

Pollard: Governed Semantic Execution Trees for Framework-Independent AI Agent Control and Replay

Muntaser Syed
Florida Institute of Technology
Melbourne, Florida, USA

Marius C. Silaghi
Florida Institute of Technology
Melbourne, Florida, USA

Abstract—Agent frameworks can persist checkpoints, trace tool calls, and limit resource use, but these facilities often assign different identities to execution, governance, and replay. We present Pollard, a framework-independent runtime based on governed semantic execution trees. A step identity commits to canonical input, kind, attempt, and parent identity; the same tree records completion, refusal, resource settlement, and branch history. Strict replay returns a verified stored result without invoking the wrapped operation, while a transactional store coordinates reservations across concurrent workers. Under a signed deterministic freeze, two state-model replicas each executed 6400000 transitions without an invariant violation, 0 of 122 retained-lease PostgreSQL groups violated an exact limit, strict replay made 0 external calls in 155 observations, and all 1470 semantic mutations were detected with 0 alarms on 120 controls. A 360-pair framework matrix preserved reference behavior and verified 720 trees. A separate cost-capped hosted study exercised live provider paths, and every completed Pollard trial produced a verified tree. Provider compatibility and the frozen budget boundary limited the hosted utility comparison, so it is reported as integration evidence rather than benchmark-wide equivalence. Together, the studies support the core mechanisms under explicit lease, boundary, and seal-custody assumptions.

I. INTRODUCTION

Tool-using language model agents perform work through a sequence of model calls, retrievals, tool invocations, and external side effects. A final answer does not reveal which resources were consumed, why a tool call was permitted, or whether a later reconstruction repeated the same execution. Recent work documents budget overruns in deployed agent systems [1], proposes formal resource contracts [2], and identifies execution provenance as a basis for process-level accountability [3]. These lines of work expose a common systems problem: control decisions and execution evidence need a shared identity.

Existing systems address parts of this problem. Runtime policy languages mediate actions before execution [4]. Agent workbenches record tool responses and inject repeatable faults [5]. Graph runtimes persist checkpoints and support forks, although resumed nodes may invoke models and tools again [6]. Tool caches reduce repeated calls by keying requests from tool names and arguments [7]. None of these mechanisms, by itself, establishes that a stored result, a resource reservation, and an authorization decision refer to one ancestry-bound semantic step.

Pollard introduces a runtime primitive called a governed semantic execution tree. A caller retains its own control loop and wraps selected model or tool operations as semantic steps. Before a step runs, Pollard derives an identity from the step input and its parent, evaluates resource and action gates, and records the decision. After execution, it settles measured resource use and stores the result. In strict replay mode, a matching step reads the recorded result and does not call the wrapped operation.

This paper makes four contributions:

- A formal execution model that binds semantic identity to causal ancestry while leaving orchestration with the caller.
- Admission and settlement rules for multi-dimensional resources, including exact safety conditions and an explicit bound for estimation error.
- A runtime implementation with in-memory, SQLite, and PostgreSQL stores, action registration, refusal nodes, branching, replay, sealing, and telemetry adapters.
- A signed evaluation of adversarial concurrency, replay nonexecution, causal separation from argument-only caching, integrity, storage, framework boundaries, and cost-capped hosted integration.

The central question is not whether content addressing, checkpoints, or resource counters are individually new. The question is whether one minimal execution identity can make governance and replay mutually checkable without requiring a new agent framework.

II. PROBLEM SETTING

A. Execution provenance

Provenance systems represent how entities and activities contribute to a result. The W3C PROV data model supplies a domain independent vocabulary for entities, activities, agents, and derivations [8]. Scientific workflow systems demonstrate that automatic provenance capture and durable workflow execution can support reproducibility at scale [9]. Agent execution adds two complications. First, model and tool calls may be nondeterministic. Second, a runtime may need to refuse an action before the action produces an output.

Pollard records operational provenance. It does not infer claim level evidence, semantic justification, or factual support.

This boundary matters because agent provenance research distinguishes an execution graph from evidence relations that support a claim [3].

B. Admission before execution

A post hoc trace can explain a call after it has happened, but it cannot prevent a known violation. Pollard therefore performs admission before invoking a wrapped operation. Admission considers estimated resource use, action registration, policy, schema validation, and confirmation. A denied attempt becomes a refusal node. The record therefore contains alternatives that were considered but not executed.

An estimate can differ from measured use. The system does not claim that every unknown future cost can be capped exactly. Instead, it separates exact meters from approximate meters and reports the assumptions needed for each guarantee.

C. Replay semantics

Replay has several meanings. A checkpoint runtime can resume from saved state and reexecute later nodes. A transport recorder can return a prior HTTP response. A tool cache can reuse a result for matching arguments. Pollard defines strict replay at the semantic step boundary: if the derived node identity and stored result match, the runtime returns that result and does not invoke the wrapped operation. This definition is testable with a sentinel that fails if any external operation is called.

D. Trust and threat boundary

The design addresses four classes of failure. First, concurrent workers may race during admission or repeat reserve and settle requests after losing an acknowledgment. Second, a process or database connection may fail between admission, external execution, settlement, and node persistence. Third, replay may find a node under the wrong ancestry or a stored result whose digest no longer matches. Fourth, an operator or storage fault may alter node fields, results, membership, or ordering. The evaluation injects each class at a declared boundary.

Pollard trusts its canonicalizer, hash implementation, runtime code, and the transactional semantics of the selected store. Exact shared-budget safety additionally trusts a completing owner to retain its reservation through settlement. Tamper evidence trusts an externally held seal digest when store deletion or replacement is in scope. These assumptions are explicit because no library can preserve a global cap during an unavailable database while also admitting new work, and a seal kept only beside the protected data cannot expose wholesale replacement.

The caller chooses which operations to wrap. Pollard cannot govern a model or tool invocation that bypasses that boundary, cannot validate the truth of a model response, and cannot reconstruct external world state from an old tool result. Consequently, integration coverage is measured against an out-of-band boundary proxy, and replay claims concern nonexecution and stored-result integrity rather than renewed correctness of an external service.

III. GOVERNED SEMANTIC EXECUTION TREES

A. Tree and identity

Let an execution tree be $T = (V, E)$ with a distinguished root. Every non-root node has exactly one parent already in V , so an edge records causal placement rather than merely data similarity. A node is

$$n = (p, k, a, x, y, m), \quad (1)$$

where p is the parent identifier, k is the step kind, a is the attempt number, x is the identity input, y is an optional stored result, and m is metadata. Completion and refusal are represented by stored node kinds and results; an operation that raises before producing a result is not converted into a completed node. Let C_I be deterministic identity serialization and d_I an identity domain separator. The semantic identifier is

$$I(n) = H(d_I \parallel C_I(p, k, a, x)). \quad (2)$$

The canonical identity domain contains null, booleans, integers, strings, lists, and string-keyed objects. Floating point values and raw bytes are rejected because platform-dependent encodings would weaken identity stability. Metadata and the stored result do not determine $I(n)$. A result is serialized once by a separate stable JSON serializer C_R and committed as $D(n) = H(d_R \parallel C_R(y))$.

The current cursor identifies the parent for the next step. Branching copies a cursor without copying nodes; rollback moves a cursor to an ancestor; pruning marks metadata for later garbage collection. These operations change reachability or future ancestry, not an existing node identity. Identical semantic inputs under the same parent converge on one identifier. Identical inputs under different parents remain distinct unless the hash function collides. This choice gives up some argument-only cache reuse to prevent a match from erasing causal history.

B. Admission and settlement transition

For scope s , let B_s be settled use, L_s its componentwise limit, and R_s the active reservations. Reservation j carries estimate vector e_j , owner identity, and expiry. A new step i is admitted only if the store atomically establishes

$$B_s + \sum_{j \in R_s} e_j + e_i \leq L_s \quad (3)$$

for every applicable scope. The same transaction creates i in R_s . The wrapped operation then either raises, or returns y_i with measured charge c_i . A successful settlement atomically removes e_i and adds c_i to B_s . A call failure releases the reservation and propagates the exception. A gate failure appends a refusal node containing the blocked kind and a digest of the blocked payload, then returns control by raising without invoking the operation.

Reservations are leases because a worker can disappear after admission. A lease may expire and make its estimate available to another worker. Therefore exact safety applies to calls whose owners retain the lease until settlement. If

an expired owner later completes, its result is evidence of work performed outside that premise and is reported separately rather than folded into the zero-violation contrast.

C. Replay and sealing

Record mode executes and stores a missing step. Hybrid mode returns a stored result on a matching identity hit and otherwise records. Strict replay requires the node and result to exist and verify; a miss or divergence fails without calling the supplied operation. Verification recomputes $I(n)$, checks parent existence and traversal order, and recomputes $D(n)$ from the stored result text.

For deterministic preorder $(n_0, \dots, n_t)$, Pollard forms a rolling seal. Let z_{-1} be empty. Each entry is

$$z_i = H(d_S \parallel C_I(i, I(n_i), p_i, k_i, D(n_i), z_{i-1})). \quad (4)$$

The report publishes z_t and the ordered entries. The index and previous digest expose reordering or truncation; node and result digests expose semantic changes. Detection of total store replacement still requires z_t to be held outside the store.

D. Guarantees

Lemma 1 (Ancestry separation). If two nodes differ in p , k , a , or canonical x , then their identifiers differ unless H collides. A changed ancestor therefore changes each descendant identifier along that path.

Proof sketch. The differing field changes the domain-separated input to H . Each child commits to its parent’s identifier, so induction extends the difference to every descendant on the path.

Theorem 1 (Exact retained-lease safety). Suppose reserve and settle are serializable, each completing owner retains its reservation, and $c_i = e_i$ for every admitted step. Then $B_s + \sum_{j \in R_s} e_j \leq L_s$ is invariant for every scope s , and settled use cannot exceed L_s .

Proof sketch. The empty state satisfies the invariant. Reserve preserves it by the admission predicate. Exact settlement replaces e_i with the equal c_i and release only removes an estimate. Serializable transitions therefore preserve the invariant by induction. Because all active estimates are nonnegative, $B_s \leq L_s$.

Corollary 1 (Estimated-meter bound). Under the same transactional and lease assumptions, let $\epsilon_i = \max(0, c_i - e_i)$. Observed overshoot is at most $\sum_i \epsilon_i$ over the relevant settlement interval. The bound does not turn an estimated meter into an exact cap; it identifies the quantity that calibration must control.

Proposition 1 (Replay nonexecution). If strict replay returns, the returned result belongs to the requested identity, its digest verifies, and the caller-supplied operation was not invoked. A sentinel operation that always fails makes the last condition directly testable.

Proposition 2 (Rolling-seal sensitivity). Under collision resistance and an externally retained final digest, any change to a sealed node identity, parent, kind, stored result, membership,

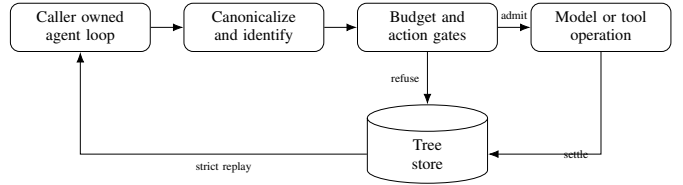


Fig. 1. Pollard mediates semantic steps without owning the agent loop. The replay path returns a stored result without invoking the wrapped operation.

or deterministic order changes the final seal. Metadata absent from the seal remains outside this claim.

Table I separates each guarantee from its required premise and its observable boundary case. This separation is central to the evaluation: a violated premise is retained as a result, not relabeled as evidence for the narrower theorem.

IV. SYSTEM DESIGN

Figure 1 shows the execution path. The caller owns planning and control flow. Pollard receives a semantic step request, derives its node identity, and consults resource and action gates. A denied step is recorded immediately. An admitted step either reads an existing result in replay mode or invokes the caller supplied operation. The store then settles resource use and persists the outcome.

A. Boundary and execution lifecycle

Pollard is a Python library rather than an agent scheduler. The caller opens a run, selects the parent cursor, and wraps a model call, tool call, or note. This placement is deliberate: only the application knows whether a framework node, provider request, retry, or tool callback is the semantic unit that should share identity with policy and replay. Synchronous and asynchronous runtimes implement the same lifecycle.

Before admission, the runtime validates the identity payload and computes the prospective node identifier. Record mode proceeds to gates and execution. Hybrid mode uses an identity match that has a stored result and records on a miss. Strict replay requires an existing result and verifies the stored path before returning it; any mismatch fails closed. Avoided charges remain mutable reporting metadata outside semantic identity.

On a record path, precheck estimates are collected from every meter and applied to each nested budget scope. Structural gates then validate a registered tool’s name, version, schema, policy, and confirmation. A refusal is persisted before the corresponding exception is raised. Only after all gates pass does the runtime reserve shared capacity and invoke the caller function. Successful execution stops measurement, settles the reservation, constructs the result-bearing node, and advances the cursor. A function exception releases the reservation and propagates without fabricating a successful node.

This ordering does not claim exactly-once external side effects. If a provider or tool completes but the database becomes unavailable before settlement and node persistence finish, Pollard cannot safely infer whether repeating the external action is acceptable. The database operation is retryable; the external action remains an application-level ambiguity.

TABLE I
GUARANTEES, REQUIRED PREMISES, AND EXPLICITLY EXCLUDED OUTCOMES.

Guarantee	Required premises	Boundary outside the guarantee
Ancestry separation	Stable canonicalization, domain separation, and collision-resistant hashing	Caller omits a causal input or invokes work outside the selected boundary
Exact resource safety	Serializable reserve-settle, exact estimates, and lease retained through completion	Estimate error, expired owner completion, or unavailable shared store
Strict replay nonexecution	Exact identity hit, stored result present, ancestry and digest verification	Reexecution mode or unwrapped external side effects
Refusal soundness	Every governed action passes the gate before invocation	Caller bypasses the runtime or policy cannot classify the action
Rolling tamper evidence	Deterministic traversal, protected hash implementation, externally retained final digest	Mutable metadata excluded from the seal or loss of external custody

B. Stores and concurrency

The store interface separates node persistence from coordination. The memory store is intended for local execution. HashRope supplies append-oriented persistence with content interning. SQLite provides durable single-host storage and local arbitration. PostgreSQL supplies transactional reservations for concurrent processes and hosts. Store insertion is idempotent for an identical identity tuple and rejects an identifier conflict.

PostgreSQL reserve and settle requests carry a unique reservation identifier plus canonical request or charge digests. After connection loss, the client reconnects once and repeats the transaction. A committed reserve is recovered by the same identifier; a duplicate settle must carry the same charge digest and does not debit twice. If the retry cannot distinguish a committed operation from a failed one, the API raises an explicit uncertainty exception instead of guessing. This protects database accounting, but does not erase ambiguity about an external operation that ran between database failures.

A running call renews its lease at a fixed cadence derived from one third of the duration and bounded between 10 ms and 30 s. The runtime tracks the last confirmed deadline. If renewal is not confirmed before then, a completed result is settled and stored when possible, annotated with lease loss, and reported through an exception carrying its reservation and node identifiers. Its charge is excluded from retained-lease claims. Operators must set the lease above plausible database interruption, renewal latency, and scheduling stalls.

Schema state is also fail closed. PostgreSQL schema version 2 stores an explicit singleton version and validates the expected layout. Unknown, absent, or forward versions are refused. Migration requires drained workers and reservations, a backup, the recognized migration, and explicit reopen. Live restore includes node and arbiter tables; audit-only recovery need not reopen admission. Thus gate rejection records a refusal without execution; lost reserve or settle acknowledgment is recovered idempotently or reported as uncertain; an operation exception releases its reservation without a completed node; lease loss is durable when persistence succeeds; and an unknown schema never opens implicitly.

C. Deployment controls

Production hardening separates compatibility coverage from systems evidence. Package integration tests exercise PostgreSQL 14 through 18 and Python 3.10 through 3.13, while the confirmatory concurrency study uses one pinned PostgreSQL 17.6 instance with synchronous commit enabled. The wider matrix checks schema creation, version refusal, recognized migration, reserve and settle idempotence, and ordinary reconnect behavior. E2 supplies the narrower but adversarial evidence for process races, delayed operations, lost acknowledgments, database restart, and lease loss.

The operator procedure is deliberately conservative. Before migration, workers are drained and active reservations must be reconciled. A database backup is taken before the recognized forward migration; an unknown or newer version is never modified automatically. Restoring service requires both node and arbiter tables, followed by explicit schema validation. Seal custody is backed up independently from the governed store. These steps do not create high availability, but they prevent an application restart from silently converting an unknown database or uncertain settlement into accepted work.

D. Action admission

Actions are registered with a name, version, JSON schema, side-effect declaration, and specification digest. The registry digest commits to the sorted set of action specifications and can be attached to a run. Admission rejects unknown actions, version mismatches, invalid arguments, policy violations, or missing confirmation. Schema annotations such as descriptions, titles, and defaults remain in the specification digest even when they do not change validation. This permits discovery through MCP and other tool registries without making a moving external registry authoritative during replay.

These checks are structural. A policy callback can inspect declared tool metadata and arguments, but Pollard does not classify arbitrary content as safe or unsafe and does not replace a semantic guardrail language such as AgentSpec [4]. The specification digest instead makes the exact action contract part of the recorded operating context.

E. Integrity and telemetry

Verification checks node identity, result digests, parent existence, and deterministic child order before a subtree is replayed, sealed, exported, or imported. Export writes a self-contained ordered manifest and its rolling seal; import verifies the full artifact before inserting any node. These checks expose changes within the sealed material under the stated hash assumption. They do not prevent a privileged operator from replacing both the store and a locally held seal, and metadata excluded from the seal is not protected.

The reference custody sink therefore writes the seal digest, store and root identifiers, sequence, time, algorithm, and signer identity to a separate SQLite database. It refuses to share the Pollard store file. The sink is a reference boundary, not a key-management system: signing, independent storage, access control, retention, and custody review remain operator responsibilities.

Telemetry exports operation kind, timing, outcome, and selected digests without making trace or span identifiers part of semantic identity. Payload and result content need not enter telemetry. This preserves compatibility with observability conventions such as OpenTelemetry [10] while keeping operational correlation distinct from the integrity boundary.

F. Framework integration

Adapters remain intentionally thin. LangGraph functions, OpenAI Agents custom models and function tools, pydantic-ai function models, and MCP clients call the same Pollard model and tool primitives; Pollard does not translate their control graphs into its own scheduler. Framework events are recorded separately from model and tool boundaries. This distinction prevents a three-node graph from being misreported as three provider calls and allows an independent proxy to expose work that bypasses the chosen boundary.

V. EVALUATION

The evaluation asks whether one semantic identity can support admission, replay, and audit without taking ownership of the caller’s agent loop. Deterministic studies ran from the signed tag `ictai-confirmatory-v1`. Hosted attempts used separate signed tags, an independent spend guard, immutable terminal records, and no model substitution. All prespecified terminal outcomes are retained in the analysis.

A. Design and analysis

Each study fixes its unit, factors, oracle, exclusions, and terminal states. E1 uses an independent identity and budget state model. E2 runs fresh PostgreSQL process groups with barrier schedules and connection, acknowledgment, restart, retry, and lease faults. E3 pairs recording and replay with an out-of-band call sentinel, exact state digest, ancestry divergences, result mutations, and an argument-only cache. E4 measures each cell in a fresh process. E6 mutates exported artifacts, native objects, log bytes, and relational rows while an independent seal or snapshot remains outside the target store. E7 compares

manual wrapping with LangGraph, OpenAI Agents, pydantic-ai, and MCP paths while an external proxy counts model and tool boundaries.

E5 uses six pinned hosted model identifiers and 15 result-blind selected τ^3 and AgentDojo tasks [11]–[13]. Its 81 adjacent pairs compare unwrapped execution with Pollard recording under inactive gates. Provider retries and hosted tools are disabled. A shared 50,000-input and 4,000-output token ledger applies to each trial, and the independent guard reserves worst-case cost before generation. The authorized total is 8.00 USD, with a 7.00 USD hard stop and 1.00 USD unavailable to the study.

Binary zero-event results use one-sided exact 95 percent bounds at the independent-unit level. E5 utility differences use task-cluster resampling with 10,000 fixed-seed replicates. Its prespecified analysis assigns zero utility to terminal errors rather than excluding them. Latency and cost are paired only within complete pairs. Timing maxima in E4 establish an observed resource envelope, not an asymptotic claim. Table II summarizes scope and completion.

B. Identity, admission, and replay

Both E1 replicas reproduced the same claim digest. Each executed 6400000 seeded transitions after bounded exploration, with 0 full-model violations. Every one of the 10 required ablations exposed its preregistered witness. The result supports identity convergence, ancestry separation, refusal evidence, exact-budget invariants, strict replay, and seal sensitivity within the modeled bounds.

E2 completed 542 one-host process groups and 2768 admission attempts. None of the 122 exact Pollard groups whose completing owner retained its lease violated conservation; the one-sided group-level upper bound is 0.0243. Each weaker exact comparator produced barrier-synchronized overshoot. Estimate ratios behaved as predicted rather than as exact caps. In 2 suspended-owner cases, expired capacity was reused and the original owner later completed, producing visible overshoot outside the retained-lease theorem. Lost reserve acknowledgment, duplicate settlement, reconnect, restart, and ambiguous settlement retries reached terminal audited outcomes.

Across 895 E3 cells, strict replay entered the wrapped operation 0 times in 155 observations. The divergence oracle localized all 150 changed paths, all 60 stored-result mutations were rejected, and hybrid replay produced no unsafe reuse. In contrast, the argument-only cache created 930 unsafe aliases. This is the direct cost of omitting ancestry from a reuse key.

Table III collects the mechanism controls that distinguish a working guarantee from a passing workload. Each control changed only the named premise or instrumentation boundary. The three weaker exact counters each accumulated 280 units of overshoot. Boundary ablations preserved application behavior but made their omissions visible to the independent proxy. These outcomes show that the independent oracles discriminate the claimed mechanism from weaker alternatives.

TABLE II
FROZEN STUDIES, INDEPENDENT UNITS, AND TERMINAL COVERAGE.

Study	Unit	Principal oracle and contrasts	Terminal evidence
E1	State or seeded trace	Independent identity, budget, replay, and seal model; ten ablations	2 exact replicas, 6400000 transitions each
E2	Process group or injected fault	Conservation; transactional reservation versus three weaker counters; lease boundary	542 groups, 2768 attempts, 20 fault cases
E3	Record-replay pair	External-call sentinel, state digest, divergence oracle, cache aliasing	895 cells, 310 external seal publications
E4	Fresh process	Store, depth, branch, payload reuse, interning, lifecycle operation	132 cells, 660 measured processes
E5	Reset task pair	Native scorer, token ledger, spend guard, inactive recording	E5-A: 162 terminal trials; E5-B: 20 terminal trials
E6	Trace and mutation	Independent verifier or externally held native snapshot	1,680 terminal cases, 120 clean controls
E7	Manual-framework pair	Out-of-band event proxy, exact state and text parity	360 pairs, 720 verified trees

TABLE III
PREREGISTERED MECHANISM CONTROLS AND OBSERVED WITNESSES.

Control	Observed witness
Post-hoc, local-lock, or settled-only counter	280 aggregate overshoot units each
Expired reservation owner	2 late completions outside exact theorem
Argument-only result cache	930 ancestry-unsafe aliases
Model-only boundary	120 declared tool events absent
Tool-only boundary	240 declared model events absent
Permitted metadata edits	0 alarms in 120 controls

C. Resource, integrity, and portability results

All 132 E4 cells completed their five measured starts after warmup, totaling 660 fresh processes. The largest observed process took 408.47 seconds and reached 968.95 MiB. In the four targeted high-reuse interning contrasts, enabled-to-disabled bytes per node had median ratio 0.349, but the ratios ranged from 0.252 on SQLite to 0.959 on PostgreSQL. We therefore report the individual contrasts and do not claim one store-independent saving or unwrapped runtime overhead.

E6 detected 1470 of 1470 semantic mutations and raised 0 semantic alarms on 120 permitted metadata controls. It separately classified 30 representation-only changes, 90 constraint-prevented mutations, and 90 representation-inapplicable cases. Whole-store deletion and replacement were judged against externally retained digests; local verification alone is not credited for that threat.

Every E7 pair matched the manual reference in final state and output. All 720 Pollard trees verified. Full-boundary cells captured 360 of 360 declared events. Model-only and tool-only ablations omitted exactly the boundaries they did not wrap. Thus portability is supported for the tested releases and public interfaces, not for arbitrary background calls or framework extensions.

D. Hosted validation and integration findings

E5-A exercised all 162 frozen trials, yielding 34 completions and 128 visible compatibility errors. Strict-schema tool limits, SDK token-count serialization, unresolved schema references, and weak exception capture identified concrete portability requirements for the hosted adapter. Retaining every terminal pair yielded 11 unwrapped and 10 inactive-Pollard utility successes, one unwrapped-only discordance, and mean paired difference -0.0123 with task-cluster interval [-0.0385, 0.0000]. All 16 completed Pollard trials produced verified

trees. Because harness-provider compatibility determined most terminal outcomes, the interval is reported for completeness and not interpreted as utility equivalence.

A separately signed corrective attempt incorporated the identified adapter fixes and passed exact token-count requests for all six largest frozen tool payloads without generation. E5-B then produced 20 terminal trials, or 10 complete pairs: 16 completed and 4 returned explicit terminal errors. Each condition had five utility successes, with no discordance among these pairs, and all eight completed Pollard trials verified. Two Gemini 2.5 Pro trials reported model unavailability after successful token-count preflight, while two Gemini 2.5 Flash-Lite trials returned parallel tool calls outside the frozen single-call policy. These observations distinguish request-shape validation from live generation compatibility.

Guard reconciliation found that an E5-A child process had continued after its foreground terminal was interrupted, which made the initial E5-B freeze understate prior settlement. E5-B was stopped, the active reservation was charged at the full ambiguous ceiling, and process liveness plus guard state were reconciled. Conservative settlement across both attempts is \$4.35704260, below the 7.00 USD hard stop, with no active reservation and the 1.00 USD reserve untouched. The remaining \$2.64295740 cannot fund the full frozen corrective schedule, and reducing the matrix after observing results is prohibited. This closeout preserves the preregistered budget and matrix rules. E5 therefore contributes end-to-end integration and portability evidence; it does not estimate benchmark-wide effectiveness.

E. Interpretation

The evidence supports three mechanism claims and supplies bounded external-validation evidence. First, ancestry is operational rather than decorative: removing it from the cache key produced unsafe reuse, while full strict replay returned verified bytes without a call. Second, shared admission is safe under the theorem’s exact-meter and retained-lease premises. E2 preserved that invariant and distinguished the cases in which a premise no longer holds. Third, seal verification detects the declared semantic changes when the final digest has custody outside the attacked store. These conclusions rely on independent sentinels, models, snapshots, or proxies rather than on Pollard accepting its own records.

The hosted study addresses a different question: whether frozen provider stacks can carry the same boundary into live

task execution. Every completed Pollard trial produced a valid tree, and E5-B observed no utility discordance across its ten complete pairs. Compatibility outcomes dominate E5-A, and E5-B is a result-dependent prefix rather than the frozen matrix. The result is therefore a verified end-to-end integration path plus a concrete portability boundary, not a utility-equivalence estimate and not a negative result about Pollard’s deterministic mechanisms.

The hosted observations also yield operational requirements. Metadata access and token counting are necessary but insufficient provider gates. A model can accept token counting and reject generation, and schema dialects can differ on references, strict-tool counts, object closure, and parallel calls. A production deployment should validate its exact request shape, treat provider model identifiers as time-bound dependencies, preserve raw error responses, and supervise the actual child process rather than only its terminal. These controls establish provider portability before Pollard receives a completed boundary to govern.

F. Reproducibility boundary

Run manifests record package and paper commits, environment and database versions, model identifiers, request settings, task and schema digests, limits, terminal reasons, and settlements. Deterministic aggregates link to private raw inputs through a digest ledger. Hosted closeout records manifest-set digests and the external guard reconciliation without publishing licensed prompts or provider transcripts. Reproduction can audit every reported aggregate, but independent hosted reruns would be new stochastic experiments rather than exact replays.

The artifact keeps executable analysis separate from raw custody. Versioned summaries and TeX macros are regenerated from private manifest directories whose file-set digests are published. Protocol validators reject missing terminal states, stale source digests, unsupported model substitutions, and unsettled guard records. The deterministic suite reports an aggregate digest over its claim projections, while timing remains outside exact digest equality. This split permits exact audit of correctness claims without pretending that wall-clock measurements or hosted sampling are deterministic.

VI. RELATED WORK

Resource governance. Agent Contracts formalizes multi-dimensional constraints, temporal bounds, success criteria, and conservation under delegation [2]. Token Budgets catalogs production incidents and studies an affine-typed Rust mitigation that makes several delegation errors compile-time failures [1]. LiteLLM offers gateway-level budgets and rate limits for users and teams [14]. Pollard does not claim that preflight accounting is new. Its focus is the coupling of admission and settlement to the same ancestry-bound record used for replay.

Runtime policy enforcement. AgentSpec defines a language for runtime constraints and pre-action mediation [4]. A trace based assurance framework combines contracts, fault injection, deterministic replay, and governance over message action traces [15]. Pollard contributes a concrete semantic

identity and transactional store protocol. Its structural action registry is narrower than a general safety policy language.

Replay, checkpoints, and caching. LangGraph persists graph checkpoints and can fork from earlier state, while downstream nodes reexecute during time travel [6]. AgentCheck records MCP tool responses and uses replay to reproduce injected failures [5]. ToolCaching studies admission and eviction for repeated tool requests [7]. Pollard gives up some cross-context cache reuse in exchange for ancestry separation and strict nonexecution at a semantic boundary selected by the caller.

Provenance and audit. W3C PROV provides a general model of entities, activities, agents, and derivation [8]. AiIDA records workflow provenance and supports durable, high throughput scientific computation [9]. Recent agent research surveys typed execution graphs, evidence relations, and recovery requirements [3]. Lifecycle audit trails connect technical and governance records through tamper evident ledgers [16]. Pollard addresses step level operational provenance. It does not establish claim level evidence or organizational accountability by itself.

Positioning. These systems often identify different objects: a policy action, checkpoint, transport response, tool request, provenance activity, or accounting principal. Pollard does not replace them. It supplies a semantic step identifier that can be attached to their records when the application intends the same causal operation. The evaluation therefore avoids feature-count comparisons. Instead, it tests the consequences of the identity choice directly: ancestry changes must separate results, strict replay must avoid a call, reservation state must survive retry, and an omitted integration boundary must be observable. This framing also limits the novelty claim. Content addressing, leases, policy callbacks, checkpoints, and hash chains are established techniques; the contribution is their executable coupling at a caller-selected agent boundary and the failure model required to make that coupling auditable.

VII. LIMITATIONS AND THREATS TO VALIDITY

Semantic scope. Pollard replays recorded results, not a hosted model’s sampling process. Strict replay establishes nonexecution and stored-byte integrity; it cannot prove that the original service was correct or recreate changed external state. The caller also chooses the boundary. Work that bypasses an adapter is neither governed nor recorded.

Resource assumptions. Exact safety requires serializable reservation updates, exact estimates, and a lease held through settlement. Approximate meters inherit estimation error. An expired owner can finish after replacement work, as E2 demonstrated. During a store partition, Pollard refuses admission or reports uncertainty; it does not provide consensus, provider-side cancellation, or recovery of unreported usage. E2 used one physical host, so its results do not cover multi-host clock behavior, network partitions, or production contention.

Integrity and measurement. Content addressing evidences change under a hash assumption. A privileged adversary can replace a store and a colocated seal unless the final

digest has independent custody. The reference sink separates files but does not provide key management, transparency, or independent administration. E4 reports one machine’s absolute envelope and selected storage contrasts. Mutation and clean-control counts cover the declared corpus, not all faults or adversaries.

External validity. E7 tests selected framework releases and public interfaces with a small deterministic graph. E5 uses 15 tasks, six model identifiers, and a strict single-call policy. It exercised the full E5-A schedule and ten corrective pairs across three provider families, but compatibility constraints limited the usable paired sample. Model availability also changed between metadata and generation requests. These observations bound the hosted utility inference; they do not alter the deterministic mechanism results or establish benchmark-wide equivalence, model rankings, or small utility effects.

Researcher control. Signed tags, result-blind task selection, immutable terminal records, digest ledgers, fixed terminal treatment, and explicit closeout reduce selective reporting. They do not remove workload selection bias or substitute for an independent reproduction. E5 also shows that a signed manifest must be paired with process-liveness and external-guard reconciliation before a later freeze.

VIII. CONCLUSION

Pollard tests a systems proposition: admission, execution, refusal, replay, and audit become mutually checkable when they share one ancestry-bound semantic identity. The caller retains the agent loop and selects model and tool boundaries; Pollard supplies the identity, transactional reserve-settle life-cycle, strict stored-result replay, structural action admission, and rolling evidence.

The signed deterministic suite supports those mechanisms under their stated premises. State exploration preserved all invariants while every ablation exposed a witness. Retained-lease PostgreSQL groups stayed within exact limits, strict replay made no external call, all declared semantic mutations were detected without a control alarm, and 360 framework pairs preserved manual behavior while verifying 720 trees. Mechanism controls confirmed that these outcomes depend on transactional reservation, retained leases, ancestry-aware reuse, complete boundary placement, and external seal custody.

Hosted validation extended the evaluation into live provider paths. Every completed Pollard trial yielded a verified tree, and the corrective sample showed no observed utility discordance. Provider compatibility and fixed-budget rules left the planned benchmark comparison incomplete, so we make no utility-equivalence claim. The main result remains narrow and testable: within a correctly instrumented boundary, retained lease, and externally anchored seal, one semantic tree can connect governance decisions to replayable execution evidence.

REFERENCES

- [1] S. Khan, “Token budgets: An empirical catalog of 63 LLM-agent budget-overrun incidents, with an affine-typed rust mitigation as a case study,” *arXiv preprint arXiv:2606.04056*, 2026. [Online]. Available: <https://arxiv.org/abs/2606.04056>
- [2] Q. Ye and J. Tan, “Agent contracts: A formal framework for resource-bounded autonomous AI systems,” *arXiv preprint arXiv:2601.08815*, 2026. [Online]. Available: <https://arxiv.org/abs/2601.08815>
- [3] Y. Wang, J. Zhang, T. Cai, Z. Liu, Q. Sun, Z. Sun, Z. Wu, M. Dong, M. Zheng, X. Yin, and Y. Zhu, “From agent traces to trust: A survey of evidence tracing and execution provenance in LLM agents,” *arXiv preprint arXiv:2606.04990*, 2026. [Online]. Available: <https://arxiv.org/abs/2606.04990>
- [4] H. Wang, C. M. Poskitt, and J. Sun, “AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents,” *arXiv preprint arXiv:2503.18666*, 2025, accepted by the 48th IEEE/ACM International Conference on Software Engineering, 2026. [Online]. Available: <https://arxiv.org/abs/2503.18666>
- [5] A. Mazumder and N. jahan Lia, “AgentCheck: A reproduce-intervene-mitigate workbench for LLM agents over MCP,” *arXiv preprint arXiv:2607.11098*, 2026. [Online]. Available: <https://arxiv.org/abs/2607.11098>
- [6] LangChain, “Use time-travel,” LangGraph documentation, 2026, accessed 2026-07-21. [Online]. Available: <https://docs.langchain.com/oss/python/langgraph/use-time-travel>
- [7] Y. Zhai, D. Shen, J. Luo, and B. Yang, “ToolCaching: Towards efficient caching for LLM tool-calling,” *arXiv preprint arXiv:2601.15335*, 2026. [Online]. Available: <https://arxiv.org/abs/2601.15335>
- [8] World Wide Web Consortium, “PROV-DM: The PROV data model,” World Wide Web Consortium, W3C Recommendation, Apr. 2013. [Online]. Available: <https://www.w3.org/TR/prov-dm/>
- [9] S. P. Huber, S. Zoupanos, M. Uhrin, L. Talirz, L. Kahle, R. Häuselmann, D. Gresch, T. Müller, A. V. Yakutovich, C. W. Andersen, F. F. Ramirez, C. S. Adorf, F. Gargiulo, S. Kumbhar, E. Passaro, C. Johnston, A. Merkys, A. Cepellotti, N. Mounet, N. Marzari, B. Kozinsky, and G. Pizzi, “AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance,” *Scientific Data*, vol. 7, p. 300, 2020. [Online]. Available: <https://www.nature.com/articles/s41597-020-00638-4>
- [10] OpenTelemetry Authors, “OpenTelemetry semantic conventions 1.43.0,” OpenTelemetry specification, 2026, accessed 2026-07-21. [Online]. Available: <https://opentelemetry.io/docs/specs/semconv/>
- [11] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan, “ τ -bench: A benchmark for tool-agent-user interaction in real-world domains,” *arXiv preprint arXiv:2406.12045*, 2024. [Online]. Available: <https://arxiv.org/abs/2406.12045>
- [12] V. Barres and B. Shi, “ τ^3 -Bench: Fixing airline + retail,” Sierra Research, Feb. 2026, accessed 2026-07-21. [Online]. Available: <https://taubench.com/blog/tau3-task-fixes.html>
- [13] E. DeBenedetti, J. Zhang, M. Balunović, L. Beurer-Kellner, M. Fischer, and F. Tramèr, “AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents,” *arXiv preprint arXiv:2406.13352*, 2024. [Online]. Available: <https://arxiv.org/abs/2406.13352>
- [14] LiteLLM, “Budgets, rate limits,” LiteLLM documentation, 2026, accessed 2026-07-21. [Online]. Available: <https://docs.litellm.ai/docs/proxy/users>
- [15] C. Paduraru, P.-L. Bouruc, and A. Stefanescu, “A trace-based assurance framework for agentic AI orchestration: Contracts, testing, and governance,” *arXiv preprint arXiv:2603.18096*, 2026. [Online]. Available: <https://arxiv.org/abs/2603.18096>
- [16] V. Ojewale, H. Suresh, and S. Venkatasubramanian, “Audit trails for accountability in large language models,” *arXiv preprint arXiv:2601.20727*, 2026. [Online]. Available: <https://arxiv.org/abs/2601.20727>