Macroscopic Interpretability for Autonomous Agents: A Trace-First Approach

Dendrer: Procedural Causality and Agent Version Control

Abstract

Consider an autonomous agent that violates a compliance policy at tick 40. Post-mortem investigation reveals the cause: a low-confidence observation committed at tick 12 quietly corrupted the agent's state, making the violation at tick 40 structurally inevitable. Without the right infrastructure, this causal chain is invisible — all you have is a policy violation and a pile of logs. With it, `bisect` locates tick 12 in $\lceil \log_2 40 \rceil = 6$ steps.

Existing interpretability research treats understanding as a property of model internals: representations, activations, and attention dynamics. We show that, under standard sources of nondeterminism in agentic systems, sound causal localization over trajectories requires a replayable, immutable transition trace. More precisely, we prove an impossibility result: within the class of agent executions representable as sequences of committed transitions applied by a reducer, removing any of three invariants — immutability, reducer purity, predicate derivability from committed state — yields executions where bisect cannot be guaranteed to identify the originating transition of a behavioral failure.

We introduce **macroscopic interpretability**: a redefinition of interpretability for long-horizon autonomous systems as *procedural causality* — identifying which externally committed state transition caused a behavioral outcome — rather than *representational causality* — identifying which internal computation produced a model output.

We formalize the minimal conditions under which trajectory-level causal inference is sound. "Minimal" means: remove any single invariant and there exists an execution in our model class where bisect cannot be guaranteed. We derive Agent Version Control (AVC): primitives — fork, diff, bisect, cherry-pick — that become possible under these conditions and unsound without them.

We further argue that trust in autonomous systems should be modeled on institutional accountability rather than cognitive transparency: agents treated as institutional actors in accountable environments, not transparent cognitive objects whose internals must be fully understood before they can be trusted.

---

1. Introduction

Autonomous agents — systems that pursue goals over extended time horizons using external tools, accumulated memory, and sequences of decisions — present interpretability challenges that are qualitatively different from those of static language models.

For a model answering a bounded query, the relevant unit of analysis is the forward pass: what internal representations were activated, which tokens attended to which, what circuits produced the output. Microscopic interpretability methods — mechanistic circuit analysis, probing classifiers, activation patching — address this level productively.

For an agent pursuing a goal across dozens or hundreds of decision steps, the relevant unit of analysis is the *trajectory*: the full sequence of perceptions, decisions, actions, and state changes from goal declaration to final artifact. Trajectories are temporal, systemic, and emergent. A failure at step 40 may be causally traceable to a low-confidence observation at step 12. No microscopic analysis of step 40 in isolation can recover this causal structure.

We do not argue that microscopic interpretability is wrong. We argue that it addresses a different causal question. Microscopic interpretability asks: *what internal computation produced this output?* — representational causality. Macroscopic interpretability asks: *what sequence of externally committed state transitions caused this behavior?* — procedural causality. For long-horizon agentic systems, procedural causality is the primary unit of interpretability. A failure at step 40 whose origin lies in a low-confidence observation at step 12 is not recoverable by representational analysis of step 40. It requires procedural analysis of the full trajectory.

These are orthogonal questions. We address the second.

Converging demand, absent formal substrate. Several concurrent research threads are independently identifying this gap. Work on long-horizon mechanistic interpretability (2026) calls for mid-trajectory interventions and credit assignment across tool-use sequences, but proposes no runtime primitives. AgentRR (2025) records and replays agent traces for safety conformance, but relies on heuristic summarization rather than pure predicates or formal bisect. CausalPlan (2025) applies structural causal models to multi-agent planning to identify invalid causal actions — the right causal framing, but at the planning layer rather than the execution trace. Synergetics (2025) documents version-control conflicts and loss-of-context in production agent deployments as concrete pain points. Each thread demands causal navigability over trajectories. None proposes a formal substrate for it.

This paper provides that substrate: immutable traces with predicate-derivability, prefix-monotone bisect with soundness guarantees, and contract-enforced context safety for AVC operations. The timing is not coincidental — it is a response to converging demand from multiple directions arriving simultaneously.

1.1 Contributions

First, we formalize macroscopic interpretability as deterministic replayability over an immutable, append-only event trace, and introduce *procedural causality* as the correct unit of interpretability for long-horizon agentic systems. We specify the minimal invariants this requires: immutability of the committed trace, reducer purity, and predicate derivability from committed state. Canonical state serialization, epsilon-quantization, and intent/transition separation are the engineering instantiations of these three invariants — they are how the invariants are satisfied in practice, not additional invariants.

Second, we establish a causal underdetermination result relative to a well-defined model class: agent executions representable as sequences of committed transitions applied by a reducer. Within this class, removing any single invariant — immutability, reducer purity, predicate derivability — yields executions where bisect cannot be guaranteed to identify the originating transition of a behavioral failure. “Minimal” means: each invariant is independently necessary; the set is jointly sufficient.

Third, we introduce the audit fidelity / behavioral equivalence distinction (Section 3.4):

the audit substrate must be deterministic; the world need not be.

Fourth, we derive AVC — fork, diff, bisect, cherry-pick — as primitives that are sound under these conditions and introduce three mechanisms that make this tractable in practice: replay isolation, intent context hash, and the progress contract interface.

Fifth, we argue that trust in autonomous systems should be modeled on institutional accountability rather than cognitive transparency — and that this is not merely a useful framing but the correct unit of analysis for systems whose behavior emerges from sequences of committed decisions.

1.2 Scope

This paper addresses **debugging, auditability, reproducibility, and operational trust.**

We make no claims about alignment, safety, or fundamental cognitive understanding. The determinism guarantees described here are conditional on controlled execution environments; we discuss the conditions and their limits in Section 8.

---

2. Representational vs. Procedural Causality

Microscopic interpretability methods have produced genuine scientific value. Circuit-level analysis has revealed how models implement modular computations. Probing classifiers have shown that internal representations encode structured knowledge. Activation patching has enabled causal tracing within individual forward passes.

These methods address *representational causality*: identifying which internal computation produced a given model output. This is the right question for bounded, stateless inference.

For autonomous agents operating over extended trajectories, the relevant causal question is different: *what sequence of externally committed state transitions caused this behavioral outcome?* We term this **procedural causality**.

The distinction matters because procedural failures — the class of failures most common in deployed agent systems — are invisible to representational analysis:

- A failure at tick 40 caused by a low-confidence observation at tick 12 cannot be recovered by analyzing the model's internal state at tick 40 in isolation.
- A behavioral drift that compounds across 30 ticks has no single representational cause — it is an emergent property of the trajectory.
- A policy violation that results from a sequence of individually valid decisions has no internal representation to probe — it exists only at the trajectory level.

Procedural causality requires a faithful, complete, replayable record of the trajectory. Representational causality does not. These are orthogonal analytical frameworks addressing orthogonal failure modes. A complete interpretability program for autonomous agents requires both.

For an operator deploying an autonomous agent in production, the questions that determine trust are procedural:

- Why did the agent take that action at step 23?
- At what point in the execution did reasoning degrade?
- Is the failure reproducible, or was it a one-off?
- Can we prove that two runs of the same goal produced equivalent behavior?

None of these questions are answered by analyzing internal representations. They require a faithful external record of what the system did — ordered, typed, replayable, navigable.

This is the gap macroscopic interpretability addresses.

---

3. Formalizing Macroscopic Interpretability

3.1 The Trace as Primary Artifact

We define a **trace** T as an append-only, ordered sequence of typed state transitions:

```
T = [t$_1$, t$_2$, ..., t$_n$]
```

where each transition t_i is a tuple:

```
t$_i$ = (tick, ts, agent_id, goal_id, transition_type, state_before,
    state_after,
        intent, action, result, confidence, uncertainty_notes, policy,
    anomalies)
```

Invariants: - Append-only: no transition is modified or deleted after commit - Ordered: t_i.tick < t_{i+1}.tick for all i - Typed: transition_type drawn from a locked ontology - Sufficient: T is sufficient to deterministically reconstruct any state s_i - Hash-chained: each transition commits a hash over its payload and its predecessor

Hash chaining (immutability invariant):

The append-only claim is not self-enforcing without a cryptographic commitment. Each committed transition carries a chain hash:

```
t_1.chain_hash = H(serialize(t_1.payload))
t_i.chain_hash = H(serialize(t_i.payload) || t_{i-1}.chain_hash)  for i >
    1
```

This makes tampering detectable: modifying any t_i invalidates the chain hash of every subsequent transition. Verification is O(N) — replay the chain and confirm each hash. The chain need not be anchored externally to provide meaningful tamper evidence within a run; optional periodic export of the chain tip (a Merkle root) to an external append-only log strengthens the guarantee for multi-party audit contexts.

This closes the gap between the architectural claim ("the trace is immutable") and the operational guarantee ("we can detect if it was mutated"). Without hash chaining, immutability is a policy claim. With it, immutability is a verifiable property.

3.2 The Intent / Transition Separation

The central architectural decision that makes the trace faithful:

The LLM never mutates state directly. It proposes intent. The runtime commits transitions.

Formally: - The LLM produces an *intent* I: a structured proposal containing transition_type, confidence, uncertainty_notes, and proposed action - The runtime validates I against policy, computes state_after via a deterministic reducer function R: (state_before, intent) $\rightarrow$ state_after - The runtime commits the resulting transition t_i to the trace

This separation ensures that the trace records what *actually happened*, not what the model believed was happening. It is the foundation on which all AVC primitives depend.

3.3 Deterministic Replayability

We define macroscopic interpretability formally as:

> **A system is macroscopically interpretable if and only if, given trace T and initial state s_0, the reducer R can deterministically reconstruct any state s_i by replaying transitions $t_1 \ldots t_i$.**

Distinction from event sourcing: Traditional event sourcing guarantees reconstructability of state from a sequence of recorded events. Macroscopic interpretability requires two additional properties: *predicate-derivability* — behavioral predicates must be evaluable as pure functions over committed state alone — and *prefix-monotonicity* — violation predicates must be stable once triggered. Together these properties support sound causal localization via bisect. Without them, a system may be reconstructable but not causally navigable.

This requires: - **Canonical state serialization:** deterministic field ordering, normalized numeric comparison with explicit tolerance thresholds, no non-deterministic values in diffable state - **Pure reducer:** R is a pure function; same inputs always produce same outputs. Critically, R must handle floating-point comparison via epsilon tolerance rather than exact equality. Confidence scores and progress deltas are floats; bitwise equality checks will produce false diffs and false replay divergences. Define explicit epsilon thresholds per field type (e.g., `confidence_epsilon = 0.001`) and encode them in the schema, not in ad-hoc comparator logic.

Float handling in canonical serialization: ε-tolerance is implemented as *quantization during serialization*, not as comparator magic at diff time. Before hashing or comparing state, floats are rounded to the declared precision (e.g., confidence rounded to 3 decimal places). This means σ(state) is stable across floating-point representations and produces consistent bytes for hashing. Comparators then use exact byte comparison on the serialized form. One rounding rule per field type, declared in the schema — not inferred at comparison time. - **Anomaly derivability:** anomaly detection is a pure function over committed state; no dependence on external calls, randomness, or time

If any invariant fails, replayability fails, and AVC collapses.

Feasibility assumption: these invariants require a snapshotable execution environment (Firecracker, WASM). See Section 9.2.

3.3.1 Causal Underdetermination (The Impossibility Argument)

Model class M (formally):

Let M be the class of agent executions (T, R, s_0, P) where: - T = [$t_1 \ldots t_n$] is a finite, ordered sequence of committed transitions - R : State × Transition → State is the reducer - s_0 is the initial state - P : State → {ok, violation} is the behavioral predicate - $s_i = R(s_{i-1}, t_i)$ for all i ∈ [1..n]

Theorem (Causal Underdetermination): Within model class M, if any of the following conditions hold — (1) T is mutable, (2) R is not a pure function, or (3) P depends on context outside committed state — then there exists no deterministic algorithm A with access only to (T, R, s_0, P) that can guarantee identification of the violation onset tick $k^* = \min\{k : P(s_k) = \text{violation}\}$.

Proof by counterexample construction, one per condition:

(1) Mutable trace. Let T be a trace where $P(s_{12})$ = ok at audit time, but t_{12} was modified after original commitment. The original execution had $P(s_{12})$ = violation; the modification replaced t_{12} with t'_{12} such that $R(s_{11}, t'_{12})$ yields a state where P = ok. Any algorithm A operating over the current T will reconstruct a trajectory consistent with T and conclude $k^* \neq 12$. But the true onset was 12. A cannot detect this because it has no access to the pre-modification record. Since T can be modified to make any tick appear as the onset, A cannot guarantee correct identification for any execution in this class. □

(2) Impure reducer. Let R be nondeterministic: on two evaluations of the same (state, transition) pair, R may return s or s' where P(s) = ok and P(s') = violation. Construct T where the true onset is tick k. A calls replay(T[:k], R, s_0) and observes $P(s_k)$ = ok (R happened to return s at this evaluation). A concludes $k^* > k$ and searches the right half. On a subsequent evaluation at tick k, R returns s', $P(s_k)$ = violation, but A has already committed to the wrong half. The returned onset tick is wrong. Since R's nondeterminism can place the apparent onset anywhere in [k, n], A cannot guarantee correctness. □

(3) External predicate. Let P(state) call an external API that returns values v_1, v_2 at different times such that $P(s_k)$ = ok at time τ_1 and $P(s_k)$ = violation at time τ_2, for the same committed state s_k. Construct T where the true onset is k. A evaluates $P(s_k)$ at time τ_1 and observes ok; at time τ_2 it observes violation. The binary search partition is unstable: the same tick may be "good" in one half of the search and "bad" in the re-evaluation. A has no mechanism to distinguish the onset from any other tick, because P provides no stable signal over committed state. A cannot guarantee correctness for any execution where the external state changes between evaluations. □

In all three cases, A may return an incorrect onset tick with no way to distinguish a correct result from an incorrect one.

Note on scope: This result holds within M. Richer recording schemes — logging additional context, snapshotting more state — remain within M if they satisfy the three conditions. The theorem does not preclude such extensions; it specifies what the recording scheme must guarantee as a minimum.

3.3.2 Formal Soundness

We define the formal objects and state the soundness condition.

Two distinct failure signal types (previously conflated as "anomaly"):

- `kernel.anomaly` — runtime-detected invariant violations: replay divergence, schema mismatch, missing recorded tool output, reducer non-determinism. Always decidable. Non-negotiable. Emitted by the runtime, not by domain logic.
- `predicate.violation` — institution-defined policy breaches: compliance failures, risk threshold crossings, quality metric degradation. Pluggable via ProgressContract. Domain-specific. Decidable only if the predicate satisfies purity requirements.

```
Let:
  T  = agent trajectory (ordered, finite sequence of committed transitions
    ) in M
  R  = reducer: (state, transition) \textrightarrow{} state           --
    must be pure
  K  = kernel.anomaly function: state \textrightarrow{} {anomaly | $\
    emptyset$} -- always pure; runtime-enforced
  P  = predicate.violation: state \textrightarrow{} {violation | $\
    emptyset$}   -- pure iff registered via ProgressContract
  $\sigma$  = canonical serialization: state \textrightarrow{} bytes
      -- deterministic, $\varepsilon$-tolerant for floats

A trajectory-level interpretability system is sound iff:
  1. R is pure:          $\forall$ s, t:  R(s, t) = R(s, t)          (
    deterministic reconstruction)
  2. $\sigma$ is canonical:     $\forall$ s:     $\sigma$(s) is
    deterministically ordered with $\varepsilon$-tolerant comparison
  3. K is derivable:     $\forall$ t$_i$$\in$T:  K(t$_i$.state_after)
    requires no external context
  4. T is replayable:    Replay(T, s$_0$) = s$_n$                  (
    reconstruction holds end-to-end)
```

Under these conditions, bisect over `kernel.anomaly` is unconditionally sound. Bisect over `predicate.violation` is sound iff the registered predicate satisfies purity — enforced at registration time, not assumed.

These are the **minimal invariants for macroscopic interpretability** within model class M: each is independently necessary; together they are sufficient for sound bisect (see Section 3.3.3 for the additional predicate regularity requirement).

Predicate evaluation window. A predicate P may evaluate over `state_before` (the state prior to the transition being committed) or `state_after` (the state produced by the reducer after the transition). This must be declared explicitly in the contract — the evaluation window is not assumed. For instantaneous predicates (e.g. `risk_score > threshold`), the distinction rarely matters. For accumulating predicates (e.g. `total_paid_to_date > policy_limit`), it is critical: evaluating `state_before` misses the contribution of the current transition entirely. The default is `state_after` unless the contract specifies otherwise.

3.3.3 Predicate Regularity (Prefix-Monotonicity)

Soundness of the formal conditions in 3.3.2 is necessary but not sufficient for bisect. Binary search requires an additional property of the predicate itself.

The requirement: P must be **prefix-monotone** — once it returns `violation` at tick k, it must continue to return `violation` for all $j > k$. Equivalently, the set of violating ticks forms a contiguous suffix interval [k..n]. This is exactly the stability condition binary search requires.

Why it matters — a concrete example:

Consider a financial agent where P evaluates `state.risk_score > threshold`. A risk score that crosses the threshold and stays crossed is prefix-monotone:

```
tick:          1    2    3    4    5    6    7    8
risk_score:    0.61 0.68 0.71 0.79 0.83 0.87 0.91 0.94
P(s):          ok   ok   ok   VIOL VIOL VIOL VIOL VIOL
                              $\uparrow$ k*=4: onset tick, bisect returns
   this correctly
```

Binary search on this trace: test tick 4 → violation, test tick 2 → ok, test tick 3 → ok → onset = tick 4. Correct.

Now consider a quality score that oscillates above and below threshold:

```
tick:          1    2    3    4    5    6    7    8
quality:       0.9  0.7  0.8  0.6  0.9  0.7  0.8  0.6
P(s):          ok   VIOL ok   VIOL ok   VIOL ok   VIOL
```

Binary search: test tick 4 → violation, test tick 2 → violation, test tick 1 → ok → returns onset = tick 2. But tick 3 is ok. The returned onset is wrong — there is no stable onset tick. Bisect cannot be sound over this predicate.

Separation of concerns:

- *Required for bisect:* prefix-monotonicity of P
- *Not required for replayability:* a non-monotone predicate is still pure and replayable — it can be used for monitoring, dashboards, and diff. It is rejected only at bisect registration time.

The monotone lift $P^{\uparrow}$ — bridging real-world oscillation.

Most real-world metrics oscillate: risk scores fluctuate, budgets recover, compliance confidence varies. This does not prevent bisect — it requires a precise distinction between the *base predicate* P and the *bisect predicate* $P^{\uparrow}$.

Bisect does not require domain reality to be monotone. It requires the predicate *used for bisect* to be monotone. A non-monotone base predicate P can be lifted to a monotone predicate $P^{\uparrow}$ via the **monotone closure**:

```
$P^{\uparrow}$(s_k) = violation  iff  there exists j <= k such that P(s_j)
    = violation
```

$P^{\uparrow}$ latches on first violation and never recovers. This does not distort business logic — P remains available for monitoring and dashboards. It changes the bisect lens: $P^{\uparrow}$ finds the tick where the domain first entered a violation-eligible state, regardless of subsequent recovery.

```
base predicate P (oscillating, not bisect-safe):
tick:     1    2    3    4    5    6    7    8
quality:  0.9  0.7  0.8  0.6  0.9  0.7  0.8  0.6
P(s):     ok   VIOL ok   VIOL ok   VIOL ok   VIOL

monotone lift $P^{\uparrow}$ (bisect-safe):
$P^{\uparrow}$(s):  ok   VIOL VIOL VIOL VIOL VIOL VIOL VIOL
```

```
               k*=2: first violation onset, bisect returns correctly
```

The contract schema expresses this explicitly:

```
violations:
  - id: quality_degradation
    base_predicate: "state.quality_score < 0.75"
    bisect_predicate: monotone_lift  # kernel applies $P^{\uparrow}$
   automatically
    pure: true
```

This is not a workaround. It is a formal monotone lift — a standard construction from order theory applied to predicate sequences. The institution defines P (what constitutes a violation in domain terms); the kernel applies $P^{\uparrow}$ for bisect. The onset tick returned is the first tick where P was ever true — the causally meaningful answer for remediation.

`kernel.anomaly` is unconditionally prefix-monotone by construction: once an invariant is breached and emitted to the trace, it remains in the trace. Institutions defining custom predicates via `ProgressContract` may register either a prefix-monotone P directly, or a base predicate for which the kernel applies $P^{\uparrow}$, enforced at registration via the `--verify-monotone` flag (see Appendix C).

Two formally distinct outputs of bisect:

1. **Violation onset tick k*** — the first k such that `P(s$_k$) = violation`. What bisect computes. Requires prefix-monotonicity. Exact.
2. **Causal contributor set** (heuristic) — the minimal set of earlier transitions whose removal would have prevented the violation. Requires counterfactual analysis via fork. Dendrer provides this as `causal_chain` output, labeled heuristic. Do not conflate with the onset tick.

3.4 Audit Fidelity vs. Behavioral Equivalence

The mitigations described above — recording LLM outputs, excluding wall-clock time, recording tool outputs for replay — mean that replay reconstructs *what was recorded*, not *what the agent would do if run again live.*

This is **audit fidelity**, not **behavioral equivalence**:

- **Audit fidelity:** the trace reconstructs every state and decision. Sufficient for post-mortem analysis, bisect, and regulatory compliance.
- **Behavioral equivalence:** the agent would produce the same trajectory if re-run. A much stronger claim that depends on model determinism, environment stability, and tool idempotency in ways that are not fully controllable in production.

AVC primitives — fork, diff, bisect, cherry-pick — require only audit fidelity. Behavioral equivalence is out of scope.

A third boundary condition — epistemic truth:

Macroscopic interpretability does not guarantee epistemic truth of inputs. It guarantees procedural accountability over committed state transitions. An agent that receives and commits a plausible-but-false observation is behaving correctly from the kernel's perspective: the transition is immutable, replayable, and auditable. Whether the observation was true is a separate question. Truth validation and source verification are orthogonal concerns implementable as policy layers over the same trace substrate — for example, as a `ProgressContract` predicate that scores artifact provenance — but are not provided by the kernel and are not claimed by this paper.

On restore from snapshot, the runtime replays the trace tail against the snapshot state and verifies that the reconstructed state matches the committed state_after of the final transition. On divergence, the runtime emits a `kernel.anomaly` event and fails closed.

If execution cannot be replayed deterministically, it is considered a failure.

3.5 Handling External Nondeterminism

A critical clarification: the system is not fragile against external chaos. It is fragile against **chaos leaking into the replay boundary**. These are different failure modes with different mitigations.

The `kernel.anomaly` invariant applies to *internal* divergence — replay producing a different state than what was committed. It does not apply to the external world changing between a live run and an audit replay. The external world is *expected* to change. The runtime must ensure it never reaches the reducer during replay.

The Dirty Trace Scenario

Consider an agent that calls a weather API during a live run on Monday, receives `"severe storm"`, and cancels logistics accordingly. On Friday, an auditor replays the trace to verify the decision. The API now returns `"clear skies"`.

Without replay isolation, the reducer sees `"clear skies"`, produces a different `state_after`, diverges from the committed trace, and emits `kernel.anomaly`. The audit fails. The system blames itself for external change. This is wrong.

With replay isolation, the GenServer intercepts the tool call and injects the historically recorded payload. The external world is frozen. The reducer sees `"severe storm"` exactly as it did on Monday. Replay succeeds. The decision is auditable.

Replay Isolation: Conceptual Implementation

```
function execute_tool(tool, params, env):
    if env.mode == LIVE:
        result = tool.call(params)          # real world call
        record(env.tick, tool.name, result) # store in trace
        return Ok(result)

    if env.mode == REPLAY:
        recorded = lookup(env.recorded_outputs, env.tick)
        if recorded is None:
            # Tool was called here in original run but output missing
            # Structural divergence --- emit kernel.anomaly
            return Error(replay_tool_missing, env.tick, tool.name)
        return Ok(recorded.output)          # inject historical payload
                                            # never call the real tool
```

In replay mode, the function performs no IO. The supervisor may restart the process, but it is initialized in REPLAY mode with recorded outputs from the snapshot. External chaos is encapsulated — not permitted to reach the reducer.

See Appendix E for the Elixir/OTP implementation.

The invariant restated precisely:

> The system fails closed when *internal* state diverges from committed state. It handles *external* nondeterminism by encapsulation — recording outputs at commit time and injecting them at replay time. The two failure modes are architecturally distinct and handled differently.

On irreversible real-world effects:

Replay does not undo side effects. If the agent deleted a file at tick 23, the file remains deleted after replay. This is correct behavior, not a limitation. The kernel guarantees that the *committed record of what the agent decided and observed* is replayable — not that the external world effects of those decisions are reversible. At tick 23 during replay, the executor injects the historically recorded tool output (`{result: "file deleted, ok"}`) without re-executing the deletion. The agent's reasoning trajectory is faithfully reconstructed; the filesystem is not touched.

This is the same guarantee `git log` provides: every commit that deleted a file is permanently in the history, auditable and navigable, without un-deleting the file. Reversibility of external effects is an orthogonal concern — addressable via compensating transactions or reversible execution environments — and is outside the scope of this paper.

This is not a relaxation of determinism. It is the mechanism that makes determinism achievable in a non-deterministic world.

On the relationship between tracing and cognition:

The trace is not a constraint on cognition; it is a constraint on commitment. Agents may explore freely within a tick — reasoning, hypothesizing, reconsidering. Only externally meaningful state transitions must be ledgered. The boundary is between speculation and commitment, not between thought and action.

---

4. Agent Version Control (AVC)

Pseudocode convention: Python-style throughout. The contract reference implementation (Section 4.4) uses Elixir for pattern-matching clarity; Appendix E is Elixir throughout. Language choice is an implementation detail.

The trace is not a log. It is a versioned history of cognition.

Once the trace satisfies the invariants in Section 3, a version control primitive set becomes possible — analogous to VCS for code, but operating on reasoning trajectories.

4.1 Fork

Analogy: `git branch`

Spawn a new agent from any point in an existing trace.

```
dendrer fork --run <run_id> --from tick <N>
```

- Load snapshot at or before tick N
- Replay transitions up to tick N (replay verification mandatory)
- Start new agent with new run_id from restored state
- Emit `run.fork` transition with lineage metadata:

```
{
  "transition_type": "run.fork",
  "parent_run_id": "...",
  "parent_tick": N,
  "fork_reason": "..."
}
```

Fork enables controlled experimentation: same goal, same history to tick N, different policy or tool configuration from that point forward.

4.2 Diff

Analogy: `git diff`

Compare two traces transition by transition. Diffs are semantic, not textual.

```
dendrer diff <runA> <runB> [--from tick N] [--to tick M]
```

Diff types: - `structural` — different transition types at same tick - `semantic` — same type, different intent/action/result - `confidence` — same transition, different confidence score - `outcome` — same path, different final state

Critical requirement: canonical state serialization (Section 3.3) must hold. A diff that lies is worse than no diff. Floating point without tolerance thresholds, unordered maps, and timestamps in state all produce misleading diffs.

4.3 Bisect

Analogy: `git bisect`

Binary search a trace to find the originating transition of a behavioral failure. Most agent failures are not where they appear. A policy violation at tick 40 often originates from a flawed observation at tick 12. Bisect finds tick 12.

```
dendrer bisect <run_id> --anomaly <type>
dendrer bisect <run_id> --policy-flip <transition_type>
dendrer bisect <run_id> --predicate "state.risk_score > threshold"
dendrer bisect <run_id> --divergence <runB>
```

Bisect is grounded in invariants, not progress.

The primary bisect signal is hard, derivable predicates — not heuristic progress scores:

- `kernel.anomaly` events — runtime-detected invariant violations
- `policy.decision` flips — a transition allowed at tick 5 denied at tick 13
- State predicate failures — `risk_score > threshold`, `budget_remaining < 0`

- Invariant breaches — sequence violations, schema mismatches, reducer divergence

These are binary, deterministic, and derivable from committed state alone. They do not require progress quantification. They do not require heuristics. They are either present in the committed state or they are not.

Progress-based bisect (`--regression`) is supported as an optional extension (see Appendix D) but is explicitly not a core invariant. If the progress signal degrades, bisect over hard predicates remains sound.

Why this requires anomaly functions to be pure reducers:

Git bisect works because test suites are deterministic and good/bad is binary. Dendrer bisect requires the same: the anomaly predicate must be a pure function over committed state — no I/O, no external calls, no randomness, no time.

If an anomaly function performs I/O, it must be rejected at registration time. This is not a guideline. It is a runtime enforcement requirement. If this slips, bisect becomes flaky, and the entire AVC value proposition crumbles.

Example: policy flip bisect

```
dendrer bisect trip-planner-a --policy-flip action.request

\textrightarrow{} BISECT: testing tick 20... policy=deny FAIL
\textrightarrow{} BISECT: testing tick 10... policy=allow OK
\textrightarrow{} BISECT: testing tick 15... policy=deny FAIL
\textrightarrow{} BISECT: testing tick 12... policy=allow OK
\textrightarrow{} ORIGIN: tick 13 | transition_type=observation.add
  intent: "retrieved hotel pricing from unverified source"
  confidence: 0.31 | uncertainty_notes: "source reliability unknown"
  effect: risk_score crossed threshold \textrightarrow{} policy flipped to
     deny at tick 14

\textrightarrow{} CAUSAL CHAIN:
  tick 13  observation.add      \textrightarrow{} introduced unverified
    source data
  tick 14  policy.decision      \textrightarrow{} risk_score=0.82 exceeded
     threshold=0.75; deny
  tick 17  action.request       \textrightarrow{} denied; agent attempted
    workaround
  tick 20  kernel.anomaly       \textrightarrow{} invariant breach
    detected

\textrightarrow{} SUGGESTION: fork --from tick 12 --policy "require source
     verification"
```

The causal chain is the output, not just the origin tick.

Bisect does not terminate at identification. It traces the propagation: which subsequent transitions were causally downstream of the origin, what state changes they induced, and how the failure compounded. This transforms bisect from a pointer into a narrative — the story of how one low-confidence observation at tick 13 became a kernel anomaly at tick 20.

This is the feature that makes macroscopic interpretability operationally powerful. It transforms "something went wrong" into "here is exactly how it went wrong, transition by transition, and here is where to fork to prevent it."

4.4 Context-Safe AVC: The Intent Context Hash

A critical distinction between code version control and agent version control:

Code is inert. A cherry-picked Python function has no memory, no accumulated context, no observations that justified its existence. The worst case is a syntax error caught at compile time.

Agent intent is contextual. An intent to execute a high-risk action was justified by specific observations in a specific state context. Cherry-picking that intent into a branch that lacks those observations is not a version control operation — it is a context violation that produces silent disasters.

Example: The Financial Agent

Branch A: agent observes `user.risk_threshold = MAX`, decides to execute a large buy order. Intent committed at tick 23. An engineer cherry-picks that buy intent into Branch B — a conservative simulation where `user.risk_threshold = MIN`. Under naive Git semantics, Branch B executes the large buy order and destroys the portfolio. No error. No warning. No crash. Silent disaster.

The architectural problem: who decides the dependencies?

A naive implementation asks the LLM to declare its own critical state dependencies at commit time. This is self-referential: the same system that proposes the action also specifies what constraints must hold for that action to be valid. In safety-critical systems, this is architecturally unacceptable — the fox designing the henhouse. An LLM may underspecify, misspecify, or adversarially omit dependencies. If the hash is built on an incomplete dependency set, every downstream validation is corrupted from the origin. The silent disaster is not prevented — it is certified.

The solution: separate declaration from verification

Critical state dependencies must be determined by the runtime, not the model. We define a two-layer architecture:

Layer 1 — Static contracts per `transition_type` (kernel-enforced):

For transition types in high-risk namespaces, the TaskPack declares required state fields as a static contract. The kernel enforces this contract before commit, independently of anything the LLM declares:

```
TRANSITION_CONTRACTS = {
  "execute_transaction": requires([:cash_balance, :risk_limit, :
    position_size]),
  "action.request":      requires([:policy_decision, :tool_allowlist]),
  "plan.update":         requires([:goal_id, :progress_estimate])
}
```

If the required fields are absent from committed state at commit time, the commit fails deterministically. No LLM declaration is consulted. No model output is trusted. This is institutional-grade: it is mathematically enforceable and does not depend on model behavior.

Layer 2 — Intent context hash (contract as truth source, model as additive):

The kernel computes required dependencies deterministically from `transition_type` and the TaskPack — not from model output. The model may declare additional dependencies as advisory constraints, but cannot weaken or substitute the required set:

```
required = contract_required(transition_type)    # kernel-computed; model
    cannot alter
extra    = declared_deps - required              # optional advisory
    constraints
# The hash payload binds field values, not just field names
hash_payload = { field => quantized(state[field]) for field in (required
    UNION extra) }

intent_context_hash = H(
  intent.transition_type  ||
  intent.action           ||
  serialize(hash_payload)    # deterministic serialization of field\
    textrightarrow{}value pairs
)
```

The hash is generated *after* Layer 1 contract validation succeeds. If the model omits a contract-required field, the commit fails at Layer 1 before the hash exists. The guarantee: *the hash certifies that the kernel contract was satisfied — not that the model believed it was.*

This is real systems security: `required` = mandatory access control, `extra` = advisory constraints. An attacker cannot weaken the required set by omission, and cannot produce a valid hash before the contract passes.

Context validation before import (pseudocode):

```
function validate_import(intent, source_hash, target_state, contracts):
    # Step 1: resolve what this transition type requires (kernel-computed)
    required = contracts[intent.transition_type]

    # Step 2: verify required fields present in target state
    missing = required - fields_present(target_state)
    if missing is not empty:
        return Error(context_violation, missing_deps=missing,
                     risk=SILENT_DISASTER_PREVENTED)

    # Step 3: verify source hash commits to required fields' values (not
    just presence)
    # hash_covers_required checks that required field values from
    source_state_snapshot
    # are committed in the hash --- field names alone are not sufficient
    if not hash_covers_required(source_hash, required,
    source_state_snapshot):
        return Error(hash_contract_mismatch, reason=HASH_UNDERSPECIFIED)

    # Step 4: check advisory constraints; warn if missing, do not reject
    extra = hash_extra_deps(source_hash) - required
    advisory_missing = extra - fields_present(target_state)
    if advisory_missing is not empty:
        emit Warning(advisory_deps_missing=advisory_missing)

    return Ok(context_compatible)
```

If validation fails at Steps 1–3, the import is rejected with an explicit, auditable reason.

Advisory warnings at Step 4 are logged but do not block import.

The security property this achieves:

The hash is not a model's self-assessment of its own correctness. It is a runtime attestation that the contract for this transition type was satisfied at commit time in the source branch. Importing into a target branch re-validates that the same contract is satisfiable there.

Silent disasters are prevented by construction, not by model discipline. AVC operations become institutional-grade cognitive safety checks — verifiable against a declared contract, not against a model's declared intentions.

Minimally viable TaskPack contract: four required pillars

A contract is not a loose schema. It is an institutional gatekeeper — the static truth source the kernel consults before committing any transition to the trace. To be minimally viable under Dendrer, a TaskPack contract must provide four things. (A non-software domain example — insurance claims adjudication — appears in Appendix G.)

Pillar 1 — Namespace and transition registry. The contract must have a versioned unique identifier (e.g., `acme.logistics.v1`) and explicitly map every `transition_type` it governs. The version identifier is what prevents logic contamination during `fork` and `cherry-pick` — importing an intent from a v1 contract into a v2 context is detectable and rejectable.

Pillar 2 — Layer 1 static state dependencies (mandatory, kernel-enforced). For each governed transition type, the contract declares which fields must exist in `state_before`, their required types, quantization rules, and nullability. This is the nuclear shield: the kernel validates this independently of anything the LLM declared. Type enforcement ensures the pure reducer cannot crash during replay. Quantization rules ensure canonical serialization is consistent across environments and hardware. Nullability is a first-class field attribute: a nullable field must exist in state but may be null — this is a different contract from a non-nullable required field, and its absence or null value is itself meaningful committed state that violation predicates can reason over.

Pillar 3 — Layer 2 intent context hash requirements. The contract declares which state fields must be committed into the intent context hash (mandatory hash seeds), and which additional fields the LLM is permitted to declare as advisory dependencies. Mandatory seeds are the fields whose values at commit time are cryptographically bound to the intent — if `execute_transaction` is committed, `price_at_tick` must be in the hash, not merely present in state.

Pillar 4 — Progress and violation predicates. The contract declares at least one prefix-monotone violation predicate as a pure function over committed state, with explicit purity attestation (no I/O, no external calls). Without this, the contract is behaviorally blind — the kernel can validate inputs but cannot detect when the agent's behavior has drifted from policy.

Reference implementation (Elixir):

```
defmodule Dendrer.Contracts.FinancialV1 do
  @moduledoc """
  Minimally viable Dendrer contract for financial transaction agents.
  Version: acme.financial.v1
  Governs: execute_transaction, action.request
  """

  # Pillar 1: Transition registry
  def governs?(transition_type) do
    transition_type in ["execute_transaction", "action.request"]
  end

  # Pillar 2: Layer 1 static state dependencies (kernel-enforced)
  def required_fields("execute_transaction"), do: [:cash_balance, :
    risk_limit,
                                                   :position_size, :
    risk_threshold]
  def required_fields("action.request"),      do: [:policy_decision, :
    tool_allowlist]
  def required_fields(_),                     do: []

  # Pillar 2: Canonical serialization / quantization rules
  def precision_rule(:cash_balance),  do: {:round, 2}   # $0.01 precision
  def precision_rule(:risk_limit),    do: {:round, 4}   # 0.0001 precision
  def precision_rule(:risk_threshold), do: {:round, 4}  # aligned with
    risk_limit
  def precision_rule(:position_size), do: {:round, 0}   # integer shares
  def precision_rule(_),              do: :skip

  # Pillar 3: Mandatory hash seeds for execute_transaction
  def hash_seeds("execute_transaction"), do: [:risk_threshold, :
    cash_balance,
                                             :position_size]
  def hash_seeds(_),                     do: []

  # Pillar 3: Advisory dependency allowlist
  def advisory_allowlist("execute_transaction"), do: [:
    market_volatility_index,
                                                      :portfolio_exposure
    ]
  def advisory_allowlist(_),                     do: []

  # Pillar 4: Violation predicate (prefix-monotone, pure, no I/O)
  # purity: true --- safe for bisect
  def evaluate(state) do
    cond do
      state.risk_score > state.risk_threshold ->
        {:violation, {:risk_exceeded, state.risk_score, state.
    risk_threshold}}
      state.cash_balance < 0 ->
        {:violation, {:negative_balance, state.cash_balance}}
      true ->
        {:ok, 1.0 - state.risk_score / state.risk_threshold}
    end
  end
end
```

The version string `acme.financial.v1` is the contract's identity in the AVC system. When an engineer attempts to `cherry-pick` an intent committed under `acme.financial.v1` into a branch running under `acme.financial.v2`, the kernel detects the contract mismatch before evaluating any field-level checks. Contract versioning is the first line of defense against logic contamination across forks.

The honest caveat: contract completeness is a human responsibility.

The Layer 1 contract is only as strong as its schema. If a critical field — say, `is_simulation` — was never declared as required for `execute_transaction`, the contract passes and an unsafe import goes through. The system cannot protect against fields its authors did not anticipate. This is analogous to memory safety in systems languages: the contract reduces the attack surface from "anything the model might omit" to "anything the contract author omitted." That is a meaningful reduction, but not a guarantee of completeness.

In practice, contracts should be authored by domain experts with explicit review, versioned alongside the TaskPack, and expanded as new failure modes are discovered in production. The `is_simulation` class of error is not architectural — it is a process failure. The architecture makes such failures auditable and traceable after the fact; preventing them requires engineering discipline, not a smarter kernel.

Controlled contract evolution: `policy.contract_extension`.

A concern with static contracts is rigidity: as domains evolve, new fields become safety-relevant and existing contracts become insufficient. Allowing runtime mutation of contracts destroys the determinism and auditability the architecture provides.

The correct solution is to treat contract evolution as a first-class committed transition. Dendrer defines a reserved transition type:

```
transition_type: policy.contract_extension
payload:
  target_contract:   acme.financial.v1
  adds_required:     [counterparty_kyc_status]
  adds_hash_seeds:   [counterparty_kyc_status]
  adds_violations:
    - id: unverified_counterparty
      condition: "state.counterparty_kyc_status != 'verified'"
      pure: true
      monotone: true
  effective_from_tick: 47
  approved_by:       human_supervisor_id_or_policy_hash
```

Properties: - Must be approved (human gate or higher-level policy transition) - Recorded immutably in the trace, hash-chained like any other transition - Replayable: replay respects the effective_from_tick boundary - Does not modify any transition prior to effective_from_tick

This preserves the core guarantees: determinism (the contract in effect at any tick is derivable from the trace), auditability (every contract change is on ledger), and autonomy (the institution can evolve policy without rebuilding the runtime).

The critical distinction: the *model* cannot extend contracts — only the institution can, via an explicit approved transition. Contract evolution is on ledger, not off it.

4.5 The Remediation Spectrum

Not all failures identified by `bisect` are remediable via `fork`. This is a critical distinction that determines how practitioners should respond to a bisect result.

Bisect identifies the violation onset tick. It does not prescribe the fix.

The appropriate remediation depends on where the failure originates in the causal hierarchy:

Contingent failures — the committed state at tick k contained bad data: a wrong tool output, an unverified observation, a policy threshold misconfiguration. The reasoning was sound given the state it received; the state itself was the problem. These are recoverable via fork:

```
\textrightarrow{} DIAGNOSIS: contingent (state-level)
  fork --from tick 12 --tool-policy "require source verification"
  The agent's reasoning was locally valid. The input data was not.
```

Policy failures — the policy gate permitted a transition it should have denied, or denied one it should have permitted. The state was accurate; the enforcement was wrong. Also recoverable via fork with an updated policy contract:

```
\textrightarrow{} DIAGNOSIS: contingent (policy-level)
  fork --from tick 12 --policy "risk_threshold = 0.65"
  The agent reasoned correctly under the declared policy. The policy was
    miscalibrated.
```

Foundational failures — the failure originates upstream of the committed state: in the goal specification, the system prompt, the TaskPack definition, or the fundamental framing of the task. The trajectory is *logic-contaminated* from an early tick. No amount of forking from within the run will fix it, because the reasoning pattern that produced the failure is baked into the context the LLM received at boot.

```
\textrightarrow{} DIAGNOSIS: foundational (prompt-level)
  bisect identified symptom at tick 12; root cause is upstream of tick 0
  DO NOT fork --- re-run from scratch with revised goal specification
  This trace is a post-mortem artifact, not a fork seed.
```

The trace remains valuable in all three cases — but its role differs:

Failure type	Trace role	Recommended action
Contingent (state)	Fork seed	`fork --from tick k` with corrected tool/data
Contingent (policy)	Fork seed	`fork --from tick k` with updated policy contract
Foundational	Post-mortem artifact	Full re-run with revised specification

The distinction between contingent and foundational is a judgment call that bisect cannot make automatically. The causal chain output provides the evidence; the practitioner determines the diagnosis. A future extension could expose a `--diagnosis` flag that prompts structured classification at bisect completion, producing an auditable remediation decision alongside the origin tick.

On the "video recording" critique:

A common objection to trace-based debugging is that finding tick 12 does not give you a steering wheel — if you fork from tick 12 with a slightly different state, the LLM may take a completely different path anyway. This objection is valid for foundational failures. It is not valid for contingent failures, where the fork changes the input data or policy that caused the specific degradation, and the LLM's subsequent reasoning is expected to differ precisely because the bad input has been corrected.

Macroscopic interpretability does not promise that forking always produces a better run. It promises that you can identify exactly what was wrong, classify whether it is recoverable without a full re-run, and if so, from which tick to fork. That is the steering wheel — not over the LLM's cognition, but over the environment the LLM reasons within.

4.6 The Progress Contract Interface

The paper has treated progress quantification as an open problem. We now reframe it as a **design feature**: progress is not Dendrer's responsibility to define. It is an interface that institutions implement.

This is the same relationship a kernel has to applications. Linux does not define what a "correct" database query is. It provides syscalls. The database defines correctness. Dendrer does not define what a "successful" agent run means for your compliance policy. It provides the interface. Your institution defines success.

The architectural shift:

Instead of: > "Progress quantification is an open problem."

We declare: > "Progress is a pluggable contract. Dendrer provides the interface and > the evaluation infrastructure. The institution provides the predicate."

The Progress Contract (interface definition)

```
interface ProgressContract:
    # Must be a pure function over committed state
    # No I/O, no external calls, no randomness
    # Violation of purity makes bisect unsound

    evaluate(state) -> Ok(score: float[0.0..1.0])
                     | Violation(reason: term)
                     | Uncertain(score: float)

    name()    -> String
    version() -> String
```

Example implementations (pseudocode):

```
# Built-in: always available, grounded in kernel invariants
KernelAnomalyFree.evaluate(state):
    if state.kernel_anomalies is empty:
        return Ok(1.0)
    return Violation(kernel_anomaly=state.kernel_anomalies)
```

```
# Institution-defined: tax compliance
AcmeTaxComplianceV2.evaluate(state):
    score = compliance_engine.score(state.artifacts, ACME_TAX_RULES)
    if score >= 0.95:
        return Ok(score)
    return Violation(policy_breach=score, rules=ACME_TAX_RULES.violations(
    state))

# Institution-defined: financial risk
WallStreetRiskThreshold.evaluate(state):
    risk = state.risk_score
    if risk <= state.risk_threshold:
        return Ok(1.0 - risk)
    return Violation(risk_exceeded=(risk, state.risk_threshold))
```

See Appendix E for the Elixir/OTP implementation using `@behaviour`.

CLI consequences — bisect recovers full precision:

```
# Built-in predicate --- always available
dendrer bisect run-id --predicate kernel.anomaly_free

# Institution-defined --- bisect with your compliance lens
dendrer bisect run-id --predicate acme.tax_compliance_v2
dendrer bisect run-id --predicate ws.risk_threshold

# The output is now institution-specific:
\textrightarrow{} BISECT using predicate: acme.tax_compliance_v2 v2.1.3
\textrightarrow{} testing tick 32... {:ok, 0.98} OK
\textrightarrow{} testing tick 16... {:violation, {:policy_breach, 0.71,
    [...]}} FAIL
\textrightarrow{} testing tick 24... {:violation, {:policy_breach, 0.83,
    [...]}} FAIL
\textrightarrow{} testing tick 20... {:ok, 0.96} OK
\textrightarrow{} ORIGIN: tick 21 | transition_type=artifact.write
  violation: tax_policy_rule_47b not satisfied in generated artifact
\textrightarrow{} CAUSAL CHAIN: tick 21 \textrightarrow{} tick 24 \
    textrightarrow{} tick 32 (full violation propagation)
\textrightarrow{} SUGGESTION: fork --from tick 20 --note "verify rule 47b
    before artifact.write"
```

What this achieves:

Bisect is no longer searching for a universal definition of "wrong." It is searching for the exact tick where *your institution's policy* was first violated. The search is binary and deterministic because the predicate is a pure function. The result is legally and operationally meaningful because it speaks your institution's language.

This transforms institutional accountability from a philosophical posture into an executable workflow. Each institution uploads its own compliance definition. Dendrer runs it deterministically over the immutable trace. The audit is reproducible, the violation is pinpointed, and the fork suggestion gives engineers a concrete remediation path.

The kernel does not know what justice is. It knows how to enforce it once you define it.

4.7 Bisect Implementation

The following pseudocode makes explicit that `replay()` is **not LLM inference**. It is deterministic state reconstruction via the pure reducer. No model is called. No external I/O occurs. The same trace always produces the same state.

```
def replay(trace, reducer, s0):
    """
    Deterministic state reconstruction from committed transitions.
    No LLM inference. No external calls. Pure function.
    reducer: (state, transition) -> state
    """
    state = s0
    for transition in trace:
        state = reducer(state, transition)
    return state

def bisect(trace, predicate, reducer, s0):
    """
    Binary search for the first tick where predicate transitions to
   violation.
    Returns the violation onset tick k: smallest k s.t. predicate(s_k) =
   violation.

    Requirements (enforced at registration time):
    - predicate must be pure: state -> {ok | violation}  (no I/O, no
   randomness)
    - predicate must be prefix-monotone: if predicate(s_i)=violation then
      predicate(s_j)=violation for all j > i. Without this, binary search
   may
      return a tick where the predicate happens to be ok between two
   violation
      regions --- an incorrect result even under perfect replay.
   Predicates that
      are not prefix-monotone are rejected at bisect registration time.
    - reducer must be a pure function: (state, transition) -> state
    - state serialization must be canonical ($\varepsilon$-quantized for
   floats, not raw)

    If any requirement fails, bisect is unsound and registration is
   rejected.
    """
    low, high = 0, len(trace)
    while low < high:
        mid = (low + high) // 2
        # Deterministic reconstruction --- not inference, not I/O
        state_mid = replay(trace[:mid], reducer, s0)
        if predicate(state_mid) == "violation":
            high = mid    # violation present --- search left half
        else:
            low = mid + 1  # ok --- search right half
    return low  # violation onset tick: first k where predicate(s_k) =
   violation

def causal_chain(trace, origin_tick, predicate, reducer, s0):
    """
    Trace the propagation from origin_tick forward.
    Returns the sequence of transitions causally downstream of the origin.
    """
    chain = []
    state = replay(trace[:origin_tick], reducer, s0)
    for transition in trace[origin_tick:]:
        state = reducer(state, transition)
        if predicate(state) or transition.type in CAUSAL_TRANSITION_TYPES:
            chain.append((transition.tick, transition.type, state))
    return chain
```

The complexity follows directly: for a trace of N transitions, bisect requires at most $\lceil \log_2$ N$\rceil$ calls to `replay()`.

- **Without snapshots:** each `replay()` reconstructs from s_0 in O(N) steps → total O(N log N).
- **With snapshots every K ticks:** each `replay()` starts from the nearest snapshot within K ticks → O(K) steps per call → total O(K log N).

O(log N) evaluation would require O(1) reducer application per tick, which is not guaranteed for general reducers. The snapshot interval K is the practical tuning knob: smaller K reduces per-call cost at the expense of snapshot storage.

4.8 Evaluation Protocol and Benchmark Commitments

On synthetic evaluation

The coherence of bisect under controlled conditions follows directly from the invariants. If the reducer is pure, the predicate is derivable from committed state, and serialization is canonical, bisect operates in $\lceil \log_2$ N$\rceil$ steps with 100% accuracy by construction. Demonstrating this in a synthetic environment where all invariants hold by design is a proof of internal consistency — not an empirical result.

A reviewer who controls the environment can always make their algorithm work. We do not present synthetic fault injection as evidence of operational validity. We present it as a formal coherence check: if these numbers do not hold, the invariants are broken somewhere and that is what must be fixed.

The evaluation that matters

Operational validity requires demonstrating bisect on a trace produced by a real, noisy, LLM-driven agent — one where the invariants must be *enforced* rather than *assumed*. Specifically:

1. Extract a real failure trace from an existing agent framework (LangChain, AutoGPT, CrewAI, or equivalent) or instrument a GPT-4 agent under Dendrer's runtime to produce a native trace.
2. Define an institution-specific predicate (e.g., `policy_compliant`, `budget_constraint_satisfied`) as a pure function over committed state.
3. Inject a real behavioral failure — not synthetic — and verify that bisect localizes the origin transition correctly.
4. Measure: origin tick accuracy, steps required, causal chain completeness, and predicate false-positive rate under noise.

A single real trace with a real bug located correctly changes the perception of this work more than 10,000 synthetic runs. That is the empirical commitment.

Operational definition of "correctly identified" for the real-trace experiment: bisect is considered correct if it returns violation onset tick k such that (a) the registered predicate evaluates to `violation` at s_k and to `ok` at s_{k-1}, and (b) manual audit of the trace confirms that tick k is the first transition after which the system's behavior was meaningfully constrained by the violation. Condition (a) is mechanically verifiable; condition (b) requires human review and establishes ground truth against which the causal chain output is assessed.

Benchmark commitments (subsequent work)

Experiment	Metric	Target
Synthetic coherence (N=64, 100 runs)	Bisect steps $\leq \log_2$ N	Provable
Real LLM trace (1 trace, real failure)	Origin correctly identified	Verification
Contract enforcement (50 cherry-picks)	Zero silent imports on violation	Safety check
Predicate purity enforcement	Zero impure predicates accepted	Invariant check

Synthetic trace for formal reference

For reproducibility, we include one synthetic trace demonstrating the traversal pattern. This is a structural reference, not an empirical result:

```
run_id: logistics-planner-1  |  predicate: kernel.policy_compliant
trace length: 64  |  bisect steps: 6 (= $\lceil$log$_2$ 64$\rceil$)

tick | transition_type    | predicate | notes
-----|-------------------|-----------|------------------------------------------

 01  | observation.add   |  PASS OK   | weather_api \textrightarrow{} "
   severe storm" recorded
 08  | observation.add   |  PASS OK   | depends_on: [route.status, weather
   .condition]
 13  | observation.add   |  PASS OK   | alt_source introduced; confidence
   =0.31
 14  | policy.decision   |  FAIL FAIL   | risk_score=0.82 > threshold=0.75
     \textleftarrow{} ORIGIN
 17  | action.request    |  FAIL FAIL   | denied; workaround attempted
 20  | kernel.anomaly    |  FAIL FAIL   | invariant breach

bisect traversal: tick 32FAIL \textrightarrow{} 16FAIL \textrightarrow{} 8
   OK \textrightarrow{} 12OK \textrightarrow{} 14FAIL \textrightarrow{}
   13OK  \textrightarrow{}  ORIGIN: tick 14

causal chain:
  tick 13  observation.add  \textrightarrow{} unverified source introduced
  tick 14  policy.decision  \textrightarrow{} risk_score crossed threshold
    ; deny
  tick 17  action.request   \textrightarrow{} denied; workaround attempted
  tick 20  kernel.anomaly   \textrightarrow{} invariant breach

suggestion: fork --from tick 13 --policy "require source verification"
```

5. Accountability as the Basis for Trust

5.1 The Institutional Model

The dominant implicit model for AI trust is cognitive transparency: if we understand what is happening inside the system, we can trust it. This model motivates microscopic interpretability research and much of the mechanistic work in the field.

We propose an alternative: for systems whose behavior emerges from long trajectories rather than single inferences, institutional accountability provides a more operationally tractable trust model than cognitive transparency.

Large-scale human institutions — legal systems, financial markets, scientific publishing, democratic governance — do not earn trust through introspective transparency. The insurance claims contract in Appendix G shows this model applied concretely: an auditor verifies institutional policy by reading the contract, not the agent code. We cannot read the deliberations inside a judge's mind or a central bank's decision process. They earn trust through:

- Recorded decisions that can be examined
- Structured processes for challenge and appeal
- Consequences for behavior that departs from declared policy
- Accountable environments where nothing happens off-ledger

For agents whose failure modes are trajectory-level rather than inference-level — where a policy violation at tick 40 originates from a flawed observation at tick 12 — this model is not merely philosophically attractive. It is the appropriate unit of analysis, because the behavior requiring explanation spans ticks, not tokens.

5.2 The Trace as Accountability Infrastructure

Every decision the agent makes is on ledger. Every tool call has provenance. Every state change can be traced to the intent that proposed it and the policy that allowed it. The full trajectory of cognition from goal declaration to final artifact is available for examination, challenge, and audit.

This does not require understanding what happens inside the model during any given forward pass. It requires building the right environment around it — one where nothing happens without a record and every record is replayable.

On institutional failure and error correction:

Large institutions make large mistakes. This is expected — it is not an argument against institutional structures, it is an argument for better error correction infrastructure within them. The 2008 financial crisis, pharmaceutical disasters, large-scale engineering failures: in each case, the failure itself was not the terminal problem. The terminal problem was the inability to rapidly identify the origin, understand the causal chain, and prevent propagation.

The institutional accountability model is consistent with an error-correction view of knowledge growth: trust in a system increases not by eliminating the possibility of error, but by building environments where errors are rapidly identifiable and correctable. The trace is that environment. AVC is the error-correction machinery that operates over it.

What macroscopic interpretability offers is not the elimination of agent mistakes. It is something more significant: **agent behavior can be made more mechanically auditable than typical human institutional processes allow.** A human analyst who made a consequential decision in 2007 can reconstruct perhaps 60% of their reasoning from emails, meeting notes, and memory — months later, under adversarial conditions, with inevitable gaps and self-serving distortions. An agent under Dendrer can reconstruct everything it committed to the trace, mechanically, in $\lceil \log_2 N \rceil$ steps, down to the exact tick where the flawed observation entered the state. The trace is complete by construction — bounded only by what the runtime was configured to record, not by memory, time, or adversarial pressure.

This is a qualitative difference from even the most rigorous human documentation practices. In GxP-regulated environments — pharmaceutical manufacturing, clinical trials, laboratory research — human operators work under some of the most demanding documentation requirements in any industry, precisely because the stakes demand it. And yet even there, an irreducible gap persists: not because practitioners fail, but because human cognition and human action run on different substrate than human documentation. Reasoning happens first; the record is always a representation of it, constructed afterward, inevitably incomplete in ways that are difficult to even detect. The best GxP systems minimize this gap through training, process design, and audit culture. They cannot eliminate it.

Agents have no gap between cognition and record. Every state transition either passed through the kernel — in which case it is in the trace, immutably, with full causal context — or it did not happen at all. For the first time, "as executed" and "as documented" are the same artifact by construction. Not by discipline. Not by regulation. By architecture.

This is not a claim that agents are more reliable than humans. It is a claim that the *accountability infrastructure* around agents can be qualitatively stronger than anything available to human institutions — and that this infrastructure is buildable now, with the mechanisms described in this paper.

5.3 Complementarity with Microscopic Approaches

Microscopic interpretability asks: *what is the model thinking?* Macroscopic interpretability asks: *what did the system do, and can we reproduce it exactly?*

These are orthogonal. A complete interpretability program for autonomous agents requires both levels:

- Microscopic analysis to understand model representations and failure modes at the level of individual inference steps
- Macroscopic analysis to understand behavioral trajectories, causal fault localization, and operational accountability across extended runs

We do not compete with microscopic approaches. We address a different layer — one that is currently underserved and directly actionable in production systems.

---

6. Incremental Adoption Model

The full Dendrer runtime — pure reducer, prefix-monotone predicates, coordinator-based commit ordering, effectful tool idempotency — is the idealized system. Requiring teams to adopt it entirely before receiving value is not a deployment strategy. It is a barrier.

Why not just logs?

Log-based debugging and trace-based debugging are not the same thing, even when both store the same raw data:

	Logs	Dendrer Trace
Ordering	Observational, may be unordered	Committed, coordinator-assigned tick order
Replay	Not guaranteed — logs describe what happened, not how to reconstruct it	Verified — replay reconstructs exact committed state
Bisect	Not sound — no stable predicate over log entries	Sound — binary search over committed state with pure predicates
Causal navigation	Manual — engineer reads logs and hypothesizes	Mechanical — bisect + causal chain extraction
Diff	String comparison	Semantic diff over structured state fields

A log answers: *what did the system emit?* A trace answers: *what sequence of committed state transitions produced this outcome, and from which tick can I fork to prevent it?*

The adoption model is therefore staged. Each stage delivers standalone value and builds on the trace substrate required by subsequent stages. Migration is additive, never destructive: a team at Stage 1 does not need to rebuild anything to reach Stage 2.

Stage 0 — Observability Only

Teams keep their existing runtime (LangChain, CrewAI, or any other framework). Dendrer provides structured trace capture — LLM outputs, tool calls, policy decisions — written to immutable append-only storage. No replay guarantee yet. No reducer purity required. No prefix-monotonicity enforcement.

This is better structured observability, organized around transition types rather than raw logs. The trace is already in the format that Stages 1–3 will operate over.

Value delivered: conceptual fork view, diff between runs, visual trace browser. *What is not yet available:* deterministic replay, sound bisect, contract enforcement. *Who this serves:* any team using an existing agent framework who wants structured postmortem capability without rebuilding their runtime.

Stage 0 adapter (LangChain example):

```python
from dendrer import TraceWriter

writer = TraceWriter(run_id="my-agent-run", storage="append-only")

# Wrap your existing LangChain callback handler
class DendrerAdapter(BaseCallbackHandler):

    def on_llm_end(self, response, **kwargs):
        writer.emit({
            "transition_type": "observation.add",
            "tick": writer.next_tick(),
            "payload": response.generations[0][0].text,
            "confidence": kwargs.get("confidence"),
        })

    def on_tool_end(self, output, **kwargs):
        writer.emit({
            "transition_type": "action.result",
            "tick": writer.next_tick(),
            "tool": kwargs.get("tool_name"),
            "output": output,   # recorded for replay in Stage 1
        })
```

No reducer. No purity requirements. The trace is written in Dendrer's transition schema, ready to be upgraded to Stage 1 replay without restructuring.

Stage 1 — Replay with Divergence Diagnostics

Teams opt into recording tool outputs in the trace and freezing external calls during replay. A deterministic state reconstruction layer is introduced. Reducer purity is not yet strictly enforced — impurity is detected and flagged rather than rejected, giving teams visibility into what would need to change to reach Stage 2.

Value delivered: reliable bisect over hard predicates, reproducible postmortems, identification of which reducer operations are non-pure. *What is not yet available:* contract enforcement, context-safe cherry-pick. *Who this serves:* teams who have experienced a production failure they could not fully diagnose from logs alone. One real bisect on a real failure trace converts this audience.

Terminology note: this stage enables replay with divergence detection — impure reducer operations are flagged, not rejected. True deterministic replay (where replay is guaranteed to reconstruct committed state) requires reducer purity, which is enforced at Stage 3.

Stage 2 — Contract Enforcement

Teams begin defining static transition contracts for high-risk operations. `required_fields`, quantization rules, and intent context hash seeds are declared for sensitive transition types. Context validation on cherry-pick is enabled. Full reducer purity is not required — mixed environments are supported. Contract enforcement is opt-in per transition type, not system-wide.

Value delivered: production safety for sensitive operations, silent disaster prevention on AVC imports, auditable dependency declarations. *What is not yet available:* coordinator-based commit ordering, effectful tool idempotency classes. *Who this serves:* enterprise teams with compliance requirements (GxP, financial regulation, legal accountability) who need to demonstrate that specific high-risk transitions were validated before execution.

Stage 3 — Full Trace-First Runtime

The complete Dendrer runtime. Pure reducer enforced at registration. Prefix-monotone predicates required for bisect registration. Coordinator serializes all commits. Effectful tools declare their idempotency class. The TaskPack is the institutional gatekeeper for all transitions.

By this stage, teams are already fully dependent on the trace infrastructure built across Stages 0–2. The migration from Stage 2 to Stage 3 is a tightening of enforcement, not a re-architecture.

Value delivered: full soundness guarantees for bisect, complete AVC primitive set, institutional-grade accountability across all transitions. *Who this serves:* teams deploying agents in regulated environments or high-stakes autonomous workflows where the full formal guarantees are required.

The strategic consequence of staged adoption:

Stage 0 competes with LangSmith and Langfuse on their own ground — but with trace structure designed to migrate upward. Every team that adopts Stage 0 is building on the substrate that Stages 1–3 require. There is no throwaway work. The path is one-directional and the switching cost increases at each stage — not because the system locks teams in, but because the trace history accumulated at each stage is genuinely valuable and not portable to systems that lack the structural guarantees Dendrer provides.

---

7. What Good Looks Like

```
BOOT: goal="Plan 3-day Basel\textrightarrow{}Ticino trip under CHF 600"
SPAWN: 6 agents (researcher, planner, budgeter, skeptic, verifier,
   coordinator)

TICK 01: observation.add       | confidence=0.91 | progress=+0.08
TICK 02: plan.update           | confidence=0.84 | conflict detected \
   textrightarrow{} resolved by policy
TICK 03: action.request        | confidence=0.61 | uncertainty high \
   textrightarrow{} exploration spawned
TICK 04: action.result         | tool_failure    \textrightarrow{} agent
   restarted by supervisor
TICK 05: kernel.anomaly        | hallucination detected \textrightarrow{}
   process killed \textrightarrow{} restored from tick 03
TICK 06: goal.complete         | confidence=0.94 | progress=+0.31 \
   textrightarrow{} return condition met

DONE: artifact=plan.md | replay_trace=47_transitions | replay_verified=
   true
```

This is not a chatbot output. It is a runtime receipt.

Every line is a committed transition. The full trace is replayable. Any tick is forkable. Any anomaly is bisectable. Nothing happened silently.

---

8. Relationship to Existing Work

8.1 Event Sourcing and CQRS

The foundational intellectual debt of this paper is to event sourcing (Fowler, 2005) and Command Query Responsibility Segregation (Young, 2010). Event sourcing grounds system state in an append-only log of domain events, enabling state reconstruction by replaying the event stream from an initial snapshot. CQRS separates write models (commands that mutate state) from read models (queries over projected state), a separation structurally analogous to Dendrer's intent/commit boundary.

Macroscopic interpretability builds on these foundations and extends them in three specific ways that distributed systems event sourcing does not address:

1. Predicate-derivability as a first-class requirement. Event sourcing guarantees reconstructability of state. It makes no claim about whether behavioral predicates can be evaluated over committed events without external context. The distinction between `kernel.anomaly` (always derivable) and `predicate.violation` (derivable only if pure) has no direct analogue in event sourcing literature — it arises specifically from the need to support sound binary search over agent trajectories.

2. Prefix-monotonicity as a bisect precondition. Git bisect (Torvalds, 2005) is the closest operational precedent: binary search over a commit history with a good/bad predicate. Git bisect assumes a stable partition. This paper formalizes that assumption as prefix-monotonicity, derives the conditions under which it holds for agent predicates, and enforces it at predicate registration time. Neither event sourcing nor CQRS literature formalizes this requirement because they do not support binary search over state trajectories as a first-class operation.

3. Intent context hash as a context-safe import primitive. Cherry-pick in code version control operates over inert artifacts. The intent context hash introduces a cryptographic commitment over the state observations that justified an intent, enabling context-safety checks that have no analogue in distributed event logs or code VCS systems.

8.2 Causal Inference and Structural Causal Models

Beckers and Halpern (2019) formalize causal abstraction — the relationship between high-level and low-level causal models — and derive conditions under which causal claims at one level of abstraction are consistent with a more detailed model below. This paper operates at a single level of abstraction (committed state transitions) but is in close dialogue with that framework: the causal chain output of bisect is a heuristic approximation of the minimal intervention set that Halpern and Pearl's actual causation literature would formalize.

The key difference is operational: structural causal models require specifying counterfactual worlds, which is tractable for toy domains but expensive for production agent systems with complex state. Macroscopic interpretability constrains itself to claims derivable from the committed trace — no counterfactual specification required. This is a deliberate limitation in exchange for tractability. The causal contributor set (Section 4.5) is explicitly labeled heuristic rather than a formal actual-cause claim in the sense of Halpern (2016).

8.3 Distributed Systems: Log-as-Truth and Immutable Databases

Kreps (2013), "The Log: What every software engineer should know about real-time data's unifying abstraction," establishes the theoretical foundation: an append-only, ordered log is the universal substrate for distributed system state. Kafka's implementation of this principle — log compaction, offset-addressable replay, and consumer fan-out — is the production precedent that demonstrates append-only logs are operationally viable at scale. Dendrer's immutable trace is a direct descendant of this lineage, adapted for a different consumer.

Datomic (VanderHart & Fogus, 2012) extends this principle to queryable databases: the database is an immutable, append-only log of datoms; any past state is reconstructible by replaying from the beginning or from a snapshot; time is a first-class dimension of the data model. AVC's fork, diff, and bisect are natural operations on a Datomic-style substrate — fork creates a branch point in the datom log, diff compares projections at two time coordinates, bisect binary-searches the time axis. Dendrer applies these patterns to agent behavior rather than application data.

The distinction between both systems and Dendrer is purpose: distributed logs and immutable databases are designed for state reconstructability and queryability. The trace additionally requires predicate-derivability — the ability to evaluate pure behavioral predicates at arbitrary historical states — and prefix-monotone enforcement for binary search soundness. These requirements are invisible in the database and distributed systems contexts because their consumers do not run binary search over the log with pure predicates.

8.4 Agent Frameworks and Observability

Execution frameworks. LangGraph, AutoGen, DSPy, and CrewAI provide structured multi-agent orchestration with varying degrees of trajectory visibility. LangGraph's checkpoint mechanism is the closest structural precedent — it allows state snapshots and conditional re-execution. The gap is formal: LangGraph checkpoints are engineering conveniences, not a formal model with soundness guarantees. There is no equivalent of the pure reducer invariant, no bisect primitive, and no predicate registration with monotonicity enforcement. Dendrer is positioned as an interpretability and accountability layer above these runtimes, not a replacement. The Stage 0 adapter (Section 6) demonstrates how existing LangChain-based agents can emit Dendrer-structured traces without rebuilding their runtime.

Observability tools. LangSmith and Langfuse provide observability over agent runs — logging LLM calls, tool outputs, token usage, and session metadata. These tools answer "what did the model receive and return?" Dendrer answers a different question: "what sequence of committed state transitions caused this outcome, and from which tick can we fork to prevent recurrence?"

System	Deterministic Replay	Pure State Reducer	Causal Bisect	Primary Abstraction
LangSmith	Partial	No	No	Observability / tracing
Langfuse	Partial	No	No	Observability / cost
Arize	No	No	No	Drift monitoring
LangGraph	Partial (checkpoints)	No	No	Workflow orchestration

System	Deterministic Replay	Pure State Reducer	Causal Bisect	Primary Abstraction
AutoGen	No	No	No	Multi-agent coordination
Dendrer	**Yes (invariant-enforced)**	**Yes**	**Yes**	**Procedural causality**

These are complementary layers. A team using LangGraph for execution and LangSmith for observability can add Dendrer for causal localization and AVC without replacing either. The substrate is additive.

8.5 Mechanistic Interpretability and Taxonomy

Bereska and Lukacs (2024), "Mechanistic Interpretability for AI Safety — A Review," provide a taxonomy (Table 1) structured along four axes: causal direction (observational vs. interventional), locality (local vs. global), phase (post-hoc vs. intrinsic), and comprehensiveness (partial vs. holistic). Within that taxonomy, macroscopic interpretability occupies a specific and previously underrepresented cell: **interventional, global, post-hoc, holistic**.

Microscopic methods (circuit analysis, activation patching, probing) are interventional but local and partial — they target specific components of a single forward pass. Macroscopic interpretability is interventional at the level of committed decision sequences (bisect is an intervention on the trace), global (it spans the full trajectory, not a single layer), post-hoc (analysis after execution, over the committed record), and holistic (it addresses the behavioral outcome of the full run, not a single feature or token).

The Bereska taxonomy makes explicit that neither level subsumes the other — they are different cells in the same classification space, operating at different granularities of the same causal question.

Olah et al. and Anthropic's mechanistic interpretability work address orthogonal causal questions. Microscopic interpretability asks: what internal computation produced this model output? Macroscopic interpretability asks: what sequence of committed decisions produced this behavioral outcome? Trace-level analysis surfaces behavioral phenomena — policy violations, hallucination propagation, low-confidence observation chains — that circuit-level work should eventually explain at the representational level.

MAIA (Hernandez et al., MIT, 2023), an automated interpretability agent for multimodal systems, demonstrates that interpretability can itself be automated through agentic processes. This is complementary to the macroscopic approach: MAIA operates over model internals; Dendrer operates over committed behavioral trajectories. The two could be composed — MAIA explaining what happened inside the model at tick k, Dendrer explaining why the system committed to tick k given the state at tick k-1.

Figure 1 (conceptual): Representational vs. Procedural Causality

```
REPRESENTATIONAL CAUSALITY (microscopic)
-----------------------------------------
  Input tokens
       |
       v
  [Attention layers]  ---- circuit analysis, activation patching
```

```
       |
       v
  [MLP / residual]    ---- feature identification, probing
       |
       v
  Output token        <---- "why did the model output X?"

  Unit of analysis: forward pass
  Time horizon: single inference
  Method: weight inspection, activation patching


PROCEDURAL CAUSALITY (macroscopic)
------------------------------------
  Goal declaration (tick 0)
       |
       v  committed transition
  observation.add (tick 12)  <---- low-confidence source
       |                          [bisect locates this]
       v  committed transition
  plan.update     (tick 15)
       |
       v  committed transition
  policy.decision (tick 23)  <---- risk_score crossed threshold
       |
       v  committed transition
  kernel.anomaly  (tick 40)  <---- "why did the system violate policy?"

  Unit of analysis: committed state transition
  Time horizon: full trajectory (N ticks)
  Method: deterministic replay + binary search over pure predicates
```

The two approaches are not competing — they address different causal questions at different timescales. Mechanistic work explains what happened inside the model at a single inference step. Macroscopic work explains which sequence of external commitments produced the observable behavioral outcome. A complete interpretability program for autonomous agents requires both layers: microscopic analysis to understand model failure modes; macroscopic analysis to localize and recover from them in production.

8.6 Agentic and Trajectory-Level Interpretability

Two recent works directly address the gap this paper responds to, from different angles.

Beyond Static Mechanistic Interpretability: Agentic Long-Horizon Tasks (Martian, 2026) argues that forward-pass analysis is insufficient for agents on long-horizon tasks such as SWEBench, where failure modes involve hypothesis drift across many steps rather than a single bad inference. The post specifically frames the open problem as *temporal credit assignment and intervention* — identifying which earlier decision caused a later failure, and intervening at that point rather than at the symptom.

This paper is a direct response to that framing. The distinction is methodological: the Martian post identifies the *need* for trajectory-level interpretability as a research direction; this paper proposes a formal substrate that makes temporal credit assignment tractable — bisect over an immutable trace with a pure predicate performs exactly the “temporal credit assignment” the post calls for, and fork provides the “intervention” primitive. The contribution is not advocacy for the approach but a formal model that operationalizes it with soundness guarantees.

Interpretable Traces, Unexpected Outcomes (Mallen et al., TMLR under review, 2025) studies chain-of-thought trace interpretability in LLMs via finetuning and human evaluation. The findings are quantitatively precise: verbose DeepSeek R1 traces improve task performance via supervised finetuning but score 3.39/5 on human interpretability (Likert) with cognitive load reaching 4.59/5 — worse than shorter traces. More striking: correct traces do not correlate with human accuracy in identifying correct reasoning, with only a 28% true-positive rate (Tables 2 and 4).

This result is directly and quantitatively relevant to Dendrer's design. A system that required human-legible traces would inherit exactly the failure mode Mallen et al. document: longer, more complete traces are *less* interpretable to humans, not more. Macroscopic interpretability sidesteps this entirely: the bisect operation does not require a human to read the trace — it requires the predicate to be evaluable over committed state by a machine. The 28% true-positive finding is evidence that human-legibility and causal navigability are not merely *different* properties — they are in active tension. Optimizing for one degrades the other.

8.7 Formal Verification

Formal verification proves properties before execution; macroscopic interpretability examines properties after execution and enables controlled re-execution via fork. Complementary: formal methods could prove properties of the reducer and anomaly functions that macroscopic interpretability then depends on. A formally verified reducer is a pure reducer by construction — satisfying the core invariant without runtime enforcement. This is an attractive direction for high-assurance deployments.

8.8 RLHF / RLAIF and Process Reward Models

RLHF shapes global behavioral policy; AVC performs surgical trajectory intervention. AVC identifies the exact transition where behavior degraded and forks from that point with an altered policy. The two approaches operate at different granularities and time horizons and are composable: RLHF sets the prior; AVC corrects production deviations from it.

Process Reward Models (Lightman et al., 2023) assign rewards to intermediate reasoning steps — conceptually aligned with the per-tick signal structure of the trace. A PRM could serve as the optional progress scoring function in Section 9.5, with the constraint that it must satisfy purity requirements to be usable in bisect. Integration is left for future work.

---

9. Constraints and Open Problems

9.1 Threat Model

The security properties described in this paper hold under an explicit threat model. We name the threats and bound the scope of our claims.

In scope (addressed by the architecture):

- *Underspecified intent dependencies:* mitigated by Layer 1 static contracts (Section 4.4). The kernel enforces required fields independently of model output.
- *Silent context violations in AVC imports:* mitigated by intent context hash and two-stage validation. Imports are rejected with explicit reasons, not silently accepted.
- *External world mutation between run and replay:* mitigated by replay isolation (Section 3.5). Recorded tool outputs are injected; live IO is blocked in replay mode.
- *Non-deterministic predicate evaluation:* mitigated by purity enforcement at registration time. Impure predicates are rejected before they can corrupt bisect.

Out of scope (future work):

- *Adversarial model intent:* a model that systematically misrepresents its reasoning to evade contract checks. Contracts are necessary but not sufficient against a sufficiently adversarial model. Static analysis of intent declarations is future work.
- *TaskPack contract incompleteness:* contracts are authored by humans and cannot anticipate every critical field. A field omitted from a contract (e.g., `is_simulation` for `execute_transaction`) will not be validated at cherry-pick time. Contract completeness is an engineering process responsibility — the architecture makes omissions auditable and traceable after the fact but cannot prevent them. See Section 4.4 for the full treatment.
- *Trace tampering and key management:* the immutability guarantee assumes tamper-evident storage with appropriate key management. Cryptographic trace signing is an engineering requirement not specified here.
- *Tool output fabrication:* a compromised tool that returns false outputs at commit time poisons the trace at the source. Trust in tool outputs is assumed; attestation mechanisms are future work.
- *Epistemic truth of committed observations:* macroscopic interpretability guarantees procedural accountability over committed state transitions, not epistemic truth of their content. An agent that commits a plausible-but-false observation produces a valid, replayable transition. Truth validation and source verification are orthogonal concerns implementable as policy layers over the trace substrate but are out of scope here.
- *PII and privacy:* the trace records all committed state, which may include personal data. Pre-commit redaction hooks and field-level encryption are required for production deployment but are outside the scope of this paper.

This threat model is not exhaustive. It establishes what the architecture currently guarantees and where honest engineering work remains.

The threat model claims are currently qualitative. Quantification experiments (cherry-pick failure rates with/without Layer 1 contracts; contract incompleteness audits) are deferred to the first production deployment. See Section 11.2.

9.2 Determinism is Conditional

The replayability guarantees described here hold under controlled execution environments. Real production systems introduce sources of non-determinism:

- **Nondeterministic LLM APIs:** temperature > 0 produces different outputs on rerun. Mitigation: record full LLM outputs in the trace; replay uses recorded outputs, not re-inference.

- **Time:** timestamps in state break canonical serialization. Mitigation: exclude wall-clock time from diffable state; use tick as the temporal index.
- **External API calls:** tool outputs may change between original run and replay. Mitigation: record tool outputs in the trace; replay uses recorded outputs.
- **Concurrency:** parallel agents introduce ordering non-determinism. Mitigation: coordinator serializes commits; trace ordering is coordinator-assigned.

Under these mitigations, determinism is achievable for replay purposes. The system replays what was recorded, not what would happen if re-run live. This is the correct semantics for audit and debugging.

Snapshotable execution environment:

The snapshot and restore mechanism requires that agent state be fully serializable at any tick boundary. This assumes a controlled execution environment where the process state, memory, and tool context can be captured and restored deterministically. In practice this means agents must not depend on ambient process state that is not explicitly tracked in the committed state — open file descriptors, mutable global variables, or background threads whose state is not snapshotted. Containerized environments (e.g., Firecracker microVMs, WASM runtimes) provide the strongest guarantees here. The paper assumes a snapshotable environment; production deployments must verify this assumption holds for their infrastructure.

9.3 Anomaly Derivability

Any anomaly detector that makes external calls, uses randomness, or depends on non-persisted context violates the purity invariant and makes bisect unreliable. This must be enforced at registration time, not detected after the fact.

9.4 Canonical State Serialization

Diff reliability depends entirely on canonical state serialization. Floating point comparison without tolerance thresholds, unordered map iteration, and non-deterministic UUID generation in state fields all produce misleading diffs. This is a solvable engineering problem but requires explicit discipline and schema enforcement.

Threshold precision alignment:

A subtle but consequential edge case arises when predicate thresholds and quantization precision are not co-declared. If a predicate evaluates `risk_score > 0.850` and the schema quantizes `risk_score` to 3 decimal places, values in the range `[0.8495, 0.8505)` are ambiguous: accumulated floating-point drift across a long trace may push a value across the quantization boundary, changing the predicate result without any semantically meaningful change in the underlying signal.

The mitigation is schema discipline, not architectural change: threshold values in predicates must be declared at the same precision as their quantization rule, and predicate authors are advised to declare guard bands around sensitive thresholds (`risk_score > 0.86` rather than `risk_score > 0.850` when the meaningful boundary is approximately 0.85). The system cannot enforce this automatically — it is a contract authoring discipline. Values that legitimately sit near a threshold boundary represent genuine ambiguity in the domain, not a failure of the architecture.

9.5 Progress Quantification (Optional Scoring Layer)

Progress quantification — measuring how close an agent is to its goal at each tick — is explicitly **not a core bisect invariant**. Bisect is grounded in hard, derivable predicates: invariant violations, policy flips, state predicate failures. These are binary and deterministic. Progress is neither.

Progress is an optional scoring layer, useful for monitoring, dashboards, and regression-mode bisect (`--regression`), but not load-bearing for the soundness of AVC primitives. Treat it as a heuristic, not an invariant.

That said, a practical partial solution exists for bounded, declarative goals:

A three-signal composite, each derivable from committed state:

Signal 1 — Structural completion. At BOOT, the goal decomposes into a declared sub-goal tree via the TaskPack. Progress is the fraction of sub-goals with at least one committed `artifact.write` or `goal.complete` transition. Coarse but deterministic.

Signal 2 — Constraint satisfaction. TaskPack declares explicit constraints (budget, time, scope). A penalty term for violated constraints, derivable from committed state without external calls.

Signal 3 — Skeptic survival rate. Fraction of recent transitions that survived skeptic challenge without modification. Proxy for internal reasoning coherence.

```
progress(tick) = w$_1$ $\cdot$ structural_completion
               + w$_2$ $\cdot$ constraint_satisfaction
               - w$_3$ $\cdot$ (1 - skeptic_survival_rate)
```

Weights are declared in the TaskPack. Progress is domain-specific by design.

Critical boundary: bisect must never depend on this signal for soundness. If progress is unavailable or unreliable, bisect falls back to hard predicates and remains fully sound. Progress degradation is one possible bisect signal among many — not the primary one. See Appendix D for progress-based bisect extensions.

Open empirical question: correlation with real task success.

The composite progress signal (structural completion + constraint satisfaction + skeptic survival rate) is theoretically motivated but not empirically validated. The open question is: how well does this composite correlate with actual goal completion in LLM-driven agent traces?

Empirical validation is deferred. The progress signal should be treated as a domain-calibrated heuristic until correlation against real task success is measured. See Section 11.4 for the experimental design.

9.6 Tool Commutativity and Idempotency Requirements

True semantic merge of agent states where branches have taken real-world actions with side effects remains an open problem. Current artifact-seeded re-run semantics are honest about this limitation.

For rebase to be viable, tools must satisfy explicit idempotency requirements. We define three idempotency classes that every tool in a TaskPack must declare:

Class	Definition	AVC behavior
`read`	No side effects; always safe to re-execute	Re-execute freely during rebase
`idempotent_write`	Same input always produces same effect	Re-execute safely during rebase
`effectful`	Side effects that may not be repeatable	Use recorded output; never re-execute

Tool commutativity — whether two tool calls can be reordered without changing the outcome — is a separate concern from idempotency. Rebase preserves causal ordering of intents precisely to avoid requiring commutativity. If two intents in Branch B depend on each other's outputs, rebase replays them in tick order and the dependency is naturally preserved.

Architectural rule: Tools declared `effectful` must have their outputs recorded in the trace at commit time. The trace entry for an `effectful` tool call is the canonical record of what happened. Rebase and replay use that record; they never re-execute the tool. This is the only way to make AVC safe for agents that send emails, charge payments, or mutate external databases.

Full merge semantics for systems with non-commutative, non-idempotent tool calls requires either reversible execution or a formal theory of state reconciliation. This is deferred to future work.

9.7 Scale

Traces grow with agent runtime. Long-running agents with many tools produce large traces. Efficient storage, indexing, and bisect over large traces requires engineering attention. Snapshot-based compression (compact snapshot every N ticks, store only delta after) is the standard approach; optimal N depends on workload.

---

10. Conclusion

We have argued that macroscopic interpretability — treating agent cognition as a versioned, replayable, navigable artifact — is a tractable and practically useful complement to microscopic interpretability for autonomous agent systems.

The core formalization: **a system is macroscopically interpretable if and only if, given its trace and initial state, a deterministic reducer can reconstruct any intermediate state by replaying transitions.**

From this foundation, Agent Version Control (AVC) becomes possible: fork, diff, bisect, and cherry-pick over reasoning trajectories — enabling the same fearless experimentation that version control enabled for code.

Three mechanisms make this concrete in practice:

Replay isolation encapsulates external nondeterminism. The external world changes; the audit does not collapse. The GenServer intercepts tool calls in replay mode and injects historical payloads. Chaos is contained, not treated as corruption.

Intent context hash makes AVC operations cognitive safety checks, not file copy operations. An intent cannot be imported into a branch that lacks the state observations that justified it. Silent disasters are prevented by construction.

Progress contract interface transforms an open problem into a design feature. Dendrer does not define what success means for your institution — it provides the interface. Each institution uploads its compliance definition. Bisect runs through your policy lens, producing legally and operationally meaningful results.

The philosophical reframing: trust in autonomous systems should be modeled on institutional accountability rather than cognitive transparency. Agents should be treated as institutional actors operating in accountable environments — where every decision is on ledger, every action has provenance, and every trajectory is replayable.

Microscopic interpretability asks what the model thinks. Macroscopic interpretability asks what the system did, how it propagated, and from which exact tick you can fork to prevent recurrence.

For production systems, both matter. For operational trust today, the second is more tractable and more immediately actionable.

The trace is the map. AVC is how you navigate it. The progress contract is how your institution defines the territory.

---

11. Open Problems

The following problems are deferred from the main text, consolidated here for reviewers and future contributors.

11.1 AVC Extensions: Rebase and Merge

Rebase — replay Branch B's committed intents onto Branch A's state in tick order via the pure reducer, producing a linear trace. Requires tool idempotency declarations. Non-trivial to implement safely for effectful tools.

Merge — seed a new run with selected artifacts from parallel branches. The Git analogy breaks: tool calls do not commute, side effects may have already occurred, state may be globally incompatible. Honest implementation is a new run seeded from artifacts, not history reconstruction. Open research problem.

Both become meaningful after fork, diff, and bisect are proven in production.

11.2 Contract Completeness via Static Analysis

Layer 1 contracts cannot anticipate every critical field. Can static analysis over a corpus of agent traces automatically surface fields that should be declared as required but are not? This would close the contract incompleteness gap without requiring schema authors to anticipate every failure mode in advance.

11.3 Formal Proof of Bisect Soundness

Section 3.3.1 provides counterexample constructions. A complete formal proof in a proof assistant (Lean, Coq) would provide the strongest foundation for high-assurance deployment.

11.4 Progress Signal Empirical Validation

The composite progress signal (structural completion + constraint satisfaction + skeptic survival rate) requires validation against real task success. See Section 9.5 for the experimental design.

11.5 Adversarial Intent Declaration

A model that systematically misrepresents its reasoning to evade contract checks is not addressed. Static analysis of intent declarations for strategic underspecification is future work.

11.6 Privacy-Preserving Traces

The trace records all committed state, which may include PII. Field-level encryption, pre-commit redaction hooks, and selective disclosure mechanisms are required for regulated deployments but not specified here.

Appendix A: Transition Ontology (V0)

The ontology has three categories with different authorship and gating rules:

```
-- AGENT TRANSITIONS (authored by the LLM, gated by kernel contract checks
   ) --

observation.add       --- agent records a new observation
plan.update           --- agent updates its current plan
action.request        --- agent proposes a tool call
action.result         --- tool call result recorded
memory.write          --- agent writes to long-term memory
policy.decision       --- policy gate allow/deny/escalate
anomaly.flag          --- agent flags uncertainty or anomaly
goal.complete         --- goal success criteria met

-- SYSTEM TRANSITIONS (authored by the runtime, never by the LLM) --

kernel.anomaly        --- runtime-detected invariant violation (always
   decidable)
predicate.violation   --- institution-defined policy breach (decidable if
   predicate is pure)
run.fork              --- new run spawned from existing trace at tick N
run.cherrypick        --- intent imported from another run, context-
    validated
bisect.result         --- bisect completed, origin tick identified
diff.summary          --- diff completed between two runs
causal.chain          --- propagation trace from origin tick to failure
    surface

-- GOVERNANCE TRANSITIONS (authored by the institution, requires explicit
    approval) --

policy.contract_extension
                      --- evolves a TaskPack contract on-ledger; approved
    by human
                        or higher-level policy; effective from declared
    tick;
                        does not retroactively modify prior transitions;
                        recorded and hash-chained like all other
    transitions
                        (see Section 4.4 for full schema and properties)
```

Note: `run.merge` is omitted from V0. Merge is deferred — see Section 11.1.

Authorship rule: the LLM may only author Agent Transitions. Attempting to emit a System or Governance transition type is a `kernel.anomaly`. This boundary is enforced by the kernel, not by the model.

Appendix B: Canonical Event Schema

```
{
  "tick": 42,
  "ts": "2026-02-23T17:21:05Z",
  "agent_id": "agent-planner-1",
  "goal_id": "goal-abc123",
  "run_id": "trip-planner-a",
  "transition_type": "action.request",
  "state_before": {"progress": 0.31},
  "state_after": {"progress": 0.34},
  "intent": {
    "description": "call search tool for pricing data",
    "confidence": 0.64,
    "uncertainty_notes": "source reliability unknown"
  },
  "action": {
    "tool": "http_get",
    "params": {"url": "https://sbb.ch/..."}
  },
  "result": {
    "status": "ok",
    "output": {"price_chf": 48}
  },
  "policy": {
    "decision": "allow",
    "checks": ["scope", "allowlist", "budget"],
    "policy_version": "v0.1",
    "notes": null
  },
  "metrics": {
    "latency_ms": 380,
    "tokens_used": 312
  },
  "anomalies": []
}
```

Appendix C: AVC CLI Reference

Command syntax:

```
# Fork: spawn new agent from tick N
dendrer fork --run <run_id> --from tick <N> [--reason <string>]

# Diff: compare two traces
dendrer diff <runA> <runB>
dendrer diff <runA> <runB> --from tick 5 --to tick 20
dendrer diff <runA> <runB> --type semantic

# Bisect: find violation onset tick via binary search
# Predicate must be registered and prefix-monotone --- kernel validates at
     registration time
dendrer bisect <run_id> --predicate <module>
dendrer bisect <run_id> --predicate kernel.anomaly_free
dendrer bisect <run_id> --policy-flip <transition_type>
dendrer bisect <run_id> --predicate "<state_expression>"
dendrer bisect <run_id> --divergence <runB>
dendrer bisect <run_id> --regression <metric>   # extension only; non-
    monotone; see Appendix D

# Predicate registration (validates purity + prefix-monotonicity before
    use in bisect)
dendrer predicate register <module> [--verify-monotone --sample-trace <
    run_id>]

# Example: rejected registration (non-monotone predicate)
$ dendrer predicate register acme.quality_oscillator --verify-monotone \
    --sample-trace logistics-run-47

  testing predicate: acme.quality_oscillator
  FAIL NOT prefix-monotone
    tick 14: violation  FAIL
    tick 17: ok         OK  \textleftarrow{} predicate recovered ---
    binary search would return wrong tick
    tick 22: violation  FAIL

  REJECTED: predicate may be used for monitoring and diff but not for
    bisect
  Suggestion: use kernel.anomaly_free or define a predicate that does not
    recover

# Cherry-pick: context-validated intent import
dendrer cherrypick --run <run_id> --tick <N> --into <target_run>
```

Mock terminal output: cherry-pick with context violation (Section 4.4)

```
$ dendrer cherrypick --run financial-agent-A --tick 23 --into financial-
   agent-B

  resolving kernel contract for: execute_transaction
  required fields: [:risk_threshold, :cash_balance, :position_size]

  checking source state at tick 23...
  OK risk_threshold   MAX
  OK cash_balance     100000
  OK position_size    5000

  computing intent_context_hash...
  OK hash: sha256:a3f9...c12e  (covers required $\cup$ advisory deps)

  checking target branch state: financial-agent-B
  OK cash_balance     500
  OK position_size    200
  FAIL risk_threshold   MIN  \textleftarrow{}  contract requires match;
   source=MAX target=MIN

[!]  CONTEXT VIOLATION --- IMPORT REJECTED

  kernel contract:  execute_transaction
  missing match:    risk_threshold (source=MAX, target=MIN)
  risk:             SILENT_DISASTER_PREVENTED

  The intent at tick 23 was justified by risk_threshold=MAX.
  Target branch lacks this observation. Import would execute
  a large buy order under a conservative risk profile.
  No state was modified. No transition was committed.

  Suggested actions:
    (a) fork financial-agent-A --from tick 22
        then re-run with target risk profile
    (b) update financial-agent-B risk_threshold before import
        then retry cherrypick
```

Mock terminal output: bisect with remediation diagnosis (Section 4.5)

Contingent (state-level) — fork seed:

```
$ dendrer bisect logistics-run-47 --predicate kernel.policy_compliant

  verifying predicate: kernel.policy_compliant
  OK pure function (no I/O detected)
  OK prefix-monotone (verified against registration contract)

  bisect over 64 transitions  |  predicate: kernel.policy_compliant

  tick 32 \textrightarrow{} compliant OK
  tick 16 \textrightarrow{} compliant OK
  tick 24 \textrightarrow{} violation FAIL
  tick 20 \textrightarrow{} compliant OK
  tick 22 \textrightarrow{} violation FAIL
  tick 21 \textrightarrow{} compliant OK
  ORIGIN: tick 22  |  observation.add
  confidence: 0.31  |  source: unverified_feed_3
  effect: risk_score=0.83 crossed threshold=0.75 \textrightarrow{} policy
    denied at tick 23

  CAUSAL CHAIN:
    tick 22  observation.add    \textrightarrow{}  unverified source
    introduced
    tick 23  policy.decision    \textrightarrow{}  risk_score exceeded
    threshold; deny
    tick 27  action.request     \textrightarrow{}  denied; workaround
    attempted
    tick 32  kernel.anomaly     \textrightarrow{}  invariant breach

  DIAGNOSIS: contingent (state-level)
  The agent's reasoning was locally valid. The input data was not.

  REMEDIATION:
    dendrer fork --run logistics-run-47 --from tick 21
                 --tool-policy "require source verification before
    observation.add"
```

Foundational (prompt-level) — post-mortem artifact:

```
$ dendrer bisect planning-run-12 --predicate acme.budget_compliance_v1

  bisect over 48 transitions  |  predicate: acme.budget_compliance_v1

  tick 24 \textrightarrow{} violation FAIL
  tick 12 \textrightarrow{} violation FAIL
  tick 06 \textrightarrow{} violation FAIL
  tick 03 \textrightarrow{} violation FAIL
  tick 01 \textrightarrow{} violation FAIL

  ORIGIN: tick 01  |  plan.update
  note: violation present from first committed transition

  DIAGNOSIS: foundational (prompt-level)
  Violation onset at tick 1 indicates the goal specification or TaskPack
  policy is incompatible with the compliance predicate. No committed
  state is recoverable via fork --- the trajectory is logic-contaminated
  from boot.

  REMEDIATION:
    DO NOT fork --- forking from within this run will not resolve the
    issue.
    This trace is a post-mortem artifact.

    Recommended actions:
      (a) review goal specification for budget constraint alignment
      (b) update TaskPack policy: acme.budget_compliance_v1 requirements
      (c) re-run from scratch with revised specification
      (d) use this trace as reference for what to avoid
```

Appendix D: Progress-Based Bisect (Extension)

Progress regression bisect (`dendrer bisect --regression <metric>`) is an optional extension to core bisect. It uses the progress scoring layer (Section 8.4) as the bisect predicate rather than a hard invariant.

When it is useful: - Monitoring long runs for drift before a hard failure occurs - Comparing two runs where neither produced a hard anomaly but one performed better - Post-hoc analysis of goal completion quality

When it is unreliable: - Open-ended tasks with underspecified success criteria - Tasks where progress weights are miscalibrated - Any case where the three-signal composite does not reflect actual goal proximity

Usage:

```
dendrer bisect <run_id> --regression progress
dendrer bisect <run_id> --regression constraint_satisfaction
dendrer bisect <run_id> --regression skeptic_survival_rate
```

Critical rule: if progress-based bisect and invariant-based bisect disagree on the origin tick, invariant-based bisect takes precedence. Progress is a heuristic. Invariants are not. The system must make this precedence explicit in its output — not leave it to the user to reconcile.

Appendix E: Elixir/OTP Reference Implementation

This appendix contains the Elixir/OTP implementation of the mechanisms described in the main body using language-agnostic pseudocode. The runtime is built on BEAM (the Erlang virtual machine) with OTP supervision trees. The language choice is an implementation detail hidden behind the external HTTP/gRPC API.

E.1 Replay Isolation (ToolExecutor)

```
defmodule Dendrer.Runtime.ToolExecutor do

  # Live mode: execute against real world, record output in trace
  def execute(tool, params, %{mode: :live} = env) do
    result = tool.call(params)
    # Note: recorded_at is for operational logging only.
    # It must NOT enter diffable state --- exclude from canonical
   serialization.
    recorded = %{tool: tool.name, params: params, output: result, tick:
   env.tick}
    {:ok, result, recorded}
  end

  # Replay mode: world is frozen --- inject historical payload, block real
     IO
  def execute(tool, params, %{mode: :replay, tick: tick} = env) do
    case Map.fetch(env.recorded_outputs, tick) do
      {:ok, recorded} ->
        {:ok, recorded.output, nil}
      :error ->
```

```
        {:error, {:replay_tool_missing, tick, tool.name}}
    end
  end

end
```

E.2 Context Validation (ContextValidator)

```
defmodule Dendrer.AVC.ContextValidator do

  def validate_import(intent, source_hash, target_state, contracts) do
    required = Map.fetch!(contracts, intent.transition_type)
    missing = Enum.reject(required, &state_contains?(target_state, &1))

    case missing do
      [] ->
        case hash_covers_contract?(source_hash, required) do
          true  -> {:ok, :context_compatible}
          false -> {:error, {:hash_contract_mismatch, :hash_underspecified
   }}
        end
      missing ->
        {:error, {:context_violation,
          %{missing_deps: missing,
            intent: intent.transition_type,
            risk: :silent_disaster_prevented}}}
    end
  end

  defp state_contains?(state, dep), do: Map.has_key?(state, dep)
  defp hash_covers_contract?(hash, required), do: IntentContextHash.covers
   ?(hash, required)

end
```

E.3 Progress Contract Interface

```
defmodule Dendrer.ProgressContract do
  @moduledoc """
  Implementations must be pure functions over committed state.
  No I/O. No external calls. No randomness. Purity enforced at
   registration.
  """

  @callback evaluate(map()) ::
    {:ok, float()}
    | {:violation, term()}
    | {:uncertain, float()}

  @callback name()    :: String.t()
  @callback version() :: String.t()
end

defmodule Dendrer.Progress.KernelAnomalyFree do
  @behaviour Dendrer.ProgressContract
  def name,    do: "kernel.anomaly_free"
  def version, do: "v1"
  def evaluate(state) do
```

```
    if Enum.empty?(state["kernel_anomalies"]),
      do: {:ok, 1.0},
      else: {:violation, {:kernel_anomaly, state["kernel_anomalies"]}}
  end
end

# Template for institution-defined predicate
defmodule MyInstitution.Progress.CompliancePolicy do
  @behaviour Dendrer.ProgressContract
  def name,    do: "myinstitution.compliance_v1"
  def version, do: "v1"
  def evaluate(state) do
    # Pure function over state only --- no external calls
    score = PolicyEngine.score(state["artifacts"], __MODULE__.rules())
    if score >= 0.95,
      do: {:ok, score},
      else: {:violation, {:policy_breach, score}}
  end
end
```

Appendix G: Domain Example — Insurance Claims Adjuster Contract

The following TaskPack contract illustrates the four-pillar contract model (Section 4.4) in a non-software domain. Insurance claims adjudication is one of the most regulated decision environments outside of pharmaceutical manufacturing — every approval decision has legal, financial, and audit consequences. The contract below encodes a minimal institutional policy that the kernel enforces independently of model output.

This example is intended for readers who want to see the contract model applied outside of engineering workflows, and for reviewers evaluating the institutional accountability argument in Section 5.

```
id: claims_adjuster_v1
namespace: insurance.claims

# Pillar 1: Namespace and transition registry
# Governs: approve_payout, request_info

contracts:
  # Pillar 2: Layer 1 static state dependencies (kernel-enforced)
  # Pillar 3: Layer 2 intent context hash seeds
  approve_payout:
    required_state:
      claim_id:             {type: string}
      policy_limit:         {type: float, precision: 2}
      payout_amount:        {type: float, precision: 2}
      # state_before.total_paid_to_date is the running total BEFORE this
      # transition is applied. The post-state total is computed by the
      # reducer: state_after.total_paid_to_date =
      #   state_before.total_paid_to_date + payout_amount
      # The violation predicate evaluates over state_after, ensuring
      # the check reflects the committed effect of this transition.
      total_paid_to_date:   {type: float, precision: 2}
      # nullable: true --- absence of a URL is itself meaningful and must
      # be committable to state (e.g. during intake before doc upload).
      # The unverified_payout predicate then catches the null case.
      proof_of_loss_url:     {type: string, nullable: true}
      # ISO 8601 timestamp of when deductible was confirmed, or null.
      # Stronger than boolean: the audit trail records *when* confirmation
      # occurred, enabling temporal audits ("was deductible confirmed
      # before or after the claim was filed?")
      deductible_confirmed_at: {type: string, nullable: true}
    # Layer 2: claim_id added --- binds this approval to a specific claim,
    # preventing cherry-pick of an approval into a different claim's
    context.
    hash_seeds: [claim_id, policy_limit, proof_of_loss_url, payout_amount]

  request_info:
    required_state:
      claim_id:           {type: string}
      missing_field_name: {type: string}
    hash_seeds: [claim_id]

# Pillar 4: Prefix-monotone violation predicates (pure, no I/O)
violations:
  - id: over_limit_payout
    name: "Cumulative Payout Exceeds Policy Limit"
    scope: [approve_payout]
    # Evaluates over state_after --- the post-transition committed state.
    # state_after.total_paid_to_date includes the current payout_amount.
    # Prefix-monotone: running total is append-only, never decreases.
    state: after
    condition: "state.total_paid_to_date > state.policy_limit"
    pure: true
    monotone: true

  - id: unverified_payout
    name: "Payout Without Proof of Loss"
    scope: [approve_payout]
    # Catches both null and empty string --- either indicates no document.
    condition: "state.proof_of_loss_url == null or state.proof_of_loss_url
     == ''"
    pure: true
    monotone: true

  - id: deductible_not_confirmed
    name: "Payout Without Deductible Confirmation Timestamp"
    scope: [approve_payout]
    # Null timestamp means deductible was never confirmed.
    # Stronger than boolean: presence of timestamp is auditable evidence.
    condition: "state.deductible_confirmed_at == null"
    pure: true
    monotone: true
```

Reading this contract:

The three violations are independent and each bisect-safe. Together they form a minimal auditable policy for claims approval: no payout may exceed the policy limit (cumulative, checked against post-transition state), no payout may occur without a verified proof of loss document, and no payout may be issued without a deductible confirmation timestamp.

Four design decisions worth noting:

Pre vs. post state. `over_limit_payout` explicitly evaluates `state_after` — the committed state after `payout_amount` has been added to `total_paid_to_date`. Evaluating `state_before` would miss the effect of the current transition entirely. For accumulating predicates, always specify which state window the condition applies to.

Nullable fields. `proof_of_loss_url` is nullable because its absence is meaningful — during intake, before document upload, the field legitimately does not exist. Making it non-nullable would prevent the agent from committing valid intermediate states. The `unverified_payout` predicate handles both null and empty string, closing the gap that a naive `== null` check would miss.

Timestamp over boolean. `deductible_confirmed_at` records *when* confirmation occurred, not merely *whether* it occurred. This enables temporal audit queries ("was the deductible confirmed before or after the claim was filed?") that a boolean cannot support. For regulated domains, the timestamp is evidence; the boolean is just a flag.

`claim_id` in hash seeds. Adding `claim_id` to the hash seeds means an approval intent is cryptographically bound to a specific claim. Cherry-picking an approval from claim A into a run processing claim B fails the context check before any predicate runs. Without this, an agent could theoretically replay a valid approval from one claim context into another.

A note on timestamp representation. `deductible_confirmed_at` is typed as `string` here for YAML portability, with the expectation that values conform to ISO 8601 with timezone offset (e.g. `2026-02-26T14:30:00+01:00`). A production schema should use a dedicated `datetime` type with explicit timezone enforcement and serialization rules — both to prevent ambiguous comparisons during replay and to ensure canonical serialization across environments. The string representation is sufficient for demonstration purposes but should not be used in regulated deployments without a stricter type contract.

An auditor can verify the institutional policy by reading this contract without reading a line of agent code. The kernel enforces it without trusting model output. This is the "as executed = as documented" property (Section 5.2) applied to a regulated decision: not by discipline or regulation, but by architecture.

Appendix F: Glossary

Term	Definition
Agent trajectory	A finite, ordered sequence of committed transitions T = $[t_1 \ldots t_n]$ produced by an agent run
Audit fidelity	The property that replay reconstructs every state the agent passed through and every decision it made — distinct from behavioral equivalence
Behavioral equivalence	The stronger claim that the agent would produce the same trajectory if run again live — not claimed by this paper

Term	Definition
bisect	Binary search over an immutable trace to find the violation onset tick k* = min{k : P(s_k) = violation}; requires prefix-monotone, pure predicate
Causal contributor set	Heuristic: the minimal set of earlier transitions whose removal would have prevented a violation; distinct from the onset tick
cherry-pick	AVC operation importing a committed intent from one branch into another; validated by the intent context hash against Layer 1 contracts
Committed transition	A state change that has passed kernel validation and been written immutably to the trace
Contract (TaskPack)	Static, versioned specification declaring required state fields, quantization rules, hash seeds, and violation predicates for a set of transition types
diff	AVC operation comparing two traces field-by-field over committed state; requires canonical serialization
fork	AVC operation spawning a new agent run from a committed state at tick k; the forked run shares history up to k
Foundational failure	A failure whose origin is upstream of committed state — in the goal specification, system prompt, or TaskPack — not remediable via fork
Contingent failure	A failure caused by bad input data or misconfigured policy at a specific tick; remediable via fork from that tick
Intent	An LLM-proposed state transition, not yet committed; exists in the intent proposal layer before kernel validation
intent_context_hash	Cryptographic commitment over `{transition_type, action, {field: quantized_value} for required UNION advisory deps}`; certifies kernel contract was satisfied
kernel.anomaly	Runtime-detected invariant violation: replay divergence, schema mismatch, missing tool output, reducer non-determinism. Always decidable. Unconditionally prefix-monotone
Layer 1 contract	Static, kernel-enforced field requirements per transition_type; independent of model output
Layer 2 hash	Intent context hash generated after Layer 1 passes; binds field values (not names) to the intent
Logic-contaminated	A trajectory where the failure originates in the goal specification or prompt; the entire trace is a post-mortem artifact, not a fork seed
Macroscopic interpretability	Interpretability defined as procedural causality: identifying which committed state transition caused a behavioral outcome
Model class M	Agent executions (T, R, s_0, P) where T is a committed transition sequence, R is the reducer, s_0 the initial state, P the predicate

Term	Definition
predicate.violation	Institution-defined policy breach evaluated over committed state; decidable only if the predicate is pure and prefix-monotone
Prefix-monotone	A predicate P where once $P(s_k)$ = violation, $P(s_j)$ = violation for all $j > k$; the set of violating ticks forms a suffix interval [k..n]
Procedural causality	Causal attribution at the level of committed state transitions, not internal model computations
ProgressContract	Interface for institution-defined progress/violation predicates; enforces purity and prefix-monotonicity at registration time
Pure reducer	A reducer $R : (\text{State} \times \text{Transition}) \rightarrow \text{State}$ that is a deterministic pure function; same inputs always yield same output
Reducer	The pure function R that maps (state, transition) → state; the computational heart of deterministic replay
Replay	Deterministic reconstruction of any historical state s_k by applying R to transitions $t_1 \ldots t_k$ from s_0; requires pure reducer and canonical serialization
Replay isolation	The mechanism that injects recorded tool outputs during replay, blocking live IO; separates internal divergence from external nondeterminism
Snapshotable environment	An execution environment where full agent state can be captured and restored at any tick boundary (e.g., Firecracker, WASM)
TaskPack	The complete specification bundle for an agent task: goal, policy, tool allowlist, transition contracts, and violation predicates
Tick	The kernel-assigned monotonic index of a committed transition; the temporal coordinate of the trace
Trace	The append-only, immutable, coordinator-ordered sequence of all committed transitions for an agent run
Violation onset tick k*	The first tick k such that $P(s_k)$ = violation; what bisect computes; distinct from the causal contributor set