Executable Monotonic Continuation

Runtime Data Structures, Steering Absorption, and Benchmarking Trajectory Adequacy

Flyxion
Independent Researcher

August 2026

Abstract

Monotonic Continuation: Intelligence Across Unequal Timescales develops a theoretical account of multi-timescale cognitive architecture organized around a single ordering relation,

$$S_t \preceq S_{t+1},$$

read as continuational adequacy rather than sequential execution, monotonic accumulation of facts, or byte-for-byte preservation of state. A later state is adequate when it preserves the distinctions, commitments, unresolved questions, and admissible continuations that remain relevant from the earlier state while allowing those structures to be transformed, compressed, translated, or repaired. The source theory therefore treats intelligence not principally as the production of correct answers at isolated moments, but as the capacity to continue without unnecessarily destroying what remains worth continuing.

That account is deliberately a work of formal ontology rather than systems engineering. It distinguishes presently known states K, unresolved but reachable states U_R, and presently unreachable states U_P; defines steering as a trajectory-conditioned intervention; distinguishes explicit steering from steering absorbed into a transition rule; introduces hierarchical memory and cross-model translation; and treats repair as a family of nested processes operating on unequal temporal horizons. It does not, however, specify what an unresolved-but-reachable state looks like in memory, how a runtime determines that such a state has become reachable, how repeated steering is safely converted into standing authorization, how continuational adequacy can be approximated when the full reachable set is computationally inaccessible, or how the resulting architecture should be compared experimentally with conventional agents.

This companion paper addresses those questions by treating the ontology as an implementation specification. We introduce persistent Epistemic Frames for representing unresolved computation across asynchronous cycles; a category-indexed authorization lattice for recording steering history without collapsing trust into a scalar; an executable steering-absorption algorithm in which repeated approval is necessary but not sufficient for promotion; and a hierarchical repair pipeline corresponding to the theoretical budgets

$$R_\mu \subseteq R_m \subseteq R_L.$$

Because the exact relation $S_t \preceq S_{t+1}$ is generally undecidable or computationally intractable for realistic systems, the implementation does not claim to test continuational adequacy directly. Instead it maintains operational surrogates: prerequisite graphs, invariant sets, reversible checkpoints, reachable-set summaries, provenance records, and delayed validation results. This distinction between the theoretical object and its runtime proxy is central. An implementation that silently identifies its proxy with $\mathcal{R}(S_t)$ would reproduce, at the engineering level, the premature-resolution error the theory was introduced to avoid.

We then propose a trajectory-oriented benchmark suite. Its principal measures are foreclosure rate, epistemic-state retention, steering leverage, steering-absorption safety, repair efficiency, and

recovery depth. Unlike conventional single-turn accuracy or pass/fail benchmarks, these metrics evaluate what happens to the space of admissible future action as an agent proceeds. A system can therefore receive credit for correctly declining to resolve an unresolved question, and can be penalized for an apparently successful intermediate action that irreversibly destroys a later valid solution.

Interactive software engineering provides the initial experimental domain because it combines long trajectories, heterogeneous timescales, observable side effects, partial reversibility, executable tests, and naturally occurring human steering. We specify controlled comparisons among direct-execution, synchronous repair, speculative-planning, and monotonic-continuation agents, together with ablations designed to isolate the contribution of epistemic-state representation, authorization absorption, and multi-horizon repair.

The resulting proposal is not offered as a proof of the source ontology. It is a falsifiable engineering interpretation of it. Its purpose is to determine whether systems that explicitly preserve unresolved state and continuational option value behave differently, and measurably better, from systems optimized primarily for immediate resolution.

1 From Theory to Runtime

The source theory's distinction among K, U_R, and U_P begins from a simple observation with substantial architectural consequences: absence of an answer is not one computational state.

A system may lack an answer because the answer has not yet been computed, because a slower process is currently computing it, because some prerequisite information has not yet arrived, because the problem is presently outside the system's capabilities, because the question has been deliberately deferred, or because the system has silently forgotten that the question existed. Conventional interfaces often collapse all of these into a null field, timeout, exception, low-confidence answer, or missing value.

For a single-clock program this collapse may be tolerable. For a multi-timescale cognitive architecture it is destructive. A fast process must be capable of continuing while a slow process remains unfinished, and the slow process must later be able to re-enter the trajectory without reconstructing from scratch why its work mattered.

The implementation therefore begins by turning unresolvedness into persistent state.

1.1 The Epistemic Frame

The basic runtime object is an Epistemic Frame:

```
EpistemicFrame<T> {
    id: UUID
    status: EpistemicStatus

    query: QueryDescriptor
    payload: Option<T>
    confidence: Option<Float>

    trajectory_id: UUID
    parent_frames: Set<UUID>
    prerequisites: Set<UUID>
    dependents: Set<UUID>

    created_at: Timestamp
    last_updated_at: Timestamp
    next_review_at: Option<Timestamp>

    assigned_clock: ClockTier
    repair_horizon: Duration

    provenance: List<EvidenceRef>
    commitments: Set<Commitment>
    invariants: Set<Invariant>

    continuation_summary: ContinuationSummary
    checkpoint_ref: Option<CheckpointID>
```

```
    revision: Integer
}
```

with

EpistemicStatus = KNOWN | UNRESOLVED_REACHABLE | UNRESOLVED_UNREACHABLE | INVALIDATE

The final two implementation states do not enlarge the source theory's epistemology. They are bookkeeping distinctions needed to prevent historical information from disappearing when a frame changes. A proposition once treated as K may later be invalidated; an unresolved question may be replaced by a better-formulated one. Deleting the old frame would destroy trajectory provenance. The runtime therefore distinguishes the current epistemic classification from the historical lifecycle of the object carrying it.

This is already one practical consequence of monotonic continuation: correction should normally be represented as supersession rather than erasure.

1.2 Why Option$\langle T \rangle$ Is Not Enough

The distinction can be seen most clearly by comparing the Epistemic Frame with the ordinary type

$$\text{Option}\langle T \rangle = \text{Some}(T) \mid \text{None}$$

None says only that T is absent. It cannot distinguish

$$U_R \neq U_P.$$

A richer type can:

```
Epistemic<T> =
    Known(T)
  | Reachable(PendingHandle)
  | Unreachable(PrerequisiteSet)
```

This is the software analogue of the theoretical insistence that *not yet* be representable.

The point is not terminological. Suppose a repository agent is asked to modify an API whose behavior depends on a full integration test taking twelve minutes. The interaction process can inspect the code in seconds. At $t = 5$ seconds it knows that the API is ambiguous but also knows that the integration test is capable of resolving the ambiguity.

Representing the state as U_R allows the system to say, computationally rather than rhetorically:

> this distinction remains unresolved, but a known process is currently capable of resolving it.

Representing the same state as None loses the fact that resolution is underway. Representing it as K fabricates closure. Representing it as U_P loses known reachability.

These are behaviorally different mistakes.

1.3 Reachability as Dependency

Let the runtime maintain a prerequisite graph $G_P = (V, E)$, where each $v \in V$ is an Epistemic Frame and $(v_i, v_j) \in E$ means that resolution of v_j depends on v_i.

Then a minimal operational definition is

$$E_j \in U_R \iff \bigwedge_{E_i \in \text{Pred}(E_j)} E_i \in K \ \wedge \ \text{ResolverAvailable}(E_j).$$

Conversely, $E_j \in U_P$ when at least one indispensable prerequisite remains unavailable or no currently registered resolver can make progress on the frame.

This makes reachability dynamic rather than intrinsic. A question may move $U_P \to U_R$ without its semantic content changing at all. What changed is the surrounding computational world.

That distinction matters especially in tool-using systems. Installing a compiler, obtaining repository access, receiving a test result, loading a larger model, or receiving a human clarification can all transform an unreachable question into a reachable one.

1.4 The Continuation Summary

A bare cryptographic fingerprint cannot tell us whether two semantically different reachable sets overlap. It can establish identity, but not adequacy. We therefore define a structured continuation summary $\widehat{\mathcal{R}}(S_t)$ as an approximation to the theoretical $\mathcal{R}(S_t)$:

```
ContinuationSummary {
    hard_invariants: Set<Invariant>
    soft_commitments: Set<Commitment>
    open_frames: Set<UUID>
    admissible_action_classes: Set<Category>
    forbidden_action_classes: Set<Category>
    unresolved_branches: Set<BranchID>
    reversible_checkpoint: CheckpointID
    semantic_digest: Embedding
    exact_digest: Hash
}
```

The notation is intentionally $\widehat{\mathcal{R}}$ rather than $\mathcal{R}$. The runtime does not possess the full future reachable set. In any sufficiently complex environment, computing that set would itself require solving most of the task.

The approximation therefore records what the system currently has grounds to preserve.

This gives an operational adequacy predicate $S_t \widehat{\preceq} S_{t+1}$ defined by a conjunction of checks such as

$$\text{HardInv}(S_t) \subseteq \text{HardInv}(S_{t+1}), \qquad \text{OpenRequired}(S_t) \subseteq \text{Recoverable}(S_{t+1}),$$

and

$$\text{AdmissibleBranches}(S_t) \not\perp \text{AdmissibleBranches}(S_{t+1}).$$

The last condition deliberately does not require every branch to survive. Intelligent action necessarily eliminates possibilities. The relevant question is whether a transition eliminates possibilities without adequate grounds.

That is the operational form of foreclosure.

1.5 Resolution Is Not Necessarily Monotonic in Confidence

An important implementation consequence follows. Monotonic continuation must not be confused with monotonically increasing confidence.

A slow process can discover that an apparently settled frame was less secure than previously believed: $K \to U_R$. At first glance this looks non-monotonic. Under the source relation it need not be. If reopening the question restores distinctions that were incorrectly collapsed, the new state can be continuationally superior despite containing less certainty.

Thus

$$S_t \preceq S_{t+1} \not\Rightarrow \text{Confidence}(S_{t+1}) \geq \text{Confidence}(S_t).$$

A runtime that forbids $K \to U_R$ transitions would therefore confuse epistemic monotonicity with continuational monotonicity.

The state machine should instead permit $U_P \to U_R \to K$ as the ordinary resolution path while also permitting $K \to U_R$ under contradiction, failed validation, changed prerequisites, or newly acquired evidence.

1.6 Asynchronous Resolution

The event loop can now be written more precisely.

Algorithm 1: Asynchronous Epistemic Resolution

```
Input:
    EpistemicFrame E
    StateSnapshot S_t
    ContinuationSummary R_hat_t

Output:
    Updated frame E'
    optional RepairEvent

1   RefreshPrerequisites(E)

2   if E.status == UNRESOLVED_UNREACHABLE then
3       if PrerequisitesSatisfied(E)
4          and ResolverAvailable(E) then
5              E.status <- UNRESOLVED_REACHABLE
6              E.assigned_clock <- SelectClock(E)
7              E.repair_horizon <- AllocateRepairHorizon(E, S_t)
8       end if
9   end if

10  if E.status == UNRESOLVED_REACHABLE then
```

```
11      result <- AsyncComputeStep(E, S_t, E.repair_horizon)

12      if result.status == SETTLED then
13          candidate <- CommitCandidate(E, result)
14          if Validate(candidate, E.invariants) then
15              E.status <- KNOWN
16              E.payload <- result.value
17              E.provenance <- E.provenance U result.evidence
18          else
19              EmitRepairEvent(E, VALIDATION_FAILURE)
20          end if

21      else if result.status == PROGRESS then
22          R_hat_next <- SummarizeContinuation(result.next_state)

23          if ApproxAdequate(R_hat_t, R_hat_next) then
24              E <- AdvanceWithoutResolution(E, result)
25          else
26              RestoreCheckpoint(E.checkpoint_ref)
27              EmitRepairEvent(E, FORECLOSURE_RISK)
28          end if

29      else if result.status == BLOCKED then
30          E.status <- UNRESOLVED_UNREACHABLE
31          E.prerequisites <-
                E.prerequisites U result.missing_requirements
32      end if
33  end if

34  Persist(E)
35  return E
```

The PROGRESS case is crucial. Conventional agent loops often treat every computational episode as if it ought to terminate in an answer. Here a computation can succeed by improving the state from which later computation will continue.

That is:

$$\text{progress} \neq \text{resolution}.$$

A system can learn which hypothesis is impossible, discover which file contains the relevant implementation, narrow an error to one subsystem, or establish that two proposed fixes are observationally equivalent without yet knowing which final patch is correct. All of these can constitute successful transitions under $\preceq$.

1.7 Garbage Collection Without Epistemic Amnesia

Persistent unresolved state creates an obvious systems problem: if every uncertainty becomes a durable object, memory grows without bound.

The answer cannot simply be deletion after a timeout, because time alone does not imply irrelevance.

Instead frames are garbage-collected only when one of three conditions holds:

$$\text{Superseded}(E), \qquad \text{NoDependents}(E) \wedge \text{TaskClosed}(E), \qquad \text{or} \qquad \text{ProvablyIrrelevant}(E, S_t).$$

Even then, the full frame need not remain in active memory. It can be compressed into archival provenance:

$$E \to \text{Summary}(E).$$

This gives the runtime the same distinction the source theory makes between preserving a trajectory and preserving its literal substrate. Continuational memory is allowed to compress. What it must not do is silently erase distinctions that later states still depend upon.

2 Category Authorization and Steering Absorption

Steering absorption becomes considerably more interesting when treated as an executable process rather than a metaphor for learning.

Suppose a user repeatedly approves a software agent's proposed local source edits. Initially each transition has the form

$$S_{t+1} = F(S_t, u_t^{(c)}),$$

where $u_t^{(c)}$ is an explicit authorization belonging to category c.

After sufficient history, the user may no longer need to supply that authorization:

$$S_{t+1} = F^{(c)}(S_t).$$

The obvious implementation is a trust score.

That implementation is wrong.

A scalar trust variable would allow evidence from safe actions to leak into unrelated dangerous actions. An agent that has correctly read ten thousand files has acquired almost no evidence that it should be allowed to rewrite git history. An agent that has safely modified documentation has acquired little evidence about whether it should automatically perform database migrations.

Authorization therefore has topology.

2.1 The Authorization Lattice

Let C be a partially ordered taxonomy:

```
action
  fs
     read
     write
         docs
         tests
         local_src
```

```
        generated
  process
      inspect
      execute
  net
      read
      mutate
  vcs
      inspect
      commit
      push
      rewrite_history
```

Authorization is then a map

$$\tau : C \to L, \qquad L = \{\,\text{Never}, \text{Explicit}, \text{Bounded}, \text{Unconditional}\,\}.$$

The essential point is that $\tau(c_1) \not\Rightarrow \tau(c_2)$ unless an explicit inheritance rule exists between c_1 and c_2.

This makes trust not a quantity but a structured policy surface.

2.2 Authorization Requires Scope

Even category is insufficient. Approval of `fs.write.local_src` in one repository should not automatically authorize the same operation in every repository. The complete authorization key is therefore closer to (c, σ, χ), where c is action category, σ is scope, and χ is a constraint context.

```
AuthorizationRecord {
    category: CategoryPath
    scope: ScopeDescriptor

    level: AuthorizationLevel

    approvals: Integer
    rejections: Integer
    consecutive_approvals: Integer

    invariant_constraints: Set<Predicate>
    max_effect_radius: EffectRadius
    reversibility_required: Bool

    promoted_at: Option<Timestamp>
    last_violation: Option<ViolationRecord>

    evidence_window: RingBuffer<DecisionRecord>
}
```

The same action can consequently have $\tau(\texttt{fs.write}, \texttt{repo-A}) = \text{Bounded}$ while $\tau(\texttt{fs.write}, \texttt{repo-B}) = \text{Explicit}$.

This captures something closer to actual human trust: permission is learned in a domain.

2.3 Approval Is Evidence, Not Proof

Repeated approval provides evidence for $F^{(c)}(S_t) \approx F(S_t, u_t^{(c)})$, but cannot establish it universally.

Let $A_c = (a_1, \ldots, a_n)$ be the approval history for category c, and define a posterior estimate

$$p_c = \mathbb{P}(\text{autonomous action acceptable} \mid A_c, \mathcal{H}_c).$$

Promotion can then require $p_c > \alpha_c$ together with invariant satisfaction and an effect-radius constraint.

This is better than a raw streak because ten approvals of nearly identical trivial edits should provide less evidence than ten approvals distributed across the category's operational range.

A useful evidence score is therefore

$$E_c = n_c \cdot D_c \cdot (1 - V_c),$$

where n_c is approval count, $D_c \in [0, 1]$ measures diversity of approved contexts, and V_c is observed violation rate.

Steering absorption occurs only when $E_c \geq \eta_c$.

The architecture is therefore learning not merely that the user repeatedly said yes, but approximately where the yeses occurred.

2.4 Reversibility Changes the Absorption Threshold

The repair-budget framework gives a principled reason for category-specific thresholds.

Let $\text{rev}(c) \in [0, 1]$ measure practical reversibility and let $d(c)$ measure maximum downstream effect radius. Then the promotion threshold can be made proportional to

$$\eta_c \propto \frac{d(c)}{\text{rev}(c) + \epsilon}.$$

A local source edit performed on a checkpointed branch may be highly reversible and therefore eligible for rapid absorption. A remote database deletion may be effectively irreversible and should require an enormous threshold or simply remain $\tau(c) = \text{Explicit}$ permanently.

This connects steering absorption directly to repair budgets. Autonomy is safest where mistakes remain inside a cheap repair horizon.

2.5 The Absorption Engine

Algorithm 2 can therefore be strengthened:

Algorithm 2: Steering Absorption and Authorization Escalation

```
Input:
    category c
    scope sigma
    proposed action a_t
    feedback f_t
```

```
    state S_t
    authorization map tau

1   r <- tau[c, sigma]
2   effect <- EstimateEffectRadius(a_t)
3   reversible <- EstimateReversibility(a_t)

4   if f_t == APPROVED then
5       AppendEvidence(r, a_t, S_t, APPROVED)
6       r.consecutive_approvals += 1

7       evidence <- ComputeEvidenceStrength(r)
8       risk <- ComputeRisk(effect, reversible)

9       if evidence >= PromotionThreshold(c, risk) then
10          if InvariantsHold(r, S_t, a_t)
11             and CoverageSufficient(r.evidence_window)
12             and NoUnresolvedHighRiskFrameDependsOn(a_t) then
13                 r.level <- Promote(r.level)
14                 r.promoted_at <- now()
15          end if
16      end if

17  else if f_t == REJECTED then
18      AppendEvidence(r, a_t, S_t, REJECTED)
19      r.consecutive_approvals <- 0
20      r.level <- Demote(r.level)
21      OpenEpistemicFrame(
            question = "Why was category c rejected here?",
            status = UNRESOLVED_REACHABLE
        )
22  end if

23  tau[c, sigma] <- r
24  return tau
```

Line 21 is important. Rejection should not merely reduce a score. It is new information about the boundary of the category.

The runtime should ask: was the category wrong, or was its scope too broad? A rejection of one generated-file edit need not prove that generated-file editing should never be autonomous. It may reveal a missing subcategory: $c \to \{c_{\text{safe}}, c_{\text{confirm}}\}$.

Thus steering absorption can refine the authorization taxonomy itself.

That is closer to learning than simple permission accumulation.

2.6 Absorption Debt

Promotion creates a new systems quantity: absorption debt.

Once explicit steering disappears, the system stops receiving confirmation data on every action. The evidence stream that justified autonomy therefore becomes thinner precisely when autonomy begins.

Define

$$D_{\text{abs}}^{(c)}(t) = \frac{\text{autonomous actions since last audit}}{\text{validated autonomous actions} + 1}.$$

As D_{abs} rises, the runtime should periodically reintroduce validation:

$$D_{\text{abs}}^{(c)} > \delta_c \;\Rightarrow\; \text{audit}.$$

This prevents steering absorption from becoming an irreversible one-way ratchet.

Autonomy is therefore not *confirmation → no confirmation forever*. It is *dense confirmation → sparse confirmation → renewed confirmation when evidence degrades*.

That distinction becomes especially important under distribution shift.

3 Multi-Horizon Repair as an Executable Scheduler

The relation $R_\mu \subseteq R_m \subseteq R_L$ is too important to remain merely a benchmark variable. The repair hierarchy needs an implementation.

Let R_μ, R_m, R_L denote micro-, meso-, and long-horizon repair processes. They differ not merely in duration but in scope of state they are permitted to reconsider.

A micro-repair might correct syntax while preserving the current plan. A meso-repair may reconsider several recent actions. A long repair may reopen an architectural assumption that has governed the entire trajectory. Thus

$$\text{Scope}(R_\mu) \subseteq \text{Scope}(R_m) \subseteq \text{Scope}(R_L).$$

3.1 Repair Events

```
RepairEvent {
    id: UUID
    trajectory_id: UUID

    detected_at_step: Integer
    suspected_origin_step: Integer

    severity: Float
    affected_frames: Set<UUID>
    violated_invariants: Set<Invariant>

    minimum_candidate_tier: RepairTier
    assigned_tier: RepairTier

    checkpoint_candidates: List<CheckpointID>
}
```

The distinction between `detected_at_step` and `suspected_origin_step` is essential.

Many trajectory errors are delayed. A bad abstraction chosen at S_7 may not cause a failing integration test until S_{31}. Repairing only S_{31} treats the symptom as the cause. Define error depth

$$d_e = t_{\text{detect}} - t_{\text{origin}}.$$

Large d_e should preferentially escalate repair to a slower tier.

3.2 Tier Selection

A simple scheduler can define a repair score

$$\Gamma(e) = w_1 d_e + w_2 r_e + w_3 n_e + w_4(1 - \text{rev}_e),$$

where r_e is effect radius, n_e is the number of implicated invariants, and rev_e is reversibility. Then

$$R(e) = \begin{cases} R_\mu, & \Gamma(e) < \gamma_1, \\ R_m, & \gamma_1 \le \Gamma(e) < \gamma_2, \\ R_L, & \Gamma(e) \ge \gamma_2. \end{cases}$$

The exact thresholds are empirical parameters, not theoretical constants. The architecture's claim is only structural: cheap local repair should be attempted where the evidence indicates a local failure, while evidence of deeper trajectory corruption should permit increasingly large portions of the past to be reconsidered.

3.3 Repair Is Not Rollback

A particularly important distinction is between repair and rollback.

Rollback produces $S_t \to S_k$, $k < t$. Repair ideally produces $S_t \to S'_{t+1}$ where S'_{t+1} incorporates information learned between k and t while avoiding the failed commitment made at k.

This is why the trajectory log matters. A system that merely restores an old checkpoint may repeat the same mistake. Repair should instead construct

$$S'_{t+1} = \text{Revise}(S_k, \Delta_{k:t}, e_t),$$

where $\Delta_{k:t}$ is the useful information accumulated after the defective decision and e_t is the evidence revealing the defect.

This is monotonic continuation in perhaps its clearest executable form: go back without going blank.

4 Benchmarking Trajectory Adequacy

This section benchmarks three actual mechanisms rather than two mechanisms plus an abstract repair hierarchy. The fundamental unit of evaluation is no longer an answer. It is a trajectory

$$T = (S_0, a_0, S_1, a_1, \ldots, S_N)$$

together with a changing set of admissible task completions. The benchmark asks not merely whether the agent eventually succeeded, but what it destroyed, preserved, deferred, repaired, and learned on the way.

4.1 Foreclosure Rate

Hard foreclosure:

$$\mathcal{R}^*(S_t) \neq \emptyset, \qquad \mathcal{R}^*(S_{t+1}) = \emptyset.$$

Soft foreclosure occurs when valid continuations remain but an unnecessary fraction has been destroyed:

$$\phi_s(t) = 1 - \frac{|\mathcal{R}^*(S_{t+1})|}{|\mathcal{R}^*(S_t)|}.$$

Because exact reachable sets will usually be unavailable, the benchmark can approximate them through enumerated solution branches, test-equivalence classes, or counterfactual replay. This gives Φ_{hard} and Φ_{soft}.

A system that repeatedly reduces ten viable solutions to one viable solution is behaviorally different from one that preserves several until evidence justifies commitment, even if both eventually pass the tests.

4.2 Epistemic State Retention Index

ESRI should penalize both premature closure and forgotten uncertainty:

$$\text{ESRI} = \frac{N_{\text{correctly retained and eventually resolved}}}{N_{\text{ground-truth delayed-resolution frames}}}.$$

Its failures decompose into

$$E_{\text{premature}} = \mathbb{P}(U_R \to K_{\text{wrong}}) \qquad \text{and} \qquad E_{\text{abandon}} = \mathbb{P}(U_R \to U_{P,\text{spurious}}).$$

These are conceptually different. The first is impatience. The second is amnesia.

4.3 Resolution Latency Ratio

A useful new metric is

$$\Lambda_R = \frac{t_{\text{declared resolution}}}{t_{\text{earliest justified resolution}}}.$$

Values $\Lambda_R < 1$ indicate premature resolution. Values near 1 indicate appropriately timed resolution. Very large values indicate pathological conservatism: the architecture knows enough to decide but continues deferring.

This prevents monotonic continuation from winning the benchmark simply by refusing to commit.

4.4 Continuation Option Value

For benchmark tasks where valid branches can be enumerated, define

$$\Omega_t = \log\bigl(1 + |\mathcal{R}^*(S_t)|\bigr).$$

The relevant quantity is not maximizing Ω_t indefinitely. A competent agent should eliminate branches as evidence accumulates. Instead compare option loss with information gain:

$$Q_t = \frac{-\Delta\Omega_t}{\Delta I_t + \epsilon}.$$

Large option loss with little new evidence signals unjustified commitment. This gives a quantitative approximation to the intuition behind premature resolution:

Do not spend future option value without acquiring information that warrants spending it.

4.5 Conditional Steering Leverage

The information-theoretic form is theoretically attractive but difficult to estimate from a single trajectory. We retain it as the ideal metric and add an empirical surrogate.

Define intervention compression:

$$\mathcal{L}_{\text{emp}} = \frac{\Delta\mathbb{P}_{\text{success}}(u_t \mid S_t)}{|u_t|}.$$

Or, with paired counterfactual runs,

$$\mathcal{L}_{\text{cf}} = \frac{D(T_{u_t}, T_{\varnothing})}{|u_t|},$$

where D measures divergence toward the correct solution trajectory. This is experimentally estimable: clone the state at S_t, run one branch with the steering intervention and one without it, and measure how much the intervention changes downstream success.

The source quantity $I_{\text{eff}}(u_t; S_{t+1} \mid S_t)$ then remains the theoretical interpretation rather than being falsely presented as something directly observable from a log.

4.6 Safe Absorption Rate

Speed and safety should be reported separately before optionally combining them. Define

$$T_{\text{abs}}^{(c)} = \text{number of confirmations before promotion}$$

and

$$P_{\text{safe}}^{(c)} = \mathbb{P}(\text{no invariant violation} \mid \text{autonomous category } c).$$

Then

$$V_{\text{abs}}^{(c)} = \frac{P_{\text{safe}}^{(c)}}{T_{\text{abs}}^{(c)}}.$$

This makes the tradeoff transparent. A system that never absorbs steering has excellent safety but infinite absorption time. A system that absorbs immediately has excellent speed but potentially catastrophic safety. The desirable region lies on a Pareto frontier rather than at the maximum of either quantity alone.

4.7 Repair Efficiency

Let A_i be repair attempts at tier i, S_i successful repairs, and c_i their cost. Then

$$\text{REI} = \frac{\sum_i w_i S_i}{\sum_i c_i A_i},$$

with success weighted by prevented downstream cost. More revealing still is repair escalation rate

$$E_{i \to j} = \mathbb{P}(R_i \text{ fails and escalates to } R_j).$$

A good hierarchy should exhibit high local resolution for genuinely local failures but should not artificially suppress escalation when the defect is structural. Thus a low escalation rate is not automatically good.

4.8 Repair Depth Accuracy

Because the benchmark can seed known defects at different trajectory depths, it can measure whether the architecture selects an appropriate repair horizon:

$$\text{RDA} = 1 - \frac{|d_{\text{selected}} - d_{\text{true}}|}{d_{\max}}.$$

This tests one of the most distinctive claims of the architecture: errors belong to different clocks.

5 Experimental Methodology

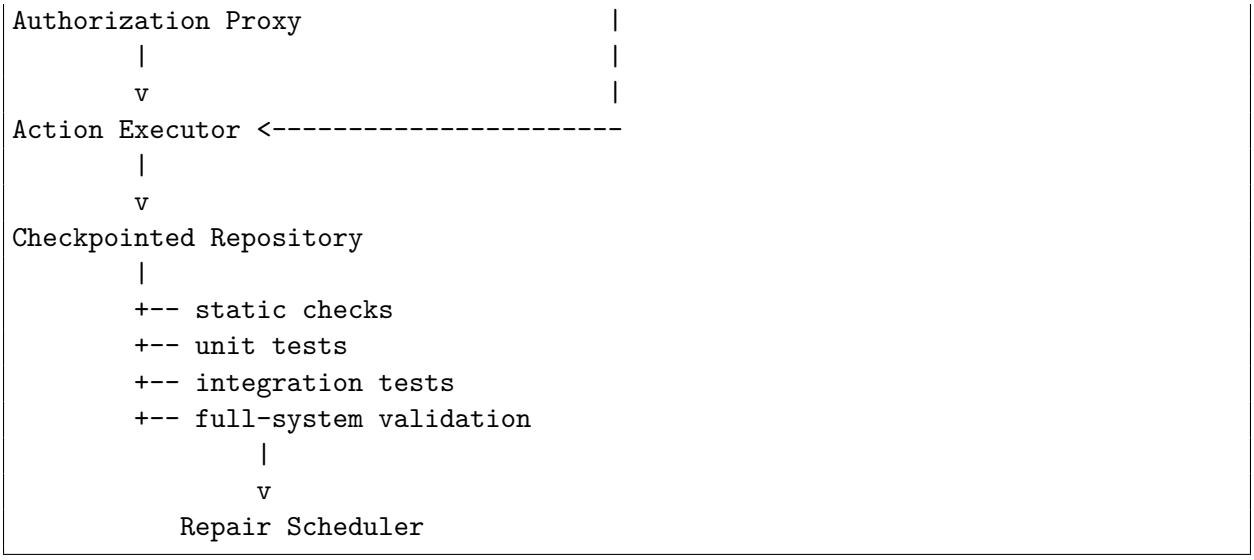
Interactive software engineering remains an excellent initial domain, but the methodology should make a sharper distinction between task correctness, trajectory correctness, and counterfactual recoverability.

A final passing test suite establishes only task correctness. Two agents can both produce the correct patch while taking radically different trajectories. One may preserve several valid architectural possibilities until sufficient evidence arrives. The other may make a sequence of destructive guesses and happen, through repeated rollback, to land on the right answer. A benchmark of monotonic continuation must distinguish them.

5.1 Experimental Testbed

Each run therefore occurs in a fully checkpointable environment:

```
Supervisor / Oracle
        |
        v
Interaction Layer
        |
        v
Epistemic Frame Store --------> Slow Deliberation Workers
        |                                |
        v                                |
```



Every side-effecting transition is performed first against a checkpointed branch or container snapshot. This makes counterfactual replay possible.

5.2 Evaluated Architectures

The four original architectures remain useful, but they should share the same base model wherever possible. Otherwise a stronger underlying model could masquerade as a stronger architecture. Thus the controlled comparison is:

$$M_{\text{base}} + A_1, \quad M_{\text{base}} + A_2, \quad M_{\text{base}} + A_3, \quad M_{\text{base}} + A_4.$$

Only orchestration changes. The benchmark therefore tests architecture, not raw model capability.

5.3 Ablations

This paper especially needs ablations. The full system should be compared against

$$A_{\text{MC}} - \{U_R\}, \quad A_{\text{MC}} - \{\tau\}, \quad A_{\text{MC}} - \{R_L\}, \quad A_{\text{MC}} - \{\text{continuation checks}\}, \quad A_{\text{MC}} - \{\text{absorption audits}\}.$$

These answer much more informative questions than simply asking whether the full system beats ReAct. For example, if removing explicit U_R representation leaves foreclosure unchanged, then one of the paper's most important implementation claims has failed. If removing authorization absorption leaves task success unchanged but doubles supervisor tokens, absorption has demonstrated efficiency rather than correctness. If removing R_L preserves ordinary benchmark scores but catastrophically worsens deeply seeded architectural errors, that directly supports the unequal-timescale hypothesis.

5.4 Task Families

The 150-task corpus should include at least four rather than three distributions. Alongside foreclosure-risk, delayed-resolution, and nested-repair tasks, add **false-resolution traps** — tasks deliberately

constructed so that a locally plausible answer passes cheap validation but fails delayed validation. For example: lint passes, unit tests pass, integration test fails; or: the requested API works, but backward compatibility breaks.

These are ideal tests because they instantiate the paper's entire thesis in miniature: a fast clock reports apparent completion while a slower clock contains information showing that completion was premature.

5.5 Counterfactual Replay

At selected trajectory states S_t, fork execution:

$$S_t \to \{\, T_{\text{commit}},\ T_{\text{defer}},\ T_{\text{steered}} \,\}.$$

This permits direct measurement of questions otherwise hidden in ordinary logs. Did deferral actually preserve a successful branch? Did the human's three-token intervention alter the trajectory? Would autonomous execution after absorption have made the same choice? Was a repair genuinely necessary, or would the original trajectory have recovered without it?

Counterfactual replay turns several philosophical claims into experimentally distinguishable alternatives.

6 Threats to Validity

This deserves its own section because otherwise the operationalization can appear more exact than it is.

Proxy collapse. The benchmark never observes the theoretical $\mathcal{R}(S_t)$ directly. It observes test suites, branch enumerations, invariants, oracle judgments, and counterfactual runs. Consequently, $\widehat{\mathcal{R}}(S_t) \neq \mathcal{R}(S_t)$ must remain an explicit methodological assumption throughout the paper.

Oracle leakage. If the simulated supervisor constructs hints by comparing directly with a ground-truth patch, it may provide information no human supervisor would naturally possess. Results should therefore be reported separately for patch-aware oracle steering and human-derived or specification-derived steering.

Benchmark overdetermination. Real software tasks often admit multiple correct patches. `ActionMatchesGroundTruthTrajectory` is therefore too strict if interpreted literally. The oracle should judge semantic constraints, tests, and admissible solution classes rather than demand resemblance to a single reference patch.

Deferral gaming. A system could artificially improve foreclosure rate by refusing to act. Resolution Latency Ratio and total task completion therefore have to be reported alongside Φ.

Repair-by-brute-force. An architecture could obtain excellent final success by repeatedly restoring checkpoints and exploring alternatives exhaustively. Compute-normalized metrics are necessary to distinguish intelligent preservation from expensive search.

Authorization distribution shift. Repeated safe behavior under one region of a category does not imply safety throughout that category. This is precisely why evidence diversity, scoped authorization, and post-absorption auditing are required.

These are not peripheral caveats. They mark the boundary between the theoretical relation and what an executable system can actually know about itself.

7 Hypotheses

The primary hypothesis is that explicit representation of unresolved-but-reachable state reduces premature commitment without producing pathological indecision:

$$H_1 : \quad \Phi_{\text{MC}} < \Phi_{\text{baseline}} \qquad \text{while} \qquad \Lambda_R^{\text{MC}} \approx 1.$$

The second concerns repair locality:

$$H_2 : \quad \mathbb{P}(R_\mu \mid e_\mu) > \mathbb{P}(R_\mu \mid e_L),$$

meaning the scheduler should preferentially solve shallow errors cheaply while escalating genuinely deep errors rather than treating every failure alike.

The third concerns steering:

$$H_3 : \quad \frac{\partial \mathcal{L}_{\text{steer}}}{\partial t} > 0$$

over coherent trajectories, up to saturation. Later interventions should require fewer bits to produce comparable directional effects because accumulated trajectory state supplies more of their interpretation.

The fourth concerns absorption:

$H_4 : \quad T_{\text{abs}}^{(c)} \downarrow$ with repeated consistent supervision, while $P_{\text{safe}}^{(c)}$ remains statistically indistinguishable from explic

The fifth concerns the architecture as a whole:

$$H_5 : \quad \begin{aligned} A_{\text{MC}} &> A_{\text{MC}} - \{U_R\}, \\ A_{\text{MC}} &> A_{\text{MC}} - \{\tau\}, \\ A_{\text{MC}} &> A_{\text{MC}} - \{R\} \end{aligned}$$

on their corresponding trajectory metrics even when final task success is similar.

That last possibility is particularly important. The theory does not predict merely that monotonic-continuation systems solve more benchmark tasks. Its stronger claim is that two systems with the same final success rate can differ systematically in the quality of the trajectories by which they arrive there.

8 What Would Falsify the Operational Account?

The operationalization would be weakened if explicit U_R representation produced no measurable reduction in premature commitment, if equivalent performance could be obtained from ordinary retry loops at equal cost, if steering leverage failed to increase with accumulated trajectory context, if authorization histories did not permit safe category-specific absorption, or if multi-horizon repair performed no better than a single undifferentiated repair process once total compute was controlled.

More strongly, if agents that aggressively collapse uncertainty into immediate decisions consistently outperform explicit unresolved-state architectures not only in latency but in final success, recovery cost, supervisor burden, and robustness under delayed validation, then the systems interpretation proposed here would have little empirical justification.

Likewise, if trajectory-aware metrics fail to predict anything beyond ordinary endpoint success, then the central experimental premise of the paper would be undermined. The claim is precisely that something consequential exists between S_0 and S_N that endpoint evaluation fails to capture.

The benchmark must therefore demonstrate that its trajectory quantities possess independent explanatory or predictive value.

9 Conclusion: Making Continuation Executable

The source theory begins from an asymmetry that ordinary computational descriptions tend to hide. Interaction, deliberation, consolidation, repair, and architectural revision do not occur on the same clock. A system that nevertheless demands that every relevant distinction resolve at the fastest clock creates an artificial pressure toward premature closure.

The implementation proposed here takes the opposite approach. It gives unresolved questions addresses. It records why they remain unresolved. It assigns them clocks. It preserves their dependencies across interaction cycles. It allows apparent knowledge to reopen when slower evidence contradicts it. It treats human steering as structured information whose repeated regularities can become executable policy, while preserving category, scope, reversibility, and audit boundaries. It treats repair not as generic retry but as a hierarchy in which increasingly slow processes are permitted to reconsider increasingly deep portions of the trajectory.

The resulting runtime does not literally compute $S_t \preceq S_{t+1}$. That relation remains the theoretical object. What the runtime can do is maintain evidence that a transition has preserved what currently appears necessary for adequate continuation:

$$S_t \preceq S_{t+1} \quad \rightsquigarrow \quad \widehat{S_t \preceq S_{t+1}}.$$

The hat matters. It marks the difference between an ontology of continuation and an engineering system trying, under finite computation and incomplete information, to behave as though that ontology were operationally important.

That limitation is not an embarrassment to the architecture. It is one of its central design constraints. A system built around explicit incompleteness should be the last system to mistake its own approximation for the thing approximated.

The experimental question is consequently not whether a machine can certify that every one of its transitions preserves all possible futures. No realistic machine can. The question is whether making unresolvedness, reversibility, steering history, repair depth, and continuation option value explicit produces trajectories that are measurably less brittle than architectures in which these quantities remain implicit.

If it does, monotonic continuation acquires an engineering interpretation. Intelligence across unequal timescales would then be visible not merely in what a system eventually answers, but in the structure of what it refuses to destroy while the answer is still becoming possible.

References

[1] Yao, S., J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. "ReAct: Synergizing Reasoning and Acting in Language Models." *International Conference on Learning Representations*, 2023.

[2] Jimenez, C. E., J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" *International Conference on Learning Representations*, 2024.

[3] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM* 21, no. 7 (1978): 558–565.

[4] Chandy, K. M., and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems* 3, no. 1 (1985): 63–75.

[5] Kwon, W., Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention." *Proceedings of the 29th Symposium on Operating Systems Principles*, 611–626, 2023.