^2 T_{\text{classical}}$.

The value of stating the result in this form is operational: it gives an explicit step-down rule for live runs. You do not need to guess how much to reduce drive energy after improving coherence; you can compute S shot-by-shot and apply the predicted S^2 factor, then measure when yield begins to degrade.

10.1 Barrier scale design space

10.1.1 Why choose $S = 1/(1 + C_\varphi + C_\sigma)$

We define a *barrier scale* S that multiplies a baseline classical barrier proxy η:

$$\eta_{\text{RS}} := S\,\eta.$$

Here S is *not* a probability; it is a dimensionless multiplier. The model is constructed so that:

- $S = 1$ means **no RS barrier reduction** (completely incoherent/unsynchronized).
- Larger coherence/synchronization should **reduce** the effective barrier, i.e. make S **smaller**.
- The map should be bounded and saturating: coherence can help a lot but cannot drive the barrier negative.

The simplest monotone bounded choice that is symmetric in the two levers and has the correct limiting behavior is:

$$S(C_\varphi, C_\sigma) = \frac{1}{1 + C_\varphi + C_\sigma}.$$

This choice is not claimed to be uniquely forced by mathematics; it is an *applied model choice* with three pragmatic benefits:

1. **Monotonic and bounded**: $S \in (0, 1]$ automatically, with no tuning.
2. **Additive contributions**: C_φ and C_σ contribute additively inside the denominator, which is identifiable experimentally by varying one while holding the other fixed.
3. **Direct temperature scaling**: the algebra leads to a clean S^2 temperature/drive trade (derived below).

10.1.2 Alternative monotone forms and identifiability

Any monotone bounded map with $S(0,0) = 1$ could be used (e.g. $\exp(-k(C_\varphi + C_\sigma))$ or $1/(1 + k_1 C_\varphi + k_2 C_\sigma)$). Empirically, the choice is determined by which functional form best linearizes the relationship between measured yield and the inferred effective exponent. For this paper, we intentionally choose the lowest-complexity form and treat deviations as empirical signal (to be captured by calibration updates rather than by speculative functional inflation).

10.1.3 Bounds, pathological inputs, and clamping policies

At the facility interface layer, inputs are clamped to $[0, 1]$ and the denominator is guarded so that malformed inputs cannot create negative or infinite barrier scales. This is an engineering safety requirement: it ensures the run-control system never receives undefined values.

10.1.4 Lean formalization (executable barrier scale and S^2 lever)

The executable implementation used for run-time computation is:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure RSCoherenceInput where
  phiCoherence : Float
  ledgerSync : Float

structure RSBarrierScaleOutput where
  barrierScale : Float

def computeRSBarrierScale (input : RSCoherenceInput) : RSBarrierScaleOutput :=
  let denom := 1.0 + input.phiCoherence + input.ledgerSync
  let s := if denom > 0 then 1.0 / denom else 1.0
  ⟨s⟩

def temperatureScaleFromBarrier (barrierScale : Float) : Float :=
  barrierScale * barrierScale

def effectiveTemperatureGain (barrierScale : Float) : Float :=
  let s2 := barrierScale * barrierScale
  if s2 > 0 then 1.0 / s2 else 0.0

structure RSShotEfficiencyOutput where
  phiCoherence : Float
  ledgerSync : Float
  barrierScale : Float
  temperatureScale : Float   -- S^2
  effectiveTempGain : Float  -- 1/S^2

def computeRSShotEfficiency (phiC : Float) (ledgerS : Float) : RSShotEfficiencyOutput :=
  let s := (computeRSBarrierScale ⟨phiC, ledgerS⟩).barrierScale
  let s2 := temperatureScaleFromBarrier s
  let gain := effectiveTemperatureGain s
  ⟨phiC, ledgerS, s, s2, gain⟩
```

10.2 Effective exponent and probability proxy

10.2.1 Baseline barrier proxy η and RS-effective exponent

Let η denote any classical barrier-cost proxy that enters reaction rates exponentially (for example, a Gamow-like exponent for Coulomb suppression). RS uses the *same* baseline proxy but modifies it by a coherence-dependent multiplier:

$$\eta_{\mathrm{RS}} = S(C_\varphi, C_\sigma)\, \eta.$$

This is the applied statement that coherence reduces the effective recognition barrier. It is precisely the same mathematical structure we formalized in the fusion reaction network model, where η is `gamowExponent` and η_{RS} is `rsGamowExponent`:

```
-- Lean excerpt: IndisputableMonolith/Fusion/ReactionNetworkRates.lean
structure RSCoherenceParams where
  phiCoherence : ℝ
  phiCoherence_nonneg : 0 ≤ phiCoherence
  phiCoherence_le_one : phiCoherence ≤ 1
  ledgerSync : ℝ
  ledgerSync_nonneg : 0 ≤ ledgerSync
  ledgerSync_le_one : ledgerSync ≤ 1

def rsBarrierScale (c : RSCoherenceParams) : ℝ :=
  1 / (1 + c.phiCoherence + c.ledgerSync)

def rsGamowExponent (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) : ℝ :=
  rsBarrierScale c * gamowExponent params cfgA cfgB

def rsTunnelingProbability (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) : ℝ :=
  Real.exp (-rsGamowExponent c params cfgA cfgB)
```

10.2.2 What the exponential means in RS

In RS terms, the exponential is not "wavefunction leakage"; it is a compact way of stating that *commitment probability concentrates on low-cost paths*. When the effective barrier cost is reduced, the ledger can commit with higher probability per unit opportunity (per unit time, per unit attempt, or per unit exposure of the relevant state manifold). The exact meaning of "unit attempt" is facility-dependent; the prediction is about monotone ordering and scaling.

10.3 Temperature trade

10.3.1 Why $\eta(T) \propto 1/\sqrt{T}$ is the right first-order proxy

In a wide class of barrier-limited processes (including Coulomb-suppressed fusion in the nonresonant regime), the dominant temperature dependence of the suppression exponent scales approximately like

$$\eta(T) \approx \frac{K}{\sqrt{T}},$$

for some effective constant K (absorbing charges, reduced mass, and other slowly varying terms). This is not a claim that cross sections are exactly $K/\sqrt{T}$; it is a claim about the leading dependence of the barrier exponent entering the exponential weight.

10.3.2 Derivation of the $T \mapsto S^2 T$ lever

Assume the baseline barrier proxy satisfies $\eta(T) = K/\sqrt{T}$. Then the RS-effective exponent is

$$\eta_{\mathrm{RS}}(T) = S\frac{K}{\sqrt{T}}.$$

Define an *effective classical temperature* T_{eff} that would yield the same exponent without RS barrier reduction:

$$\eta(T_{\mathrm{eff}}) = \eta_{\mathrm{RS}}(T).$$

Substitute $\eta(T) = K/\sqrt{T}$:

$$\frac{K}{\sqrt{T_{\text{eff}}}} = S\frac{K}{\sqrt{T}} \quad \Longrightarrow \quad \sqrt{T_{\text{eff}}} = \frac{\sqrt{T}}{S} \quad \Longrightarrow \quad T_{\text{eff}} = \frac{T}{S^2}.$$

Equivalently, to match a classical run at temperature $T_{\text{classical}}$ while applying RS barrier reduction S, you can reduce temperature to:

$$T_{\text{needed}} = S^2\, T_{\text{classical}}.$$

10.3.3 Interpretation and magnitude

Two equivalent ways to use the formula operationally:

- **Effective-temperature view**: at a fixed physical temperature T, raising coherence (reducing S) increases T_{eff} as $1/S^2$.
- **Step-down view**: after increasing coherence to achieve S, reduce temperature/drive by S^2 and test whether yield holds.

Example: if $C_\varphi = C_\sigma = 1$, then $S = 1/3$, so $T_{\text{needed}} = (1/9)\, T_{\text{classical}}$ and $T_{\text{eff}} = 9T$.

10.3.4 What would falsify this scaling

Because this is an applied scaling law (not an identity), falsification is straightforward: if controlled experiments show that yield does *not* remain approximately invariant under the predicted step-down $T \mapsto S^2 T$ when S is increased (and all other knobs are held fixed within uncertainty), then either (i) $\eta(T) \propto 1/\sqrt{T}$ is not the correct dominant proxy for the regime, (ii) the operational coherence metrics do not track the true RS barrier reduction, or (iii) the barrier reduction is not multiplicative in the way modeled. Each failure mode is actionable and leads to a specific refinement of the experimental program.

10.4 Predicted efficiency surfaces

10.4.1 Holding classical knobs fixed: what must remain constant

For the S^2 scaling to be interpretable, the experiment must hold fixed (as tightly as the facility allows) the non-coherence variables that also affect yield: target family, geometry, fill, compression, and baseline drive shape. If these drift with S, the observed effect will be confounded.

10.4.2 Trading S vs temperature/drive energy

At the most basic level, this section predicts a one-parameter family of equivalence classes:

$$(T, S) \sim (T', 1) \quad \text{if} \quad T' = \frac{T}{S^2}.$$

In practice, we do not assume perfect equivalence; we use it as a *control law* to choose the next shot's temperature/drive step after measuring coherence improvements in the current shot.

11 Run Protocol: Turning Coherence into Measured Temperature Efficiency

This paper makes a sharp applied claim: once we can compute C_φ and C_σ reliably, we can compute S and *use it as a control variable*. The claim is not complete until we describe how a facility turns that variable into measured efficiency. That is the purpose of this section.

The run protocol is designed to do two things simultaneously:

1. **Identify causal effect**: show that increasing coherence (as measured by C_φ, C_σ) produces a predictable shift in yield/ignition behavior at fixed classical conditions.
2. **Convert effect into efficiency**: use the predicted S^2 scaling to reduce temperature/drive and determine the practical operating envelope.

We emphasize that the protocol is *scientific*: it is constructed to distinguish RS-driven effects from confounds. It is not a device-construction recipe.

11.1 Design principles for causal attribution

11.1.1 Confound control: why randomization is mandatory

Fusion shots are high-variance. Many slow variables drift: alignment, calibration, hardware conditioning, target variability, and environment. If we simply "turn on coherence control" and observe higher yield later, we will have learned nothing. Therefore, within any run block we must randomize the order of conditions so that drift averages out.

The minimum design is an A/B randomized block:

- **A (baseline)**: nominal schedule/control (no coherence pumping beyond baseline).
- **B (coherence-enhanced)**: modified schedule/control intended to increase C_φ and/or C_σ.

Within a block of $2N$ shots, randomly permute the N baseline and N enhanced conditions.

11.1.2 Negative controls ("break the mechanism")

To validate that the mechanism is actually coherence-mediated, include negative controls that target the coherence pathways directly:

- **Timing shuffle**: deliberately decorrelate event timing relative to the expected schedule. This should reduce C_φ while leaving classical energy roughly unchanged.
- **Phase detune**: deliberately reduce channel phase alignment at the burn-relevant window. This should reduce C_φ primarily through the phase-alignment term.
- **Symmetry scramble**: introduce controlled asymmetry (or stop symmetry corrections) to increase ledger value and reduce C_σ.

If these controls reduce S in the *wrong* direction (i.e. do not reduce coherence metrics, or do not reduce yield as predicted), the bridge from facility signals to coherence variables must be reconsidered.

11.1.3 Safety envelopes and abort criteria

Any coherence experiment must remain within safety/operability envelopes. In the RS formalism, that envelope is expressed as a certificate-style PASS predicate based on the symmetry ledger and per-mode bounds. The (Lean) certificate structure ties together the scheduler and the ledger:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Certificate.lean
structure Certificate (Actuator Mode : Type _) [Fintype Mode] [DecidableEq Mode]
    (L : ℕ) where
  scheduler : PhiScheduler Actuator L
  ledgerCfg : LedgerConfig (Mode := Mode)
  bounds : ModeThresholds (Mode := Mode)
  ledgerThreshold : ℝ
  ledgerThreshold_nonneg : 0 ≤ ledgerThreshold

namespace Certificate
variable (cert : Certificate (Actuator := Actuator) (Mode := Mode) L)

def passes (ratios : ModeRatios (Mode := Mode)) : Prop :=
  Fusion.pass cert.ledgerCfg cert.bounds cert.ledgerThreshold ratios
```

Operationally, this means: the run protocol may vary coherence controls only within a region where ledger-based PASS criteria remain satisfied (or where abort criteria are triggered).

11.2 Shot record schema and auditability

11.2.1 Why the shot record is part of the scientific claim

Because C_φ and C_σ are computed from facility logs, the scientific unit of analysis is the *shot record*, not just the final yield. Every shot must be replayable: a third party should be able to recompute C_φ, C_σ, and S from the recorded inputs.

11.2.2 Scheduler trace: what must be logged

The schedule is not just "a planned waveform"; it is a sequence of events with timestamps, actuator assignments, and jitter compliance. The scheduler formalization makes explicit what must be recorded:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Scheduler.lean
structure PhiScheduler (Actuator : Type _) (L : ℕ)
    extends PhiWindowSpec L where
  assignment : Actuator → Finset (Fin L)
  jitterBound : ℝ
  jitter_nonneg : 0 ≤ jitterBound

namespace PhiScheduler
structure Update (Actuator : Type _) (L : ℕ) where
  actuator : Actuator
  window : Fin L
  timestamp : ℝ

def respectsAssignment (trace : List (Update Actuator L)) : Prop :=
  ∀ e ∈ trace, (e.window ∈ (sched.assignment e.actuator))

def jitterBounded (trace : List (Update Actuator L)) : Prop :=
  ∀ u v, (u, v) ∈ trace.zipWith Prod.mk trace.tail →
    |v.timestamp - u.timestamp| ≤ sched.jitterBound
```

In practice, the facility logs `Update` events. The paper's requirement is simple: the logged trace must be sufficient to validate (i) assignment compliance and (ii) jitter boundedness.

11.2.3 Diagnostics record: raw values, calibration version, ratios

For ledgerSync we require a diagnostic record that includes raw values and the calibration version used to map raw values to ratios. (The calibration is a first-class seam; it is not hidden.)

11.2.4 Metric record: intermediate components

For C_φ, we require intermediate components `jitterRMS`, `skewRMS`, and `phaseAlignment`. For C_σ, we require `ledgerValue` and the parameters used for normalization (weights, threshold, ledgerScale).

11.2.5 Certificate bundles for reproducibility

The executable interface provides a simple JSON-like certificate bundle representation. This is how the facility exports the computation in a structured and replayable format:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure CertificateBundle where
  moduleName : String
  version : String := "1.0"
  timestamp : String
  inputHash : String
  outputs : List (String × String)
  passed : Bool
  theoremRef : String

def certifyPhiCoherence (input : PhiCoherenceInput) : CertificateBundle :=
  let out := computePhiCoherence input
  ⟨"PhiCoherence", "1.0", "2026-01-21",
   s!"n={input.measuredTimes.length},channels={input.channelPhases.length}",
   [("jitterRMS", toString out.jitterRMS),
    ("skewRMS", toString out.skewRMS),
    ("phaseAlignment", toString out.phaseAlignment),
    ("phiCoherence", toString out.phiCoherence)],
   out.phiCoherence ≥ 0.0,
   "Executable metric (facility-calibrated)"⟩

def certifyLedgerSync (input : LedgerSyncInput) : CertificateBundle :=
  let out := computeLedgerSync input
  ⟨"LedgerSync", "1.0", "2026-01-21",
   s!"modes={input.weights.length}",
   [("ledgerValue", toString out.ledgerValue),
    ("passed", toString out.passed),
    ("ledgerSync", toString out.ledgerSync)],
   out.ledgerSync ≥ 0.0,
   "Executable metric derived from ledger"⟩
```

11.3 Controlled step-down experiment (the efficiency move)

11.3.1 Goal and invariants

Goal: quantify how much temperature/drive energy can be reduced without yield loss, as a function of S.

Invariants: in a step-down block we hold fixed (as tightly as possible) the classical non-coherence variables that also influence yield: target family, geometry, fill, compression, and baseline drive shape.

11.3.2 Procedure

The procedure is:

1. **Baseline characterization**: run a randomized A/B block at fixed temperature/drive to estimate how much S can be improved by coherence controls alone.

2. **Compute** S: for each shot, compute C_φ, C_σ, S and record them with certificate bundles.

3. **Apply the predicted step**: once a stable improved S is achieved, reduce temperature/drive by the factor S^2 (Section "Barrier Scale").

4. **Stop at degradation**: continue stepping down until yield proxy drops outside a pre-registered tolerance band; record the break point.

This converts the theory into an empirical efficiency curve: maximal achievable step-down as a function of realized S.

11.4 Minimal analysis plan (explicit, robust)

11.4.1 Primary endpoints

Choose one primary yield proxy Y (facility-specific), and treat all other signals as secondary. The protocol is robust only if endpoints are pre-registered and not post-hoc selected.

11.4.2 Core scaling check

If the dominant barrier proxy behaves as $\eta(T) \approx K/\sqrt{T}$ and the RS effect is multiplicative, then yield should be approximately invariant under the transformation $T \mapsto S^2 T$ at fixed S (within noise). Empirically, this can be tested by comparing (baseline) and (coherence-improved + step-down) conditions matched by the predicted effective temperature.

11.4.3 Robustness checks

Use negative controls (timing shuffle, phase detune, symmetry scramble) to confirm that the effect tracks the coherence variables rather than drifting classical parameters.

12 Implementation Notes and Code References

This section explains how to implement the paper's core ideas as an auditable facility workflow. The guiding principle is **separation of concerns**:

- **Model layer (math, $\mathbb{R}$)**: the definitions and monotone theorems that specify what the RS mechanism *means*. This layer is where we state barrier scaling and the directionality claims.
- **Executable layer (facility, Float)**: deterministic computations on shot logs producing C_φ, C_σ, S, and S^2.
- **Bridge layer (diagnostics & calibration)**: how raw diagnostics become ratios and then a ledger value with uncertainty/version tracking.
- **Certificate layer (audit)**: how to package computations so that every shot is replayable and reviewable.

The paper is scientifically meaningful only if the implementation makes it impossible to "hand-wave" coherence: the computation must be explicit, logged, and re-runable.

12.1 Core RS primitive: the unique cost functional

Everything downstream is built on the unique symmetric convex cost:

```
-- Lean excerpt: IndisputableMonolith/Cost.lean
namespace IndisputableMonolith
namespace Cost

noncomputable def Jcost (x : ℝ) : ℝ := (x + x⁻¹) / 2 - 1

lemma Jcost_unit0 : Jcost 1 = 0 := by
  simp [Jcost]
```

This is the reason the implementation uses `Jcost` everywhere: it is the canonical recognition cost (T5 uniqueness), so it is the correct object to use for ledgers, asymmetry costs, and barrier-cost proxies.

12.2 Symmetry ledger and PASS predicate (what "safe to run" means)

The certificate layer used in the run protocol ultimately reduces to two checks:

1. a global bound on the **ledger value** (sum of weighted per-mode costs),
2. and per-mode upper bounds on ratios (**withinThresholds**).

These are defined at the math layer as:

```
-- Lean excerpt: IndisputableMonolith/Fusion/SymmetryLedger.lean
structure LedgerConfig where
  weights : Mode → ℝ
  weights_nonneg : ∀ m, 0 ≤ weights m

structure ModeRatios where
  ratio : Mode → ℝ
  ratio_pos : ∀ m, 0 < ratio m

def ledger (cfg : LedgerConfig (Mode := Mode)) (r : ModeRatios (Mode := Mode)) : ℝ :=
  ∑ m, cfg.weights m * Cost.Jcost (r.ratio m)

structure ModeThresholds where
  upper : Mode → ℝ
  upper_nonneg : ∀ m, 0 ≤ upper m

def withinThresholds (bounds : ModeThresholds (Mode := Mode))
    (r : ModeRatios (Mode := Mode)) : Prop :=
  ∀ m, r.ratio m ≤ bounds.upper m

def pass (cfg : LedgerConfig (Mode := Mode))
    (bounds : ModeThresholds (Mode := Mode)) (Λ : ℝ)
    (r : ModeRatios (Mode := Mode)) : Prop :=
  ledger cfg r ≤ Λ ∧ withinThresholds bounds r
```

This is the formal meaning of "PASS" in the fusion-control stack. In practice, the facility computes the Float-level ledger value (Section "Operational Coherence Variables") for speed, but the semantics are the same: the run is considered within envelope if the ledger is below a declared threshold and each mode ratio remains within a per-mode tolerance.

12.3 Diagnostics bridge (how raw measurements become ratios)

The diagnostics bridge exists because experimental systems do not directly output dimensionless ratios. Instead, they output raw mode amplitudes or deviations. The bridge formalization makes the calibration seam explicit and versioned, so the analysis cannot silently change later:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure Calibration where
  version : String
  toRatio : DiagnosticMode → ℝ → ℝ
  monotone : ∀ m x y, x ≤ y → toRatio m x ≤ toRatio m y
  ideal_maps_to_one : ∀ m, toRatio m 0 = 1
  uncertainty : ℝ
  uncertainty_pos : 0 < uncertainty
  uncertainty_small : uncertainty ≤ 0.1

def diagnosticLedger (cfg : BridgeConfig) (meas : DiagnosticMeasurement) : ℝ :=
  cfg.modes.foldl (fun acc m =>
    acc + cfg.weights m * Cost.Jcost (measurementToRatios cfg meas m)
  ) 0
```

Operational implication: every run analysis must include the calibration version and its stated uncertainty, otherwise C_σ is not interpretable.

12.4 Executable metrics and certificate bundles (what ships to the facility)

The facility does not need $\mathbb{R}$-level proofs in real time; it needs deterministic metrics computed from logs. The executable layer defines stable I/O structures and computations. The key artifact is a **certificate bundle** that makes the computation auditable:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure CertificateBundle where
  moduleName : String
  version : String := "1.0"
  timestamp : String
  inputHash : String
  outputs : List (String × String)
  passed : Bool
  theoremRef : String
```

Each shot should emit certificate bundles for:

- C_φ (including `jitterRMS`, `skewRMS`, `phaseAlignment`),
- C_σ (including `ledgerValue`, weights/threshold parameters),
- S and S^2 (temperature step-down factor).

12.5 Certification artifacts (what must be logged and why)

12.5.1 Required fields

At minimum, the certificate artifacts must contain:

- **Input hash**: a deterministic hash of the raw inputs used to compute the metric.
- **Outputs**: not only the scalar but also intermediate components (to localize failures).
- **Versioning identifiers**: schedule hash, calibration version, and software version.
- **PASS/FAIL**: whether the shot is within declared envelopes (ledger threshold and per-mode bounds).

12.5.2 Versioning: schedule hash + calibration version

The same nominal run can produce different computed coherence if the facility changes a calibration, a timing extraction rule, or a schedule table. Therefore, (i) the schedule table (or its hash), and (ii) the calibration version string are mandatory.

12.5.3 Deterministic replay and audit

The point of using executable Lean definitions is reproducible replay: given the logged inputs, every metric can be recomputed exactly, eliminating disputes about post-processing.

12.5.4 Auditability and failure modes

This architecture localizes failure:

- If yield drops but C_φ is high, inspect `phaseAlignment` vs jitter/skew to see whether the "high" coherence is coming from timing rather than phase.
- If C_σ drifts, the calibration version and uncertainty are explicit so the drift can be attributed to physics vs calibration changes.
- If a shot violates safety envelopes, the PASS predicate decomposes into a ledger bound and per-mode bounds, allowing immediate diagnosis of which mode triggered abort.

13 Limitations and Calibration Seams

This paper is intentionally explicit about what is *structurally forced* by the RS framework versus what is an *empirical seam* that must be calibrated at a particular facility. The difference matters operationally: forced structure determines what we should *control* and what monotone relationships must hold; seams determine what we must *measure, fit, and validate*.

13.1 Forced structure vs calibrated seams

13.1.1 What is forced (directionality)

RS forces the directionality claim:

> Increasing coherence and synchronization reduces the effective recognition barrier cost, and thus weakly increases commitment probability at fixed classical conditions.

This paper encodes that directionality by construction (multiplicative barrier reduction) and uses formal monotonicity statements at the model layer (Section "RS Framing" and "Barrier Scale"). Operationally: if our computed C_φ and C_σ increase, the effective barrier exponent should not increase.

13.1.2 What is not forced (functional form and mapping)

Two parts are **not** forced by mathematics:

1. The exact *functional form* of $S(C_\varphi, C_\sigma)$.
2. The *mapping from facility signals* (timing logs, diagnostic raw values) to the operational scalars C_φ and C_σ.

These are seams. They must be established by calibration and validation experiments (including negative controls) and may be revised as the facility learns.

13.2 Facility seams (measurement and calibration)

13.2.1 Seam A: extracting phases and timestamps

The C_φ computation requires phase angles and measured timestamps. The paper does not assume a unique method for phase extraction (Hilbert transform, PLL phase readout, heterodyne, etc.); it only requires that the method be consistent, logged, and stable under replay.

13.2.2 Seam B: scale parameters (jitterScale, skewScale)

The mapping from RMS timing errors to the bounded score is controlled by scale parameters. Those parameters are explicit in the executable interface, meaning they are part of the calibration surface:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure PhiCoherenceInput where
  expectedTimes : List Float
  measuredTimes : List Float
  channelPhases : List Float
  channelTimeOffsets : List Float := []
  jitterScale : Float := 1e-12
  skewScale : Float := 1e-12
```

Interpretation: the facility must choose a scale (or a schedule-dependent scale) such that the resulting C_φ tracks meaningful changes rather than noise or instrument quantization.

13.2.3 Seam C: diagnostics-to-ratios calibration

Ledger synchronization depends on mapping diagnostic raw values to dimensionless ratios with the ideal mapping to 1. This is an explicit calibration object with versioning and uncertainty:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure Calibration where
  version : String
  toRatio : DiagnosticMode → ℝ → ℝ
  monotone : ∀ m x y, x ≤ y → toRatio m x ≤ toRatio m y
  ideal_maps_to_one : ∀ m, toRatio m 0 = 1
  uncertainty : ℝ
  uncertainty_pos : 0 < uncertainty
  uncertainty_small : uncertainty ≤ 0.1
```

Limitation: if calibration drifts or the uncertainty envelope is wrong, C_σ becomes misleading. This is why calibration versions and raw values are mandatory in the shot record.

13.2.4 Seam D: ledger weights and normalization scale

The ledger value is a weighted sum of per-mode costs; the weights (and the scale used to map ledger value to C_σ) are facility choices. They should be pre-registered for a given run campaign, then revised only through explicit version increments.

13.3 Model seams (physics approximations)

13.3.1 Seam E: choice of S functional form

We chose $S = 1/(1 + C_\varphi + C_\sigma)$ because it is monotone, bounded, and leads to a clean S^2 temperature/drive lever. This is not unique. If data strongly suggests a different functional form (e.g.

different relative weights of C_φ vs C_σ), the model should be updated and the update versioned.

13.3.2 Seam F: multiplicative exponent assumption

The model assumes coherence reduces the effective barrier *multiplicatively*:

$$\eta_{\mathrm{RS}} = S\,\eta.$$

If experiments show the effect is additive in the exponent, saturating in a different way, or dependent on additional hidden variables, then the model must expand. This is not a failure of RS as an architecture; it is refinement of the applied proxy.

13.3.3 Seam G: regime validity of $\eta(T) \propto 1/\sqrt{T}$

The $T \mapsto S^2T$ lever depends on a standard approximation $\eta(T) \approx K/\sqrt{T}$. If the facility operates in a regime dominated by resonances, transport limits, hydrodynamic instabilities, or non-thermal distributions, that approximation may not capture the dominant temperature dependence. In that case, the correct lever is still "coherence reduces barrier cost" but the temperature scaling exponent will differ and must be fitted.

13.4 Formal seam capsules in Lean (how we represent "unknowns")

To keep the formal stack honest, RS implementation uses two explicit patterns for seams:

1. **Hypothesis capsules**: properties stated as `Prop` with explicit parameters.
2. **Calibration envelope hypotheses**: structures that must be populated by empirical bounds before traceability claims can be derived.

13.4.1 Hypothesis capsules (control theory assumptions)

The fusion control formalization intentionally isolates assumptions that are not yet derived:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Formal.lean
def phi_interference_bound_hypothesis
  {Actuator : Type _} {L : ℕ}
  (S : InterferenceSetting (Actuator := Actuator) L)
  (baseline : Baseline) : Prop :=
  ∃ κ : ℝ, 0 < κ ∧ κ < 1

def jitter_robust_feasibility_hypothesis
  {Actuator : Type _} {L : ℕ}
  (S : PhiScheduler Actuator L) : Prop :=
  ∀ trace : List (PhiScheduler.Update Actuator L),
    S.respectsAssignment trace → S.jitterBounded trace → ∃ exec : S.Execution, exec.trace = trace
```

Interpretation: these are *explicitly named seams*. A facility can treat them as assumptions to be validated experimentally (for a given hardware stack), or as targets for deeper derivations.

13.4.2 Calibration envelope hypothesis (traceability from diagnostics to observables)

The bridge from "ledger computed from ratios" to "observable asymmetry in measurements" is encoded as an explicit envelope hypothesis:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure TraceabilityHypothesis (cfg : BridgeConfig) where
  lower_bound : ℝ
  lower_bound_pos : 0 < lower_bound
  upper_bound : ℝ
  upper_bound_pos : 0 < upper_bound
  offset : ℝ
  offset_nonneg : 0 ≤ offset

theorem traceability (cfg : BridgeConfig) (hyp : TraceabilityHypothesis cfg)
    (meas1 meas2 : DiagnosticMeasurement)
    (hLedgerDecrease : diagnosticLedger cfg meas2 ≤ diagnosticLedger cfg meas1) :
    observableAsymmetry meas2 cfg.modes ≤
    observableAsymmetry meas1 cfg.modes + hyp.offset / hyp.lower_bound := by
  sorry
```

The presence of the `TraceabilityHypothesis` structure is the key point: it is the formal location where calibration bounds enter the proof stack. The `sorry` indicates that the full calibration envelope formalization is not yet completed in the repository and must be treated as an explicit seam (either proven later or replaced with empirically certified bounds).

13.5 How seams close (what to do when predictions fail)

If the predicted scaling fails (e.g. step-down by S^2 causes yield loss earlier than expected), this does not automatically invalidate RS; it identifies which seam must be refined:

- **If negative controls do not change C_φ or yield**: phase/timing extraction is wrong.
- **If C_φ, C_σ move but yield does not**: barrierScale functional form or multiplicative assumption is wrong.
- **If the directionality reverses**: the metrics are not tracking the RS coherence variables.
- **If the directionality holds but scaling exponent differs**: the regime violates $\eta(T) \propto 1/\sqrt{T}$; fit the correct exponent.

This is the core reason the paper insists on versioned calibration objects and deterministic replay: it allows scientific iteration without ambiguity about what changed.

14 Discussion and Future Work

This paper is written in a deliberately operational way: it defines coherence variables, defines a barrier scale, and defines a step-down rule. That structure makes it possible to state what counts as a scientific breakthrough, what should be attempted next, and how the program evolves if early results do not match the simplest scaling model.

14.1 What a "breakthrough" looks like in data

In this work, a breakthrough is not a single high-yield shot. It is a *repeatable causal relationship* between:

$$\text{coherence variables } (C_\varphi, C_\sigma) \;\Rightarrow\; S \;\Rightarrow\; \text{yield/ignition behavior.}$$

Concretely, a breakthrough looks like all of the following holding simultaneously:

1. **Metric controllability**: the facility can reliably increase C_φ and/or C_σ on command (beyond baseline drift), and negative controls reliably decrease them.

2. **Directionality**: shots with larger C_φ and C_σ do not require *more* classical drive to achieve the same yield proxy, after controlling for confounds.

3. **Step-down validity**: applying $T \mapsto S^2 T$ preserves yield to within a pre-registered tolerance band until a clear break point.

The reason we require all three is that each eliminates a different failure mode: metric controllability eliminates the "we did not actually change coherence" loophole; directionality eliminates post-hoc story-telling; and step-down validity is the operational efficiency gain.

14.2 Implications for lower-temperature operation and energy budget

The S^2 lever has a direct meaning for energy budgets. If a classical operating point requires a driver energy or temperature parameter $T_{\text{classical}}$, and the facility can stably reach barrier scale S, then the first-order prediction is:

$$T_{\text{needed}} = S^2 \, T_{\text{classical}}.$$

In energy terms, this is the simplest imaginable kind of improvement: it is a multiplicative reduction in the dominant barrier-driven requirement. The practical benefit is twofold:

- **Lower peak demands**: reduced driver peak power and reduced sensitivity to hardware limits (timing, jitter, staging).
- **Greater robustness**: the same physical temperature behaves as if it were hotter by $1/S^2$, increasing margin.

In RS terms, the facility is spending engineering effort on producing a cleaner, more synchronized pre-commitment ledger evolution, rather than spending that effort solely on heating.

14.3 Connection to nuclear decay (alpha decay) as an independent testbed

Fusion experiments are expensive and high-dimensional. An ideal science program includes independent lower-cost testbeds that probe the *same mechanism*: barrier-mediated commitment with exponential sensitivity.

Alpha decay provides such a testbed because it is a barrier-limited transition with a classical Gamow factor proxy and an exponential rate law. The RS modification is formally parallel to the fusion modification: coherence reduces an effective barrier factor. The Lean formalization makes this explicit:

```
-- Lean excerpt: IndisputableMonolith/Nuclear/AlphaDecay.lean
structure RSCoherenceParams where
  phiCoherence : ℝ
  phiCoherence_nonneg : 0 ≤ phiCoherence
  phiCoherence_le_one : phiCoherence ≤ 1
  ledgerSync : ℝ
  ledgerSync_nonneg : 0 ≤ ledgerSync
  ledgerSync_le_one : ledgerSync ≤ 1

def rsBarrierScale (c : RSCoherenceParams) : ℝ :=
  1 / (1 + c.phiCoherence + c.ledgerSync)

def gamowFactorClassical (Z_daughter Q : ℝ) : ℝ :=
  if Q > 0 then Z_daughter * Real.sqrt (1 / Q) else 0
```

```
def gamowFactor (c : RSCoherenceParams) (Z_daughter Q : ℝ) : ℝ :=
  rsBarrierScale c * gamowFactorClassical Z_daughter Q

def decayConstant (c : RSCoherenceParams) (Z_daughter Q preformation : ℝ) : ℝ :=
  if Q > 0 then preformation * Real.exp (-2 * gamowFactor c Z_daughter Q) else 0
```

This testbed suggests a concrete future-work path: design experiments that modulate the proposed coherence variables in a controlled nuclear setting and measure whether the effective barrier proxy behaves monotonically. The main advantage is that the mechanism is easier to isolate from hydrodynamics and confinement.

14.4 Control modes: coherence pumping, burn-window gating, closed-loop barrier control

From the RS perspective, there are three natural control modes:

1. **Coherence pumping**: a pre-burn segment whose goal is to maximize C_φ and/or C_σ prior to the burn-relevant window.

2. **Burn-window gating**: allocate the tightest jitter bounds and best phase alignment specifically to the short time interval when burn sensitivity is maximal.

3. **Closed-loop barrier control**: treat S as a control variable; adapt scheduling and symmetry corrections shot-by-shot to maximize coherence subject to PASS envelopes.

The symmetry-control stack already includes a formal contraction-style narrative: if the symmetry proxy satisfies a contraction inequality at successive times, it decays geometrically to a bounded region. This provides a template for future "certified improvement" results:

```
-- Lean excerpt: IndisputableMonolith/Fusion/SymmetryProxy.lean
structure ContractionCert where
  eta : ℝ
  eta_pos : 0 < eta
  eta_lt_one : eta < 1
  xi : ℝ
  xi_nonneg : 0 ≤ xi

def satisfiesContraction (cfg : LedgerConfig (Mode := Mode))
    (r : TimeDependentRatios Mode)
    (cert : ContractionCert)
    (t_prev t_next : ℝ) : Prop :=
  symmetryProxy cfg r t_next ≤ (1 - cert.eta) * symmetryProxy cfg r t_prev + cert.xi

theorem proxy_bounded_under_contraction
    (cfg : LedgerConfig (Mode := Mode))
    (r : TimeDependentRatios Mode)
    (cert : ContractionCert)
    (t_seq : ℕ → ℝ)
    (h_contract : ∀ k, satisfiesContraction cfg r cert (t_seq k) (t_seq (k + 1))) :
    ∀ k, symmetryProxy cfg r (t_seq k) ≤
      (1 - cert.eta) ^ k * symmetryProxy cfg r (t_seq 0) +
      cert.xi / cert.eta := by
  -- proof in file
  sorry
```

Future work is to extend the same pattern to barrier-scale control: identify sufficient certificate-level conditions under which S improves monotonically across repeated shots (within a fixed hardware envelope).

14.5 Failure analysis: what it means if S increases but yield does not

If S improves but yield does not, the outcome is still scientifically valuable: it identifies which seam is wrong. The main cases are:

- **Metric mismatch**: C_φ or C_σ is not tracking the relevant physical coherence; revise the measurement pipeline (phase extraction, calibration).
- **Regime mismatch**: yield is limited by non-barrier factors (transport, hydrodynamic instability, confinement losses). In that case barrier reduction may be real but not dominant.
- **Model mismatch**: the barrier reduction may not be multiplicative or may depend on additional state variables not captured by (C_φ, C_σ).

In RS terms: we have reduced the recognition barrier, but the system is failing for reasons that are not barrier-limited. The remedy is to change what is being controlled, not to abandon the coherence concept.

14.6 Roadmap: from metrics to closed-loop controllers to certified runs

The roadmap is:

1. **Metric hardening**: validate C_φ and C_σ with negative controls and drift audits; freeze versions for a campaign.
2. **Barrier-scale calibration**: fit the best $S(C_\varphi, C_\sigma)$ form and confirm directionality.
3. **Efficiency mapping**: generate the empirical step-down curve (max safe step-down vs realized S).
4. **Closed-loop control**: optimize schedules and symmetry corrections to maximize S subject to PASS constraints.
5. **Certification tightening**: progressively replace seams (`TraceabilityHypothesis`, control assumptions) with facility-certified bounds and, where possible, full proofs.

The central philosophy is the same throughout: keep the seams explicit, versioned, and testable; do not let them become hidden assumptions.

15 Conclusion

This paper's goal was not to re-argue the foundations of Recognition Science, but to take a single RS-native reinterpretation—**tunneling as ledger commitment across an effective recognition barrier**—and push it all the way into an applied fusion control program.

The conclusion can therefore be stated in three layers: (i) a conceptual thesis, (ii) an operational claim that can be validated in live runs, and (iii) a verification/certification path that keeps the inevitable empirical seams explicit.

15.1 Summary of the coherence-control thesis

Conceptual thesis. In RS, the barrier is not primarily a spatial wall; it is a *cost-gap* between stable ledger states, and a transition is a *commitment event*. The practical corollary is that *coherence* is not cosmetic: coherence changes the effective barrier cost, and thus changes transition rates at fixed classical conditions.

Operational thesis. The facility should treat coherence as a controlled variable. In this paper we defined two operational scalars:

$$C_\varphi \in [0,1] \quad \text{(timing/phase coherence)}, \qquad C_\sigma \in [0,1] \quad \text{(symmetry/ledger synchronization)}.$$

We then defined a barrier scale $S(C_\varphi, C_\sigma) \in (0,1]$ and used it to define an RS-effective barrier exponent $\eta_{\mathrm{RS}} = S\,\eta$.

Efficiency lever. Under a standard barrier proxy $\eta(T) \propto 1/\sqrt{T}$, the coherence effect becomes a directly usable lever:

$$T_{\mathrm{needed}} = S^2\, T_{\mathrm{classical}}, \qquad T_{\mathrm{eff}} = \frac{T}{S^2}.$$

This is the applied science claim: *if* coherence reduces the effective barrier cost multiplicatively in the exponent, then coherence improvements can be converted into a measured temperature/drive reduction via the S^2 step-down rule.

Canonical executable summary (Lean). The facility-facing computation pipeline is deterministic and auditable. The following excerpt is the minimal executable kernel that turns computed coherence variables into the efficiency lever:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure RSCoherenceInput where
  phiCoherence : Float
  ledgerSync : Float

def computeRSBarrierScale (input : RSCoherenceInput) : Float :=
  let denom := 1.0 + input.phiCoherence + input.ledgerSync
  if denom > 0 then 1.0 / denom else 1.0

def temperatureScaleFromBarrier (S : Float) : Float := S * S
def effectiveTemperatureGain (S : Float) : Float :=
  let S2 := S * S
  if S2 > 0 then 1.0 / S2 else 0.0
```

Everything else in the paper (logging, certificates, negative controls, calibration versioning) exists to ensure that when this kernel is used in live runs, its inputs are meaningful and its outputs are scientifically interpretable.

15.2 Immediate next experiments

The next experiments should be chosen to close the minimal number of seams while maximizing information:

1. **Metric controllability test**: demonstrate that the facility can deliberately increase and decrease C_φ and C_σ using designed interventions (and that negative controls move them in the expected direction).

2. **Directionality test**: at fixed classical drive, show that larger (C_φ, C_σ) does not require higher drive to achieve the same yield proxy (after randomized blocking).

3. **First step-down**: once a stable improved S is achieved, apply the predicted reduction $T \mapsto S^2 T$ and test whether yield remains within a pre-registered tolerance band.

If these succeed, the program immediately graduates from "interesting correlation" to a repeatable efficiency improvement mechanism.

15.3 Longer-term verification and certification path

The long-term plan is to replace implicit assumptions with explicit, versioned artifacts:

- **Calibration hardening**: freeze calibration versions and uncertainty envelopes; audit drift and re-derive ratios/ledger values with deterministic replay.
- **Model refinement**: if the simple S functional form or multiplicative exponent assumption is inaccurate, update it in a controlled, versioned way (treat it as a seam, not a hidden change).
- **Independent testbeds**: validate the same barrier-reduction mechanism in lower-cost contexts (e.g. alpha decay barrier proxies) to reduce dependence on high-variance fusion shots.
- **Certified operations**: progressively tighten PASS envelopes and integrate certificate artifacts so that coherence improvements are achieved within formally specified safety constraints.

The core discipline is simple: *RS gives a direction and a control target; experiments tell us the correct calibration.* By keeping seams explicit, the program remains scientific even when early models are refined.

A Lean Traceability Map

This appendix is a literal map between the scientific objects used in the paper and their Lean formalizations in the repository. Its purpose is operational: if a team member wants to know "what exactly do we mean by X?", they can find the defining Lean object and replay the associated computation or theorem.

Rule for this appendix: we avoid external links; we identify modules by repository paths and include minimal excerpts of the defining code.

A.1 T0–T8 forcing chain: module-by-module mapping

The RS-first-principles story (logic, existence, discreteness, ledger, recognition, unique cost, φ, 8-tick, and $D = 3$) is consolidated into a single "spine" module:

- `IM/Foundation/UnifiedForcingChain.lean`

(Throughout this appendix, `IM` abbreviates `IndisputableMonolith`.)

That file imports each forcing step module and packages them into named statements `t0_holds`, `t1_holds`, ... For example:

```
-- Lean excerpt: IndisputableMonolith/Foundation/UnifiedForcingChain.lean
import IndisputableMonolith.Foundation.LawOfExistence
import IndisputableMonolith.Foundation.LogicFromCost
import IndisputableMonolith.Foundation.DiscretenessForcing
import IndisputableMonolith.Foundation.LedgerForcing
import IndisputableMonolith.Foundation.PhiForcing
import IndisputableMonolith.Foundation.DimensionForcing
import IndisputableMonolith.Foundation.RecognitionForcing
import IndisputableMonolith.Cost

structure T0_Logic_Forced : Prop where
  consistency_cheap : ∃ c : LogicFromCost.ConsistentConfig, LogicFromCost.consistent_cost c = 0
```

```
  contradiction_expensive : ∀ c : LogicFromCost.ContradictionConfig,
    LogicFromCost.contradiction_cost c > 0 ∨ LogicFromCost.IsLogicalContradiction c

theorem t0_holds : T0_Logic_Forced := { ... }
```

For quick navigation, the forcing steps correspond to the following primary modules:

- **T0**: `IM/Foundation/LogicFromCost.lean`
- **T1**: `IM/Foundation/LawOfExistence.lean`
- **T2**: `IM/Foundation/DiscretenessForcing.lean`
- **T3**: `IM/Foundation/LedgerForcing.lean`
- **T4**: `IM/Foundation/RecognitionForcing.lean`
- **T5**: `IM/Cost.lean`
- **T6**: `IM/Foundation/PhiForcing.lean`
- **T7**: `IM/Foundation/EightTick.lean`
- **T8**: `IM/Foundation/DimensionForcing.lean`

A.2 QM bridge: Hilbert space, observables, measurement, and correspondence

The quantum emergence stack used by the paper (phases/interference, Born rule weights, collapse as ledger commitment) is formalized in:

- `IM/Quantum/` (directory)

The directory-level index is documented in `IM/Quantum/README.lean`:

```
-- Lean excerpt: IndisputableMonolith/Quantum/README.lean
import IndisputableMonolith.Quantum.Correspondence

-- Module Structure
--   Quantum/HilbertSpace.lean
--   Quantum/Observables.lean
--   Quantum/LedgerBridge.lean
--   Quantum/Measurement.lean
--   Quantum/Correspondence.lean
```

The minimal foundational objects are:

- **Hilbert space carrier**: `RSHilbertSpace` and `NormalizedState`
- **Observable algebra**: `Observable`, `Projector`, `Hamiltonian`
- **Ledger ↔ Hilbert bridge**: `LedgerToHilbert`, `RHatCorrespondence`
- **Measurement semantics**: ledger commitment as collapse
- **Correspondence capsule**: explicit hypothesis surface for RS ≃ QM

Example (Hilbert carrier):

```
-- Lean excerpt: IndisputableMonolith/Quantum/HilbertSpace.lean
class RSHilbertSpace (H : Type*) extends
  SeminormedAddCommGroup H,
  InnerProductSpace ℂ H,
  CompleteSpace H,
  TopologicalSpace.SeparableSpace H

structure NormalizedState (H : Type*) [RSHilbertSpace H] where
  vec : H
  norm_one : ‖vec‖ = 1
```

A.3 Fusion barrier scaling and operational metrics

This paper's applied mechanism lives at the boundary between:

- a **model layer** (real-number definitions and monotone theorems), and
- an **executable layer** (Float computations from facility logs).

A.3.1 Model layer (barrier scale and monotonicity)

The RS barrier scaling used throughout the paper is formalized in:

- `IM/Fusion/ReactionNetworkRates.lean`

The key objects are `RSCoherenceParams`, `rsBarrierScale`, the RS-adjusted exponent, and the monotone theorem that RS coherence cannot reduce tunneling probability:

```
-- Lean excerpt: IndisputableMonolith/Fusion/ReactionNetworkRates.lean
structure RSCoherenceParams where
  phiCoherence : ℝ
  phiCoherence_nonneg : 0 ≤ phiCoherence
  phiCoherence_le_one : phiCoherence ≤ 1
  ledgerSync : ℝ
  ledgerSync_nonneg : 0 ≤ ledgerSync
  ledgerSync_le_one : ledgerSync ≤ 1

def rsBarrierScale (c : RSCoherenceParams) : ℝ :=
  1 / (1 + c.phiCoherence + c.ledgerSync)

def rsTunnelingProbability (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) : ℝ :=
  Real.exp (-rsGamowExponent c params cfgA cfgB)

theorem rsTunnelingProbability_ge_classical (c : RSCoherenceParams) (params : GamowParams)
    (cfgA cfgB : NuclearConfig) :
    Real.exp (-gamowExponent params cfgA cfgB) ≤ rsTunnelingProbability c params cfgA cfgB := by
  -- proof in file
  ...
```

A.3.2 Executable layer (facility computation of C_φ and C_σ)

The facility-computable metrics and certificate bundles are defined in:

- `IM/Fusion/Executable/Interfaces.lean`

This is where the paper's operational definitions live: `computePhiCoherence`, `computeLedgerSync`, `computeRSBarrierScale`, `computeRSShotEfficiency`, and the `CertificateBundle` schema used for audit/replay.

A.3.3 Diagnostics bridge and traceability envelope

The mapping from raw diagnostics to ratios and ledger values is formalized in:

- `IM/Fusion/DiagnosticsBridge.lean`

This module also contains the explicit `TraceabilityHypothesis` capsule that represents calibration-envelope assumptions used to connect ledger changes to observable asymmetry changes.

A.3.4 Run protocol stack: scheduler, ledger, certificate

The run protocol and safety envelope rely on:

- `IM/Fusion/Scheduler.lean` (scheduler traces, jitter boundedness)
- `IM/Fusion/SymmetryLedger.lean` (ledger value, PASS predicate)
- `IM/Fusion/Certificate.lean` (bundling scheduler + ledger constraints)

The central idea is that the paper's "coherence improvement" experiments must operate inside a certificate-defined envelope: whatever we vary to raise C_φ and C_σ must still satisfy `pass` constraints derived from the symmetry ledger.

B Operational Metric Definitions (Executable)

This appendix collects the *facility-facing* (Float-level) definitions of the coherence and efficiency metrics used throughout the paper. The purpose is deterministic replay: given the raw shot logs and the chosen calibration parameters, a third party can recompute exactly the same C_φ, C_σ, and S values.

Important distinction: these are *executable* definitions, not the real-number model layer. They are designed for robust operation on imperfect data streams, which means they include clamping, truncation to shorter lists, and explicit default parameters.

B.1 C_φ: timing errors, skew, phase alignment, normalization

B.1.1 Definition summary

Inputs:

- expected event times (seconds),
- measured event times (seconds),
- channel phases (radians),
- optional channel time offsets (seconds),
- two scale parameters (`jitterScale`, `skewScale`).

Outputs:

- `jitterRMS`, `skewRMS` (diagnostic components),
- `phaseAlignment` (mean resultant length in $[0, 1]$),
- $C_\varphi \in [0, 1]$ (combined coherence scalar).

B.1.2 Robustness conventions

- **Truncation**: timing errors are computed elementwise and truncate to the shorter list so missing log entries do not crash the computation.
- **Empty inputs**: empty lists yield conservative outputs (e.g. phaseAlignment = 0).
- **Clamping**: C_φ is clamped to $[0, 1]$.

B.1.3 Lean executable definition

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
def clamp01 (x : Float) : Float :=
  if x < 0.0 then 0.0 else if x > 1.0 then 1.0 else x

def rms (xs : List Float) : Float :=
  match xs.length with
  | 0 => 0.0
  | n =>
      let meanSq := xs.foldl (fun acc x => acc + x * x) 0.0 / n.toFloat
      Float.sqrt meanSq

def timingErrors (expected measured : List Float) : List Float :=
  match expected, measured with
  | e :: es, m :: ms => (m - e) :: timingErrors es ms
  | _, _ => []

def meanResultantLength (phases : List Float) : Float :=
  match phases.length with
  | 0 => 0.0
  | n =>
      let cosSum := phases.foldl (fun acc θ => acc + Float.cos θ) 0.0
      let sinSum := phases.foldl (fun acc θ => acc + Float.sin θ) 0.0
      clamp01 (Float.sqrt (cosSum * cosSum + sinSum * sinSum) / n.toFloat)

structure PhiCoherenceInput where
  expectedTimes : List Float
  measuredTimes : List Float
  channelPhases : List Float
  channelTimeOffsets : List Float := []
  jitterScale : Float := 1e-12
  skewScale : Float := 1e-12

structure PhiCoherenceOutput where
  jitterRMS : Float
  skewRMS : Float
  phaseAlignment : Float
  phiCoherence : Float

def computePhiCoherence (input : PhiCoherenceInput) : PhiCoherenceOutput :=
  let errs := timingErrors input.expectedTimes input.measuredTimes
  let jrms := rms errs
  let srms := rms input.channelTimeOffsets
  let phaseAlign := meanResultantLength input.channelPhases
  let timingScore :=
    if input.jitterScale > 0 then
      1.0 / (1.0 + (jrms / input.jitterScale) * (jrms / input.jitterScale))
    else 0.0
  let skewScore :=
    if input.skewScale > 0 then
      1.0 / (1.0 + (srms / input.skewScale) * (srms / input.skewScale))
    else 0.0
  let phiC := clamp01 (phaseAlign * timingScore * skewScore)
  ⟨jrms, srms, phaseAlign, phiC⟩
```

B.2 C_σ: ratios, ledger computation, normalization

B.2.1 Definition summary

C_σ is designed to quantify how close the system is to a symmetry-ideal ledger state (near unity ratios) as a scalar in $[0, 1]$. The implementation supports two input modes:

- ratios provided directly (`ratios`),
- or raw diagnostic values provided with a minimal affine calibration (`rawValues` + `calibration`).

The ledger itself is a weighted sum of per-mode J values.

B.2.2 Robustness conventions

- **Truncation**: weights/ratios are zipped; mismatched lengths truncate to the shorter list.
- **Positivity handling**: `jCostFloat` returns 0 when a ratio is non-positive (defensive, not a physical claim).
- **Normalization**: C_σ is a monotone decreasing function of ledgerValue and is clamped to $[0, 1]$.

B.2.3 Lean executable definition

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure LedgerInput where
  weights : List Float
  ratios : List Float

structure LedgerOutput where
  ledgerValue : Float
  passed : Bool
  threshold : Float

def jCostFloat (x : Float) : Float :=
  if x > 0 then (x + 1/x) / 2 - 1 else 0

def computeLedger (input : LedgerInput) (threshold : Float) : LedgerOutput :=
  let pairs := input.weights.zip input.ratios
  let ledgerVal := pairs.foldl (fun acc (w, r) => acc + w * jCostFloat r) 0.0
  ⟨ledgerVal, ledgerVal ≤ threshold, threshold⟩

structure AffineRatioCalibration where
  gains : List Float

def applyAffineCalibration (cal : AffineRatioCalibration) (raw : List Float) : List Float :=
  match cal.gains, raw with
  | g :: gs, x :: xs => (1.0 + g * x) :: applyAffineCalibration ⟨gs⟩ xs
  | _, _ => []

structure LedgerSyncInput where
  weights : List Float
  ratios : List Float := []
  rawValues : List Float := []
  calibration : AffineRatioCalibration := ⟨[]⟩
  threshold : Float := 0.1
  ledgerScale : Float := 0.1

structure LedgerSyncOutput where
  ledgerValue : Float
  passed : Bool
```

```
  ledgerSync : Float

def computeLedgerSync (input : LedgerSyncInput) : LedgerSyncOutput :=
  let ratios :=
    if input.ratios.isEmpty then
      applyAffineCalibration input.calibration input.rawValues
    else
      input.ratios
  let ledgerOut := computeLedger ⟨input.weights, ratios⟩ input.threshold
  let Λ := input.ledgerScale
  let sync :=
    if Λ > 0 then clamp01 (1.0 / (1.0 + ledgerOut.ledgerValue / Λ)) else 0.0
  ⟨ledgerOut.ledgerValue, ledgerOut.passed, sync⟩
```

B.3 S, S^2, and effective-temperature gain

B.3.1 Definition summary

Given the two facility scalars (C_φ, C_σ), the executable stack computes:

- $S = 1/(1 + C_\varphi + C_\sigma)$ (with a safety guard if the denominator is non-positive),
- S^2 (the predicted temperature/drive step-down factor),
- $1/S^2$ (effective-temperature gain).

B.3.2 Lean executable definition

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure RSCoherenceInput where
  phiCoherence : Float
  ledgerSync : Float

structure RSBarrierScaleOutput where
  barrierScale : Float

def computeRSBarrierScale (input : RSCoherenceInput) : RSBarrierScaleOutput :=
  let denom := 1.0 + input.phiCoherence + input.ledgerSync
  let s := if denom > 0 then 1.0 / denom else 1.0
  ⟨s⟩

def temperatureScaleFromBarrier (barrierScale : Float) : Float :=
  barrierScale * barrierScale

def effectiveTemperatureGain (barrierScale : Float) : Float :=
  let s2 := barrierScale * barrierScale
  if s2 > 0 then 1.0 / s2 else 0.0

structure RSShotEfficiencyOutput where
  phiCoherence : Float
  ledgerSync : Float
  barrierScale : Float
  temperatureScale : Float
  effectiveTempGain : Float

def computeRSShotEfficiency (phiC : Float) (ledgerS : Float) : RSShotEfficiencyOutput :=
  let s := (computeRSBarrierScale ⟨phiC, ledgerS⟩).barrierScale
  let s2 := temperatureScaleFromBarrier s
  let gain := effectiveTemperatureGain s
  ⟨phiC, ledgerS, s, s2, gain⟩
```

C Calibration and Diagnostics Bridge

This appendix describes the *measurement seam* that connects the paper's abstract symmetry and ledger notions to real facility diagnostics. In RS terms, the ledger is the canonical accounting structure; in experimental terms, the facility sees images, spectra, and mode decompositions with calibration drift, bias, and uncertainty.

The bridge has one job: turn raw diagnostic signals into a ratio vector r with a well-defined meaning (ideal $\mapsto 1$), so that ledger-based PASS predicates and ledger-derived C_σ scores are scientifically interpretable and replayable.

C.1 Raw diagnostics → ratios: calibration models

C.1.1 Diagnostic modes

In symmetric ICF-style settings, the most common diagnostic representation is a decomposition into spherical harmonic modes (often written $P_0, P_2, P_4, \ldots$). The bridge therefore starts by formalizing a diagnostic mode and a standard mode set:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure DiagnosticMode where
  degree : ℕ
  is_even : degree % 2 = 0

def standardModes : List DiagnosticMode := [
  ⟨0, rfl⟩, ⟨2, rfl⟩, ⟨4, rfl⟩, ⟨6, rfl⟩
]
```

Operationally, this is a declaration of what "modes we track" means. A facility may choose a different set (e.g. include higher modes) but must version that choice.

C.1.2 Calibration object (versioned, monotone, ideal $\mapsto 1$)

The central calibration object is a mapping from raw diagnostic values to dimensionless ratios:

$$\text{ratio}(m) = \texttt{toRatio}(m, \text{raw}(m)), \quad \text{with} \quad \texttt{toRatio}(m, 0) = 1.$$

The key required property is monotonicity: larger raw deviations should not map to smaller ratios without being explicitly justified and documented.

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure Calibration where
  version : String
  toRatio : DiagnosticMode → ℝ → ℝ
  monotone : ∀ m x y, x ≤ y → toRatio m x ≤ toRatio m y
  ideal_maps_to_one : ∀ m, toRatio m 0 = 1
  uncertainty : ℝ
  uncertainty_pos : 0 < uncertainty
  uncertainty_small : uncertainty ≤ 0.1
```

This is the formal statement that calibration is *not* a hidden spreadsheet: it is a first-class, versioned object with a declared uncertainty envelope.

C.1.3 Raw diagnostic measurement record

The diagnostic record required for replay includes the raw values (as a function of mode), a timestamp, and a shot identifier:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure DiagnosticMeasurement where
  rawValues : DiagnosticMode → ℝ
  timestamp : ℝ
  shotId : String
```

C.2 Uncertainty envelopes and drift

C.2.1 Observable asymmetry proxy

Before mapping into ratios and a ledger, the bridge defines an *observable asymmetry proxy* directly from raw values. This is the "ground-level" quantity an experimentalist can compute without any RS interpretation:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
def observableAsymmetry (meas : DiagnosticMeasurement) (modes : List DiagnosticMode) : ℝ :=
  modes.foldl (fun acc m => acc + (meas.rawValues m)^2) 0

theorem observableAsymmetry_nonneg (meas : DiagnosticMeasurement) (modes : List DiagnosticMode) :
    0 ≤ observableAsymmetry meas modes := by
  ...
```

This proxy is not the ledger; it is the measurement-side anchor. The traceability goal of the bridge is to bound how ledger changes imply changes in this observable proxy, within calibration uncertainty.

C.2.2 Drift and uncertainty as explicit seams

The calibration object contains a declared `uncertainty` bound. In the formal stack, drift and uncertainty appear in two places:

- the calibration's stated uncertainty envelope (a bound on mapping error),
- the traceability offset term (the "slack" by which observable comparisons are allowed to shift even when the ledger decreases).

Operational rule: if calibration is updated, its version string must change, and the uncertainty envelope must be re-stated; otherwise longitudinal analysis becomes uninterpretable.

C.3 Versioning and traceability

C.3.1 Bridge configuration and ledger computation

The bridge configuration packages calibration, mode selection, weights, and coupling:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure BridgeConfig where
  calibration : Calibration
  modes : List DiagnosticMode
  weights : DiagnosticMode → ℝ
  weights_pos : ∀ m, 0 < weights m
  coupling : ℝ
  coupling_pos : 0 < coupling

def measurementToRatios (cfg : BridgeConfig) (meas : DiagnosticMeasurement) :
    DiagnosticMode → ℝ :=
  fun m => cfg.calibration.toRatio m (meas.rawValues m)
```

```
def diagnosticLedger (cfg : BridgeConfig) (meas : DiagnosticMeasurement) : ℝ :=
  cfg.modes.foldl (fun acc m =>
    acc + cfg.weights m * Cost.Jcost (measurementToRatios cfg meas m)
  ) 0
```

This is the canonical Real-level definition of "ledger computed from diagnostics." The executable interface used in live operations is a Float approximation of this object; the semantics are the same, and the differences are part of the calibration seam.

C.3.2 Traceability envelope hypothesis

To connect ledger changes to observable asymmetry changes, the bridge introduces an explicit envelope hypothesis:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure TraceabilityHypothesis (cfg : BridgeConfig) where
  lower_bound : ℝ
  lower_bound_pos : 0 < lower_bound
  upper_bound : ℝ
  upper_bound_pos : 0 < upper_bound
  offset : ℝ
  offset_nonneg : 0 ≤ offset

theorem traceability (cfg : BridgeConfig) (hyp : TraceabilityHypothesis cfg)
    (meas1 meas2 : DiagnosticMeasurement)
    (hLedgerDecrease : diagnosticLedger cfg meas2 ≤ diagnosticLedger cfg meas1) :
    observableAsymmetry meas2 cfg.modes ≤
    observableAsymmetry meas1 cfg.modes + hyp.offset / hyp.lower_bound := by
  sorry
```

Interpretation: traceability is *not* assumed for free. It is granted when the facility can populate `TraceabilityHypothesis` with empirically justified bounds (and ideally prove the theorem without `sorry` by formalizing the envelope in full).

C.3.3 Diagnostic certificate (metadata + replay anchors)

Finally, the bridge defines a diagnostic certificate record that carries the key metadata for audit:

```
-- Lean excerpt: IndisputableMonolith/Fusion/DiagnosticsBridge.lean
structure DiagnosticCertificate where
  passed : Bool
  ledgerValue : ℝ
  observableValue : ℝ
  calibrationVersion : String
  shotId : String
  timestamp : ℝ
  ledger_below_threshold : ledgerValue ≤ 0.1 → passed = true

def generateCertificate (cfg : BridgeConfig) (meas : DiagnosticMeasurement)
    (threshold : ℝ) : DiagnosticCertificate where
  passed := diagnosticLedger cfg meas ≤ threshold
  ledgerValue := diagnosticLedger cfg meas
  observableValue := observableAsymmetry meas cfg.modes
  calibrationVersion := cfg.calibration.version
  shotId := meas.shotId
  timestamp := meas.timestamp
  ledger_below_threshold := by
    intro h
    sorry
```

Operationally, this is the formal "receipt" that ties an observed diagnostic measurement to a ledger value and a PASS/FAIL decision under a declared threshold *with an explicit calibration*

version. This is precisely the artifact needed to prevent silent recalibration from changing past conclusions.

D Run Protocol Checklist

This appendix is a practical checklist for executing the paper's program in a facility. It is organized so that a run lead can verify (i) the shot record is replayable, (ii) controls and randomization prevent confounds, and (iii) acceptance/abort criteria are explicit.

The checklist is written to match the formal objects used elsewhere in the paper:

- **Scheduler trace**: `PhiScheduler.Update` events; predicates `respectsAssignment`, `jitterBounded`.
- **Safety envelope**: the symmetry-ledger PASS predicate `pass`.
- **Metrics**: executables `computePhiCoherence`, `computeLedgerSync`, `computeRSBarrierScale`.

D.1 Required logs per shot

D.1.1 Shot identity and version anchors (mandatory)

- **Shot ID** (unique stable identifier).
- **Timestamp** (wall clock and/or facility time base).
- **Software version(s)** for metric computation and preprocessing.
- **Schedule hash** (exact schedule table used to generate expected times).
- **Calibration version string** for every diagnostic calibration used.

Without these anchors, "replay" is impossible and longitudinal comparisons are uninterpretable.

D.1.2 Scheduler trace logs (what the scheduler predicates need)

Log the actuator/window/timestamp update events sufficient to evaluate:

- assignment compliance (`respectsAssignment`),
- jitter boundedness (`jitterBounded`).

The Lean objects the checklist is meant to satisfy are:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Scheduler.lean
structure PhiScheduler (Actuator : Type _) (L : ℕ)
    extends PhiWindowSpec L where
  assignment : Actuator → Finset (Fin L)
  jitterBound : ℝ
  jitter_nonneg : 0 ≤ jitterBound

namespace PhiScheduler
structure Update (Actuator : Type _) (L : ℕ) where
  actuator : Actuator
  window : Fin L
  timestamp : ℝ

def respectsAssignment (trace : List (Update Actuator L)) : Prop :=
```

```
  ∀ e ∈ trace, sched.allowed e.actuator e.window

def jitterBounded (trace : List (Update Actuator L)) : Prop :=
  ∀ {u v : Update Actuator L},
    (u, v) ∈ trace.zipWith Prod.mk trace.tail →
      |v.timestamp - u.timestamp| ≤ sched.jitterBound
```

Operationally: record the update stream; record the declared jitter bound; record the actuator assignment table.

D.1.3 Timing and phase logs (what C_φ needs)

For C_φ computation and audit, log:

- expected event times list,
- measured event times list,
- channel phase vector at the burn-relevant window,
- optional channel time offsets,
- chosen `jitterScale` and `skewScale`.

Additionally, log the intermediate outputs `jitterRMS`, `skewRMS`, and `phaseAlignment`.

D.1.4 Diagnostics logs (what C_σ and traceability need)

For ledgerSync and traceability:

- raw diagnostic mode values (per mode),
- the calibration map version(s) used to convert raw values to ratios,
- the computed ratio vector (or enough information to recompute it),
- the weight vector and the threshold(s) used in the ledger computation.

If `TraceabilityHypothesis` bounds are being claimed, log the bound parameters and their justification documents as part of the campaign record.

D.1.5 Certificate bundles (replay receipts)

Emit certificate bundles for:

- C_φ (with intermediate components),
- C_σ (with `ledgerValue` and PASS/FAIL),
- S and S^2 (step-down factor),
- any campaign-specific safety certificate (ledger PASS predicate).

D.2 Control suite and randomization

D.2.1 Minimum A/B randomized block design

For a block of $2N$ shots:

- N baseline shots (A),
- N coherence-enhanced shots (B),
- randomize order within the block.

Record the randomization seed and the realized sequence (so the analysis can be replayed).

D.2.2 Negative controls (mechanism-breaking interventions)

Include at least one negative control per campaign:

- **Timing shuffle**: intentionally decorrelate event timing relative to expected times.
- **Phase detune**: intentionally degrade phase alignment while keeping classical energy similar.
- **Symmetry scramble**: intentionally increase asymmetry to raise ledger value and reduce C_σ.

These controls validate that the coherence metrics are tracking the intended mechanisms.

D.2.3 Hold-constant list (anti-confound invariants)

Within a step-down series, hold constant (as tightly as possible): target family, geometry, fill, compression settings, and baseline drive shape. Record any deviations explicitly.

D.3 Acceptance thresholds and stopping rules

D.3.1 Safety envelope: ledger PASS predicate

Before running coherence-enhancement or step-down, freeze:

- ledger weights,
- per-mode bounds,
- ledger threshold Λ.

The formal PASS predicate the experiment must satisfy is:

```
-- Lean excerpt: IndisputableMonolith/Fusion/SymmetryLedger.lean
def pass (cfg : LedgerConfig (Mode := Mode))
    (bounds : ModeThresholds (Mode := Mode)) (Λ : ℝ)
    (r : ModeRatios (Mode := Mode)) : Prop :=
  ledger cfg r ≤ Λ ∧ withinThresholds bounds r
```

Operationally: if PASS fails, the shot is outside the declared symmetry envelope and the coherence-efficiency claims should not be evaluated on it without explicit exception handling.

D.3.2 Primary endpoint and tolerance band (pre-register)

Choose one primary yield proxy Y and a tolerance rule that defines "no degradation" under step-down. Pre-register this before running the step-down series to avoid post-hoc endpoint selection.

D.3.3 Step-down stopping rule

Stop stepping down when either:

- yield proxy falls outside the pre-registered tolerance band, or
- PASS envelope fails (safety/operability violation), or
- diagnostics drift exceeds calibration uncertainty envelope (measurement seam violation).

D.3.4 Post-block audit (required)

After each randomized block:

- replay metrics from logged inputs and verify they match stored outputs,
- verify randomization integrity (no systematic drift alignment),
- verify calibration versions did not change mid-block without being recorded.

E Statistical Methods

Fusion shots are high-variance and drift-prone. A coherence-control claim is only meaningful if the analysis plan prevents three classical failure modes:

1. **Selection bias**: we report only the best-looking subset (post-hoc endpoint choice).
2. **Drift confounding**: slow hardware/target drift is mistaken for a coherence effect.
3. **Multiple testing**: enough exploratory slices guarantee "significance" by chance.

This appendix defines an explicit statistical plan that is compatible with RS-first-principles thinking: *inference is recognition*, and recognition weights are cost-weighted.

E.1 RS basis: inference as cost-weighted recognition

The paper's metric layer produces a shot record $(Y_k, T_k, C_{\varphi k}, C_{\sigma k}, S_k)$. The purpose of statistics is to infer whether the observed yield proxy Y behaves as a function of the RS-effective barrier proxy (through S) rather than as a function of drift.

RS provides a native formal language for "probability = cost-weighting." At finite recognition temperature, weights are Gibbs weights and comparisons are naturally expressed via KL divergence. The Lean formalization in `IM/Thermodynamics/RecognitionThermodynamics.lean` is:

```
-- Lean excerpt: IndisputableMonolith/Thermodynamics/RecognitionThermodynamics.lean
structure RecognitionSystem where
  TR : ℝ
  TR_pos : 0 < TR

noncomputable def gibbs_weight (sys : RecognitionSystem) (x : ℝ) : ℝ :=
  exp (- Jcost x / sys.TR)

noncomputable def partition_function {Ω : Type*} [Fintype Ω]
    (sys : RecognitionSystem) (X : Ω → ℝ) : ℝ :=
  ∑ ω, gibbs_weight sys (X ω)

noncomputable def gibbs_measure {Ω : Type*} [Fintype Ω]
    (sys : RecognitionSystem) (X : Ω → ℝ) (ω : Ω) : ℝ :=
```

```
  gibbs_weight sys (X ω) / partition_function sys X

noncomputable def kl_divergence {Ω : Type*} [Fintype Ω] (q p : Ω → ℝ) : ℝ :=
  ∑ ω, if q ω > 0 ∧ p ω > 0 then q ω * log (q ω / p ω) else 0

theorem kl_divergence_nonneg {Ω : Type*} [Fintype Ω]
    (q p : Ω → ℝ) (hq : ∀ ω, 0 ≤ q ω) (hp : ∀ ω, 0 < p ω)
    (hq_sum : ∑ ω, q ω = 1) (hp_sum : ∑ ω, p ω = 1) :
    0 ≤ kl_divergence q p := by
  -- proof in file
  ...
```

Interpretation for applied work: the statistical goal is to select (or compare) empirical models in a way that corresponds to minimizing a divergence / cost gap, while respecting that the facility is not sampling i.i.d. draws but executing a controlled intervention process.

E.2 Power analysis and effect-size targets

E.2.1 Primary estimand (what we are trying to detect)

The step-down program implies a sharp estimand. Let Y be a primary yield proxy, and define a log-scale outcome $Z := \log(Y + \epsilon)$ with a small $\epsilon > 0$ if needed for zeros. The canonical step-down claim is that a coherence-improved shot at T_{needed} should match a baseline shot at $T_{\text{classical}}$ when:

$$T_{\text{needed}} = S^2 T_{\text{classical}}.$$

Define matched pairs (or matched blocks) where one condition is baseline and one is step-down with the predicted mapping. The primary estimand is:

$$\Delta := \mathbb{E}[Z_{\text{step-down}} - Z_{\text{baseline}}].$$

Under the simplest RS scaling model, $\Delta \approx 0$ until the break point.

E.2.2 Effect-size target

Choose a minimum practically important difference δ on the log scale:

$$|\Delta| \geq \delta \quad \text{is practically meaningful.}$$

The choice of δ must be made pre-run and should be tied to engineering thresholds (e.g. a percentage yield drop that is unacceptable).

E.2.3 Variance model and blocking

Use randomized blocks to control drift. Let σ_{within} be the within-block standard deviation of Z. Estimate σ_{within} from pilot blocks before executing deep step-down.

E.2.4 Closed-form sample size (paired design)

For a paired difference design with two-sided type-I error α and power $1 - \beta$, a standard approximation is:

$$N \approx \left(\frac{z_{1-\alpha/2} + z_{1-\beta}}{\delta / \sigma_d} \right)^2,$$

where σ_d is the standard deviation of paired differences. Because shot distributions are often heavy-tailed, we recommend a pilot block to estimate σ_d and then a simulation-based power check (bootstrap/permutation within blocks).

E.2.5 Secondary estimands

Secondary analyses can include:

- a regression of Z on $1/\sqrt{T/S^2}$ (effective-temperature representation),
- ignition success as a binary endpoint,
- slope of yield degradation vs step-down factor (break-point curve).

These must be declared as secondary to avoid multiplicity pitfalls.

E.3 Sequential designs and false-discovery control

E.3.1 Why sequential designs are appropriate

Step-down experiments are naturally sequential: each new temperature/drive level depends on the last achieved S. Therefore the analysis plan should allow early stopping for **success** (clear invariance under step-down) or **failure** (early yield degradation or safety envelope failure) without inflating false positives.

E.3.2 Group-sequential monitoring (alpha spending)

For each step-down stage j, evaluate the pre-registered primary estimand Δ_j within the current randomized block. Use a conservative alpha-spending rule (e.g. O'Brien–Fleming-style) to permit multiple interim looks while maintaining overall type-I error. Pre-register:

- the number of interim looks,
- the alpha-spending schedule,
- the stopping boundaries for success/futility.

E.3.3 Exploration → exploitation as a decision-temperature schedule (RS view)

The statistical process itself is an exploration/exploitation problem: early runs explore parameter space and validate metrics; later runs exploit the best schedule/control. The RS/decision layer formalizes exactly this tradeoff via a Gibbs distribution over options:

```
-- Lean excerpt: IndisputableMonolith/Decision/DecisionThermodynamics.lean
@[ext] structure DecisionTemperature where
  T : ℝ
  nonneg : 0 ≤ T

noncomputable def decision_gibbs_weight (T : DecisionTemperature) (cost : ℝ) : ℝ :=
  if T.T > 0 then exp (- cost / T.T) else if cost = 0 then 1 else 0

noncomputable def decision_partition {Ω : Type*} [Fintype Ω]
    (T : DecisionTemperature) (costs : Ω → ℝ) : ℝ :=
  ∑ ω, decision_gibbs_weight T (costs ω)

noncomputable def decision_probability {Ω : Type*} [Fintype Ω] [Nonempty Ω]
    (T : DecisionTemperature) (costs : Ω → ℝ) (ω : Ω) : ℝ :=
  decision_gibbs_weight T (costs ω) / decision_partition T costs
```

Interpretation: the "analysis policy" should explicitly schedule exploration (more diverse conditions, higher entropy) before exploitation (tight focus near the best-performing coherence configuration), rather than mixing them implicitly and losing interpretability.

E.3.4 False-discovery control

Multiple testing arises from:

- multiple yield proxies,
- multiple candidate barrier-scale functional forms,
- multiple negative controls,
- multiple step-down stages.

Mitigation strategy:

1. declare one primary endpoint and one primary analysis,
2. treat all others as exploratory and control FDR (Benjamini–Hochberg) or familywise error (Bonferroni/Holm) depending on scientific risk tolerance,
3. require that any "exploratory discovery" be confirmed in a fresh randomized block.

E.4 Robustness checks and negative controls

E.4.1 Negative controls (mechanism checks)

Run the negative controls defined in the Run Protocol (timing shuffle, phase detune, symmetry scramble). The statistical expectation is directional:

- negative controls reduce C_φ and/or C_σ,
- yield should not improve under those interventions at fixed classical drive,
- observed changes should align with the component metrics (jitter vs phase vs ledger value).

E.4.2 Permutation tests within randomized blocks

Because the noise distribution may be non-Gaussian, prefer permutation tests within each randomized block for the primary estimand Δ, using the block's randomization as the reference distribution.

E.4.3 Drift diagnostics

Include time-index covariates and calibration-version indicators in secondary regression checks. If calibration version changes mid-block, treat it as a protocol violation unless explicitly pre-registered.

E.4.4 Sensitivity to measurement uncertainty

For C_σ, treat calibration uncertainty as an explicit perturbation and perform a sensitivity analysis: recompute ledger values under worst-case envelope perturbations and confirm that the primary conclusions are stable (or quantify instability).

F Certificate Bundle Examples

This appendix provides concrete certificate bundle specimens for the three facility-facing outputs that matter most for this paper:

1. C_φ: timing/phase coherence certificate,
2. C_σ: symmetry ledger synchronization certificate,
3. RS shot efficiency scalars: S, S^2, and $1/S^2$.

Design intent. A certificate bundle is a replayable "receipt" for one computation. It is meant to be exported as JSON (or any equivalent serialization) and then checked in audit pipelines. The bundle includes:

- a module name and version,
- a timestamp,
- an input hash (enough to reproduce the computation),
- key-value output pairs (as strings for system interoperability),
- a PASS/FAIL bit,
- and a theorem reference string (used as an internal traceability tag).

F.1 Lean formalization: certificate bundle schema and issuers

The executable schema and issuing functions are defined in:
`IM/Fusion/Executable/Interfaces.lean`.
The relevant excerpt is:

```
-- Lean excerpt: IndisputableMonolith/Fusion/Executable/Interfaces.lean
structure CertificateBundle where
  moduleName : String
  version : String := "1.0"
  timestamp : String
  inputHash : String
  outputs : List (String × String)
  passed : Bool
  theoremRef : String

def certifyRSBarrierScale (input : RSCoherenceInput) : CertificateBundle :=
  let output := computeRSBarrierScale input
  ⟨"RSBarrierScale",
   "1.0",
   "2026-01-21",
   s!"phiCoherence={input.phiCoherence},ledgerSync={input.ledgerSync}",
   [("barrierScale", toString output.barrierScale)],
   output.barrierScale > 0.0,
   "Fusion.ReactionNetworkRates.rsBarrierScale_pos (model-layer)"⟩

def certifyPhiCoherence (input : PhiCoherenceInput) : CertificateBundle :=
  let out := computePhiCoherence input
  ⟨"PhiCoherence",
   "1.0",
   "2026-01-21",
   s!"n={input.measuredTimes.length},channels={input.channelPhases.length}",
   [("jitterRMS", toString out.jitterRMS),
    ("skewRMS", toString out.skewRMS),
```

```
    ("phaseAlignment", toString out.phaseAlignment),
    ("phiCoherence", toString out.phiCoherence)],
   out.phiCoherence ≥ 0.0,
   "Executable metric (facility-calibrated)"⟩

def certifyLedgerSync (input : LedgerSyncInput) : CertificateBundle :=
  let out := computeLedgerSync input
  ⟨"LedgerSync",
   "1.0",
   "2026-01-21",
   s!"modes={input.weights.length}",
   [("ledgerValue", toString out.ledgerValue),
    ("passed", toString out.passed),
    ("ledgerSync", toString out.ledgerSync)],
   out.ledgerSync ≥ 0.0,
   "Executable metric derived from ledger"⟩
```

The rest of this appendix provides example inputs and example serialized bundles.

F.2 Example: C_φ certificate

F.2.1 Example input (perfect timing + perfect phase alignment)

Choose a minimal example with perfect agreement between expected and measured times, and phases perfectly aligned at 0 radians:

```
-- Lean example input (illustrative)
def examplePhiInput : PhiCoherenceInput :=
  { expectedTimes := [0.0, 1.0]
    measuredTimes := [0.0, 1.0]
    channelPhases := [0.0, 0.0]
    channelTimeOffsets := [0.0, 0.0]
    jitterScale := 1e-12
    skewScale := 1e-12 }
```

Under this input, the intended behavior is: `jitterRMS` $= 0$, `skewRMS` $= 0$, `phaseAlignment` $= 1$, and $C_\varphi = 1$.

F.2.2 Example certificate bundle (JSON-like rendering)

An example serialized bundle (field names match the Lean structure) is:

```
{
  "moduleName": "PhiCoherence",
  "version": "1.0",
  "timestamp": "2026-01-21",
  "inputHash": "n=2,channels=2",
  "outputs": {
    "jitterRMS": "0.0",
    "skewRMS": "0.0",
    "phaseAlignment": "1.0",
    "phiCoherence": "1.0"
  },
  "passed": true,
  "theoremRef": "Executable metric (facility-calibrated)"
}
```

F.3 Example: C_σ certificate

F.3.1 Example input (unity ratios)

Choose two modes with unit weights and unity ratios. This should produce `ledgerValue` $= 0$, PASS under any nonnegative threshold, and $C_\sigma = 1$:

```
-- Lean example input (illustrative)
def exampleLedgerSyncInput : LedgerSyncInput :=
  { weights := [1.0, 1.0]
    ratios := [1.0, 1.0]
    threshold := 0.1
    ledgerScale := 0.1 }
```

F.3.2 Example certificate bundle (JSON-like rendering)

```
{
  "moduleName": "LedgerSync",
  "version": "1.0",
  "timestamp": "2026-01-21",
  "inputHash": "modes=2",
  "outputs": {
    "ledgerValue": "0.0",
    "passed": "true",
    "ledgerSync": "1.0"
  },
  "passed": true,
  "theoremRef": "Executable metric derived from ledger"
}
```

F.4 Example: RS shot efficiency report line items

This paper uses a small set of RS shot efficiency scalars derived from (C_φ, C_σ):

$$S, \quad S^2, \quad \frac{1}{S^2}.$$

These are produced by the executable function `computeRSShotEfficiency` (Appendix "Operational Metric Definitions").

F.4.1 Example computation and expected values

Take the (idealized) coherence pair $C_\varphi = 1$, $C_\sigma = 1$. Then $S = 1/(1+1+1) = 1/3$, $S^2 = 1/9$, and $1/S^2 = 9$.

F.4.2 Example report line items (JSON-like rendering)

This is not currently emitted as a dedicated `CertificateBundle` in the executable layer, but the recommended report items for the shot record are:

```
{
  "phiCoherence":  "1.0",
  "ledgerSync":    "1.0",
  "barrierScale":  "0.3333333",
  "temperatureScale": "0.1111111",
  "effectiveTempGain": "9.0"
}
```

If desired, a facility can wrap these line items into a `CertificateBundle` with `moduleName = "RSShotEfficiency"` and an `inputHash` that records the inputs and calibration versions used to compute C_φ and C_σ.