OPERATION KOMOREBI

REALIZATION CALCULUS



RESEARCH LIBRARY

AVOC

Artifact value, guarded operations, and calibrated planning

Lean and evidence guide

Komorebi P.

Standardized publication edition / 8 October 2026

REALIZATION CALCULUS

Reading scope

Actual source declarations, proof structure, module maps, negative controls, and pinned reproduction instructions. Archived kernel evidence is distinguished from fresh source and receipt checks.

Abstract

This companion explains the cumulative AVOC v0.41.0 formalization through its actual source declarations, hypotheses, proof structure, and executable contracts. It connects information refinement and finite decision theory to guarded planning, semantic artifact verification, cost accounting, recovery, finite calibration, and operation lifecycle recording. The final statistical composition bounds the unconditional probability of issuing an uncertified policy episode under a finite rational source model. Source-faithful excerpts, module maps, negative controls, and reproduction commands make each claim traceable. The retained release records report 222 clean local target builds, 2,268 named theorem/lemma axiom reports, and 135 substantive negative-test rejections. This publication performs fresh source and receipt checks but does not claim a fresh Lean compilation.

Contents

1 How to read the formal evidence **3**

1.1 Three kinds of evidence . . . 3

1.2 Compiler and axiom provenance . . . 4

2 A map of the cumulative theory **4**

2.1 Lean conventions used below . . . 5

3 Representations, fibers, and exact sufficiency **5**

3.1 The collision obstruction . . . 6

4 Costs turn information into a decision problem **7**

5 Finite stochastic control and exact quotients **7**

5.1 What an exact stochastic quotient must preserve . . . 8

6 Approximate abstraction has an explicit horizon cost **8**

7 Bayesian normalization and its stability boundary **9**

7.1 Small evidence mass amplifies perturbations . . . 10

8 Decision compression uses Q signatures **10**

9 Guarded plans preserve feasibility on every branch **11**

10 From local action gaps to whole-policy regret **12**

10.1 Approximate Q selection . . . 12

11 Primitive model certificates and executable planning **13**

12 Verification cannot repair a lost specification **14**

13 Proof bytes are bound to artifact identity **15**

14 Retry control and nonrefundable attempt accounting **16**

15 Commit recovery is a transition theorem **17**

15.1 Robust quote recovery has an additional identity 17

16 Verified upgrades retain search costs **18**

17 A checked Boolean semantic cost library **18**

18 Typed affine semantics and rank-certified factorization **19**

18.1 A rank witness proves the minimum factor width 19

19 Protected uncertain quotes become robust plans **20**

20 Finite calibration is an exact rational construction **22**

20.1 A concrete formal checkpoint 23

21 A prospective schedule permits data-dependent selection **23**

22 The primitive source law determines binomial prefix counts **24**

23 The raw recorder separates pending intents from terminal records **25**

23.1 The inductive invariant 26

24 The generative recorder proves actual sample correspondence **26**

24.1 The source is constructed, not authenticated 28

25 Global gates and the final issued-policy theorem **28**

25.1 The certificate being issued 29

25.2 Why the composition is substantial 30

26 The indexed Lean executor and the JSON boundary **30**

27 Positive checks and substantive negative controls **31**

27.1 What a negative test establishes 31

27.2 Resource failures were retained 32

28 Reproduce the formal checkpoint **32**
28.1 Check payload and rebuild local objects . . . 33
28.2 Audit exact names and run negative controls . . . 33
28.3 Replay actual executable comparisons . . . 33
28.4 Run a single recorder example . . . 33

29 The physical sampling boundary is sharp **34**
29.1 Censoring and conditional guarantees . . . 34

30 What is closed and what remains open **34**

A Source map and declaration inventory **35**

B Selected exact theorem names **40**

C Retained formal and executable reference files **43**

1 How to read the formal evidence

The main paper states the mathematical and operational results. The visual guide follows concrete examples. This companion answers a different question: which precise Lean objects and declarations support each result, and what must a user supply before the theorem applies? It is written for a mathematician or developer who wants to inspect that boundary without reading every module in sequence.

The formal checkpoint is `AVOC_Formal_v0.41.0`. Its umbrella `AVOC.lean` imports the cumulative theory. Its source archive retains earlier development failures, negative controls, independent reviews, runtime fixtures, and historical release records. The publication does not edit those sources. An assertion about a theorem below refers to that checkpoint, rather than to a newly created proof file.

1.1 Three kinds of evidence

There are three separate objects to inspect. A Lean theorem is a universally quantified statement about its defined objects, with explicit hypotheses. A kernel audit establishes that the elaborated proof term depends only on the reported axioms. A finite executable comparison establishes agreement on the declared tested family. None of these should be silently substituted for another.

For example, the theorem that recorder samples form a source prefix quantifies over every successful finite run of the defined generative machine. The 69,905-word Python comparison exercises a bounded family of raw recorder commands. The latter provides adapter evidence. It does not prove the generative theorem or identify a physical provider with its source model.

Object	Reported scope	Interpretation
Mathematical modules	195 non-audit modules	Source organization, including modules with no named theorem
Named theorem/lemma inventory	2,268 declarations	Includes two private helpers, and many elementary supporting results
Clean local builds	222 targets	Mathematical modules, umbrella and executable/reporting targets
Kernel axiom reports	2,268 exact names	Archived matching membership, with no missing or duplicate report
Negative controls	135 substantive rejections	Archived false proof attempts rejected for mathematical reasons
Fresh publication check	Source and receipt consistency	No kernel or Lean endpoint execution here

These counts describe verification coverage. They do not count independent discoveries. The inventory is more useful as a navigable list of exact declarations than as a single headline number.

1.2 Compiler and axiom provenance

The pinned toolchain is Lean 4.34.0-rc2, commit `6a10ac8c22beadecabdbb0919c2b50214762f91d`. The pinned mathlib commit is `aeb72f1e9981b6d35b426b16dca7759be3e7ba06`. The release's union of reported axioms is `propext`, `Classical.choice`, and `Quot.sound`. This is the standard classical Lean foundation used by the checkpoint. The recorded clean builds used warnings as errors and rebuilt local objects from empty. External dependency caches were checked against pins. Their entire source trees were not rebuilt as part of that local build claim.

The present environment has no available Lean or Lake compiler. Therefore **fresh kernel compilation is not run here**. The companion preserves the distinction in its inventory JSON, captions, and reproduction instructions. It supplies no newly invented axiom report and no new proof file presented as compiled.

2 A map of the cumulative theory

AVOC starts with representations and targets, then adds costs, evidence, actions, probability, and persistent protocol state. The later recorder block does not replace the earlier theory. It closes a particular lifecycle correspondence and composes the actual resulting samples with the existing calibration and policy statements.

Layer	Representative modules	Main obligation
Information	Core, Refinement, RepresentationLattice	A representation preserves the declared target
Decision	Control, Dynamic, Stochastic, Bayesian	An action or plan has its stated value
Approximation	ApproximateStochastic, QRateDistortion, GuardedModelError	Quantified error propagates to policy regret
Artifacts	CertificateLogic, ArtifactBinding, BooleanCostClosure, TypedAffineProgram	Submitted bytes and semantics satisfy a specific checker contract
Operational control	AttemptAccounting, VerifiedUpgrade, LazyStagedControl	Paid actions preserve credit and legal state
Recovery	CommitRecovery, DurableUpgrade, RobustQuoteRecovery	Recorded committed transitions recover safely
Uncertain quotes	QuoteAuthority, RobustQuotePlanning	One compatible realization supports intervals and regret
Calibration	FiniteCalibration, CalibrationSchedule, ObservationStream	A fixed scheduled interval construction has its coverage bound
Observation lifecycle	ObservationRecorder, RecorderReveal, RecorderPolicy	Actual model-generated terminal samples feed the issued policy

This table is a reading map, not a complete import graph. The source inventory records every immediate import. A useful distinction is between an abstract existence theorem, an executable implementation theorem, and a proof that a particular operational model realizes the abstract object. The program often contains all three, but in different files and at different levels of scope.

2.1 Lean conventions used below

`Prop` contains propositions, while `Type` contains data. A `structure` can hold both data fields and proof fields. For example, a rational probability row carries nonnegativity and a sum-to-one proof. `Finset` is a duplicate-free finite set. `List` is an ordered sequence, where duplicates must be controlled separately. `Fin(n+1)` represents an integer count between zero and n, with that bound carried in its type.

`Option` distinguishes an accepted result from no result. `Except` additionally retains an error value. A `noncomputabledef` may use a classical existence selector. It establishes a mathematical value or policy, not an executable runtime bound. The executable planner and codec modules are the place to inspect algorithms and decoding.

Source excerpts below preserve the actual Lean spelling. Mathematical symbols are rendered through the listings style. A statement excerpt stops before its proof body and is not presented as a standalone Lean file. Full proof excerpts are explicitly identified. The complete source remains the authority for surrounding variables, instances, and imports.

3 Representations, fibers, and exact sufficiency

The foundational preorder is deterministic factorization. If $\phi : X \to A$ refines $\psi : X \to B$, there is a decoder $d : A \to B$ such that $d \circ \phi = \psi$. No particular byte format, compression algorithm, or probability

law is assumed here. The declared target $T : X \to Y$ is sufficient from ϕ exactly when it factors through ϕ.

Full proof excerpt. `Komorebi.AVOC.refines_trans`. Source: `AVOC/Core.lean`, line 32.

```
@[trans] theorem refines_trans {X : Type u} {A : Type v} {B : Type w} {C : Type z}
    {φ : X → A} {ψ : X → B} {χ : X → C}
    (hφψ : Refines φ ψ) (hψχ : Refines ψ χ) : Refines φ χ := by
  rcases hφψ with ⟨d₁, hd₁⟩
  rcases hψχ with ⟨d₂, hd₂⟩
  refine ⟨d₂ ∘ d₁, ?_⟩
  intro x
  simp only [Function.comp_apply, hd₁ x, hd₂ x]
```

The proof builds the composite decoder. Transitivity is not an informal analogy between encodings. It is function composition with the two commuting-square equations inserted into the proof term.

3.1 The collision obstruction

The simplest failure certificate is a pair of inputs that the representation merges while the target distinguishes them.

Full proof excerpt. `Komorebi.AVOC.collision_obstructs_sufficiency`. Source: `AVOC/Core.lean`, line 87.

```
theorem collision_obstructs_sufficiency
    {X : Type u} {Z : Type v} {Y : Type w}
    {φ : X → Z} {T : X → Y} {x y : X}
    (hφ : φ x = φ y) (hT : T x ≠ T y) : ¬ TaskSufficient φ T := by
  intro hs
  exact hT (sufficient_preserves_target_distinctions hs hφ)
```

The proof uses the supplied decoder to derive equality of the two target values, then contradicts the target distinction. A reviewer can inspect this before considering any cardinality or optimization statement. One genuine task-relevant collision already rules out exact sufficiency.

Statement excerpt. `Komorebi.AVOC.sufficient_iff_fiber_refinement`. Source: `AVOC/Core.lean`, line 124.

```
theorem sufficient_iff_fiber_refinement
    {X : Type u} {Z : Type v} {Y : Type w} [Nonempty Y]
    {φ : X → Z} {T : X → Y} :
    TaskSufficient φ T ↔ ∀ x y, SameFiber φ x y → SameFiber T x y
```

The reverse implication needs `NonemptyY`, because a decoder must be defined even on representation states never reached by an input. Lean uses classical choice for a representative of a reached fiber and a default value elsewhere. The condition is mathematical, not a proof that an efficient decoder can be recovered from arbitrary input data.

For a family of queries, `JointTarget` stores the complete vector of answers. `sufficient_joint_iff_each` shows that exact recovery of every query is equivalent to exact recovery of that vector. The target is

a canonical coarsest sufficient representation up to mutual factorization. This uniqueness is relative to the declared query family. It does not make a representation adequate for future, undeclared questions.

4 Costs turn information into a decision problem

Information preservation alone cannot choose whether to compute. The `Control` module supplies action scores, feasible metrics, a stop action, confidence-sensitive answering, and a weak-link generation/verification model. Its finite maximizer results use a nonempty feasible action set. They do not claim that finding the maximizer is free.

Full proof excerpt. `Komorebi.AVOC.weakLinkUtility_positive_iff`. Source: `AVOC/Control.lean`, line 174.

```
theorem weakLinkUtility_positive_iff
    (p value generationCost verificationCost : ℝ) :
    0 < weakLinkUtility p value generationCost verificationCost ↔
      generationCost + verificationCost < p * value := by
  unfold weakLinkUtility
  constructor <;> intro h <;> nlinarith
```

This arithmetic threshold states exactly when expected admitted value exceeds generation plus verification cost. It does not prove the input success probability, monetary value, or cost measurement. Those quantities are model inputs. Later quote authority and calibration modules explain how some inputs can be bound to retained evidence.

`stop_optimal_of_nonpositive` formalizes a genuine option to stop. It prevents a model from being forced into a negative-value action merely because all available computations are unattractive. The queue lemmas separately track the imbalance between generation and absorption. Increasing generation capacity can increase an unverified backlog without improving admitted yield.

5 Finite stochastic control and exact quotients

The finite stochastic layer uses rational probability tables. Its `FiniteStochasticControlSystem` carries an admissible action set, a transition row for every state/action pair, row nonnegativity, row normalization, and rational rewards. Expected continuation is a finite weighted sum.

Statement excerpt. `Komorebi.AVOC.stochastic_finite_horizon_optimal_action_exists`. Source: `AVOC/Stochastic.lean`, line 81.

```
theorem stochastic_finite_horizon_optimal_action_exists
    {S : Type u} {A : Type v} [Fintype S] [DecidableEq A]
    (sys : FiniteStochasticControlSystem S A) (n : ℕ) (s : S) :
    ∃ a ∈ sys.actions,
      stochasticHorizonValue sys (n + 1) s = stochasticQValue sys n s a
```

The Bellman recurrence has an optimal action at every finite horizon. This result is an existence and equality statement. The compiler's treatment of the definition as `noncomputable` matters if an application needs executable search rather than a proof that a maximum exists.

5.1 What an exact stochastic quotient must preserve

A deterministic encoding can preserve reward yet fail to preserve future control. Exact stochastic abstraction requires the relevant aggregated transition probabilities to be constant on fibers as well. The quotient proof groups the raw expectation by coarse state, then transports the Bellman recurrence through that equality.

Statement excerpt. `Komorebi.AVOC.quotientStochasticControlSystem_horizonValue_eq`. Source: `AVOC/StochasticQuotient.lean`, line 151.

```
theorem quotientStochasticControlSystem_horizonValue_eq
    {S : Type u} {A : Type v} {Z : Type w}
    [Fintype S] [Fintype Z] [DecidableEq Z] [DecidableEq A]
    (sys : FiniteStochasticControlSystem S A)
    (φ : S → Z) (hsurj : Function.Surjective φ)
    (h : ProjectedStochasticControlSufficient sys φ) :
    ∀ n s,
      stochasticHorizonValue (quotientStochasticControlSystem sys φ hsurj) n (φ s) =
        stochasticHorizonValue sys n s
```

Statement excerpt. `Komorebi.AVOC.quotient_stochasticBellmanOptimalAt_iff`. Source: `AVOC/StochasticQuotient.lean`, line 219.

```
theorem quotient_stochasticBellmanOptimalAt_iff
    {S : Type u} {A : Type v} {Z : Type w}
    [Fintype S] [Fintype Z] [DecidableEq Z] [DecidableEq A]
    (sys : FiniteStochasticControlSystem S A)
    (φ : S → Z) (hsurj : Function.Surjective φ)
    (h : ProjectedStochasticControlSufficient sys φ)
    (n : ℕ) (s : S) (a : A) :
    StochasticBellmanOptimalAt (quotientStochasticControlSystem sys φ hsurj) n (φ s) a ↔
      StochasticBellmanOptimalAt sys n s a
```

These declarations identify value and optimality in the source and quotient systems under the stated sufficient-abstraction hypotheses. They are not claims about unrestricted stochastic differential equations. Their scope is the finite rational control objects named in the signatures.

6 Approximate abstraction has an explicit horizon cost

An approximate stochastic abstraction supplies reward error ε, transition-row ℓ^1 error δ, and a reward envelope R. The proof separates pointwise value error from transport of the continuation value. Nonnegative normalized weights keep pointwise error unchanged under expectation. Row error multiplies the remaining value envelope.

Statement excerpt. `Komorebi.AVOC.approxBudget_closed_form`. Source: `AVOC/ApproximateStochastic.lean`, line 264.

```
theorem approxBudget_closed_form (ε δ R : ℚ) (n : ℕ) :
    approxBudget ε δ R n =
      (n : ℚ) * ε + δ * R * ((n : ℚ) * ((n : ℚ) - 1) / 2)
```

The closed form is

$$B_n = n\varepsilon + \delta R \frac{n(n-1)}{2}.$$

The linear term comes from one reward discrepancy per stage. The triangular term comes from the remaining-horizon magnitude accumulated through transition transport. This formula is a finite-horizon bound, not a contraction estimate or a claim that long-horizon error remains bounded.

Statement excerpt. `Komorebi.AVOC.approximate_stochastic_horizon_value_bound_closed`. Source: `AVOC/ApproximateStochastic.lean`, line 386.

```
theorem approximate_stochastic_horizon_value_bound_closed
    {S : Type u} {A : Type v} {Z : Type w}
    [Fintype S] [Fintype Z] [DecidableEq Z] [DecidableEq A]
    (raw : FiniteStochasticControlSystem S A)
    (absSys : FiniteStochasticControlSystem Z A)
    (φ : S → Z) (ε δ R : ℚ)
    (h : ApproximateStochasticAbstraction raw absSys φ ε δ R)
    (n : ℕ) (s : S) :
    |stochasticHorizonValue raw n s - stochasticHorizonValue absSys n (φ s)| ≤
      (n : ℚ) * ε + δ * R * ((n : ℚ) * ((n : ℚ) - 1) / 2)
```

`ApproximateComposition` proves transport through successive abstraction maps, including ℓ^1 contraction under pushforward. A composed model must still satisfy the complete combined certificate. Multiplying or adding informal error summaries is not a replacement for its hypotheses.

7 Bayesian normalization and its stability boundary

AVOC's `RationalBelief` is a finite normalized nonnegative rational mass function. Observation channels and evidence likelihoods supply the inputs to predictive probability and posterior updates. The positive predictive-mass hypothesis is explicit in the genuine normalized update. A separate totalized definition handles observations of zero mass, with their contribution to expected policy value proved to be zero.

Statement excerpt. `Komorebi.AVOC.bayesUpdate_mul_predictive`. Source: `AVOC/Bayesian.lean`, line 92.

```
theorem bayesUpdate_mul_predictive
    {Ω : Type u} [Fintype Ω]
    {O : Type v} [Fintype O]
    (b : RationalBelief Ω) (E : RationalChannel Ω O) (o : O)
    (hpos : 0 < predictiveProb b E o) (ω : Ω) :
    (bayesUpdate b E o hpos).prob ω * predictiveProb b E o =
      b.prob ω * E.prob ω o
```

Statement excerpt. `Komorebi.AVOC.bayesianPolicyValue_eq_stochasticDecisionValue`. Source: `AVOC/Bayesian.lean`, line 215.

```
theorem bayesianPolicyValue_eq_stochasticDecisionValue
    {Ω : Type u} [Fintype Ω]
    {O : Type v} [Fintype O]
```

```
    {A : Type w}
    (b : RationalBelief Ω) (utility : Ω → A → ℚ)
    (E : RationalChannel Ω O) (policy : O → A) :
    bayesianPolicyValue b utility E policy =
      stochasticDecisionValue b.prob utility E policy
```

The first equation lets a proof cancel the normalizer safely. The second connects signal-conditioned Bayesian decisions with the finite stochastic decision-value formulation. The equivalence is about those defined values, not about a black-box posterior engine.

7.1 Small evidence mass amplifies perturbations

Approximate conditioning requires a positive lower bound ζ on the relevant evidence masses. With prior ℓ^1 error at most α, likelihood error at most β, and an upper likelihood bound K, the combined theorem gives a bound of the form

$$\|b^L - c^M\|_1 \le \frac{2\beta}{\zeta} + \frac{2K\alpha}{\zeta}.$$

Statement excerpt. `Komorebi.AVOC.conditionByWeight_l1_stability_combined`. Source: `AVOC/ApproximateBayesian.lean`, line 227.

```
theorem conditionByWeight_l1_stability_combined
    {Ω : Type u} [Fintype Ω]
    (b c : RationalBelief Ω)
    (L M : Ω → ℚ)
    (hL : ∀ ω, 0 ≤ L ω) (hM : ∀ ω, 0 ≤ M ω)
    (K α β ζ : ℚ)
    (hK : 0 ≤ K) (hMK : ∀ ω, M ω ≤ K)
    (hα : 0 ≤ α) (hprior : beliefL1Distance b c ≤ α)
    (hβ : 0 ≤ β) (hlik : ∀ ω, |L ω - M ω| ≤ β)
    (hζ : 0 < ζ)
    (hbL : ζ ≤ evidenceMass b L)
    (hbM : ζ ≤ evidenceMass b M)
    (hcM : ζ ≤ evidenceMass c M) :
    beliefL1Distance
        (conditionByWeight b L hL (lt_of_lt_of_le hζ hbL))
        (conditionByWeight c M hM (lt_of_lt_of_le hζ hcM))
      ≤ 2 * β / ζ + 2 * (K * α) / ζ
```

The source includes a separate `BayesianInstability` module. A rare observation can sharply amplify differences. The denominator assumptions above are a substantive boundary, not an inconvenient proof artifact to omit from the statement. They also explain why compression guarantees must be tailored to evidence and decision semantics.

8 Decision compression uses Q signatures

A representation need not retain every feature of a state to preserve a particular finite-horizon decision. A Q signature records action values. The `QRateDistortion` module characterizes approximate decoding through fiber diameters and gives separation lower bounds on the number of codes.

Statement excerpt. `Komorebi.AVOC.exists_qSignatureApproximation_iff_fiberDiameter`. Source: `AVOC/QRateDistortion.lean`, line 125.

```
theorem exists_qSignatureApproximation_iff_fiberDiameter
    {S : Type u} [Fintype S]
    {A : Type v} {Z : Type w} [DecidableEq Z]
    (φ : S → Z) (Q : S → A → ℚ) (η : ℚ) :
    (∃ Qbar : Z → A → ℚ, QSignatureApproximation φ Q Qbar η) ↔
      QFiberDiameterBound φ Q η
```

Statement excerpt. `Komorebi.AVOC.q_margin_preserves_unique_optimum`. Source: `AVOC/QRateDistortion.lean`, line 256.

```
theorem q_margin_preserves_unique_optimum
    {S : Type u} {A : Type v} {Z : Type w}
    (φ : S → Z) (Q : S → A → ℚ) (Qbar : Z → A → ℚ) (η : ℚ)
    (happrox : QSignatureApproximation φ Q Qbar η)
    {s : S} {aStar : A}
    (hmargin : ∀ a, a ≠ aStar → 2 * η < Q s aStar - Q s a) :
    ∀ a, a ≠ aStar → Qbar (φ s) a < Qbar (φ s) aStar
```

The first result formalizes the radius-versus-diameter relationship through a midpoint decoder on each finite fiber. The second makes the action margin explicit: if the original unique maximizer is separated by more than 2η, approximate decoding with error η preserves it. A representation can therefore be adequate at one horizon or margin and inadequate at another.

The finite code-count bounds are relative to a separated family and the specified Q coordinates. They do not establish a general theory of arbitrary neural latent representations. They provide a precise certificate format an application can try to satisfy.

9 Guarded plans preserve feasibility on every branch

The guarded system changes the action set with the state. Its state space need not be finite, although the observation type and each feasible action set are finite. A belief state can therefore occur without forcing a finite enumeration of all possible beliefs.

`Plan A O n` is an adaptive observation tree of depth n. The `Legal` predicate checks the selected action and every continuation, including branches of zero probability. A comparator may use a different future action on every observation history. The Bellman optimum still bounds every such legal finite tree.

Statement excerpt. `Komorebi.AVOC.Guarded.safe_optimal_plan_exists`. Source: `AVOC/GuardedControl.lean`, line 177.

```
theorem safe_optimal_plan_exists (sys : System S A O) (safe : S → Prop)
    (hpres : Preserves sys safe) (n : ℕ) (s : S) (hs : safe s) :
    ∃ p : Plan A O n,
      Legal sys s p ∧ TreeSafe sys safe s p ∧
      planValue sys s p = value sys n s ∧
      ∀ q : Plan A O n, Legal sys s q → planValue sys s q ≤ planValue sys s p
```

Safety follows from a proved one-step preservation property for the concrete system. The declaration

does not assume that any external action preserves an arbitrary invariant. Applications must discharge that obligation with their actual action semantics.

10 From local action gaps to whole-policy regret

This proof is one of the main reusable connections in the archive. If a policy chooses an admissible action whose Bellman gap at stage k is at most e_k, the whole expected plan loss is at most $\sum_{k<n} e_k$. The proof is induction on the horizon and retains every branch's bound.

Full proof excerpt. `Komorebi.AVOC.Guarded.plan_regret_le`. Source: `AVOC/GuardedPolicyRegret.lean`, line 25.

```
theorem plan_regret_le (sys : System S A O) (err : ℕ → ℚ)
    {n : ℕ} (s : S) (p : Plan A O n) (hp : NearGreedy sys err s p) :
    value sys n s - planValue sys s p ≤ regretBudget err n := by
  induction n generalizing s with
  | zero => cases p; simp [value, planValue]
  | succ n ih =>
      cases p with
      | node a next =>
          have hsum : expectation sys (value sys n) s a -
              (∑ o, sys.prob s a o * planValue sys (sys.step s a o) (next o)) ≤
              regretBudget err n := by
            calc
              expectation sys (value sys n) s a -
                  (∑ o, sys.prob s a o * planValue sys (sys.step s a o) (next o))
                = ∑ o, sys.prob s a o *
                    (value sys n (sys.step s a o) -
                     planValue sys (sys.step s a o) (next o)) := by
                      simp [expectation, mul_sub, Finset.sum_sub_distrib]
              _ ≤ ∑ o, sys.prob s a o * regretBudget err n := by
                apply Finset.sum_le_sum
                intro o _
                exact mul_le_mul_of_nonneg_left (ih _ _ (hp.2 o)) (sys.prob_nonneg s a o)
              _ = regretBudget err n := by
                rw [← Finset.sum_mul, sys.prob_sum_one]; simp
          have hlocal := hp.1
          unfold qValue at hlocal
          rw [regretBudget_succ]
          change value sys (n + 1) s -
            (sys.reward s a + ∑ o, sys.prob s a o *
              planValue sys (sys.step s a o) (next o)) ≤ _
          linarith
```

At the successor step, the proof writes the difference of expectations as the expectation of differences. The induction hypothesis bounds every continuation by the previous regret budget. Nonnegative probabilities permit multiplication of each inequality. Their sum is one, so that common bound survives averaging. Adding the local action gap gives the next budget.

10.1 Approximate Q selection

Suppose $\widehat{Q}_k$ approximates each feasible action value within η_k, while the selected action is within τ_k of the estimated maximum. Comparing the true optimum to the estimate, then to the selected estimate,

then back to the true selected value gives $2\eta_k + \tau_k$.

Full proof excerpt. `Komorebi.AVOC.Guarded.decoded_action_gap`. Source: `AVOC/GuardedPolicyRegret.lean`, line 74.

```
theorem decoded_action_gap (sys : System S A O) (n : ℕ) (s : S)
    (estimate : A → ℚ) (eta tau : ℚ) (a : A) (ha : a ∈ sys.actions s)
    (he : ∀ b ∈ sys.actions s, |qValue sys n s b - estimate b| ≤ eta)
    (hm : ∀ b ∈ sys.actions s, estimate b ≤ estimate a + tau) :
    value sys (n + 1) s - qValue sys n s a ≤ 2 * eta + tau := by
  have hx := optimalAction_mem sys n s
  have hl := (abs_le.mp (he a ha)).1
  have hu := (abs_le.mp (he (optimalAction sys n s) hx)).2
  have hmax := hm (optimalAction sys n s) hx
  rw [optimalAction_realizes]
  linarith
```

Statement excerpt. `Komorebi.AVOC.Guarded.decoded_policy_regret`. Source: `AVOC/GuardedPolicyRegret.lean`, line 102.

```
theorem decoded_policy_regret (sys : System S A O) (choose : ℕ → S → A)
    (estimate : ℕ → S → A → ℚ) (eta tau : ℕ → ℚ)
    (ha : ∀ n s, choose n s ∈ sys.actions s)
    (he : ∀ n s a, a ∈ sys.actions s → |qValue sys n s a - estimate n s a| ≤ eta n)
    (hm : ∀ n s a, a ∈ sys.actions s →
      estimate n s a ≤ estimate n s (choose n s) + tau n)
    (n : ℕ) (s : S) :
    0 ≤ value sys n s - planValue sys s (policyPlan sys choose n s) ∧
    value sys n s - planValue sys s (policyPlan sys choose n s) ≤
      regretBudget (fun k => 2 * eta k + tau k) n
```

The conclusion concerns expected finite-horizon value. It is not a pathwise upper bound on realized loss. This distinction remains in force when the policy is chosen from calibrated quote intervals later in the theory.

11 Primitive model certificates and executable planning

The `GuardedModelError` certificate packages an admissible abstraction, reward discrepancy, transport discrepancy, and table residuals. Its induction turns those local obligations into a Q-error budget, then applies the guarded policy regret theorem. The point is to avoid taking a desired whole-policy bound as an unexplained input.

Statement excerpt. `Komorebi.AVOC.Guarded.model_decoded_policy_regret`. Source: `AVOC/GuardedModelError.lean`, line 149.

```
theorem model_decoded_policy_regret (raw : System S A O) (model : System Z A O)
    (phi : S → Z) (W : ℕ → Z → ℚ) (eps : ℚ) (transport residual : ℕ → ℚ)
    (hc : ModelCertificate raw model phi W eps transport residual)
    (estimate : ℕ → S → A → ℚ) (choose : ℕ → S → A)
    (codeError tau : ℕ → ℚ)
    (ha : ∀ n s, choose n s ∈ raw.actions s)
```

```
    (he : ∀ n s a, a ∈ raw.actions s →
      |tableQ model (W n) (phi s) a - estimate n s a| ≤ codeError n)
    (hm : ∀ n s a, a ∈ raw.actions s → estimate n s a ≤ estimate n s (choose n s) + tau n)
    (n : ℕ) (s : S) :
    0 ≤ value raw n s - planValue raw s (policyPlan raw choose n s) ∧
    value raw n s - planValue raw s (policyPlan raw choose n s) ≤
      regretBudget (fun k => 2 * (modelQBudget eps transport residual k + codeError k) + tau k) n
```

The executable planner uses finite action lists, bounded depth, and a fallback selector. `choose_allowed` proves admissibility. Its work analysis and guarantee modules account for truncation and computation charge separately. The fallback must satisfy its specified legal-action contract. A computation budget does not justify returning an arbitrary action.

Statement excerpt. `Komorebi.AVOC.ExecutablePlanner.lookahead_policy_regret`. Source: `AVOC/PlannerGuarantee.lean`, line 123.

```
theorem lookahead_policy_regret (m : Model S A O) (sys : Guarded.System S A O)
    (h : Realizes m sys) (fallback : S → A)
    (hf : ∀ s, fallback s ∈ sys.actions s) (R : ℚ)
    (hr : ∀ s a, a ∈ sys.actions s → |sys.reward s a| ≤ R)
    (depth : Nat → Nat) (hd : ∀ k, depth k ≤ k + 1) (n : Nat) (s : S) :
    0 ≤ Guarded.value sys n s - Guarded.planValue sys s
      (Guarded.policyPlan sys (fun k t => choose m fallback (depth k) t) n s) ∧
    Guarded.value sys n s - Guarded.planValue sys s
      (Guarded.policyPlan sys (fun k t => choose m fallback (depth k) t) n s) ≤
      Guarded.regretBudget (fun k => 2 * ((k + 1 - depth k : Nat) : ℚ) * R) n
```

Full proof excerpt. `Komorebi.AVOC.ExecutablePlanner.charged_regret`. Source: `AVOC/PlannerGuarantee.lean`, line 147.

```
theorem charged_regret (optimal gross loss computeCost maxCost : ℚ)
    (hlo : 0 ≤ optimal - gross) (hgap : optimal - gross ≤ loss)
    (hc : 0 ≤ computeCost) (hcap : computeCost ≤ maxCost) :
    0 ≤ optimal - chargedReturn gross computeCost ∧
      optimal - chargedReturn gross computeCost ≤ loss + maxCost := by
  unfold chargedReturn
  constructor <;> linarith
```

The last declaration makes a simple but consequential distinction. A gross policy result can be close to optimal while the computation needed to obtain it is expensive. The charged result adds the permitted computation expense to the policy loss bound. It is a stated tariff/model quantity, not an empirical performance measurement of every host operation.

12 Verification cannot repair a lost specification

The `Verification` module defines a sound gate by implication from gate acceptance to truth. Its admission theorem follows from that implication. The `Specification` and `VerifierEnsemble` modules separate shared specification error from verifier failure.

Statement excerpt. `Komorebi.AVOC.sound_gate_acceptance`. Source: `AVOC/Verification.lean`, line 13.

```
theorem sound_gate_acceptance
    {C : Type*} {accepted : Finset C} {gate truth : C → Prop}
    (hsound : SoundGate gate truth)
    (hadmit : ∀ c ∈ accepted, gate c) :
    ∀ c ∈ accepted, truth c
```

Statement excerpt. `Komorebi.AVOC.cannot_beat_shared_specification_floor`. Source: `AVOC/VerifierEnsemble.lean`, line 208.

```
theorem cannot_beat_shared_specification_floor
    {Ω : Type u} [Fintype Ω]
    (b : RationalBelief Ω)
    {specificationError wrong : Finset Ω}
    (hfloor : specificationError ⊆ wrong) :
    ¬ eventProb b wrong < eventProb b specificationError
```

Multiple verifiers can reduce the component of error captured by their joint failure event under the stated factorization assumptions. They cannot recover information that the shared specification has erased. The examples include perfectly correlated failures and genuinely factorized coordinate failures, so a product-of-marginals conclusion is not used without a premise that justifies it.

13 Proof bytes are bound to artifact identity

The certificate checker is a specified finite derivation language. Its soundness theorem relates successful checking to the language's propositional semantics. It is not a universal parser for arbitrary Lean proof files.

Statement excerpt. `Komorebi.AVOC.Certificate.check_sound`. Source: `AVOC/CertificateLogic.lean`, line 158.

```
theorem check_sound (ctx : List Formula) (target : Formula) (proof : Derivation)
    (h : check ctx target proof = true) (v : Nat → Prop) (hc : Satisfies v ctx) :
    Denote v target
```

The binding layer records target, environment, dependencies, proof data, generation, and pending ticket. A successful receipt cannot simply be pasted onto a different target or a new generation. The exact checks matter because a valid statement about one artifact is not evidence for an altered artifact.

Statement excerpt. `Komorebi.AVOC.Binding.validated_sound`. Source: `AVOC/ArtifactBinding.lean`, line 72.

```
theorem validated_sound (i : Identity) (t : Ticket) (r : Receipt)
    (h : validate i t r = true) (v : Nat → Prop) (hc : Satisfies v (context i)) :
    Denote v i.statement
```

Statement excerpt. `Komorebi.AVOC.Binding.successful_receipt_not_replayable`. Source: `AVOC/ArtifactBinding.lean`, line 161.

```
theorem successful_receipt_not_replayable (s : Slot) (i : Identity) (r : Receipt) (t : Ticket)
    (h : (consume s i r).2 = some t) : (consume (consume s i r).1 i r).2 = none
```

One-shot consumption concerns the defined pending-ticket state. It does not authenticate a remote endpoint. Protected local storage and stable identity meaning remain application obligations. The source and negative tests include target, dependency, environment, proof-byte, generation, and stale-ticket mismatch cases.

14 Retry control and nonrefundable attempt accounting

The retry model compares stopping against another trial with a failure continuation. `NoRevival` states that a nonpositive immediate retry value stays nonpositive after a fault. It is an explicit condition, rather than a claim about every changing operational environment.

Statement excerpt. `Komorebi.AVOC.Retry.choose_threshold`. Source: `AVOC/RetryControl.lean`, line 209.

```
theorem choose_threshold (h : NoRevival m) (n : Nat) (s : S) :
    choose m (n + 1) s = (m.enabled s && decide (0 < immediate m s))
```

The attempt ledger reserves funds and assigns fresh generations. Renewing an attempt consumes its declared fee and count. Publishing, settling, abandoning, and replaying have different contracts. A rejected or abandoned attempt does not erase already incurred charges.

Statement excerpt. `Komorebi.AVOC.Attempts.charge_not_refunded_on_abandon`. Source: `AVOC/AttemptAccounting.lean`, line 206.

```
theorem charge_not_refunded_on_abandon (l r : Ledger) (t : Ticket)
    (h : abandon l t = .ok r) :
    r.charged = l.charged ∧ r.confirmed = l.confirmed ∧ r.terminal = true
```

Statement excerpt. `Komorebi.AVOC.Attempts.complete_trace_bounds`. Source: `AVOC/AttemptAccounting.lean`, line 321.

```
theorem complete_trace_bounds (lim : Limits) (es : List Event) (r : Ledger)
    (h : run lim initial es = .ok r) :
    r.confirmed ≤ r.charged ∧ r.charged = (es.map charge).sum ∧
    r.charged ≤ lim.credits ∧ (es.map attemptCount).sum ≤ lim.attempts
```

The ledger also rejects stale publication and retry after publication. Even a zero-fee attempt can exhaust the attempt-count limit. A pure arithmetic credit bound is distinct from control of the number of external invocations.

15 Commit recovery is a transition theorem

Recovery stores a sequence, pre-state, action, committed result, post-state, and last acknowledged commit. A fresh acknowledgement must agree with the defined checked transition. An exact duplicate acknowledgement can leave the checkpoint unchanged. An unfinished intent supplies no durable outcome from which the machine may invent a report.

Statement excerpt. `Komorebi.AVOC.CommitRecovery.acknowledge_idempotent`. Source: `AVOC/CommitRecovery.lean`, line 126.

```
theorem acknowledge_idempotent (j : Job) (c d : Checkpoint j)
    (r : Commit j) (v : StoreView) (h : acknowledge j c r v = .ok d) :
    acknowledge j d r v = .ok d
```

Statement excerpt. `Komorebi.AVOC.CommitRecovery.recovered_safe`. Source: `AVOC/CommitRecovery.lean`, line 170.

```
theorem recovered_safe (j : Job) (c d : Checkpoint j) (o : Outcome j)
    (v : StoreView) (hs : Safe j c) (h : recover j c o v = .ok d) : Safe j d
```

These are universal statements about the checkpoint machine. They do not prove that a physical operation ran exactly once. An ID may be acknowledged once after a committed result even if an external system executed work in a manner the model does not observe. A new ID for the same physical work is not excluded by global ID uniqueness alone.

15.1 Robust quote recovery has an additional identity

The later recovery layer retains the whole frozen quote evidence alongside the selected action. Changed quote evidence invalidates acknowledgement. The prepared intent is tied to the original robust recommendation. The acknowledgement transition by itself checks a supplied action and outcome. To say that action is the original recommendation, the caller must retain the prepare provenance as well.

Full proof excerpt. `Komorebi.AVOC.Affine.RobustQuoteRecovery.quote_mutation_rejected`. Source: `AVOC/RobustQuoteRecovery.lean`, line 127.

```
@[simp] theorem quote_mutation_rejected (md : QModel n m) (c : Checkpoint md)
    (r : Commit md) (out : RobustQuote.Outcome md) (actual : StoreView md) (ev : EvidenceView n m)
    (h : r.intent.quoteEvidence ≠ ev) :
    acknowledge md c r out actual ev = .error "quote evidence changed after planning" := by
  simp [acknowledge,h]
```

Statement excerpt. `Komorebi.AVOC.Affine.RobustQuoteRecovery.acknowledge_checked_transition`. Source: `AVOC/RobustQuoteRecovery.lean`, line 228.

```
theorem acknowledge_checked_transition (md : QModel n m) (c : Checkpoint md)
    (a : Action) (out : RobustQuote.Outcome md) (t : RobustQuote.RuntimeState md)
    (hc : RobustQuote.confirm md c.state a out = some t)
    (hl : c.last ≠ some (commitFor md c a out t)) :
    acknowledge md c (commitFor md c a out t) out (view t) (evidence md) =
```

```
      .ok ⟨c.next+1,t,some (commitFor md c a out t)⟩
```

The inherited SQLite adapter uses rollback-journal transactions and `synchronous=FULL`. Its intended atomic publication relies on the SQLite, OS, and filesystem contract. The bounded process-exit suite is implementation evidence. It is not a Lean proof of disk writes or a universal power-loss certification.

16 Verified upgrades retain search costs

An upgrade authority fixes the protected contract, objective, and core. A funded trial is charged even when rejected. Promotion requires checked contract preservation and a strict improvement under the stated criterion. The active utility can improve while the entire campaign remains unprofitable because search and switching were expensive.

Statement excerpt. `Komorebi.AVOC.Upgrade.funded_trial_is_charged`. Source: `AVOC/VerifiedUpgrade.lean`, line 136.

```
theorem funded_trial_is_charged [DecidableEq V] (a : Authority V)
    (s : State V) (p : Proposal V) (hf : Funded a s p) :
    (step a s p).next.evaluationSpent = s.evaluationSpent + a.evaluationCost p.candidate
```

Statement excerpt. `Komorebi.AVOC.Upgrade.certified_upgrade_campaign`. Source: `AVOC/VerifiedUpgrade.lean`, line 260.

```
theorem certified_upgrade_campaign [DecidableEq V] (a : Authority V) (v : V)
    (hc : a.Contract v) (ps : List (Proposal V)) :
    let s := run a (initial v) ps
    a.Contract s.active ∧
    s.evaluationSpent + s.switchSpent ≤ a.credits ∧
    (s.guaranteedGain : Int) - (s.evaluationSpent : Int) ≤ campaignNet a s
```

The campaign theorem packages preserved contract, credit bounds, accounting, and a net lower bound. Its result is relative to the model's measurements and tariffs. It gives no claim that arbitrary LLM-generated improvements meet the checker or that the quoted cost equals physical wall time.

17 A checked Boolean semantic cost library

The Boolean backend has three Boolean inputs, hence eight input assignments and 256 Boolean truth signatures. Its program language uses literals, variables, negation, conjunction, disjunction, and exclusive-or. The cost model is eager tree nodes. A checked bank supplies one witness program for each signature and constructor inequalities sufficient for a global lower bound.

Full proof excerpt. `Komorebi.AVOC.BoolClosure.query_globally_minimum`. Source: `AVOC/BooleanCostClosure.lean`, line 234.

```
theorem query_globally_minimum (b : CheckedBank) (p q : Program) (h : Equivalent q p) :
    nodes (query b p) ≤ nodes q := by
  have parts := check_bank_parts b.rows b.valid
  exact witness_global_minimum algebra (cost b.rows) (witness b.rows) parts.2.2 parts.2.1
```

```
    (sem p) q ((sem_eq_iff q p).mpr h)
```

Statement excerpt. `Komorebi.AVOC.BoolClosure.query_idempotent_cost`. Source: `AVOC/BooleanCostClosure.lean`, line 249.

```
theorem query_idempotent_cost (b : CheckedBank) (p : Program) :
    nodes (query b (query b p)) = nodes (query b p)
```

The global minimum quantifies over programs in this declared language and cost model. It is not universal program minimization. A bank with correct semantic witnesses but false lower inequalities must be rejected. A checksum protects the bytes used for a run, but changing a checksum cannot create a valid lower bound certificate.

The retained library format identifies the input order, operator language, cost model, and 256-row correspondence. Reopening performs validation again. An old manifest's initial validation is historical evidence, not a substitute for the current checker result.

18 Typed affine semantics and rank-certified factorization

Affine programs are indexed by input and output dimensions. Sequential composition must match dimensions. Normalization produces a linear matrix and bias vector, and `normalize_correct` identifies its action on every input. Finite probes are complete for this affine language because zero determines the bias and basis vectors determine the linear part.

Statement excerpt. `Komorebi.AVOC.Affine.finite_probe_complete`. Source: `AVOC/TypedAffineProgram.lean`, line 56.

```
theorem finite_probe_complete (a : ∀ {n m}, Atom n m → Signature R n m)
    {n m} (p q : Program Atom n m)
    (h0 : run a p 0 = run a q 0)
    (hb : ∀ j : Fin n, run a p (basis j) = run a q (basis j)) : Equivalent a p q
```

Statement excerpt. `Komorebi.AVOC.Affine.context_congr`. Source: `AVOC/TypedAffineProgram.lean`, line 94.

```
theorem context_congr (a : ∀ {n m}, Atom n m → Signature R n m)
    {i o n m} (c : Context Atom i o n m) {p q : Program Atom i o}
    (h : Equivalent a p q) : Equivalent a (plug c p) (plug c q)
```

The language restriction is essential. Agreement on zero and a basis does not identify an arbitrary nonlinear function. Context congruence permits a checked equivalent component to be substituted inside a typed larger program.

18.1 A rank witness proves the minimum factor width

A two-stage factorization represents the linear matrix A as VU through an intermediate space. The complete upper witness checks composition of both affine signatures, including their offsets. The lower witness consists of matrices L, H satisfying $LAH = I_k$, where k is the proposed width. Rank inequalities

then give $k \leq \operatorname{rank}(A)$, while the exact width-$k$ factorization gives $\operatorname{rank}(A) \leq k$. Any other exact two-stage factorization must have width at least k. No runtime rank calculation is trusted by this gate.

Statement excerpt. `Komorebi.AVOC.Affine.Factorization.width_minimal`. Source: `AVOC/AffineFactorization.lean`, line 66.

```
theorem width_minimal {f : Signature R n m} {c : Candidate R n m}
    (h : Valid f c) {k : Nat} (a : Signature R n k) (b : Signature R k m)
    (he : ∀ x, eval b (eval a x) = eval f x) : c.width ≤ k
```

Statement excerpt. `Komorebi.AVOC.Affine.Factorization.checked_optimal`. Source: `AVOC/AffineFactorization.lean`, line 96.

```
theorem checked_optimal [DecidableEq R] {f : Signature R n m} {c : Candidate R n m}
    (h : check f c = true) :
    (∀ x, execute (plan c) x = eval f x) ∧
    f.linear.rank = c.width ∧
    ∀ (k : Nat) (a : Signature R n k) (b : Signature R k m),
      (∀ x, eval b (eval a x) = eval f x) →
      c.width ≤ k ∧ factorCost n m c.width ≤ factorCost n m k
```

The optimizer compares its declared original, dense, and factor menus. A minimum factor width does not automatically make setup economical at every call count. The choice and reuse modules retain setup costs and amortization. The paid and adaptive rank-probe modules also charge information acquisition. Finding evidence for a cheap factorization is itself an action.

19 Protected uncertain quotes become robust plans

`QuoteAuthority` binds quotes to exact protected identities. Natural setup intervals, rational gross intervals, and probability intervals carry their ordering and probability bounds as proof fields. A compatible `Realization` supplies one true setup, gross value, and probability within each retained interval.

The conservative nominal model reserves upper setup, credits lower gross, and uses lower success probability. The true-system comparator keeps the same conservative upper-setup feasibility guard. It does not silently give an oracle more spending capacity by changing that reservation rule.

Statement excerpt. `Komorebi.AVOC.Affine.QuoteAuthority.identity_mismatch_rejected`. Source: `AVOC/QuoteAuthority.lean`, line 171.

```
@[simp] theorem identity_mismatch_rejected {n m : Nat} {k : Kind}
    (a : Authority n m k) (q : RawQuote n m)
    (h : q.identity ≠ a.approved.identity) : validate a q = none
```

Statement excerpt. `Komorebi.AVOC.Affine.RobustQuote.certified_dominance`. Source: `AVOC/RobustQuotePlanning.lean`, line 506.

```
theorem certified_dominance (md : QModel n m) (h : Nat) (s : State md)
    (a : Action) (_ha : a ∈ LazyStaged.actions (nominal md) s)
```

```
    (hd : ∀ b ∈ LazyStaged.actions (nominal md) s, upperQ md h s b ≤ lowerQ md h s a)
    (r : Realization md) :
    ∀ b ∈ (trueSystem md r).actions s,
      Guarded.qValue (trueSystem md r) h s b ≤ Guarded.qValue (trueSystem md r) h s a
```

Statement excerpt. `Komorebi.AVOC.Affine.RobustQuote.robust_policy_regret`. Source: `AVOC/RobustQuotePlanning.lean`, line 555.

```
theorem robust_policy_regret (md : QModel n m) (r : Realization md)
    (h : Nat) (s : State md) :
    0 ≤ Guarded.value (trueSystem md r) h s -
      Guarded.planValue (trueSystem md r) s
        (Guarded.policyPlan (trueSystem md r) (robustPolicy md) h s) ∧
    Guarded.value (trueSystem md r) h s -
      Guarded.planValue (trueSystem md r) s
        (Guarded.policyPlan (trueSystem md r) (robustPolicy md) h s) ≤
      Guarded.regretBudget (fun k => 2 * globalQRadius md k) h
```

The preceding `certified_dominance` is sound but compares the selected action with every feasible action, including itself. That condition forces its local Q radius to vanish, as `CalibrationBridge.self_comparison_forces_zero_radius` proves. It is therefore too restrictive for most uncertain selected actions.

The useful corrected certificate in `CalibrationBridge` checks only distinct feasible competitors. It treats self-comparison by reflexivity and permits a positive selected radius. Its executable Boolean checker uses the same competitor-only condition.

Full proof excerpt. `Komorebi.AVOC.Affine.CalibrationBridge.competitor_dominance_sound`. Source: `AVOC/CalibrationBridge.lean`, line 21.

```
theorem competitor_dominance_sound (md : QModel n m) (h : Nat)
    (s : State md) (a : Action) (hd : CompetitorDominance md h s a)
    (r : Realization md) :
    ∀ b ∈ (trueSystem md r).actions s,
      Guarded.qValue (trueSystem md r) h s b ≤
        Guarded.qValue (trueSystem md r) h s a := by
  intro b hb
  by_cases hba : b = a
  · subst b
    exact le_rfl
  · exact (q_between md r h s b).2.trans
      ((hd.2 b hb hba).trans (q_between md r h s a).1)
```

Statement excerpt. `Komorebi.AVOC.Affine.CalibrationBridge.checked_competitor_dominance_optimal`. Source: `AVOC/CalibrationBridge.lean`, line 57.

```
theorem checked_competitor_dominance_optimal (md : QModel n m) (h : Nat)
    (s : State md) (a : Action) (hc : competitorDominanceCheck md h s a = true)
    (r : Realization md) :
    ∀ b ∈ (trueSystem md r).actions s,
      Guarded.qValue (trueSystem md r) h s b ≤
```

```
        Guarded.qValue (trueSystem md r) h s a
```

Action-specific Q intervals can thus certify dominance over distinct competitors. Otherwise the robust choice maximizes its lower score and the policy obtains the accumulated Q-radius regret bound. Quotes remain frozen throughout the policy episode. Spending records conservative reservations. A refund or an equality between reserved spending and physical expenditure is not part of this theorem.

20 Finite calibration is an exact rational construction

For n Bernoulli reports and k successes, the mass polynomial is

$$M_{n,k}(p) = \binom{n}{k} p^k (1-p)^{n-k}.$$

On the cell $[i/m, (i+1)/m]$, the checker uses the upper envelope

$$U_{n,k,m,i} = \binom{n}{k} \left(\frac{i+1}{m}\right)^k \left(1 - \frac{i}{m}\right)^{n-k}.$$

It retains a cell when that envelope is strictly larger than $\alpha/(n+1)$. The returned band is the hull of retained cells. If none is retained, it returns $[0,1]$. No floating-point likelihood evaluation or asymptotic normal approximation is used in this formal construction.

Full proof excerpt. `Komorebi.AVOC.FiniteCalibration.excluded_mass_le`. Source: `AVOC/FiniteCalibration.lean`, line 153.

```
theorem excluded_mass_le (n k m : Nat) (hm : 0 < m) {alpha p : ℚ}
    (hp0 : 0 ≤ p) (hp1 : p ≤ 1) (hout : ¬ Contains n k m alpha p) :
    mass n k p ≤ threshold n alpha := by
  obtain ⟨i,him,hil,hiu⟩ := grid_covers m hm hp0 hp1
  have hnot : i ∉ retained n k m alpha := by
    intro hi
    exact hout (retained_cell_in_hull hi hil hiu)
  have henv : envelope n k m i ≤ threshold n alpha := by
    by_contra h
    apply hnot
    exact mem_filter.mpr ⟨mem_range.mpr him, lt_of_not_ge h⟩
  exact (mass_le_envelope n k m i hp0 hp1 hil hiu).trans henv
```

Every rational $p \in [0,1]$ lies in a grid cell. If the returned hull excludes p, that covering cell was not retained. Its envelope is therefore at most the threshold, and the true count mass is at most the envelope. There are $n+1$ possible counts, giving the total error allowance.

Full proof excerpt. `Komorebi.AVOC.FiniteCalibration.missMass_le`. Source: `AVOC/FiniteCalibration.lean`, line 172.

```
theorem missMass_le (n m : Nat) (hm : 0 < m) {alpha p : ℚ}
    (ha : 0 ≤ alpha) (hp0 : 0 ≤ p) (hp1 : p ≤ 1) :
    missMass n m alpha p ≤ alpha := by
  unfold missMass
  calc
```

```
    _ ≤ ∑ _k ∈ range (n+1), threshold n alpha := by
      apply sum_le_sum
      intro k hk
      split
      · unfold threshold; positivity
      · exact excluded_mass_le n k m hm hp0 hp1 ⟨_⟩
    _ = alpha := by
      simp [threshold]
      field_simp
```

The grid restricts the reported endpoints, not the true rational probability. The source theorem handles endpoint probabilities and sample size zero. Its coverage premise is a binomial count law, whose connection to observations is proved separately under a source contract.

20.1 A concrete formal checkpoint

For ten successes, ten cells, and allowance $1/10$, the retained cells are $\{6,7,8,9\}$ and the band is $[3/5,1]$. Ten failures return $[0,2/5]$. Neither sample claims empirical certainty.

Full proof excerpt. `Komorebi.AVOC.FiniteCalibration.ten_successes_retained`. Source: `AVOC/FiniteCalibration.lean`, line 188.

```
theorem ten_successes_retained : retained 10 10 10 (1/10) = {6,7,8,9} := by
  ext i
  by_cases hi : i < 10
  · interval_cases i <;> norm_num [retained,threshold,envelope,cellLower,cellUpper]
  · have h6 : i ≠ 6 := by omega
    have h7 : i ≠ 7 := by omega
    have h8 : i ≠ 8 := by omega
    have h9 : i ≠ 9 := by omega
    simp [retained,hi,h6,h7,h8,h9]
```

Full proof excerpt. `Komorebi.AVOC.FiniteCalibration.ten_successes_lower`. Source: `AVOC/FiniteCalibration.lean`, line 198.

```
theorem ten_successes_lower : lower 10 10 10 (1/10) = 3/5 := by
  norm_num [lower,ten_successes_retained,cellLower,min']
```

These are exact Lean declarations. The present publication does not rerun their kernel checks. It reads their source and retained release evidence.

21 A prospective schedule permits data-dependent selection

A finite schedule fixes checkpoint identities, sample sizes, grids, and nonnegative allowances before the data used by those checkpoints. A finite union bound controls the event that any checkpoint fails. Selection may then depend on the data without requiring independent checkpoints.

Full proof excerpt. `Komorebi.AVOC.CalibrationSchedule.scheduled_failure_le`. Source: `AVOC/CalibrationSchedule.lean`, line 74.

```
theorem scheduled_failure_le [Fintype J] (law : Law Ω)
    (bad : J → Ω → Prop) (alpha : J → ℚ)
    (h : ∀ j, eventMass law (bad j) ≤ alpha j) :
    eventMass law (fun ω => ∃ j, bad j ω) ≤ ∑ j, alpha j := by
  exact (finite_union_le law bad).trans (sum_le_sum (fun j _ => h j))
```

Statement excerpt. `Komorebi.AVOC.CalibrationSchedule.selected_failure_le`. Source: `AVOC/CalibrationSchedule.lean`, line 83.

```
theorem selected_failure_le [Fintype J] (law : Law Ω)
    (bad : J → Ω → Prop) (alpha : J → ℚ)
    (h : ∀ j, eventMass law (bad j) ≤ alpha j) (select : Ω → J) :
    eventMass law (fun ω => bad (select ω) ω) ≤ ∑ j, alpha j
```

The price of selection is the sum of the allocated errors across the declared schedule. The selected checkpoint does not inherit just its own nominal error without further reasoning. The archive's fair-Boolean adaptive-selection counterexample makes this distinction executable and formal.

The schedule is not unrestricted optional stopping. Arbitrarily many unallocated future checkpoints cannot be introduced after seeing the data and covered by the same finite sum. Within this model, checkpoint dependence is allowed, but the indexed family and its allowances must be in place.

22 The primitive source law determines binomial prefix counts

`ObservationStream` uses a finite rational probability space. A sample is a subset of $\mathrm{Fin}(n)$, marking its successful coordinates. `ConditionalLaw` states that, for any previously specified pattern and any coordinate not yet specified, the joint mass of that pattern and a fresh success equals p times the pattern's mass. This avoids dividing by a conditioning event that might have zero mass.

Definition excerpt. `AVOC/ObservationStream.lean`, line 81.

```
def ConditionalLaw (law : Law Ω) (n : Nat) (p : ℚ) (stream : Ω → Sample n) : Prop :=
  ∀ (seen target : Sample n) (i : Fin n), i ∉ seen →
    eventMass law (fun ω => Matches seen target (stream ω) ∧ i ∈ stream ω) =
      p * eventMass law (fun ω => Matches seen target (stream ω))
```

Statement excerpt. `Komorebi.AVOC.ObservationStream.conditional_implies_product`. Source: `AVOC/ObservationStream.lean`, line 132.

```
theorem conditional_implies_product (law : Law Ω) (n : Nat) (p : ℚ)
    (stream : Ω → Sample n) (h : ConditionalLaw law n p stream) :
    ProductLaw law n p stream
```

Statement excerpt. `Komorebi.AVOC.ObservationStream.prefix_count_fiber_eq_binomial`. Source: `AVOC/ObservationStream.lean`, line 250.

```
theorem prefix_count_fiber_eq_binomial (law : Law Ω) (N : Nat) (p : ℚ)
    (stream : Ω → Sample N) (h : ConditionalLaw law N p stream)
```

```
    (n : Nat) (hn : n ≤ N) (k : Fin (n + 1)) :
    eventMass law (fun ω => streamCount (fun ω => truncateSample n hn (stream ω)) ω = k) =
      FiniteCalibration.mass n k.val p
```

Induction on revealed coordinates establishes the product law. Summing over patterns with k successes gives the binomial count mass. Truncation inherits the fresh-coordinate law, so fixed first-n prefixes have the required count law. Merely checking every one-coordinate marginal is insufficient. The correlated-pair examples have fair marginals and fail the product law.

The finite rational source contract is a premise. It is not inferred from the syntax of a log, or from the fact that an operation completed successfully. That obligation is retained through the final recorder-policy composition.

23 The raw recorder separates pending intents from terminal records

The immutable inherited calibration log cannot turn an unfinished record into a changed terminal record with the same operation ID. The new recorder fixes the lifecycle representation by keeping pending intents in a separate list. Its terminal log continues to obey immutable ID reuse.

Definition excerpt. `AVOC/ObservationRecorder.lean`, line 9.

```
structure State (Key : Type) where
  pending : List (Nat × Key)
  log : List (Event Key)
  deriving DecidableEq
```

Definition excerpt. `AVOC/ObservationRecorder.lean`, line 37.

```
def step (s : State Key) : Command Key → Option (State Key)
  | .begin id key =>
      if terminalUsed s id then
        if ∃ e ∈ s.log, e.operation = id ∧ e.key = key then some s else none
      else if pendingUsed s id then
        if (id, key) ∈ s.pending then some s else none
      else some ⟨s.pending ++ [(id, key)], s.log⟩
  | .finish e =>
      if e.outcome = .unfinished then none
      else if e ∈ s.log then some s
      else if terminalUsed s e.operation then none
      else if (e.operation, e.key) ∈ s.pending then
        some ⟨s.pending.filter (fun p => p.1 != e.operation), s.log ++ [e]⟩
      else none
```

A fresh begin appends an intent. An exact pending begin is unchanged. A late begin for an already completed ID and the same identity is unchanged and cannot reopen that operation. Fresh finish requires the matching pending intent, removes its ID, and appends one terminal event. Exact terminal replay is unchanged. A changed identity, changed outcome, missing intent, or unfinished terminal status is rejected.

23.1 The inductive invariant

`WellFormed` requires duplicate-free pending IDs, duplicate-free terminal IDs, disjoint pending and terminal IDs, and no unfinished terminal entry. The initial state satisfies it. The begin and finish cases preserve it, and induction extends preservation to every successful finite command list.

Full proof excerpt. `Komorebi.AVOC.ObservationRecorder.run_wellFormed`. Source: `AVOC/ObservationRecorder.lean`, line 238.

```
theorem run_wellFormed {cs : List (Command Key)} {s : State Key}
    (h : run cs = some s) : WellFormed s :=
  runFrom_wellFormed initial_wellFormed h
```

Statement excerpt. `Komorebi.AVOC.ObservationRecorder.accepted_terminal_retained_once`. Source: `AVOC/ObservationRecorder.lean`, line 395.

```
theorem accepted_terminal_retained_once {s t u : State Key} (hs : WellFormed s)
    {e : Event Key} {cs : List (Command Key)}
    (hf : step s (.finish e) = some t) (hr : runFrom t cs = some u) :
    e ∈ u.log ∧ (u.log.map Event.operation).count e.operation = 1
```

Terminal history grows by suffix append. An accepted terminal event remains exactly once under every later successful continuation. Success and rejection contribute one Boolean report each. Faults are retained for lifecycle accountability but contribute no Boolean sample.

Statement excerpt. `Komorebi.AVOC.ObservationRecorder.fault_preserves_samples`. Source: `AVOC/ObservationRecorder.lean`, line 341.

```
theorem fault_preserves_samples {s t : State Key} {id : Nat} {key : Key}
    (h : step s (.finish ⟨id, key, .fault⟩) = some t) (target : Key) :
    samples target t.log = samples target s.log
```

Full proof excerpt. `Komorebi.AVOC.ObservationRecorder.inherited_unfinished_terminal_conflict`. Source: `AVOC/ObservationRecorder.lean`, line 359.

```
theorem inherited_unfinished_terminal_conflict (id : Nat) (key : Key)
    (outcome : Outcome) (ht : outcome ≠ .unfinished) :
    record [⟨id, key, .unfinished⟩] ⟨id, key, outcome⟩ = none := by
  apply conflicting_replay
  · simpa using ht
  · exact ⟨⟨id, key, .unfinished⟩, by simp, rfl⟩
```

Global operation-ID uniqueness is across identities. It is an invariant of this reducer's state. It does not forbid a physical caller from using a fresh ID for duplicated external work.

24 The generative recorder proves actual sample correspondence

The raw recorder receives an external terminal label. The source-generative recorder has a different interface: begin, report, and fault. A fresh report constructs its label from the next latent source Boolean

for that identity. Replayed success or rejection is checked before the source is inspected, so a replay consumes no new Boolean even after source exhaustion.

Definition excerpt. `AVOC/RecorderReveal.lean`, line 46.

```
def step (s : State Key) : Command Key → Option (State Key)
  | .begin id key => commit s (.begin id key) none
  | .fault id key => commit s (.finish ⟨id, key, .fault⟩) none
  | .report id key =>
      if ⟨id, key, CalibrationLog.Outcome.success⟩ ∈ s.recorder.log then
        commit s (.finish ⟨id, key, .success⟩) none
      else if ⟨id, key, CalibrationLog.Outcome.rejection⟩ ∈ s.recorder.log then
        commit s (.finish ⟨id, key, .rejection⟩) none
      else match s.reveal.remaining key with
        | [] => none
        | bit :: _ =>
            commit s (.finish ⟨id, key, ObservationReveal.checkedOutcome bit⟩) (some key)
```

Its state retains the raw recorder, reveal state, emitted ordinary commands, and effective reveal trace. The `Tracks` invariant connects all four. This is stronger than separately supplying a next-bit equality for each accepted external event. It proves that the whole generated command list projects to the raw recorder and the inherited reveal process.

Full proof excerpt. `Komorebi.AVOC.RecorderReveal.run_refines`. Source: `AVOC/RecorderReveal.lean`, line 224.

```
theorem run_refines (source : Key → List Bool) (commands : List (Command Key))
    {s : State Key} (h : run source commands = some s) :
    ObservationRecorder.run s.emitted = some s.recorder ∧
      ∀ key, CalibrationLog.samples key s.recorder.log =
        (ObservationReveal.run source s.effectiveTrace).reports key := by
  have ht := run_tracks source commands h
  refine ⟨ht.recorder_projection, ?_⟩
  intro key
  rw [ht.samples, ht.reveal_projection]
```

Full proof excerpt. `Komorebi.AVOC.RecorderReveal.run_samples_eq_take`. Source: `AVOC/RecorderReveal.lean`, line 234.

```
theorem run_samples_eq_take (source : Key → List Bool) (commands : List (Command Key))
    {s : State Key} (h : run source commands = some s) (key : Key) :
    CalibrationLog.samples key s.recorder.log =
      (source key).take (CalibrationLog.samples key s.recorder.log).length := by
  rw [(run_refines source commands h).2 key]
  exact ObservationReveal.run_reports_eq_take source s.effectiveTrace key
```

Writing L_k for the actual terminal sample list at identity k and S_k for the latent source, the conclusion is

$$L_k = \mathrm{take}_{|L_k|}(S_k).$$

Faults, begins, and exact replays do not consume a source position. A fresh report at an exhausted source is rejected. The finite proof applies to all successful command lists, including replay-containing lists.

24.1 The source is constructed, not authenticated

The theorem's transition definition constructs the next label. It cannot be used as a test that an arbitrary physical label happens to equal the next latent bit. A raw log may be well formed and fully resolved without belonging to the statistical generative model. The physical correspondence is the main remaining sampling obligation.

25 Global gates and the final issued-policy theorem

The final block runs the complete generative command list. Rejection yields no counts. Any pending operation anywhere yields no counts, even if an unrelated selected identity has enough reports. A fully resolved run reads each fixed first-n checkpoint from its actual terminal log. Both false and true reports count toward readiness. Only true reports count as successes.

Definition excerpt. `AVOC/RecorderPolicy.lean`, line 12.

```
def gatedCounts (n : J → Nat) (key : J → K)
    (source : Ω → K → List Bool) (commands : Ω → List (RecorderReveal.Command K))
    (j : J) (ω : Ω) : Option (Fin (n j + 1)) :=
  match RecorderReveal.run (source ω) (commands ω) with
  | none => none
  | some s => if s.recorder.pending = [] then
      ObservationPolicy.availableCount (n j) (CalibrationLog.samples (key j) s.recorder.log)
    else none
```

Full proof excerpt. `Komorebi.AVOC.RecorderPolicy.pending_run_no_plan`. Source: `AVOC/RecorderPolicy.lean`, line 79.

```
theorem pending_run_no_plan {r c : Nat} (md : QModel r c)
    (n m : J → Nat) (alpha : J → ℚ) (key : J → K)
    (source : Ω → K → List Bool) (commands : Ω → List (RecorderReveal.Command K))
    (selectWarm selectCold : Ω → J) (ω : Ω) (s : RecorderReveal.State K)
    (h : RecorderReveal.run (source ω) (commands ω) = some s)
    (hp : s.recorder.pending ≠ []) :
    recorderPlan md n m alpha key source commands selectWarm selectCold ω = none := by
  apply Affine.ObservedCalibration.observedPlan_none_warm
  exact pending_run_no_count n key source commands ω s h hp (selectWarm ω)
```

Full proof excerpt. `Komorebi.AVOC.RecorderPolicy.gatedCounts_faithful`. Source: `AVOC/RecorderPolicy.lean`, line 37.

```
theorem gatedCounts_faithful
    (n : J → Nat) (key : J → K)
    (source : Ω → K → List Bool) (commands : Ω → List (RecorderReveal.Command K))
    (stream : (j : J) → Ω → Sample (n j))
    (hprefix : ∀ j ω, sampleList (stream j ω) = (source ω (key j)).take (n j)) :
    ObservedCalibration.Faithful n (fun j => streamCount (stream j))
      (gatedCounts n key source commands) := by
  intro j ω k hk
  unfold gatedCounts at hk
  cases hr : RecorderReveal.run (source ω) (commands ω) with
```

```
  | none => simp [hr] at hk
  | some s =>
    simp only [hr] at hk
    split at hk
    · have hav := ObservationPolicy.availableCount_value (n j)
        (CalibrationLog.samples (key j) s.recorder.log) k hk
      apply Fin.ext
      change k.val = (stream j ω).card
      rw [hav.2, RecorderReveal.run_samples_eq_take (source ω) (commands ω) hr,
        List.take_take, Nat.min_eq_left hav.1, ← hprefix, sampleList_count_true]
    · simp at hk
```

The faithfulness proof does not assume that the observed run was completed or that its samples are a source prefix. It derives the relevant correspondence on every branch where a count is returned, using the whole-run refinement. No-count branches remain no-count branches.

25.1 The certificate being issued

`EpisodeCertified` states existence of a quote realization with the specified warm/cold true probabilities and deterministic setup/gross values, together with the frozen-episode Q, value, and expected regret guarantee. Deterministic setup/gross interval membership remains supplied by a separate `DeterministicRealization`. Sampling only calibrates the probability fields.

Definition excerpt. `AVOC/CalibratedPolicy.lean`, line 21.

```
def EpisodeCertified {r c : Nat} (md : QModel r c) (wp cp : ℚ)
    (ws cs : Nat) (wg cg : ℚ) : Prop :=
  ∃ realization : Realization md,
    realization.warm.probability = wp ∧ realization.cold.probability = cp ∧
      realization.warm.setup = ws ∧ realization.cold.setup = cs ∧
      realization.warm.gross = wg ∧ realization.cold.gross = cg ∧
      FrozenEpisodeGuarantee md realization
```

For per-key latent sources satisfying `ConditionalLaw`, fixed scheduled checkpoints, positive grids, and nonnegative allowances, the final declaration bounds the unconditional event of issuing an uncertified episode.

Full proof excerpt. `Komorebi.AVOC.RecorderPolicy.recorder_issued_failure_le`. Source: `AVOC/RecorderPolicy.lean`, line 129.

```
theorem recorder_issued_failure_le {r c : Nat}
    (law : Law Ω) (md : QModel r c) (d : DeterministicRealization md)
    (N : K → Nat) (p : K → ℚ) (base : (k : K) → Ω → Sample (N k))
    (hlaw : ∀ k, ConditionalLaw law (N k) (p k) (base k))
    (n m : J → Nat) (alpha : J → ℚ) (key : J → K)
    (hn : ∀ j, n j ≤ N (key j))
    (hm : ∀ j, 0 < m j) (ha : ∀ j, 0 ≤ alpha j)
    (hp0 : ∀ k, 0 ≤ p k) (hp1 : ∀ k, p k ≤ 1)
    (commands : Ω → List (RecorderReveal.Command K))
    (selectWarm selectCold : Ω → J) (wp cp : ℚ)
    (hwparam : ∀ ω, p (key (selectWarm ω)) = wp)
    (hcparam : ∀ ω, p (key (selectCold ω)) = cp) :
    eventMass law (fun ω => ∃ planned,
```

```
      recorderPlan md n m alpha key (fun ω k => sampleList (base k ω)) commands
        selectWarm selectCold ω = some planned ∧
      ¬ EpisodeCertified planned wp cp d.warmSetup d.coldSetup d.warmGross d.coldGross) ≤
      ∑ j, alpha j := by
  have hg := recorder_episode_coverage law md d N p base hlaw n m alpha key hn hm ha
    hp0 hp1 commands selectWarm selectCold wp cp hwparam hcparam
  have hc := eventMass_add_compl law (fun ω => ∃ planned,
      recorderPlan md n m alpha key (fun ω k => sampleList (base k ω)) commands
        selectWarm selectCold ω = some planned ∧
      ¬ EpisodeCertified planned wp cp d.warmSetup d.coldSetup d.warmGross d.coldGross)
  simp only [not_exists, not_and, not_not] at hc
  linarith
```

In mathematical notation, with $A = \sum_j \alpha_j$,

$$\mathbb{P}\big(\text{a plan is issued and is not EpisodeCertified}\big) \le A.$$

The probability is over the original finite rational space. Rejected, unfinished, and insufficient-sample trajectories remain in it. They issue no plan. The theorem does not divide by the chance of issuance and does not assert a posterior probability for the observed interval.

25.2 Why the composition is substantial

The final theorem no longer takes successful execution, absence of pending intents, completion, a count law, log-prefix equality, or effective-trace equality as extra premises. The machine and its refinement derive those properties where they are needed. The still-explicit assumptions are the primitive source law, prospective schedule, selected parameter strata, and deterministic field membership. A policy need not be issued, and no positive productivity, termination, or collection-cost lower bound follows.

26 The indexed Lean executor and the JSON boundary

The executable reducer records the index of the first rejected command. Its successful state is proved equal to the raw core run for every finite list of already-decoded commands.

Full proof excerpt. `Komorebi.AVOC.RecorderCodec.replayFrom_toOption`. Source: `AVOC/RecorderCodec.lean`, line 18.

```
theorem replayFrom_toOption {Key : Type} [DecidableEq Key] (s : State Key)
    (index : Nat) (commands : List (Command Key)) :
    (replayFrom s index commands).toOption = runFrom s commands := by
  induction commands generalizing s index with
  | nil => rfl
  | cons c cs ih =>
    cases h : step s c with
    | none => simp [replayFrom, runFrom, h, Except.toOption]
    | some t => simpa [replayFrom, runFrom, h] using ih t (index + 1)
```

This theorem covers the executor, not universal correctness of JSON decoding, host-language parsing, input-file checks, or Python's separate reducer. Those layers are exercised by finite interface tests. A malformed request is an input error; a semantically rejected command list returns no final snapshot. All commands are parsed before an output snapshot is exposed.

The recorder endpoint accepts integer IDs and keys through 2,147,483,647. Single requests have at most 4,096 commands. Batches have at most 256 words and 16,384 commands total. The wrapper accepts regular nonsymlink files of at most 4 MiB. Exact arities and terminal tags matter. The optional injective registry exports calibration events only after all pending operations are resolved. Its evidence identifiers are deterministic labels, not signatures or external proof certificates.

```
["observation-recorder-v1", [
  ["begin", 0, 1],
  ["finish", 0, 1, "success"],
  ["finish", 0, 1, "success"],
  ["begin", 0, 1]
]]
```

The defined result has no pending intent and one success terminal record. The two later commands are exact replays. It is ready in the lifecycle sense, but readiness does not assert enough calibration samples or any source law.

27 Positive checks and substantive negative controls

The retained positive oracle enumerates all command words of lengths zero through four over sixteen commands, two IDs, and two keys. That is 69,905 Python/reference words. The actual Lean endpoint is compared on lengths zero through three, plus twelve focused longer scenarios, for 4,381 words. This precise family is adapter evidence, not an exhaustive proof over arbitrary IDs or arbitrarily long lists.

The inherited regressions cover robust quote execution, selected-only workflows, recovery, lazy staged execution, and calibration. Historical v0.40 observation-process evidence remains historical. Its unchanged input pins are checked, but it is not relabeled as a fresh v0.41 run.

27.1 What a negative test establishes

A false theorem attempted with a proof tactic is a useful control when its failure is substantive. A syntax error, missing import, startup failure, timeout, or crash is not mathematical rejection. The retained runner classifies these outcomes separately and requires the exact current membership.

Negative control. `tests/negative/FalseRecorderUnbegunFinish.lean`. Archived as a substantive rejection.

```
import AVOC.ObservationRecorder
open Komorebi.AVOC.ObservationRecorder
/-- Must fail: a terminal outcome requires the matching previously recorded intent. -/
example : step (initial : State String) (.finish ⟨7, "warm", .success⟩) =
    some ⟨[], [⟨7, "warm", .success⟩]⟩ := by decide
```

Negative control. `tests/negative/FalseRecorderReplayDuplicatesSample.lean`. Archived as a substantive rejection.

```
import AVOC.ObservationRecorder
open Komorebi.AVOC.ObservationRecorder
open Komorebi.AVOC.CalibrationLog
/-- Must fail: exact terminal replay cannot duplicate the checked statistical sample. -/
```

```
example : (run [.begin 7 "warm", .finish ⟨7, "warm", .success⟩,
    .finish ⟨7, "warm", .success⟩]).map (fun s => samples "warm" s.log) =
    some [true, true] := by decide
```

Negative control. `tests/negative/FalseRecorderPendingPlan.lean`. Archived as a substantive rejection.

```
import AVOC.RecorderReveal
open Komorebi.AVOC.RecorderReveal
/-- Must fail: an unresolved begun operation blocks issuance. -/
example :
    (run (fun _ : Unit => [true]) [.begin 0 ()]).map (fun s => issue s (7 : Nat)) =
      some (some 7) := by decide
```

Negative control. `tests/negative/FalseRecorderReplayConsumesNext.lean`. Archived as a substantive rejection.

```
import AVOC.RecorderReveal
open Komorebi.AVOC.RecorderReveal
/-- Must fail: replay of the same report consumes no second source outcome. -/
example :
    (run (fun _ : Unit => [false, true]) [.begin 0 (), .report 0 (), .report 0 ()]).map
      (fun s => Komorebi.AVOC.CalibrationLog.samples () s.recorder.log) =
        some [false, true] := by decide
```

These controls challenge the absence of an intent, replay sample duplication, pending-state issuance, and consuming the next bit on replay. Their complete logs remain in the archive. The positive universal declarations establish the defined properties. The negative attempts help detect a missing hypothesis or an accidentally weakened checker.

27.2 Resource failures were retained

Two early bounded recorder runs produced no scientific result. A 2 GiB virtual memory cap prevented Lean thread creation, and an 8 GiB cap prevented startup mappings. A later diagnostic measured about 8.56 GiB peak virtual space and 3.66 GiB resident memory. The final archived run removed the address-space cap, retained wall/CPU/log/core bounds, and recorded the inherited 20 GiB cgroup limit separately. These changes concern execution feasibility. They do not strengthen the mathematical claim. The failed manifests and diagnostic correction remain part of the evidence.

28 Reproduce the formal checkpoint

Use Linux x86-64 or WSL, Python 3.10 or later, and the separately retained portable Lean kit with the exact package revisions in `lake-manifest.json`. The source archive omits the runtime and compiled project/dependency objects. The portable toolchain is an input to reproduction, not a newly built component of this publication.

28.1 Check payload and rebuild local objects

Run the following from the formal project root. The prior ZIP is needed only to independently repeat the inherited-source comparison. Preserve each command result rather than treating a command printed in a document as execution.

```
export AVOC_LEAN_KIT=/absolute/path/Komorebi_Portable_Lean_4.34.0-rc2
python3 tools/check_manifest.py
python3 tools/check_inherited_v041.py /absolute/path/AVOC_Formal_v0.40.0.zip
python3 tools/inventory_v041.py
python3 tools/verify_v041.py --prepare
python3 tools/build_v041_parallel.py --seconds 45 --workers 3 --per-target-seconds 180
```

Repeat the build command after each invocation exits, until it reports current 222 of 222, zero remaining, and no failed target. Do not repeat `--prepare` between build slices. It removes local objects. The build lock prevents overlapping release builds, and resumption checks source, imports, outputs, and logs. No test using those objects should run while preparation or rebuilding is underway.

28.2 Audit exact names and run negative controls

```
python3 tools/verify_v041.py --audit --seconds 600
python3 tools/run_negative_v041.py --seconds 45 --workers 2 --per-file-seconds 90
python3 tools/source_scan_v041.py
```

Repeat the negative command until 135 of 135 substantive rejections are current. The audit expects exactly 2,268 named reports, with only the three allowed axioms. A zero-hit source scan is a heuristic check. It remains separate from compilation and the actual axiom reports.

28.3 Replay actual executable comparisons

```
python3 checks/test_observation_recorder.py --lean --output results/v0.41.0/oracle_replay.json
python3 tools/run_inherited_regression_v041.py --case all
python3 tools/release_check_v041.py
python3 tools/package_v041.py --output /chosen/AVOC_Formal_v0.41.0.zip
python3 tools/check_manifest.py
```

The new output filename preserves the original canonical oracle result and bounded-run manifest. The inherited runner writes isolated v0.41 evidence and relocates its fixture paths. The release gate checks exact membership, current hashes, compiler provenance, the runtime-bound oracle, baseline data, independent source/tool reviews, and regressions. Changing reviewed source requires renewed review, not just an edited digest.

28.4 Run a single recorder example

```
python3 tools/observation_recorder.py examples/observation_recorder/complete.json
python3 tools/run_recorder_lean.py examples/observation_recorder/complete.json
python3 tools/observation_recorder.py examples/observation_recorder/complete.json --registry examples/
    observation_recorder/registry.json
```

Each request starts from empty and returns a pure snapshot. These commands perform no paid operation, retry, durable journal mutation, or policy issuance. The inherited calibration CLI is a separate endpoint.

Connecting the two by a registry exports compatible syntax, not physical statistical authentication.

29 The physical sampling boundary is sharp

Complete logging and eventual resolution do not by themselves establish an IID stream in finalization order. Consider two independent fair attempted outcomes. If the successful outcome finishes first whenever the two differ, the first logged bit is true unless both attempts failed. Its success probability is $3/4$, not $1/2$. Both attempts can be fully and honestly logged. The bias comes from order, not missing data.

The retained exact twenty-attempt example completes every success before any rejection. If at least ten of the twenty fair outcomes succeed, the first ten completed reports all succeed. The formal calibration construction then returns $[3/5, 1]$, excluding $1/2$, at nominal allowance $1/10$. That event has mass

$$\frac{308333}{524288} > \frac{1}{10}.$$

This is exact finite review evidence, not a newly added Lean theorem and not a counterexample to the source-generative formal theorem. It refutes a proposed physical interpretation that identifies completion-ordered reports with unbiased initiation outcomes without a coupling argument.

29.1 Censoring and conditional guarantees

The inherited archive also retains outcome-dependent censoring and conditioning-on-completion examples. An unconditional bad-issuance bound may hold while the same nominal bound conditioned on completion fails. A fault that consumes an attempted bit can bias retained reports. A nonconsuming fault in the formal model has a different meaning. Applications must specify which process they actually implement.

The remaining route is a precise physical-attempt-to-report-order model or an initiation-indexed protocol with proved treatment of pending holes, failures, replay, and ordering. Authentication, honest outcome checks, complete provenance, identity stability, and stationarity are additional obligations. No parser, ID invariant, or finite comparison silently supplies them.

30 What is closed and what remains open

The current source closes the pure finite recorder lifecycle, whole-run refinement to source prefixes, indexed Lean execution correspondence, and composition with fixed scheduled calibration and frozen-episode policy certificates. These are strong, precisely scoped results. They remain useful even though physical coupling is separate.

Open scope includes universal host/parser/storage refinement, authenticated providers and inputs, multi-episode error/reward/cost accumulation, permitted within-episode updates, empirical workload measurements, arbitrary real or infinite sources, drift, and unrestricted stopping. Earlier semantic backends retain their stated languages and cost models. The archive does not claim universal program minimization or measured LLM productivity.

The companion's fresh source inventory verifies the 195 non-audit module membership, the recorded line for all 2,268 named declarations, and the 188 theorem-bearing module digests supplied by the declaration-source receipt. It also checks the consistency of retained build, axiom, negative-test, and final release summaries. The full machine-readable result includes every immediate import, every

source hash, and the declaration statement excerpts. It labels all current kernel and actual Lean runtime executions as not run here.

A Source map and declaration inventory

The tables below are generated from the actual checkpoint. The machine-readable inventory retains full qualified names, source paths, source lines, imports, and SHA-256 digests. The source-only refresh command is

```
python3 scripts/lean_inventory.py
```

Run it from the publication root. Its result is a source/receipt check, not a Lean build. The formal project's own inventory and kernel audit remain the authoritative source for elaborated declarations and dependencies.

Module	Lines	Named	Immediate local imports
AdaptiveProbeCodec	120	0	AdaptiveRankProbe, AffineCodec
AdaptiveProbeControl	273	22	ExecutablePlanner
AdaptiveProbeExamples	99	10	AdaptiveProbeControl
AdaptiveRankProbe	69	4	AdaptiveProbeControl, AffineRankProbe
AffineCodec	155	2	AffineFusion
AffineExamples	68	12	AffineFusion
AffineFactorChoice	144	16	AffineFactorization
AffineFactorCodec	97	1	AffineFactorChoice, AffineCodec
AffineFactorExamples	85	16	AffineFactorCodec
AffineFactorization	136	16	AffineFusion
AffineFusion	105	12	TypedAffineProgram
AffineProbeCodec	116	2	PaidRankProbe, AffineReuseCodec
AffineProbeExamples	67	13	AffineProbeCodec
AffineRankProbe	81	9	AffineSearchGuard
AffineReuse	141	18	AffineFactorChoice
AffineReuseCodec	235	11	AffineSearchGuard, AffineFactorCodec
AffineReuseExamples	65	14	AffineReuseCodec, AffineFactorExamples
AffineSearchGuard	176	21	AffineReuse
AffineSignature	112	16	mathlib / none
AffineTerminal	202	17	AffineReuseCodec, MultiLevelEconomics
AffineTerminalCodec	149	3	AffineTerminal, MultiLevelCodec
ApproximateBayesian	266	8	BayesianEvidence, ApproximateStochastic
ApproximateBayesianCompression	67	2	ApproximateBayesian, BayesianCompression, ApproximateComposition
ApproximateComposition	130	5	ApproximateStochastic

Module	Lines	Named	Immediate local imports
ApproximateExamples	55	7	ApproximateComposition
ApproximateStochastic	443	19	StochasticQuotient
ArchiveExamples	129	13	ArchiveReplay, Retrospective
ArchiveReplay	233	12	HistoryState, OpenWorld, BayesianEvidence
ArtifactBinding	198	25	CertificateLogic
ArtifactExamples	79	14	CertificateResearch
ArtifactWorkflow	296	16	CertificateResearch, FilteredObservation
ArtifactWorkflowControl	170	13	ArtifactWorkflow
ArtifactWorkflowExamples	300	34	ArtifactWorkflowSource
ArtifactWorkflowGuarantee	151	8	ArtifactWorkflowControl, BayesianGuardedBridge
ArtifactWorkflowSource	109	8	ArtifactWorkflowGuarantee, BindingCodec
AtomicCommitModel	139	10	CommitRecovery
AttemptAccounting	330	29	mathlib / none
AttemptCodec	71	2	AttemptAccounting
AttemptExamples	32	9	AttemptAccounting
AttemptRecovery	143	10	AttemptAccounting, CommitRecovery
Bayesian	233	10	StochasticBlackwell
BayesianCompression	146	6	OpenWorld
BayesianControl	275	13	Bayesian
BayesianEvidence	161	6	BayesianControl
BayesianExamples	145	15	BayesianEvidence
BayesianGuardedBridge	168	5	BayesianObservationTransport, GuardedModelError
BayesianInstability	170	13	ApproximateBayesian, BayesianExamples
BayesianObservationTransport	202	11	ApproximateBayesian
BayesianTransportExamples	56	7	BayesianObservationTransport, BayesianExamples
BeliefValueTables	77	6	BayesianObservationTransport
BindingCodec	109	3	ArtifactBinding, CertificateCodec
Blackwell	141	5	Core
BooleanCostClosure	261	20	SemanticCostCertificate
BooleanProgram	230	23	mathlib / none
BoundedRecommendation	105	8	PlannerGuarantee
BoundedSynthesis	211	24	BooleanProgram
CalibratedPolicy	78	2	CalibrationSchedule, CalibrationBridge
CalibrationBridge	209	17	RobustQuoteCodec
CalibrationCodec	108	0	FiniteCalibration, CalibrationBridge
CalibrationLog	112	14	mathlib / none
CalibrationSchedule	290	20	FiniteCalibration
CertificateCodec	64	1	CertificateLogic
CertificateLogic	209	10	mathlib / none

Module	Lines	Named	Immediate local imports
CertificateResearch	143	9	ArtifactBinding, ResearchPolicy
CertifiedOperationalCatalogue	181	13	MultiLevelOperational
CommitRecovery	219	17	LiveExecution
Composition	37	3	Core
Context	153	7	Core
Control	222	9	Core
ControlSufficiency	395	20	Dynamic
ControlSufficiencyExamples	55	4	ControlSufficiency
Core	281	24	mathlib / none
CostCertificateExamples	54	8	SemanticLibrary
DurableUpgrade	375	22	VerifiedUpgrade
DurableUpgradeExamples	61	11	DurableUpgradeRuntime
DurableUpgradeLaws	215	15	DurableUpgrade
DurableUpgradeRuntime	140	4	DurableUpgradeLaws, UpgradeRuntime
Dynamic	73	4	Control
EpistemicState	77	5	State
ExactRateDistortion	495	30	RateDistortionLower
ExactRateDistortionExamples	90	7	GreedyRateDistortion, RateDistortionExamples
Examples	99	0	mathlib / none
ExecutablePlanner	179	11	GuardedModelError
FilteredObservation	91	8	BayesianObservationTransport
FiniteCalibration	231	23	mathlib / none
FiniteEvents	82	8	Bayesian
GreedyRateDistortion	269	18	ExactRateDistortion
GuardedControl	204	17	BayesianControl
GuardedModelError	168	11	GuardedPolicyRegret
GuardedPolicyRegret	129	9	GuardedControl
HistoryArchive	192	7	ArchiveReplay, HistoryState
HistoryExamples	66	4	HistoryState, OpenWorld
HistoryState	265	8	BayesianControl
HorizonBehavior	240	13	StateDependentControl
HorizonContext	94	4	HorizonBehavior, Context
HorizonDecisionRateDistortion	132	6	QRateDistortionFinite
HorizonExamples	36	3	HorizonContext
Investment	50	4	mathlib / none
LazyStagedCodec	252	0	LazyStagedRecovery, StagedAffineCodec
LazyStagedControl	463	32	StagedAffineControl, StagedAffineOperational
LazyStagedExamples	158	10	LazyStagedRecovery, StagedAffineExamples
LazyStagedRecovery	272	18	LazyStagedControl
LiveExecution	241	19	PlannerRuntime
LiveExecutionCodec	163	11	LiveExecution
ModelBridgeExamples	171	14	BayesianGuardedBridge, BeliefValueTables
MultiLevelCodec	161	0	MultiLevelRank, AffineCodec, PlannerRuntime

Module	Lines	Named	Immediate local imports
MultiLevelEconomics	76	6	MultiLevelRank, AffineSearchGuard
MultiLevelExamples	89	7	MultiLevelInformation
MultiLevelInformation	341	29	ExecutablePlanner
MultiLevelOperational	187	13	AffineTerminal, MultiLevelRank
MultiLevelRank	204	18	MultiLevelInformation, AffineRankProbe
ObservationCensoring	230	22	CalibrationSchedule
ObservationPolicy	166	7	ObservationStream, ObservationReveal, ObservedCalibration
ObservationRecorder	404	37	CalibrationLog
ObservationReveal	186	18	CalibrationLog
ObservationStream	296	24	CalibrationSchedule
ObservedCalibration	143	6	CalibratedPolicy
OpenWorld	146	10	BayesianEvidence, ApproximateComposition
OperationalExamples	169	14	CertifiedOperationalCatalogue
PaidRankProbe	245	20	AffineRankProbe
ParallelSchedule	110	8	ScheduleCompletion
Pareto	169	8	Control
PlannerExamples	52	11	WorkflowPlanner, ArtifactWorkflowExamples
PlannerGuarantee	155	9	PlannerWork
PlannerRuntime	242	11	WorkflowPlanner, BoundedRecommendation
PlannerWork	132	9	ExecutablePlanner
ProjectedStochasticSufficiency	274	13	StochasticSufficiency
ProofScheduling	217	23	mathlib / none
QRateDistortion	289	12	ExactRateDistortion
QRateDistortionExamples	85	8	QRateDistortionFinite
QRateDistortionFinite	283	20	QRateDistortion
QValueBridge	85	5	QRateDistortionFinite
QuoteAuthority	292	30	LazyStagedControl
RateDistortionExamples	30	2	RateDistortionLower
RateDistortionLower	165	7	ApproximateStochastic
RecorderCodec	81	1	ObservationRecorder, CalibrationCodec
RecorderPolicy	155	7	RecorderReveal, ObservationPolicy
RecorderReveal	320	24	ObservationRecorder, ObservationReveal
RecoveryCodec	84	4	AtomicCommitModel, LiveExecutionCodec
RecoveryExamples	45	6	RecoveryCodec
RecursiveSynthesis	161	19	SemanticIndex
Refinement	158	9	Core
RepresentationLattice	83	7	OpenWorld
ResearchController	266	18	GuardedControl, SchedulingCertificates

Module	Lines	Named	Immediate local imports
ResearchControllerExamples	296	29	ResearchPolicy, BayesianExamples
ResearchModelGuarantee	45	1	BayesianGuardedBridge, ResearchPolicy
ResearchPolicy	156	13	ResearchController, GuardedPolicyRegret
Retrospective	170	16	BayesianCompression, BayesianExamples
RetryBayesian	167	16	RetryControl, BayesianEvidence
RetryControl	217	15	GuardedPolicyRegret
RetryDecisionCodec	138	3	RetryLedger, AttemptCodec
RetryEvaluation	69	5	RetryBayesian
RetryExamples	128	18	RetryLedger
RetryLedger	96	5	RetryEvaluation, AttemptRecovery
RobustQuoteCodec	511	11	RobustQuoteRecovery, LazyStagedCodec, AffineReuseCodec, AffineFactorCodec
RobustQuoteExamples	240	15	RobustQuoteRecovery, LazyStagedExamples
RobustQuotePlanning	637	40	QuoteAuthority, GuardedModelError, ApproximateStochastic, ExecutablePlanner
RobustQuoteRecovery	238	10	RobustQuoteRuntime
RobustQuoteRuntime	295	20	RobustQuotePlanning
ScheduleCompletion	169	12	ProofScheduling
SchedulingCertificates	77	7	ScheduleCompletion
SchedulingExamples	74	9	ParallelSchedule, SchedulingCertificates
SemanticCostCertificate	93	6	BooleanProgram
SemanticIndex	250	27	BoundedSynthesis
SemanticLibrary	191	18	BooleanCostClosure
SemanticLibraryCodec	132	2	SemanticLibrary
SemanticPromotion	49	3	SemanticSearchBudget, SynthesisPromotion
SemanticSearchBudget	146	8	SemanticSynthesisWork
SemanticSynthesisExamples	51	9	SemanticSearchBudget
SemanticSynthesisRuntime	125	0	SemanticPromotion
SemanticSynthesisWork	177	17	RecursiveSynthesis
Specification	132	8	Verification
StagedAffineCodec	167	0	StagedAffineOperational, AffineTerminalCodec, PlannerRuntime
StagedAffineControl	310	22	AffineTerminal, ExecutablePlanner
StagedAffineExamples	181	13	StagedAffineOperational, AffineFactorExamples, AffineReuseExamples
StagedAffineOperational	129	13	StagedAffineControl
State	70	3	Core

Module	Lines	Named	Immediate local imports
StateDependentControl	261	12	ControlSufficiency
Stochastic	112	6	Blackwell
StochasticBlackwell	503	23	Stochastic
StochasticExamples	28	3	StochasticQuotient
StochasticQuotient	242	8	ProjectedStochasticSufficiency
StochasticSufficiency	116	4	StochasticBlackwell
SynthesisExamples	62	14	SynthesisPromotion
SynthesisPromotion	136	7	BoundedSynthesis, DurableUpgradeLaws
SynthesisRetry	43	4	SynthesisExamples, RetryEvaluation
SynthesisRuntime	195	2	SynthesisPromotion
TypedAffineProgram	153	10	AffineSignature
UpgradeRuntime	114	4	UpgradeSpecialization
UpgradeSpecialization	193	21	VerifiedUpgrade, RetryEvaluation
Verification	45	3	State
VerifiedUpgrade	321	27	mathlib / none
VerifierEnsemble	261	14	FiniteEvents, Specification
VerifierExamples	113	11	VerifierEnsemble, BayesianExamples
WorkflowPlanner	171	11	PlannerGuarantee, ArtifactWorkflowSource

B Selected exact theorem names

The following map gives the exact qualified names of every source excerpt in this companion. The corresponding namespace is the qualified prefix before the final declaration name. Source lines refer to v0.41.0, not a generated publication file.

Qualified declaration and module	Line
Komorebi.AVOC.refines_trans AVOC/Core.lean	32
Komorebi.AVOC.collision_obstructs_sufficiency AVOC/Core.lean	87
Komorebi.AVOC.sufficient_iff_fiber_refinement AVOC/Core.lean	124
Komorebi.AVOC.weakLinkUtility_positive_iff AVOC/Control.lean	174
Komorebi.AVOC.stochastic_finite_horizon_optimal_action_exists AVOC/Stochastic.lean	81
Komorebi.AVOC.quotientStochasticControlSystem_horizonValue_eq AVOC/StochasticQuotient.lean	151
Komorebi.AVOC.quotient_stochasticBellmanOptimalAt_iff AVOC/StochasticQuotient.lean	219
Komorebi.AVOC.approxBudget_closed_form AVOC/ApproximateStochastic.lean	264

Qualified declaration and module	Line
`Komorebi.AVOC.approximate_stochastic_horizon_value_bound_closed` `AVOC/ApproximateStochastic.lean`	386
`Komorebi.AVOC.bayesUpdate_mul_predictive` `AVOC/Bayesian.lean`	92
`Komorebi.AVOC.bayesianPolicyValue_eq_stochasticDecisionValue` `AVOC/Bayesian.lean`	215
`Komorebi.AVOC.conditionByWeight_l1_stability_combined` `AVOC/ApproximateBayesian.lean`	227
`Komorebi.AVOC.exists_qSignatureApproximation_iff_fiberDiameter` `AVOC/QRateDistortion.lean`	125
`Komorebi.AVOC.q_margin_preserves_unique_optimum` `AVOC/QRateDistortion.lean`	256
`Komorebi.AVOC.Guarded.safe_optimal_plan_exists` `AVOC/GuardedControl.lean`	177
`Komorebi.AVOC.Guarded.plan_regret_le` `AVOC/GuardedPolicyRegret.lean`	25
`Komorebi.AVOC.Guarded.decoded_action_gap` `AVOC/GuardedPolicyRegret.lean`	74
`Komorebi.AVOC.Guarded.decoded_policy_regret` `AVOC/GuardedPolicyRegret.lean`	102
`Komorebi.AVOC.Guarded.model_decoded_policy_regret` `AVOC/GuardedModelError.lean`	149
`Komorebi.AVOC.ExecutablePlanner.lookahead_policy_regret` `AVOC/PlannerGuarantee.lean`	123
`Komorebi.AVOC.ExecutablePlanner.charged_regret` `AVOC/PlannerGuarantee.lean`	147
`Komorebi.AVOC.sound_gate_acceptance` `AVOC/Verification.lean`	13
`Komorebi.AVOC.cannot_beat_shared_specification_floor` `AVOC/VerifierEnsemble.lean`	208
`Komorebi.AVOC.Certificate.check_sound` `AVOC/CertificateLogic.lean`	158
`Komorebi.AVOC.Binding.validated_sound` `AVOC/ArtifactBinding.lean`	72
`Komorebi.AVOC.Binding.successful_receipt_not_replayable` `AVOC/ArtifactBinding.lean`	161
`Komorebi.AVOC.Retry.choose_threshold` `AVOC/RetryControl.lean`	209
`Komorebi.AVOC.Attempts.charge_not_refunded_on_abandon` `AVOC/AttemptAccounting.lean`	206
`Komorebi.AVOC.Attempts.complete_trace_bounds` `AVOC/AttemptAccounting.lean`	321
`Komorebi.AVOC.CommitRecovery.acknowledge_idempotent` `AVOC/CommitRecovery.lean`	126
`Komorebi.AVOC.CommitRecovery.recovered_safe` `AVOC/CommitRecovery.lean`	170
`Komorebi.AVOC.Affine.RobustQuoteRecovery.quote_mutation_rejected` `AVOC/RobustQuoteRecovery.lean`	127
`Komorebi.AVOC.Affine.RobustQuoteRecovery.acknowledge_checked_transition` `AVOC/RobustQuoteRecovery.lean`	228

Qualified declaration and module	Line
`Komorebi.AVOC.Upgrade.funded_trial_is_charged` `AVOC/VerifiedUpgrade.lean`	136
`Komorebi.AVOC.Upgrade.certified_upgrade_campaign` `AVOC/VerifiedUpgrade.lean`	260
`Komorebi.AVOC.BoolClosure.query_globally_minimum` `AVOC/BooleanCostClosure.lean`	234
`Komorebi.AVOC.BoolClosure.query_idempotent_cost` `AVOC/BooleanCostClosure.lean`	249
`Komorebi.AVOC.Affine.finite_probe_complete` `AVOC/TypedAffineProgram.lean`	56
`Komorebi.AVOC.Affine.context_congr` `AVOC/TypedAffineProgram.lean`	94
`Komorebi.AVOC.Affine.Factorization.width_minimal` `AVOC/AffineFactorization.lean`	66
`Komorebi.AVOC.Affine.Factorization.checked_optimal` `AVOC/AffineFactorization.lean`	96
`Komorebi.AVOC.Affine.QuoteAuthority.identity_mismatch_rejected` `AVOC/QuoteAuthority.lean`	171
`Komorebi.AVOC.Affine.RobustQuote.certified_dominance` `AVOC/RobustQuotePlanning.lean`	506
`Komorebi.AVOC.Affine.RobustQuote.robust_policy_regret` `AVOC/RobustQuotePlanning.lean`	555
`Komorebi.AVOC.FiniteCalibration.excluded_mass_le` `AVOC/FiniteCalibration.lean`	153
`Komorebi.AVOC.FiniteCalibration.missMass_le` `AVOC/FiniteCalibration.lean`	172
`Komorebi.AVOC.FiniteCalibration.ten_successes_retained` `AVOC/FiniteCalibration.lean`	188
`Komorebi.AVOC.FiniteCalibration.ten_successes_lower` `AVOC/FiniteCalibration.lean`	198
`Komorebi.AVOC.CalibrationSchedule.scheduled_failure_le` `AVOC/CalibrationSchedule.lean`	74
`Komorebi.AVOC.CalibrationSchedule.selected_failure_le` `AVOC/CalibrationSchedule.lean`	83
`Komorebi.AVOC.ObservationStream.conditional_implies_product` `AVOC/ObservationStream.lean`	132
`Komorebi.AVOC.ObservationStream.prefix_count_fiber_eq_binomial` `AVOC/ObservationStream.lean`	250
`Komorebi.AVOC.ObservationRecorder.run_wellFormed` `AVOC/ObservationRecorder.lean`	238
`Komorebi.AVOC.ObservationRecorder.accepted_terminal_retained_once` `AVOC/ObservationRecorder.lean`	395
`Komorebi.AVOC.ObservationRecorder.fault_preserves_samples` `AVOC/ObservationRecorder.lean`	341
`Komorebi.AVOC.ObservationRecorder.inherited_unfinished_terminal_conflict` `AVOC/ObservationRecorder.lean`	359
`Komorebi.AVOC.RecorderReveal.run_refines` `AVOC/RecorderReveal.lean`	224
`Komorebi.AVOC.RecorderReveal.run_samples_eq_take` `AVOC/RecorderReveal.lean`	234

Qualified declaration and module	Line
Komorebi.AVOC.RecorderPolicy.pending_run_no_plan AVOC/RecorderPolicy.lean	79
Komorebi.AVOC.RecorderPolicy.gatedCounts_faithful AVOC/RecorderPolicy.lean	37
Komorebi.AVOC.RecorderPolicy.recorder_issued_failure_le AVOC/RecorderPolicy.lean	129
Komorebi.AVOC.RecorderCodec.replayFrom_toOption AVOC/RecorderCodec.lean	18
Komorebi.AVOC.Affine.CalibrationBridge.competitor_dominance_sound AVOC/CalibrationBridge.lean	21
Komorebi.AVOC.Affine.CalibrationBridge.checked_competitor_dominance_optimal AVOC/CalibrationBridge.lean	57

C Retained formal and executable reference files

- START_HERE.md, FORMAL_THEORY.md, PROOF_BOUNDARY.md: current scope, mathematical composition, and explicit open boundaries.
- THEOREM_INDEX.md, results/v0.41.0/declaration_inventory.json: exact 2,268-name source inventory.
- AUDIT.json, results/v0.41.0/axiom_audit_summary.json, results/v0.41.0/full_axiom_audit.log: retained axiom coverage.
- results/v0.41.0/build_progress.json, results/v0.41.0/release_verification.json: clean target and release receipts.
- RECORDER_EXECUTABLE.md, OBSERVATION_EXECUTABLE.md: precise endpoint and bounded observation experiment contracts.
- results/v0.41.0/negative_all/summary.json and its per-control logs: exact substantive-rejection evidence.
- results/v0.41.0/oracle_run/manifest.json, results/v0.41.0/oracle_results.json: runtime-bound new endpoint evidence.
- results/v0.41.0/review_completion_order.py and its JSON result: exact ordering obstruction outside the source-generative premise.
- EXTERNAL_TRUST.md, RECOVERY_FORMAT.md, LIBRARY_FORMAT.md: physical storage, journal, and certificate format boundaries.
- REPRODUCE.md, lean-toolchain, lake-manifest.json: pinned reproduction inputs and workflow.